Стивен Халим, Феликс Халим

Спортивное программирование

Спортивное программирование

МФТИ
Moscow Workshops
mail.ru group

МФТИ
Moscow Workshops
ДМК ИЗДАТЕЛЬСТВО
mail.ru group

Стивен Халим, Феликс Халим

Спортивное программирование

Competitive Programming 3

The New Lower Bound of Programming Contests

Steven Halim, Felix Halim

Спортивное программирование

Новый нижний предел соревнований по программированию

Стивен Халим, Феликс Халим

ДМК Пресс
издательство
Москва, 2020

УДК 004.02, 004.424
ББК 22.18
Х17

Халим С., Халим Ф.
Х17 Спортивное программирование / пер. с англ. Н. Б. Желновой, А. В. Снастина. – М.: ДМК Пресс, 2020. – 604 с.: ил.

ISBN 978-5-97060-758-9

Книга содержит задачи по программированию, аналогичные тем, которые используются на соревнованиях мирового уровня (в частности, ACM ICPC и IOI). Помимо задач разного типа приводятся общие рекомендации для подготовки к соревнованиям, касающиеся классификации заданий, анализа алгоритмов и пр. Кроме стандартных тем (структуры данных и библиотеки, графы, математика, вычислительная геометрия) авторы затрагивают и малораспространенные – им посвящена отдельная глава.

В конце каждой главы приводятся краткие решения заданий, не помеченных звездочкой, или даются подсказки к ним. Задания сложного уровня (помеченные звездочкой) требуют самостоятельной проработки.

Издание адресовано читателям, которые готовятся к соревнованиям по программированию или просто любят решать задачи по информатике. Для изучения материала требуются элементарные знания из области методологии программирования и знакомство хотя бы с одним из двух языков программирования – C/C++ или Java.

УДК 004.02, 004.424
ББК 22.18

ISBN 978-5-97060-758-9 (рус.)

Содержание

Издательство «ДМК Пресс» выражает благодарность техническим редакторам книги «Спортивное программирование», а именно:

- **Олегу Христенко** – главный судья Moscow Workshops; технический координатор Олимпиадных школ МФТИ, Moscow Workshops Juniors и международных сборов по программированию для подготовки к соревнованиям по программированию; сопредседатель жюри Moscow Programming Contest; соавтор курса «Быстрый старт в спортивное программирование» в рамках RuCode;
- **Филиппу Руховичу** – к.ф.-м.н.; преподаватель кафедры алгоритмов и технологий программирования МФТИ; двукратный призер и победитель Всероссийской олимпиады школьников по информатике; финалист ACM ICPC 2014; четырехкратный призер NEERC (2010–2013); сотренер бронзовых призеров ICPC 2019; методист отделения информатики летних и зимних Олимпиадных школ МФТИ; соавтор курса «Быстрый старт в спортивное программирование» в рамках RuCode;
- **Владиславу Невструеву** – преподаватель летних и зимних Олимпиадных школ МФТИ, Летней компьютерной школы; преподаватель «Открытых Московских тренировок»; автор задач на олимпиады: «Квалификационный этап Moscow Programming Contest», «Когнитивные технологии», «Муниципальный этап Всероссийской олимпиады школьников»; соавтор курса «Быстрый старт в спортивное программирование» в рамках RuCode.

Коллеги из Национального университета Сингапура оказывают значительное влияние на развитие сообщества олимпиадного программирования во всем мире. С молодым исследователем в области компьютерных наук Стивеном Халимом и его коллегой профессором Сан Теком мы в 2019 году провели сборы Discover Singapore в рамках международного образовательного проекта Moscow Workshops, который зародился 8 лет назад на кампусе Московского физико-технического института (МФТИ). Сингапур в 2020 году готовился стать городом проведения международной олимпиады школьников по информатике IOI. Хотя в планы вмешалась пандемия, олимпиада все же пройдет в онлайн-режиме, а очно город примет IOI в 2021 году.

В России достаточно развито соревновательное программирование, популярны олимпиады и чемпионаты на школьном и студенческом уровне. Благодаря высоким достижениям российских студентов на международных соревнованиях по программированию, заявку на проведение чемпионата мира по программированию «ICPC World Finals 2020» выиграла Москва, принимающим вузом стал МФТИ. Российские студенты последние 20 лет уверенно доминируют на этом соревновании, а МФТИ – единственный вуз в мире, который на сегодняшний день имеет непрерывную череду медалей ICPC - больше трех. Мы создали самый большой образовательный проект в мире по алгоритмическому программированию Moscow Workshops, который принес новые возможности более 3,5 тысячи студентов 61 страны. Независимым критерием подтверждения их успехов становятся результаты на чемпионате ICPC и старт карьеры в ведущих компаниях ИТ-индустрии.

На кампусе московского Физтеха с 2018 года проводится отбор и подготовка национальной сборной на IOI. Также к этой олимпиаде школьников со всего мира могут подготовиться в школе Moscow Workshops Juniors. В 2019 году российская сборная взяла 4 золотые медали IOI, трое из этих ребят учились в Juniors, как и еще 12 медалистов из сборных Беларуси, Азербайджана, Киргизии, Дании, Сирии, Турции, Болгарии и Индии. Первые выпускники лагеря уже показывают выдающиеся успехи: в прошлом году стартап AI Factory, созданный выпускником Juniors Александром Машрабовым с напарниками, купила компания Snap за $166 млн. В 2020 году мы совместно с 10 российскими вузами на принципах открытости, равенства, уважения, единства запустили первый учебный фестиваль по алгоритмическому программированию и искусственному интеллекту RuCode для всех желающих попробовать свои силы. В рамках фестиваля мы выпустили вводный онлайн-курс «Быстрый старт в спортивное программирование» на платформе Stepik, провели интенсивы и чемпионаты по искусственному интеллекту и алгоритмическому программированию. Мероприятия охватили 12 тысяч человек, в том числе взрослых, и мы будем проводить фестиваль и дальше, чтобы популяризировать ИТ-знания. RuCode был задуман так, чтобы сильные технические вузы из разных регионов России смогли передать свои лучшие практики учащимся, дать широкий спектр образовательных методик. Книга Стивена Халима – это тоже экскурс, но в систему подготовки коллег с Востока, которые так же стабильно показывают выдающиеся результаты на соревнованиях по программированию. Уверены, что книга поможет изучить все самые необходимые алгоритмы и поднять ваш уровень знаний в программировании до уровня чемпионов мира.

Алексей Малеев,
проректор МФТИ,
основатель международного образовательного проекта Moscow Workshops

МФТИ

Moscow Workshops

На протяжении всего существования Mail.ru Group мы ставили перед собой амбициозные цели, главная из которых – стать центром притяжения самых ярких талантов, способных генерировать новые идеи, оставаясь при этом на стыке практики и академии.

Выиграть борьбу за таланты можно, только позволив каждому участнику раскрыть свой потенциал, в том числе и в рамках чемпионатов. За почти 10 лет мы провели их больше 100, ориентируясь на ведущие мировые практики. В процессе работы мы поняли, что конкуренция и соревновательный дух, вступающие в синергию с опытом и экспертизой, позволяют достигать по-настоящему заметных высот.

Уверен, что книга «Спортивное программирование» Стивена Халима и Феликса Халима поможет не только школьникам и студентам, поставившим перед собой амбициозную цель добиться заметных результатов на международных олимпиадах ICPC и IOI, но и всем тем, кто мечтает достичь новых профессиональных высот и стать более конкурентоспособным при решении сложных IT-задач.

Желаю вам увлекательного чтения. Дерзайте!

Дмитрий Смыслов,
вице-президент по персоналу и образовательным проектам

mail.ru group

Спортивное программирование в некотором роде определило мою жизнь. Я увлекся разработкой еще на ZX Spectrum, позже принял участие в школьной олимпиаде, задачи которой показались довольно простыми… и не смог остановиться. Программирование увлекло меня свободой, которую оно дает для выражения мыслей: здесь практически нет рамок, и любую задумку можно реализовать быстро и эффективно. Победа на олимпиадах помогла поступить на факультет ВМК МГУ, а затем пройти путь от джуна-разработчика до руководителя отдела и совладельца бизнеса.

Сейчас я издаю медиа Tproger. В комментариях мы часто сталкиваемся с мнением, что для работы спортивное программирование бесполезно: «в повседневной жизни требуется решать задачи, которые никак не связаны с теми, что предлагаются на контестах вроде ICPC». Я не согласен.

Конечно, напрямую навыки спортивного программирования не помогут. Но придумывание оптимальных алгоритмов, воспитание в себе внимания к деталям, да хотя бы просто разминка для ума значительно упрощают достижение успехов в карьере разработчика. Именно по этой причине я всегда с симпатией относился к кандидатам, которые участвовали в олимпиадах по программированию или просто нарешивали задачки из архивов. Везде есть исключения, но такие люди чаще проектировали и реализовывали действительно хорошие продукты.

Готовы попробовать? Регистрируйтесь на любом сайте с архивом задач, берите эту книгу в помощники, и вперед.

Если вы уже участвуете в контестах, то найдете в книге «Спортивное программирование» Стивена Халима и Феликса Халима много полезных техник и советов, чтобы стать лучше.

И помните, если будет сложно – это нормально. Не останавливайтесь, только так можно достичь настоящих высот в любом деле.

Алексей Михайлишин,
генеральный директор Tproger

Вступление

Однажды (это случилось 11 ноября 2003 года) я получил письмо по электронной почте, автор которого писал мне:

> «Я должен сказать, что на сайте университета Вальядолида Вы положили начало новой ЦИВИЛИЗАЦИИ. Своими книгами (он имел в виду «Задачи по программированию: пособие по подготовке к олимпиадам по программированию» [60], написанное в соавторстве со Стивеном Скиеной) Вы вдохновляете людей на подвиги. Желаю Вам долгих лет жизни, чтобы служить человечеству, создавая новых супергероев – программистов».

Хотя это было явным преувеличением, письмо заставило меня задуматься. У меня была мечта: создать сообщество вокруг проекта, который я начал как часть моей преподавательской работы в университете Вальядолида. Я мечтал создать сообщество, которое объединило бы множество людей с разных концов света, связанных единой высокой целью. Выполнив несколько запросов в поисковике, я быстро нашел целое онлайн-сообщество, объединявшее несколько сайтов, обладавших всем тем, чего не хватало сайту университета Вальядолида.

Сайт «Методы решения задач» Стивена Халима, молодого студента из Индонезии, показался мне одним из наиболее интересных. Я увидел, что однажды мечта может стать реальностью, потому что на этом сайте я нашел результат кропотливой работы гения в области алгоритмов и программирования. Более того, цели, о которых он говорил, вполне соответствовали моей мечте: служить человечеству. И еще я узнал, что у него есть брат, Феликс Халим, разделяющий его увлечение программированием.

До начала нашего сотрудничества прошло довольно много времени, но, к счастью, все мы продолжали параллельно работать над реализацией этой мечты. Книга, которую вы сейчас держите в руках, является лучшим тому доказательством.

Я не могу представить лучшего дополнения для архива задач университета Вальядолида. Эта книга использует множество примеров с сайта университета Вальядолида, тщательно отобранных и разбитых по категориям по типам задач и методам их решения. Она оказывает огромную помощь пользователям данного сайта. Разобрав и решив большинство задач по программированию из этой книги, читатель сможет легко решить не менее 500 задач, предложенных «Онлайн-арбитром» университета Вальядолида, и попасть в число 400–500 лучших среди 100 000 пользователей «Онлайн-арбитра».

Книга «Спортивное программирование» также подходит для программистов, которые хотят улучшить свои позиции на региональных и международных соревнованиях по программированию. Ее авторы прошли через эти соревнования (ICPC и IOI) вначале как участники, а теперь и как тренеры. Она также рекомендуется новичкам – как говорят Стивен и Феликс во введении: «Книгу рекомендуется прочесть не один, а несколько раз».

Кроме того, она содержит исходный код на C++, реализующий описанные алгоритмы. Понимание задачи – это одно, знание алгоритма ее решения – другое, а правильная реализация решения через написание краткого и эффективного кода – третья непростая задача. Прочитав эту книгу трижды, вы поймете, что ваши навыки программирования значительно улучшились и, что еще важнее, вы стали более счастливым человеком, чем были раньше.

Мигель А. Ревилла,
Университет Вальядолида,
создатель сайта «Онлайн-арбитр»,
член Международного оргкомитета ICPC
http://uva.onlinejudge.org; http://livearchive.onlinejudge.org

Участники финального мирового турнира в Варшаве, 2012.
Слева направо: Фредерик Нимеля, Карлос, Мигель Ревилла, Мигель-младший, Феликс, Стивен

Предисловие

Эта книга обязательна к прочтению для каждого программиста, участвующего в соревнованиях по программированию. Овладеть содержанием данной книги необходимо (но, возможно, недостаточно) для того, чтобы сделать шаг вперед от простого обычного программиста до одного из лучших программистов в мире.

Для кого написана эта книга:

- для студентов университетов, участвующих в ежегодных всемирных студенческих соревнованиях по программированию ICPC [66] в качестве участников региональных соревнований (включая финальные мировые турниры);
- для учащихся средних или старших классов школ, принимающих участие в ежегодной Международной олимпиаде по информатике (IOI) [34] (включая национальные или региональные олимпиады);
- для тренеров сборных команд по программированию, преподавателей и наставников, которые ищут полноценные учебные материалы для своих воспитанников [24];
- для всех, кто любит решать задачи по программированию. Для тех, кто больше не имеет права участвовать в ICPC, проводятся многочисленные состязания по программированию, в том числе TopCoder Open, Google CodeJam, интернет-конкурс по решению задач (IPSC) и т. д.

Требования к уровню подготовки

Эта книга *не* предназначена для начинающих программистов. Она написана для читателей, которые имеют хотя бы элементарные знания из области методологии программирования, знакомы хотя бы с одним из двух языков программирования – C/C++ или Java (а еще лучше с обоими), освоили начальный курс по структурам данных и алгоритмам (обычно он включается в программу первого года обучения в университете для специальностей в области информатики) и знакомы с простым алгоритмическим анализом (по крайней мере, им знакома нотация Big O). Третье издание было значительно переработано и дополнено, так что эта книга также может быть использована в качестве *дополнительного материала для начального курса по структурам данных и алгоритмам*.

Участникам ICPC

Мы знаем, что вряд ли возможно одержать победу в ICPC, просто прочитав новую версию этой книги и усвоив ее содержание. Хотя мы включили в нее много полезного материала – гораздо больше, чем содержалось в первых двух изда-

ниях, – мы понимаем, что для достижения этой цели требуется гораздо больше, чем может дать данная книга. Для читателей, которые хотят большего, мы приводим дополнительные ссылки на полезные источники. Однако мы полагаем, что после освоения содержания этой книги ваша команда будет чувствовать себя намного увереннее в будущих состязаниях ICPC. Мы надеемся, что эта книга послужит как источником вдохновения, так и стимулом к участию в соревнованиях ICPC во время вашей учебы в университете.

Участникам IOI

Большая часть наших советов для участников ACM ICPC пригодится и вам. Программы ICPC и IOI в значительной мере схожи, однако IOI на данный момент исключает темы, перечисленные в табл. П1. Вы можете пропустить соответствующие главы этой книги, отложив их до тех пор, пока не поступите в университет (и не присоединитесь к команде этого университета, участвующей в ICPC). Тем не менее полезно изучить эти методы заранее, так как дополнительные знания могут помочь вам в решении некоторых задач в IOI.

Мы знаем, что нельзя получить медаль IOI, просто основательно проштудировав эту книгу. Мы включили многие разделы программы IOI в данную книгу и надеемся, что вы сможете добиться достойного результата на олимпиадах IOI. Однако мы хорошо понимаем, что задачи, предлагаемые на IOI, требуют сильных навыков решения задач и огромного творческого потенциала – тех качеств, которые вы не сможете получить, только читая учебник. Книга может дать вам знания, но в конечном итоге вы должны проделать огромную работу. С практикой приходит опыт, а с опытом приходит навык. Так что продолжайте практиковаться!

Слева направо: Дэниел, мистер Чонг, Раймонд, Стивен, Чжан Сюн, д-р Рональд, Чуанци

Таблица П1. Темы, не входящие в программу IOI [20]

Тема	**В этой книге:**
Структуры данных: система непересекающихся множеств	Раздел 2.4.2
Теория графов: нахождение компонентов сильной связности, поток в сети, двудольные графы	Разделы 4.2.1, 4.6.3, 4.7.4
Математические задачи: операции с очень большими числами BigInteger, теория вероятностей, игра «Ним»	Разделы 5.3, 5.6, 5.8
Обработка строк: суффиксные деревья и массивы	Раздел 6.6
Более сложные темы: A*/IDA*	Раздел 8.2
Нестандартные задачи	Глава 9

Преподавателям и наставникам

Эта книга используется как учебное пособие на курсе Стивена CS3233 «Олимпиадное программирование» в Школе программирования Национального университета Сингапура. Продолжительность курса CS3233 составляет 13 учебных недель, его примерная программа приведена в табл. П2. Слайды для этого курса в формате PDF опубликованы на веб-сайте данной книги. Коллегам – преподавателям и наставникам рекомендуется изменять программу в соответствии с потребностями студентов. В конце каждой главы данной книги приводятся подсказки или краткие решения заданий, не помеченных звездочкой. Некоторые из помеченных заданий довольно сложны и не имеют ни ответов, ни решений. Их можно использовать в качестве экзаменационных вопросов или конкурсных заданий (разумеется, сначала нужно их решить).

Эта книга также применяется в качестве дополнительного материала в программе курса CS2010 «Структуры данных и алгоритмы», в основном для иллюстрации реализации нескольких алгоритмов и выполнения заданий по программированию.

Таблица П2. Программа курса CS3233 «Олимпиадное программирование»

Неделя	Тема	В этой книге:
01	Введение	Глава 1, разделы 2.2, 5.2, 6.2–6.3, 7.2
02	Структуры данных и библиотеки	Глава 2
03	Полный поиск, стратегия «разделяй и властвуй», «жадный» алгоритм	Разделы 3.2–3.4; 8.2
04	Динамическое программирование: основы	Разделы 3.5; 4.7.1
05	Динамическое программирование: техники	Разделы 5.4; 5.6; 6.5; 8.3
06	Командные соревнования в середине семестра	Главы 1–4; часть главы 9
–	Перерыв в середине семестра (домашнее задание)	
07	Графы, часть 1 (поток в сети)	Раздел 4.6; часть главы 9
08	Графы, часть 2 (поиск соответствий)	Раздел 4.7.4; часть главы 9
09	Математика (обзор)	Глава 5
10	Обработка строк (основные навыки, массив суффиксов)	Глава 6
11	(Вычислительная) Геометрия (библиотеки)	Глава 7
12	Более сложные темы	Раздел 8.4; часть главы 9
13	Заключительные командные соревнования	Главы 1–9 (не ограничиваясь только ими)

Для курсов по структурам данных и алгоритмам

В этом издании содержание данной книги было переработано и дополнено таким образом, что первые четыре главы книги стали более доступными для студентов первого курса, специализирующихся в области информатики и теории вычислительных систем. Темы и упражнения, которые мы сочли относительно сложными для начинающих, были перенесены в главы 8 и 9. Надеемся, что студенты, только начинающие свой путь в области компьютерных наук, не испугаются трудностей, просматривая первые четыре главы.

Глава 2 была основательно переработана. Ранее раздел 2.2 представлял собой просто список классических структур данных и их библиотек. В данном издании мы расширили описание и добавили множество упражнений, чтобы эту книгу можно было также использовать как пособие для курса по структурам данных, особенно с точки зрения *деталей реализации*.

О четырех подходах к решению задач, обсуждаемых в главе 3 этой книги, часто рассказывается в различных курсах по *алгоритмам*.

Текст предисловия также был переработан, чтобы помочь новым студентам в области информатики ориентироваться в структуре книги.

Часть материала из главы 4 можно использовать в качестве дополнительной литературы либо в качестве наглядного примера реализации. Этот материал пригодится для повышения уровня знаний в области *дискретной математики* [57, 15] или для начального курса по *алгоритмам*. Мы также представили новый взгляд на методы динамического программирования как на алгоритмы, использующие ориентированные ациклические графы. К сожалению, во многих учебниках по компьютерным наукам такие темы до сих пор встречаются редко.

Всем читателям

Поскольку эта книга охватывает множество тем, обсуждаемых вначале более поверхностно, затем более глубоко, ее рекомендуется прочитать не один, а несколько раз. Книга содержит много упражнений (их около 238) и задач по программированию (их около 1675); практически в каждом разделе предлагаются различные задания и упражнения. Вы можете сначала пропустить эти упражнения, если решение слишком сложно или требует дополнительных знаний и навыков, чтобы потом вернуться к ним после изучения следующих глав. Решение предложенных задач углубит понимание понятий, изложенных в этой книге, поскольку они обычно включают интересные практические аспекты обсуждаемой темы. Постарайтесь их решить – время, потраченное на это, точно не будет потрачено впустую.

Мы считаем, что данная книга актуальна и будет интересна многим студентам университетов и старших классов. Олимпиады по программированию, такие как ICPC и IOI, по крайней мере, еще много лет будут иметь похожую программу. Новые поколения студентов должны стремиться понять и усвоить элементарные знания из этой книги, прежде чем переходить к более серьезным задачам. Однако слово «элементарные» может вводить в заблуждение – пожалуйста, загляните в содержание, чтобы понять, что мы подразумеваем под «элементарным».

Как ясно из названия этой книги, ее цель – развить навыки программирования и тем самым поднять уровень всемирных олимпиад по программированию, таких как ICPC и IOI. Поскольку все больше участников осваивают ее содержание, мы надеемся, что 2010 год (год, когда было опубликовано первое издание этой книги) стал переломным моментом, знаменующим существенное повышение стандартов соревнований по программированию. Мы надеемся помочь большему количеству команд справиться более чем с двумя ($\geqslant 2$) задачами на будущих олимпиадах ICPC и позволить большему количеству участников получить более высокие ($\geqslant 200$) баллы на будущих олимпиадах IOI. Мы также надеемся, что многие тренеры ICPC и IOI во всем мире (особенно в Юго-Восточной Азии) выберут эту книгу и оценят помощь, которую она оказывает при выборе материала для подготовки к соревнованиям тем, без кого студенты не могут обойтись на олимпиадах по программированию, – преподавателям и наставникам. Мы, ее авторы, будем очень рады, что нам удалось внести свой вклад в распространение необходимых «элементарных» знаний для олимпиадного программирования, поскольку в этом заключается основная цель нашей книги.

Принятые обозначения

Эта книга содержит множество примеров кода на языке C/C++, а также отдельные примеры кода на Java (в разделе 5.3). Все фрагменты кода, включенные в книгу, выделяются моноширинным шрифтом.

Для кода на C/C++ в данной книге мы часто используем директивы `typedef` и макросы – функции, которые обычно применяются программистами, участ-

вующими в соревнованиях, для удобства, краткости и скорости кодирования. Однако мы не можем использовать аналогичные методы для Java, поскольку они не содержат аналогичных функций. Вот несколько примеров наших сокращений для кода C/C++:

```
// Отключение некоторых предупреждений компилятора (только для пользователей VC++)
#define _CRT_SECURE_NO_DEPRECATE

// Сокращения, используемые для наиболее распространенных типов данных в олимпиадном
программировании
typedef long long       ll;                    // комментарии, которые смешиваются с кодом,
typedef pair<int, int>  ii;                              // выравниваются подобным образом
typedef vector<ii>      vii;
typedef vector<int>     vi;
#define INF 1000000000              // 1 миллиард, что предпочтительнее, чем сокращение 2B,
                                                        // в алгоритме Флойда-Уоршелла
// Общие параметры memset
//memset(memo, -1, sizeof memo);            // инициализация таблицы, сохраняющей значения
вычислений
                                        // в динамическом программировании, значением -1
//memset(arr, 0, sizeof arr);                              // очистка массива целых чисел
// Мы отказались от использования "REP" and "TRvii" во втором и последующих
// изданиях, чтобы не запутывать программистов
```

Следующие сокращения часто используются как в примерах кода на C/C++, так и в примерах кода на Java:

```
// ans = a ? b : c;                         // для упрощения: if (a) ans = b; else ans = c;
// ans += val;                             // для упрощения: ans = ans + val; и ее вариантов
// index = (index + 1) % n;                          // index++; if (index >= n) index = 0;
// index = (index + n - 1) % n;              // index - - ; if (index < 0) index = n - 1;
// int ans = (int)((double)d + 0.5);               // для округления до ближайшего целого
// ans = min(ans, new_computation);                                  // сокращение min/max
// альтернативная форма записи, не используемая в этой книге: ans <?= new_computation;
// в некоторых примерах кода используются обозначения: && (AND) и || (OR)
```

Категории задач

К маю 2013 года Стивен и Феликс совместно решили 1903 задачи по программированию, размещенные на сайте университета Вальядолида (что составляет 46,45 % от общего числа задач, размещенных на этом сайте). Около 1675 из них мы разбили по категориям и включили в эту книгу. В конце 2011 года список задач «Онлайн-арбитра» на сайте университета Вальядолида пополнился некоторыми задачами из Live Archive. В этой книге мы приводим оба индекса задач, однако основным идентификатором, используемым в разделе предметного указателя данной книги, является номер задачи, присвоенный ей на сайте университета Вальядолида. Задачи распределялись по категориям в соответствии со схемой «балансировки нагрузки»: если задачу можно разделить на две или более категорий, то она будет отнесена к категории, к которой в настоящий момент относится меньшее число задач. Таким образом, вы можете

заметить, что некоторые задачи были «ошибочно» классифицированы, и категория, к которой отнесена данная задача, может не соответствовать методу, который вы использовали для ее решения. Мы можем лишь гарантировать, что если задача X определена в категорию Y, то нам удалось решить задачу X с помощью метода, обсуждаемого в разделе, посвященном категории задач Y.

Мы также ограничили число задач в каждой категории: в каждую из категорий мы включили не более 25 (двадцати пяти) задач, разбив их на отдельные дополнительные категории.

Если вы хотите воспользоваться подсказками к любой из решенных задач, воспользуйтесь указателем в конце этой книги, а не пролистывайте каждую главу – это позволит сэкономить время. Указатель содержит список задач с сайта университета Вальядолида / Live Archive с номером задачи (используйте двоичный поиск!) и номера страниц, на которых обсуждаются упомянутые задачи (а также структуры данных и/или алгоритмы, необходимые для их решения). В третьем издании подсказки занимают более одной строки, и они стали гораздо понятнее, чем в предыдущих изданиях.

Используйте различные категории задач для тренировки навыков в различных областях! Решение, по крайней мере, нескольких задач из каждой категории (особенно тех, которые мы пометили **знаком** *) – это отличный способ расширить ваши навыки решения задач. Для краткости мы ограничились максимум тремя основными задачами в каждой категории, пометив их особо.

Изменения во втором издании

Первое и второе издания этой книги существенно различаются. Авторы узнали много нового и решили сотни задач из области программирования за один год, разделяющий эти два издания. Мы получили отзывы от читателей, в частности от студентов Стивена, прослушавших курс CS233 во втором семестре 2010/2011 года, и включили их предложения во второе издание.

Краткий перечень наиболее важных изменений во втором издании:

- во-первых, изменился дизайн книги. Увеличилась плотность информации на каждой странице. В частности, мы уменьшили междустрочный интервал, сделали более компактным размещение мелких рисунков на страницах. Это позволило нам не слишком сильно увеличивать толщину книги, существенно расширив ее содержание;
- были исправлены некоторые незначительные ошибки в наших примерах кода (как те, что приведены в книге, так и их копии, опубликованные на ее веб-сайте). Все примеры кода теперь имеют более понятные комментарии;
- исправлены некоторые опечатки, грамматические и стилистические ошибки;
- помимо того что мы улучшили описание многих структур данных, алгоритмов и задач программирования, мы также существенно расширили содержание каждой из глав:
  1) добавлено множество новых специальных задач в начале книги (раздел 1.4);

2) в книгу включено обсуждение булевых операций (операций над битами) (раздел 2.2), неявных графов (раздел 2.4.1) и структур данных дерева Фенвика (раздел 2.4.4);
3) расширена часть, посвященная динамическому программированию: дано более четкое объяснение динамического программирования по принципу «от простого к сложному», приведено решение $O(n \log k)$ для задачи LIS, решение о нахождении суммы подмножества в задаче о рюкзаке (Рюкзак 0–1), рассмотрено решение задачи о коммивояжере методами динамического программирования (с использованием операций с битовыми масками) (раздел 3.5.2);
4) реорганизован материал по теории графов в обсуждении следующих тем: методы обхода графа (поиск в глубину и поиск в ширину), построение минимального остовного дерева, нахождение кратчайших путей (из заданной вершины во все вершины и между всеми парами вершин), вычисление максимального потока и обсуждение задач, относящихся к специальным графам. Добавлены новые темы: обсуждение алгоритма Прима, использование методов динамического программирования на графах при обходе неявных направленных ациклических графов (раздел 4.7.1), эйлеровы графы (раздел 4.7.3) и алгоритм поиска аугментальных цепей (раздел 4.7.4);
5) переработан материал, касающийся обсуждения математических методов (глава 5): специальные задачи, использование BigInteger (в Java), комбинаторика, теория чисел, теория вероятностей, поиск циклов, теория игр (новый раздел) и степень (квадратной) матрицы (новый раздел). Улучшена подача материала с точки зрения легкости восприятия их читателями;
6) добавлен материал, позволяющий получить элементарные навыки обработки строк (раздел 6.2), расширен круг задач, связанных с работой со строками (раздел 6.3), включая сопоставление строк (раздел 6.4), расширен раздел, посвященный теме суффиксных деревьев / массивов (раздел 6.6);
7) расширен набор геометрических библиотек (глава 7), в частности библиотек, позволяющих выполнять операции над точками, линиями и многоугольниками;
8) добавлена глава 8, в которой обсуждаются вопросы декомпозиции задач, расширенные методы поиска (A*, поиск с ограничением глубины, итеративное углубление, IDA*), расширенные методы динамического программирования (использование битовой маски, задача о китайском почтальоне, обзор общих положений динамического программирования, обсуждение лучших применений метода динамического программирования и некоторые более сложные темы);

- был переработан и улучшен иллюстративный материал книги. Также были добавлены новые иллюстрации, помогающие сделать объяснение материала более понятным;
- первое издание книги было написано в основном с точки зрения участника ICPC и программиста на C++. Материал второго издания более

сбалансирован и включает в себя информацию, чья целевая аудитория – участники соревнований IOI. Также во втором издании намного увеличилось число примеров на Java. Однако мы пока не включаем в книгу примеры на других языках программирования;

- это издание книги также включает материалы веб-сайта Стивена «Методы решения»: «однострочные подсказки» для каждой задачи и список задач с указанием категорий в конце этой книги. Теперь решение 1000 задач из списка задач «Онлайн-арбитра» на сайте университета Вальядолида – вполне достижимая цель (мы считаем, что это по силам серьезному студенту 4-го курса университета);
- некоторые примеры в первом издании используют устаревшие задачи. Во втором издании эти примеры были заменены или обновлены;
- Стивен и Феликс решили добавить в эту книгу еще 600 задач по программированию из тестирующей системы и онлайн-архива с сайта университета Вальядолида. Мы также добавили в книгу множество практических упражнений с подсказками или краткими решениями;
- книга пополнилась краткими биографиями изобретателей структур данных и авторов алгоритмов, заимствованными из Википедии [71] и из других источников, чтобы вы могли немного больше узнать о людях, оставивших свой след в области программирования.

Изменения в третьем издании

За два года, прошедших с момента выхода второго издания, мы значительно улучшили и дополнили материалы книги, включенные в третье издание.

Краткий перечень наиболее важных изменений в третьем издании:

- мы немного увеличили размер шрифта в третьем издании (12 пунктов) по сравнению со вторым изданием (11 пунктов) и изменили размеры полей. Надеемся, что многие читатели отметят, что текст стал более читабельным. Мы также увеличили размер иллюстраций. Это, однако, привело к тому, что книга стала значительно толще;
- мы изменили макет книги. Теперь почти каждый раздел начинается с новой страницы;
- мы добавили еще *много* практических упражнений в каждый из разделов книги – как не отмеченных звездочкой (предназначенных для самопроверки; подсказки и решения для этих задач приведены в конце каждой главы), так и помеченных звездочкой * (дополнительных задач; решение не предусмотрено). Материалы практических упражнений соседствуют с обсуждением соответствующей темы в тексте глав;
- Стивен и Феликс решили еще 477 задач по программированию из тестирующей системы и онлайн-архива с сайта университета Вальядолида (UVa), которые впоследствии были добавлены в эту книгу. Таким образом, мы сохранили значительный (около 50 %, а точнее 46,45 %) охват задач, представленных на сайте «Онлайн-арбитра», даже несмотря на то, что «Онлайн-арбитр» сильно вырос за тот же период времени. Эти новые

задачи были выделены курсивом. Некоторые из новых задач заменили старые, которые рекомендуется обязательно решить. Все задачи теперь всегда размещаются в конце раздела;

- сейчас мы с уверенностью можем сказать, что способные студенты могут достичь впечатляющих успехов в решении задач (более 500 решенных задач тестирующей системы на сайте университета Вальядолида) всего за один университетский семестр (4 месяца), прочитав эту книгу;
- перечень новых (или переработанных) материалов по главам:
  1) введение главы 1 было адаптировано для читателей, которые плохо знакомы с олимпиадным программированием. Мы разработали более строгие форматы ввода-вывода (I/O) для типичных задач программирования и общих процедур их решения;
  2) мы добавили еще одну линейную структуру данных 'deque' (дек) в разделе 2.2, а также углубили разъяснение практически всех структур данных, обсуждаемых в главе 2, особенно в разделах 2.3 и 2.4;
  3) в главе 3 мы более подробно обсудим различные методы полного перебора: вложенные циклы, итеративную генерацию подмножеств/перестановок и рекурсивный поиск. Новое: интересный трюк для написания и печати решений с использованием «нисходящего» метода динамического программирования, обсуждение алгоритма Кадана о поиске максимальной суммы диапазона для одномерного случая;
  4) в главе 4 мы заменили белые/серые/черные метки (унаследованные из источника [7]) их стандартными обозначениями, заменили термин «максимальный поток» на «сетевой поток». Мы также сослались на опубликованную научную работу автора алгоритма, чтобы читатель мог ознакомиться и понять оригинальную идею, лежащую в основе алгоритма. Мы обновили диаграммы неявного направленного ациклического графа в классических задачах динамического программирования в разделе 3.5;
  5) в главе 5 мы расширили круг обсуждаемых специальных задач математики, обсуждение интересной операции Java BigInteger: `isProbablePrime`, добавили и расширили несколько часто используемых формул комбинаторики и разновидностей алгоритмов решета, дополнили и пересмотрели разделы по теории вероятностей (раздел 5.6), поиску циклов (раздел 5.7) и теории игр (раздел 5.8);
  6) в главе 6 мы переписали раздел 6.6, сделав понятнее объяснение суффиксного бора / дерева / массива суффиксов, и заново определили понятие завершающего символа;
  7) главу 7 мы разбили на два основных раздела и улучшили качество кода библиотеки;
  8) в главу 8 (а также последующую главу 9) включены наиболее сложные темы, ранее обсуждавшиеся в главах 1–7 второго издания. Новое: обсуждение более сложной процедуры возвратной рекурсии, поиска в пространстве состояний, метода «встреча посередине», некоторые неочевидные приемы с использованием сбалансированного дерева двоичного поиска в качестве мемотаблицы и углубленный раздел о декомпозиции задач;

9) новая глава 9. Добавлены различные редкие темы, появляющиеся время от времени на олимпиадах по программированию. Некоторые из них просты, однако многие достаточно сложны и могут сильно повлиять на ваш рейтинг в соревнованиях по программированию.

Сайты, сопутствующие книге

Официальный веб-сайт этой книги имеет адрес sites.google.com/site/stevenhalim, на нем размещена электронная версия примеров исходного кода из данной книги и слайды в формате PDF, используемые в курсе Стивена CS3233.

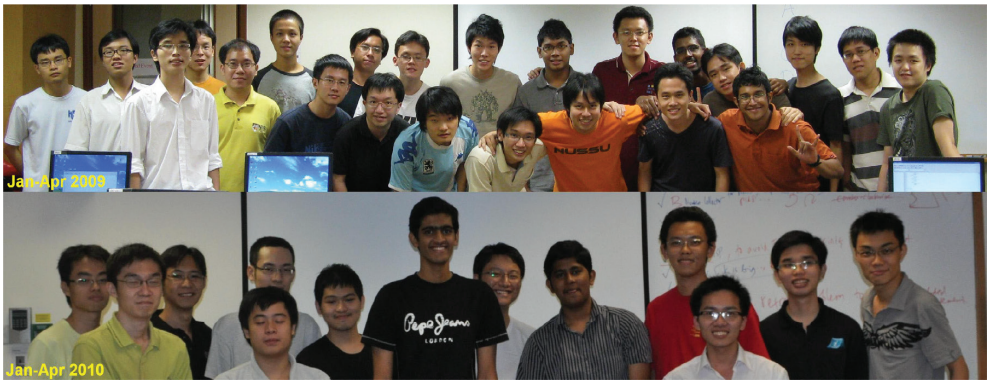
Все задачи по программированию здесь собраны в приложении uhunt.felixhalim.net и размещены в архиве задач сайта университета Вальядолида: uva.onlinejudge.org.

Новое в третьем издании: многие алгоритмы теперь сопровождаются интерактивной визуализацией: www.comp.nus.edu.sg/~stevenha/visualization.

Благодарности к первому изданию

От Стивена: я хочу поблагодарить:

- Бога, Иисуса Христа и Святого Духа за данный мне талант и страсть к олимпиадному программированию;
- мою любимую жену Грейс Сурьяни за то, что она позволила мне потратить драгоценное время, отведенное нам с ней, на этот проект;
- моего младшего брата и соавтора, Феликса Халима, за то, что он поделился многими структурами данных, алгоритмами и приемами программирования, внеся значительный вклад в улучшение этой книги;
- моего отца Линь Цзе Фонга и мать Тан Хой Лан за данное нам воспитание и мотивацию, необходимые для достижения хороших результатов в учебе и труде;



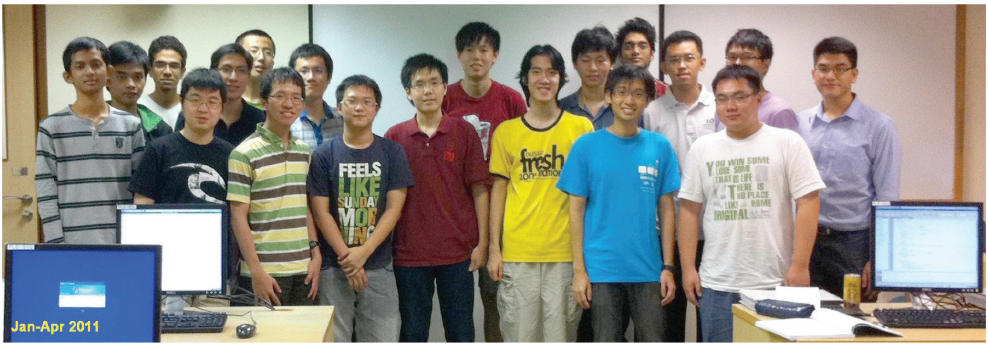
- Школу программирования Национального университета Сингапура за предоставленную мне возможность работать в этом университете и пре-

подавать учебный курс CS3233 «Олимпиадное программирование», на основе которого и была создана эта книга;
- профессоров и преподавателей, работавших и работающих в Национальном университете Сингапура, давших мне навыки олимпиадного программирования и наставничества: профессора Эндрю Лима Леонга Чи, доцента Тана Сун-Тека, Аарона Тан Так Чой, доцента Сунг Вин Кина, доктора Алана Ченга Хо-Луна;
- моего друга Ильхама Вината Курния за корректуру рукописи первого издания;
- помощников преподавателей курса CS3233 и наставников команд – участников олимпиад ACM ICPC в NUS: Су Чжана, Нго Минь Дюка, Мелвина Чжан Чжиюна, Брамандию Рамадхана;
- моих студентов, обучавшихся на курсе CS3233 во втором семестре 2008/2009 учебного года и вдохновивших меня на создание заметок к моим занятиям, а также студентов, посещавших этот курс во втором семестре 2009/2010 учебного года, которые проверили содержание первого издания этой книги и положили основу онлайн-архива задач Live Archive.

Благодарности ко второму изданию

От Стивена: я также хочу поблагодарить:

- первых 550 покупателей первого издания (по состоянию на 1 августа 2011 года). Ваши положительные отзывы вдохновляют нас!
- помощника преподавателя курса CS3233 Национального университета Сингапура Виктора Ло Бо Хуай;
- моих студентов, обучавшихся на курсе CS3233 во втором семестре 2010/2011 учебного года, которые участвовали в технической подготовке и презентации второго издания (далее их фамилии приведены в алфавитном порядке): Алдриана Обая Муиса, Бах Нгок Тхань Конга, Чэнь Цзюньчэна, Девендру Гоял, Фикрила Бахри, Хасана Али Аскари, Харта Виджая, Хун Даи Тана, Ко Цзы Чуна, Ли Ин Конга, Питера Панди, Раймонда Хенди Сьюзанто, Сима Венлунга Рассела, Тан Хианг Тата, Тран Конг Хоанг, Юань Юаня и еще одного студента, который предпочел остаться анонимным;

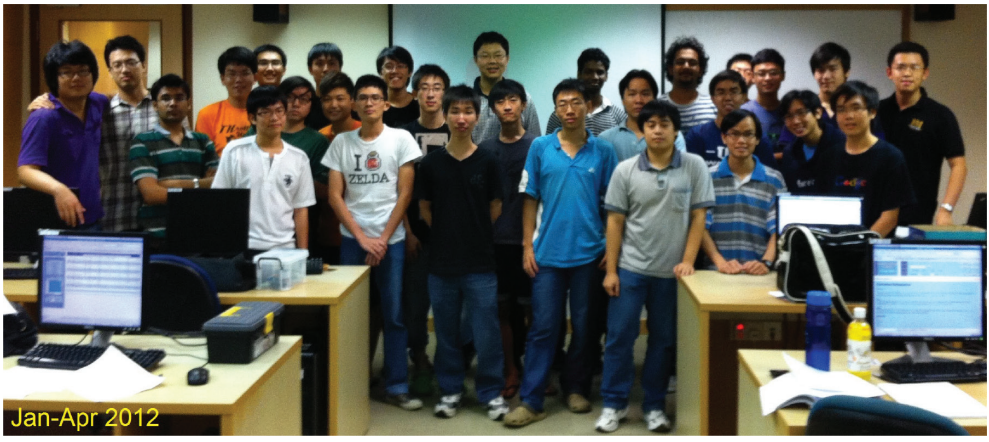


- читателей корректуры: семерых студентов, обучавшихся на курсе CS3233 (выше их имена подчеркнуты), и Тай Вэньбинь;
- и последнее, но не менее важное: я хочу поблагодарить мою жену, Грейс Сурьяни, за то, что она позволила мне провести время за работой над очередным изданием книги, когда она была беременна нашим первым ребенком, Джейн Анджелиной Халим.

Благодарности к третьему изданию

От Стивена: еще раз я хочу поблагодарить:

- 2000 покупателей второго издания (по состоянию на 24 мая 2013 года). Спасибо :);
- помощников преподавателя курса CS3233 Национального университета Сингапура, участвовавших в проведении курса за последние два года: Харта Виджая, Тринь Туан Фуонг и Хуан Да;
- моих студентов, обучавшихся на курсе CS3233 во втором семестре 2011/2012 учебного года, которые участвовали в технической подготовке и презентации второго издания (далее их фамилии приведены в алфавитном порядке): Цао Шэна, Чуа Вэй Куан, Хань Юй, Хуан Да, Хуинь Нгок Тай, Ивана Рейнальдо, Джона Го Чу Эрна, Ле Вьет Тьена, Лим Чжи Цинь, Налина Иланго, Нгуен Хоанг Дуй, Нгуен Фи Лонга, Нгуен Куок Фонг, Паллава Шингаля, Пан Чжэнъяна, Пан Ян Хана, Сун Янью, Тан Ченг Йонг Десмонд, Тэй Вэньбинь, Ян Маньшена, Чжао Яна, Чжоу Имина и двух других студентов, которые предпочли остаться анонимными;



- читателей корректуры: шестерых студентов, обучавшихся на курсе CS3233 во втором семестре 2011/2012 учебного года (выше их имена подчеркнуты), и Хьюберта Тео Хуа Киана;
- моих студентов, обучавшихся на курсе CS3233 во втором семестре 2012/2013 учебного года, которые участвовали в технической подготовке и презентации второго издания (далее их фамилии приведены в алфа-

витном порядке): Арнольда Кристофера Короа, Цао Луу Куанг, Лим Пуай Линг Полин, Эрика Александра Квик Факсаа, Джонатана Дэррила Виджаи, Нгуен Тан Сы Нгуен, Нгуен Чыонг Дуй, Онг Мин Хуэй, Пан Юйсюань, Шубхам Гоял, Судханшу Хемку, Тан Бинбин, Трин Нгок Кхана, Яо Юцзянь, Чжао Юэ и Чжэн Найцзя;

- центр развития преподавания и обучения (CDTL) Национального университета Сингапура за предоставление начального финансирования для создания веб-сайта визуализации алгоритмов;
- мою жену Грейс Сурьяни и мою дочь Джейн Анджелину за их любовь.

Стивен и Феликс Халим
Сингапур, 24 мая 2013 г.

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги. Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Скачивание исходного кода

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» и авторы книги очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторах этой книги

Стивен Халим, PhD[1]
stevenhalim@gmail.com

Стивен Халим в настоящее время преподает в Школе программирования Национального университета Сингапура (SoC, NUS). Он ведет несколько курсов по программированию в университете Сингапура, начиная с основ методологии программирования, далее переходя к курсам среднего уровня, отноящимся к структурам данных и алгоритмам, а также курс «Олимпиадное программирование», в качестве учебного материала к которому используется эта книга. Стивен является наставником команд Национального университета Сингапура, участвующих в соревнованиях по программированию ACM ICPC, а также наставником сингапурской команды IOI. Будучи студентом, участвовал в нескольких региональных состязаниях ACM ICPC (Сингапур 2001, Айзу 2003, Шанхай 2004). На сегодняшний день он и другие наставники Национального университета Сингапура успешно подготовили две команды финалистов соревнований мирового уровня ACM ICPC World (2009–2010; 2012–2013). Также их студенты завоевали две золотые, шесть серебряных и семь бронзовых наград на олимпиаде IOI (2009–2012).

Стивен женат на Грейс Сурьяни Тиозо; супруги воспитывают дочь, Джейн Анджелину Халим.

Феликс Халим, PhD[2]
felix.halim@gmail.com

Феликс Халим получил степень PhD в Школе программирования Национального университета Сингапура (SoC, NUS). С точки зрения соревнований по программированию, у Феликса гораздо более яркая репутация, чем у его старшего брата. Он был участником IOI 2002 (представлял Индонезию). Команды ICPC, в которых он участвовал (в то время он

---

[1] Кандидатская диссертация на тему: «An Integrated White+Black Box Approach for Designing and Tuning Stochastic Local Search Algorithms» (Интегрированный подход «белого + черного ящика» для разработки и усовершенствования алгоритмов стохастического локального поиска), 2009.

[2] Кандидатская диссертация на тему: «Solving Big Data Problems: from Sequences to Tables and Graphs» (Решение задач, использующих большие данные: от последовательностей к таблицам и графам), 2012.

был участником соревнований от университета Бина Нусантара), заняли на региональных соревнованиях в Маниле (ACM ICPC Manila Regional 2003–2004–2005) 10-е, 6-е и 10-е места соответственно. Затем в последний год его команда, наконец, выиграла гаосюнские региональные соревнования (ACM ICPC Kaohsiung Regional 2006) и стала финалистом соревнований мирового уровня в Токио (ACM ICPC World Tokyo 2007), заняв 44-е место. По окончании карьеры в ICPC продолжил участвовать в соревнованиях TopCoder. В настоящий момент Феликс Халим работает в компании Google, г. Маунтин-Вью, США.

Список сокращений

A*: A со звездочкой
ACM: Assoc of Computing Machinery – Ассоциация вычислительной техники
AC: Accepted – принято
APSP: All-Pairs Shortest Paths – кратчайшие расстояния между всеми вершинами
AVL: Adelson-Velskii Landis (BST) – АВЛ-дерево (алгоритм Адельсон-Вельского и Ландиса)
BNF: Backus Naur Form – форма Бэкуса–Наура
BFS: Breadth First Search – поиск в ширину
BI: Big Integer – большое целое
BIT: Binary Indexed Tree – двоичное индексированное дерево
BST: Binary Search Tree – дерево двоичного поиска
CC: Coin Change – размен монет
CCW: Counter ClockWise – против часовой стрелки
CF: Cumulative Frequency – накопленная частота
CH: Convex Hull – выпуклая оболочка
CS: Computer Science – компьютерные науки
CW: ClockWise – по часовой стрелке
DAG: Directed Acyclic Graph – направленный ациклический граф
DAT: Direct Addressing Table – таблица с прямой адресацией
D&C: Divide and Conquer – «разделяй и властвуй»
DFS: Depth First Search – поиск в глубину
DLS: Depth Limited Search – поиск с ограничением глубины
DP: Dynamic Programming – динамическое программирование
DS: Data Structure – структура данных
ED: Edit Distance – расстояние редактирования
FIFO: First In First Out – принцип FIFO («первым пришел – первым ушел»)
FT: Fenwick Tree – дерево Фенвика
GCD: Greatest Common Divisor – НОД (наибольший общий делитель)
ICPC: Intl Collegiate Prog Contest – Международная студенческая олимпиада по программированию
IDS: Iterative Deepening Search – поиск с итеративным углублением
IDA*: Iterative Deepening A Star – итеративное углубление A*
IOI: Intl Olympiad in Informatics – международные олимпиады по информатике
IPSC: Internet Problem Solving Contest – интернет-конкурс по решению задач
LA : Live Archive [33]
LCA: Lowest Common Ancestor – самый низкий общий предок
LCM: Least Common Multiple – наименьшее общее кратное; НОК
LCP: Longest Common Prefix – наибольший общий префикс
LCS1: Longest Common Subsequence – наибольшая общая подпоследовательность

LCS2: Longest Common Substring – наибольшая общая подстрока
LIFO: Last In First Out – принцип LIFO («последним пришел – первым ушел»)
LIS: Longest Increasing Subsequence – наибольшая возрастающая подпоследовательность
LRS: Longest Repeated Substring – самая длинная повторяющаяся подстрока
LSB: Least Significant Bit – наименьший значащий бит
MCBM: Max Cardinality Bip Matching – максимальное по мощности паросочетание на двудольном графе
MCM: Matrix Chain Multiplication – умножение матричной цепи
MCMF: Min-Cost Max-Flow – максимальный поток минимальной стоимости
MIS: Maximum Independent Set – максимальное независимое множество
MLE: Memory Limit Exceeded – ошибка превышения лимита памяти
MPC: Minimum Path Cover – минимальное покрытие путями
MSB: Most Significant Bit – самый старший двоичный разряд
MSSP: Multi-Sources Shortest Paths – задача о кратчайших путях из заданных вершин во все
MST: Minimum Spanning Tree – минимальное остовное дерево
MWIS: Max Weighted Independent Set – независимое множество с максимальным весом
MVC: Minimum Vertex Cover – минимальное вершинное покрытие
OJ: Online Judge – «Онлайн-арбитр»
PE: Presentation Error – ошибка представления
RB: Red-Black (BST) – черно-красное (дерево двоичного поиска)
RMQ: Range Min (or Max) Query – запрос минимального (максимального) значения из диапазона
RSQ: Range Sum Query – запрос суммы диапазона
RTE: Run Time Error – ошибка времени выполнения
SSSP: Single-Source Shortest Paths – кратчайшие пути из заданной вершины во все остальные
SA: Suffix Array – массив суфиксов
SCC: Strongly Connected Component – компонент сильной связности
SPOJ: Sphere Online Judge – «Онлайн-арбитр» в Sphere
ST: Suffix Tree – дерево суффиксов
STL: Standard Template Library – стандартная библиотека шаблонов
TLE: Time Limit Exceeded – превышение лимита времени
USACO: USA Computing Olympiad – олимпиада по программированию в США
UVa: University of Valladolid [47] – университет Вальядолида
WA: Wrong Answer – неверный ответ
WF: World Finals – финальные соревнования на кубок мира

Глава 1

Введение

Я хочу участвовать в финальных соревнованиях на кубок мира ACM ICPC!

– Прилежный студент

1.1. Олимпиадное программирование

Основная идея в спортивном программировании такова: «Вам дают известные задачи в области компьютерных наук, решайте их как можно быстрее!»

Давайте прочитаем эту фразу более внимательно, обращая внимание на термины. Термин «известные задачи в области компьютерных наук» подразумевает, что в олимпиадном программировании мы имеем дело с *уже решенными* задачами, а не с задачами, требующими исследовательского подхода (где до сих пор нет решений). Эти задачи уже были решены ранее (по крайней мере, автором задач). «Решать их» подразумевает, что мы[1] должны углубить наши знания в области компьютерных наук. Мы должны достичь необходимого уровня, позволяющего создавать работающий код, который решает эти задачи, по крайней мере мы должны получить в установленные сроки тот же результат, что и автор задачи, на неизвестных нам тестовых данных, подготовленных разработчиком задачи[2]. Необходимость решить задачу «как можно быстрее» – это одна из целей подобных соревнований: ведь соревнования в скорости – в природе человека.

---

[1] Некоторые соревнования по программированию проводятся в командах, чтобы стимулировать командную работу, ведь инженеры-программисты обычно не работают в одиночку в реальной жизни.

[2] Публикуя постановку задачи, но не фактические тестовые данные для нее, олимпиадное программирование побуждает соревнующихся в решении задач проявлять креативность и аналитическое мышление, чтобы обдумать все возможные варианты решения задачи и протестировать свой код. В реальной жизни разработчикам программного обеспечения приходится изобретать тесты и много раз тестировать свое программное обеспечение, чтобы убедиться, что оно соответствует требованиям клиентов.

Пример. Архив задач университета Вальядолида [47], задача № 10 911 (Формирование команды для викторины).

Краткое описание задачи:
Пусть (x, y) – координаты дома, в котором живет студент, на двухмерной плоскости. Есть $2N$ студентов, которых мы хотим объединить в N групп. Пусть d_i – расстояние между домами двух студентов в группе i.
Формируем N групп таким образом, чтобы значение cost = $\sum_{i=1}^{N} d_i$ было минимальным.
Выведите минимальное значение cost. Ограничения: $1 \leqslant N \leqslant 8$ и $0 \leqslant x, y \leqslant 1000$.

Пример входных данных:
$N = 2$; Координаты $2N = 4$ дома: {1, 1}, {8, 6}, {6, 8} и {1, 3}.

Пример выходных данных:
cost = 4,83.

Сможете ли вы решить эту задачу?
Если да, сколько времени вам потребуется, чтобы написать работающий код?
Подумайте и постарайтесь не переворачивать эту страницу сию же секунду!

Рис. 1.1 ❖ Иллюстрация к задаче UVa 10911 – Forming Quiz Teams

Теперь спросите себя: к какому из описанных ниже программистов вы можете отнести себя? Обратите внимание, что если вам непонятен материал или терминология из этой главы, вы можете повторно вернуться к нему, прочитав эту книгу еще один раз.

- Неконкурентоспособный программист A (начинающий программист)
  Шаг 1: читает условия задачи и не понимает ее. (Эта задача для него новая.)
  Шаг 2: пытается написать какой-то код, в отчаянии смотрит на непонятные входные и выходные данные.
  Шаг 3: понимает, что все его попытки решения не зачтены (AC не получено), использует «жадный» алгоритм (см. раздел 3.4). Попарное

объединение студентов, чьи дома находятся на кратчайшем расстоянии между ними, дает неверный ответ (WA).
Предпринимает наивную попытку использовать полный перебор: использует возвратную рекурсию (см. раздел 3.2) и комбинации из всех возможных пар, что приводит его к превышению лимита времени (TLE).

- Неконкурентоспособный программист B (сдается)
  Шаг 1: читает условия задачи и вспоминает, что он где-то читал об этом раньше.
  Но также он помнит, что не научился решать такие задачи…
  Он не знает о решении, относящемся к области динамического программирования (DP) (см. раздел 3.5).
  Шаг 2: пропускает эту задачу и читает другую из списка задач, предложенных на соревнованиях.
- Все еще неконкурентоспособный программист C (решает задачу медленно)
  Шаг 1: читает условия задачи и понимает, что это сложная задача: поиск идеального соответствия с минимальным весом на небольшом произвольном взвешенном графе. Однако, поскольку объем файла входных данных невелик, эту задачу можно решить средствами динамического программирования (DP). Состояние в динамическом программировании – это **битовая маска**, описывающая состояние соответствия, и при сопоставлении не соответствующих условию проживания на минимальном расстоянии студентов *i* и *j* включаются два бита *i* и *j* битовой маски (раздел 8.3.1).
  Шаг 2: пишет код процедуры ввода-вывода, использует нисходящий рекурсивный метод динамического программирования, пишет тесты, отлаживает программу…
  Шаг 3: через 3 часа его решение получает оценку «зачтено» (AC) – оно проходит все тесты с использованием секретного набора тестовых данных.
- Конкурентоспособный программист D
  Завершает все шаги, предпринятые неконкурентоспособным программистом C, за время, не превышающее 30 минут.
- Высококонкурентоспособный программист E
  Высококонкурентоспособный конкурентоспособный программист (например, обладатель «красного рейтинга» на сайте TopCoder [32]) решит эту «хорошо известную» задачу за 15 минут.

Мы хотим обратить ваше внимание на то, что успехи в спортивном программировании – не конечная цель, а лишь средство достижения цели. Истинная конечная цель состоит в том, чтобы подготовить ученых в области информатики и программистов, которые создают более качественное программное обеспечение и ставят перед собой очень серьезные исследовательские задачи. Учредители студенческого командного чемпионата мира по программированию (ICPC) [66] придерживаются этой концепции, и мы, авторы, солидарны с ними.

Написав эту книгу, мы внесли свой вклад в подготовку нынешнего и будущих поколений к тому, чтобы стать более конкурентоспособными в решении хорошо известных задач из области программирования, часто предлагаемых в заданиях последних состязаний ICPC и на Международной олимпиаде по информатике (IOI).

Упражнение 1.1.1. Описанная выше стратегия неконкурентоспособного программиста A, использующего «жадные» алгоритмы, дает правильный результат для тестового примера, показанного на рис. 1.1. Пожалуйста, приведите *лучший* контрпример.

Упражнение 1.1.2. Проанализируйте временную сложность решения задачи методом полного перебора, представленным неконкурентоспособным программистом A и приведенным выше, чтобы понять, почему оно получает вердикт TLE.

Упражнение 1.1.3*. На самом деле продуманное решение, использующее возвратную рекурсию, *с ограничением*, все же поможет решить эту задачу. Решите эту задачу без использования таблицы динамического программирования.

1.2. Как стать конкурентоспособным

Если вы хотите стать похожими на конкурентоспособных программистов D или E из приведенного выше примера – то есть если вы хотите, чтобы вас выбрали (обычно отбор происходит на уровне стран, отобранные кандидаты становятся членами сборной команды) для участия в соревнованиях IOI [34], или же вы хотите стать одним из членов команды, представляющей ваш университет в ICPC [66] (соревнования первого и второго уровней проводятся внутри страны, далее идут региональные и мировые турниры), либо вы желаете отличиться в других соревнованиях по программированию – то эта книга определенно для вас!

В последующих главах вы узнаете все о задачах, предполагающих использование структур данных и алгоритмов – начиная с элементарных задач, переходя к задачам средней сложности и завершая сложными задачами[1], которые часто предлагались в недавно прошедших олимпиадах по программированию. Эти задачи были собраны из многих источников [50, 9, 56, 7, 40, 58, 42, 60, 1, 38, 8, 59, 41, 62, 46] (см. рис. 1.4). Вы не только получите представление о структурах данных и алгоритмах, но и узнаете, как их эффективно реализовать и применять в решении олимпиадных задач. Кроме того, книга дает вам много советов по программированию, основанных на нашем собственном опыте, который может быть полезен при участии в соревнованиях по программированию. Мы начнем эту книгу с нескольких общих советов.

---

[1] Восприятие материала, представленного в этой книге, зависит от вашей подготовки. Найдете ли вы представленный в ней материал сложным или среднего уровня сложности, зависит от ваших навыков программирования до прочтения этой книги.

1.2.1. Совет 1: печатайте быстрее!

Это не шутка! Хотя этому совету можно не придавать большого значения, поскольку соревнования ICPC и особенно IOI – это не соревнования в скорости печати, мы часто оказывались свидетелями ситуаций, когда команды ICPC, которые опускались на более низкие строчки в таблице рейтинга соревнований, представляли правильное решение всего лишь на несколько минут позже своих соперников; мы видели расстроенных участников соревнований IOI, которые упускали возможность получить баллы за правильно решенные задачи, просто не успев дописать код при решении задач методом перебора. Если вы и ваши соперники решаете одинаковое количество задач за отведенное время, это происходит благодаря вашим навыкам программирования (способности создавать лаконичный и надежный код) и… скорости, с которой вы набираете текст.

Попробуйте пройти тест на сайте http://www.typingtest.com – следуйте инструкциям, опубликованным на этом сайте. У Стивена скорость набора текста – около 85–95 знаков в минуту, а у Феликса – около 55–65 знаков в минуту. Если ваша скорость печати намного меньше этих цифр, отнеситесь к нашему первому совету серьезно!

Помимо возможности быстрого и правильного ввода букв и цифр, вам также необходимо ознакомиться с положением символов, часто используемых в языках программирования, таких как круглые скобки (), фигурные скобки {}, квадратные скобки [] и угловые скобки <>, точка с запятой ; и двоеточие :, одинарные кавычки ' ', используемые для обозначения символов, двойные кавычки " ", используемые для обозначения строк, амперсанд &, вертикальная черта (или «пайп») |, восклицательный знак ! и т. д.

В качестве небольшого упражнения попробуйте набрать код на C++, приведенный ниже, как можно быстрее.

```
#include <algorithm>                  // если вы не понимаете, как работает этот код на C++,
#include <cmath>                       // сначала прочитайте учебники по программированию...
#include <cstdio>
#include <cstring>
using namespace std;
          /* Формирование команд; ниже представлено решение задачи UVa 10911 */
                  // использование глобальных переменных – плохая практика при разработке
                                                                    // программного обеспечения,
int N, target;                       // но вполне подходит для олимпиадного программирования
double dist[20][20], memo[1 << 16];              // 1 << 16 = 2^16, заметим, что max N = 8

double matching(int bitmask) {                                      // состояние DP = bitmask
                                    // инициализируем 'memo' значением -1 в функции main
  if (memo[bitmask] > -0.5)                          // это состояние было вычислено ранее
    return memo[bitmask];                                          // просмотр таблицы memo
  if (bitmask == target)                             // все студенты уже разбиты на группы
    return memo[bitmask] = 0;                                      // значение cost равно 0

  double ans = 2000000000.0;                 // инициализируем переменную большим значением
```

```
  int p1, p2;
  for (p1 = 0; p1 < 2 * N; p1++)
    if (!(bitmask & (1 << p1)))
      break;                                          // найдем первый выключенный бит
  for (p2 = p1 + 1; p2 < 2 * N; p2++)                 // затем пытаемся сопоставить p1
    if (!(bitmask & (1 << p2)))                     // с другим битом p2, также выключенным
      ans = min(ans,                                  // вычисляем минимальное значение
                dist[p1][p2] + matching(bitmask | (1 << p1) | (1 << p2)));

  return memo[bitmask] = ans;                       // сохраняем результат в таблице memo
}

int main() {
  int i, j, caseNo = 1, x[20], y[20];
  // freopen("10911.txt", "r", stdin);                // подаем файл со входными данными,
                                                     // на стандартный ввод
  while (scanf("%d", &N), N) {                       // да, мы так можем :)
    for (i = 0; i < 2 * N; i++)
      scanf("%*s %d %d", &x[i], &y[i]);                        // '%*s' пропускает имена
    for (i = 0; i < 2 * N - 1; i++)                 // строим таблицу попарных расстояний
      for (j = i + 1; j < 2 * N; j++)               // вы когда-нибудь использовали 'hypot'?
        dist[i][j] = dist[j][i] = hypot(x[i] - x[j], y[i] - y[j]);

    // использование динамического программирования для поиска
    // идеального соответствия с минимальным весом
    // на небольшом произвольном взвешенном графе
    for (i = 0; i < (1 << 16); i++) memo[i] = -1.0;    // задаем значение -1 для всех ячеек
    target = (1 << (2 * N)) - 1;
    printf("Случай %d: %.2lf\n", caseNo++, matching(0));
} } // возвращаем 0;
```

Объяснение этого решения «динамическое программирование на битовых масках» приведено в разделах 2.2, 3.5 и 8.3.1. Не пугайтесь, если вы еще не до конца поняли, как решается данная задача.

1.2.2. Совет 2: быстро классифицируйте задачи

В ICPC участникам (командам) предоставляется несколько задач (обычно 7–14 задач) разных типов. По нашим наблюдениям за тем, как проходили соревнования ICPC в странах Азиатского региона, мы можем разбить задачи на категории и примерно определить, какой процент от общего числа задач составляют задачи каждой категории (см. табл. 1.1).

В IOI участникам дается 6 заданий, которые они должны решить в течение двух туров, по три задачи на каждый тур (на соревнованиях 2009–2010 гг. давалось 8 заданий, которые нужно было решить в течение двух дней). Эти задачи в основном относятся к строкам 1–5 и 10 в табл. 1.1; число задач, относящихся к строкам 6–10 в табл. 1.1, в олимпиаде IOI значительно меньше. Более подробно содержание задач IOI раскрыто в опубликованной программе IOI 2009 года [20] и классификации задач IOI за 1989–2008 гг. [67].

Таблица 1.1. Классификация задач, предложенных на последних олимпиадах по программированию ACM ICPC (Азия)

№	Класс задач	В этой книге	Число предложенных задач
1	AdHoc	Раздел 1.4	1–2
2	Полный перебор (итеративный/рекурсивный)	Раздел 3.2	1–2
3	«Разделяй и властвуй»	Раздел 3.3	0–1
4	«Жадные» алгоритмы (как правило, оригинальные)	Раздел 3.4	0–1
5	Динамическое программирование (как правило, оригинальные)	Раздел 3.5	1–3
6	Графы	Глава 4	1–2
7	Математические задачи	Глава 5	1–2
8	Обработка строк	Глава 6	1
9	Вычислительная геометрия	Глава 7	1
10	Сложные/редкие темы	Главы 8–9	1–2
Всего			8–17 (≈⩽ 14)

Классификация, приведенная в табл. 1.1, взята из [48] и никак не претендует на полноту и завершенность. Некоторые методы (например, сортировка) исключены из классификации как «тривиальные»; обычно они используются только в качестве «подпрограмм» в более серьезных задачах. Мы также не включаем в классификацию раздел «рекурсия», так как она входит в такие категории задач, как возвратная рекурсия или динамическое программирование. Мы также опускаем раздел «структуры данных», поскольку использование эффективной структуры данных можно считать неотъемлемой частью решения более сложных задач. Конечно, решение задач не всегда может быть сведено к единственному методу: задача может быть разделена на несколько подзадач, относящихся к разным классам. Например, алгоритм Флойда–Уоршелла можно классифицировать и как решение задачи нахождения кратчайших расстояний между всеми вершинами графа (см. раздел 4.5), и как решение задачи с использованием алгоритмов динамического программирования (см. раздел 3.5). Алгоритмы Прима и Краскала являются решением задачи построения минимального остовного дерева (см. раздел 4.3) и в то же время могут быть отнесены к «жадным» алгоритмам (см. раздел 3.4). В разделе 8.4 мы рассмотрим более сложные задачи, для решения которых требуется использовать несколько алгоритмов и/или структур данных.

В будущем эта классификация может измениться. Возьмем, к примеру, динамическое программирование. Этот метод не был известен до 1940-х годов и не входил в программы ICPC и IOI до середины 1990-х годов, но в настоящее время это один из обязательных видов задач на олимпиадах по программированию. Так, например, в финале чемпионата мира ICPC 2010 было предложено более трех задач, относящихся к этому классу (из 11 предлагавшихся для решения).

Однако наша главная цель – не просто сопоставить задачи и методы, необходимые для их решения, как это сделано в табл. 1.1. Ознакомившись с большин-

ством тем, изложенных в этой книге, вы сможете распределять задачи, относя их к одному из трех типов, как показано в табл. 1.2.

Таблица 1.2. Категории задач (в более компактной форме)

№	Категория задач	Уверенность и ожидаемая скорость решения
A	Я решал задачи такого типа	Я уверен, что смогу решить такую задачу (быстро)
B	Я где-то встречал такую задачу	Но я знаю, что пока не могу решить ее
C	Я никогда не встречал задач такого типа	См. обсуждение ниже

Чтобы быть *конкурентоспособным*, то есть занимать высокие места на олимпиадах по программированию, вы должны отнести абсолютное большинство предложенных задач к типу A и минимизировать количество задач, которые вы относите к типу B. Вам необходимо приобрести прочные знания в области алгоритмов и совершенствовать свои навыки программирования, чтобы легко решать многие классические задачи. Однако, чтобы выиграть олимпиаду по программированию, вам также необходимо совершенствовать и оттачивать навыки решения задач (например, умение сводить решение предложенной задачи к решению известных задач, видеть тонкости и «хитрости» в задаче, использовать нестандартные подходы к решению задач и т. д.) – так, чтобы вы (или ваша команда) смогли найти решение для сложных, нетривиальных задач типа C, участвуя в региональных и всемирных олимпиадах IOI или ICPC.

Таблица 1.3. Упражнение: классифицируйте эти задачи с сайта университета Вальядолида (UVa)

№ UVa	Название	Категория задачи	Подсказка
10360	Rat Attack	Полный перебор или динамическое программирование	Раздел 3.2
10341	Solve It		Раздел 3.3
11292	Dragon of Loowater		Раздел 3.4
11450	Wedding Shopping		Раздел 3.5
10911	Forming Quiz Teams	Динамическое программирование с использованием битовой маски	Раздел 8.3.1
11635	Hotel Booking		Раздел 8.4
11506	Angry Programmer		Раздел 4.6
10243	Fire! Fire!! Fire!!!		Раздел 4.7.1
10717	Mint		Раздел 8.4
11512	GATTACA		Раздел 6.6
10065	Useless Tile Packers		Раздел 7.3.7

Упражнение 1.2.1. Прочитайте условия задач с сайта университета Вальядолида [47], перечисленных в табл. 1.3, и определите, к каким категориям они относятся (для двух задач в данной таблице мы уже проделали это упражнение). После прочтения нашей книги вам будет легко выполнить такое упражнение – в книге обсуждаются все методы, необходимые для решения этих задач.

1.2.3. Совет 3: проводите анализ алгоритмов

После того как вы разработали алгоритм для решения конкретной задачи на олимпиаде по программированию, вы должны задать себе вопрос: учитывая ограничения по объему входных данных (обычно указанные в содержательной постановке задачи), может ли разработанный алгоритм с его временной сложностью и пространственной сложностью (сложностью по памяти) уложиться в отведенный лимит времени и памяти, указанный для этой задачи?

Иногда существует несколько способов решения задачи. Некоторые подходы могут оказаться неверными, другие – недостаточно быстрыми, а третьи могут выйти за пределы разумной оптимизации. Хорошая стратегия – провести мозговой штурм, перебирая множество подходящих алгоритмов, а затем выбрать простейшее решение, которое работает (то есть работает достаточно быстро, чтобы преодолеть ограничение по времени и памяти и при этом все же дать правильный ответ)[1].

Современные компьютеры достаточно мощны и могут выполнять[2] до 100 млн (или 10^8; 1 млн = 1 000 000) операций за несколько секунд. Вы можете использовать эту информацию, чтобы определить, уложится ли ваш алгоритм в отведенное время. Например, если максимальный размер входных данных n равен 100 КБ (или 10^5; 1 КБ = 1000), а ваш текущий алгоритм имеет временную сложность $O(n^2)$, то простейшие вычисления покажут, что $(100 \text{ КБ})^2$ или 10^{10} – это очень большое число, и ваш алгоритм будет отрабатывать за время порядка сотни секунд. Таким образом, вам нужно будет разработать более быстрый (и при этом правильный) алгоритм для решения задачи. Предположим, вы нашли тот, который работает с временной сложностью $O(n \log_2 n)$. Теперь расчеты показывают, что $10^5 \log_2 10^5$ – это всего лишь $1{,}7 \times 10^6$, и здравый смысл подсказывает, что алгоритм (который теперь должен работать менее чем за секунду), скорее всего, работает достаточно быстро, чтобы уложиться во временные ограничения.

При определении того, подходит ли ваше решение, ограничения задачи играют не меньшую роль, чем временная сложность вашего алгоритма. Предположим, что вы можете разработать только относительно простой алгоритм, для которого легко написать код, но который работает с кошмарной временной сложностью $O(n^4)$. Это может показаться неподходящим решением, но если $n \leqslant 50$, то вы действительно решили задачу. Вы можете реализовать свой алгоритм $O(n^4)$, поскольку 50^4 составляет всего 6,25 млн, и ваш алгоритм должен отрабатывать примерно за секунду.

Обратите внимание, однако, что порядок сложности не обязательно указывает на фактическое количество операций, которые будет выполнять ваш ал-

---

[1] **Обсуждение:** это действительно так – на олимпиадах по программированию выбор простейшего алгоритма, который работает, крайне важен для успеха в соревнованиях. Тем не менее на занятиях в аудитории, где нет жестких временных ограничений, полезно потратить больше времени на решение определенной задачи с использованием наилучшего алгоритма. Поступая таким образом, мы повышаем уровень своей подготовки. Если в будущем мы столкнемся с более сложной версией задачи, у нас будет больше шансов получить и реализовать правильное решение!

[2] Используйте приведенную здесь цифру для приблизительных расчетов. Значение производительности, разумеется, будет различаться для разных компьютеров.

горитм. Если каждая итерация включает в себя большое количество операций (много вычислений с плавающей точкой или значительное число итераций циклов), или же если ваша реализация имеет высокие «накладные расходы» при выполнении (множество повторяющихся циклов, несколько проходов или даже высокие затраты на операции ввода-вывода либо исполнение программы), ваш код может работать медленнее, чем ожидалось. Однако обычно это не представляет серьезной проблемы, поскольку авторы задачи устанавливают ограничения по времени таким образом, чтобы хорошо написанный код, реализующий алгоритм с подходящей сложностью по времени, получил оценку «зачтено» (AC).

Анализируя сложность вашего алгоритма с учетом определенного размера входных данных и указанных ограничений по времени и памяти, вы можете выбрать правильную стратегию: решить, следует ли вам пытаться реализовать свой алгоритм (что отнимет драгоценное время в соревнованиях ICPC и IOI), попытаться улучшить свой алгоритм или же переключиться на другие задачи, предложенные на соревнованиях.

Как уже упоминалось в предисловии к этой книге, мы не будем подробно обсуждать концепцию алгоритмического анализа. Мы предполагаем, что у вас уже есть этот элементарный навык. Существует множество справочников и книг (например, «Введение в алгоритмы» («Introduction to Algorithms») [7], «Разработка алгоритмов» («Algorithm Design») [38], «Алгоритмы» («Algorithms») [8], и т. д.), которые помогут вам понять следующие обязательные понятия/методы в алгоритмическом анализе:

- базовый анализ сложности по времени и по памяти для итерационных и рекурсивных алгоритмов:
  - алгоритм с k вложенными циклами, состоящими примерно из n итераций, имеет сложность по времени $O(n^k)$;
  - если у вас имеется рекурсивный алгоритм с b рекурсивными вызовами на уровень и глубина рекурсии составляет L уровней, то сложность такого алгоритма будет приблизительно $O(b^L)$, однако подобная оценка – лишь грубая верхняя граница. Фактическая сложность алгоритма будет зависеть от того, какие действия выполняются на каждом уровне и возможно ли упрощение;
  - алгоритм динамического программирования или другая итерационная процедура, которая обрабатывает двумерную матрицу $n \times n$, затрачивая $O(k)$ на ячейку, отрабатывает за время $O(k \times n^2)$. Это более подробно объясняется в разделе 3.5;
- более продвинутые методы анализа:
  - докажите правильность алгоритма (это особенно важно для «жадных» алгоритмов, о которых идет речь в разделе 3.4), чтобы свести к минимуму вероятность получения оценки тестирующей системы «Неправильный ответ» (WA) для вашей задачи;
  - выполните амортизационный анализ (в качестве примера см. главу 17 в [7]). Амортизационный анализ, хотя и редко применяется на олимпиадах по программированию, позволяет свести к минимуму вероятность получения оценки тестирующей системы «Превышение лимита времени» (TLE) или, что еще хуже, случаи, когда вы отвергаете

свой алгоритм из-за того, что он слишком медленный, и бросаете решение этой задачи, переключаясь на другую, хотя на самом деле ваш алгоритм работает достаточно быстро;
- выполните анализ алгоритма на основе выходных данных, поскольку время работы алгоритма также зависит от размера выходных данных, – это снизит вероятность того, что тестирующая система выдаст вердикт «Превышение лимита времени» (TLE) для вашего решения. Например, алгоритм поиска строки длиной m в длинной строке с помощью дерева суффиксов (которое уже построено) будет работать за время $O(m+occ)$. Время выполнения этого алгоритма зависит не только от размера входных данных m, но и от размера выходых данных – количества вхождений occ (более подробно об этом рассказывается в разделе 6.6);

❍ знание следующих ограничений:
- $2^{10} = 1024 \approx 10^3$, $2^{20} = 1\,048\,576 \approx 10^6$;
- 32-разрядные целые числа со знаком (`int`) и 64-разрядные целые числа со знаком (`long long`) имеют верхние пределы $2^{31} - 1 \approx 2\times10^9$ (что позволяет работать с числами, ограничивающимися приблизительно 9 десятичными разрядами) и $2^{63} - 1 \approx 9\times10^{18}$ (что позволяет работать с числами, ограничивающимися приблизительно 18 десятичными разрядами) соответственно;
- беззнаковые целые можно использовать, если требуются только неотрицательные числа. 32-разрядные целые числа без знака (`unsigned int`) и 64-разрядные целые числа без знака (`unsigned long long`) имеют верхние пределы $2^{32} - 1 \approx 4\times10^9$ и $2^{64} - 1 \approx 1{,}8\times10^{19}$ соответственно;
- если вам нужно хранить целые числа, значения которых превосходят 2^{64}, используйте методы работы с большими числами (см. раздел 5.3);
- число перестановок для множества из n элементов равно $n!$, число комбинаций для такого множества равно 2^n;
- наилучшая сложность по времени алгоритма сортировки на основе сравнения составляет $\Omega(n\log_2 n)$;
- обычно сложность по времени алгоритмов $O(n\log_2 n)$ вполне приемлема для решения большинства олимпиадных задач;
- наибольший размер входных данных для типичных задач олимпиады по программированию не должен превышать 1 млн. Кроме того, нужно принимать во внимание, что «узким местом» будет время чтения входных данных (процедура ввода-вывода);
- среднестатистический процессор, выпущенный в 2013 году, выполняет около 100 млн = 10^8 операций за несколько секунд.

Многие начинающие программисты пропускают этот этап и сразу же начинают реализовывать первый (наивный) алгоритм, пришедший им в голову, впоследствии убеждаясь, что выбранная структура данных или алгоритм недостаточно эффективны (или неверны). Наш совет всем участникам ICPC[1]: воз-

[1] В отличие от задач, предлагаемых на олимпиадах ICPC, задачи, которые решаются на IOI, обычно имеют нескольких возможных решений (частичных или полных), каждое из которых имеет различную временную сложность и оценивается таким

держитесь от написания кода, пока не убедитесь, что ваш алгоритм работает правильно и достаточно быстро.

Чтобы дать вам некоторые общие ориентиры временной сложности различных алгоритмов и помочь определить, что означает «достаточно быстро» в вашем случае, мы привели некоторые примеры в табл. 1.4. Подобные цифры также можно найти во многих других книгах по структурам данных и алгоритмам. Эта таблица составлена *участником олимпиады по программированию* с учетом контекста и специфики решения задач. Обычно ограничения размера входных данных приводятся в содержательной постановке задачи. Предполагая, что типичный процессор может выполнить 100 млн операций примерно за 3 секунды (что является стандартным ограничением по времени в большинстве задач, опубликованных на сайте университета Вальядолида (UVa) [47]), мы можем определить «худший» алгоритм, который все еще может уложиться в эти пределы времени. Обычно самый простой алгоритм имеет наихудшую временную сложность, но если он работает достаточно быстро, чтобы уложиться в отведенное время, просто используйте его!

Таблица 1.4. Эмпирическое правило определения наихудшей сложости по времени для алгоритма, получающего оценку жюри «зачтено» (AC), при различных объемах входных данных n при прохождении одного и того же теста (при условии что ваш процессор позволяет выполнять 100 млн операций за 3 с)

n	**Наихудшая сложость алгоритма по времени**	**Комментарий**
⩽ [10–11]	$O(n!)$, $O(n^6)$	Например: подсчет перестановок (см. раздел 3.2)
⩽ [15–18]	$O(2^n \times n^2)$	Например: решение задачи коммивояжера методами динамического программирования (ДП) (см. раздел 3.5.2)
⩽ [18–22]	$O(2^n \times n)$	Например: задачи динамического программирования с использованием операций с битовой маской (см. раздел 8.3.1)
⩽ 100	$O(n^4)$	Например: трехмерная задача динамического программирования (DP) + $O(n)$loop, ${}_nC_{k=4}$,
⩽ 400	$O(n^3)$	Например: алгоритм Флойда–Уоршелла (см. раздел 4.5)
⩽ 2K	$O(n^2 \log_2 n)$	Например: вложенные циклы с глубиной вложения = 2 + структуры данных на деревьях (см. раздел 2.3)
⩽ 10K	$O(n^2)$	Например: сортировка пузырьком/выбором/вставкой (см. раздел 2.2)
⩽ 1M	$O(n \log_2 n)$	Например: сортировка слиянием, построение дерева отрезков (см. раздел 2.3)
⩽ 100M	$O(n)$, $O(\log_2 n)$, $O(1)$	Большинство сложностей на олимпиадах по программированию возникают в случае $n \leqslant 1M$ («узкое место» – операции ввода-вывода)

образом, что за решение каждой из частей задачи начисляются баллы. Чтобы получить драгоценные баллы, можно попробовать решить задачу «в лоб», набрав таким образом несколько баллов и получив возможность лучше понять задачу. При этом за представленное «медленное» решение жюри не снимет баллы, поскольку в соревнованиях IOI скорость не является одним из главнейших факторов при оценке решения задач. Решив задачу перебором, постепенно улучшайте ваше решение, чтобы получить больше очков.

Упражнение 1.2.2. Ответьте на следующие вопросы, используя имеющиеся у вас знания о классических алгоритмах и их временной сложности. После того как вы прочитаете эту книгу до конца, попробуйте выполнить данное упражнение снова.

1. Существует n веб-страниц ($1 \leqslant n \leqslant 10M$). Каждая i-я веб-страница имеет рейтинг страницы r_i. Вы хотите выбрать 10 страниц с наивысшим рейтингом. Какой из описанных методов будет работать лучше:
   a) вы загрузите рейтинги всех n веб-страниц в память, отсортируете (см. раздел 2.2) их в порядке убывания рейтинга, взяв первые 10;
   b) вы используете очередь с приоритетом («кучу») (см. раздел 2.3).
2. Для заданной целочисленной матрицы Q, имеющей размерность $M \times N$ ($1 \leqslant M, N \leqslant 30$), определите, существует ли ее подматрица размера $A \times B$ ($1 \leqslant A \leqslant M$, $1 \leqslant B \leqslant N$), для которой среднее значение элементов матрицы mean(Q) = 7. Вы:
   a) построите все возможные подматрицы и проверите выполнение условия mean(Q) = 7 для каждой из подматриц. Временная сложность этого алгоритма составит $O(M^3 \times N^3)$;
   b) построите все возможные подматрицы, но реализуете алгоритм, имеющий временную сложность $O(M^2 \times N^2)$, используя следующий метод: _____.
3. Пусть имеется список L, состоящий из $10K$ целых чисел, и вам нужно часто вычислять значение суммы элементов списка, начиная с i-го и заканчивая j-м, т. е. `sum(i, j)`, где `sum(i, j)` = `L[i] + L[i + 1] + ... + L[j]`. Какую структуру данных нужно при этом использовать:
   a) простой массив (см. раздел 2.2);
   b) простой массив, предварительно обработанный с использованием метода динамического программирования (см. разделы 2.2 и 3.5);
   c) сбалансированное двоичное дерево поиска (см. раздел 2.3);
   d) двоичную кучу (см. раздел 2.3);
   e) дерево отрезков (см. раздел 2.4.3);
   f) двоичное индексированное дерево (дерево Фенвика) (см. раздел 2.4.4);
   g) суффиксный массив (см. раздел 6.6.2) или альтернативный вариант, суффиксный массив (см. раздел 6.6.4).
4. Для заданного множества S из N точек, случайно разбросанных по 2D-плоскости ($2 \leqslant N \leqslant 1000$), найдите две точки, принадлежащие множеству S, которые имеют наибольшее евклидово расстояние между ними. Подходит ли для решения данной задачи алгоритм полного перебора с временной сложностью $O(N^2)$, который перебирает все возможные пары точек:
   a) да, полный перебор подходит;
   b) нет, мы должны найти другой вариант решения. Мы должны использовать следующий метод: _____ .

5. Вы должны вычислить кратчайший путь между двумя вершинами на взвешенном ориентированном ациклическом графе (Directed Acyclic Graph, DAG), для которого выполняются условия $|V|, |E| \leqslant 100K$. Алгоритм(ы) какого типа можно использовать на олимпиаде по программированию (то есть с ограничением по времени приблизительно 3 секунды):
   a) динамическое программирование (см. разделы 3.5, 4.2.5 и 4.7.1);
   b) поиск в ширину (см. разделы 4.2.2 и 4.4.2);
   c) алгоритм Дейкстры (см. раздел 4.4.3);
   d) алгоритм Форда–Беллмана (см. раздел 4.4.4);
   e) алгоритм Флойда–Уоршелла (см. раздел 4.5).
6. Какой алгоритм создает список первых $10K$ простых чисел, обладая при этом лучшей временной сложностью (см. раздел 5.5.1):
   a) решето Эратосфена (см. раздел 5.5.1);
   b) проверка истинности выражения `isPrime(i)` для каждого числа $i \in [1..10K]$ (см. раздел 5.5.1).
7. Вы хотите проверить, является ли факториал числа n, то есть $n!$, числом, которое делится без остатка на целое число m. $1 \leqslant n \leqslant 10\,000$. Как вы выполните эту проверку:
   a) напишете `n! % m == 0`;
   b) наивный подход, приведенный выше, не будет работать, вы используете следующий метод: _____ (см. раздел 5.5.1).
8. Задание аналогично вопросу **4**, но с большим набором точек: $N \leqslant 1M$ – и одним дополнительным ограничением: точки случайным образом разбросаны по 2D-плоскости. Тогда:
   a) все еще можно использовать полный перебор, упомянутый в вопросе 3;
   b) наивный подход, приведенный выше, не будет работать, вы используете следующий метод: _____ (см. раздел 7.3.7).
9. Вы хотите найти и перечислить все вхождения подстроки P (длиной m) в (длинную) строку T (длины n). Ограничения длины строк: n и m имеют максимум $1M$ символов. Тогда:
   a) вы используете следующий фрагмент кода на C++:

```
for (int i = 0; i < n; i++) {
  bool found = true;
  for (int j = 0; j < m && found; j++)
    if (i + j >= n || P[j] != T[i + j]) found = false;
  if (found) printf("P is found at index %d in T\n", i);
}
```

   b) наивный подход, приведенный выше, не будет работать, и вы используете следующий метод: _____ (см. раздел 6.4 или 6.6).

1.2.4. Совет 4: совершенствуйте свои знания языков программирования

На олимпиадах по программированию ICPC[1] поддерживается несколько языков программирования, включая C/C++ и Java.

Какие языки программирования нужно стремиться освоить?

Основываясь на своем опыте, мы предпочитаем C++ со встроенной стандартной библиотекой шаблонов (STL), но полагаем, что нужно также освоить Java. Хотя Java работает медленнее, в Java есть мощные встроенные библиотеки и API, такие как BigInteger/BigDecimal, GregorianCalendar, Regex и т. д.

Кроме того, программы на Java легче отлаживать благодаря возможности трассировки стека на виртуальной машине в случае сбоя программы (в отличие от дампов ядра или аварийного завершения программы в C/C++). С другой стороны, C/C++ также имеет свои достоинства. В зависимости от решаемой задачи каждый из этих языков может оказаться лучшим для реализации решения в кратчайшие сроки.

Предположим, что в задаче требуется вычислить 25! (факториал 25). Это очень большое число: 15 511 210 043 330 985 984 000 000. Оно намного превышает самый большой встроенный базовый целый тип данных (`unsigned long long`: $2^{64}-1$). Поскольку в C/C++ нет встроенной арифметической библиотеки произвольной точности, нам потребовалось бы реализовать ее с нуля.

Однако код на Java очень прост (подробнее – в разделе 5.3). В этом случае использование Java определенно сокращает время кодирования.

```
import java.util.Scanner;
import java.math.BigInteger;

class Main {                                    // стандартное имя класса в "Онлайн-арбитре"
  public static void main(String[] args) {
    BigInteger fac = BigInteger.ONE;
    for (int i = 2; i <= 25; i++)
      fac = fac.multiply(BigInteger.valueOf(i));                  // это есть в библиотеке!
    System.out.println(fac);
} }
```

Освоение и понимание всех возможностей вашего любимого языка программирования также важно. Рассмотрим случай с нестандартным форматом ввода: первая строка ввода представляет собой целое число *N*. За ним следуют *N* строк, каждая из которых начинается с символа «0», за ним следует точка («.»), за ней идет неизвестное число цифр (до 100 цифр), и, наконец, строка заканчивается тремя точками («...»):

---

[1] Личное мнение: по состоянию на 2012 год язык Java все еще не поддерживался в IOI. Олимпиады IOI проводятся на трех языках программирования: C, C++ и Pascal. С другой стороны, на финальных соревнованиях на кубок мира ICPC (и, следовательно, на большинстве региональных) можно использовать C, C++ и Java для выполнения заданий. Поэтому, вероятно, «лучшим» языком является C++, так как он поддерживается в обоих соревнованиях и имеет мощную библиотеку STL. Если участники IOI решат освоить C++, они смогут использовать тот же язык (но на более высоком уровне мастерства), участвуя в ACM ICPC, выполняя задания отборочных туров на уровне университета.

```
3
0.1227...
0.517611738...
0.7341231223444344389923899277...
```

Ниже приводится один из возможных вариантов решения:

```
#include <cstdio>
using namespace std;

int N;                                  // использовать глобальные переменные в олимпиадах
                                              // по программированию – хорошая стратегия
char x[110];                            // сделайте привычкой устанавливать размер массива
                                                    // немного больше необходимого
int main() {
  scanf("%d\n", &N);
  while (N--) { // мы просто объявляем цикл N, N-1, N-2, ..., 0
    scanf("0.%[0-9]...\n", &x);                          // '&' является необязательным,
                                                  // если x является массивом символов
                                    // примечание: если вас удивляет код, приведенный выше,
                                       // посмотрите описание scanf на www.cppreference.com
    printf("the digits are 0.%s\n", x);
} } // return 0;
```

Файл исходного кода: ch1_01_factorial.java; ch1_02_scanf.cpp

Не многие программисты на C/C++ знают о возможности использования scanf/printf, включенной в стандартную библиотеку ввода-вывода C, в реализации поиска регулярного выражения. Хотя scanf/printf являются стандартными процедурами ввода-вывода в C, они также могут использоваться в коде C++. Многие программисты на C++ привыкают постоянно использовать cin/cout, хотя они гораздо менее гибкие, чем scanf/printf, и работают гораздо медленнее.

На олипиадах по программированию, особенно ICPC, время, требуемое на написание кода, *не должно* быть вашим основным ограничением. Как только вы найдете «худший алгоритм, получающий положительную оценку», который уложится в лимит времени для вашей задачи, вы должны быстро и без проблем перевести его в безошибочный код!

Теперь попробуйте выполнить некоторые из упражнений ниже. Если вам потребуется написать более 10 строк кода для выполнения какого-либо из них, вам следует вернуться к изучению языков программирования и усовершенствовать свои знания.

Владение языками программирования, которые вы используете, знакомство с их встроенными процедурами чрезвычайно важны и очень помогут вам на олимпиадах по программированию.

Упражнение 1.2.3. Напишите максимально краткий работающий код для решения следующих задач.

1. Используя Java, считайте входные данные в формате double (например: 1.4732, 15.324547327 и т. д.) и выведите их (echo) в формате поля для

цифр фиксированной ширины, где ширина поля составляет семь знаков, оставляя три знака после десятичной точки (например: `ss1.473`, `s15.325` и т. д., где «s» обозначает пробел).

2. Если задано целое число n ($n \leqslant 15$), выведите число π с n цифр после десятичной точки (с округлением) (например, для $n = 2$ выведите 3.14; для $n = 4$ выведите 3.1416; для $n = 5$ выведите 3.14159).
3. Для заданной даты определите день недели (понедельник, ..., воскресенье), на который приходится этот день (например, 9 августа 2010 г. – дата выхода первого издания этой книги – понедельник).
4. Для последовательности из n случайных целых чисел выведите отдельные (уникальные) целые числа в отсортированном порядке.
5. Даны: даты рождения n людей (различные действительные числа в формате ДД, ММ, ГГГГ. Упорядочите их сначала в порядке возрастания месяцев рождения (ММ), затем по возрастанию дат рождения (ДД) и, наконец, по возрасту людей в порядке увеличения возраста.
6. Дан список отсортированных целых чисел L размером до $1M$ элементов. Определите, входит ли число v в L, выполнив не более чем 20 операций сравнения (более подробно см. раздел 2.2).
7. Сгенерируйте все возможные сочетания {'A', 'B', 'C', ..., 'J'} для первых $N = 10$ букв английского алфавита (см. раздел 3.2.1).
8. Сгенерируйте все возможные подмножества чисел {0, 1, 2, ..., $N - 1$}, для $N = 20$ (см. раздел 3.2.1).
9. Пусть задана строка, представляющая число в системе счисления с основанием X; преобразуйте ее в эквивалентную строку в системе счисления с основанием Y, где $2 \leqslant X, Y \leqslant 36$. Например: «FF» в шестнадцатеричной системе счисления ($X = 16$) равно 255 в десятичной системе счисления ($Y_1 = 10$) и 11111111 в двоичной системе счисления ($Y_2 = 2$). См. раздел 5.3.2.
10. Определим специальное слово как строчную букву алфавита, за которой следуют две цифры подряд.
    Для заданной строки замените все специальные слова с длиной три символа на три звезды «***», например:

    S = «строка: a70 и z72 будут заменены, aa24 и a872 не будут»

    должно быть преобразовано в

    S = «строка: *** и *** будут заменены, aa24 и a872 не будут».
11. Дано *правильное* математическое выражение, содержащее в одной строке символы «+», «-», «*», «/», «(» и «)». Вычислите значение этого выражения.
    (Например, довольно сложное, но правильное выражение `3 + (8 - 7,5) * 10/5 - (2 + 5 * 7)` должно давать результат `-33.0` при выполнении вычислений со стандартным приоритетом операций.)

1.2.5. Совет 5: овладейте искусством тестирования кода

Вы думаете, что решили конкретную задачу. Вы определили тип, к которому относится задача, разработали алгоритм для ее решения, убедились, что ал-

горитм (с используемыми им структурами данных) будет работать, укладываясь в отведенное время (и в пределах ограничений по памяти), учитывая его временную сложность (и сложность по памяти), вы реализовали алгоритм, но ваше решение все еще не зачтено (т. е. не получило вердикт тестирующей системы «AC»).

В зависимости от того, в каких соревнованиях по программированию вы участвуете, вы можете получить или не получить баллы за частичное решение задачи. На олимпиадах ICPC вы будете получать баллы только за каждую полностью решенную задачу, если код, написанный вашей командой, пройдет все секретные тесты, разработанные для этой задачи. Только в этом случае вы получите оценку тестирующей системы «зачтено» (AC). Все другие оценки, такие как ошибка представления (PE), неправильный ответ (WA), превышение лимита времени (TLE), превышение лимита памяти (MLE), ошибка времени выполнения (RTE) и т. д., не принесут желанных баллов вашей команде. На олимпиадах IOI в настоящее время (2010–2012) используется система подсчета подзадач. Тестовые примеры, которые выполняются, чтобы проверить правильность решения, разбиты на подзадачи, которые обычно представляют собой более простые варианты исходной задачи с меньшими ограничениями по вводу. Вы получите баллы за решение подзадачи только в том случае, если ваш код проходит все тесты в нем.

В любом случае вам необходимо уметь разрабатывать хорошие, полные и сложные тестовые примеры. Пример ввода-вывода, приведенный в описании задачи, по своей природе тривиален и, следовательно, обычно не является хорошим тестом для проверки правильности вашего кода.

Вместо того чтобы тратить попытки (и, таким образом, терять время, а в ICPC еще и накапливать штрафное время), вы можете разработать сложные тестовые примеры для тестирования написанного кода на вашей собственной машине[1]. Убедитесь, что ваш код способен пройти эти тесты (иначе нет смысла отправлять ваше решение, так как оно может быть неверным – если только вы не хотите проверить, что авторы задачи предусмотрели при разработке тестов найденные вами контрпримеры).

Некоторые наставники поощряют своих студентов соревноваться друг с другом, разрабатывая тестовые примеры. Если тесты студента А могут «сломать» код студента Б, то студент А получит бонусные баллы. Вы можете попробовать этот способ, тренируя свою команду :).

Основываясь на своем опыте, мы можем дать несколько рекомендаций по разработке хороших тестовых примеров.

Обычно это те шаги, которые предпринимали авторы задач.

1. Ваши тестовые примеры должны включать в себя примеры, приведенные в задании, так как выходные данные приведенных примеров гаран-

[1] Среды и инструменты программирования, доступные участникам олимпиад по программированию, различаются для разных конкурсов. Это может поставить в невыгодное положение участников, которые слишком сильно полагаются на какую-либо современную интегрированную среду разработки (IDE) – например, Visual Studio, Eclipse и т. д. – при отладке кода. Возможно, будет полезно попрактиковаться в программировании с помощью одного лишь текстового редактора и компилятора!

тированно будут правильными. Используйте команду `fc` в Windows или `diff` в UNIX, чтобы проверить, что выводит ваш код (если даны примеры входных данных), и сравнить выходные данные вашего кода с выходными данными из примера. Избегайте ручного сравнения, поскольку люди склонны к ошибкам и плохо справляются с такими задачами, особенно в случае, когда результат выводится в строго определенном формате (пример: наличие пустой строки между контрольными примерами и после каждого контрольного примера). Для этого скопируйте и вставьте образец входных данных и пример выходных данных из описания задачи, а затем сохраните их в файлы (назовите их, например, `input` и `output`, или дайте им какое-либо еще осмысленное название). Затем после компиляции вашей программы (предположим, что имя исполняемого файла `a.out` – оно присваивается исполняемому файлу в g++ по умолчанию) выполните ее, перенаправив ввод-вывод: `./a.out < input > myoutput`. Наконец, выполните сравнение `diff myoutput output`, чтобы выделить все (даже незаметные при ручном сравнении) различия, если таковые существуют.

2. Для задач с мультитестами (см. раздел 1.3.2), вы должны включить два одинаковых тестовых примера подряд для одного запуска программы. Оба примера должны вывести одинаковые правильные ответы (о которых заранее известно, что они правильные). Это помогает определить, не забыли ли вы инициализировать какие-либо переменные: если первый экземпляр выдает правильный ответ, а второй – нет, вероятно, вы не инициализировали свои переменные.
3. Ваши тестовые примеры должны включать краевые случаи для определения потенциальных проблем, которые происходят при превышении каких-либо предельно допустимых параметров. Посмотрите на задачу глазами ее автора, думайте, как он, и постарайтесь найти наихудший вариант для вашего алгоритма, выявив такие проблемы.
4. Ваши тестовые примеры должны включать большие объемы данных. Увеличивайте размер входных данных постепенно до максимальных пределов, указанных в описании задачи. Используйте большие объемы тестовых данных с тривиальной структурой, которые легко проверить с помощью ручных вычислений, и большие объемы случайных тестовых данных, чтобы проверить, укладывается ли написанный вами код в ограничения по времени, выдавая результат, похожий на верный (поскольку корректность здесь будет трудно проверить). Иногда ваша программа может работать на небольших тестовых примерах, но выдавать неправильный ответ, «вылетать» или превышать ограничения по времени при увеличении объема входных данных. Если это происходит, проверьте свой код на наличие ошибок переполнения, связанных ошибок или улучшите свой алгоритм.
5. Хотя такие ситуации редко встречаются на соревнованиях по программированию, не думайте, что входные данные всегда будут правильно отформатированы, если в описании задачи это не указано явно (особенно для плохо написанной задачи). Попробуйте добавить дополнительные пробелы (или как пробелы, так и символы табуляции) во входные данные и проверьте, сможет ли ваш код считать их правильно, без сбоев.

Однако, даже тщательно выполнив все описанные выше шаги, вы все равно можете не получить оценку «зачтено» AC. На соревнованиях олимпиады ICPC вы с вашей командой можете проверить вердикт тестирующей системы и просмотреть текущую таблицу результатов (обычно они доступны в течение первых четырех часов соревнования), чтобы определить стратегию своих дальнейших действий. На соревнованиях IOI участник может увидеть только вердикт тестирующей системы для тестов, на которых запускалась задача, таблица результатов ему недоступна. Приобретая опыт участия в таких конкурсах, вы сможете выбирать более успешную стратегию.

Упражнение 1.2.4. Ситуационная ориентация.
(В основном это применимо для ICPC; для IOI это не так актуально.)

1. Вы получаете оценку тестирующей системы «неверный ответ» (WA) для очень простой задачи. Что вы будете делать:
   a) не станете решать эту задачу и переключитесь на следующую;
   b) попытаетесь улучшить производительность вашего решения (оптимизация кода / лучший алгоритм);
   c) создадите сложные тестовые примеры, чтобы найти ошибку;
   d) (в командных соревнованиях) попросите своего товарища по команде заново решить задачу.
2. Вы получаете оценку тестирующей системы «превышение лимита времени» (TLE) для представленного решения $O(N^3)$.
   Тем не менее максимальное значение N составляет всего 100. Что вы будете делать:
   a) не станете решать эту задачу и переключитесь на следующую;
   b) попытаетесь улучшить производительность вашего решения (оптимизация кода / лучший алгоритм);
   c) создадите сложные тестовые примеры, чтобы найти ошибку.
3. Вернитесь к вопросу 2: что вы будете делать в случае, если максимальное значение N составляет 100 000?
4. Еще раз вернитесь к вопросу 2. Что вы будете делать в случае, если максимальное значение N равно 1000, выходной результат зависит только от размера входного N и у вас еще остается четыре часа времени до конца соревнований?
5. Вы получаете вердикт тестирующей системы «ошибка времени выполнения» (RTE). Ваш код (на ваш взгляд) отлично работает на вашем компьютере. Что вы должны сделать?
6. Через тридцать минут после начала соревнования вы посмотрели на таблицу лидеров. Множество других команд решили задачу X, которую ваша команда не пыталась решить. Что вы будете делать?
7. В середине соревнования вы смотрите на таблицу лидеров. Ведущая команда (предположим, что это не ваша команда) только что решила задачу Y. Что вы будете делать?
8. Ваша команда потратила два часа на сложную задачу. Вы отправили несколько вариантов решения, выполненных разными членами команды.

Все решения были оценены как неверные. Вы понятия не имеете, что не так. Что вы будете делать?

9. До окончания соревнования остается один час. У вас есть одна оценка «неверный ответ» (WA) для выполненной задачи и одна свежая идея для решения другой задачи. Что вы (или ваша команда) будете делать:
   a) оставите нерешенной задачу, получившую оценку WA, и переключитесь на другую задачу, попытавшись ее решить;
   b) будете настаивать на том, что необходимо отладить код, получивший оценку WA. Вам не хватает времени, чтобы начать работать над решением новой задачи;
   c) (в ICPC) распечатаете код задачи, получившей оценку WA. Попросите двух других членов команды изучить его, пока вы переключаетесь на другую задачу, пытаясь в итоге сдать обе задачи.

1.2.6. Совет 6: практикуйтесь и еще раз практикуйтесь!

Конкурентоспособные программисты, как настоящие спортсмены, должны регулярно тренироваться, чтобы быть в форме.

Здесь мы приводим список нескольких веб-сайтов с ресурсами, которые могут помочь улучшить ваши навыки решения задач по программированию. Мы верим, что успех приходит в результате постоянных усилий, направленных на улучшение себя.

Архив задач Университета Вальядолида (UVa, Испания) [47] содержит задачи прошлых олимпиад, проводимых ICPC (местных, региональных и вплоть до финалов чемпионатов мира), а также задачи из других источников, в том числе различные задачи, в разное время предлагавшиеся на конкурсах, проводимых в Университете Вальядолида. Вы можете попробовать решить эти задачи и представить свои решения онлайн-арбитру. Онлайн-арбитр оценит правильность вашего кода и вскоре выдаст свою оценку. Попробуйте решить задачи из этой книги, и, возможно, вы увидите свое имя в списке 500 победителей на этом сайте :-).

В мае 2013 года необходимо было решить более 542 задач (⩾ 542), чтобы попасть в рейтинг топ-500. Стивен занимает в нем 27-е место (решив 1674 задачи), а Феликс – 37-е (решив 1487 задач) из 149 008 зарегистрированных пользователей сайта Университета Вальядолида (всего на сайте размещено 4097 задач).

Рис. 1.2 ❖ *Слева*: архив задач с тестирующей системой Университета Вальядолида; *справа*: онлайн-архив ICPC

Сетевым «побратимом» архива задач университета Вальядолида является онлайн-архив задач ICPC [33], в котором собраны почти все недавние серии задач с региональных и международных финалов ICPC начиная с 2000 года. Тренируйтесь здесь, если хотите преуспеть в будущих соревнованиях ICPC. Обратите внимание, что в октябре 2011 года около сотни задач с ресурса Live Archive (включая задачи, вошедшие во второе издание этой книги) были также включены в архив задач Университета Вальядолида.

Национальная олимпиада США по информатике USACO имеет очень полезный обучающий веб-сайт [48] с онлайн-конкурсами, которые помогут вам освоить навыки программирования и решения задач. Эти материалы ориентированы больше на участников IOI, чем на участников ICPC. Открывайте данный сайт и тренируйтесь.

Sphere Online Judge (SPOJ) [61] – еще один интерактивный сайт, оценивающий решения задач, на котором квалифицированные пользователи могут добавлять свои задачи. Этот сайт довольно популярен в таких странах, как Польша, Бразилия и Вьетнам.

Мы опубликовали несколько задач, придуманных нами, на сайте SPOJ.

Рис. 1.3 ❖ *Слева*: вход на учебный раздел сайта USACO; *справа*: Sphere Online Judge

TopCoder часто устраивает онлайн-раунды [32], состоящие из трех задач разного уровня сложности, которые необходимо решить за 1–2 часа. После завершения фазы программирования вам предоставляется возможность «сломать» код других участников, разрабатывая сложные тестовые примеры. Этот онлайн-ресурс использует систему цветовых рейтингов («красный», «желтый», «синий» и т. д.), чтобы отличить участников, которые действительно хорошо решают сложные задачи, с более высоким рейтингом от просто прилежных участников, решающих большее число более простых задач.

1.2.7. Совет 7: организуйте командную работу (для ICPC)

Этому последнему навыку не так просто научить, но вот некоторые идеи, которые стоит попробовать для улучшения работы вашей команды:

- практикуйтесь писать код на чистом листе бумаги (это полезно, поскольку вы можете писать код в то время, когда ваш товарищ по команде использует компьютер. Когда настанет ваша очередь использовать компьютер, вы можете просто набрать код как можно быстрее, а не тратить время на размышления перед компьютером);
- сформируйте привычку «отправить и распечатать»: если ваш код получит оценку «зачтено» (AC), просто выкиньте распечатку. Если же вам все еще не удалось получить оценку AC, вычитайте код, используя эту рас-

печатку (и пусть ваш товарищ по команде использует компьютер для решения другой задачи). Однако помните: отладка кода без компьютера – непростой для освоения навык;
- если ваш товарищ по команде в настоящее время пишет код для своего алгоритма, подготовьте тесты для его кода, включив тестовые данные для крайних случаев, проверяющие код в случаях превышения каких-либо предельно допустимых параметров (надеюсь, его код выдержит все эти испытания);
- особый совет: подружитесь со своими товарищами по команде «в реальной жизни», вне тренировок и соревнований.

1.3. Начинаем работу: простые задачи

Примечание. Этот раздел можно пропустить, если вы уже много лет участвуете в олимпиадах по программированию.

Данный раздел адресован новичкам в олимпиадном программировании.

1.3.1. Общий анализ олимпиадной задачи по программированию

Олимпиадная задача по программированию обычно включает в себя следующие элементы:

- **фоновое повествование** / **описание задачи**. Обычно описание более простых задач составляется так, чтобы обмануть участников, придав простым задачам вид более сложных – например, добавив «дополнительную информацию». Участники соревнований должны быть в состоянии отфильтровать все несущественные детали и сосредоточиться на главном. Например, в задаче UVa 579 – ClockHands все вводные абзацы, кроме последнего предложения, посвящены истории часов и совершенно не связаны с реальной задачей. Однако описание более сложных задач обычно максимально кратко – они уже достаточно сложны без дополнительных «украшений»;
- **описание входных и выходных данных**. В этом разделе вам дадут подробную информацию о формате входных данных и о том, в каком формате вы должны вывести выходные данные. Эта часть обычно написана более формальным языком. Хорошая задача должна иметь четкие входные ограничения, так как одна и та же задача может быть решена с помощью разных алгоритмов в зависимости от того, какие входные ограничения для нее поставлены (см. табл. 1.4);
- **образец входных данных и образец выходных данных**. Авторы задач обычно предоставляют участникам только тривиальные тестовые примеры. Образец входных/выходных данных предназначен для участников соревнований, чтобы проверить их базовое понимание задачи и может ли их код разобрать входные данные, представленные в заданном формате, и выдать правильные выходные данные, используя заданный фор-

мат вывода. Не отправляйте в тестирующую систему свой код, если он даже не прошел тест с заданным примером входных/выходных данных. См. раздел 1.2.5, где говорится о тестировании кода перед отправкой;

- **подсказки или сноски**. В некоторых случаях авторы задачи могут оставлять подсказки или добавлять сноски для более детального описания задачи.

1.3.2. Типичные процедуры ввода/вывода

Мультитест

Правильность вашего кода, решающего задачу с олимпиады по программированию, обычно проверяется запуском вашего кода на *нескольких* тестовых примерах. Вместо того чтобы использовать множество отдельных файлов для тестовых примеров, в современных соревнованиях по программированию иногда используется один файл с несколькими тестовыми примерами (называемый мультитест). В этом разделе мы приведем пример задачи с мультитестом на основе очень простой задачи: для двух целых чисел в одной строке, выведите их сумму в одной строке. Мы проиллюстрируем три возможных формата ввода/вывода:

- количество тестов приведено в первой строке входных данных;
- несколько тестовых примеров заканчиваются специальными значениями (обычно это нули);
- несколько тестовых примеров завершаются символом EOF (конец файла).

```
Исходный код на C/C++                              | Пример входных | Пример выходных
                                                   |      данных    |      данных
---------------------------------------------------------------------------------
int TC, a, b;                                      | 3              | 3
scanf("%d", &TC); // число тестовых примеров       | 1 2            | 12
while (TC--) { // сокращенная запись:              | 5 7            | 9
               // повторять, пока значение         |                |
               // переменной не станет равно 0     |                |
  scanf("%d %d", &a, &b); // вычисление ответа     | 6 3            |----------------
  printf("%d\n", a + b);  // "на лету"             |----------------|
}
---------------------------------------------------------------------------------
int a, b;                                          | 1 2            | 3
// остановиться, когда оба целых числа равны 0     | 5 7            | 12
while (scanf("%d %d", &a, &b), (a || b))           | 6 3            | 9
  printf("%d\n", a + b);                           | 0 0            |----------------
---------------------------------------------------------------------------------
int a, b;                                          | 1 2            | 3
// scanf возвращает число считанных элементов      | 5 7            | 12
while (scanf("%d %d", &a, &b) == 2)                | 6 3            | 9
// либо вы можете проверить наличие                |----------------|----------------
// символа EOF во входных данных, т. е.            |                |
// while (scanf("%d %d", &a, &b) != EOF)           |                |
  printf("%d\n", a + b);                           |                |
```

Номера примеров и пустые строки

Для оформления решений ряда задач с мультитестом нужно, чтобы выходные данные для каждого тестового примера сопровождались номером этого тестового примера. В некоторых случаях также требуется, чтобы выходные данные, относящиеся к различным тестовым примерам, были отделены пустой строкой, т. е. в конце выходных данных для каждого тестового примера добавлялась пустая строка. Давайте изменим код для решения простой задачи, описанной выше, добавив номер тестового примера в выходные данные так, чтобы выходные данные имели такой формат: `"Case [НОМЕР]: [ОТВЕТ]"` – и в конце данных, относящихся к каждому из тестовых примеров, добавлялась пустая строка. Предполагая, что набор входных данных завершается символом EOF, мы можем написать такой код:

```
Исходный код на C/C++                          | Пример входных | Пример выходных
                                               |      данных    |      данных
-----------------------------------------------------------------------------
int a, b, c = 1;                               | 1 2            | Пример 1: 3
while (scanf("%d %d", &a, &b) != EOF)          | 5 7            |
  // обратите внимание на два \ n               | 6 3            | Пример 2: 12
  printf("Case %d: %d\n\n", c++, a + b);       |----------------|
                                               |                | Пример 3: 9
                                               |                |
                                               |                |-----------------
```

Для некоторых других задач от нас требуется добавлять пустые строки только между наборами данных, относящихся к разным тестовым примерам. Если мы воспользуемся подходом, описанным выше, то добавим дополнительную строку в конце файла с выходными данными. Это приведет к тому, что за решение данной задачи мы получим вердикт «ошибка представления» (PE), а в более современных системах и вовсе вердикт «неверный ответ» (WA). Чтобы избежать этого, нам нужно использовать следующий код:

```
Исходный код на C/C++                          | Пример входных | Пример выходных
                                               |      данных    |      данных
-----------------------------------------------------------------------------
int a, b, c = 1;                               | 1 2            | Пример 1: 3
while (scanf("%d %d", &a, &b) != EOF) {        | 5 7            |
  if (c > 1) printf("\n"); // 2-й пример и т. д. | 6 3            | Пример 2: 12
  printf("Case %d: %d\n", c++, a + b);         |----------------|
}                                              |                | Пример 3: 9
                                               |                |----------------
```

Переменное количество входных данных

Давайте немного изменим простую задачу, рассмотренную выше. Каждый тестовый пример (каждая строка входных данных) теперь будет содержать целое число k ($k \geqslant 1$), за которым следуют k целых чисел. Нам теперь требуется вывести сумму этих k целых чисел. Предполагая, что набор входных данных завершается символом EOF и номера тестовых примеров выводить не нужно, мы можем написать такой код:

Исходный код на C/C++	Пример входных данных	Пример выходных данных

`int k, ans, v;`
`while (scanf("%d", &k) != EOF) {`
`  ans = 0;`
`  while (k--) { scanf("%d", &v); ans += v; }`
`  printf("%d\n", ans);`
`}`
	1 1
	2 3 4
	3 8 1 1
	4 7 2 9 3
	5 1 1 1 1 1
		1
		7
		10
		21
		5

Упражнение 1.3.1*. Что, если автор задачи решит сделать входные данные немного более сложными и неоднородными? Теперь строки входных данных уже не будут содержать набор целых чисел. Вместо этого вам будет предложено сложить все целые числа в каждом тестовом примере (каждой строке входных данных). Подсказка: см. раздел 6.2.

Упражнение 1.3.2*. Перепишите весь исходный код на C/C++, приведенный в разделе 1.3.2, на Java.

1.3.3. Начинаем решать задачи

Нет лучшего способа начать свой путь в олимпиадном программировании, чем решить несколько задач. Чтобы помочь вам выбрать задачи, с которых можно начать знакомство с олимпиадным программированием, из 4097 задач, предлагаемых к решению в архиве задач на сайте университета Вальядолида [47], мы составили список из нескольких самых простых задач Ad Hoc, то есть не требующих знания специальных алгоритмов. Более подробно о задачах Ad Hoc будет рассказываться в следующем разделе 1.4.

- **Очень легкие задачи.** Вы должны получить оценку тестирующей системы «AC»[1] за решение этих задач, потратив не более 7 минут[2] на решение каждой задачи! Если вы новичок в олимпиадном программировании, мы настоятельно рекомендуем вам начать с решения некоторых задач из этой категории, после того как вы выполните упражнения предыдущего раздела 1.3.2.
  Примечание. Поскольку каждая категория содержит множество задач, которые вы можете попробовать решить, мы (при помощи шрифтовых выделений) *выделили* не более трех (3) задач в каждой категории, которые вы **обязательно должны попытаться решить, ***. Мы считаем, что это самые интересные и хорошие задачи.
- **Легкие задачи.** Мы разделили категорию задач «Легкие» на две более мелкие подкатегории. Задачи, отнесенные к категории «Легкие», все еще просты, но они «немного» сложнее, чем «Очень легкие задачи».

[1] Не расстраивайтесь, если вы не сможете это сделать. Есть много причин, почему ваш код может не получить оценку «AC».

[2] Семь минут – лишь приблизительная оценка. Некоторые из этих задач можно решить, написав всего одну строчку кода.

❍ **Задачи средней сложности: на одну ступеньку выше легких.** Здесь мы перечислим некоторые другие специальные задачи, которые могут быть немного сложнее (или длиннее), чем задачи из категории «Легкие».

- Очень легкие задачи из архива задач на сайте университета Вальядолида (решаются менее чем за 7 минут)

  1. UVa 00272 – TEX Quotes (просто заменить все двойные кавычки на кавычки в формате TEX ())
  2. *UVa 01124 – Celebrity Jeopardy* (LA 2681, просто повторить (echo) / вывести текст входных данных)
  3. UVa 10550 – Combination Lock (просто сделайте то, что требуется в задаче)
  4. UVa 11044 – Searching for Nessy (можно решить, написав всего одну строчку кода / формулу)
  5. **UVa 11172 – Relational Operators *** (очень простая задача; можно решить, написав всего одну строчку кода)
  6. UVa 11364 – Parking (последовательный просмотр данных для получения l & *r*, ответ: 2 ∗ (*r* – l))
  7. **UVa 11498 – Division of Nlogonia *** (просто используйте операторы if-else)
  8. UVa 11547 – Automatic Answer (можно решить, написав всего одну строчку кода; временная сложность *O*(1))
  9. **UVa 11727 – Cost Cutting *** (отсортируйте три числа и получите медиану)
  10. UVa 12250 – Language Detection (LA 4995, Куала–Лумпур'10; проверка if-else)
  11. *UVa 12279 – Emoogle Balance* (просто последовательный просмотр данных)
  12. *UVa 12289 – One-Two-Three* (просто используйте операторы if-else)
  13. *UVa 12372 – Packing for Holiday* (просто проверьте, все ли значения *L*, *W*, *H* ⩽ 20)
  14. *UVa 12403 – Save Setu* (элементарно)
  15. *UVa 12577 – Hajj-e-Akbar* (элементарно)

- Легкие задачи (чуть сложнее, чем очень легкие задачи)

  1. UVa 00621 – Secret Research (анализ случая только для четырех возможных результатов)
  2. **UVa 10114 – Loansome Car Buyer** * (просто смоделируйте процесс)
  3. UVa 10300 – Ecological Premium (игнорируйте количество животных)
  4. UVa 10963 – The Swallowing Ground (для объединения двух блоков промежутки между их столбцами должны быть одинаковыми)
  5. UVa 11332 – Summing Digits (простая рекурсия)
  6. **UVa 11559 – Event Planning *** (один проход при последовательном просмотре данных)

7. UVa 11679 – Sub-prime (смоделируйте ситуацию, затем проверьте, выполняются ли условия, что у всех банков неотрицательный резерв (величина резерва ⩾ 0))
8. UVa 11764 – Jumping Mario (один последовательный просмотр данных для подсчета высоких + низких прыжков)
9. **UVa 11799 – Horror Dash *** (один последовательный просмотр для поиска максимального значения)
10. UVa 11942 – Lumberjack Sequencing (проверьте, отсортированы ли входные данные по возрастанию/убыванию)
11. UVa 12015 – Google is Feeling Lucky (просмотрите список дважды)
12. *UVa 12157 – Tariff Plan* (LA 4405, Куала–Лумпур'08, рассчитайте и сравните)
13. *UVa 12468 – Zapping* (легко решить; есть только четыре возможности)
14. *UVa 12503 – Robot Instructions* (простая симуляция)
15. *UVa 12554 – A Special … Song* (симулятор)
16. IOI 2010 – Cluedo (используйте три указателя)
17. IOI 2010 – Memory (используйте два прохода при последовательном просмотре данных)

- Задачи средней сложности: на одну ступеньку выше легких задач (решение может занять 15–30 минут, но эти задачи все еще не слишком сложны)

1. UVa 00119 – Greedy Gift Givers (смоделируйте процесс отдачи и получения)
2. **UVa 00573 – The Snail *** (симуляция; обращайте особое внимание на граничные случаи!)
3. UVa 00661 – Blowing Fuses (имитация)
4. **UVa 10141 – Request for Proposal *** (решается с помощью однократного последовательного просмотра данных)
5. UVa 10324 – Zeros and Ones (упростите с помощью массива 1D: счетчик изменений)
6. UVa 10424 – Love Calculator (просто сделайте то, что требуется в задаче)
7. UVa 10919 – Prerequisites? (обработайте требования при чтении входных данных)
8. **UVa 11507 – Bender B. Rodriguez… *** (симуляция, if-else)
9. UVa 11586 – Train Tracks (решение «в лоб» приведет к превышению лимита времени (TLE); найдите структуру)
10. UVa 11661 – Burger Time? (последовательный просмотр данных)
11. UVa 11683 – Laser Sculpture (достаточно одного прохода при последовательном просмотре данных)
12. UVa 11687 – Digits (симуляция; простое решение)
13. **UVa 11956 – Brain****** (симуляция; игнорируйте «.»)
14. UVa 12478 – Hardest Problem… (попробуйте одно из восьми имен)
15. IOI 2009 – Garage (симуляция)
16. IOI 2009 – POI (сортировка)

1.4. Задачи Ad Hoc

Мы закончим эту главу обсуждением того типа задач в ICPC и IOI, который указан первым в нашей классификации, – это специальные задачи. Согласно USACO [48], задачи Ad Hoc – это задачи, которые «не могут быть отнесены к какому-либо еще типу», поскольку каждое описание задачи и соответствующее решение уникальны. Многие такие задачи довольно просты (как показано в разделе 1.3), однако это не относится ко всем задачам Ad Hoc.

Задачи Ad Hoc часто включаются в программу соревнований по программированию. На соревнованиях ICPC одна-две задачи из каждых десяти относятся к категории задач Ad Hoc. Если такая задача проста, то обычно именно она станет первой задачей, которую соревнующиеся команды решат на олимпиаде по программированию. Однако были случаи, когда решения задач Ad Hoc были слишком сложны и некоторые команды стратегически откладывали их до последнего часа. В региональном соревновании ICPC, в котором принимают участие около 60 команд, ваша команда заняла бы место в нижней половине турнирной таблицы (с 30-го по 60-е место), если бы она смогла решить только специальные задачи.

На олимпиаде в IOI 2009 и 2010 гг. каждый день 11 соревнований в списке предлагаемых к решению задач была одна простая задача, и обычно это была задача Ad Hoc. Если вы являетесь участником IOI, вы определенно не получите медаль за решение двух простых задач Ad Hoc в течение двух дней соревнований. Однако чем быстрее вы сможете решить эти две простые задачи, тем больше времени у вас останется для работы над другими сложными задачами.

Ниже мы приводим много задач Ad Hoc, которые решили в UVa Online Judge [47], разбив их по категориям. Мы считаем, что вы можете решить большинство из этих задач, не используя сложные структуры данных или алгоритмы, которые будут обсуждаться в последующих главах. Многие из этих специальных задач являются «простыми», но в некоторых из них могут быть «хитрые ловушки». Попробуйте решить несколько задач из каждой категории, прежде чем вы начнете читать следующую главу.

Примечание. Для небольшого числа задач, хотя они и включены в главу 1, могут потребоваться навыки программирования, о которых рассказывается в последующих главах, – например, знание линейных структур данных (массивов), о которых рассказывается в разделе 2.2, знание поиска, выполняемого с помощью возвратной рекурсии, о котором говорится в разделе 3.2, и т. д. Вы можете вернуться к этим более сложным специальным задачам позже, после того как усвоите необходимые понятия.

Категории специальных задач:

- **Игры (карты)**
  Есть много задач Ad Hoc, связанных с популярными играми. Многие из них относятся к картам и карточным играм. Как правило, в таких задачах вам нужно будет проанализировать строки входных данных (см. раздел 6.3), поскольку игральные карты имеют как масти (D / Diamond (бубновая) / ♦, C / Club (трефовая) / ♣, H / Heart (червонная) / ♥ и S / Spades (пиковая) / ♠), так и ранги, указывающие, какие карты внутри масти

«старше», «выше» или «более значимы», чем другие (обычно : 2 < 3 < … < 9 < T / Ten («десятка») < J / Jack (валет) < Q / Queen (дама) < K / King (король) < A / Ace (туз).

Неплохая идея – превратить эти сложные строки, состоящие из символов (букв) и цифр, в целочисленные индексы. Пример такого превращения: D2 → 0, D3 → 1, …, DA → 12, C2 → 13, C3 → 14, .., SA → 51. После этого вы можете работать с целочисленными индексами.

- **Игры (шахматы)**
  Шахматы – еще одна популярная игра, которая встречается на олимпиадах по программированию.
  Некоторые из задач этой категории относятся к задачам Ad Hoc, они перечислены в этом разделе. Некоторые из них – это задачи на комбинаторику, например подсчет количества способов размещения восьми ферзей на шахматной доске 8×8. О них речь пойдет в главе 3.
- **Прочие игры**, легкие и сложные (или более трудоемкие)
  Помимо карточных игр и шахмат, в программы соревнований по программированию вошли многие другие популярные игры: крестики-нолики, «камень, ножницы, бумага», настольные игры, бинго, боулинг и т. д. Знакомство с правилами этих игр может быть полезно, но большинство правил игры приведены в описании задачи, чтобы участники, незнакомые с этими играми, не оказались в невыгодном положении.
- Задачи, связанные с **палиндромами**
  Это также классические задачи. Палиндром – это слово (или последовательность слов), которое читается одинаково справа налево и слева направо. Наиболее распространенная стратегия проверки того, является ли слово палиндромом, состоит в том, чтобы переходить от первого символа к *среднему* и проверять, совпадают ли символы в соответствующей позиции сзади. Например, «ABCDCBA» – это палиндром. Для решения некоторых более сложных задач, связанных с палиндромами, прочитайте раздел 6.5, где рассказывается о динамическом программировании и строковых алгоритмах.
- Задачи, связанные с **анаграммами**
  Это еще одна из разновидностей классических задач. Анаграмма – слово (или фраза), буквы которого можно переставить так, чтобы получить другое слово (или фразу). Общая стратегия проверки того, являются ли два слова анаграммами, заключается в сортировке букв слов и сравнении результатов. Рассмотрим пример: `wordA = 'cab'`, `wordB = 'bca'`. После сортировки получим: `wordA = 'abc'` и `wordB = 'abc'` также являются анаграммами. См. раздел 2.2, где рассказывается о различных методах сортировки.
- Интересные задачи из **реальной жизни**, легкие и сложные (или более утомительные)
  Это одна из самых интересных категорий задач из архива задач университета Вальядолида. Мы считаем, что подобные задачи интересны тем, кто плохо знаком с компьютерными науками. Тот факт, что мы пишем программы для решения реальных проблем, может стать дополнительным стимулом для мотивации. Кто знает, может быть, вы сможете получить новую (и интересную) информацию из описания задачи!

- Задачи Ad Hoc, связанные со **временем**
  В этих задачах используются такие понятия, относящиеся ко времени, как даты, время и календари. Это также задачи из реальной жизни. Как мы упоминали ранее, решать такие задачи может быть немного более интересно, чем остальные. Некоторые из этих задач будет гораздо легче решить, если вы узнаете больше про класс в Java, названный GregorianCalendar, так как он имеет много библиотечных функций, работающих со временем.
- Задачи – **«пожиратели времени»**
  Это специальные задачи, которые составлены с таким расчетом, чтобы сделать решение долгим и трудным. Такие задачи, если они будут включены в программу олимпиады по программированию, определят команду, в состав которой входит наиболее *эффективный* программист – человек, который может реализовать сложные, но правильные решения в условиях ограниченного времени. Тренеры команды должны по возможности добавлять подобные задачи в свои учебные программы.
- Задачи Ad Hoc в **других главах**
  Множество иных специальных задач мы перенесли в другие главы, поскольку для их решения потребуются знания, превосходящие элементарные навыки программирования. К ним относятся:
  - задачи Ad Hoc, связанные с использованием базовых линейных структур данных (особенно массивов), перечислены в разделе 2.2;
  - задачи Ad Hoc, связанные с математическими вычислениями, перечислены в разделе 5.2;
  - задачи Ad Hoc, связанные с обработкой строк, перечислены в разделе 6.3;
  - задачи Ad Hoc, связанные с геометрией, перечислены в разделе 7.2;
  - задачи Ad Hoc, перечисленные в главе 9.

Советы: решив ряд программных задач, вы начнете реализовывать шаблоны в своих решениях. Некоторые конструкции достаточно часто используются в олимпиадном программировании, и для них особенно полезно использовать сокращения. С точки зрения C/C++, такими конструкциями могут быть: библиотеки для включения (cstdio, cmath, cstring и т. д.), сокращенные наименования типов данных (`ii`, `vii`, `vi` и т. д.), основные процедуры ввода-вывода (freopen, формат многократного ввода и т. д.), макросы цикла (например, `#define REP (i, a, b) for (int i = int(a); i <= int(b); i++)` и т. д.) и некоторые другие. Конкурентоспособный программист, использующий C/C++, может сохранить их в заголовочном файле, например «Competition.h». Если использовать такой заголовочный файл, то решение каждой задачи будет начинаться с простого `#include <Competition.h>`. Однако ограничьте использование этих советов лишь решением задач на соревнованиях по программированию и не распространяйте подобные практики за пределами соревнований – особенно в индустрии разработки программного обеспечения.

Упражнения по программированию, относящиеся к решению задач Ad Hoc:

- Игры (карты)
  1. UVa 00162 – Beggar My Neighbor (симулятор карточной игры; простая задача)
  2. **UVa 00462 – Bridge Hand Evaluator** * (симуляция; карты)
  3. UVa 00555 – Bridge Hands (карточная игра)
  4. UVa 10205 – Stack 'em Up (карточная игра)
  5. *UVa 10315 – Poker Hands* (утомительная задача)
  6. **UVa 10646 – What is the Card?** * (перемешать карты по некоторому правилу, затем получить определенную карту)
  7. UVa 11225 – Tarot scores (еще одна карточная игра)
  8. UVa 11678 – Card's Exchange (на самом деле просто задача, требующая работы с массивом)
  9. ***UVa 12247 – Jollo*** * (интересная карточная игра; простая, но вам потребуется хорошее знание логики, чтобы составить правильные тесты)

- Игры (шахматы)
  1. UVa 00255 – Correct Move (проверьте правильность шахматных ходов)
  2. **UVa 00278 – Chess** * (для шахмат существует специальная формула, выражаемая в аналитическом виде)
  3. **UVa 00696 – How Many Knights** * (специальная задача, шахматы)
  4. UVa 10196 – Check The Check (специальная задача, шахматы, утомительная)
  5. **UVa 10284 – Chessboard in FEN** * (нотация Форсайта–Эдвардса – это стандартное обозначение для описания позиций доски в шахматной игре)
  6. UVa 10849 – Move the bishop (шахматы)
  7. UVa 11494 – Queen (специальная задача, шахматы)

- Прочие игры, легкие
  1. UVa 00340 – Master-Mind Hints (определить сильные и слабые совпадения)
  2. **UVa 00489 – Hangman Judge** * (просто сделайте то, что требуется в задаче)
  3. UVa 00947 – Master Mind Helper (задача похожа на UVa 340)
  4. **UVa 10189 – Minesweeper** * (симуляция минера, аналогично UVa 10279)
  5. UVa 10279 – Mine Sweeper (используйте двумерный массив; аналогично UVa 10189)
  6. UVa 10409 – Die Game (просто смоделируйте движения игральных костей)
  7. UVa 10530 – Guessing Game (используйте массив флагов размерности 1D)

8. **UVa 11459 – Snakes and Ladders** * (смоделируйте это, аналогично UVa 647)
9. *UVa 12239 – Bingo* (попробуйте все 90^2 пар, посмотрите, все ли цифры в [0..*N*] в них присутствуют)

- Прочие игры, сложные (или более трудоемкие)
  1. UVa 00114 – Simulation Wizardry (имитация работы автомата для игры в пинбол)
  2. UVa 00141 – The Spot Game (симуляция, проверка узоров)
  3. UVa 00220 – Othello (следуйте правилам игры; немного нудная задача)
  4. UVa 00227 – Puzzle (разбор и анализ входных данных, манипулирование массивом)
  5. UVa 00232 – Crossword Answers (сложная задача на работу с массивами)
  6. UVa 00339 – SameGame Simulation (см. описание задачи)
  7. UVa 00379 – HI-Q (см. описание задачи)
  8. **UVa 00584 – Bowling** * (моделирование, игры, понимание прочитанного)
  9. UVa 00647 – Chutes and Ladders (детская настольная игра, также см. UVa 11459)
  10. UVa 10363 – Tic Tac Toe (проверить правильность игры в крестики-нолики, хитрая задача)
  11. **UVa 10443 – Rock, Scissors, Paper** * (работа с двумерными массивами)
  12. **UVa 10813 – Traditional BINGO** * (прочитайте описание и сделайте то, что требуется в задаче)
  13. UVa 10903 – Rock-Paper-Scissors (подсчитайте выигрыши + потери, выведите среднее количество выигрышей)

- Палиндромы
  1. UVa 00353 – Pesky Palindromes (перебор всех подстрок)
  2. **UVa 00401 – Palindromes** * (простая проверка палиндрома)
  3. UVa 10018 – Reverse and Add (специальная проверка, математика, проверка палиндрома)
  4. **UVa 10945 – Mother Bear** * (палиндром)
  5. **UVa 11221 – Magic Square Palindrome** * (мы будем иметь дело с матрицей)
  6. UVa 11309 – Counting Chaos (проверка палиндрома)

- Анаграммы
  1. UVa 00148 – Anagram Checker (использует поиск в обратном порядке)
  2. **UVa 00156 – Ananagram** * (решение будет проще, если использовать `algorithm::sort`)
  3. **UVa 00195 – Anagram** * (решение будет проще, если использовать `algorithm::next permutation`)
  4. **UVa 00454 – Anagrams** * (задачи на анаграммы)
  5. UVa 00630 – Anagrams (II) (специальные задачи, работа со строками)

6. UVa 00642 – Word Amalgamation (просмотрите небольшой словарь, чтобы найти список возможных анаграмм)
7. UVa 10098 – Generating Fast, Sorted… (очень похоже на UVa 195)

- Интересные задачи из реальной жизни, более простые
  1. **UVa 00161 – Traffic Lights *** (это типичная ситуация на дороге)
  2. UVa 00187 – Transaction Processing (проблема учета)
  3. UVa 00362 – 18,000 Seconds Remaining (типичная ситуация загрузки файла)
  4. **UVa 00637 – Booklet Printing *** (приложение в программном обеспечении драйвера принтера)
  5. *UVa 00857 – Quantiser* (MIDI, приложение в компьютерной музыке)
  6. UVa 10082 – WERTYU (иногда встречается такая опечатка)
  7. UVa 10191 – Longest Nap (вы можете использовать результат в своем расписании)
  8. UVa 10528 – Major Scales (все, что нужно знать из сольфеджио, – в описании задачи)
  9. UVa 10554 – Calories from Fat (вас интересует ваш вес?)
  10. **UVa 10812 – Beat the Spread *** (обращайте особое вниание на граничные случаи!)
  11. UVa 11530 – SMS Typing (пользователи мобильных телефонов сталкиваются с этой проблемой каждый день)
  12. UVa 11945 – Financial Management (немного отформатировать выходные данные)
  13. UVa 11984 – A Change in Thermal Unit (преобразование °F в °C и наоборот)
  14. UVa 12195 – Jingle Composing (посчитайте количество правильных тактов)
  15. UVa 12555 – Baby Me (один из первых вопросов, которые задают, когда рождается новый ребенок; требуется небольшая обработка входных данных)

- Интересные задачи из реальной жизни, более сложные (или более утомительные)
  1. UVa 00139 – Telephone Tangles (рассчитайте телефонный счет; работа со строками)
  2. UVa 00145 – Gondwanaland Telecom (аналогично UVA 139)
  3. *UVa 00333 – Recognizing Good ISBNs* (примечание: у этой задачи есть «глючные» тестовые данные с пустыми строками, которые потенциально могут вызвать множество ошибок, отнесенных к ошибкам представления (PE))
  4. UVa 00346 – Getting Chorded (музыкальный аккорд, мажор/минор)
  5. **UVa 00403 – Postscript *** (эмуляция драйвера принтера, утомительно)
  6. *UVa 00447 – Population Explosion* (имитационная модель жизни)
  7. UVa 00448 – OOPS (утомительное «шестнадцатеричное» преобразование в «язык ассемблера»)

8. *UVa 00449 – Majoring in Scales* (вам будет проще, если у вас есть музыкальное образование)
9. UVa 00457 – Linear Cellular Automata (упрощенная симуляция игры «Жизнь»; идея похожа на UVa 447; поищите в интернете этот термин)
10. UVa 00538 – Balancing Bank Accounts (предпосылка задачи вполне реальна)
11. **UVa 00608 – Counterfeit Dollar** * (классическая задача)
12. UVa 00706 – LC-Display (что мы видим на старом цифровом дисплее)
13. ***UVa 01061 – Consanguine Calculations*** * (LA 3736 –заключительные всемирные соревнования, Токио'07; кровное родство = кровь; в этой задаче задаются возможные комбинации типов крови и резус-фактора; задачу можно решить путем перебора всех восьми возможных типов крови + резус, учитывая информацию, приведенную в описании задачи)
14. UVa 10415 – Eb Alto Saxophone Player (о музыкальных инструментах)
15. UVa 10659 – Fitting Text into Slides (это делают обычные программы для подготовки презентаций)
16. UVa 11223 – O: dah, dah, dah (утомительное преобразование кода Морзе)
17. UVa 11743 – Credit Check (алгоритм Луна для вычисления контрольной цифры номера пластиковой карты; поищите в интернете, чтобы узнать о нем подробнее)
18. *UVa 12342 – Tax Calculator* (расчет налогов может быть довольно сложным)

- Время

1. UVa 00170 – Clock Patience (симуляция, время)
2. UVa 00300 – Maya Calendar (задача Ad Hoc, время)
3. **UVa 00579 – Clock Hands** * (задача Ad Hoc, время)
4. **UVa 00893 – Y3K** * (используйте Java GregorianCalendar; аналогично UVa 11356)
5. UVa 10070 – Leap Year or Not Leap... (нечто большее, чем обычная проверка на високосные годы)
6. *UVa 10339 – Watching Watches* (нужно найти формулу)
7. UVa 10371 – Time Zones (просто сделайте то, что требуется в описании задачи)
8. UVa 10683 – The decadary watch (простой алгоритм преобразования часов)
9. UVa 11219 – How old are you? (обращайте особое вниание на граничные случаи!)
10. UVa 11356 – Dates (очень простое решение, если использовать Java GregorianCalendar)
11. UVa 11650 – Mirror Clock (тут потребуется математика)
12. UVa 11677 – Alarm Clock (аналогично UVa 11650)
13. **UVa 11947 – Cancer or Scorpio** * (задача решается проще, если использовать Java GregorianCalendar)

14. UVa 11958 – Coming Home (будьте осторожны с «после полуночи»)
15. UVa 12019 – Doom's Day Algorithm (григорианский календарь; получить DAY_OF_WEEK (день недели))
16. UVa 12136 – Schedule of a Married Man (LA 4202, Дакка'08; проверьте время)
17. *UVa 12148 – Electricity* (задачу легко решить, используя григорианский календарь; используйте метод «add», чтобы добавить один день к предыдущей дате и посмотреть, совпадает ли она с текущей датой)
18. *UVa 12439 – February 29* (включение-исключение; множество случаев, когда значения выходят за допустимые пределы; будьте внимательны)
19. *UVa 12531 – Hours and Minutes* (углы между двумя стрелками часов)

- Задачи – «пожиратели времени»

1. UVa 00144 – Student Grants (симуляция)
2. *UVa 00214 – Code Generation* (просто смоделируйте процесс; будьте осторожны с вычитанием (–), делением (/) и инвертированием (@); утомительная задача)
3. UVa 00335 – Processing MX Records (симуляция)
4. UVa 00337 – Interpreting Control… (симуляция, зависящая от выходных данных)
5. UVa 00349 – Transferable Voting (II) (симуляция)
6. UVa 00381 – Making the Grade (симуляция)
7. UVa 00405 – Message Routing (симуляция)
8. **UVa 00556 – Amazing *** (симуляция)
9. *UVa 00603 – Parking Lot* (смоделируйте процесс)
10. *UVa 00830 – Shark* (очень трудно получить оценку AC, одна небольшая ошибка = WA)
11. *UVa 00945 – Loading a Cargo Ship* (смоделируйте данный процесс погрузки груза)
12. UVa 10033 – Interpreter (специальная задача, моделирование)
13. *UVa 10134 – AutoFish* (нужно быть очень осторожным, обращая внимание на тонкости)
14. UVa 10142 – Australian Voting (симуляция)
15. UVa 10188 – Automated Judge Script (симуляция)
16. UVa 10267 – Graphical Editor (симуляция)
17. *UVa 10961 – Chasing After Don Giovanni* (утомительная задача на моделирование)
18. UVa 11140 – Little Ali's Little Brother (специальная задача)
19. UVa 11717 – Energy Saving Micro… (хитрое моделирование)
20. **UVa 12060 – All Integer Average *** (LA 3012, Дакка'04, формат выходных данных)
21. **UVa 12085 – Mobile Casanova *** (LA 2189, Дакка'06; следите за тем, чтобы не получить оценку «PE»)
22. *UVa 12608 – Garbage Collection* (симуляция с несколькими случаями, когда значения выходят за допустимые пределы)

1.5. Решения упражнений, не помеченных звездочкой

Упражнение 1.1.1. Простой тестовый пример, позволяющий исключить использование «жадных» алгоритмов, – это $N = 2$, {(2, 0), (2, 1), (0, 0), (4, 0)}. «Жадный» алгоритм будет неправильно объединять {(2, 0), (2, 1)} и {(0, 0), (4, 0)} (при этом значение cost = 5.000), в то время как оптимальным решением является пара {(0, 0) , (2, 0)} и {(2, 1), (4, 0)} (значение cost = 4,236).

Упражнение 1.1.2. Для наивного полного перебора, подобного тому, который описан в тексте, нужно до ${}_{16}C_2 \times {}_{14}C_2 \times \ldots \times {}_2C_2$ операций; для самого большого тестового примера с $N = 8$ – слишком большой набор данных.

Тем не менее существуют способы сокращения пространства поиска, чтобы полный перебор все еще мог быть применим.

В качестве дополнительного упражнения попробуйте решить задачу из **упражнения 1.1.3***!

Упражнение 1.2.1. Заполненная табл. 1.3 приведена ниже.

№ UVa	Название	Категория задачи	Подсказка
10360	Rat Attack	Полный перебор или динамическое программирование	Раздел 3.2
10341	Solve It	«Разделяй и властвуй» (метод деления пополам)	Раздел 3.3
11292	Dragon of Loowater	«Жадный» алгоритм (не классический)	Раздел 3.4
11450	Wedding Shopping	Динамическое программирование (не классическое)	Раздел 3.5
10911	Forming Quiz Teams	Динамическое программирование с использованием битовой маски (не классическое)	Раздел 8.3.1
11635	Hotel Booking	Граф (декомпозиция: алгоритм Дейкстры + поиск в ширину)	Раздел 8.4
11506	Angry Programmer	Граф (теорема Форда–Фалкерсона о максимальном потоке и минимальном разрезе)	Раздел 4.6
10243	Fire! Fire!! Fire!!!	Динамическое программирование на деревьях (минимальное покрытие вершин)	Раздел 4.7.1
10717	Mint	Декомпозиция: полный перебор + математика	Раздел 8.4
11512	GATTACA	Строки (массив суффиксов, наибольший общий префикс, самая длинная повторяющаяся подстрока)	Раздел 6.6
10065	Useless Tile Packers	Геометрия (выпуклая оболочка + площадь многоугольника)	Раздел 7.3.7

Упражнение 1.2.2. Вы:

1) (b) используете очередь с приоритетом (кучу) (раздел 2.3);
2) (b) используете запрос суммы на отрезке (2D) (раздел 3.5.2);
3) если список L является статическим, то (b) простой массив, предварительно обработанный с использованием методов динамического программирования (см. разделы 2.2 и 3.5). Если список L является динамическим, то лучший ответ – (f) двоичное индексированное дерево (дерево Фенвика) (см. раздел 2.4.4). Его проще реализовать, чем (e) дерево отрезков;

4) (a) да, полный перебор подходит (раздел 3.2);
5) (a) динамическое программирование с временной сложностью $O(V + E)$ (разделы 3.5, 4.2.5 и 4.7.1). Однако ответ (c) алгоритм Дейкстры (временная сложность $O((V + E)\log V)$ также возможен, поскольку дополнительный коэффициент $O(\log V)$ все еще «мал» для значений V, не превышающих 100K;
6) (a) решето Эратосфена (раздел 5.5.1);
7) (b) наивный подход, описанный выше, не будет работать. Надо разложить $n!$ и m на простые множители и посмотреть, содержатся ли (простые) множители m в разложении на множители $n!$ (раздел 5.5.5);
8) (b) наивный подход, приведенный выше, не будет работать. Нужно найти другой путь. Сначала найдите выпуклую оболочку из N точек с временной сложностью $O(n \log n)$ (раздел 7.3.7). Пусть количество точек в $CH(S) = k$. Поскольку точки разбросаны случайным образом, k будет намного меньше N. Затем найдите две самые дальние точки, рассмотрев все пары точек в $CH(S)$ с временной сложностью $O(k^2)$;
9) (b) наивный подход будет работать слишком медленно. Используйте алгоритм КМП или массив суффиксов (раздел 6.4 или 6.6).

Упражнение 1.2.3. Код Java приведен ниже:

```
// Java-код для задачи 1 (предполагается, что все необходимые
// операции импорта выполнены)
class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    double d = sc.nextDouble();
    System.out.printf("%7.3f\n", d);                        // да, в Java тоже есть printf!
} }
```

```
// C++ код для задачи 2 (предполагается, что все необходимые
// директивы include выполнены)
int main() {
  double pi = 2 * acos(0.0);                      // это более точный способ вычисления pi
  int n; scanf("%d", &n);
  printf("%.*lf\n", n, pi);                      // это способ манипулировать шириной поля
}
```

```
// Java-код для задачи 3 (предполагается, что все необходимые
// операции импорта выполнены)
class Main {
  public static void main(String[] args) {
    String[] names = new String[]
      { "", "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
    Calendar calendar = new GregorianCalendar(2010, 7, 9);                // 9 August 2010
                              // обратите внимание, что нумерация месяцев начинается с 0,
                                          // поэтому нам нужно использовать 7 вместо 8
    System.out.println(names[calendar.get(Calendar.DAY_OF_WEEK)]);                // "Wed"
} }
```

```
// C++ код для задачи 4 (предполагается, что все необходимые
// директивы include выполнены)
```

```
#define ALL(x) x.begin(), x.end()
#define UNIQUE(c) (c).resize(unique(ALL(c)) - (c).begin())

int main() {
  int a[] = {1, 2, 2, 2, 3, 3, 2, 2, 1};
  vector<int> v(a, a + 9);
  sort(ALL(v)); UNIQUE(v);
  for (int i = 0; i < (int)v.size(); i++) printf("%d\n", v[i]);
}
```

```
// C++ код для задачи 5 (предполагается, что все необходимые
// директивы include выполнены)
typedef pair<int, int> ii;            // будем использовать естественный порядок сортировки
typedef pair<int, ii> iii;       // примитивных типов данных, которые мы попарно объединили

int main() {
  iii A = make_pair(ii(5, 24), -1982);                  // заменим порядок ДД / ММ / ГГГГ
  iii B = make_pair(ii(5, 24), -1980);                                    // на ММ, DD,
  iii C = make_pair(ii(11, 13), -1983);      // затем используем ОБРАТНУЮ ПЕРЕСТАНОВКУ YYYY
  vector<iii> birthdays;
  birthdays.push_back(A); birthdays.push_back(B); birthdays.push_back(C);
  sort(birthdays.begin(), birthdays.end());                                // вот и все :)
}
```

```
// C++ код для задачи 6 (предполагается, что все необходимые
// директивы include выполнены)
int main() {
  int n = 5, L[] = {10, 7, 5, 20, 8}, v = 7;
  sort(L, L + n);
  printf("%d\n", binary_search(L, L + n, v));
}
```

```
// C++ код для задачи 7 (предполагается, что все необходимые
// директивы include выполнены)
int main() {
  int p[10], N = 10; for (int i = 0; i < N; i++) p[i] = i;
  do {
    for (int i = 0; i < N; i++) printf("%c ", 'A' + p[i]);
    printf("\n");
  }
  while (next_permutation(p, p + N));
}
```

```
// C++ код для задачи 8 (предполагается, что все необходимые
// директивы include выполнены)
int main() {
  int p[20], N = 20;
  for (int i = 0; i < N; i++) p[i] = i;
  for (int i = 0; i < (1 << N); i++) {
    for (int j = 0; j < N; j++)
      if (i & (1 << j))                                          // если бит j включен
        printf("%d ", p[j]);                          // элемент является частью множества
    printf("\n");
} }
```

```
// Java-код для задачи 9 (предполагается, что все необходимые
// операции импорта выполнены)
class Main {
  public static void main(String[] args) {
    String str = "FF"; int X = 16, Y = 10;
    System.out.println(new BigInteger(str, X).toString(Y));
} }
```

```
// Java-код для задачи 10 (предполагается, что все необходимые
// операции импорта выполнены)
class Main {
  public static void main(String[] args) {
    String S = "line: a70 and z72 will be replaced, aa24 and a872 will not";
    System.out.println(S.replaceAll("(^| )+[a-z][0-9][0-9]( |$)+", " *** "));
} }
```

```
// Java-код для задачи 11 (предполагается, что все необходимые
// операции импорта выполнены)
class Main {
  public static void main(String[] args) throws Exception {
    ScriptEngineManager mgr = new ScriptEngineManager();
    ScriptEngine engine = mgr.getEngineByName("JavaScript");              // "жульничество"
    Scanner sc = new Scanner(System.in);
    while (sc.hasNextLine()) System.out.println(engine.eval(sc.nextLine()));
} }
```

Упражнение 1.2.4. Ситуационная ориентация (соображения авторов приведены в скобках)

1. Вы получаете оценку тестирующей системы «неверный ответ» (WA) для очень простой задачи. Что вы будете делать:
   a) не станете решать эту задачу и переключитесь на следующую (**плохо, ваша команда проиграет**);
   b) попытаетесь улучшить производительность вашего решения (оптимизация кода / лучший алгоритм) (**бесполезно**);
   c) создадите сложные тестовые примеры, чтобы найти ошибку (**наиболее логичный ответ**);
   d) (в командных соревнованиях) попросите своего товарища по команде заново решить задачу (**это может быть реалистичным вариантом, так как вы могли не понять задачу и неправильно решить ее. В этом случае вы не должны подробно объяснять задачу своему партнеру по команде, который повторно решит ее. Тем не менее ваша команда потеряет драгоценное время**).
2. Вы получаете оценку тестирующей системы «превышение лимита времени» (TLE) для представленного решения $O(N^3)$.
   Тем не менее максимальное значение N составляет всего 100. Что вы будете делать:
   a) не станете решать эту задачу и переключитесь на следующую (**плохо, ваша команда проиграет**);
   b) попытаетесь улучшить производительность вашего решения (оптимизация кода / лучший алгоритм) (**плохо, мы не должны получить**

оценку TLE для алгоритма с временной сложностью $O(N^3)$ для случая $N \leqslant 400$);

   c) создадите сложные тестовые примеры, чтобы найти ошибку (**это правильный ответ – может быть, в некоторых тестах ваша программа входит в бесконечный цикл**).

3. Вернитесь к вопросу 2: что вы будете делать в случае, если максимальное значение N составляет 100 000? (**Если $N > 400$, у вас может не быть иного выбора, кроме как повысить производительность текущего алгоритма или использовать другой, более быстрый алгоритм.**)
4. Еще раз вернитесь к вопросу 2. Что вы будете делать в случае, если максимальное значение N равно 1000, выходной результат зависит только от размера входного N и у вас еще остается четыре часа времени до конца соревнований? (**Если выходные данные зависят только от N, вы *можете* заранее рассчитать все возможные решения, запустив алгоритм $O(N^3)$ в фоновом режиме и позволив партнеру использовать компьютер первым. Как только ваше решение $O(N^3)$ завершит работу, вы получите все ответы. Вместо этого отправьте ответ $O(1)$, если он не превышает «ограничение размера исходного кода», установленное тестирующей системой.**)
5. Вы получаете вердикт тестирующей системы «ошибка времени выполнения» (RTE). Ваш код (на ваш взгляд) отлично работает на вашем компьютере.

   Что вы должны сделать? (**Наиболее распространенными причинами появления ошибки RTE обычно являются слишком малые размеры массивов или переполнение стека / ошибки бесконечной рекурсии. Разработайте тестовые примеры, которые помогут обнаружить эти ошибки в вашем коде.**)
6. Через тридцать минут после начала соревнования вы посмотрели на текущую таблицу результатов. Множество других команд, которые решили задачу X, которую ваша команда не пыталась решить. Что вы будете делать? (**Один из членов команды должен немедленно решить задачу X, поскольку это может быть относительно легко. Такая ситуация – плохая новость для вашей команды, поскольку это серьезное препятствие на пути к высокому месту в турнирной таблице на олимпиаде.**)
7. В середине соревнования вы смотрите на таблицу лидеров. Ведущая команда (предположим, что это не ваша команда) только что решила задачу Y. Что вы будете делать? (**Если ваша команда не «задает темп», то неплохой идеей будет «игнорировать» действия ведущей команды и вместо этого сосредоточиться на решении задач, которые ваша команда определила как «решаемые». К середине соревнования ваша команда должна ознакомиться со всеми задачами из множества предложенных задач и примерно определить задачи, которые можно решить, исходя из текущих возможностей вашей команды.**)
8. Ваша команда потратила два часа на сложную задачу. Вы отправили несколько вариантов решения, выполненных разными членами команды. Все решения были оценены как неверные.

Вы понятия не имеете, что не так. Что вы будете делать? (**Настало время отказаться от решения этой задачи. Не перегружайте компьютер, пусть ваш товарищ по команде решит еще одну задачу. Либо ваша команда действительно неправильно поняла задачу, либо – хотя такое случается очень редко – решение автора на самом деле неверно. В любом случае, это не очень хорошая ситуация для вашей команды.**)

9. До окончания соревнования остается один час. У вас есть одна оценка «неверный ответ» (WA) для выполненной задачи и одна свежая идея для решения другой задачи. Что вы (или ваша команда) будете делать (**в шахматной терминологии это называется «эндшпиль»**):
   a) оставите нерешенной задачу, получившую оценку WA, и переключитесь на другую задачу, попытавшись ее решить (**ОК для индивидуальных соревнований, таких как IOI**);
   b) будете настаивать на том, что вам необходимо отладить код, получивший оценку WA. Вам не хватает времени, чтобы начать работать над решением новой задачи (**если идея для решения другой задачи предполагает написание сложного кода, требующего больших усилий, то решение сосредоточиться на коде, получившем оценку WA, может быть неплохой идеей: в противном случае вы рискуете получить в конце соревнований два неполных / не получивших «проходную» оценку AC решения**);
   c) (в ICPC) распечатаете код задачи, получившей оценку WA. Попросите двух других членов команды изучить его, пока вы переключаетесь на другую задачу, пытаясь решить еще две задачи (**если решение для другой задачи может быть написано менее чем за 30 минут, то реализуйте его, пока ваши товарищи по команде изучают распечатанный код, пытаясь найти ошибку в коде для задачи, получившей оценку WA**).

1.6. Примечания к главе 1

К этой главе, а также последующим главам существует множество дополнительных материалов: учебников (см. рис. 1.4) и интернет-ресурсов. Ниже мы приводим несколько ссылок.

- Чтобы улучшить свои навыки набора текста, как упомянуто в нашем совете 1, вы можете поиграть в игры, развивающие навыки набора текста, доступные в интернете.
- Совет 2 взят из текста введения на сайте подготовки к олимпиадам USACO [48].
- Более подробную информацию для совета 3 можно найти во многих книгах по информатике (например, главы 1–5, 17 из [7]).
- Онлайн-ссылки для совета 4:
  - http://www.cppreference.com и http://www.sgi.com/tech/stl/ для библиотеки STL в C++;
  - http://docs.oracle.com/javase/7/docs/api/ для API Java.

Рис. 1.4 ❖ Некоторые источники,
которые вдохновили авторов на написание этой книги

Вам не нужно запоминать все библиотечные функции, но полезно запомнить функции, которые вы часто используете.

- Чтобы узнать больше о лучших практиках тестирования (совет 5), возможно, стоит прочитать несколько книг по разработке программного обеспечения.
- Есть много других сайтов, на которых вы можете проверить свои решения олимпиадных задач онлайн, помимо тех, которые упомянуты в совете 6, например:
  - Codeforces, http://codeforces.com/;
  - Онлайн-арбитр Пекинского университета (POJ), http://poj.org;
  - Онлайн-арбитр университета Чжэцзяна (ZOJ), http://acm.zju.edu.cn;
  - Онлайн-арбитр Университета Тяньцзиня, http://acm.tju.edu.cn/toj;
  - Онлайн-арбитр Уральского государственного университета, http://acm.timus.ru;
  - URI Online Judge, http://www.urionlinejudge.edu.br и т. д.
- О командных соревнованиях (совет 7) читайте в [16].

В этой главе мы познакомили вас с олимпиадным программированием. Тем не менее конкурентоспособный программист должен быть способен решать не только задачи Ad Hoc в соревновании по программированию. Мы надеем-

ся, что вы будете получать удовольствие и подпитывать свой энтузиазм, читая и изучая новые темы в *следующих* главах этой книги. Прочитав книгу до конца, перечитайте ее еще раз. Во второй раз попытайтесь решить ≈ 238 письменных упражнений и ≈ 1675 задач по программированию.

Таблица 1.5. Статистические данные, относящиеся к различным изданиям книги

Параметр	**Первое издание**	**Второе издание**	**Третье издание**
Число страниц	13	19 (+46 %)	32 (+68 %)
Письменные упражнения	4	4	6 + 3* = 9 (+125 %)
Задачи по программированию	34	160 (+371 %)	173 (+ 8%)

Глава 2

Структуры данных и библиотеки

Если я видел дальше, то лишь потому, что я стою на плечах гигантов.

– Исаак Ньютон

2.1. Общий обзор и мотивация

Структура данных (data structure, DS) – это средство хранения и организации данных. Различные структуры данных имеют свои сильные стороны. Поэтому при разработке алгоритма важно выбрать тот, который обеспечивает эффективные операции вставки, поиска, удаления, запроса и/или обновления, в зависимости от ваших потребностей. Хотя структура данных сама по себе не решает задачи на соревнованиях по программированию (а алгоритм, работающий с ней, решает), использование подходящей эффективной структуры данных для предложенной задачи может дать существенный выигрыш по времени выполнения вашего кода. Между оценкой «принято» (AC) и «превышение лимита времени» (TLE) для вашей задачи – огромная пропасть, преодолеть которую можно, используя эффективную структуру данных. Существует много способов организовать одни и те же данные, и иногда один способ оказывается лучше, чем другой, в определенном контексте. Мы рассмотрим несколько примеров в этой главе. Глубокое знакомство со структурами данных и библиотеками, обсуждаемыми в этой главе, крайне важно для понимания алгоритмов, которые используют их в последующих главах.

Как указано в предисловии к этой книге, мы **предполагаем**, что вы *знакомы* с основными структурами данных, перечисленными в разделах 2.2–2.3, и, следовательно, мы не будем рассматривать их в этой книге. Вместо этого мы просто подчеркнем тот факт, что существуют встроенные реализации для этих элементарных структур данных в библиотеке STL C++ и Java API[1]. Если

[1] Даже в этом третьем издании мы по-прежнему в основном используем код C++ для иллюстрации реализации решений. Эквивалентные примеры на Java можно найти на сайте, где размещены дополнительные материалы для этой книги.

вы чувствуете, что слабо знакомы с какими-либо терминами или структурами данных, упомянутыми в разделах 2.2–2.3, найдите эти термины и понятия в различных справочных книгах[1], которые охватывают подобные темы, включая классические книги, такие как «Introduction to Algorithms» («Введение в алгоритмы») [7], «Data Abstraction and Problem Solving» («Абстракция данных и решение задач») [5, 54], «Data Structures and Algorithms» («Структуры данных и алгоритмы») [12] и т. д. Вернитесь к чтению этой книги лишь тогда, когда вы познакомитесь хотя бы с основными понятиями, лежащими в основе этих структур данных.

Обратите внимание, что в олимпиадном программировании вам потребуется лишь знать достаточно об этих структурах данных, чтобы иметь возможность выбирать и использовать правильные структуры данных для каждой конкретной задачи олимпиады. Вы должны понимать сильные и слабые стороны, а также оценивать временную сложность и сложность по памяти типичных структур данных. Теорию, лежащую в их основе, определенно хорошо прочитать, но ее часто можно пропустить, поскольку встроенные библиотеки предоставляют готовые к использованию и надежные реализации сложных структур данных. Это не слишком хорошая практика, однако вы обнаружите, что часто этого бывает достаточно. Многие участники смогли использовать эффективные (с временной сложностью $O(\log n)$ для большинства операций) библиотечные реализации C++ STL `map` (или Java `TreeMap`) для хранения динамических коллекций пар «ключ–данные» без понимания того, что лежащая в основе структура данных представляет собой сбалансированное двоичное дерево поиска, или использовали очередь приоритетов C++ STL `priority_queue` (или `PriorityQueue` в Java) для упорядочения очереди элементов без понимания того, что базовая структура данных представляет собой (*как правило, двоичную*) кучу. Обе эти структуры данных обычно рассматриваются преподавателями на занятиях на первом курсе по информатике.

Эта глава состоит из трех частей. Раздел 2.2 содержит основные линейные структуры данных и основные операции, которые они поддерживают. Раздел 2.3 охватывает базовые нелинейные структуры данных, такие как (сбалансированные) двоичные деревья поиска (Binary Search Trees, BST), (двоичные) кучи и хеш-таблицы, а также основные операции с ними. В разделах 2.2–2.3 приводится краткое обсуждение каждой структуры данных, с акцентом на важные библиотечные процедуры, которые существуют для операций со структурами данных. Тем не менее специальные структуры данных, которые часто используются в соревнованиях по программированию, такие как битовая маска и операции с битами (см. рис. 2.1), обсуждаются более подробно. В разделе 2.4 приводятся структуры данных, для которых не существует встроенной реализации, и, следовательно, от нас потребуется создание наших собственных библиотек. В разделе 2.4 структуры данных рассматриваются более подробно, чем в разделах 2.2–2.3.

---

[1] Материалы в разделе 2.2–2.3, как правило, охватываются учебными планами по структурам данных первого года обучения по специальности «Информатика». Учащимся старших классов, желающим принять участие в IOI, предлагается заняться независимым изучением этих материалов.

Ценные дополнения в этой книге

Поскольку эта глава – первая, в которой углубленно рассматривается тема олимпиадного программирования, мы подчеркнем несколько изменений и дополнений, сделанных в данной книге, которые вы заметите в этой и следующих главах.

Ключевой особенностью нашей книги является сопровождающий ее сборник эффективных, полностью реализованных примеров как на C/C++, так и на Java, который отсутствует во многих других книгах по информатике, останавливающихся на «уровне псевдокода» при демонстрации структур данных и алгоритмов. Эти примеры были в книге с самого первого издания. Важные фрагменты исходного кода были включены в книгу[1], а полный исходный код размещен на сайте sites.google.com/site/stevenhalim/home/material. Ссылка на каждый исходный файл указывается в основном тексте в виде поля, как показано ниже:

Файл исходного кода: *chx_yy_name.cpp/java*

Еще одной сильной стороной этой книги является сборник практических упражнений и задач по программированию (выполнить эти практические упражнения вы можете, зарегистрировавшись на сайте тестирующей системы университета Вальядолида [47]; этот сайт интегрирован с uHunt – приложением для сбора и обработки статистических данных – см. приложение A). В третьем издании мы добавили еще много упражнений. Мы разделили практические задания на неотмеченные и помеченные звездочкой. Практические задания (упражнения) без звездочки предназначены для использования в основном в целях самоконтроля; их решения приведены в конце каждой главы. Помеченные упражнения могут использоваться в качестве дополнительных заданий; мы не предоставляем решения для них, вместо этого мы можем дать некоторые полезные советы.

В третьем издании мы добавили визуализацию[2] для многих структур данных и алгоритмов, описанных в этой книге [27]. Мы считаем, что визуализация будет огромным преимуществом для студентов, читающих эту книгу. На данный момент (24 мая 2013 г.) визуализация размещена на сайте: www.comp.nus.edu.sg/~stevenha/visualization. Ссылка на каждый элемент визуализации включена в текст в виде поля, как показано ниже:

Инструментальное средство визуализации: www.comp.nus.edu.sg/~stevenha/visualization

---

[1] Однако мы решили не включать код, относящийся к разделам 2.2–2.3, в основной текст, потому что он в большинстве случаев «тривиален» для многих читателей, за исключением, возможно, нескольких полезных приемов.

[2] Она создана с использованием HTML5 Canvas и JavaScript.

2.2. Линейные структуры данных – встроенные библиотеки

Структура данных классифицируется как линейная структура данных, если ее элементы образуют линейную последовательность, то есть ее элементы упорядоченно располагаются слева направо (или сверху вниз). Уверенные знания основных линейных структур данных, приведенных ниже, имеют решающее значение на современных олимпиадах по программированию.

- Статический массив (встроенная поддержка в C/C++ и Java)
  Это явно наиболее часто используемая структура данных на олимпиадах по программированию. В тех случаях, когда существует набор последовательных данных, которые должны быть сохранены и впоследствии доступны с использованием индексов, статический массив является наиболее естественной структурой данных. Поскольку максимальный размер входных данных обычно упоминается в формулировке задачи, размер массива можно объявить, используя максимальный размер входных данных с небольшим дополнительным буфером («защитным устройством»), добавленным для безопасности – чтобы избежать ошибки времени выполнения (RTE). Как правило, в соревнованиях по программированию используются одномерные, двумерные и трехмерные массивы – для решения задач редко требуются массивы более высокой размерности. Типичные операции с массивами включают в себя доступ к элементам по их индексам, сортировку элементов, выполнение последовательного просмотра данных или двоичного поиска в отсортированном массиве.
- Динамический массив: `vector` (либо `ArrayList` (более быстрый)) в C++ STL (`Vector` в Java)
  Эта структура данных похожа на статический массив, за исключением того, что динамический массив предназначен и адаптирован для изменения размера массива во время выполнения программы. Вместо массива лучше использовать вектор, если размер последовательности элементов неизвестен во время компиляции. Обычно для повышения производительности мы инициализируем размер (`reserve()` или `resize()`), исходя из предполагаемого размера набора данных. Типичные операции с векторами в C++ STL, используемые в олимпиадном программировании: `push_back()`, `at()`, оператор `[]`, `assign()`, `clear()`, `erase()` и итераторы для обхода содержимого векторов.

Файл исходного кода: *ch2_01_array_vector.cpp/java*

Обсудим две операции, обычно выполняемые над массивами: сортировка и поиск. Эти две операции поддерживаются в C++ и Java.
Существует много алгоритмов сортировки, упомянутых в книгах по информатике [7, 5, 54, 12, 40, 58], например:

1) алгоритмы сортировки, основанные на сравнении (временная сложность $O(n^2)$): сортировка «пузырьком» / сортировка выбором / сортировка вставкой и т. д. Эти алгоритмы (очень) медленные, и их обычно не используют на олимпиадах по программированию, хотя их понимание может помочь вам решить определенные задачи;
2) алгоритмы сортировки на основе сравнения (временная сложность $O(n \log n)$): сортировка слиянием / сортировка кучей / быстрая сортировка и т. д. Эти алгоритмы, как правило, широко используются в олимпиадном программировании, поскольку сложность $O(n \log n)$ является оптимальной для сортировки на основе сравнения. Поэтому в большинстве случаев эти алгоритмы сортировки выполняются на«наилучшее возможное» время (см. ниже специальные алгоритмы сортировки). Кроме того, эти алгоритмы хорошо известны, и, следовательно, нам не нужно «изобретать велосипед»[1] – мы можем просто использовать `sort`, `partial_sort` или `stable_sort` в библиотеке C++ STL `algorithm` (или `Collections.sort` в Java) для обычных задач сортировки. Нам нужно только указать требуемую функцию сравнения, а библиотечные процедуры сделают все остальное;
3) специальные алгоритмы сортировки (временная сложность $O(n)$): сортировка подсчетом / поразрядная сортировка / сортировка группировками и т. д. Хотя эти методы сортировки редко используются, эти специальные алгоритмы полезно знать, поскольку они могут сократить требуемое время сортировки для определенных видов данных. Например, сортировка подсчетом может применяться к целочисленным данным, которые находятся в небольшом диапазоне значений (см. раздел 9.32).

Существует три распространенных метода поиска элемента в массиве:

1) линейный поиск (сложность $O(n)$): рассмотрим каждый элемент от индекса 0 до индекса $n - 1$ (по возможности избегайте этого);
2) двоичный поиск (сложность $O(\log n)$): используйте `lower_bound`, `upper_bound` или `binary_search` в библиотеке C++ STL `algorithm` (или `Collections.binarySearch` в Java). Если входной массив не отсортирован, необходимо отсортировать массив хотя бы один раз (используя один из описанных выше алгоритмов сортировки $O(n \log n)$) перед выполнением одного двоичного поиска (или выполнением операции двоичного поиска несколько раз);
3) поиск с хешированием (сложность $O(1)$): это полезный метод для использования, если требуется быстрый доступ к известным значениям. Если выбрана подходящая хеш-функция, вероятность возникновения коллизии пренебрежимо мала. Тем не менее эта техника использует-

[1] Однако иногда нам нужно «изобретать велосипед» для решения определенных задач, связанных с сортировкой, например для инвертированного индекса (см. раздел 9.14).

ся редко, и мы можем обойтись без нее[1] для большинства (олимпиадных) задач.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/sorting.html

Файл исходного кода: *ch2_02_algorithm_collections.cpp/java*

- Массив логических значений: `bitset` в C++ STL (`BitSet` в Java)
  Если наш массив должен содержать только логические значения (1/истина и 0/ложь), мы можем использовать альтернативную структуру данных, отличную от массива, – `bitset` (набор битов) в C++ STL. Структура `bitset` поддерживает полезные операции, такие как `reset()`, `set()`, оператор `[]` и `test()`.

Файл исходного кода: *ch5_06_primes.cpp/java, also see Section 5.5.1*

- Битовые маски. Небольшие по объему наборы логических значений (встроенная поддержка в C/C++ / Java)
  Целое число сохраняется в памяти компьютера в виде последовательности (или строки) битов. Таким образом, мы можем использовать целые числа для эффективного представления небольшого множества логических переменных. В этом случае все операции над множествами включают только побитовую обработку для соответствующего целого числа; такой способ *намного более эффективен* по сравнению с операциями в C++ STL `vector<bool>`, `bitset` или `set<int>`. Скорость, которую могут дать эти операции, важна в олимпиадном программировании. *Некоторые* важные операции, которые используются в этой книге, проиллюстрированы ниже.

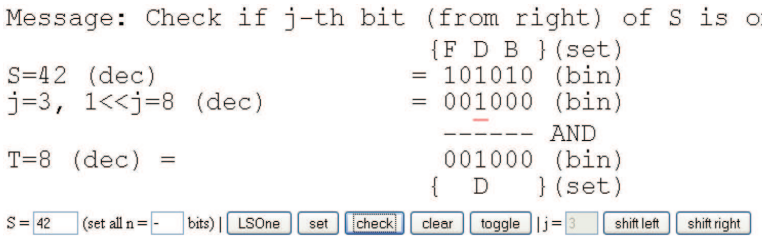


Рис. 2.1 ❖ Визуализация битовой маски

[1] Однако вопросы о хешировании часто появляются на собеседовании для работников IT.

1. Представление: 32-битное (или 64-битное) знаковое целое число для множества элементов размерностью до 32 (или 64)[1]. Во всех примерах ниже используется 32-разрядное знаковое целое число, обозначаемое как *S*.

```
Пример:                                5| 4| 3| 2| 1| 0 <- значения индекса
                                                          начинаются с 0,
                                                          увеличиваются справа
                                                          налево
                                     32|16| 8| 4| 2| 1 <- степени числа 2
S = 34 (в десятичной системе 10) = 1| 0| 0| 0| 1| 0 (в двоичной системе)
                                      F| E| D| C| B| A <- альтернативные алфавитные
                                                          обозначения
```

   В приведенном выше примере целое число *S* = 34 (или 100010 в двоичном представлении) также представляет небольшое множество {1, 5} со схемой индексации, начинающейся с 0 и организованной в порядке увеличения значимости цифр (или {B, F} с использованием альтернативного алфавитного обозначения): в *S* включены второй и шестой биты (считая справа).
2. Чтобы умножить или разделить целое число на 2, нужно только сдвинуть биты, представляющие это целое число, влево или вправо соответственно. Эта операция (особенно операция сдвига влево) показана на следующих нескольких примерах ниже. Обратите внимание, что «отбрасывание» крайнего бита в операции сдвига вправо автоматически округляет вниз результат деления на 2, например: 17/2 = 8.

```
S                 = 34 (десятичное) =  100010 (двоичное)
S = S << 1 = S * 2 = 68 (десятичное) = 1000100 (двоичное)
S = S >> 2 = S / 4 = 17 (десятичное) =   10001 (двоичное)
S = S >> 1 = S / 2 =  8 (десятичное) =    1000 (двоичное) <- МЗБ "отброшен"
                                                  (МЗБ = младший значащий бит)
```

3. Чтобы включить (установить в 1) *j*-й элемент множества (со схемой индексации, начинающейся с 0), используйте побитовое «ИЛИ»: `S | = (1 << j)`.

```
S = 34 (десятичное) = 100010 (двоичное)
j = 3, 1 << j       = 001000 <- бит '1' смещен влево 3 раза
                      -------- OR (результат 'true', если один из битов 'true')
S = 42 (десятичное) = 101010 (двоичное) // обновляем S до нового значения 42
```

4. Чтобы проверить, равен ли единице *j*-й элемент множества, используйте побитовое «И»: `T = S & (1 << j)`.
   Если `T = 0`, то *j*-й элемент набора равен нулю (выключен). Если `T! = 0` (точнее, `T = (1 << j)`), то *j*-й элемент набора равен единице (включен). См. рис. 2.1, где представлен один такой пример.

[1] Во избежание проблем с представлением дополнительного кода используйте 32-/64-битное знаковое целое число для представления битовых масок только до 30/62 элементов соответственно.

```
S = 42 (десятичное) = 101010 (двоичное)
j = 3, 1 << j       = 001000 <- бит '1' смещен влево 3 раза
                     -------- AND (результат 'true', только если значения обоих
                                   битов 'true')
T = 8 (десятичное)  = 001000 (двоичное) -> не ноль, 3-й элемент включен
S = 42 (десятичное) = 101010 (двоичное)
j = 2, 1 << j       = 000100 <- бит '1' смещен влево 2 раза
                     -------- AND
T = 0 (десятичное)  = 000000 (двоичное) -> ноль, 2-й элемент выключен
```

5. Чтобы очистить/обнулить *j*-й элемент набора, используйте[1] побитовое «И» в комбинации с побитовым «НЕ» `S &= ~(1 << j)`.

```
S = 42 (десятичное) = 101010 (двоичное)
j = 1, ~(1 << j)    = 111101 <- '~' - это побитовое "НЕ"
                     -------- AND
S = 40 (десятичное) = 101000 (двоичное) // обновляем S до нового значения 40
```

6. Чтобы переключить (инвертировать состояние) *j*-го элемента набора, используйте побитовое XOR (побитовое исключающее «ИЛИ»): `S ^= (1 << j)`.

```
S = 40 (десятичное) = 101000 (двоичное)
j = 2, (1 << j)     = 000100 <- бит '1' смещен влево 2 раза
                     -------- XOR <- результат 'true', если оба бита имеют
                                     разные значения
S = 44 (десятичное) = 101100 (двоичное) // обновляем S до нового значения 44
S = 40 (десятичное) = 101000 (двоичное)
j = 3, (1 << j)     = 001000 <- бит '1' смещен влево 3 раза
                     -------- XOR <- результат 'true', если оба бита имеют
                                     разные значения
S = 32 (десятичное) = 100000 (двоичное) // обновляем S до нового значения 32
```

7. Чтобы получить значение младшего, т. е. первого справа значащего, бита, который включен (равен единице), используйте операцию `T = (S & (-S))`.

```
 S =  40 (десятичное) = 000...000101000 (32 бита, двоичное)
-S = -40 (десятичное) = 111...111011000 (дополнение двух)
                       ----------------- AND
 T =   8 (десятичное) = 000...000001000 (3-й бит справа включен)
```

8. Чтобы сделать равными единице (включить) все биты во множестве размера *n*, используйте операцию `S = (1 << n) - 1` (будьте осторожны, не получите ошибку переполнения).

```
Пример для n = 3
S + 1 = 8 (десятичное)  = 1000 <- бит '1' смещен влево 3 раза
                            1
                         ------ -
S     = 7 (десятичное)  =  111 (двоичное)
```

[1] Используйте скобки при выполнении операций с битами, чтобы избежать случайных ошибок, возникающих из-за неправильного понимания приоритета операторов.

```
Пример для n = 5
S + 1 = 32 (десятичное) = 100000 <- бит '1' смещен влево 5 раз
                                1
                           -------- -
S     = 31 (десятичное) =  11111 (двоичное)
```

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/bitmask.html

Файл исходного кода: *ch2_03_bit_manipulation.cpp/java*

Многие операции с битами в наших примерах исходного кода на C/C++ написаны как макроопределения препроцессора (но при этом написаны явно в примерах кода на Java, поскольку Java не поддерживает макросы).

- Список: `list` в C++ STL (`LinkedList` в Java)
  Хотя эта структура данных почти всегда упоминается в учебниках по структуре данных и алгоритмам, связный список обычно не используется в типичных (олимпиадных) задачах по программированию. Это связано с неэффективностью доступа к элементам (должен выполняться последовательный просмотр данных с начала или с конца списка), а использование указателей может привести к ошибкам времени выполнения, если оно неправильно реализовано. В этой книге почти все варианты решений, когда должен был использоваться список, были заменены более гибкой реализацией – `vector` в C++ STL (или `Vector` в Java).
  Единственным исключением, вероятно, является задача UVa 11988 – Broken Keyboard (a.k.a. Beiju Text) (Сломанная клавиатура, или кусочный текст), где вам необходимо динамически поддерживать (связный) список символов и эффективно вставлять новый символ в любом месте списка, то есть в начале («голова»), в текущей позиции или в конце («хвост») (связного) списка. Из 1903 задач с сайта университета Вальядолида, решенных авторами этой книги, это, вероятно, будет единственной задачей на использование списка.
- Стек: `stack` в C++ STL (`Stack` в Java)
  Эта структура данных часто используется в реализации алгоритмов, решающих определенные задачи (например, сопоставление скобок в разделе 9.4, калькулятор, вычисляющий постфиксное выражение, и преобразование инфиксного выражения в постфиксное (см. раздел 9.27), поиск компонентов сильной связности (см. раздел 4.2.9) и алгоритм Грэхема (см. раздел 7.3.7). Стек допускает только операции вставки наверх (push) (с временной сложностью $O(1)$) и удаления сверху (pop) с временной сложностью $O(1)$. Такое поведение обычно называется «последним вошел – первым вышел» (LIFO) и напоминает укладку и разборку поленницы в реальном мире. Операции стека, реализованные в C++ STL, включают `push()`/`pop()` (вставку/удаление из вершины стека), `top()` (получение содержимого из вершины стека) и `empty()`.

- Очередь: `queue` в C++ STL (`Queue` в Java[1])
  Эта структура данных используется в таких алгоритмах, как поиск в ширину (Breadth First Search, BFS), о котором рассказывается в разделе 4.2.2. Очередь допускает только операции добавления в конец очереди (с временной сложностью $O(1)$) – постановку в очередь – и удаления из начала очереди (с временной сложностью $O(1)$) – исключение из очереди. Это поведение называется «первым пришел – первым обслужен» (FIFO), оно соответствует обслуживанию очередей в реальном мире. Операции очереди в C++ STL включают `push()`/`pop()` (вставка в конце / удаление в начале очереди), `front()`/`back()` (получение содержимого конца/начала очереди) и `empty()`.
- Двусторонняя очередь (дек): `deque` в C++ STL (`Deque` в Java[2])
  Эта структура данных очень похожа на динамический массив (вектор) и очередь, описанные выше, за исключением того, что структура `deque` поддерживает быстрые (с временной сложностью $O(1)$) операции вставки и удаления как в ее начале, так и в ее конце. Эта особенность важна в определенном алгоритме, например алгоритм скользящего окна, о котором рассказывается в разделе 9.31. Операции двусторонней очереди в C++ STL включают `push_back()`, `pop_front()` (аналогично обычной очереди), `push_front()` и `pop_back()` (они являются специфичными для `deque`).

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/list.html

Файл исходного кода: *ch2_04_stack_queue.cpp/java*

Упражнение 2.2.1*. Предположим, вам дан несортированный массив S из n целых чисел. Решите каждую из следующих задач, используя наилучшие алгоритмы, которые вы можете придумать, и проанализируйте их временные сложности. Предположим следующие ограничения: $1 \leqslant n \leqslant 100K$, так что решения $O(n^2)$ теоретически невозможны в условиях олимпиады.

1. Определите, содержит ли S одну или несколько пар повторяющихся целых чисел.
2. * Дано целое число v, найдите два целых числа $a, b \in S$, таких что $a + b = v$.
3. * Продолжение вопроса 2: что вы будете делать в случае, если данный массив S уже отсортирован?
4. * Выведите целые числа из S, попадающие в диапазон $[a...b]$ (включая границы), в отсортированном порядке.

---

[1] `Queue` в Java – это только интерфейс, который обычно создается с помощью `LinkedList`.

[2] 10 `Deque` в Java также является интерфейсом. `Deque` обычно создается с помощью `LinkedList`.

5. * Определите длину самого длинного возрастающего непрерывного подмассива в S.
6. Определите медиану (50-й процентиль) *S*. Предположим, что *n* нечетно.

Упражнение 2.2.2: Есть несколько других эффектных приемов применения операций над битами, но они редко используются. Пожалуйста, решите эти задачи с помощью операций над битами.

1. Нужно получить остаток целочисленного деления *S* на *N* (где *N* является степенью числа 2), например: $S = (7)_{10}$ % $(4)_{10} = (111)_2$ % $(100)_2 = (11)_2 = (3)_{10}$.
2. Определите, является ли *S* степенью числа 2. Например, $S = (7)_{10} = (111)_2$ не является степенью 2, но $(8)_{10} = (100)_2$ является степенью 2.
3. Выключите последний ненулевой бит в *S*, например: $S = (40)_{10} = (10\underline{1}000)_2 \rightarrow S = (32)_{10} = (10\underline{0}000)_2$.
4. Включите последний нулевой бит в *S*, например: $S = (41)_{10} = (1010\underline{0}1)_2 \rightarrow S = (43)_{10} = (1010\underline{1}1)_2$.
5. Выключите последнюю последовательную цепочку ненулевых битов в *S*, например: $S = (39)_{10} = (100\underline{111})_2 \rightarrow S = (32)_{10} = (100\underline{000})_2$.
6. Включите последнюю последовательную цепочку нулевых битов в *S*, например: $S = (36)_{10} = (100100)_2 \rightarrow S = (39)_{10} = (100111)_2$.
7. * Решите задачу UVa 11173 – Grey Codes (код Грея), используя операции над битами, реализованные в виде одной строчки кода для каждого тестового примера, то есть найдите *k*-й элемент кода Грея.
8. * Для задачи UVa 11173, описанной выше: если известен элемент кода Грея, найдите его позицию *k* с помощью операций над битами.

Задачи по программированию с использованием линейных структур данных (и алгоритмов) и соответствующих библиотек

- Операции с одномерными массивами (размещение объектов в определенном порядке): `vector` в C++ STL (или `Vector`/`ArrayList` в Java)
  1. *UVa 00230 – Borrowers* (разбор строк, см. раздел 6.2; ведение списка отсортированных книг; ключ сортировки: имена авторов в первую очередь и, если есть связь, по названию; размер входных данных небольшой, хотя и не указан; нам не нужно использовать сбалансированное двоичное дерево поиска)
  2. UVa 00394 – Mapmaker (любой *n*-мерный массив хранится в памяти компьютера как одномерный массив; прочитайте условия и сделайте то, что требуется в задаче)
  3. UVa 00414 – Machined Surfaces (получите самые длинные интервалы, состоящие из символов «B»)
  4. UVa 00467 – Synching Signals (последовательный просмотр данных, булев флаг)
  5. UVa 00482 – Permutation Arrays (может потребоваться использовать string tokenizer, поскольку размер массива не указан. См. раздел 6.2)

6. UVa 00591 – Box of Bricks (сложите все предметы; получите среднее; суммируйте суммарные абсолютные разности каждого предмета от среднего, разделенного на два)
7. *UVa 00665 – False Coin* (используйте булевы флаги, изначально предположите, что все монеты – фальшивые; если «=», все монеты слева и справа не являются фальшивыми монетами; если «<» или «>», все монеты не слева и не справа – нефальшивые; проверьте, осталась ли в конце только одна, предположительно фальшивая, монета)
8. UVa 00755 – 487-3279 (таблица с прямой адресацией; преобразуйте все буквы, кроме Q & Z, в числа 2–9; оставьте «0»–»9» в формате 0–9; отсортируйте целые числа; найдите дубликаты, если таковые имеются)
9. **UVa 10038 – Jolly Jumpers *** (используйте булевы флаги для проверки [1..*n* – 1]).
10. UVa 10050 – Hartals (булев флаг)
11. UVa 10260 – Soundex (таблица с прямой адресацией для отображения «саундекс»-кода)
12. UVa 10978 – Let's Play Magic (работа со строковым массивом)
13. *UVa 11093 – Just Finish it up* (последовательный просмотр данных, кольцевой массив, немного сложнее)
14. UVa 11192 – Group Reverse (массив символов)
15. UVa 11222 – Only I did it (используйте несколько одномерных массивов, чтобы упростить эту задачу)
16. **UVa 11340 – Newspaper *** (таблица с прямой адресацией; см. «Хеширование» в разделе 2.3)
17. UVa 11496 – Musical Loop (храните данные в одномерном массиве, подсчитывайте «пики» звукового сигнала)
18. UVa 11608 – No Problem (используйте три массива: число созданных задач; число задач, которые необходимо создать; число доступных задач)
19. UVa 11850 – Alaska (для каждого целого числа от 0 до 1322, характеризующего местоположение в милях, определите, может ли Бренда достичь (где-нибудь в пределах 200 миль) каких-либо станций зарядки электромобилей?)
20. *UVa 12150 – Pole Position* (простая операция)
21. ***UVa 12356 – Army Buddies **** (решение аналогично удалению в двусвязных списках, но мы все еще можем использовать одномерный массив для базовой структуры данных)

- Операции с двумерными массивами
  1. UVa 00101 – The Blocks Problem (имитация «стека»; но нам также необходим доступ к содержимому каждого стека, поэтому лучше использовать двумерный массив)
  2. UVa 00434 – Matty's Blocks (своего рода задача о видимости в геометрии, решаемая с помощью операций с двумерными массивами)
  3. UVa 00466 – Mirror Mirror (основные функции: вращение и отражение)

4. UVa 00541 – Error Correction (подсчитайте число единиц («1») для каждой строки/столбца; все они должны быть четными; если есть ошибка, проверьте, расположена ли она в той же строке и столбце)
5. UVa 10016 – Flip-flop the Squarelotron (утомительная задача)
6. UVa 10703 – Free spots (используйте двумерный массив, состоящий из булевых значений, размером 500×500)
7. **UVa 10855 – Rotated squares** * (строковый массив; поверните на 90° по часовой стрелке)
8. **UVa 10920 – Spiral Tap** * (смоделируйте процесс)
9. UVa 11040 – Add bricks in the wall (нетривиальные операции с двумерным массивом)
10. UVa 11349 – Symmetric Matrix (используйте тип данных long long, чтобы избежать проблем)
11. UVa 11360 – Have Fun with Matrices (делай, как просили)
12. **UVa 11581 – Grid Successors** * (смоделируйте процесс)
13. UVa 11835 – Formula 1 (сделайте то, что требуется в задаче)
14. *UVa 12187 – Brothers* (смоделируйте процесс)
15. *UVa 12291 – Polyomino Composer* (сделайте то, что требуется в задаче; немного нудно)
16. *UVa 12398 – NumPuzz I* (симуляция в обратном направлении; не забудьте про mod 10)

- `algorithm` в C++ STL (`Collections` в Java)

1. UVa 00123 – Searching Quickly (измененная функция сравнения; используйте сортировку)
2. **UVa 00146 – ID Codes** * (используйте `next_permutation`)
3. UVa 00400 – Unix ls (эта команда очень часто применяется в UNIX)
4. UVa 00450 – Little Black Book (утомительная задача с сортировкой)
5. *UVa 00790 – Head Judge Headache* (решается аналогично UVa 10258)
6. UVa 00855 – Lunch in Grid City (сортировка, медиана)
7. UVa 01209 – Wordfish (LA 3173, Манила'06) (используйте `next` и `prev_permutation` в STL)
8. *UVa 10057 – A mid-summer night...* (включает поиск медианы; используйте `sort`, `upper_bound`, `lower_bound` в STL и некоторые проверки)
9. **UVa 10107 – What is the Median?** * (найти медиану расширяющегося/динамического списка целых чисел; задача все еще решается с помощью нескольких вызовов `nth_element` в `algorithm`)
10. UVa 10194 – Football a.k.a. Soccer (сортировка по нескольким полям, используйте `sort`)
11. **UVa 10258 – Contest Scoreboard** * (сортировка по нескольким полям, используйте `sort`)
12. *UVa 10698 – Football Sort* (сортировка по нескольким полям, используйте `sort`)
13. UVa 10880 – Colin and Ryan (используйте `sort`)
14. UVa 10905 – Children's Game (модифицированная функция сравнения, используйте `sort`)

15. UVa 11039 – Building Designing (используйте `sort`, затем посчитайте различные знаки)
16. UVa 11321 – Sort Sort and Sort (будьте осторожны с операцией MOD над отрицательными числами!)
17. UVa 11588 – Image Coding (использование `sort` упрощает задачу)
18. UVa 11777 – Automate the Grades (использование `sort` упрощает задачу)
19. UVa 11824 – A Minimum Land Price (использование `sort` упрощает задачу)
20. *UVa 12541 – Birthdates* (LA6148, Хат Яй'12, использование `sort`, выберите самых младших и самых старых)

- Операции с битами (`bitset` в C++ STL (`BitSet` в Java) и битовая маска)
  1. UVa 00594 – One Little, Two Little… (работа с битовой строкой с помощью `bitset`)
  2. UVa 00700 – Date Bugs (задачу можно решить с помощью `bitset`)
  3. UVa 01241 – Jollybee Tournament (LA 4147, Джакарта'08, легкая задача)
  4. ***UVa 10264 – The Most Potent Corner** ** (сложные операции с битовой маской)
  5. *UVa 11173 – Grey Codes* (шаблон D & C или операция с битами, занимающая одну строчку кода)
  6. UVa 11760 – Brother Arif… (отдельные проверки строк + столбцов; используйте два набора битов)
  7. **UVa 11926 – Multitasking *** (используйте `bitset` (1M), чтобы проверить, свободен ли слот)
  8. **UVa 11933 – Splitting Numbers *** (операции с битами)
  9. IOI 2011 – Pigeons (эта задача упрощается, если использовать операции с битами, но для ее окончательного решения требуется гораздо больше)

- `list` в C++ STL (`LinkedList` в Java)
  1. UVa 11988 – Broken Keyboard… * (редкая задача; используйте связный список)

- `stack` в C++ STL (`Stack` в Java)
  1. UVa 00127 – «Accordian» Patience («перетасовывание» стека)
  2. **UVa 00514 – Rails *** (используйте `stack` для моделирования процесса)
  3. **UVa 00732 – Anagram by Stack *** (используйте `stack` для моделирования процесса)
  4. ***UVa 01062 – Containers** ** (LA 3752, заключительные всемирные соревнования, Токио'07, моделирование с использованием `stack`; максимальный ответ – 26 стеков; существует решение с временной сложностью $O(n)$)
  5. UVa 10858 – Unique Factorization (используйте `stack` для решения этой задачи)

     Также смотрите: неявные вызовы `stack` в рекурсивных функциях; преобразование/оценка Postfix в разделе 9.27.

- `stack` и `deque` в C++ STL (`Queue` и `Deque` в Java)
  1. UVa 00540 – Team Queue (измененная «очередь»)
  2. **UVa 10172 – The Lonesome Cargo… *** (используйте `queue` и `stack`)
  3. **UVa 10901 – Ferry Loading III *** (моделирование с использованием `queue`)
  4. UVa 10935 – Throwing cards away I (моделирование с использованием `queue`)
  5. **UVa 11034 – Ferry Loading IV *** (моделирование с использованием `queue`)
  6. *UVa 12100 – Printer Queue* (моделирование с использованием `queue`)
  7. *UVa 12207 – This is Your Queue* (используйте `queue` и `deque`)
     Также смотрите: использование `queue` для поиска в ширину (раздел 4.2.2).

2.3. Нелинейные структуры данных – встроенные библиотеки

Для некоторых задач линейное хранилище – не лучший способ организации данных. С помощью эффективных реализаций нелинейных структур данных, показанных ниже, вы сможете работать с данными быстрее, тем самым ускоряя алгоритмы, которые основываются на них.

Например, если вам нужен динамический[1] набор пар данных (например, пар ключ–значение), то, используя `map` из C++ STL, как показано ниже, можно получить производительность, эквивалентную $O(\log n)$ для операций вставки/поиска/удаления, написав всего несколько строк кода (которые все равно придется писать самостоятельно. В то же время хранение тех же данных внутри статического массива, состоящего из элементов `struct`, может иметь временную сложность $O(n)$ для вставки/поиска/удаления, и вам придется самостоятельно писать больше строчек кода в поисках обходного пути.

- Сбалансированное двоичное дерево поиска: `map`/`set` в C++ STL (`TreeMap`/`TreeSet` в Java)
  Дерево двоичного поиска – один из способов организации данных в древовидной структуре. В каждом поддереве с корнем в точке x выполняется следующее свойство дерева двоичного поиска: элементы в левом поддереве элемента x меньше, чем x, а элементы в правом поддереве элемента x больше (или равны) x. По сути, это применение стратегии «разделяй и властвуй» (см. также раздел 3.3). Такая организация данных (см. рис. 2.2) позволяет выполнять операции с временной сложностью $O(\log n)$: `search(key)`, `insert(key)`, `findMin()`/`findMax()`, `successor(key)`/`predecessor(key)` и `delete(key)`, поскольку в худшем случае, при просмотре данных от корня к листу, выполняются только операции с временной

[1] Содержание динамической структуры данных часто модифицируется с помощью операций вставки/удаления/обновления.

сложностью $O(\log n)$ (подробный разбор см. в [7, 5, 54, 12]). Однако это верно лишь в том случае, если дерево двоичного поиска является сбалансированным.

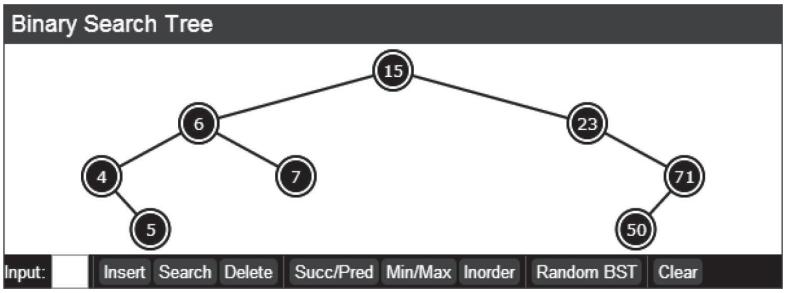


Рис. 2.2 ❖ Пример дерева двоичного поиска

Безошибочная реализация сбалансированных деревьев двоичного поиска, таких как, например, сбалансированные по высоте деревья Адельсон-Вельского и Ландиса (АВЛ-деревья)[1] или красно-черные деревья двоичного поиска (RB-деревья)[2], является трудоемкой задачей, и ее трудно выполнить в условиях олимпиадного программирования, когда ваше время сильно ограничено (если, конечно, вы не написали заранее библиотеку кода, см. раздел 9.29). К счастью, в C++ STL есть `map` и `set` (а в Java есть `TreeMap` и `TreeSet`), которые *обычно* являются реализациями RB-дерева; это гарантирует, что основные операции со сбалансированными деревьями двоичного поиска, такие как вставка, поиск и удаление, выполняются за время $O(\log n)$. Попрактиковавшись в использовании этих двух шаблонных классов C++ STL (или Java API), вы сможете серьезно сэкономить драгоценное время во время соревнований по программированию! Разница между этими двумя структурами данных проста: `map` из C++ STL (и `TreeMap` в Java) хранит пары (ключ → данные), тогда как `set` из C++ STL (и `TreeSet` в Java) хранит только ключ. Для большинства задач мы используем `map` (чтобы действительно отобразить данные), а не `set` (`set` используется только для эффективного определения существо-

[1] АВЛ-дерево было первым самобалансирующимся деревом двоичного поиска, которое было изобретено. АВЛ-деревья – это, по сути, традиционные деревья двоичного поиска с дополнительным свойством: высоты двух поддеревьев любой вершины в АВЛ-дереве могут отличаться не более чем на единицу. Операции перебалансировки (поворота) выполняются, при необходимости, во время операций вставки и удаления, чтобы поддерживать тот инвариант и, следовательно, сохранять дерево приблизительно сбалансированным.

[2] Красно-черное дерево – это еще одна разновидность самобалансирующихся деревьев двоичного поиска, в котором каждая вершина имеет цвет: красный или черный. В RB-деревьях корневая вершина, все листья и оба потомка каждой красной вершины – черные. Каждый простой путь от вершины к любому из ее дочерних листьев содержит одинаковое количество черных вершин. При выполнении операций вставки и удаления RB-дерево будет поддерживать все эти инварианты, чтобы дерево оставалось сбалансированным.

вания определенного ключа). Однако у применения стандартных библиотек есть и отрицательная сторона. Если мы используем реализации стандартных библиотек, становится трудно или совсем невозможно дополнить деревья двоичного поиска (добавить дополнительные данные). Попробуйте выполнить упражнение 2.3.5* и прочтите раздел 9.29, где эта тема обсуждается более подробно.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/bst.html

Файл исходного кода: *ch2_05_map_set.cpp/java*

- Куча: `priority_queue` в C++ STL (`PriorityQueue` в Java)
  Куча – это еще один способ организации данных в дереве. (Двоичная) куча также является двоичным деревом, подобным дереву двоичного поиска, но с одним дополнительным ограничением: она должна быть *полным*[1] деревом. Полные двоичные деревья могут эффективно храниться в компактном массиве размером $n + 1$, который во многих случаях предпочтительнее явного представления дерева. Например, массив A = {N/A, 90, 19, 36, 17, 3, 25, 1, 2, 7} является компактным представлением массива, показанного на рис. 2.3, где индекс 0 игнорируется. Можно перейти от определенного индекса (вершины) i к его родительскому, левому и правому дочерним элементам, используя простые операции с индексами: $i/2$, $2 \times i$ и $2 \times i + 1$ соответственно. Эти операции с индексами могут быть выполнены быстрее, если использовать операции над битами (см. раздел 2.2): `i >> 1`, `i << 1` и `(i << 1) + 1` соответственно.
  Вместо свойств, характерных для деревьев двоичного поиска, организация невозрастающей пирамиды (max-heap) использует свойство кучи: в каждом поддереве с корнем в x элементы в левом и правом поддеревьях x меньше (или равны) x (см. рис. 2.3). Это также относится к применению концепции «разделяй и властвуй» (см. раздел 3.3). Такое свойство гарантирует, что вершина (или корень) двоичной кучи всегда является максимальным элементом. В куче нет понятия «поиск» (в отличие от деревьев двоичного поиска). Вместо этого куча позволяет быстро извлекать (удалять) максимальный элемент `ExtractMax()` и вставлять новые элементы `Insert(v)` – и то, и другое легко достигается с помощью обхода дерева от корня к листу или от листа к корню, занимающего время $O(\log(n))$; при необходимости выполняются операции замены для поддержания свойств невозрастающей кучи (подробнее см. в [7, 5, 54, 12]).
  Невозрастающая куча (max-heap) – это полезная структура данных для моделирования очереди с приоритетами, в которой элемент с наивысшим приоритетом (максимальный элемент) может быть удален из

[1] Полное двоичное дерево – это двоичное дерево, в котором каждый уровень, за исключением, может быть, последнего, полностью заполнен. Все вершины на последнем уровне также должны быть заполнены слева направо.

очереди (`ExtractMax()`), а новый элемент *v* может быть помещен в очередь (`Insert(v)`), причем обе операции выполняются за время $O(\log n)$. Реализация[1] `priority_queue` доступна в библиотеке `queue` в C++ STL (или `PriorityQueue` в Java). Очереди приоритетами являются важным компонентом в алгоритмах, таких как алгоритмы Прима (и Краскала) для задачи построения минимального остовного дерева (МОД) (см. раздел 4.3), алгоритма Дейкстры для задачи нахождения кратчайших путей из одной вершины во все остальные вершины графа (см. раздел 4.4.3) и алгоритм поиска A * – поиск по первому наилучшему совпадению на графе (см. раздел 8.2.5).

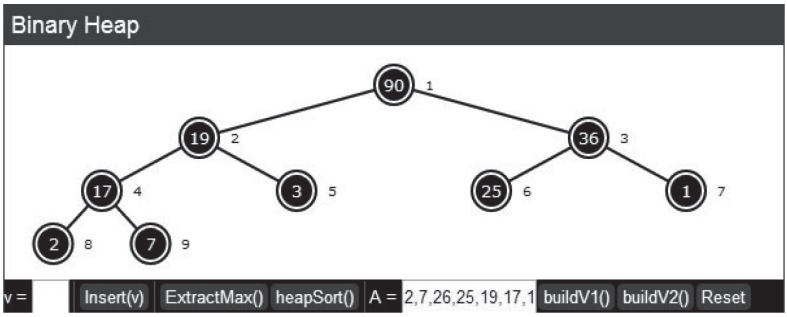


Рис 2.3 ❖ Визуализация свойств невозрастающей кучи

Эта структура данных также используется для выполнения операции `partial_sort` в библиотеке алгоритмов C++ STL. Одна из возможных реализаций заключается в обработке элементов по одному и создании кучи – невозрастающей[2] кучи из *k* элементов, при которой самый большой элемент удаляется всякий раз, когда ее размер превышает *k* (*k* – количество элементов, запрошенных пользователем). Наименьшие *k* элементов затем могут быть получены в порядке убывания путем удаления остальных элементов в невозрастающей куче. Поскольку каждая операция удаления выполняется за время $O(\log k)$, `partial_sort` имеет

[1] Стандартная реализация `priority_queue` в C++ STL представляет собой невозрастающую кучу (Max Heap) (удаление из очереди возвращает элементы в порядке убывания ключа), тогда как `PriorityQueue` в Java по умолчанию представляет собой неубывающую кучу (Min Heap) (возвращает элементы в порядке возрастания ключа). Советы: невозрастающая куча, содержащая числа, может быть легко преобразована в неубывающую кучу (и наоборот), если вставить отрицательные ключи. Это происходит потому, что применение операции смены знака для набора чисел изменит их порядок появления при сортировке. Этот прием применяется несколько раз в данной книге. Однако если очередь приоритетами используется для хранения 32-разрядных целых чисел со знаком, произойдет ошибка переполнения, если мы выполним такую операцию для числа -2^{31}, так как $2^{31} - 1$ – максимальное значение 32-разрядного целого числа со знаком.

[2] Сортировка по умолчанию `partial_sort` выдает наименьшие *k* элементов в порядке возрастания.

временную сложность[1] $O(n \log k)$. Когда $k = n$, этот алгоритм эквивалентен пирамидальной сортировке. Обратите внимание, что хотя временная сложность пирамидальной сортировки также составляет $O(n \log n)$, пирамидальная сортировка часто выполняется медленнее, чем быстрая сортировка, потому что при операциях пирамидальной сортировки происходят обращения к данным, хранящимся в далеко расположенных индексах, что делает неудобным использование кеша.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/heap.html

Файл исходного кода: *ch2_06_priority_queue.cpp/java*

- Хеш-таблица: `unordered map`[2] в C++ 11 STL (и `HashMap`/`HashSet`/`HashTable` в Java)
  Хеш-таблица – это еще одна нелинейная структура данных, но мы не рекомендуем использовать ее в олимпиадном программировании без крайней необходимости. Проектирование хорошо работающей хеш-функции часто бывает сложно, и только новая версия C++ 11 STL поддерживает эту возможность (в Java есть классы, связанные с хешем).
  Нужно заметить, что `map` или `set` в C++ STL (а также `TreeMap` или `TreeSet` в Java) обычно выполняются достаточно быстро, поскольку типичный размер входных данных (для соревнований по программированию) обычно не превышает 1*M*. В этих пределах производительность $O(1)$ при использовании хеш-таблиц и производительность $O(\log 1M)$ при использовании сбалансированных деревьев двоичного поиска не сильно отличаются. Таким образом, мы не будем подробно обсуждать хеш-таблицы в этом разделе.
  Тем не менее простую форму хеш-таблиц можно использовать в соревнованиях по программированию. «Таблицы с прямой адресацией» (Direct Addressing Table, DAT) можно рассматривать как хеш-таблицы, в которых сами ключи являются индексами или где «хеш-функция» является функцией тождества. Например, нам может потребоваться попарно сопоставить все возможные символы ASCII [0–255] и целые числа, например «a» → «3», «W» → «10», ..., «I» → «13». Для этой цели нам не нужен `map` в C++ STL или какая-либо другая форма хеширования, поскольку сам ключ (значение символа ASCII) уникален и достаточен для определения соответствующего индекса в массиве размером 256. Некоторые упражнения по программированию, которые можно выполнить, используя таблицы с прямой адресацией, перечислены в предыдущем разделе 2.2.

---

[1] Вы могли заметить, что временная сложность $O(n \log k)$, где k – размер выходного набора данных, а n – размер входного набора данных. Это означает, что алгоритм «чувствителен к выводу», поскольку его время выполнения зависит не только от размера входного набора данных, но и от количества элементов, которые он должен вывести.

[2] Обратите внимание, что C++ 11 – это новый стандарт C++, старые компиляторы могут его пока не поддерживать.

Упражнение 2.3.1. Кто-то предположил, что можно сохранить пары ключ–значение в *отсортированном массиве* элементов `struct`, чтобы мы могли использовать двоичный поиск $O(\log n)$ для приведенной выше задачи. Возможен ли такой подход? Если нет, то в чем проблема?

Упражнение 2.3.2. Мы не будем обсуждать основы операций с деревьями двоичного поиска в этой книге. Вместо этого мы используем серию подзадач для проверки вашего понимания концепций, связанных с деревьями двоичного поиска.

Рисунок 2.2 будет применяться в качестве *начального ориентира* во всех подзадачах, кроме подзадачи 2.

1. Покажите шаги, выполняемые `search(71)`, `search(7)`, а затем `search(22)`.
2. Начиная с *пустого* дерева двоичного поиска, покажите шаги, выполняемые `insert(15)`, `insert(23)`, `insert(6)`, `insert(71)`, `insert(50)`, `insert(4)`, `insert(7)` и `insert(5)`.
3. Покажите шаги, выполняемые `findMin()` (и `findMax()`).
4. Покажите *симметричный обход* этого дерева. Будут ли при этом выходные данные отсортированы?
5. Покажите шаги, выполняемые `successor(23)`, `successor(7)` и `successor(71)`.
6. Покажите шаги, выполненные `delete(5)` (лист), `delete(71)` (внутренний узел с одним дочерним элементом), а затем `delete(15)` (внутренний узел с двумя дочерними элементами).

Упражнение 2.3.3*. Предположим, вам дана ссылка на корень R двоичного дерева T, содержащего n вершин. Вы можете получить доступ к левой, правой и родительской вершинам узла, а также к его ключу через ссылку. Решите каждую из следующих задач с помощью наилучших возможных алгоритмов, которые вы можете придумать, и проанализируйте их временные сложности.

Предположим следующие ограничения: $1 \leqslant n \leqslant 100K$, так что решения $O(n^2)$ теоретически невозможны в условиях олимпиады.

1. Проверьте, является ли T деревом двоичного поиска.
2. * Выведите элементы в T, которые находятся в заданном диапазоне [a..b] в порядке возрастания.
3. * Выведите содержимое *листьев* T в порядке *убывания*.

Упражнение 2.3.4*. Симметричный обход (см. также раздел 4.7.2) стандартного (не обязательно сбалансированного) дерева двоичного поиска, как известно, размещает элементы дерева двоичного поиска в отсортированном порядке и имеет временную сложность $O(n)$. Будет ли код, приведенный ниже, также размещать элементы дерева двоичного поиска в отсортированном порядке?

Можно ли заставить его работать за общее время $O(n)$ вместо $O(\log n + (n - 1) \times \log n) = O(n \log n)$? Если это возможно, то как?

```
x = findMin(); output x
for (i = 1; i < n; i++)                          // работает ли этот цикл за O(n log n)?
  x = successor(x); output x
```

Упражнение 2.3.5*. Для некоторых (сложных) задач необходимо создавать свои собственные реализации сбалансированных деревьев двоичного поиска

из-за необходимости добавления в них дополнительных данных (см. главу 14 в [7]). Задача: решите задачу UVa 11849 – CD, которая представляет собой чистую задачу построения сбалансированного дерева двоичного поиска в вашей собственной реализации, чтобы проверить ее правильность и протестировать производительность.

Упражнение 2.3.6. Мы не будем подробно разбирать основные принципы операций с кучей в этой книге. Вместо этого мы предложим ряд вопросов, чтобы проверить ваше понимание концепций кучи.

1. На рис. 2.3 представлены начальные данные для кучи. Отобразите шаги, выполняемые командой `Insert(26)`.
2. После ответа на вопрос 1 выше отобразите шаги, выполняемые командой `ExtractMax()`.

Упражнение 2.3.7. Является ли структура, представленная компактным одномерным массивом (без учета индекса 0), отсортированная в порядке убывания, невозрастающей кучей?

Упражнение 2.3.8*. Докажите или опровергните это утверждение: «Второй по величине элемент в невозрастающей куче с $n \geqslant 3$ различными элементами всегда является одним из прямых потомков корня». Следующий вопрос: как насчет третьего по величине элемента? Где находится потенциальное местоположение (местоположения) третьего по величине элемента в невозрастающей куче?

Упражнение 2.3.9*. Пусть дан одномерный компактный массив A, содержащий n целых чисел ($1 \leqslant n \leqslant 100\,000$), которые гарантированно удовлетворяют свойству невозрастающей кучи. Выведите элементы в A, которые больше целого числа v. Каков здесь наилучший алгоритм?

Упражнение 2.3.10*. Для неотсортированного массива S из n различных целых чисел ($2k \leqslant n \leqslant 100\,000$) найдите наибольшее и наименьшее значения k ($1 \leqslant k \leqslant 32$) целых чисел в S, используя алгоритм с временной сложностью, не превышающей $O(n \log k)$. **Примечание**. Для этого упражнения предположим, что алгоритм с временной сложностью $O(n \log n)$ неприемлем.

Упражнение 2.3.11*. Одной из операций над кучей, не поддерживаемой непосредственно функцией `priority_queue` в C++ STL (и `PriorityQueue` в Java), является операция `UpdateKey(index, newKey)`, которая позволяет обновлять (увеличивать или уменьшать) значение элемента кучи (невозрастающей пирамиды) с определенным индексом. Напишите свою *собственную реализацию* двоичной кучи (невозрастающей пирамиды) с помощью этой операции.

Упражнение 2.3.12*. Еще одна полезная операция кучи – операция `DeleteKey(index)` для удаления элементов невозрастающей кучи с заданным индексом. Реализуйте эту операцию.

Упражнение 2.3.13*. Предположим, что нам нужна только операция `DecreaseKey(index, newKey)`, то есть такая операция `UpdateKey`, когда обновление значения ключа всегда делает `newKey` меньше своего предыдущего значения. Можем ли мы использовать более простой подход, чем в упражнении 2.3.11? Подсказка:

используйте «ленивое удаление», мы будем применять эту технику в нашем коде, реализующем алгоритм Дейкстры, в разделе 4.4.3.

Упражнение 2.3.14*. Возможно ли использовать сбалансированное дерево двоичного поиска (например, `set` в C++ STL или `TreeSet` в Java) для реализации очереди с приоритетами с одинаковой производительностью постановки в очередь и удаления из нее ($O(\log n)$)? Если да, то как? Есть ли потенциальные недостатки у этого варианта? Если нет, почему?

Упражнение 2.3.15*. Существует ли лучший способ реализовать очередь с приоритетами, если все ключи представляют собой целые числа в небольшом диапазоне, например [0. , , 100]? Мы ожидаем, что постановка в очередь и удаление из очереди будут иметь производительность $O(1)$. Если да, то как ее реализовать? Если нет, почему?

Упражнение 2.3.16. Какую нелинейную структуру данных следует использовать, если необходимо поддерживать следующие три динамические операции: 1) много вставок, 2) много удалений и 3) много запросов данных в отсортированном порядке?

Упражнение 2.3.17. Есть M строк. N из них уникальны ($N \leqslant M$). Какую нелинейную структуру данных, обсуждаемую в этом разделе, следует использовать, если вам нужно проиндексировать (пометить) эти M строк целыми числами из [0..N – 1]? Критерии индексации следующие: первой строке должно быть присвоено значение индекса 0, следующей строке должно быть присвоено значение индекса 1 и т. д. Однако если строка встречается повторно, ей должно быть присвоено то же значение индекса, что и ее предыдущей копии. Одним из применений данной задачи является построение графа связей на списке из названий городов (которые не являются целочисленными индексами!) и списке маршрутных магистралей между этими городами (см. раздел 2.4.1). Для этого сначала нужно перевести названия этих городов в целочисленные индексы (с которыми гораздо удобнее работать).

Задачи по программированию, решаемые с помощью библиотеки нелинейных структур данных

- `map` в C++ STL (и `TreeMap` в Java)
  1. UVa 00417 – Word Index (сгенерируйте все слова, добавьте в `map` для автоматической сортировки)
  2. UVa 00484 – The Department of... (определите частоту вхождений слова с помощью `map`)
  3. UVa 00860 – Entropy Text Analyzer (подсчет частоты вхождений)
  4. *UVa 00939 – Genes* (сопоставьте имя ребенка с его/ее геном и именами родителей с помощью `map`)
  5. *UVa 10132 – File Fragmentation* (N = количество фрагментов, B = общее количество битов всех фрагментов, разделенное на $N/2$; попробуйте все $2 \times N^2$ объединений двух фрагментов, имеющих длину B; определите один с наибольшей частотой; используйте `map`)

6. *UVa 10138 – CDVII* (сопоставьте номерные таблички со счетами, временем и положением въезда на платную дорогу)
7. **UVa 10226 – Hardwood Species** * (используйте хеширование для улучшения производительности)
8. UVa 10282 – Babelfish (задача из области составления словарей; используйте `map`)
9. UVa 10295 – Hay Points (используйте `map` для работы со словарем)
10. UVa 10686 – SQF Problem (используйте `map` для управления данными)
11. UVa 11239 – Open Source (используйте `map` и `set` для проверки предыдущих строк)
12. **UVa 11286 – Conformity** * (используйте `map` для отслеживания частот включения курсов в программу)
13. UVa 11308 – Bankrupt Baker (используйте `map` и `set` для управления данными)
14. *UVa 11348 – Exhibition* (используйте `map` и `set` для проверки уникальности)
15. ***UVa 11572 – Unique Snowflakes*** * (используйте `map` для записи индекса, отражающего частоту появления снежинок определенного размера; используйте этот метод для определения ответа в $O(n \log n)$)
16. UVa 11629 – Ballot evaluation (используйте `map`)
17. UVa 11860 – Document Analyzer (используйте `set` и `map`, последовательный просмотр данных)
18. UVa 11917 – Do Your Own Homework (используйте `map`)
19. *UVa 12504 – Updating a Dictionary* (используйте `map`; строка за строкой; немного утомительная задача)
20. *UVa 12592 – Slogan Learning of Princess* (используйте `map`; строка за строкой)

    О методе подсчета частоты вхождений см. раздел 6.3.

- `set` в C++ STL (`TreeSet` в Java)
  1. UVa 00501 – Black Box (используйте `multiset`, эффективно работающий с итераторами)
  2. **UVa 00978 – Lemmings Battle** * (симуляция, используйте `multiset`)
  3. UVa 10815 – Andy's First Dictionary (используйте `set` и `string`)
  4. UVa 11062 – Andy's Second Dictionary (задача похожа на UVa 10815; задача с некоторыми трюками)
  5. **UVa 11136 – Hoax or what** * (используйте `multiset`)
  6. **UVa 11849 – CD** * (используйте `set`, чтобы уложиться в ограничение по времени, лучший вариант: используйте хеширование!)
  7. *UVa 12049 – Just Prune The List* (используйте `multiset`)

- `priority_queue` в C++ STL (`PriorityQueue` в Java)
  1. **UVa 01203 – Argus** * (LA 3135, Пекин'04; используйте `priority_queue`)
  2. **UVa 10954 – Add All** * (используйте `priority_queue`, «жадный» алгоритм)
  3. **UVa 11995 – I Can Guess…** * (`stack`, `queue` и `priority_queue`)

     Также см.: использование `priority_queue` для операций топологиче-

ской сортировки (см. раздел 4.2.1), алгоритмы Краскала[1] (см. раздел 4.3.2), Прима (см. раздел 4.3.3), Дейкстры (см. раздел 4.4.3) и расширенные методы поиска (A*) (см. раздел 8.2.5).

2.4. Структуры данных с реализациями библиотек, написанными авторами этой книги

По состоянию на май 2013 года важные структуры данных, приведенные в этом разделе, еще не имели встроенной поддержки в C++ STL или Java API. Таким образом, чтобы быть конкурентоспособными, участники олимпиад по программированию должны подготовить корректные реализации этих структур данных. В данном разделе мы обсудим ключевые идеи и примеры реализации этих структур данных (см. также приведенный исходный код).

2.4.1. Граф

Граф – распространенная структура, которая встречается во многих задачах программирования. Граф ($G = (V, E)$), по сути, представляет собой просто набор вершин (V) и ребер (E; ребра хранят информацию о связности между вершинами в V). Позже, в главах 3, 4, 8 и 9, мы рассмотрим множество важных задач и алгоритмов, имеющих отношение к графам. В качестве вступления мы обсудим в этом разделе[2] три основных способа (есть несколько других, более редко встречающихся структур) представления графа G с V вершинами и E ребрами.

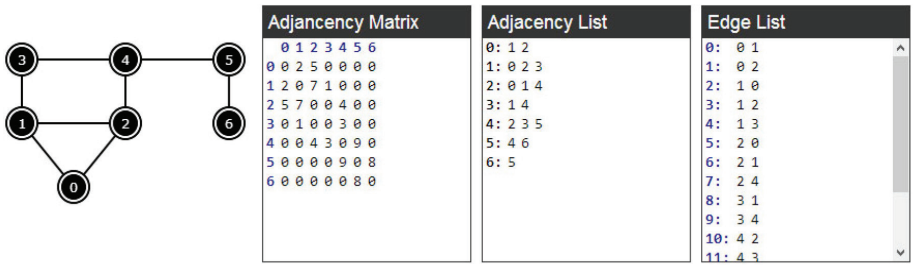


Рис. 2.4 ❖ Визуализация структуры данных графа

A. Матрица смежности, как правило, представленная в виде двумерного массива (см. рис. 2.4). В задачах олимпиадного программирования, связанных с графами, число вершин V обычно известно. Таким образом, мы

---

[1] Это еще один способ реализовать сортировку ребер с помощью алгоритма Краскала. Наша реализация (C++), приведенная в разделе 4.3.2, просто применяет `vector` + `sort` вместо `priority_queue` (пирамидальной сортировки).

[2] Наиболее уместным обозначением мощности множества S является $|S|$. Однако в этой книге мы часто будем использовать значение V или E в качестве $|V|$ или $|E|$, в зависимости от контекста.

можем построить «таблицу связности», создав статический 2D-массив: `int AdjMat[V][V]`. Это решение имеет *сложность по памяти*[1] $O(V^2)$. Для невзвешенного графа присвойте `AdjMat[i][j]` ненулевое значение (обычно 1), если между вершинами *i* и *j* есть ребро, или ноль в противном случае. Для взвешенного графа присвойте значения элементам `AdjMat[i][j] = weight(i,j)`, если между вершинами *i* и *j* есть ребро, имеющее вес `weight(i,j)`, или ноль в противном случае. Матрицу смежности нельзя использовать для хранения мультиграфа. Для простого графа без петель главная диагональ матрицы содержит только нули, т. е. `AdjMat[i][i] = 0`, $\forall i \in$ `[0..V-1]`.

Матрица смежности – хорошая тактика решения задач, если часто требуется определять связь между двумя вершинами в *небольшом плотном графе*. Однако ее не рекомендуется использовать для *больших разреженных графов*, так как для этого потребуется слишком много места ($O(V^2)$), и в двумерном массиве будет много пустых (нулевых) элементов. Для успешного решения олимпиадных задач по программированию обычно невозможно использовать матрицы смежности для случаев, когда V больше ≈ 1000. Другой недостаток матрицы смежности состоит в том, что для перечисления списка соседей вершины *v* – операции, выполняемой во многих алгоритмах на графах, – также требуется время $O(V)$, даже если у вершины есть всего лишь несколько соседей. Более компактным и эффективным представлением графа является список смежности, который обсуждается ниже.

B. Список смежности, как правило, реализуется в виде вектора вектора пар (см. рис. 2.4).

Реализация с использованием библиотеки C++ STL: `vector <vii> AdjList`, где `vii` определено как

```
typedef pair <int, int> ii; typedef vector <ii> vii;          // ярлыки типов данных
```

Реализация с использованием Java API: `Vector< Vector < IntegerPair > > AdjList`. `IntegerPair` – это простой класс Java, который содержит пару целых чисел, таких как `ii`, в коде, приведенном выше.

В списках смежности у нас есть вектор векторов пар, в котором список соседей каждой вершины *u* хранится в виде пар «данных о ребрах». Каждая пара содержит два элемента информации: индекс соседней вершины и вес ребра. Если имеется невзвешенный граф, просто сохраните его вес как 0, 1 или полностью отбросьте атрибут веса[2]. Сложность по памяти списка смежности равна $O(V + E)$, потому что если в (простом) графе имеется E двунаправленных ребер, в списке смежности будут храниться только $2E$ пар «данных о ребрах». Поскольку E обычно намного меньше,

---

[1] Мы различаем *сложность по памяти* и *сложность по времени* структур данных. Сложность по памяти – это асимптотическое значение, выражающее требования к памяти структуры данных, тогда как временная сложность – это асимптотическое значение, обозначающее время, затрачиваемое на запуск определенного алгоритма или операции над структурой данных.

[2] Для простоты мы всегда будем предполагать, что второй атрибут существует во всех реализациях графов в этой книге, хотя он не всегда используется.

чем $V \times (V-1)/2 = O(V^2)$ – максимальное число ребер в полном (простом) графе, – то списки смежности часто имеют большую эффективность в плане пространственной сложности, чем матрицы смежности. Обратите внимание, что список смежности можно использовать для хранения мультиграфа.

С помощью списков смежности мы также можем эффективно перечислить список соседей вершины v. Если v имеет k соседей, перечисление потребует $O(k)$ времени. Поскольку это одна из наиболее распространенных операций в большинстве алгоритмов на графах, рекомендуется в первую очередь рассматривать использование списков смежности в качестве представления графа. В отсутствие явных указаний на использование иных структур данных большинство алгоритмов на графах, обсуждаемых в этой книге, используют списки смежности.

C. Список ребер, обычно в форме вектора троек (см. рис. 2.4).
Реализация с использованием библиотеки C++ STL: `vector <pair <int, ii>> EdgeList`.
Реализация с использованием Java API: `Vector <IntegerTriple> EdgeList`.
`IntegerTriple` – это класс, который содержит тройку целых чисел, таких как пара `<int, ii>` выше.
В списке ребер мы храним список всех E ребер, обычно в отсортированном порядке. Для ориентированных графов мы можем хранить двунаправленное ребро с помощью двух наборов данных, по одному для каждого направления. Сложность по памяти для этого случая – очевидно, $O(E)$. Такое представление графа эффективно используется в алгоритме Краскала для поиска минимального остовного дерева (MST) (см. раздел 4.3.2), где набор неориентированных ребер должен быть отсортирован[1] по возрастанию веса. Однако сохранение информации о графе в списке ребер усложняет многие алгоритмы, где требуется перечислить ребра, соединенные с вершиной.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/graphds.html

Файл исходного кода: *ch2_07_graph_ds.cpp/java*

Неявный граф

Некоторые графы *необязательно* сохранять в структуре данных или генерировать в явном виде, для того чтобы их можно было пройти или обработать. Такие графы называются *неявными графами*. Вы встретитесь с ними в следующих главах. Неявные графы могут быть двух видов:

---

[1] Пары объектов в C++ могут быть легко отсортированы. Критерием сортировки по умолчанию является сортировка по первому элементу, а затем по второму элементу, в случае равенства. В Java мы можем написать свой собственный класс `IntegerPair`/`IntegerTriple`, который реализует `Comparable`.

1) ребра графа могут быть легко определены.
   Пример 1. Навигация по 2D-карте сетки (см. рис. 2.5A). Вершины – это ячейки в двумерной сетке символов, где символ «.» представляет землю, а символ «#» – препятствие. Края можно легко определить: между двумя соседними элементами в сетке есть ребро, если они имеют общую границу *N/S/E/W* и если оба элемента представлены символом «.» (см. рис. 2.5B).
   Пример 2. Граф ходов шахматного коня на шахматной доске 8×8. Его вершины – клетки на шахматной доске. Две клетки на шахматной доске имеют общее ребро, если они разделяются двумя клетками по горизонтали и одной клеткой по вертикали (или двумя клетками по вертикали и одной клеткой по горизонтали). Граф для первых трех рядов клеток шахматной доски по горизонтали и четырех по вертикали показан на рис. 2.5C (многие другие вершины и ребра не показаны на этом рисунке);
2) ребра можно определить по некоторым правилам.
   Пример: граф содержит *N* вершин (`[1..N]`). Между двумя вершинами *i* и *j* есть ребро, если (*i* + *j*) – простое число. См. рис. 2.5D, где показан такой граф для случая *N* = 5, и еще несколько примеров в разделе 8.2.3.

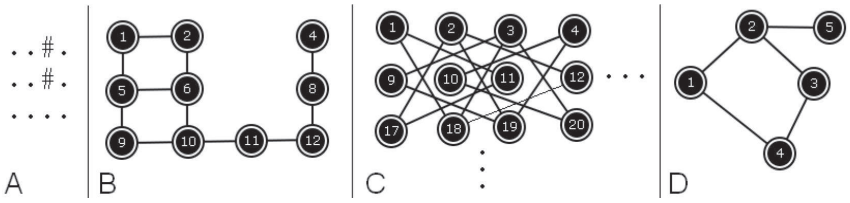


Рис. 2.5 ❖ Примеры неявных графов

Упражнение 2.4.1.1*. Создайте матрицу смежности, списки смежности и списки ребер для графов, показанных на рис. 4.1 (раздел 4.2.1) и на рис. 4.9 (раздел 4.2.9).
Подсказка: используйте инструмент визуализации структуры данных графа, показанный выше.

Упражнение 2.4.1.2*. Пусть задан (простой) граф в одном из возможных представлений – матрица смежности (МС), список смежности (СС) или список ребер (СР). Преобразуйте его в другое представление наиболее эффективным способом. В данной задаче имеется шесть возможных преобразований: МП в СП, МП в СР, СП в МП, СП в СР, СР в МП и СР в СП.

Упражнение 2.4.1.3. Если матрица смежности (простого) графа обладает следующим свойством: она совпадает с самой собой в транспонированном виде, то что это означает?

Упражнение 2.4.1.4*. Пусть задан (простой) граф, представленный матрицей смежности, выполните перечисленные ниже операции наиболее эффективным способом. После того как вы найдете, как это сделать для матрицы

смежности, проделайте то же упражнение для списков смежности, а затем со списками ребер.

1. Подсчитайте количество вершин V и направленных ребер E (предположим, что двунаправленное ребро эквивалентно двум направленным ребрам) графа.
2. * Подсчитайте входящую степень и исходящую степень заданной вершины v.
3. * Транспонируйте граф (измените направление каждого ребра).
4. * Проверьте, является ли граф полным графом K_n.
   Примечание. Полный граф – это простой неориентированный граф, в котором каждая пара различных вершин соединена одним ребром.
5. * Проверьте, является ли граф деревом (связным неориентированным графом с $E = V - 1$ ребрами).
6. * Проверьте, является ли граф звездным графом S_k.
   Примечание. Звездный граф S_k является полным двудольным графом $K_{1,k}$. Он представляет собой дерево с единственной внутренней вершиной и k листьями.

Упражнение 2.4.1.5*. Изучите другие возможные способы представления графов, отличные от описанных выше, особенно для хранения специальных графов.

2.4.2. Система непересекающихся множеств

Система непересекающихся множеств (Union-Find Disjoint Set, UFDS) – это структура данных, предназначенная для моделирования коллекции *непересекающихся множеств*, позволяющая эффективно[1] – т. е. за время $\approx O(1)$ – определить, к какому множеству принадлежит элемент (или проверить, принадлежат ли два элемента к одному множеству) и объединить два непересекающихся множества в одно. Такая структура данных может быть использована для решения задачи поиска компонент связности в неориентированном графе (см. раздел 4.2.3). Поместите каждую вершину в отдельное непересекающееся множество, затем переберите ребра графа и объедините каждые две вершины (непересекающихся множества), соединенные ребром. После этого вы легко сможете проверить, принадлежат ли две вершины одной и той же компоненте/множеству.

Эти, казалось бы, простые операции неэффективно поддерживаются в C++ STL и Java (`set` в C++ STL и `TreeSet` в Java соответственно, не предназначены для подобной цели). Иметь вектор из множеств (`vector` из элементов `set`) и просматривать каждый из них, чтобы найти, к какому множеству принадлежит элемент, слишком дорого! Использование `set_union` в C++ STL (в `algorithm`) не

[1] M операций этой структуры данных UFDS с эвристикой «сжатие пути» и ранговой эвристикой выполняются в $O(M \times \alpha(n))$. Однако, поскольку обратная функция Аккермана $\alpha(n)$ растет очень медленно, т. е. ее значение составляет чуть меньше 5 для размера набора входных данных $n \leqslant 1M$ в условиях олимпиады по программированию, мы можем рассматривать $\alpha(n)$ *как константу.*

будет достаточно эффективным, хотя оно и объединяет два множества за *линейное время*, так как нам все еще приходится иметь дело с перестановками содержимого вектора множеств! Для эффективной поддержки этих операций над множествами нам нужна более совершенная структура данных – UFDS.

Основным нововведением этой структуры данных является выбор представителя множества для представления множества. Если мы сможем гарантировать, что каждое множество представлено только одним уникальным элементом, то определение того, принадлежат ли элементы одному и тому же множеству, становится намного проще: репрезентативный «родительский» элемент можно использовать в качестве своего рода идентификатора для множества. Чтобы достичь этого, система непересекающихся множеств UFDS создает древовидную структуру, в которой непересекающиеся множества образуют лес из деревьев. Каждое дерево соответствует непересекающемуся множеству. Корень дерева определяется как репрезентативный элемент для множества. Таким образом, идентификатор репрезентативного элемента для множества можно получить, просто прослеживая цепочку родительских элементов до корня дерева, и, поскольку дерево может иметь только один корень, этот репрезентативный элемент можно использовать в качестве уникального идентификатора для множества.

Чтобы сделать это эффективно, мы храним индекс родительского элемента и (верхнюю границу) высоты дерева каждого множества (в нашей реализации `vi p` и `vi rank` соответственно). Помните, что `vi` – это наше сокращенное наименование для вектора целых чисел. `p[i]` хранит непосредственного родителя элемента `i`. Если элемент `i` является репрезентативным элементом некоторого непересекающегося множества, то `p[i] = i`, то есть петля. `rank[i]` возвращает (верхнюю границу) высоты дерева, имеющего корневой элемент *i*.

В этом разделе мы будем использовать пять непересекающихся множеств {0, 1, 2, 3, 4}, чтобы проиллюстрировать использование этой структуры данных. Мы инициализируем структуру данных таким образом, чтобы каждый элемент сам по себе был непересекающимся множеством с рангом 0, а родительский элемент каждого элемента изначально был эквивалентен самому себе.

Чтобы объединить два непересекающихся множества, мы устанавливаем репрезентативный элемент (корень) одного непересекающегося множества в качестве нового родителя репрезентативного элемента другого непересекающегося множества. Это эффективно объединяет два дерева в систему непересекающихся множеств. Таким образом, `unionSet(i, j)` приведет к тому, что оба элемента «`i`» и «`j`» будут иметь один и тот же репрезентативный элемент, прямо или косвенно. Для эффективности мы можем использовать информацию, содержащуюся в `vi rank`, чтобы установить репрезентативный элемент непересекающегося множества с *более высоким рангом* в качестве нового родителя непересекающегося множества с *более низким рангом*, тем самым *минимизируя* ранг результирующего дерева. Если оба ранга одинаковы, мы произвольно выбираем один из них в качестве нового родителя и увеличиваем результирующий ранг корня. Это ранговая эвристика. На рис. 2.6 вверху `unionSet(0, 1)` устанавливает для `p[0]` значение `1` и для `rank[1]` значение `1`. На рис. 2.6 в середине `unionSet(2, 3)` устанавливает для `p[2]` значение `3` и для `rank[3]` значение `1`.

А пока давайте предположим, что функция `findSet(i)` просто рекурсивно вызывает `findSet(p[i])`, чтобы найти репрезентативный элемент множества, возвращая `findSet(p[i])` всякий раз, когда `p[i] != i`, и `i` в противном случае. На рис. 2.6 (в нижней части), когда мы вызываем `unionSet(4, 3)`, у нас есть `rank[findSet(4)] = rank[4] = 0`, что меньше `rank[findSet(3)] = rank[3] = 1`, поэтому мы устанавливаем значение `p[4] = 3` без изменения высоты результирующего дерева – это работает ранговая эвристика. С помощью эвристики эффективно минимизируется путь от любого узла к репрезентативному элементу по цепочке «родительских» ссылок.

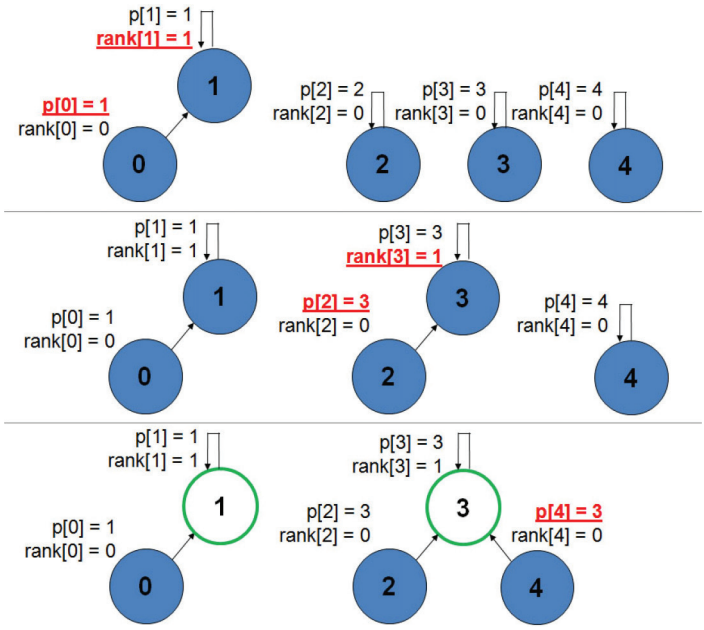


Рис. 2.6 ❖ `unionSet(0, 1)` → `(2, 3)` → `(4, 3)` и `isSameSet(0, 4)`

Внизу на рис. 2.6 `isSameSet(0, 4)` демонстрирует другую операцию для этой структуры данных. Функция `isSameSet(i, j)` просто вызывает `findSet(i)` и `findSet(j)` и проверяет, ссылаются ли они на один и тот же репрезентативный элемент. Если это так, то «`i`» и «`j`» оба принадлежат одному и тому же множеству. Здесь мы видим, что значение `findSet(0) = findSet(p[0]) = findSet(1) = 1` отличается от `findSet(4) = findSet(p[4]) = findSet(3) = 3`. Поэтому элемент 0 и элемент 4 принадлежат *различным* непересекающимся множествам.

Существует способ значительно ускорить функцию `findSet(i)`: сжатие пути. Всякий раз, когда мы находим репрезентативный (корневой) элемент непересекающегося множества, следуя по цепочке «родительских» ссылок из данного элемента, мы можем установить родительский элемент *всех* пройденных *элементов* так, чтобы он указывал прямо на корень. Любые последующие вызовы `findSet(i)` для затронутых элементов будут приводить к прохождению только одной ссылки. Это изменяет структуру дерева (чтобы сделать поиск `findSet(i)` более эффективным), но сохраняет фактическую структуру непересекающего-

ся множества. Поскольку это происходит при каждом вызове `findSet(i)`, комбинированный эффект заключается в том, чтобы время выполнения операции `findSet(i)` уменьшилось до чрезвычайно эффективного $O(M \times \alpha(n))$.

На рис. 2.7 показано «сжатие пути». Сначала мы вызываем `unionSet(0, 3)`. На этот раз мы устанавливаем значение `p[1] = 3` и обновляем значение `rank[3] = 2`. Теперь обратите внимание, что `p[0]` не изменилось, т. е. `p[0] = 1`. Это *косвенная* ссылка на (истинный) репрезентативный элемент из множества, т. е. `p[0] = 1 → p[1] = 3`. Функция `findSet(i)` фактически потребует более одного шага для прохождения цепочки «родительских» ссылок к корню. Однако как только во время этого прохода будет найден репрезентативный элемент (например, «x») для этого множества, функция `findSet(i)` сожмет путь, установив `p[i] = x`, то есть `findSet(0)` устанавливает значение `p[0] = 3`. Поэтому последующие вызовы из `findSet(i)` будут выполняться за $O(1)$. Эту простую стратегию называют эвристикой «сжатие пути». Обратите внимание, что `rank[3] = 2` теперь больше не отражает истинную высоту дерева. Вот почему `rank` выдает только верхнюю границу фактической высоты дерева. Наша реализация на C++ приведена ниже:

```
class UnionFind {                                                    // стиль ООП
private: vi p, rank;                                  // запомним: vi - это vector<int>
public:
  UnionFind(int N) { rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {                          // если это элемент другого множества
      int x = findSet(i), y = findSet(j);
      if (rank[x] > rank[y]) p[y] = x;                  // rank оставляет дерево коротким
      else {                 p[x] = y;
                             if (rank[x] == rank[y]) rank[y]++; }
} } };
```

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/ufds.html

Файл исходного кода: *ch2_08_unionfind_ds.cpp/java*

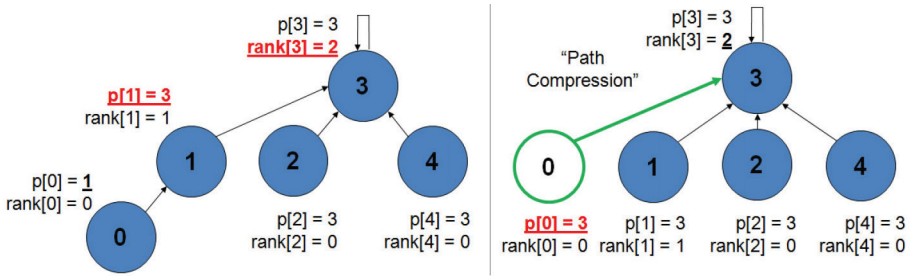


Рис. 2.7 ❖ `unionSet(0, 3)` → `findSet(0)`

Упражнение 2.4.2.1. В этой структуре данных обычно выполняются еще два запроса. Обновите код, представленный в данном разделе, для эффективной реализации следующих двух запросов: `int numDisjointSets()`, который возвращает количество непересекающихся множеств, находящихся в настоящее время в структуре, и `int sizeOfSet(int i)`, который возвращает размер множества, где в настоящее время содержится элемент `i`.

Упражнение 2.4.2.2*. Дано восемь непересекающихся множеств: {0, 1, 2, ..., 7}. Определите последовательность операций `unionSet (i, j)` для создания дерева с `rank = 3`. Возможно ли это для `rank = 4`?

Известные авторы структур данных

Джордж Буль (1815–1864) был английским математиком, философом и логиком. Он наиболее известен ученым, работающим в области компьютерных наук, как основатель булевой алгебры, основы современных цифровых компьютеров. Буль считается основателем области компьютерных наук.

Рудольф Байер (род. 1939) был профессором (почетным) информатики в Техническом университете Мюнхена. Он изобрел красно-черное дерево (RB), используемое в C++ STL `map` / `set`.

Георгий Адельсон-Вельский (1922 года рождения) – советский математик и ученый, работающий в области ВТ. Вместе с Евгением Михайловичем Ландисом изобрел АВЛ-дерево в 1962 году.

Евгений Михайлович Ландис (1921–1997) был советским математиком. Название дерева (АВЛ) является сокращением имен двух изобретателей: Адельсона-Вельского и самого Ландиса.

2.4.3. Дерево отрезков

В этом подразделе мы обсудим структуру данных, которая может эффективно отвечать на динамические запросы[1] *на отрезке*. Одним из таких запросов на отрезке является задача нахождения индекса минимального элемента массива в диапазоне `[i..j]`. Это более широко известно как задача запроса минимального значения из диапазона (Range Minimum Query, RMQ). Например, если задан массив `A` с размером $n = 7$ (ниже), `RMQ(1, 3) = 2`, так как индекс 2 содержит минимальный элемент среди `A[1]`, `A[2]` и `A[3]`. Чтобы проверить свое понимание RMQ, убедитесь, что в приведенном ниже массиве `A` `RMQ(3, 4) = 4`, `RMQ(0, 0) = 0`, `RMQ(0, 1) = 1` и `RMQ(0, 6) = 5`. Для следующих нескольких абзацев предполагается, что массив `A` все тот же.

Массив	Значения	18 17 13 19 15 11 20
A	Индексы	0 1 2 3 4 5 6

[1] Для динамических задач нам необходимо часто обновлять и запрашивать данные. Это делает методы предварительной обработки бесполезными.

Есть несколько способов реализовать RMQ. Один из тривиальных алгоритмов: просто выполнить несколько итераций перебора массива от индекса `i` до `j` и вывести индекс с минимальным значением, но этот алгоритм будет работать за время $O(n)$ на каждый запрос. Когда значение n велико и существует много запросов, такой алгоритм может оказаться неприменимым.

В этом разделе мы решаем задачу динамического RMQ с помощью *дерева отрезков*, которое является еще одним способом упорядочения данных в двоичном дереве. Существует несколько способов реализации дерева отрезков. Наша реализация использует ту же концепцию, что и компактный одномерный массив в двоичной куче, где мы применяем `vi` (наше сокращение для `vector<int>`) `st` для представления двоичного дерева. Индекс 1 (пропускаем индекс 0) является корнем, а левый и правый дочерние элементы индекса p – соответственно индексами $2 \times p$ и $(2 \times p) + 1$ (см. также обсуждение двоичной кучи в разделе 2.3). Значение `st[p]` является значением RMQ отрезка, связанного с индексом `p`.

Корень дерева отрезков представляет отрезок `[0, n-1]`. Для каждого отрезка `[L, R]`, хранящегося в индексе `p`, где `L! = R`, сегмент будет разбит на `[L, (L + R)/2]` и `[(L + R)/2 + 1, R]` в левой и правой вершинах. Левый подсегмент и правый подсегмент будут сохранены в индексах `2*p` и `(2*p) +1` соответственно. Когда `L = R`, ясно, что `st[p] = L` (или `R`). В противном случае мы будем рекурсивно строить дерево отрезков, сравнивая минимальное значение левого и правого подсегментов и обновляя `st[p]` сегмента. Этот процесс реализован в процедуре `build` ниже. Процедура `build` создает до $O(1 + 2 + 4 + 8 + \ldots + 2^{\log_2 n}) = O(2n)$ (меньших) сегментов и, следовательно, выполняется за $O(n)$. Однако, поскольку мы используем простую индексацию компактных массивов с одним основанием, нам нужно, чтобы `st` был как минимум размером $2*2^{(\log_2(n)+1)}$. В нашей реализации мы просто используем свободную верхнюю границу пространственной сложности $O(4n) = O(n)$. Для массива `A` выше соответствующее дерево отрезков показано на рис. 2.8 и 2.9.

Когда дерево отрезков готово, RMQ-запрос может быть выполнен за $O(\log n)$. Ответ на `RMQ(i, i)` тривиален – просто верните `i`. Однако для общего случая `RMQ(i, j)` необходимы дальнейшие проверки. Пусть `p1 = RMQ(i, (i + j)/2)` и `p2 = RMQ((i + j)/2 + 1, j)`. Тогда `RMQ(i, j)` равно `p1`, если `A[p1] ≤ A[p2]`, или `p2` в противном случае. Этот процесс реализован в подпрограмме `rmq`, приведенной ниже.

Рассмотрим в качестве примера запрос `RMQ(1, 3)`. Процесс на рис. 2.8 выглядит следующим образом: начнем с корня (индекс 1), который представляет сегмент `[0, 6]`. Мы не можем использовать сохраненное минимальное значение для сегмента `[0, 6] = st[1] = 5` в качестве ответа на запрос `RMQ(1, 3)`, так как это минимальное значение для более крупного сегмента[1], чем выбранный нами `[1, 3]`. От корня нам нужно только перейти к левому поддереву, поскольку корень правого поддерева представляет сегмент `[4, 6]`, который находится за пределами[2] выбранного диапазона в `RMQ(1, 3)`.

---

[1] Сегмент `[L, R]` считается большим, чем диапазон запроса `[i, j]`, если `[L, R]` не находится вне диапазона запроса и не находится внутри диапазона запроса (см. другие сноски).

[2] Сегмент `[L, R]` называется сегментом вне диапазона запроса `[i, j]`, если `i > R || J < L`.

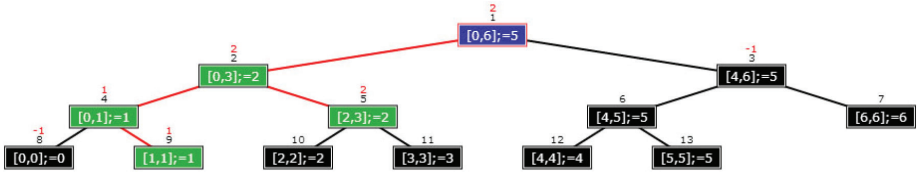


Рис. 2.8 ❖ Дерево сегментов массива A = {18, 17, 13, 19, 15, 11, 20} и `RMQ(1, 3)`

Теперь мы находимся в корне левого поддерева (индекс 2), представляющего сегмент `[0, 3]`. Этот сегмент `[0, 3]` все еще больше, чем нужный нам `RMQ(1, 3)`. Фактически `RMQ(1, 3)` пересекает как левый подсегмент `[0, 1]` (индекс 4), так и правый подсегмент `[2, 3]` (индекс 5) сегмента `[0, 3]`, поэтому мы должны исследовать оба поддерева (подсегмента).

Левый сегмент `[0, 1]` (индекс 4) из `[0, 3]` (индекс 2) еще не находится внутри `RMQ(1, 3)`, поэтому необходимо другое разделение. От сегмента `[0, 1]` (индекс 4) мы переместимся вправо к сегменту `[1, 1]` (индекс 9), который теперь находится внутри[1] `[1, 3]`. На данный момент мы знаем, что `RMQ(1, 1) = st[9] = 1`, и мы можем вернуть это значение стороне, выдавшей запрос. Правый сегмент `[2, 3]` (индекс 5) из `[0, 3]` (индекс 2) находится внутри требуемого `[1, 3]`. Из сохраненного значения внутри этой вершины мы знаем, что `RMQ(2, 3) = st[5] = 2`. Нам не нужно перемещаться дальше вниз.

Вернемся к сегменту `[0, 3]` (индекс 2): теперь мы имеем `p1 = RMQ(1, 1) = 1` и `p2 = RMQ(2, 3) = 2`. Поскольку `A[p1]> A[p2]`, так как `A[1] = 17` и `A[2] = 13`, теперь мы имеем `RMQ(1, 3) = p2 = 2`. Это окончательный ответ.

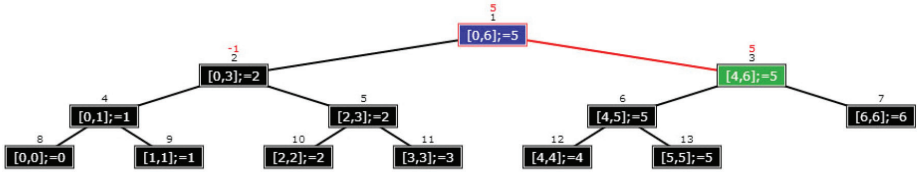


Рис. 2.9 ❖ Дерево сегментов массива A = {18, 17, 13, 19, 15, 11, 20} и `RMQ(4, 6)`

Теперь рассмотрим другой пример: `RMQ(4, 6)`. Процесс на рис. 2.9 выглядит следующим образом: мы снова начинаем с корневого сегмента `[0, 6]` (индекс 1). Так как корневой сегмент больше, чем `RMQ(4, 6)`, мы перемещаемся вправо к сегменту `[4, 6]` (индекс 3), поскольку сегмент `[0, 3]` (индекс 2) находится снаружи. Так как этот сегмент точно представляет `RMQ(4, 6)`, мы просто возвращаем индекс минимального элемента, который хранится в этой вершине, – этот индекс равен 5. Таким образом, `RMQ(4, 6) = st[3] = 5`.

Подобная структура данных позволяет нам избежать обхода ненужных частей дерева! В худшем случае у нас есть *два* пути от корня к листу, что составляет всего $O(2 \times \log(2n)) = O(\log n)$. Пример: в `RMQ(3, 4) = 4` у нас есть один путь от корня к листу от `[0, 6]` до `[3, 3]` (индекс 1 → 2 → 5 → 11) и другой от корня к листу путь от `[0, 6]` до `[4, 4]` (индекс 1 → 3 → 6 → 12).

---

[1] Сегмент `[L, R]` находится внутри диапазона запросов `[i, j]`, если `L ≥ i && R ≤ j`.

Если массив *A* является статическим (то есть неизменным после создания его экземпляра), то использование дерева сегментов для решения задачи RMQ *избыточное*, поскольку существует решение с применением методов динамического программирования (DP), которое требует $O(n \log n)$ за один раз предварительной обработки и позволяет получить производительность $O(1)$ за один запрос RMQ. Это решение с использованием динамического программирования будет обсуждаться позже в разделе 9.33.

Дерево отрезков полезно, если исходный массив часто обновляется (динамически). Например, если теперь значение A[5] изменяется с 11 на 99, то нам просто нужно обновить вершины вдоль пути от листа к корневому элементу за $O(\log n)$. Проследите путь: [5, 5] (индекс 13, st[13] остается без изменений) → [4, 5] (индекс 6, теперь st[6] = 4) → [4, 6] (индекс 3, теперь st[3] = 4) → [0, 6] (индекс 1, теперь st[1] = 2) на рис. 2.10. Для сравнения, в решении DP, представленном в разделе 9.33, требуется предварительная обработка данных за $O(n \log n)$ для обновления структуры, и потому оно неэффективно для таких динамических обновлений.

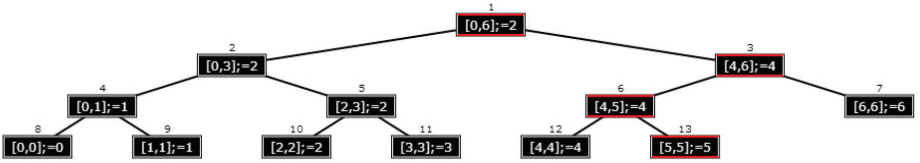


Рис. 2.10 ❖ Обновление массива A; новые значения: {18, 17, 13, 19, 15, **99**, 20}

Наша реализация дерева отрезков приведена ниже. Код, показанный здесь, поддерживает только статические запросы RMQ (*динамические* обновления оставлены в качестве упражнения для читателя).

```
class SegmentTree {                              // дерево отрезков хранится как массив кучи
private: vi st, A;                          // напомним, что такое vi: typedef vector<int> vi;
  int n;
  int left (int p) { return p << 1; }              // аналогично операциям с двоичной кучей
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                                                  // O(n)
    if (L == R)             // поскольку L == R, можно использовать любое из этих значений
      st[p] = L;                                                          // сохраняем индекс
    else {                                                  // рекурсивно вычисляем значения
      build(left(p) , L              , (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R          );
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }

  int rmq(int p, int L, int R, int i, int j) {                                  // O(log n)
    if (i > R || j < L) return -1;              // текущий сегмент - вне диапазона запроса
    if (L >= i && R <= j) return st[p];                          // внутри диапазона запроса
                        // вычисляем минимальную позицию в левой и правой частях интервала
    int p1 = rmq(left(p) , L              , (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R          , i, j);
```

```
    if (p1 == -1) return p2;    // если мы пытаемся получить доступ до сегмента вне запроса
    if (p2 == -1) return p1;                                                  // см. выше
    return (A[p1] <= A[p2]) ? p1 : p2;                              // как в функции build
  }

public:
  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();          // копируем содержимое для локального использования
    st.assign(4 * n, 0);                        // создаем достаточно большой вектор нулей
    build(1, 0, n - 1);                                          // рекурсивный вызов build
  }

  int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }                  // перегрузка
};

int main() {
  int arr[] = { 18, 17, 13, 19, 15, 11, 20 };                          // исходный массив
  vi A(arr, arr + 7);
  SegmentTree st(A);
  printf("RMQ(1, 3) = %d\n", st.rmq(1, 3));                          // ответ = индекс 2
  printf("RMQ(4, 6) = %d\n", st.rmq(4, 6));                          // ответ = индекс 5
} // return 0;
```

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/segmenttree.html

Файл исходного кода: *ch2_09_segmenttree_ds.cpp/java*

Упражнение 2.4.3.1*. Нарисуйте дерево отрезков, соответствующее массиву A = {10, 2, 47, 3, 7, 9, 1, 98, 21}. Напишите ответ для запросов: `RMQ(1, 7)` и `RMQ(3, 8)`. Подсказка: используйте визуализацию дерева отрезков, приведенную выше.

Упражнение 2.4.3.2*. В этом разделе книги мы увидели, как деревья отрезков могут использоваться для получения ответа на запросы минимума на отрезке (RMQ). Деревья отрезков также могут использоваться для получения ответа на запросы суммы на отрезке (`RSQ(i, j)`), то есть суммы элементов массива от *I* до *j*: `A[i] + A[i + 1] + ... + A[j]`. Измените приведенный выше код дерева отрезков для работы с RSQ.

Упражнение 2.4.3.3. Используя дерево отрезков, аналогичное описанному выше в упражнении 2.4.3.1, получите ответ для запросов `RSQ(1, 7)` и `RSQ(3, 8)`. Является ли дерево отрезков хорошим подходом для решения задачи, если массив A никогда не изменяется? (также см. раздел 3.5.2).

Упражнение 2.4.3.4*. Как было упомянуто в тексте этой главы, в приведенном выше коде дерева отрезков отсутствует (точечная) операция обновления `update`. Добавьте функцию `update`, работающую за $O(\log n)$, чтобы обновить значение определенного индекса (точки) в массиве *A* и одновременно обновить соответствующее дерево отрезков.

Упражнение 2.4.3.5*. Операция обновления (точечная), показанная в тексте этой главы, изменяет только значение определенного индекса в массиве `A`. Что,

если мы удалим существующие элементы массива A или вставим новые элементы в массив A? Можете ли вы объяснить, что произойдет с приведенным выше кодом дерева отрезков, и что вы должны сделать для того, чтобы он работал правильно?

Упражнение 2.4.3.6*. Существует также еще одна важная операция с деревом отрезков, которая еще не обсуждалась, – операция обновления *на отрезке*. Предположим, что определенный подмассив массива A обновлен до определенного общего значения. Можем ли мы эффективно обновить дерево отрезков? Изучите и решите задачу UVa 11402 – Ahoy Pirates – это задача, где выполняется обновление диапазона.

2.4.4. Дерево Фенвика

Дерево Фенвика, также известное как двоичное индексированное дерево (Binary Indexed Tree, BIT), было изобретено Питером М. Фенвиком в 1994 году [18]. В этой книге мы будем использовать термин «дерево Фенвика», а не BIT, чтобы отличать этот класс операций от обычных операций с битами. Дерево Фенвика является полезной структурой данных для реализации динамических частотных таблиц. Предположим, у нас есть[1] оценки за выполненное тестовое задание для m = 11 студентов f = {2,4,5,5,6,6,6,7,7,8,9}, где тестовые оценки представляют собой целочисленные значения в диапазоне [1..10], в табл. 2.1 приведена частота каждого отдельного результата теста ∈ [1..10] и накопленная частота результатов теста в диапазоне [1..i], обозначенная как cf[i], представляющая собой сумму частот результатов теста 1, 2, ..., i.

Таблица 2.1. Пример таблицы со значениями накопленной частоты

Индекс/оценка	Частота f	Накопленная частота cf	Краткий комментарий
0	–	–	Индекс 0 игнорируется (как служебное значение)
1	0	0	cf[1] = f[1] = 0.
2	1	1	cf[2] = f[1] + f[2] = 0 + 1 = 1.
3	0	1	cf[3] = f[1] + f[2] + f[3] = 0 + 1 + 0 = 1.
4	1	2	cf[4] = cf[3] + f[4] = 1 + 1 = 2.
5	2	4	cf[5] = cf[4] + f[5] = 2 + 2 = 4.
6	3	7	cf[6] = cf[5] + f[6] = 4 + 3 = 7.
7	2	9	cf[7] = cf[6] + f[7] = 7 + 2 = 9.
8	1	10	cf[8] = cf[7] + f[8] = 9 + 1 = 10.
9	1	11	cf[9] = cf[8] + f[9] = 10 + 1 = 11.
10	0	11	cf[10] = cf[9] + f[10] = 11 + 0 = 11.

Таблица со значениями накопленной частоты также может использоваться в качестве решения задачи запроса суммы на отрезке (RSQ), упомянутой

[1] Результаты тестов отсортированы в порядке возрастания для простоты, в произвольном случае они могут быть не отсортированы.

в упражнении 2.4.3.2*. Она хранит RSQ(1, i) ∀i ∈ [1..n], где n – самый большой целочисленный индекс/оценка[1]. В приведенном выше примере мы имеем n = 10, RSQ(1, 1) = 0, RSQ(1, 2) = 1, ..., RSQ(1, 6) = 7, ..., RSQ(1, 8) = 10, ..., и RSQ(1, 10) = 11. Теперь мы можем получить ответ на запрос RSQ для произвольного диапазона RSQ(i, j), когда $i \neq 1$, беря разность значений RSQ(1, j) - RSQ(1, i - 1). Например, RSQ(4, 6) = RSQ(1, 6) - RSQ(1, 3) = 7 - 1 = 6.

Если частоты являются *статическими*, то таблицу значений накопленной частоты (как, например, в табл. 2.1) можно эффективно заполнить, рассчитав ее значения с помощью простого цикла $O(n)$. Сначала задайте значение cf[1] = f[1]. Тогда для i ∈ [2..n] вычислим cf[i] = cf[i - 1] + f[i]. Это будет обсуждаться далее в разделе 3.5.2. Однако когда частоты часто обновляются (увеличиваются или уменьшаются) и впоследствии нередко выполняются запросы RSQ, лучше использовать динамическую структуру данных.

Вместо того чтобы применять дерево отрезков для реализации динамической таблицы значений накопленной частоты, мы можем реализовать гораздо более простое дерево Фенвика (сравните исходный код для обоих вариантов реализации, представленных в этом разделе и в предыдущем разделе 2.4.3). Возможно, это одна из причин, по которой дерево Фенвика включено в программу IOI [20]. Операции на дереве Фенвика также чрезвычайно эффективны, поскольку они используют методы быстрого выполнения побитовых операций (см. раздел 2.2).

В этом разделе мы будем широко использовать функцию LSOne(i) (которая на самом деле представляет собой операцию (i & (-i))); мы назвали ее так, используя название из оригинальной статьи [18]. В разделе 2.2 мы видели, что операция (i & (-i)) дает нам первый младший значащий бит в i.

Для дерева Фенвика типичная реализация – в виде массива (мы используем vector для гибкости его размера). Дерево Фенвика – это дерево, которое индексируется *битами* его *целочисленных* ключей. Эти целочисленные ключи находятся в фиксированном диапазоне [1..n] – исключая[2] индекс 0. Для олимпиад по программированию значение n может приближаться к $\approx 1M$, так что дерево Фенвика охватывает диапазон [1..1M] – это достаточно много для большинства практических (предлагаемых на конкурсе) задач. В приведенной выше табл. 2.1 оценки [1..10] представляют собой целочисленные ключи в соответствующем массиве с размером n = 10 и m = 11 элементов данных.

Определим для массива дерева Фенвика имя ft. Тогда элемент с индексом i отвечает за элементы в диапазоне [i-LSOne(i)+1..i], а ft[i] хранит накопленную частоту элементов {i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, .., i}. На рис. 2.11 значение ft[i] показано в кружке над индексом i, а диапазон [i-LSOne(i)+1..i] показан в виде круга и полосы (если диапазон охватывает более чем один индекс), расположенной над индексом i. Мы видим, что ft[4] = 2

---

[1] Необходимо различать m = количество точек данных и n = наибольшее целочисленное значение среди m точек данных. Значение n в дереве Фенвика немного отличается от других структур данных в этой книге.

[2] Мы решили следовать исходной реализации [18], которая игнорирует индекс 0, чтобы облегчить понимание операций с битами в дереве Фенвика. Обратите внимание, что индекс 0 не имеет включенных битов.

отвечает за диапазон [4-4+1..4] = [1..4], ft[6] = 5 отвечает за диапазон [6-2+1..6] = [5..6], ft[7] = 2 отвечает за диапазон [7-1+1..7] = [7..7], ft[8] = 10 отвечает за диапазон [8-8+1..8] = [1..8] и т. д.[1]

При таком размещении, если мы хотим получить накопленную частоту в интервале между [1..b], т. е. rsq (b), мы просто добавляем ft[b], ft[b'], ft[b''], ..., до тех пор, пока индекс b^i не станет равен 0. Эта последовательность индексов получается вычитанием наименьшего значащего бита через выражение операций над битами: b '= b - LSOne (b). Итерация данной битовой операции эффективно *отбрасывает* наименьший значащий бит в значении b на каждом шаге. Поскольку целое число b содержит только $O(\log b)$ бит, rsq (b) выполняется за $O(\log n)$, когда b = n. На рис. 2.11 rsq(6) = ft[6] + ft[4] = 5 + 2 = 7. Обратите внимание, что индексы 4 и 6 отвечают за диапазон [1..4] и [5..6] соответственно. Комбинируя их, мы получаем весь диапазон [1..6]. Индексы 6, 4 и 0 связаны в двоичной форме: $b = 6_{10} = (11\underline{0})_2$ можно преобразовать в $b' = 4_{10} = (\underline{1}00)_2$, а затем в $b'' = 0_{10} = (000)_2$.

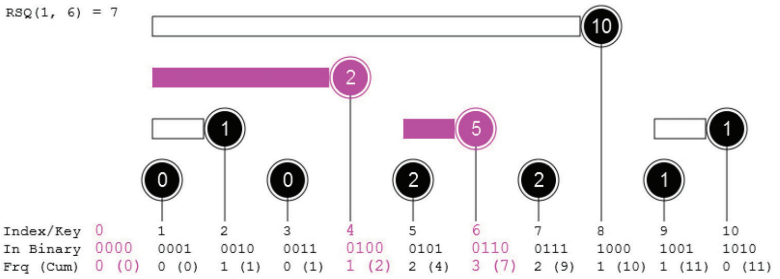


Рис. 2.11 ❖ Пример rsq(6)

Таким образом, операция i +/- LSOne(i) просто возвращает i, когда i = 0. Индекс 0 также используется в качестве завершения в функции rsq.

При наличии rsq(b) получить накопленную частоту между двумя индексами [a..b], где a! = 1, просто: оцените rsq(a, b) = rsq(b) - rsq(a - 1). Например, если мы хотим вычислить rsq(4, 6), то можем просто вернуть rsq(6) - rsq(3) = (5 + 2) - (0 + 1) = 7 - 1 = 6. Опять же, эта операция выполняется за $O(2 \times \log b) \approx O(\log n)$, когда b = n. На рис. 2.12 показано значение rsq(3).

При обновлении значения элемента по индексу k путем изменения его значения на v (обратите внимание, что v может быть как положительным, так и отрицательным), то есть вызывая adjust(k, v), мы должны обновить ft[k], ft[k'], ft[k''], ..., пока индекс k^i не превысит n. Данная последовательность индексов получается с помощью следующей итеративной операции над битами: k' = k + LSOne(k). Начиная с любого целого числа k, операция adjust(k, v) будет выполнять не более $O(\log n)$ шагов до тех пор, пока выполняется условие k > n. На рис. 2.13 операция adjust(5, 1) будет влиять на (добавлять +1 к) ft[k] при

[1] В этой книге мы не будем подробно описывать, почему эта схема работает, а вместо этого покажем, что она обеспечивает эффективные операции $O(\log n)$: операцию обновления и запрос RSQ. Заинтересовавшимся подробностями читателям рекомендуется прочитать [18].

индексах $k = 5_{10} = (101)_2$, $k' = (101)_2 + (00\underline{1})_2 = (110)_2 = 6_{10}$ и $k'' = (110)_2 + (0\underline{1}0)_2 = (1000)_2 = 8_{10}$, обновляя значения согласно выражению, приведенному выше. Обратите внимание, что если вы продолжаете линию вверх от индекса 5 на рис. 2.13, то видите, что линия действительно пересекает отрезки, за которые отвечает индекс 5, индекс 6 и индекс 8.

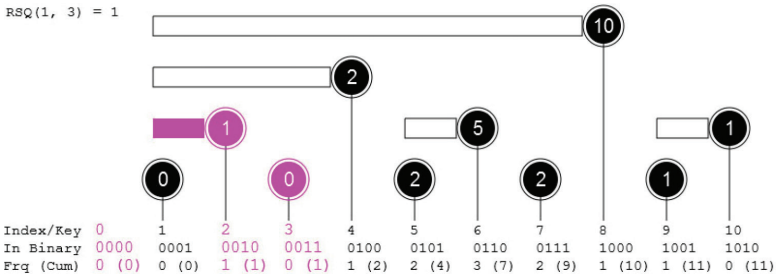


Рис. 2.12 ❖ Пример `rsq(3)`

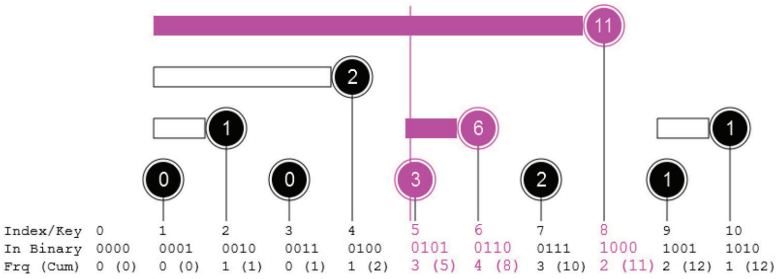


Рис. 2.13 ❖ Пример `adjust(5, 1)`

Таким образом, дерево Фенвика поддерживает как RSQ, так и операции обновления, имея пространственную сложность $O(n)$ и временную сложность $O(\log n)$, при условии что существует набор из m целочисленных ключей, которые находятся в диапазоне `[1..n]`. Это делает дерево Фенвика идеальной структурой данных для решения *динамической* задачи RSQ с дискретными массивами (*статическая* задача RSQ может быть решена с помощью простой предварительной обработки данных (за $O(n)$ в операциях обработки данных и за $O(1)$ на каждый запрос, как было сказано ранее). Наш *короткий* вариант реализации базового дерева Фенвика на C++ приведен ниже.

```
class FenwickTree {
private: vi ft;                          // напомним, что такое vi: typedef vector<int> vi;
public: FenwickTree(int n) { ft.assign(n + 1, 0); }                  // иниц. n + 1 нулей
  int rsq(int b) {                                                   // возвращаем RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }                                          // Примечание: LSOne(S) (S & (-S))
  int rsq(int a, int b) {                                            // возвращаем RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }
  // корректируем значение k-го элемента с помощью v (v может быть + ve /inc или -ve/dec)
  void adjust(int k, int v) {                          // Примечание: n = ft.size() - 1
```

```
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};

int main() {
  int f[] = { 2,4,5,5,6,6,6,7,7,8,9 };                                  // m = 11 оценки
  FenwickTree ft(10);                      // объявляем дерево Фенвика для диапазона [1..10]
                          // добавляем эти оценки вручную по одному в пустое дерево Фенвика
  for (int i = 0; i < 11; i++) ft.adjust(f[i], 1);                  // потребует O(k log n)
  printf("%d\n", ft.rsq(1, 1));  // 0 => ft[1] = 0
  printf("%d\n", ft.rsq(1, 2));  // 1 => ft[2] = 1
  printf("%d\n", ft.rsq(1, 6));  // 7 => ft[6] + ft[4] = 5 + 2 = 7
  printf("%d\n", ft.rsq(1, 10)); // 11 => ft[10] + ft[8] = 1 + 10 = 11
  printf("%d\n", ft.rsq(3, 6));  // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1
  ft.adjust(5, 2);               // демонстрация обновления дерева
  printf("%d\n", ft.rsq(1, 10)); // теперь 13
} // return 0;
```

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/bit.html

Файл исходного кода: *ch2_10_fenwicktree_ds.cpp/java*

Упражнение 2.4.4.1. Простое упражнение, состоящее из двух основных операций над битами, используемых в дереве Фенвика: каковы значения `90 - LSOne(90)` и `90 + LSOne(90)`?

Упражнение 2.4.4.2. Что, если задача, которую вы хотите решить, содержит элемент с целочисленным ключом 0? Напомним, что стандартный диапазон целочисленных ключей в коде нашей библиотеки равен `[1..n]` и что в этой реализации нельзя использовать индекс 0, поскольку он применяется в качестве завершающего условия для операции `rsq`.

Упражнение 2.4.4.3. Что, если в задаче, которую вы хотите решить, используются нецелые ключи? Например, что, если результаты тестов, показанные в табл. 2.1 выше, равны `f = {5.5, 7.5, 8.0, 10.0}` (т. е. допускается либо 0, либо 5 после десятичного знака)? Что, если результаты теста `f = {5.53, 7.57, 8.10, 9.91}` (т. е. баллы округляются с точностью до двух знаков после десятичной запятой)?

Упражнение 2.4.4.4. Дерево Фенвика поддерживает дополнительную операцию, которую мы решили оставить читателю в качестве упражнения: найти наименьший индекс с заданной накопленной частотой. Например, нам может потребоваться определить минимальный индекс/балл `i` в табл. 2.1 так, чтобы в диапазоне `[1..i]` было представлено не менее семи учащихся (индекс/оценка равен 6 для данного случая). Реализуйте эту функцию.

Упражнение 2.4.4.5*. Решите эту задачу динамического RSQ: UVa 12086 – Potentiometers, используя и дерево сегментов, и дерево Фенвика. Какое решение легче найти в этом случае?
Также см. табл. 2.2 для сравнения этих двух структур данных.

Упражнение 2.4.4.6*. Расширьте дерево Фенвика от одномерной реализации в двумерную.

Упражнение 2.4.4.7*. Деревья Фенвика обычно используются для одиночных (точечных) обновлений и запросов диапазона (суммы). Покажите, как использовать дерево Фенвика для *обновления диапазона* и одиночных запросов. Например, если задано множество интервалов с небольшими диапазонами (от 1 до максимум 1 млн), определите количество интервалов, охватывающих индекс i.

Известные авторы структуры данных

Питер М. Фенвик – почетный доцент Университета Окленда. Он изобрел двоичное индексированное дерево в 1994 году [18] как «кумулятивные таблицы частот арифметического сжатия». С тех пор двоичное индексированное дерево было включено в программу IOI [20] и использовалось во многих олимпиадных задачах из-за его эффективной, но простой в реализации структуры данных.

Таблица 2.2. Сравнение дерева сегментов и дерева Фенвика

Функция	Дерево сегментов	Дерево Фенвика
Построение дерева из массива	$O(n)$	$O(m \log n)$
Динамический RMin/MaxQ	OK	Очень ограничено
Динамический RSQ	OK	OK
Сложность запроса	$O(\log n)$	$O(\log n)$
Сложность «точечных» обновлений	$O(\log n)$	$O(\log n)$
Длина кода	Более длинный	Менее длинный

Задачи по программированию, решаемые с помощью рассмотренных структур данных

- Задачи, решаемые с помощью структур данных графа
  1. **UVa 00599 – The Forrest for the Trees *** (v – e = количество компонент связности составьте `bitset` размером 26 для подсчета количества вершин, имеющих некоторое ребро. Примечание: задачу также можно решить с помощью моделирования системы непересекающихся множеств (Union-Find))
  2. **UVa 10895 – Matrix Transpose *** (транспонируйте список смежности)
  3. UVa 10928 – My Dear Neighbours (подсчитайте число связей узла)
  4. UVa 11550 – Demanding Dilemma (графическое представление графа, матрица инцидентности)
  5. **UVa 11991 – Easy Problem from… *** (используйте AdjList)

  См. также дополнительные задачи с графами в главе 4.
- Система непересекающихся множеств (Union-Find Disjoint Sets)
  1. **UVa 00793 – Network Connections *** (тривиально; применение непересекающихся множеств)

2. UVa 01197 – The Suspects (LA 2817, Гаосюн'03, компоненты связности)
3. UVa 10158 – War (использование непересекающихся множеств с некоторой особенностью; хранение в памяти списка врагов)
4. UVa 10227 – Forests (объедините два непересекающихся множества, если они непротиворечивы)
5. **UVa 10507 – Waking up brain** * (непересекающиеся множества упрощают эту проблему)
6. UVa 10583 – Ubiquitous Religions (считайте непересекающиеся множества после всех объединений)
7. UVa 10608 – Friends (найдите набор с наибольшим элементом)
8. UVa 10685 – Nature (найдите набор с наибольшим элементом)
9. **UVa 11503 – Virtual Friends** * (сохраните заданный атрибут (размер) в элементе rep)
10. UVa 11690 – Money Matters (проверьте, равняется ли 0 общая сумма денег каждого участника описываемых событий)

- Древовидные структуры данных
  1. UVa 00297 – Quadtrees (простая задача с деревом квадратов)
  2. UVa 01232 – SKYLINE (LA 4108, Сингапур'07; простая задача, если размер входного файла мал; но так как $n \leqslant 100\,000$, мы должны использовать дерево отрезков; обратите внимание, что эта задача не подразумевает применения RSQ/RMQ)
  3. **UVa 11235 – Frequent Values** * (запрос максимального диапазона)
  4. UVa 11297 – Census (дерево квадратов с обновлениями или использование двумерного дерева отрезков)
  5. UVa 11350 – Stern-Brocot Tree (простой вопрос о структуре данных дерева)
  6. UVa 11402 – Ahoy Pirates (сегмент дерева с *ленивыми* обновлениями (*lazy* updates))
  7. UVa 12086 – Potentiometers (LA 2191, Дакка'06; задача на составление запроса суммы динамического диапазона; решается с помощью дерева Фенвика или дерева отрезков)
  8. ***UVa 12532 – Interval Product*** * (умное использование дерева Фенвика / дерева сегментов)

     Также см. структуры данных (DS) как часть решения более сложных задач в главе 8.

2.5. Решения упражнений, не помеченных звездочкой

Упражнение 2.2.1*. Подвопрос 1: сначала отсортируйте S за $O(n \log n)$, а затем выполните последовательный просмотр данных за $O(n)$, начиная со второго элемента, чтобы проверить, совпадают ли текущее целое значение и предыдущее целое значение (также прочитайте решение для упражнения 1.2.10, задача 4). Подвопрос 6: прочтите вступительный абзац главы 3 и подробное обсуждение в разделе 9.29. Решения для других подвопросов не приводятся.

Упражнение 2.2.2. Ответы (кроме подвопроса 7):

1) S & (N – 1);
2) (S & (S – 1)) == 0;
3) S & (S – 1);
4) S || (S + 1);
5) S & (S + 1);
6) S || (S – 1).

Упражнение 2.3.1. Поскольку коллекция динамическая, мы будем часто выполнять запросы на вставку и удаление. При операциях вставки может измениться порядок сортировки. Если мы храним информацию в статическом массиве, нам придется использовать одну $O(n)$ итерацию сортировки вставки после каждой вставки и удаления (чтобы закрыть пробел в массиве). Это неэффективно.

Упражнение 2.3.2.

1. `search(71)`: корень (15) → 23 → 71 (найдено);
   `search(7)`: корень (15) → 6 → 7 (найдено);
   `search(22)`: корень (15) → 23 → пустое левое поддерево (не найдено).
2. В итоге у нас будет то же самое BST, что и на рис. 2.2.
3. Чтобы найти элемент min/max, мы можем начать с корня и продолжать двигаться влево/вправо, пока не встретим вершину без левых/правых поддеревьев соответственно. Эта вершина и является ответом.
4. Мы получим отсортированный набор выходных данных: 4, 5, 6, 7, 15, 23, 50, 71. См. раздел 4.7.2, если вы незнакомы с алгоритмом обхода двоичного дерева с порядковой выборкой.
5. `successor(23)`: найдите минимальный элемент поддерева с корнем справа от 23, который является поддеревом с корнем в 71. Ответ: 50.
   `successor(7)`: 7 не имеет правого поддерева, поэтому 7 должно быть максимумом определенного поддерева.
   Это поддерево является поддеревом с корнем в 6. Родителем 6 является 15, и 6 – левое поддерево 15. По свойству BST 15 должно быть преемником 7.
   `successor(71)`: 71 является самым большим элементом и не имеет преемника.
   Примечание. Алгоритм поиска предшественника узла аналогичен.
6. `delete(5)`: мы просто удаляем 5, который является листом, из BST.
   `delete(71)`: поскольку 71 – это внутренняя вершина с одним дочерним элементом, мы не можем просто удалить 71, так как это приведет к разделению BST на два компонента. Вместо этого мы можем переставить поддерево с корнем у родителя 71 (то есть 23), в результате чего у 23 будет 50 в качестве правильного потомка.
7. `delete(15)`: поскольку 15 – это вершина с двумя дочерними элементами, мы не можем просто удалить 15, так как это приведет к разделению BST на *три* компонента. Чтобы решить эту проблему, нам нужно найти наследника 15 (то есть 23) и использовать наследника для замены 15. Затем мы удаляем старый элемент 23 из BST (теперь это не проблема). *Примечание:* мы также можем использовать `predecessor(key)` вместо `successor(key)`

во время операции delete(key) для случая, когда ключ имеет два дочерних элемента.

Упражнение 2.3.3*. Для подзадачи 1 мы запускаем обход двоичного дерева поиска с симметричным обходом за $O(n)$ и проверяем, отсортированы ли значения. Решения других подзадач не показаны.

Упражнение 2.3.6. Ответы:

1) Insert(26): вставьте 26 в качестве левого поддерева 3, поменяйте местами 26 с 3, затем поменяйте местами 26 с 19 и остановитесь. Массив невозрастающей кучи A теперь содержит {–, 90, 26, 36, 17, 19, 25, 1, 2, 7, 3};
2) ExtractMax(): поменяйте местами 90 (максимальный элемент, который будет найден после того, как мы зафиксируем свойство невозрастающей кучи (Max Heap)) и 3 (текущий самый нижний крайний правый лист / последний элемент в невозрастающей куче), поменяйте местами 3 и 36, поменяйте местами 3 и 25 и остановитесь. Массив невозрастающей кучи A теперь содержит {–, 36, 26, 25, 17, 19, 3, 1, 2, 7}.

Упражнение 2.3.7. Да, убедитесь, что все индексы (вершины) удовлетворяют свойству невозрастающей пирамиды (Max Heap).

Упражнение 2.3.16. Используйте set в C++ STL (или TreeSet в Java), поскольку он представляет собой сбалансированное BST, которое поддерживает $O(\log n)$ динамических вставок и удалений. Мы можем использовать обход по порядку для печати данных в BST в отсортированном порядке (просто используйте итераторы C++ (iterator) или Java (Iterator)).

Упражнение 2.3.17. Использование map в C++ STL (TreeMap в Java) и переменной счетчика. Также можно использовать хеш-таблицу, однако это не обязательно для соревнований по программированию. Этот прием довольно часто применяется в различных (конкурсных) задачах. Пример использования:

```
char str[1000];
map<string, int> mapper;
int i, idx;
for (i = idx = 0; i < M; i++) {                                    // idx начинается с 0
  scanf("%s", &str);
  if (mapper.find(str) == mapper.end())        // если это первое встретившееся выполненное
условие
  // в качестве альтернативы, мы также можем проверить условие:
  // mapper.count(str) больше, чем 0
    mapper[str] = idx++;              // присвоить str текущее значение idx и увеличить idx
}
```

Упражнение 2.4.1.3. Граф не ориентирован.

Упражнение 2.4.1.4*. Подзадача 1: для подсчета количества вершин графа: матрица смежности / список смежности → получить количество строк; список ребер → подсчитать количество различных вершин во всех ребрах. Чтобы подсчитать количество ребер графа: матрица смежности → сумма количества ненулевых записей в каждой строке; список смежности → сумма длины всех списков; список ребер → просто получить количество строк. Решения других подзадач не показаны.

Упражнение 2.4.2.1. Для `int numDisjointSets()` используйте дополнительный целочисленный счетчик `numSets`.

Первоначально, во время `UnionFind(N)`, установите `numSets = N`. Затем, во время `unionSet(i, j)`, уменьшите `numSets` на единицу, если `isSameSet(i, j)` вернет `false`. Теперь `int numDisjointSets()` может просто возвращать значение `numSets`.

Для `int sizeOfSet(int i)` мы используем другой набор `vi setSize(N)`, инициализированный единицами (каждый набор имеет только один элемент). Во время `unionSet(i, j)` обновите массив `setSize`, выполнив `setSize[find(j)] += setSize[find(i)]` (или наоборот, в зависимости от ранга), если `isSameSet(i, j)` вернет `false`. Теперь `int sizeOfSet(int i)` может просто вернуть значение `setSize[find(i)];`.

Эти два варианта были реализованы в примере кода *ch2_08_unionfind_ds.cpp/java*.

Упражнение 2.4.3.3. `RSQ(1, 7) = 167` и `RSQ(3, 8) = 139`. Нет, использование дерева отрезков – это излишне. Существует простое решение с применением динамического программирования, которое включает в себя этап предварительной обработки $O(n)$ и занимает $O(1)$ времени на один запрос `RSQ`(см. раздел 9.33).

Упражнение 2.4.4.1. $90 - LSOne(90) = (1011010)_2 - (10)_2 = (1011000)_2 = 88$ и $90 + LSOne(90) = (1011010)_2 + (10)_2 = (1011100)_2 = 92$.

Упражнение 2.4.4.2. Просто: сдвинуть все индексы на единицу. Индекс i в 1-м дереве Фенвика теперь относится к индексу $i - 1$ в актуальной задаче.

Упражнение 2.4.4.3. Просто: преобразование чисел с плавающей запятой в целые числа. Для первого задания мы можем умножить каждое число на два. Во втором случае мы можем умножить все числа на сто.

Упражнение 2.4.4.4. Совокупная частота сортируется, поэтому мы можем использовать *двоичный поиск*.

Изучите технику «двоичного поиска по ответу», рассмотренную в разделе 3.3. Результирующая временная сложность – $O(\log_2 n)$.

2.6. Примечания к главе 2

Основные структуры данных, упомянутые в разделах 2.2–2.3, можно найти практически в каждом учебнике по структуре данных и алгоритмам. Ссылки на встроенные библиотеки C++/Java доступны в интернете по адресу: www.cppreference.com и java.sun.com/javase/7/docs/api. Обратите внимание, что хотя доступ к этим веб-сайтам обычно предоставляется на олимпиадах по программированию, мы предлагаем вам попытаться освоить синтаксис наиболее распространенных операций библиотеки, чтобы минимизировать время написания кода во время олимпиад!

Единственным исключением является, возможно, битовая маска (bitmask). Эту *необычную* технику обычно не преподают на курсах по структурам данных и алгоритмам, но она очень важна для программистов, которые хотят быть конкурентоспособными, поскольку значительно ускоряет решение определенных задач. Эта структура данных упоминается в различных местах в данной книге,

например в некоторых итерационных методах перебора и оптимизированных процедурах поиска с возвратом (раздел 3.2.2 и раздел 8.2.1), «неклассических» задачах на нахождение диапазонов и последовательностей с использованием динамического программирования (раздел 3.5.2), (неклассическом) динамическом программировании с использованием битовой маски (раздел 8.3.1). Все они используют битовые маски вместо `vector<boolean>` или `bitset<size>`, поскольку это эффективно. Заинтересованным читателям предлагается изучить книгу «Hacker's Delight» («Восторг хакера») [69], в которой более подробно обсуждаются операции с битами.

Дополнительные ссылки на структуры данных, упомянутые в разделе 2.4, приводятся далее. Графы см. в [58] и в главах 22–26 из [7]. О системах непересекающихся множеств см. главу 21 из [7]. Деревья отрезков и другие геометрические структуры данных см. в [9]. Информацию о дереве Фенвика см. в [30]. Нужно особо отметить, что во всех наших реализациях структур данных, обсуждаемых в разделе 2.4, не используются указатели. Мы применяем либо массивы, либо векторы.

Имея больше опыта и прочитав предоставленный нами исходный код, вы сможете освоить больше приемов в применении этих структур данных. Пожалуйста, уделите время изучению исходного кода, предоставленного в этой книге и рзмещенного на сайте sites.google.com/site/stevenhalim/home/material.

В данной книге обсуждается еще несколько структур данных – структуры данных, специфичные для работы со строками (суффиксный бор / дерево / массив, обсуждаются в разделе 6.6). Тем не менее есть еще много других структур данных, которые мы не можем рассмотреть в этой книге. Если вы хотите добиться большего успеха в программировании, пожалуйста, изучите методы построения структур данных помимо тех, которые мы включили в эту книгу. Например, АВЛ-деревья, красно-черные деревья или даже «play»-деревья используются для определенных задач, требующих от вас внедрения и дополнения (добавления дополнительных данных) сбалансированных двоичных деревьев поиска (BST) (см. раздел 9.29). Полезно знать деревья интервалов (которые похожи на деревья сегментов) и счетверенные деревья (для разделения 2D-пространства), так как это может помочь вам решить определенные задачи олимпиады.

Обратите внимание, что многие из эффективных структур данных, обсуждаемых в этой книге, демонстрируют стратегию «разделяй и властвуй» (она обсуждается в разделе 3.3).

Таблица 2.3. Статистические данные, относящиеся к главе 2

Параметр	Первое издание	Второе издание	Третье издание
Число страниц	12	18 (+50 %)	35 (+94 %)
Письменные упражнения	5	12 (+140 %)	14 + 27* = 41 (+ 242%)
Задачи по программированию	43	124 (+188 %)	132 (+6 %)

Распределение количества упражнений по программированию по разделам этой главы показано ниже:

Таблица 2.4. Распределение количества упражнений по программированию по разделам главы 2

Раздел	Название	Число заданий	% в главе	% в книге
2.2	Линейные структуры данных	79	60 %	5 %
2.3	Нелинейные структуры данных	30	23 %	2 %
2.4	Библиотеки, реализованные авторами книги	23	17 %	1 %

Слева направо: Виктор, Хунг, д-р Хван, Фонг, Дюк, Тянь

Слева направо: Хьюберт, Чуанци, Цзы Чун, Стивен, Чжун Сюн, Раймонд, Мистер Чонг

Глава 3

Некоторые способы решения задач

Если у вас есть только молоток, любая проблема выглядит как гвоздь.

– Авраам Маслоу, 1962

3.1. Общий обзор и мотивация

В этой главе мы обсудим четыре способа решения, часто используемых для «штурма» задач на олимпиадах по программированию: полный перебор (a.k.a Brute Force, или метод грубой силы), стратегию «разделяй и властвуй», «жадный» подход и динамическое программирование. Всем конкурентоспособным программистам, включая участников IOI и ICPC, необходимо освоить эти подходы (не ограничиваясь только ими), чтобы иметь возможность решить поставленную задачу с помощью соответствующего «инструмента». «Удар» по каждой задаче с помощью «молотка» – полного перебора не позволит выступать на соревнованиях достаточно успешно.

Чтобы проиллюстрировать это, ниже мы обсудим четыре простые задачи, включающие массив A, содержащий $n \leq 10K$ малых целых чисел (т. е. чисел со значениями $\leq 100K$), например: A = {10, 7, 3, 5, 8, 2, 9}, $n = 7$, чтобы показать, что произойдет, если мы попытаемся решить все задачи, используя только полный перебор.

1. Найдите самый большой и самый маленький элемент A (*10 и 2 для данного примера*).
2. Найдите k-й наименьший элемент в A (*если $k = 2$, ответ равен 3 для данного примера*).
3. Найдите наибольшее число g такое, что $g = |x - y|$ для некоторых x, y в A (*8 для данного примера*).
4. Найдите самую длинную возрастающую подпоследовательность в A (*{3, 5, 8, 9} для данного примера*).

Ответ на первое задание прост: возьмите каждый элемент A и проверьте, является ли он текущим наибольшим (или наименьшим) элементом, найденным до сих пор. Это решение с помощью полного перебора имеет временную сложность $O(n)$.

Второе задание немного сложнее. Мы можем использовать приведенное выше решение для нахождения наименьшего значения и заменить его большим значением (например, 1*M*), чтобы «удалить» его. Затем мы можем снова найти наименьшее значение (второе наименьшее значение в исходном массиве) и заменить его на 1*M*. Повторяя этот процесс *k* раз, мы найдем *k*-е наименьшее значение. Такой метод работает корректно, но если $k = n/2$ (медиана), это время работы метода есть $O(n/2 \times n) = O(n^2)$. Вместо этого мы можем отсортировать массив *A* на $O(n \log n)$, возвращая ответ просто как `A[k-1]`. Однако лучшим решением для небольшого числа запросов является решение с ожидаемой временной сложностью $O(n)$, приведенное в разделе 9.29. Вышеупомянутые решения с временной сложностью $O(n \log n)$ и $O(n)$ являются решениями, использующими стратегию «разделяй и властвуй».

Для третьей задачи мы можем аналогичным образом рассмотреть все возможные пары целых чисел *x* и *y* в *A*, проверяя, является ли интервал между ними наибольшим для каждой пары. Этот подход полного перебора имеет временную сложность $O(n^2)$. Он работает, но медленно и неэффективно. Мы можем доказать, что *g* может быть получено путем нахождения разницы между наименьшим и наибольшим элементами в *A*. Эти два целых числа могут быть найдены с помощью решения первой задачи за время $O(n)$. Никакая другая комбинация двух целых чисел в *A* не может создать больший разрыв. Это решение, использующее «жадные» алгоритмы.

Для четвертой задачи попытка перебора всех $O(2^n)$ возможных подпоследовательностей с целью найти самую длинную возрастающую подпоследовательность неосуществима для всех $n \leqslant 10K$. В разделе 3.5.2 мы обсудим простое решение с использованием динамического программирования и временной сложностью $O(n^2)$, а также более быстрое «жадное» решение, сложность которого $O(n \log k)$.

Несколько советов относительно этой главы: не просто запомните решения для каждой обсуждаемой задачи, но вместо этого запомните и усвойте используемые методы мышления и решения задач. Обладать хорошими навыками решения задач гораздо важнее, нежели просто запоминать решения известных задач по программированию, когда имеешь дело с олимпиадными задачами, среди которых часто встречаются новые и требующие творческого подхода задачи.

3.2. Полный перебор

Техника *полного перебора*, также известная как метод грубой силы, или рекурсивного поиска, – метод решения задачи путем обхода всего (или части) пространства поиска для получения требуемого решения. Во время перебора нам разрешается «обрезать» (то есть не исследовать) части пространства поиска, если мы определили, что эти части точно не содержат требуемого решения.

На олимпиадах по программированию участник *должен* выбрать решение методом полного перебора, лишь когда отсутствуют другие возможные алгоритмы (например, задача перечисления *всех* перестановок {0, 1, 2, ..., $N - 1$} имеет временную сложность $O(N!)$), или когда существуют лучшие алгоритмы,

но их использование не является насущной необходимостью, так как размер набора входных данных оказывается небольшим (например, задача, использующая запрос минимального (максимального) значения из диапазона, как в разделе 2.4.3, но для статических массивов с $N \leqslant 100$, решаема с помощью цикла (для каждого из запросов) с временной сложностью $O(N)$).

Для ICPC полный перебор должен быть первым рассматриваемым решением, поскольку обычно такое решение легко придумать, запрограммировать и/или отлаживать. Помните принцип «КП»: делайте его коротким (К) и простым (П). Не содержащее ошибок решение задачи методом полного перебора никогда не должно получить вердикт жюри «Неправильный ответ» (WA) на олимпиадах по программированию, поскольку оно охватывает все пространство поиска.

Однако для многих задач программирования имеются гораздо лучшие варианты решения, чем полный перебор, как показано в разделе 3.1. Таким образом, решение, использующее полный перебор, может получить вердикт жюри о превышении лимита времени (TLE). Правильно проведя анализ, вы можете оценить вероятный результат (TLE или AC), прежде чем приступите к написанию кода (используйте табл. 1.4 в разделе 1.2.3 для проведения анализа). Если согласно вашим оценкам полный перебор уложится в ограничения по времени, то используйте его. Это даст вам больше времени для работы над более сложными задачами, в которых полный перебор окажется слишком медленным.

На IOI вам, как правило, потребуются более совершенные методы решения задач, поскольку решения, использующие полный перебор, обычно получают слишком низкие оценки, что негативно сказывается на сумме общего балла в схемах подсчета баллов за решение итоговых подзадач. Тем не менее полный перебор следует использовать, когда вы не можете найти лучшее решение – он, по крайней мере, позволит вам набрать хотя бы несколько баллов.

Иногда запуск полного перебора на небольших примерах для сложной задачи может помочь понять ее структуру через определенные закономерности в структуре выходных данных (можно визуализировать эту структуру для некоторых задач), что в дальнейшем можно использовать для разработки более быстрого алгоритма. Некоторые задачи из области комбинаторики (см. раздел 5.4) могут быть решены таким способом. Также полный перебор может выступить в качестве способа проверки на правильность решения для небольших примеров, обеспечивая дополнительную проверку для более быстрого, но нетривиального алгоритма, который вы разрабатываете.

После прочтения данного раздела у вас может сложиться впечатление, что полный перебор работает только для задач, относящихся к категории «легких», и обычно не является предполагаемым решением для задач из категории «более сложных». Это не совсем так. Существуют сложные задачи, которые можно решить только благодаря творческому подходу к использованию полного перебора. Мы поместили эти задачи в раздел 8.2.

В следующих двух разделах мы приведем несколько (сравнительно простых) примеров этого простого, но, возможно, требующего усилий подхода к решению задач. В разделе 3.2.1 мы приводим примеры, которые реализуются *итеративно*.

В разделе 3.2.2 мы приводим примеры решений, которые реализуются *рекурсивно* (возвратная рекурсия).

Наконец, в разделе 3.2.3 мы даем несколько советов, которые позволят вашему решению, особенно решению, использующему полный перебор, уложиться в ограничения по времени.

3.2.1. Итеративный полный перебор

Итеративный полный перебор (два вложенных цикла: UVa 725 – Division)

Сокращенная формулировка условия задачи: найти и вывести все пары пятизначных чисел, которые в совокупности используют все цифры от 0 до 9 (цифры 0–9 в записи пары чисел не повторяются), причем первое число, разделенное на второе, равно целому числу N, где $2 \leqslant N \leqslant 79$. То есть `abcde / fghij = N`, где каждая буква представляет отдельную цифру. Первая цифра одного из чисел в паре может быть нулевой, например для `N = 62` имеем `79546 / 01283 = 62`; `94736 / 01528 = 62`.

Быстрый анализ показывает, что `fghij` может варьироваться только от `01234` до `98765`, что составляет максимум $\approx 100K$ вариантов. Еще более точной оценкой для `fghij` является диапазон от `01234` до `98765 / N`, который имеет максимум $\approx 50K$ возможностей для $N = 2$ и становится меньше с увеличением N. Для каждого значения `fghij` мы можем получить `abcde` из `fghij * N`, а затем проверить, что все 10 цифр в записи чисел различны. Это дважды вложенный цикл с временной сложностью не более $\approx 50K \times 10 = 500K$ операций на тестовый пример (это не много). Таким образом, возможно использовать итеративный полный перебор. Основная часть кода показана ниже (мы используем хитрый прием в операциях с битами, показанный в разделе 2.2, для определения уникальности цифр):

```
for (int fghij = 1234; fghij <= 98765 / N; fghij++) {
  int abcde = fghij * N;           // таким образом, abcde и fghij содержат не более 5 цифр
  int tmp, used = (fghij < 10000);              // если цифра f=0, то мы должны ее пометить
  tmp = abcde; while (tmp) { used |= 1 << (tmp % 10); tmp /= 10; }
  tmp = fghij; while (tmp) { used |= 1 << (tmp % 10); tmp /= 10; }
  if (used == (1<<10) - 1)                  // если все цифры используются, вывести вариант
    printf("%0.5d / %0.5d = %d\n", abcde, fghij, N);
}
```

Итеративный полный перебор (множество вложенных циклов: UVa 441 – Lotto)

В соревнованиях по программированию задачи, которые можно решить с помощью одного цикла, обычно считаются простыми. Задачи, требующие итераций с двойным вложением, такие как UVa 725 – Division выше, являются более сложными, но они не обязательно считаются трудными. Конкурентоспособные программисты должны легко писать код с более чем двумя вложенными циклами.

Давайте взглянем на задачу UVa 441, которую можно кратко сформулировать следующим образом: дано $6 < k < 13$ целых чисел, требуется перечислить все возможные подмножества размера 6 этих целых чисел в отсортированном порядке.

Поскольку размер требуемого подмножества всегда равен 6, а выходные данные должны быть отсортированы лексикографически (входные данные уже отсортированы), самое простое решение – использовать шесть вложенных циклов, как показано ниже. Обратите внимание, что даже в самом большом тестовом примере, когда $k = 12$, эти шесть вложенных циклов будут давать только ${}_{12}C^6 = 924$ строки вывода. Это немного.

```
for (int i = 0; i < k; i++)                // входные данные: k отсортированных целых чисел
  scanf("%d", &S[i]);
for (int a = 0 ; a < k - 5; a++)                                    // шесть вложенных циклов!
  for (int b = a + 1; b < k - 4; b++)
    for (int c = b + 1; c < k - 3; c++)
      for (int d = c + 1; d < k - 2; d++)
        for (int e = d + 1; e < k - 1; e++)
          for (int f = e + 1; f < k    ; f++)
            printf("%d %d %d %d %d %d\n",S[a],S[b],S[c],S[d],S[e],S[f]);
```

Итеративный полный перебор (циклы + сокращение: UVa 11565 – Simple Equations)

Сокращенная формулировка условия задачи: даны три целых числа A, B и C ($1 \leqslant A, B, C \leqslant 10\,000$). Найдите три других различных целых числа x, y и z таких, что $x + y + z = A$, $x \times y \times z = B$ и $x^2 + y^2 + z^2 = C$.

Третье уравнение $x^2 + y^2 + z^2 = C$ является хорошей отправной точкой. Предполагая, что C имеет наибольшее значение 10 000, а y и z равны единице и двум (x, y, z должны быть различны), тогда возможный диапазон значений для x равен [–100, ..., 100]. Мы можем использовать те же рассуждения, чтобы получить аналогичный диапазон для y и z. Затем мы можем написать следующее итеративное решение с глубиной вложенности 3, для которого требуется $201 \times 201 \times 201 \approx 8M$ операций на тестовый пример.

```
bool sol = false; int x, y, z;
for (x = -100; x <= 100; x++)
  for (y = -100; y <= 100; y++)
    for (z = -100; z <= 100; z++)
      if (y != x && z != x && z != y &&                        // все три должны быть разными
          x + y + z == A && x * y * z == B && x * x + y * y + z * z == C) {
        if (!sol) printf("%d %d %d\n", x, y, z);
        sol = true; }
```

Обратите внимание на то, как сокращенное вычисление AND использовалось для ускорения решения путем принудительной проверки того, различны ли x, y и z перед проверкой трех формул.

Код, показанный выше, уже укладывается в ограничения по времени для решения этой задачи, но мы можем добиться большего. Мы также можем использовать второе уравнение $x \times y \times z = B$ и предположить, что $|x| <= |y| <= |z|$, откуда вывести $|x| * |x| * |x| <= B$, или $|x| <= \sqrt[3]{B}$. Новый диапазон x равен [–22, ..., 22]. Мы также можем сократить пространство поиска, используя операторы `if` для выполнения только некоторых (внутренних) циклов или используя операторы `break` и/или `continue` для останова/пропуска циклов. Приведенный ниже код работает намного быстрее, чем код, показанный выше (есть несколько других

оптимизаций, необходимых для решения усложненной версии этой задачи: см. UVa 11571 – Simple Equations – Extreme!!):

```
bool sol = false; int x, y, z;
for (x = -22; x <= 22 && !sol; x++) if (x * x <= C)
  for (y = -100; y <= 100 && !sol; y++) if (y != x && x * x + y * y <= C)
    for (z = -100; z <= 100 && !sol; z++)
      if (z != x && z != y &&
          x + y + z == A && x * y * z == B && x * x + y * y + z * z == C) {
        printf("%d %d %d\n", x, y, z);
        sol = true; }
```

Итеративный полный перебор (перестановки: UVa 11742 – Social Constraints)

Сокращенная формулировка условий задачи: есть $0 < n \leqslant 8$ зрителей фильма. Они будут сидеть в первом ряду на n последовательных свободных местах. Среди них есть $0 \leqslant m \leqslant 20$ ограничений вида «зритель фильма a и зритель фильма b должны находиться на расстоянии не более (или не менее) c мест». Вопрос прост: сколько существует возможных вариантов рассадки?

Ключевой частью для решения этой задачи является понимание того, что нам необходимо рассмотреть **все** перестановки (варианты рассадки). Как только мы осознаем этот факт, мы можем получить это простое решение, использующее «фильтрацию», с временной сложностью $O(m \times n!)$. Мы устанавливаем `counter = 0`, а затем пробуем все возможные $n!$ перестановок. Мы увеличиваем значение `counter` на 1, если текущая перестановка удовлетворяет всем m ограничениям. После того как будут рассмотрены все $n!$ перестановок, мы выводим окончательное значение счетчика. Поскольку максимальное значение n равно 8, а максимальное значение m равно 20, для самого большого контрольного примера все равно потребуется только $20 \times 8! = 806\,400$ операций – вполне жизнеспособное решение.

Если вы никогда не писали алгоритм для генерации всех перестановок набора чисел (см. **упражнение 1.2.3**, задача 7), вы можете не знать, как действовать дальше. Простое решение на C++ показано ниже.

```
#include <algorithm>     // next_permutation - внутренняя функция этой библиотеки в C++ STL
// процедура main
int i, n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7};                        // первая перестановка
do {                           // Попробуйте все возможные O(n!) перестановок, самое большое
                                                                  // входное значение 8! = 40320
  ...                               // проверьте указанное ограничение на основе 'p' за O(m)
}                                     // т. о., общая временная сложность составит O(m * n!)
while (next_permutation(p, p + n));          // это внутренняя функция <algorithm> в C++ STL
```

Итеративный полный перебор (подмножества: UVa 12455 – Bars)

Упрощенная формулировка условий задачи[1]: пусть дан список `l`, содержащий $1 \leqslant n \leqslant 20$ целых чисел, существует ли подмножество списка `l`, которое в сумме дает заданное целое число X?

---

[1] Эта задача также называется задачей о сумме элементов подмножества, см. раздел 3.5.3.

Мы можем попробовать все 2^n возможных подмножеств целых чисел, суммировать выбранные целые числа для каждого подмножества с временной сложностью $O(n)$ и посмотреть, равна ли сумма этих выбранных целых чисел X. Таким образом, общая временная сложность составляет $O(n \times 2^n)$. Для самого большого тестового примера, когда $n = 20$, это всего лишь $20 \times 2^{20} \approx 21M$. Это «много», но задачу все еще возможно решить указанным методом (каким образом это сделать, описано ниже).

Если вы никогда не использовали алгоритм для генерации всех подмножеств набора чисел (см. **упражнение 1.2.3**, задача 8), то вам может быть непонятно, как действовать дальше. Простое решение – использовать *двоичное представление* целых чисел от 0 до $2^n - 1$ для описания всех возможных подмножеств. Если вы не знакомы с операциями с битами, см. раздел 2.2. Решение может быть написано на простом C/C++, показанном ниже (также оно работает на Java). Поскольку операции с битами (очень) быстрые, $21M$ операций для самого большого контрольного примера все еще смогут выполняться менее чем за секунду. *Примечание:* возможна реализация, которая будет работать быстрее (см. раздел 8.2.1).

```
// процедура main, переменная 'I' (битовая маска) определена ранее
for (i = 0; i < (1 << n); i++) {                        // для каждого подмножества, O(2^n)
  sum = 0;
  for (int j = 0; j < n; j++)                                  // проверьте вхождение, O(n)
    if (i & (1 << j))                 // проверьте, включен ли бит 'j' в подмножестве 'I'?
      sum += l[j];                                              // если да, обработайте 'j'
  if (sum == X) break;                                    // ответ найден: битовая маска 'i'
}
```

Упражнение 3.2.1.1. Ответьте на вопрос: почему для решения задачи UVa – 725 лучше перебирать `fghij`, а не `abcde`?

Упражнение 3.2.1.2. Работает ли алгоритм 10!, который находит перестановки `abcdefghij`, для задачи UVa – 725?

Упражнение 3.2.1.3*. Java пока не имеет встроенной функции `next_permutation`. Если вы пользователь Java, напишите свою процедуру возвратной рекурсии для генерации всех перестановок.

Это похоже на возвратную рекурсию для задачи о восьми ферзях.

Упражнение 3.2.1.4*. Как бы вы решили задачу UVa – 12455, если $1 \leqslant n \leqslant 30$ и каждое целое число может быть равно 1 000 000 000? Подсказка: см. раздел 8.2.4.

3.2.2. Рекурсивный полный перебор (возвратная рекурсия)

Простая возвратная рекурсия: UVa 750 – 8 Queens Chess Problem

Сокращенная формулировка условий задачи: в шахматах (с доской 8×8) можно разместить на доске восемь ферзей так, чтобы никакие два ферзя не находились под ударом друг друга. Определите *все* возможные варианты такого раз-

мещения, если задано положение одного из ферзей (то есть клетка с координатами (a, b) должна содержать ферзя). Выведите все возможные положения в лексикографическом (отсортированном) порядке.

Самое наивное решение состоит в том, чтобы перечислить все комбинации, содержащие восемь различных клеток из $8 \times 8 = 64$ имеющихся клеток на шахматной доске, и посмотреть, можно ли разместить восемь ферзей на этих позициях без конфликтов. Однако число всех вариантов размещения $C_8^{64} \approx 4\,000\,000\,000$ – эту идею даже не стоит пытаться использовать.

Лучшее, но все же наивное решение – понять, что каждый ферзь может занимать только одну вертикаль, поэтому мы можем поместить ровно одного ферзя на каждую вертикаль. Это решение дает всего $8^8 \approx 17$ млн возможных вариантов, что однозначно лучше по сравнению с 4 000 000 000. Однако это все еще «пограничное» решение задачи. Если мы реализуем полный поиск подобным образом, мы, скорее всего, получим вердикт жюри о превышении лимита времени (TLE), особенно если для этой задачи есть несколько тестовых примеров для контрольных проверок. Мы можем применить несколько более простых оптимизаций, описанных ниже, чтобы еще больше сократить пространство поиска.

Мы знаем, что никакие два ферзя не могут занимать одну и ту же вертикаль или одну и ту же горизонталь. Используя это, мы можем еще больше упростить исходную задачу до задачи поиска допустимых перестановок из 8! позиций в рядах.

Значение `row[i]` описывает положение горизонтали, на которой находится ферзь, по вертикали `i`. Пример: `row = {1, 3, 5, 7, 2, 0, 6, 4}`, как на рис. 3.1, является одним из решений этой задачи; `row[0] = 1` означает, что ферзь на вертикали 0 помещается на горизонталь 1 и т. д. (индекс в этом примере начинается с 0). В этой модели пространство поиска уменьшается с $8^8 \approx 17M$ до $8! \approx 40K$. Это решение уже будет работать достаточно быстро, но мы все еще можем улучшить его.

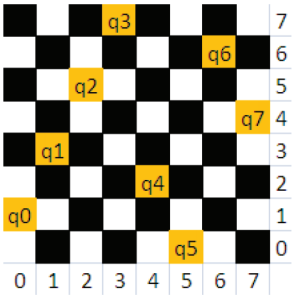


Рис. 3.1 ❖ Задача о восьми ферзях

Мы также знаем, что никакие два ферзя не могут находиться на одной из двух линий диагонали. Пусть позиция ферзя A на доске будет `(i, j)`, а ферзя B – `(k, l)`. Они атакуют друг друга, если `abs(i-k) == abs(j-l)`. Эта формула означает, что расстояния по вертикали и по горизонтали между этими двумя ферзями равны, то есть ферзи A и B взаимно размещены на одной из двух диагональных линий.

Решение, использующее возвратную рекурсию, помещает ферзей по очереди на вертикали с 0 по 7, соблюдая все ограничения, указанные выше. Наконец, когда решение, подходящее по этому критерию, найдено, проверьте, удовлетворяет ли хотя бы один из ферзей входным ограничениям, т. е. `row[b] == a`. Это решение, верхняя граница временной сложности которого составляет $O(n!)$, получит вердикт AC.

Мы предоставляем нашу реализацию ниже. Если вы никогда ранее не писали решение для возвратной рекурсии, пожалуйста, внимательно изучите его и, по возможности, напишите свое собственное решение.

```
#include <cstdlib>                                    // мы используем версию int 'abs'
#include <cstdio>
#include <cstring>
using namespace std;

int row[8], TC, a, b, lineCounter;              // использование глобальных переменных – OK

bool place(int r, int c) {
  for (int prev = 0; prev < c; prev++)          // проверка уже размещенных на доске ферзей
    if (row[prev] == r || (abs(row[prev] - r) == abs(prev - c)))
      return false;    // ферзи находятся на одной горизонтали или диагонали -> недопустимо
  return true; }
void backtrack(int c) {
  if (c == 8 && row[b] == a) {          // возможное решение: в sol, (a, b) имеется 1 ферзь
    printf("%2d      %d", ++lineCounter, row[0] + 1);
    for (int j = 1; j < 8; j++) printf(" %d", row[j] + 1);
    printf("\n"); }
  for (int r = 0; r < 8; r++)                       // переберите все возможные горизонтали
    if (place(r, c)) {                    // проверка, можно ли разместить ферзя в клетке,
                                        // имеющей эти координаты по горизонтали и вертикали
      row[c] = r; backtrack(c + 1);          // разместите ферзя здесь и выполните рекурсию
}   }

int main() {
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &a, &b); a--; b--;               // переход к индексам, начинающимся с 0
    memset(row, 0, sizeof row); lineCounter = 0;
    printf("SOLN       COLUMN\n");
    printf(" #      1 2 3 4 5 6 7 8\n\n");
    backtrack(0);                                   // генерируем все 8! кандидатов решения
                                                   // (решений в реальности меньше, чем 8!)
    if (TC) printf("\n");
} } // return 0;
```

Файл исходного кода: *ch3_01_UVa750.cpp/java*

Более сложный вариант возвратной рекурсии: UVa 11195 – Another n-Queen Problem

Сокращенная формулировка условий задачи: пусть дана шахматная доска размерностью $n \times n$ ($3 < n < 15$), где некоторые клетки считаются «непригодными»

(ферзи не могут быть помещены на эти «непригодные» клетки). Сколько существует способов разместить n ферзей на шахматной доске, чтобы два ферзя не находились под ударом друг друга? Примечание: «непригодные» клетки нельзя использовать для блокировки атаки ферзей.

Код, использующий возвратную рекурсию, приведенный выше, недостаточно быстр для $n = 14$ и не учитывает при размещении «непригодные» для размещения клетки – это сразу делает его неприменимым для решения задачи с такими условиями. Представленное ранее решение, верхняя граница временной сложности которого $O(n!)$, все еще уложится во временные ограничения при $n = 8$, но при $n = 14$ оно будет работать слишком долго. Мы должны улучшить его.

Основная проблема, связанная с приведенным ранее кодом решения задачи о размещении n ферзей, заключается в том, что он работает довольно медленно при проверке правильности позиции нового ферзя, поскольку мы сравниваем позицию нового ферзя с позициями `c-1` предыдущих ферзей (см. функцию `bool place(int r, int c)`). Лучше хранить ту же информацию, используя три массива логических значений (сейчас мы используем массивы логических значений `bitset`):

```
bitset<30> rw, ld, rd;          // для наибольшего значения n = 14, имеется 27 диагоналей
```

Первоначально все n горизонталей шахматной доски (`rw`), $2 \times n - 1$ левых диагоналей (`ld`) и $2 \times n - 1$ правых диагоналей (`rd`) не используются (все эти три массива логических значений установлены в `false`). Когда ферзь помещается на клетку (`r, c`), мы устанавливаем флаг `rw[r] = true`, чтобы запретить использование этой строки снова. Кроме того, все (`a, b`), где `abs(r - a) = abs(c - b)`, также не могут больше использоваться. После удаления функции `abs` есть две возможности: `r - c = a - b` и `r + c = a + b`. Обратите внимание, что `r + c` и `r - c` представляют собой индексы для двух диагональных линий. Поскольку `r - c` может быть отрицательным, мы добавляем смещение `n - 1` к обеим сторонам уравнения, так что `r - c + n - 1 = a - b + n - 1`. Если ферзь помещается в ячейку (`r, c`), мы устанавливаем флаг `ld[r - c + n - 1] = true` и `rd[r + c] = true`, чтобы запретить повторное использование этих двух диагоналей. С этими дополнительными структурами данных и дополнительным специфичным для задачи ограничением в UVa 11195 (`board[r][c]` не может быть «непригодной» клеткой) мы можем дополнить наш код следующим образом:

```
void backtrack(int c) {
if (c == n) { ans++; return; }                                          // решение
for (int r = 0; r < n; r++)                        // попробуйте все возможные горизонтали
if (board[r][c] != '*' && !rw[r] && !ld[r - c + n - 1] && !rd[r + c]) {
rw[r] = ld[r - c + n - 1] = rd[r + c] = true;                          // выключите флаг
backtrack(c + 1);
rw[r] = ld[r - c + n - 1] = rd[r + c] = false;                          // восстановите
} }
```

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/recursion.html

Упражнение 3.2.2.1. Код, приведенный для решения задачи UVa – 750, может быть дополнительно оптимизирован путем сокращения поиска, когда `'row[b] != a'` *ранее* во время рекурсии (не только когда `c == 8`). Измените приведенную реализацию, представив свой вариант.

Упражнение 3.2.2.2*. К сожалению, обновленное решение с использованием массивов логических значений: `rw`, `ld` и `rd` – все равно не уложится во временные ограничения и получит вердикт TLE для задачи UVa 11195 – Another n-Queen Problem. Нам необходимо еще больше ускорить решение, используя битовую маску и другой способ учесть ограничения, связанные с левой и правой диагоналями доски. Это решение будет обсуждаться в разделе 8.2.1. А пока воспользуйтесь приведенной здесь (не получившей оценку «Принято», Accepted) идеей, чтобы модифицировать и ускорить код решения задачи UVa – 750 для решения задачи UVa – 11195 и еще для двух похожих задач: UVa – 167 и UVa – 11085.

3.2.3. Советы

Наиболее азартная часть соревнования при написании кода для решения задачи полным перебором – это попытка создания такого кода, который сможет преодолеть ограничение по времени. Если ограничение по времени составляет 10 секунд (в тестирующих системах обычно не устанавливают большие значения времени при задании ограничений по времени для эффективного судейства) и ваша программа в настоящее время выполняется за ≈ 10 секунд на нескольких (может быть более одного) тестовых примерах с максимальным размером входного файла, как указано в описании задачи, однако ваш код по-прежнему получает оценку TLE, вы можете слегка оптимизировать «критическую часть»[1] в своей программе, вместо того чтобы начать решать задачу заново и использовать более быстрый алгоритм, который может оказаться трудно реализовать.

Вот несколько советов, которые вы, возможно, захотите учесть при разработке решения, использующего полный перебор для определенной задачи, чтобы повысить его шансы уложиться во временные ограничения. Написание хорошего решения, использующего полный перебор, – это само по себе искусство.

Совет 1. Фильтрация или генерация?

Программы, которые проверяют много подходящих вариантов (если не все такие варианты) и выбирают правильные (или удаляют неправильные), называются «фильтрами», например: наивное решение задачи о восьми ферзях, которое исследует ${}_{64}C_8$ возможных комбинаций с временной сложностью 8^8, итеративное решение задач UVa – 725 и UVa – 11742 и т. д. Обычно программы-«фильтры» пишутся итеративно.

Программы, которые постепенно создают решения и немедленно отбрасывают неподходящие частичные решения, называются «генераторами»:

[1] Говорят, что каждая программа тратит большую часть времени на исполнение только 10 % своего кода – критической части.

например, улучшенное рекурсивное решение задачи о восьми ферзях с его сложностью, не превосходящей $O(n!)$ плюс проверки диагоналей. Обычно программы-генераторы легче реализовать рекурсивным способом, поскольку это дает нам большую гибкость для сокращения пространства поиска.

Как правило, фильтры легче кодировать, но работают они гораздо медленнее, учитывая, что обычно гораздо сложнее итеративно обрезать большую часть пространства поиска. Используйте математику (анализ сложности), чтобы увидеть, достаточно ли хорош фильтр; и если он окажется недостаточно быстрым, вам нужно создать генератор.

Совет 2. Сокращайте пространство поиска как можно раньше, отбросив невыполнимое/ненужное

При генерации решений с использованием возвратной рекурсии (см. совет 1 выше) мы можем обнаружить частичное решение, которое никогда не приведет к полному решению. В этом случае мы можем сократить поиск и исследовать другие части пространства поиска. Пример: проверка диагонали в решении задачи о восьми ферзях выше. Предположим, что мы разместили ферзя на горизонтали `row[0] = 2`. Размещение следующего ферзя на горизонтали `row[1] = 1` или `row[1] = 3` вызовет конфликт по диагонали, а размещение следующего ферзя на горизонтали `row[1] = 2` приведет к конфликту по горизонтали. Продолжение исследования любого из этих неосуществимых частичных решений никогда не приведет к полному решению. Таким образом, мы можем отбросить эти частичные решения на данном этапе и сосредоточиться на других допустимых позициях: `row[1] = {0, 4, 5, 6, 7}`, тем самым сокращая общее время выполнения кода. Как правило, чем раньше вы сможете сократить пространство поиска, тем лучше.

В других задачах мы можем вычислить «потенциальную ценность» частичного (и все еще действительного) решения. Если потенциальная ценность пока уступает ценности наилучшего найденного решения, мы можем обрубить поиск в этом месте.

Совет 3. Используйте симметрию

В некоторых задачах встречается симметрия, и мы должны попытаться ее использовать, чтобы сократить время выполнения. В задаче о восьми ферзях имеется 92 подходящих решения, но только 12 из них уникальны (или фундаментальны/канонически), поскольку в задаче имеются осевая симметрия и симметрия относительно вращения. Вы можете использовать данный факт, генерируя лишь 12 уникальных решений, и при необходимости генерировать все 92 решения, вращая и отражая эти 12 уникальных решений. Пример: `row = {7-1, 7-3, 7-5, 7-7, 7-2, 7-0, 7-6, 7-4} = {6, 4, 2, 0, 5, 7, 1, 3}` – горизонтальное отражение конфигурации на рис. 3.1.

Однако мы должны отметить, что иногда рассмотрение симметрии может фактически усложнить код. В олимпиадном программировании это обычно не лучший способ (мы хотим, чтобы более короткий код сводил к минимуму ошибки). Если выигрыш, полученный при учете симметрии в решении задачи, незначителен, просто проигнорируйте этот совет.

Совет 4. Предварительный подсчет a.k.a. предварительные вычисления

Иногда полезно генерировать таблицы или другие структуры данных, которые ускоряют поиск результата, до того как запустится выполнение самой программы. Такой подход называется предподсчетом[1], при котором можно обменять память/пространство на время. Тем не менее, как показывает опыт недавно проведенных олимпиад по программированию, этот метод редко применим для решения задач олимпиадного программирования.

Например, поскольку мы знаем, что в стандартной шахматной задаче о восьми ферзях существует только 92 решения, мы можем создать двумерный массив `int solution[92][8]` и затем заполнить его всеми 92 подходящими перестановками позиций восьми ферзей, расставленных по горизонтали. То есть мы можем записать и запустить программу-генератор (которая может работать несколько секунд или даже минут), чтобы заполнить двумерный массив `solution`. После этого мы можем написать другую программу для простой и быстрой печати правильных перестановок в 92 предварительно рассчитанных конфигурациях, которые удовлетворяют ограничениям задачи.

Совет 5. Попробуйте решить задачу «с конца»

Некоторые олимпиадные задачи выглядят намного проще, когда они решаются «с конца» [53] (с *менее очевидной* отправной точки), чем когда решаются с помощью лобовой атаки (с *более очевидной* отправной точки). Будьте готовы к нестандартным подходам к решению задач.

Этот совет лучше всего проиллюстрировать на примере решения задачи UVa 10360 – Rat Attack: представьте 2D-массив (с размерностью до 1024×1024), содержащий координаты крыс. По ячейкам – элементам массива – распределено $n \leqslant 20\,000$ крыс.

Определите, какая ячейка с координатами (`x, y`) должна быть подвергнута газовой бомбардировке, чтобы число крыс, убитых в квадрате, расположенном в интервале от (`x-d, y-d`) до (`x + d, y + d`), было максимальным. Значение d – мощность газовой бомбы ($d \leqslant 50$). См. рис. 3.2.

Решение «в лоб» состоит в том, чтобы подойти к этой задаче самым очевидным способом: «бомбить» каждую из 10 242 ячеек и выбрать наиболее эффективное местоположение. Для каждой бомбардированной ячейки (`x, y`) мы можем выполнить последовательный просмотр данных за $O(d^2)$, чтобы подсчитать количество убитых крыс в радиусе бомбардировки в заданном квадрате. В самом худшем с точки зрения производительности случае, когда массив имеет размер 10 242 и $d = 50$, для этого требуется $1024^2 \times 502 = 2621M$ операций. Вердикт – TLE[2]!

Другой вариант – подойти к этой задаче «с конца»: создать массив `int killed[1024][1024]`. Для каждой популяции крыс в клетках с координатами (`x, y`)

---

[1] В русскоязычном сообществе употребляют термин «предподсчет» и жаргонизм «прекальк». – *Прим. ред.*

[2] Хотя ЦП на 2013 г. может выполнять ≈ 100 млн операций за несколько секунд, 2621 млн операций все равно займет слишком много времени в условиях олимпиады по программированию.

добавьте ее в `killed[i][j]`, где $|i - x| \leqslant d$ и $|j - y| \leqslant d$. Это потому, что если бомба была установлена в (`i, j`), все крысы в клетках с координатами (`x, y`) будут убиты. В процессе этой предварительной обработки выполняется $O(n \times d^2)$ операций. Затем, чтобы определить наиболее оптимальную позицию бомбардировки, мы можем просто найти координату наибольшей по величине записи в массиве `killed`, что можно сделать за 10 242 операции. Этот подход требует только $20\,000 \times 50^2 + 1024^2 = 51M$ операций; для худшего тестового примера (n = 20 000, d = 50) это приблизительно в 51 раз быстрее, чем «лобовая атака»! Данное решение получит оценку «Принято» (AC).

Рис. 3.2 ❖ UVa 10360 [47]

Совет 6. Оптимизируйте свой исходный код

Есть много хитростей, которые вы можете использовать для оптимизации вашего кода. Понимание устройства аппаратного обеспечения компьютера и его организации, особенно операций ввода-вывода, операций с памятью и работы с кешем, может помочь вам разработать лучший код. Некоторые примеры (далеко не исчерпывающие) приведены ниже.

1. Пристрастное мнение: используйте C++ вместо Java. Алгоритм, реализованный на C++, обычно работает быстрее, чем алгоритм, реализованный на Java, на многих онлайн-ресурсах, предлагающих соревнования по решению задач, в том числе и в тестирующей системе UVa [47]. На некоторых соревнованиях по программированию участникам, использующим Java, завышают (чем выше ограничения, тем легче его пройти) временные ограничения по сравнению с требованиями к скорости работы алгоритма в C++ и дают дополнительное время, чтобы учесть разницу в производительности.
2. Для участников, применяющих C/C++: используйте более быстрые функции `scanf/printf` в стиле C, а не `cin/cout`. Для участников, использующих Java: используйте более быстрые классы `BufferedReader/BufferedWriter` следующим образом:

```
BufferedReader br = new BufferedReader(                          // ускорение
  new InputStreamReader(System.in));
// Примечание. После этого необходимо разделение строк и/или разбор входных данных.

PrintWriter pr = new PrintWriter(new BufferedWriter(             // ускорение
```

```
    new OutputStreamWriter(System.out)));
// PrintWriter позволяет нам использовать функцию pr.printf()
// не забудьте вызвать pr.close() перед выходом из программы Java
```

3. Используйте быструю сортировку `algorithm::sort` в C++ STL (часть «предварительной сортировки»), имеющую *приблизительную* временную сложность $O(n \log n)$, но дружественную к кешу, а не сортировку `heapsort`, имеющую истинную временную сложность $O(n \log n)$, но не дружественную к кешу (ее операции обхода, выполняемые «от корня к листу» / «от листа к корню», охватывают широкий диапазон индексов, и, как следствие, много случаев «непопадания в кеш»).
4. Получайте доступ к двумерному массиву в порядке приоритета строк (строка за строкой), а не в порядке приоритета столбцов – многомерные массивы хранятся в памяти в порядке приоритета строк.
5. Операции с битами во встроенных целочисленных типах данных (вплоть до 64-битного целого) более эффективны, чем операции с индексами в массиве логических значений (см. битовую маску в разделе 2.2).
   Если нам нужно более 64 бит, используйте `bitset` в C++ STL, а не `vector <bool>` (например, для решета Эратосфена в разделе 5.5.1).
6. Всегда используйте более низкоуровневые структуры / типы данных, если вам не нужны дополнительные функции в более высокоуровневом (или более объемном) типе данных. Например, используйте массив с немного большим размером, чем максимальный размер входного файла, вместо использования векторов с переменным размером. Кроме того, используйте 32-разрядные целочисленные значения `int` вместо 64-разрядных `long long`, поскольку операции над 32-разрядными `int` выполняются быстрее в большинстве 32-разрядных тестирующих систем.
7. Для Java используйте более быстрый `ArrayList` (и `StringBuilder`), а не `Vector` (и `StringBuffer`). Классы `Vector` и `StringBuffer` в Java – потокобезопасные классы, но потокобезопасность не нужна в олимпиадном программировании. Примечание: в этой книге мы будем придерживаться написания `Vectors`, чтобы не сбивать с толку читателей, работающих как с языком C++, так и Java, которые используют как `vector` в C++ STL, так и `Vector` в Java.
8. Объявите большинство структур данных (особенно громоздких, например большие массивы) один раз, поместив их в глобальную область видимости. Выделите достаточно памяти, чтобы справиться с самым большим объемом входных данных. В этом случае вам не нужно будет передавать структуры данных как аргументы функции. Если у вас несколько тестовых примеров, просто очистите/сбросьте содержимое структуры данных перед выполнением каждого из тестовых примеров.
9. Если у вас есть выбор писать свой код итеративно или рекурсивно, выберите итеративный подход. Пример: итеративный метод `next_permutation` в C++ STL и итеративная генерация подмножеств с использованием битовой маски, показанные в разделе 3.2.1, (намного) быстрее, чем аналогичные подобные рекурсивные подпрограммы (в основном из-за накладных расходов при вызовах функций).

10. Доступ к массиву во (вложенных) циклах может быть медленным. Если у вас есть массив A и вы часто обращаетесь к значению `A[i]` (не изменяя его) во (вложенных) циклах, может быть полезно использовать локальную переменную `temp = A[i]` и далее работать с `temp`.
11. В C/C++ *правильное* использование макросов и встроенных функций может сократить время выполнения программы.
12. Для пользователей C++: применение массивов символов в стиле C ускорит исполнение программы по сравнению с использованием `string` в C++ STL. Для пользователей Java: будьте осторожны, работая с `String`, поскольку объекты Java `String` постоянны и неизменяемы. Операции над строками с применением `String` в Java могут быть очень медленными. Вместо этого используйте Java `StringBuilder`.

Просмотрите интернет-источники или соответствующие книги (например, [69]), чтобы найти (намного) больше информации о том, как ускорить ваш код. Потренируйтесь в навыках хакерской работы с кодом, выбрав более сложную задачу из тех, что предлагаются тестирующей системой на сайте Университета Вальядолида (UVa), где время выполнения наилучшего решения не равно 0,000 с. Отправьте несколько вариантов вашего решения, проверенного и принятого тестирующей системой, и сравните различия во времени выполнения. Заимствуйте «хакерскую» модификацию кода, которая ускоряет работу вашего кода.

Совет 7. Используйте лучшие структуры данных и алгоритмы :)

Без шуток. Использование лучших структур данных и алгоритмов всегда будет давать вам преимущество перед любыми попытками оптимизации, упомянутыми в советах 1–6 выше. Если вы уверены, что написали самый быстрый код полного перебора, какой только смогли, но он все равно получил вердикт жюри TLE, откажитесь от подхода полного перебора.

Замечания о задачах, использующих полный перебор, на олимпиадах по программированию

Основным источником материала в разделе «Полный перебор» в этой главе является учебный портал USACO [48]. Мы использовали термин «полный перебор», а не «лобовой метод» или «метод грубой силы» (с его негативными коннотациями), так как считаем, что некоторые решения, использующие полный перебор, могут быть умными и быстрыми. Мы считаем, что термин «умный метод грубой силы» немного противоречив.

Если задача решается с помощью полного перебора, будет понятно, когда следует использовать итеративный полный перебор или возвратную рекурсию. Итеративные подходы используются, когда можно легко вывести различные состояния с помощью некоторой формулы относительно определенного счетчика, и (почти) все состояния должны быть проверены; например, просмотр всех индексов массива, перечисление (почти) всех возможных подмножеств небольшого множества, генерация (почти) всех перестановок и т. д. Возвратная рекурсия используется, когда трудно получить различные состояния с помощью простого индекса и/или хочется (сильно) сократить пространство поиска, например шахматная задача о восьми ферзях. Если пространство по-

иска задачи, которая решаема с помощью полного перебора, велико, то обычно используются методы возвратной рекурсии, которые позволяют заблаговременно сократить участки пространства поиска, не удовлетворяющие поставленным условиям. Сокращение пространства поиска в итеративных вариантах полного перебора – задача не невозможная, но обычно трудная.

Лучший способ улучшить свои навыки решения задач полным перебором – это решить больше задач, использующих полный перебор. Мы предоставили список таких задач, разделенных на несколько категорий, ниже. Пожалуйста, попробуйте решить как можно больше задач, особенно те, которые отмечены звездочкой как **обязательные для выполнения ***. Позже в разделе 3.5 читатели столкнутся с дополнительными примерами возвратной рекурсии, но с добавлением техники «меморизации».

Отметим, что мы обсудим некоторые более продвинутые методы поиска позже в разделе 8.2, например использование операций с битами в возвратной рекурсии, более сложный поиск в пространстве состояний, «встреча посередине» («meet-in-the-middle», англоязычный термин более популярен), поиск A *, поиск с ограничением глубины (DLS), поиск с итеративным углублением (IDS) и итеративное углубление A * (IDA *).

Задачи по программированию, решаемые с помощью полного перебора

- Итеративный подход (один цикл, последовательный просмотр данных)
  1. UVa 00102 – Ecological Bin Packing (попробуйте все шесть возможных комбинаций)
  2. UVa 00256 – Quirksome Squares (метод грубой силы, математика, можно использовать предварительные вычисления)
  3. **UVa 00927 – Integer Sequence from… *** (используйте формулу суммы арифметической прогрессии)
  4. **UVa 01237 – Expert Enough *** (LA 4142, Джакарта'08, входные данные небольшого объема)
  5. **UVa 10976 – Fractions Again ? *** (общее число решений определяется заранее, поэтому следует дважды применить метод грубой силы)
  6. UVa 11001 – Necklace (математика, метод грубой силы, максимизирующая функция)
  7. UVa 11078 – Open Credit System (один последовательный просмотр данных)
- Итеративный подход (два вложенных цикла)
  1. UVa 00105 – The Skyline Problem (карта высоты, развертка влево-вправо)
  2. UVa 00347 – Run, Run, Runaround… (моделирование процесса)
  3. UVa 00471 – Magic Numbers (чем-то похоже на UVa 725)
  4. UVa 00617 – Nonstop Travel (попробуйте все целочисленные скорости от 30 до 60 миль в час)
  5. UVa 00725 – Division (разобрано в этом разделе)
  6. **UVa 01260 – Sales *** (LA 4843, Тэджон'10, проверить все)

7. UVa 10041 – Vito's Family (попробуйте все возможные местоположения домов, где будет проживать семья Вито)
8. **UVa 10487 – Closest Sums** * (отсортируйте, а затем выполните разбиение по парам с временной сложностью $O(n^2)$)
9. *UVa 10730 – Antiarithmetic?* (два вложенных цикла с сокращением пространства поиска; возможно, могут пройти по временному ограничению наименее требовательные к объему обрабатываемых данных и скорости работы тестовые примеры; обратите внимание, что это грубое решение работает слишком медленно для больших объемов тестовых данных, генерируемых в решении UVa 11129)
10. **UVa 11242 – Tour de France** * (плюс сортировка)
11. *UVa 12488 – Start Grid* (два вложенных цикла; моделирование процесса обгона)
12. *UVa 12583 – Memory Overflow* (два вложенных цикла; будьте осторожны с переоцениванием)

- Итеративный подход (три или более вложенных циклов, сравнительно простые задачи)
  1. UVa 00154 – Recycling (три вложенных цикла)
  2. UVa 00188 – Perfect Hash (три вложенных цикла, ответ пока не найден)
  3. **UVa 00441 – Lotto** * (шесть вложенных циклов)
  4. UVa 00626 – Ecosystem (три вложенных цикла)
  5. UVa 00703 – Triple Ties (три вложенных цикла)
  6. **UVa 00735 – Dart-a-Mania** * (три вложенных цикла, затем подсчет)
  7. **UVa 10102 – The Path in the…** * (можно написать четыре вложенных цикла, нам не нужен поиск в ширину; мы получим максимальное из минимальных расстояний от Манхэттена от «1» до ближайшего «3»)
  8. UVa 10502 – Counting Rectangles (шесть вложенных циклов, прямоугольник; не слишком сложная задача)
  9. UVa 10662 – The Wedding (три вложенных цикла)
  10. UVa 10908 – Largest Square (четыре вложенных цикла, квадрат, не слишком сложная задача)
  11. UVa 11059 – Maximum Product (три вложенных цикла, объем входных данных небольшой)
  12. *UVa 11975 – Tele-loto* (три вложенных цикла, смоделируйте игру, как указано в условиях задачи)
  13. *UVa 12498 – Ant's Shopping Mall* (три вложенных цикла)
  14. *UVa 12515 – Movie Police* (три вложенных цикла)

- Итеративный подход (три или более вложенных циклов, более сложные задачи)
  1. UVa 00253 – Cube painting (попробуйте все варианты, похожая задача в UVa 11959)
  2. UVa 00296 – Safebreaker (попробуйте все 10 000 возможных кодов, четыре вложенных цикла; используйте решение, аналогичное игре «Master-Mind»)

3. UVa 00386 – Perfect Cubes (четыре вложенных цикла с ограничением числа переборов, отбрасывая неподходящие варианты)
4. UVa 10125 – Sumsets (сортировка; четыре вложенных цикла плюс двоичный поиск)
5. UVa 10177 – (2/3/4) -D Sqr / Rects /… (две/три/четыре вложенные петли, предподсчет)
6. UVa 10360 – Rat Attack (задача также решается с использованием методов динамического программирования; максимальная сумма 1024^2)
7. UVa 10365 – Blocks (используйте три вложенных цикла с ограничением числа переборов, отбрасывая неподходящие варианты)
8. *UVa 10483 – The Sum Equals…* (два вложенных цикла для *a*, *b*; получите значение *c* из *a*, *b*; существует 354 ответа для диапазона чисел [0,01 … 255,99]; аналогично UVa 11236)
9. **UVa 10660 – Citizen attention… *** (семь вложенных циклов; манхэттенское расстояние)
10. UVa 10973 – Triangle Counting (три вложенных цикла с ограничением числа переборов, отбрасывая неподходящие варианты)
11. UVa 11108 – Tautology (пять вложенных циклов, попробуйте все $2^5 = 32$ значения, ограничивая число переборов, отбрасывая неподходящие варианты)
12. ***UVa 11236 – Grocery Store* *** (три вложенных цикла для *a*, *b*, *c*; получите значение *d* из *a*, *b*, *c*; проверьте, что число строк выходных данных равно 949)
13. UVa 11342 – Three-square (предварительно рассчитайте значения квадратов чисел от 0^2 до 224^2, используйте три вложенных цикла для генерации ответов; используйте `map`, чтобы избежать дубликатов)
14. *UVa 11548 – Blackboard Bonanza* (четыре вложенных цикла, строка, ограничение числа переборов через отбрасывание неподходящих вариантов)
15. **UVa 11565 – Simple Equations *** (три вложенных цикла, ограничение числа переборов через отбрасывание неподходящих вариантов)
16. UVa 11804 – Argentina (пять вложенных петель)
17. UVa 11959 – Dice (переберите все возможные положения игральных костей, сравните с 2)

Также см.: математическое моделирование (раздел 5.2).

- Итеративный подход (нестандартные приемы)
  1. UVa 00140 – Bandwidth (макс. *N* равно 8, используйте `next_permutation`; алгоритм в `next_permutation` является итеративным)
  2. *UVa 00234 – Switching Channels* (используйте `next_permutation`; симуляция)
  3. UVa 00435 – Block Voting (только 2^{20} возможных комбинаций коалиций)
  4. UVa 00639 – Don't Get Rooked (сгенерируйте 2^{16} комбинаций и отбросьте все неподходящие варианты)
  5. ***UVa 01047 – Zones* *** (LA 3278, заключительные всемирные соревно-

вания, Шанхай'05; обратите внимание, что $n \leqslant 20$, таким образом, можно рассмотреть все возможные поднаборы вышек; затем примените принцип включения-выключения, чтобы избежать включения слишком большого числа вышек в результирующий набор)
6. *UVa 01064 – Network* (LA 3808, заключительные всемирные соревнования Токио'07; перестановка множества сообщений, содержащего до пяти сообщений; симуляция; помните слово «последовательный»)
7. UVa 11205 – The Broken Pedometer (попробуйте все 2^{15} битовых масок)
8. UVa 11412 – Dig the Holes (`next_permutation`, найдите единственный вариант из 6!)
9. **UVa 11553 – Grid Game** * (решите, перебрав все $n!$ перестановок; вы также можете использовать методы динамического программирования и битовую маску (см. раздел 8.3.1), однако это излишне)
10. UVa 11742 – Social Constraints (данная задача обсуждалась в этом разделе)
11. *UVa 12249 – Overlapping Scenes* (LA 4994, Куала-Лумпур'10; переберите все перестановки, используйте соответствие строк)
12. *UVa 12346 – Water Gate Management* (LA 5723, Пхукет'11, переберите все 2^n комбинаций, выберите лучшую)
13. *UVa 12348 – Fun Coloring* (LA 5725, Пхукет'11, попробуйте все 2^n комбинаций)
14. *UVa 12406 – Help Dexter* (переберите все 2^p возможных битовых масок, измените все «0» на «2»)
15. *UVa 12455 – Bars* (данная задача обсуждалась в этом разделе)

- Возвратная рекурсия (легкие задачи)
  1. UVa 00167 – The Sultan Successor (шахматная задача о восьми ферзях)
  2. UVa 00380 – Call Forwarding (простая возвратная рекурсия, но нам нужно работать со строками, см. раздел 6.2)
  3. UVa 00539 – The Settlers… (самый длинный простой путь в *небольшом* общем графе)
  4. **UVa 00624 – CD** * (размер набора входных данных небольшой, возвратной рекурсии достаточно)
  5. UVa 00628 – Passwords (возвратная рекурсия; следуйте правилам, содержащимся в условии задачи)
  6. UVa 00677 – All Walks of length «*n*»… (распечатать все решения с использованием возвратной рекурсии)
  7. UVa 00729 – The Hamming Distance… (сгенерировать все возможные битовые строки)
  8. UVa 00750 – 8 Queens Chess Problem (обсуждается в этом разделе с примером исходного кода)
  9. UVa 10276 – Hanoi Tower Troubles Again (вставьте числа одно за другим)
  10. UVa 10344 – 23 Out of 5 (переставить пять операндов и три оператора)

11. UVa 10452 – Marcus, help (в каждой позиции Инди может идти вперед/влево/вправо; попробовать все варианты)
12. **UVa 10576 – Y2K Accounting Bug *** (сгенерировать все, отбросить неподходящие варианты, взять максимальное значение)
13. **UVa 11085 – Back to the 8-Queens *** (см. UVa 750, предварительные вычисления)

- Возвратная рекурсия (задачи средней сложности)

1. UVa 00222 – Budget Travel (выглядит как задача, решаемая с помощью динамического программирования, однако состояние не может быть запомнено, так как «бак» является числом с плавающей запятой; к счастью, объем входных данных невелик)
2. *UVa 00301 – Transportation* (возможное число комбинаций 2^{22}, отбрасывайте неподходящие варианты)
3. UVa 00331 – Mapping the Swaps ($n \leqslant 5$…)
4. UVa 00487 – Boggle Blitz (используйте `map` для хранения сгенерированных слов)
5. **UVa 00524 – Prime Ring Problem *** (также см. раздел 5.5.1)
6. UVa 00571 – Jugs (решение может быть условно оптимальным; добавьте флаг, чтобы избежать зацикливания)
7. **UVa 00574 – Sum It Up *** (выведите все решения с помощью возвратной рекурсии)
8. UVa 00598 – Bundling Newspaper (выведите все решения с помощью возвратной рекурсии)
9. *UVa 00775 – Hamiltonian Cycle* (возвратной рекурсии достаточно, так как пространство поиска не может быть чересчур большим; в плотном графе больше вероятность того, что мы найдем гамильтонов цикл, поэтому мы можем рано отбросить неподходящие варианты; нам НЕ нужно находить лучшее из возможных решений, как в задаче коммивояжера)
10. *UVa 10001 – Garden of Eden* (оценка верхней границы числа вариантов 2^{32} выглядит пугающе, но при эффективном сокращении мы можем преодолеть ограничение по времени, поскольку контрольный пример не предполагает экстремального случая)
11. *UVa 10063 – Knuth's Permutation* (сделайте то, что требуется в задаче)
12. *UVa 10460 – Find the Permuted String* (аналогично UVa 10063)
13. UVa 10475 – Help the Leaders (создать множество и отбросить неподходящие варианты; попробовать все варианты)
14. **UVa 10503 – The dominoes solitaire *** (макс. 13 мест)
15. *UVa 10506 – Ouroboros* (любое подходящее решение получает оценку «Принято» (AC); сгенерируйте все возможные следующие цифры (до 10, в системе счисления с основанием 10 / цифры [0..9]); проверьте, являются ли они все еще последовательностью Уроборос)
16. *UVa 10950 – Bad Code* (отсортируйте входные данные; запустите возвратную рекурсию; выходные данные должны быть отсортированы; отображаются только первые 100 строк отсортированных выходных данных)

17. UVa 11201 – The Problem with the... (возвратная рекурсия, работа со строками)
18. *UVa 11961 – DNA* (существует не более 410 возможных цепочек ДНК; кроме того, мощность мутации составляет не более $K \leqslant 5$, поэтому пространство поиска намного меньше; отсортируйте выходные данные, а затем удалите дубликаты)

- Возвратная рекурсия (более сложные задачи)
  1. *UVa 00129 – Krypton Factor* (возвратная рекурсия, проверка обработки строки, форматирование вывода)
  2. UVa 00165 – Stamps (также используется динамическое программирование; могут быть выполнены предварительные вычисления)
  3. **UVa 00193 – Graph Coloring *** (максимальное независимое множество; размер входных данных небольшой)
  4. UVa 00208 – Firetruck (возвратная рекурсия с отбрасыванием неподходящих вариантов)
  5. **UVa 00416 – LED Test *** (возвратная рекурсия, рассмотрите все варианты)
  6. UVa 00433 – Bank (Not Quite O.C.R.) (решается аналогично UVA 416)
  7. UVa 00565 – Pizza Anyone? (возвратная рекурсия с отбрасыванием большого количества неподходящих вариантов)
  8. *UVa 00861 – Little Bishops* (возвратная рекурсия с отбрасыванием неподходящих вариантов, как в рекурсивном решении задачи о восьми ферзях, затем предварительные вычисления для получения результатов)
  9. UVa 00868 – Numerical maze (попробуйте строки с 1 по N; четыре способа; некоторые ограничения)
  10. ***UVa 01262 – Password **** (LA 4845 Тэджон'10, сначала отсортируйте столбцы в двух таблицах 6×5, чтобы мы могли обрабатывать общие пароли в лексикографическом порядке; возвратная рекурсия; важно: пропустите два одинаковых пароля)
  11. UVa 10094 – Place the Guards (эта задача похожа на шахматную задачу о размещении n ферзей, но вы должны найти/использовать шаблон!)
  12. *UVa 10128 – Queue* (возвратная рекурсия с отбрасыванием неподходящих вариантов; попробуйте все $N!$ (13!) перестановок, которые удовлетворяют требованию; затем используйте предварительные вычисления для получения результатов)
  13. UVa 10582 – ASCII Labyrinth (сначала упростите сложные входные данные, затем используйте возвратную рекурсию)
  14. *UVa 11090 – Going in Cycle* (задача о цикле минимального среднего веса; разрешима с помощью возвратной рекурсии с отбрасыванием неподходящих вариантов, когда текущее среднее значение за проход превышает лучшую найденную среднюю стоимость веса для цикла)

3.3. «Разделяй и властвуй»

«Разделяй и властвуй» (сокращенно D&C, Divide and Conquer) – это подход к решению задач, в котором задача упрощается путем «деления» ее на более мелкие части и последующей «победы» над каждой частью. Данный метод подразумевает следующие «шаги»:

1) разбейте исходную задачу на подзадачи – обычно на две равные или почти равные части;
2) найдите (под)решения для каждой из этих подзадач, каждая из которых теперь проще целой;
3) при необходимости объедините эти частные решения, чтобы получить полное решение для основной задачи.

Мы видели примеры использования подхода D&C в предыдущих разделах этой книги: различные алгоритмы сортировки (например, быстрая сортировка, сортировка слиянием, пирамидальная сортировка) и двоичный поиск в разделе 2.2 используют этот подход. Способ организации данных в дереве двоичного поиска, куче, дереве сегментов и дереве Фенвика в разделах 2.3, 2.4.3 и 2.4.4 также основан на подходе D&C.

3.3.1. Интересное использование двоичного поиска

В этом разделе мы обсудим подход D&C в хорошо известном алгоритме двоичного поиска.

Мы классифицируем двоичный поиск как алгоритм «разделяй и властвуй», хотя одна из работ [40] предполагает, что его следует классифицировать как «уменьшай (наполовину) и решай», поскольку он фактически не «объединяет» результаты двух подзадач. Мы выделяем этот алгоритм, потому что многие участники знакомы с ним, но немногие знают, что его можно использовать другими неочевидными способами.

Двоичный поиск: обычное использование

Напомним, что *каноническое* использование двоичного поиска – поиск элемента в статическом отсортированном массиве. Мы проверяем середину отсортированного массива, чтобы определить, содержит ли он то, что мы ищем. Если мы получаем положительный ответ на свой вопрос и больше не хотим отвечать ни на какие другие вопросы, то поиск завершен. В противном случае мы можем определить, находится ли искомое слева или справа от среднего элемента, и продолжить поиск.

Поскольку размер пространства поиска уменьшается вдвое после каждой такой проверки, сложность этого алгоритма составляет $O(\log n)$. В разделе 2.2 мы видели, что для этого алгоритма есть встроенные библиотечные процедуры, например `algorithm::lower_bound` в C++ STL (и `Collections.binarySearch` в Java).

Это не единственный способ использовать двоичный поиск. Необходимое условие для выполнения двоичного поиска – статическая отсортированная последовательность (массив или вектор) – также может быть найдена в других

необычных структурах данных, таких как пути от корня к листу дерева (не обязательно двоичного или полного), которое удовлетворяет свойству *неубывающей пирамиды* (*min heap*). Этот вариант обсуждается ниже.

Двоичный поиск по нетривиальным структурам данных

Эта оригинальная задача называется «Мой предок» и была предложена на туре национальной олимпиады ICPC в Таиланде 2009 года. Сокращенная формулировка условий задачи: дано взвешенное (семейное) дерево, содержащее до $N \leqslant 80K$ вершин с характерной особенностью: *значения вершин увеличиваются от корня к листьям*. Найдите вершину *предка*, ближайшую к корню из начальной вершины *v*, имеющей вес не менее *P*. Таких *автономных* запросов может быть до $Q \leqslant 20K$. Посмотрите на рис. 3.3 (слева). Если $P = 4$, то ответом является вершина, помеченная буквой «B» со значением 5, так как она является предком вершины *v*, ближайшей к корню «A» и имеющей значение $\geqslant 4$. Если $P = 7$, то ответ «C», со значением 7. Если $P \geqslant 9$, ответа не существует. Самый простой вариант решения – выполнить последовательный просмотр данных за $O(N)$ для каждого запроса: начиная с заданной вершины *v*, мы перемещаемся вверх по дереву (семейству), пока не достигнем первой вершины, у которой прямой родитель имеет значение $< P$, или пока не достигнем корня. Если эта вершина имеет значение $\geqslant P$ и не является самой вершиной *v*, мы нашли решение. Поскольку существует *Q* запросов, этот вариант решения выполняется за $O(QN)$ (входное дерево может быть отсортированным связанным списком, или «канатом», длиной *N*) и получит вердикт жюри «Превышение лимита времени» (TLE) при $N \leqslant 80K$ *и* $Q \leqslant 20K$.

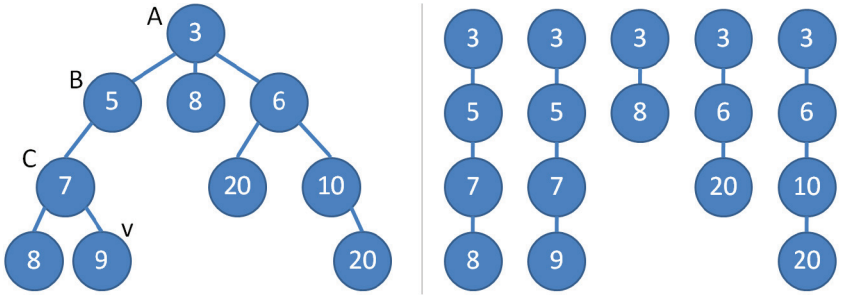


Рис. 3.3 ❖ «Мой предок» (все пять путей от корня к листу отсортированы)

Лучшее решение – хранить все $20K$ запросов (нам не нужно сразу на них отвечать). Пройдем по дереву только один раз, начиная с корня, используя линейный по сложности прямой обход дерева обхода дерева с предварительной выборкой $O(N)$ (см. раздел 4.7.2). Этот обход дерева с предварительной выборкой немного изменен, чтобы запомнить частичную последовательность от корня до текущей вершины при выполнении. Массив всегда сортируется, потому что вершины вдоль пути от корня к текущей вершине имеют увеличивающиеся веса, см. рис. 3.3 (справа). Обход дерева с предварительной выборкой для дерева, показанного на рис. 3.3 (слева), создает следующий массив, частично отсортированный от корня до текущей вершины: {{3}, {3, 5}, {3, 5, 7}, {3, 5, 7, 8},

возврат, {3, 5, 7, 9}, возврат, возврат, возврат, {3, 8}, возврат, {3, 6}, {3, 6, 20}, возврат, {3, 6, 10} и, наконец, {3, 6, 10, 20}, возврат, возврат, возврат (решено)}.

Во время обхода с предварительной выборкой, когда мы доходим до запрашиваемой вершины, мы можем выполнить **двоичный поиск** с временной сложностью $O(\log N)$ (точнее: `lower_bound`) в частичном массиве весов корней до текущей вершины, чтобы получить предка, ближайшего к корню, со значением не менее P, записывая найденные решения. Наконец, мы можем выполнить простую итерацию $O(Q)$ для вывода результатов. Общая временная сложность этого подхода составляет $O(Q \log N)$, и время выполнения можно контролировать, учитывая объем входных данных.

Метод деления пополам

Мы обсудили применение двоичного поиска для поиска элементов в статических отсортированных последовательностях. Однако принцип двоичного поиска[1] также может быть использован для поиска корня функции, которую может быть сложно непосредственно вычислить.

Пример: вы покупаете автомобиль в кредит и хотите оплатить кредит ежемесячно в размере d долларов за m месяцев. Предположим, что стоимость автомобиля изначально составляет v долларов, и банк взимает процентную ставку i % за любой невыплаченный кредит в конце каждого месяца. Какую сумму денег вы должны платить в месяц (до двух цифр после запятой)?

Предположим, что $d = 576.19$, $m = 2$, $v = 1000$ и $i = 10$ %. Через месяц ваш долг становится $1000 \times (1.1) - 576.19 = 523.81$. Через два месяца ваш долг составит $523.81 \times (1.1) - 576.19 \approx 0$. Если бы нам дали только $m = 2$, $v = 1000$ и $i = 10$ %, как бы мы определили, что $d = 576{,}19$? Другими словами, найдите корень d такой, что функция выплаты долга $f(d, m, v, i) \approx 0$.

Таблица 3.1. Иллюстрация метода деления пополам на примере функции

a	b	$d = (a + b)/2$	Статус: $f(d, m, v, i)$	Действие
0,01	1100,00	550,005	«недолет» на 54,9895	увеличить d
550,005	1100,00	825,0025	«перелет» на 522,50525	уменьшить d
550,005	825,0025	687,50375	«перелет» на 233,757875	уменьшить d
550,005	687,50375	618,754375	«перелет» на 89,384187	уменьшить d
550,005	618,754375	584,379688	«перелет» на 17,197344	уменьшить d
550,005	584,379688	567,192344	«недолет» на 18,896078	увеличить d
567,192344	584,379688	575,786016	«недолет» на 0,849366	увеличить d
…	…	…	несколько итераций спустя…	…
		576,190476	стоп; ошибка теперь меньше ε	ответ = 576,19

Простой способ решить эту задачу – использовать метод деления пополам. Мы выбираем разумный диапазон в качестве отправной точки. Мы хотим за-

[1] Мы используем термин «принцип двоичного поиска» для обозначения подхода D&C, предусматривающего сокращение диапазона возможных ответов вдвое. «Алгоритм двоичного поиска» (поиск индекса элемента в отсортированном массиве), «метод деления пополам» (поиск корня функции) и «двоичный поиск ответа» (обсуждается в следующем подразделе) – все это примеры этого принципа.

фиксировать d в диапазоне [a..b], где a = 0,01, так как должны заплатить не менее одного цента, а $b = (1 + i\,\%) \times v$, поскольку самый ранний срок оплаты – $m = 1$, если мы заплатим ровно $(1 + i\,\%) \times v$ долларов через месяц. В этом примере $b = (1 + 0.1) \times 1000 = 1100.00$ долларов. Чтобы метод деления пополам работал[1], мы должны убедиться, что значения функции для двух крайних точек, которые находятся в начальном действительном диапазоне [a..b], т. е. f(a) и f(b), имеют противоположные знаки (это верно для рассчитанных a и b выше).

Обратите внимание, что для того чтобы получить достаточно хороший ответ методом деления пополам, требуется только $O(\log_2((b - a)/\varepsilon))$ итераций (ошибка меньше, чем пороговая ошибка, которую мы можем допустить).

В этом примере метод деления пополам занимает только $\log_2 1099{,}99/\varepsilon$ попыток. При малом значении ε = 1e-9, мы получим только ≈ 40 итераций. Даже если мы будем использовать меньшее ε = 1e-15, нам все равно понадобится ≈ 60 попыток. Обратите внимание, что количество попыток *мало*. Метод деления пополам гораздо эффективнее по сравнению с исчерпывающим оцениванием каждого возможного значения d = [0.01..1100.00]/ε для этой функции, приведенной в качестве примера. Примечание: метод деления пополам можно записать с помощью цикла, в котором проверяется порядка 40–60 различных значений d (см. нашу реализацию в разделе «Двоичный поиск ответа» далее в этой книге).

Двоичный поиск ответа

Упрощенная формулировка задачи UVa 11935 – Through the Desert выглядит следующим образом: представьте, что вы исследователь, пытающийся пересечь пустыню. Вы используете джип с «достаточно большим» топливным баком, изначально полным. Во время своего путешествия вы сталкиваетесь с рядом событий, таких как «поездка» (которая потребляет топливо), «утечка газа» (еще больше уменьшает количество оставшегося топлива), «по пути встретилась заправочная станция» (что позволяет вам заправиться до полного топливного бака вашего автомобиля), «встреча с механиком» (устраняет все утечки) или «добрались до цели» (конец путешествия). Вам нужно определить наименьшую емкость топливного бака, чтобы ваш джип мог достичь цели. Ответ должен иметь точность до трех цифр после десятичной запятой.

Если нам известна емкость топливного бака джипа, то эта задача – просто задача на моделирование. С самого начала мы можем смоделировать каждое событие по порядку и определить, может ли быть достигнута цель без нехватки топлива. Проблема в том, что мы не знаем вместимость топливного бака джипа – это та величина, которую мы ищем.

Из описания задачи мы можем вычислить, что диапазон возможных ответов находится в интервале [0.000..10000.000] с точностью до трех цифр. Однако таких значений в этом интервале 10 млн. Последовательная проверка каждого значения приведет к тому, что мы получим вердикт жюри «превышение лимита времени» (TLE).

---

[1] Обратите внимание, что требования к методу деления пополам (в котором используется принцип двоичного поиска) немного отличаются от требований к алгоритму двоичного поиска, для которого необходим отсортированный массив.

К счастью, у этой задачи есть одна особенность, которую мы можем использовать. Предположим, что правильный ответ – X. Если задать для емкости топливного бака вашего джипа любое значение в диапазоне [0.000..X-0.001], это не позволит вашему джипу безопасно достигнуть цели. С другой стороны, если задать для емкости топливного бака значение в диапазоне [X..10000.000], то ваш джип сможет безопасно добраться до цели – как правило, с остатком топлива в баке. Эта особенность позволяет нам найти ответ X, используя двоичный поиск! Мы можем применить следующий код, чтобы получить решение для этой задачи.

```
#define EPS 1e-9                // это настраиваемое значение; 1e-9 обычно достаточно мало
bool can(double f) {                            // подробности этого моделирования опущены
  // вернуть true, если джип может достичь цели с емкостью топливного бака f
  // в противном случае вернуть false
}

// внутри int main()
  // используя двоичный поиск, получить ответ, затем приступить к моделированию
  double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
  while (fabs(hi - lo) > EPS) {                                  // пока ответ еще не найден
    mid = (lo + hi) / 2.0;                                    // рассмотрим серединное значение
    if (can(mid)) { ans = mid; hi = mid; }              // сохраним значение, затем перейдем
                                                                // к следующей итерации цикла
    else          lo = mid;
  }

  printf("%.3lf\n", ans);                        // после завершения цикла мы получаем ответ
```

Обратите внимание, что некоторые программисты предпочитают использовать постоянное число итераций уточнения, вместо того чтобы позволять динамически варьировать число итераций, чтобы избежать ошибок точности при проверке условия `fabs(hi - lo) > EPS`, таким образом получая бесконечный цикл. Изменения, необходимые для реализации этого подхода, показаны ниже. Остальная часть кода аналогична приведенной выше.

```
 double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
  for (int i = 0; i < 50; i++) {                    // log_2 ((10000.0 - 0.0) / 1e-9) ~= 43
    mid = (lo + hi) / 2.0;      // повторение цикла 50 раз должно дать достаточную точность
    if (can(mid)) { ans = mid; hi = mid; }
    else          lo = mid;
  }
```

Упражнение 3.3.1.1. Существует альтернативное решение для задачи UVa 11935, в котором не используется метод двоичного поиска ответа. Вы можете рассказать, какой это способ?

Упражнение 3.3.1.2*. Приведенный здесь пример включает двоичный поиск ответа, где ответом является число с плавающей запятой. Измените код, чтобы решить задачу двоичного поиска ответа, когда ответ лежит в целочисленном диапазоне.

Замечания об использовании стратегии «разделяй и властвуй» на олимпиадах по программированию

Прием «разделяй и властвуй» обычно используется в популярных алгоритмах: двоичный поиск и его варианты, сортировка слиянием / быстрая сортировка / пирамидальная сортировка и – структурах данных: дерево двоичного поиска, куча, дерево сегментов, дерево Фенвика и т. д. Однако, исходя из нашего опыта, мы считаем, что наиболее часто подход «разделяй и властвуй» в соревнованиях по программированию используется при решении задач методом двоичного поиска. Если вы хотите преуспеть на олимпиадах по программированию, потратьте время, практикуясь в различных способах его применения.

Как только вы познакомитесь с методом «двоичный поиск ответа», обсуждаемым в этом разделе, перейдите к материалу из раздела 8.4.1, который содержит несколько дополнительных задач программирования, где этот метод используется с другим алгоритмом, который мы обсудим в последних главах данной книги.

Отметим, что задач, которые не относились бы к задачам двоичного поиска и использовали бы подход D&C, не так много. Большинство решений D&C «связаны с геометрией» или «специфичны для задачи», и поэтому мы не будем подробно рассматривать их в этой книге. Однако с некоторыми из них мы встретимся в разделе 8.4.1 (двоичный поиск ответа плюс формулы геометрии), разделе 9.14 (инверсия индекса), разделе 9.21 (возведение матрицы в степень) и разделе 9.29 (выбор наименьшего элемента в массиве).

Задачи по программированию, решаемые с помощью стратегии «разделяй и властвуй»

- Двоичный поиск
  1. UVa 00679 – Dropping Balls (двоичный поиск; существуют решения, использующие операции с битами)
  2. UVa 00957 – Popes (полный поиск + двоичный поиск: `upper_bound`)
  3. UVa 10077 – The Stern-Brocot… (двоичный поиск)
  4. UVa 10474 – Where is the Marble? (простая задача: используйте `sort`, а затем `lower_bound`)
  5. ***UVa 10567 – Helping Fill Bates**** (храните увеличивающиеся индексы каждого символа из «S» в 52 векторах; для каждого запроса выполняйте двоичный поиск позиции символа в корректном векторе)
  6. UVa 10611 – Playboy Chimp (двоичный поиск)
  7. UVa 10706 – Number Sequence (двоичный поиск + некоторые математические идеи)
  8. UVa 10742 – New Rule in Euphomia (используйте «просеивание»; двоичный поиск)
  9. **UVa 11057 – Exact Sum *** (сортировка, для цены `p[i]`, проверьте, существует ли цена (M – `p[i]`), используя двоичный поиск)
  10. UVa 11621 – Small Factors (сгенерируйте числа с множителем 2 и/или 3, `sort`, `upper_bound`)
  11. *UVa 11701 – Cantor* (своеобразный троичный поиск)

12. UVa 11876 – N + NOD (N) ([lower|upper]_bound на отсортированной последовательности N)
13. ***UVa 12192 – Grapevine*** * (входной массив некоторым образом отсортирован; используйте lower_bound, чтобы ускорить поиск)
14. Задача турнира Национальной олимпиады ICPC в Таиланде 2009 года – My Ancestor (автор: Феликс Халим)

- Метод деления пополам, или двоичный поиск результата
  1. **UVa 10341 – Solve It** * (метод деления пополам, рассмотренный в этом разделе; альтернативные решения см. на http://www.algorithmist.com/index.php/UVa 10341)
  2. **UVa 11413 – Fill the…** * (двоичный поиск + симуляция)
  3. UVa 11881 – Internal Rate of Return (метод деления пополам)
  4. UVa 11935 – Through the Desert (двоичный поиск + симуляция)
  5. ***UVa 12032 – The Monkey…*** * (двоичный поиск + симуляция)
  6. *UVa 12190 – Electric Bill* (двоичный поиск + алгебраические вычисления)
  7. IOI 2010 – Quality of Living (двоичный поиск)
     Также см. «разделяй и властвуй» в задачах геометрии (см. раздел 8.4.1).

- Другие задачи, использующие стратегию «разделяй и властвуй»
  1. ***UVa 00183 – Bit Maps*** * (простая задача на использование метода «разделяй и властвуй»)
  2. IOI 2011 – Race («разделяй и властвуй»; посмотрите, использует путь решения вершину или нет)
     См. также о структурах данных, используемых стратегией «разделяй и властвуй» (см. раздел 2.3).

3.4. «Жадные» алгоритмы

Алгоритм считается «жадным», если он делает локально оптимальный выбор на каждом этапе с надеждой в конечном итоге достичь глобально оптимального решения. В некоторых случаях «жадные» алгоритмы работают – решение короткое и работает эффективно. Однако для многих других случаев это не так. Как обсуждалось в учебниках по информатике, например [7, 38], чтобы «жадный» алгоритм работал, решаемая с его помощью задача должна обладать следующими двумя свойствами:

1) иметь оптимальные подструктуры. Оптимальное решение задачи содержит оптимальные решения подзадач;
2) обладать свойством «жадности» (его трудно доказать в условиях нехватки времени на олимпиаде по программированию). Если мы сделаем выбор, который кажется лучшим на данный момент, и приступим к решению оставшейся подзадачи, то найдем оптимальное решение. Нам никогда не придется пересматривать наш предыдущий выбор.

3.4.1. Примеры

Размен монет – «жадная» версия

Формулировка условий задачи: дано целевое число центов V и список номиналов n монет, т. е. у нас есть `coinValue[i]` (в центах) для типов монет `i` ∈ `[0..n-1]`. Каково минимальное количество монет, которое мы должны использовать для размена суммы V? Предположим, что у нас неограниченный запас монет любого типа. Пример: если $n = 4$, `coinValue = {25, 10, 5, 1}` центов[1], и мы хотим набрать $V = 42$ цента, мы можем использовать следующий «жадный» алгоритм: выберите наибольший номинал монеты, не превышающий оставшуюся сумму, т. е. 42 – 25 = 17 → 17 – 10 = 7 → 7 – 5 = 2 → 2 – 1 = 1 → 1 – 1 = 0, всего 5 монет. Это оптимальное решение.

Эта задача включает в себя два компонента, необходимых для успешного применения «жадного» алгоритма:

1) имеет оптимальные подструктуры.
   Мы видели, что при наборе суммы 42 цента из монет мы использовали 25 10 5 1 1.
   Это оптимальное решение поставленной задачи, для которого потребуется всего пять монет.
   Оптимальные решения подзадач содержатся в приведенном решении с пятью монетами, т. е.:
   а) чтобы набрать 17 центов, мы можем использовать монеты 10 5 1 1 (часть решения для 42 центов);
   б) чтобы набрать 7 центов, мы можем использовать монеты 5 1 1 (также часть решения для 42 центов), и т. д.;
2) она обладает свойством «жадности»: если дано произвольное число V, мы можем «жадно» вычесть из него наибольший номинал монеты, который не превышает это число V. Можно доказать (здесь доказательство не приводится для краткости), что использование любых других стратегий не приведет к оптимальному решению – по крайней мере, для этого набора монет указанных номиналов.

Однако этот «жадный» алгоритм работает не для всех наборов монет. Возьмем для примера монеты с номиналами {4, 3, 1} центов. Чтобы получить 6 центов из такого набора монет, «жадный» алгоритм выбрал бы три монеты {4, 1, 1} вместо оптимального решения, которое использует две монеты {3, 3}. Обобщенная версия этой задачи пересматривается далее в этой книге, в разделе 3.5.2 («Динамическое программирование»).

UVa 410 – Station Balance – балансировка груза

Пусть дано $1 \leqslant C \leqslant 5$ камер, в которых может храниться 0, 1 или 2 образца, $1 \leqslant S \leqslant 2C$ образцов и список M масс образцов S; нужно определить, в какой камере следует хранить каждый образец, чтобы минимизировать «дисбаланс» загрузки камер. См. рис. 3.4 для наглядного объяснения[2].

---

[1] Наличие монеты в 1 цент гарантирует, что мы всегда можем набрать любую сумму.

[2] Так как $C \leqslant 5$ и $S \leqslant 10$, мы можем фактически решить эту задачу полным перебором. Однако проще решить данную задачу с помощью «жадного» алгоритма.

Пусть A = $(\sum_{j=1}^{S} M_j)/C$, т. е. A является средней общей массой в каждой из C камер.

Дисбаланс = $\sum_{i=1}^{C} |X_i - A|$, т. е. сумма разностей между общей массой в каждой камере, в т. ч. A, где X_i – общая масса образцов в камере i.

Рис. 3.4 ❖ Визуализация задачи UVa 410 – Station Balance

Эта задача может быть решена с помощью «жадного» алгоритма, но чтобы прийти к такому решению, мы должны сделать несколько замечаний.

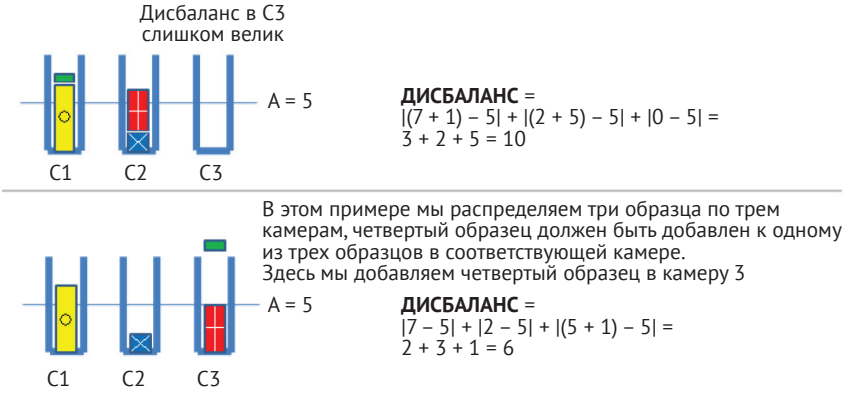


Рис. 3.5 ❖ UVa 410 – Замечания к решению задачи

Замечание 1: если существует пустая камера, обычно выгодным и не ухудшающим значение дисбаланса решением будет перемещать один образец из камеры с двумя образцами в пустую камеру. В противном случае пустая камера вносит больший вклад в дисбаланс, как показано на рис. 3.5 сверху.

Замечание 2: если $S > C$, то $S - C$ образцов должны быть размещены в камере, уже содержащей другие образцы, – принцип «раскладывания по ящикам». См. рис. 3.5 внизу.

Основная идея заключается в том, что решение этой задачи можно упростить с помощью сортировки: если $S < 2C$, добавить $2C - S$ фиктивных образцов

с массой 0. Например, $C = 3$, $S = 4$, $M = \{5, 1, 2, 7\} \rightarrow C = 3$, S = 6, $M = \{5, 1, 2, 7, 0, 0\}$. Затем отсортируйте образцы по их массе так, чтобы $M_1 \leqslant M_2 \leqslant \ldots \leqslant M_{2C-1} \leqslant M_{2C}$. В этом примере $M = \{5, 1, 2, 7, 0, 0\} \rightarrow \{0, 0, 1, 2, 5, 7\}$. При добавлении фиктивных образцов и последующей их сортировке «жадная» стратегия становится «очевидной»:

- объедините образцы с массами M_1 и M_{2C} и поместите их в камеру 1, затем
- объедините образцы с массами M_2 и M_{2C-1} и поместите их в камеру 2 и т. д.

Этот «жадный» алгоритм, известный как балансировка нагрузки, работает! См. рис. 3.6.

Трудно формально описать методы, используемые при разработке этого решения с применением «жадного» алгоритма. Поиск «жадных» решений – это искусство, так же как поиск хороших решений для полного поиска требует творчества. Совет, который вытекает из этого примера: если не существует очевидной «жадной» стратегии, попробуйте отсортировать данные или добавить некоторые «хитрости» и посмотреть, не получится ли «жадная» стратегия.

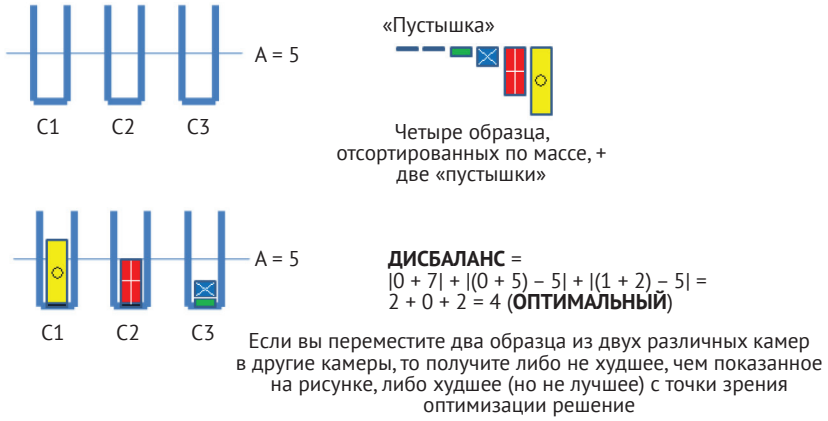


Рис. 3.6 ❖ UVa 410 – Решение с использованием «жадного алгоритма»

UVa 10382 – Watering Grass – покрытие интервала

Формулировка условий задачи: n разбрызгивателей установлены на горизонтальной полосе травы длиной L метров и шириной W метров. Каждый разбрызгиватель центрируется вертикально в своей полосе. Для каждого разбрызгивателя мы задаем его положение как расстояние от левого конца центральной линии и его радиус действия. Какое минимальное количество разбрызгивателей нужно включить, чтобы полить всю полосу травы? Ограничение: $n \leqslant 10\,000$. Иллюстрация к задаче приведена на рис. 3.7 слева. Ответ для этого контрольного примера: шесть разбрызгивателей (помеченных {A, B, D, E, F, H}). Есть два неиспользованных разбрызгивателя: {C, G}.

Мы не можем решить эту задачу методом «грубой силы», где нужно попробовать включить все возможные подмножества разбрызгивателей и посмотреть на результат, поскольку количество разбрызгивателей может доходить до

10 000. Определенно невозможно попробовать все 2^{10000} возможных подмножеств разбрызгивателей.

Эта задача на самом деле является вариантом хорошо известной «жадной» задачи, называемой задачей покрытия интервала. Тем не менее ключ к ее решению даст простой геометрический прием. Оригинальная задача покрытия интервалов имеет дело с интервалами. В этой задаче мы имеем дело с разбрызгивателями, зона действия которых – круги в горизонтальной плоскости, а не простые интервалы. Сначала мы должны попробовать немного изменить задачу, чтобы она напоминала стандартную задачу покрытия интервала.

См. рис. 3.7 справа. Мы можем преобразовать эти круги и горизонтальные полосы в интервалы.

Можем вычислить значение `dx = sqrt(`$R^2 - (W/2)^2$`)`. Предположим, что центр круга расположен в точке с координатами `(x, y)`.

Интервал, представленный этим кругом, равен `[x-dx..x + dx]`. Чтобы понять, почему это работает, обратите внимание, что дополнительный сегмент круга за пределами `dx` от `x` не полностью покрывает полосу в горизонтальной области, которую он охватывает. Если у вас возникли трудности с этим геометрическим преобразованием, см. раздел 7.2.4, в котором рассматриваются основные операции с *прямоугольным треугольником*.

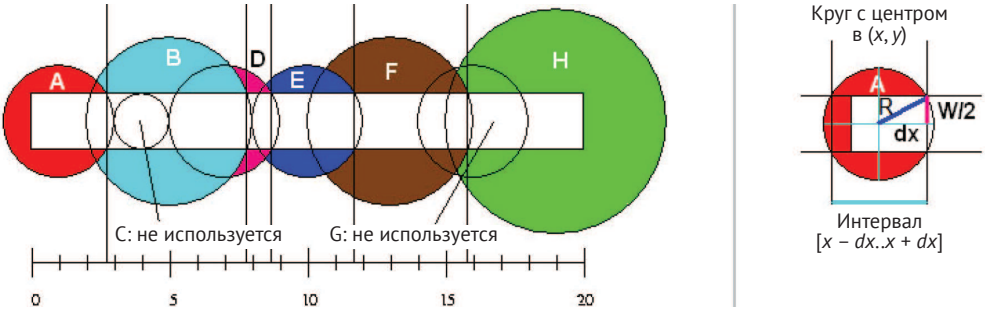


Рис. 3.7 ❖ UVa 10382 – Watering Grass

Теперь, когда мы превратили исходную задачу в задачу покрытия интервала, мы можем использовать следующий «жадный» алгоритм. Во-первых, «жадный» алгоритм сортирует интервалы путем *увеличения* левой конечной точки и *уменьшения* правой конечной точки, если возникают связи. Затем алгоритм обрабатывает интервалы по одному. Он берет интервал, который охватывает зону «как можно правее», и все же непрерывно охватывает всю горизонтальную полосу травы от крайней левой стороны до крайней правой стороны. Он игнорирует интервалы, которые уже полностью покрыты другими (предыдущими) интервалами.

Для контрольного примера, показанного на рис. 3.7 слева, этот «жадный» алгоритм сначала сортирует интервалы, чтобы получить последовательность {A, B, C, D, E, F, G, H}. Затем он обрабатывает их один за другим. Во-первых, он выбирает интервал «A» (он должен это сделать), интервал «B» (соединенный с интервалом «A»), игнорирует «C» (поскольку он входит в интервал «B»), вы-

бирает «D» (он должен это сделать, поскольку интервалы «B» и «E» не соединены, если «D» не используется), выбирает «E», выбирает «F», игнорирует «G» (так как «G» располагается не «как можно правее» и не достигает самой правой стороны полосы травы), выбирает «H» (так как он соединен с интервалом «F» и охватывает больший интервал справа, чем интервал «G», выходя за крайний правый конец полосы травы). Всего мы выбираем шесть разбрызгивателей: {A, B, D, E, F, H}. Это минимально возможное количество разбрызгивателей для данного примера.

UVa 11292 – Dragon of Loowater – сначала отсортируйте входные данные

Формулировка условий задачи: есть n голов драконов и m рыцарей ($1 \leqslant n$, $m \leqslant 20\,000$). У каждой головы дракона есть диаметр, а у каждого рыцаря – рост. Голова дракона диаметром D может быть отрублена рыцарем с ростом H, если $D \leqslant H$. Один рыцарь может отрубить только одну голову дракона. Для заданного списка диаметров голов дракона и списка ростов рыцарей нужно определить, можно ли отрубить все головы дракона. Если да, каков минимальный общий рост (сумма всех ростов) рыцарей, отрубающих головы дракону?

Есть несколько способов решить эту задачу, и мы проиллюстрируем, вероятно, самый простой из них. Эта задача является задачей о паросочетании в двудольном графе (данная тема будет обсуждаться более подробно в разделе 4.7.4), в том смысле, что мы должны максимально сочетать (соединять) определенных рыцарей с головами драконов. Однако эту проблему можно решить с использованием «жадного» алгоритма: каждую голову дракона следует рубить рыцарю с наименьшим ростом – по крайней мере, с таким же ростом, что и диаметр головы дракона. Тем не менее входные данные находятся в произвольном порядке. Если мы отсортируем список диаметров голов дракона и ростов рыцарей за $O(n \log n + m \log m)$, то следующим шагом можем выполнить последовательный просмотр данных за $O(\min(n, m))$, чтобы дать ответ. Это еще один пример, в котором сортировка входных данных может помочь найти стратегию, использующую «жадный» алгоритм.

```
gold = d = k = 0;              // массив "дракон + рыцарь" отсортирован в порядке убывания
while (d < n && k < m) {                        // еще остались головы дракона или рыцари
  while (dragon[d] > knight[k] && k < m) k++;                // находим подходящего рыцаря
  if (k == m) break;                                      // ни один рыцарь не может отрубить
                                          // эту голову дракона, рыцари не справились: S
  gold += knight[k];                                 // король платит это количество золота
  d++; k++;                                  // следующую голову дракона и рыцаря, пожалуйста
}

if (d == n) printf("%d\n", gold);                                // все головы дракона отрублены
else        printf("Рыцари не справились!\n");
```

Упражнение 3.4.1.1*. Какие из следующих наборов монет (все в центах) можно набрать с использованием «жадного» алгоритма «размена монет», ранее приведенного в этом разделе? Если «жадный» алгоритм терпит неудачу на определенном наборе номиналов монет, определите пример наименьшего набора V

центов, для которого он не может быть оптимальным. См. [51], где приведена более подробная информация о поиске таких контрпримеров.

1. $S_1 = \{10, 7, 5, 4, 1\}$
2. $S_2 = \{64, 32, 16, 8, 4, 2, 1\}$
3. $S_3 = \{13, 11, 7, 5, 3, 2, 1\}$
4. $S_4 = \{7, 6, 5, 4, 3, 2, 1\}$
5. $S_5 = \{21, 17, 11, 10, 1\}$

Замечания о задачах, использующих «жадные» алгоритмы, на олимпиадах по программированию

В этом разделе мы обсудили три классические задачи, решаемые с применением «жадных» алгоритмов: размен монет (особый случай), балансировка нагрузки и покрытие интервалов. Для этих классических задач полезно запомнить их решения (в данном случае не обращайте внимания на то, что мы говорили ранее в главе о том, чтобы не слишком полагаться на запоминание). Мы также обсудили важную стратегию решения задач, которая обычно применима к «жадным» задачам: сортировка входных данных для выяснения возможного существования «жадных» стратегий.

В этой книге есть два других классических примера «жадных» алгоритмов, например алгоритм Краскала (и Прима) для задачи о поиске минимального остовного дерева (MST) (см. раздел 4.3) и алгоритм Дейкстры для задачи нахождения кратчайших путей из одной вершины во все остальные вершины графа (SSSP) (см. раздел 4.4.3). Есть еще много известных «жадных» алгоритмов, которые мы решили не обсуждать в этой книге, поскольку они слишком «специфичны для конкретной задачи» и редко появляются на олимпиадах по программированию, например код Хаффмана [7, 38], задача о рюкзаке [7, 38], некоторые задачи планирования заданий и т. д.

Однако на олимпиадах по программированию, проводимых в наши дни (как ICPC, так и IOI), редко предлагаются к решению чисто канонические версии этих классических задач. Использование «жадных» алгоритмов для решения «неклассической» задачи обычно рискованно. Решения с применением «жадных» алгоритмов обычно не получают вердикт жюри «превышение лимита времени» (TLE), поскольку они часто более просты, чем другие варианты, однако велика вероятность получить вердикт «неверный ответ» (WA). Доказательство того, что определенная «неклассическая» задача имеет оптимальную подструктуру и может быть решена с помощью «жадного» алгоритма во время соревнования, может быть слишком трудным или трудоемким. Поэтому обычно программист должен руководствоваться следующим эмпирическим правилом:

> Если размер входых данных «достаточно мал», чтобы решения методом полного перебора или динамического программирования укладывались в отведенный лимит времени (см. раздел 3.5), используйте эти подходы, поскольку оба они будут обеспечивать правильный ответ. Используйте «жадные» алгоритмы только в том случае, если размер входных данных, указанный в формулировке задачи, слишком велик даже для наилучшего алгоритма полного перебора или динамического программирования.

Учитывая вышесказанное, все более и более очевидно, что авторы задач пытаются установить ограничения размера входных данных для задач, которые не позволяют однозначно соотнести стратегии, использующие «жадные» алгоритмы, с определенным диапазоном входных данных, чтобы участники не могли использовать размер входных данных как критерий для быстрого определения алгоритма, с помощью которого нужно решать задачу.

Мы должны отметить, что довольно сложно придумать новые «неклассические» задачи на использование «жадных» алгоритмов. Следовательно, число таких новых задач на «жадные» алгоритмы, предлагаемых на олимпиадах по программированию, меньше, чем число задач, решаемых полным перебором или с помощью динамического программирования.

Задачи по программированию, решаемые с помощью «жадных» алгоритмов (большинство подсказок опущены, чтобы над задачами нужно было подумать)

- Классические задачи, обычно более простые
  1. UVa 410 – Station Balance (обсуждалась выше в этом разделе, балансировка нагрузки)
  2. UVa 01193 – Radar Installation (LA 2519, Пекин'02, покрытие интервала)
  3. UVa 10020 – Minimal Coverage (покрытие интервала)
  4. UVa 10382 – Watering Grass (обсуждалась выше в этом разделе, покрытие интервала)
  5. ***UVa 11264 – Coin Collector*** * (вариант задачи о размене монет)
  6. **UVa 11389 – The Bus Driver Problem** * (балансировка нагрузки)
  7. *UVa 12321 – Gas Station* (покрытие интервала)
  8. ***UVa 12405 – Scarecrow*** * (более легкая задача о покрытии интервала)
  9. IOI 2011 – Elephants (оптимизированное решение с использованием «жадного» алгоритма может быть использовано до подзадачи 3, но более сложные подзадачи 4 и 5 должны решаться с использованием эффективной структуры данных)

- Использование сортировки (или входные данные уже отсортированы)
  1. UVa 10026 – Shoemaker's Problem
  2. *UVa 10037 – Bridge*
  3. UVa 10249 – The Grand Dinner
  4. UVa 10670 – Work Reduction
  5. UVa 10763 – Foreign Exchange
  6. UVa 10785 – The Mad Numerologist
  7. ***UVa 11100 – The Trip, 2007*** *
  8. UVa 11103 – WFF'N Proof
  9. *UVa 11269 – Setting Problems*
  10. **UVa 11292 – Dragon of Loowater** *
  11. UVa 11369 – Shopaholic
  12. UVa 11729 – Commando War

13. UVa 11900 – Boiled Eggs
14. ***UVa 12210 – A Match Making Problem** **
15. *UVa 12485 – Perfect Choir*

- Неклассические, более сложные задачи

1. UVa 00311 – Packets
2. *UVa 00668 – Parliament*
3. UVa 10152 – ShellSort
4. UVa 10340 – All in All
5. UVa 10440 – Ferry Loading II
6. UVa 10602 – Editor Nottobad
7. **UVa 10656 – Maximum Sum (II) ***
8. UVa 10672 – Marbles on a tree
9. UVa 10700 – Camel Trading
10. UVa 10714 – Ants
11. **UVa 10718 – Bit Mask ***
12. *UVa 10982 – Troublemakers*
13. UVa 11054 – Wine Trading in Gergovia
14. **UVa 11157 – Dynamic Frog ***
15. *UVa 11230 – Annoying painting tool*
16. *UVa 11240 – Antimonotonicity*
17. *UVa 11335 – Discrete Pursuit*
18. UVa 11520 – Fill the Square
19. UVa 11532 – Simple Adjacency…
20. UVa 11567 – Moliu Number Generator
21. *UVa 12482 – Short Story Competition*

3.5. Динамическое программирование

Динамическое программирование (далее сокращенно DP) – пожалуй, самая сложная техника решения задач среди четырех методов, обсуждаемых в этой главе. Прежде чем приступать к чтению данного раздела, убедитесь, что вы освоили материал всех предыдущих глав и их разделов. Кроме того, будьте готовы к тому, что вам придется рассматривать множество рекурсивных решений и рекуррентных соотношений.

Ключевыми навыками, которые вам необходимо развить, чтобы овладеть техникой DP, являются способности определять *состояния* задачи и отношения или *переходы* между задачами и их подзадачами. Мы использовали эти навыки ранее при объяснении возвратной рекурсии (см. раздел 3.2.2). Фактически задачи DP при входных данных небольшого размера уже могут быть решены с помощью возвратной рекурсии.

Если вы новичок в технике DP, то можете начать с предположения, что «нисходящий» метод DP («сверху вниз») является своего рода «интеллектуальным» или «более быстрым» вариантом возвратной рекурсии. В этом разделе мы объясним причины, по которым DP часто работает быстрее, чем возвратная рекурсия, для задач, решаемых с помощью техник DP.

DP в основном используется для решения задач *оптимизации* и *подсчета*. Если вы столкнулись с задачей, в которой требуется «минимизировать это», или «максимизировать то», или «сосчитать способы сделать это», то существует (высокая) вероятность того, что эта задача решается с использованием приемов DP. Большинство задач DP в соревнованиях по программированию требуют только оптимального/общего, а не самого оптимального решения, что часто облегчает решение задачи, устраняя необходимость возврата и получения решения. Однако некоторые более сложные задачи DP также требуют, чтобы оптимальное решение было выдано каким-либо образом. Этот раздел поможет улучшить и углубить наше понимание динамического программирования.

3.5.1. Примеры DP

Мы проиллюстрируем концепцию динамического программирования на примере задачи UVa 11450 – Wedding Shopping. Сокращенная формулировка условий задачи: учитывая различные варианты для каждого предмета одежды (например, три модели рубашек, две модели ремней, четыре модели обуви, …) и определенный ограниченный бюджет, наша задача состоит в том, чтобы купить одну модель каждого предмета одежды. Мы не можем потратить денег больше, чем выделенный бюджет, но мы хотим потратить максимально возможную сумму.

Входные данные состоят из двух целых чисел $1 \leqslant M \leqslant 200$ и $1 \leqslant C \leqslant 20$, где M – это бюджет, а C – количество предметов одежды, которые нужно купить, после чего следует информация о предметах одежды C. Для предмета одежды $g \in [0..C-1]$ мы получим целое число $1 \leqslant K \leqslant 20$, которое указывает количество различных моделей для этого предмета одежды g, за которым следуют K целых чисел, указывающих цену каждой модели $\in [1..K]$ этого предмета одежды g.

Выходное значение представляет собой одно целое число, которое указывает максимальную сумму денег, которую мы можем потратить на покупку одного экземпляра каждого предмета одежды без превышения бюджета. Если выделенного нам небольшого бюджета оказывается слишком мало для покупки и решения не существует, мы выводим фразу «решение отсутствует».

Предположим, у нас есть следующий тестовый пример A, где $M = 20$, $C = 3$:

- стоимость трех моделей предмета одежды $g = 0 \rightarrow 6\ 4\ \underline{8}$ // цены не отсортированы на входе;
- стоимость двух моделей предмета одежды $g = 1 \rightarrow 5\ \underline{10}$;
- стоимость четырех моделей предмета одежды $g = 2 \rightarrow \underline{1}\ 5\ 3\ 5$.

Для данного случая ответ – 19, что может быть результатом покупки подчеркнутых предметов (8 + 10 + 1). Это решение не уникально, так как решения (6 + 10 + 3) и (4 + 10 + 5) также являются оптимальными.

Однако предположим, что у нас есть контрольный пример B, где $M = 9$ (ограниченный бюджет), $C = 3$:

- стоимость трех моделей предмета одежды $g = 0 \rightarrow 6\ 4\ 8$;
- стоимость двух моделей предмета одежды $g = 1 \rightarrow 5\ 10$;
- стоимость четырех моделей предмета одежды $g = 2 \rightarrow 1\ 5\ 3\ 5$.

Для этого случая ответом будет «решение отсутствует», потому что даже если мы купим все самые дешевые модели для каждого предмета одежды, общая стоимость (4 + 5 + 1) = 10 все равно превышает наш заданный бюджет $M = 9$.

Чтобы мы могли оценить полезность динамического программирования при решении вышеупомянутой задачи, давайте посмотрим, насколько *другие* подходы, которые мы обсуждали ранее, помогут нам найти решение этой конкретной задачи.

Подход 1: «жадный» алгоритм (результат: WA (Wrong Answer) – неверный ответ)

Поскольку мы хотим максимально увеличить потраченный бюджет, одна из идей использования «жадного» алгоритма (есть и другие «жадные» способы, которые также дают в результате WA) – это выбрать самую дорогую модель каждого предмета одежды g, которая все еще соответствует нашему бюджету. Например, в приведенном выше тестовом примере *A* мы можем выбрать самую дорогую модель 3 предмета одежды $g = 0$ с ценой 8 (теперь наш оставшийся бюджет составляет 20 – 8 = 12), а затем выбрать самую дорогую модель 2 предмета одежды $g = 1$ с ценой 10 (оставшийся бюджет = 12 – 10 = 2), и, наконец, для последнего предмета одежды $g = 2$ мы можем выбрать только модель 1 с ценой 1, так как оставшийся бюджет не позволяет нам покупать другие модели с ценой 3 или 5. Эта «жадная» стратегия «работает» для контрольных примеров *A* и *B* выше и дает одинаковое оптимальное решение (8 + 10 + 1) = 19 и «нет решения» соответственно. Такой алгоритм работает очень быстро[1]: 20 + 20 + … + 20, в общей сложности 20 раз = 400 операций в худшем случае. Однако эта «жадная» стратегия не работает для многих других случаев, таких как приведенный ниже контрпример (контрольный пример C):

Контрольный пример *C*, где $M = 12$, $C = 3$:

- три модели одежды $g = 0 \rightarrow 6\ 4\ 8$;
- две модели одежды $g = 1 \rightarrow 5\ 10$;
- четыре модели одежды $g = 2 \rightarrow 1\ 5\ 3\ 5$.

Наш «жадный» алгоритм выбирает модель 3 для предмета одежды $g = 0$ с ценой 8 (оставшийся бюджет = 12 – 8 = 4), в результате чего у нас не хватает денег, чтобы купить какую-либо модель предмета одежды $g = 1$, что приводит к неправильному ответу «решение отсутствует». Одним из оптимальных решений является 4 + 5 + 3 = 12, которое расходует весь наш бюджет. Оптимальное решение не является уникальным, так как 6 + 5 + 1 = 12 также расходует весь бюджет.

Подход 2: «разделяй и властвуй» (результат: WA (Wrong Answer) – неверный ответ)

Приведенная задача не решается с использованием подхода «разделяй и властвуй». Это связано с тем, что подзадачи, на которые делится исходная задача (описанные в подразделе «Полный перебор» ниже), не являются независи-

[1] Нам не нужно сортировать цены только для того, чтобы найти модель с максимальной ценой, поскольку существует только $K \leqslant 20$ моделей. Последовательного перебора с временной сложностью $O(K)$ будет достаточно.

мыми. Поэтому мы не можем решить их отдельно, используя подход «разделяй и властвуй».

Подход 3: полный перебор (результат: TLE (Time Limit Exceeded) – превышение лимита времени)

Далее давайте посмотрим, можем ли мы решить эту задачу полным перебором (с помощью возвратной рекурсии). Одним из способов использования возвратной рекурсии в этой задаче является реализация функции `shop(money, g)` с двумя параметрами: `money` – текущая сумма денег, которые у нас есть, и `g` – текущий предмет одежды, которую мы рассматриваем как вариант покупки. Пара `(money, g)` – это состояние этой задачи. Обратите внимание, что порядок параметров не имеет значения, например `(g, money)` – также совершенно допустимое состояние. Далее в разделе 3.5.3 мы более подробно обсудим, как выбрать подходящие состояния для задачи.

Мы начинаем с суммы денег `money = M` и предмета одежды `g = 0`. Затем пробуем все возможные модели для предмета одежды `g = 0` (максимум 20 моделей). Если выбрана модель *i*, мы вычитаем цену модели *i* из `money`, а затем повторяем процесс рекурсивным способом для предмета одежды `g = 1` (который также может иметь до 20 различных моделей) и т. д. Остановимся, когда будет выбрана модель для последнего предмета одежды `g = C-1`. Если условие `money < 0` будет выполнено до того, как мы доберемся до предмета одежды `g = C-1`, то мы можем обрубить веточку дерева перебора. Из всех действительных комбинаций мы можем выбрать ту, которая дает наименьшее неотрицательное значение `money`. Это решение – как раз то, при котором мы потратим максимально возможную сумму, которая определяется как разность `(M - money)`.

Мы можем формально определить действия (переходы) рекурсивной функции следующим образом:

1) если `money < 0` (то есть значение `money` становится отрицательным),

   `shop (money, g) = −∞` (на практике мы можем просто вернуть большое отрицательное значение);

2) если была куплена модель последнего предмета одежды, т. е. `g = C`,

   `shop (money, g) = M - money` (это фактическиая сумма денег, которую мы потратили);

3) в общем случае `∀ model ∈ [1..K]` текущего предмета одежды `g`,

   `shop(money, g) = max(shop(money - price[g][model], g + 1))`.

   Мы хотим максимизировать это значение (напомним, что все случаи, когда решения у задачи не существует, дают в результате большое отрицательное значение).

Это решение работает правильно, но оно **очень медленное**! Давайте проанализируем временную сложность для наихудшего случая.

В самом большом тестовом примере предмет одежды `g = 0` имеет до 20 моделей; предмет одежды `g = 1` также имеет до 20 моделей, и все предметы одежды, включая последний предмет одежды `g = 19`, также имеют до 20 моделей. Следовательно, полный перебор выполняется за $20 \times 20 \times \ldots \times 20$ операций в худшем

случае, т. е. 20^{20} = **очень большое** число. Если нам удастся найти решение *только* с помощью алгоритма полного перебора, мы не сможем решить эту задачу.

Подход 4: нисходящее DP (результат: AC (Accepted) – принято)

Чтобы решить поставленную задачу, мы должны использовать концепцию DP, поскольку эта задача удовлетворяет двум предварительным условиям для применимости DP:

1) эта задача имеет оптимальные подструктуры[1].
   Это проиллюстрировано в третьем цикле полного перебора выше: решение подзадачи является частью решения исходной задачи. Другими словами, если мы выбираем модель *i* для предмета одежды *g* = 0, то, чтобы наш окончательный выбор был оптимальным, наш выбор для одежды *g* = 1 и далее также должен быть оптимальным выбором для уменьшившегося бюджета *M* – *price*, где *price* – цена модели *i*;
2) эта задача имеет перекрывающиеся между собой подзадачи.
   Это ключевая характеристика DP! Пространство поиска данной задачи не такое большое, как грубая оценка верхней границы 20^{20}, полученная ранее, потому что многие подзадачи перекрываются между собой!

Давайте проверим, действительно ли эта задача имеет перекрывающиеся между собой подзадачи. Предположим, что для одного предмета одежды `g` есть две модели с *одинаковой* ценой `p`. В этом случае после выбора любой модели алгоритм полного перебора перейдет к **той же** подзадаче `shop(money - p, g + 1)`. Эта ситуация также произойдет, если какая-то комбинация `money` и выбранной цены модели приведет к ситуации $\texttt{money}_1 - \texttt{p}_1 = \texttt{money}_2 - \texttt{p}_2$ для одного и того же предмета одежды `g`. В решении методом полного перебора эта ситуация приведет к тому, что вычисления для одной и той же подзадачи будут выполняться более одного раза, что крайне неэффективно.

Итак, сколько *разных* подзадач (или **состояний** в терминологии DP) существует для этой задачи? Всего лишь $201 \times 20 = 4020$. Есть только 201 возможное значение для `money` (от 0 до 200 включительно) и 20 возможных значений для предметов одежды `g` (от 0 до 19 включительно). Для каждой из подзадач вычисления должны выполняться только один раз. Если мы сможем этого достичь, мы решим нашу задачу намного быстрее.

Реализация этого решения DP удивительно проста. Если у нас уже есть рекурсивное решение для возвратной рекурсии (см. повторения – или переходы в терминологии DP, – показанные выше в подходе к решению задачи полным перебором), мы можем реализовать нисходящее DP, добавив следующие два дополнительных шага:

1) инициализируйте[2] таблицу «memo» (таблицу, хранящую промежуточные значения в памяти) фиктивными значениями, которые не исполь-

---

[1] Оптимальные подструктуры также необходимы для работы «жадных» алгоритмов, но в этой задаче отсутствует «свойство жадности», что делает ее нерешаемой с помощью «жадных» алгоритмов.

[2] Для программистов, использующих C/C++, функция `memset` в `<cstring>` – подходящий инструмент для выполнения этого шага.

зуются в задаче, например «–1». Таблица должна иметь размерность, соответствующую состояниям задачи;

2) в начале рекурсивной функции проверьте, было ли это состояние вычислено ранее:
   a) если это так, просто верните значение из таблицы «memo»; временная сложность $O(1)$.
      (Это объясняет происхождение термина «мемоизация»);
   b) если оно не было вычислено, выполните вычисление как обычно (только один раз), а затем сохраните вычисленное значение в таблице «memo», чтобы последующие вызовы этой подзадачи (состояния) немедленно возвращали результат.

Анализ простого[1] решения DP прост. Если задача имеет M различных состояний, то для ее решения потребуется $O(M)$ памяти. Если вычисление одного состояния (сложность перехода DP) требует $O(k)$ шагов, тогда общая временная сложность составляет $O(kM)$. Для задачи UVa 11450 $M = 201 \times 20 = 4020$ и $k = 20$ (так как нам нужно перебрать не более 20 моделей для каждого предмета одежды `g`). Таким образом, временная сложность составляет не более $4020 \times 20 = 80\,400$ операций на тестовый пример, что является хорошим результатом.

Мы приводим написанный нами код ниже для иллюстрации, особенно для тех, кто никогда раньше не писал код, реализующий нисходящий алгоритм DP. Внимательно изучите этот код и убедитесь, что он действительно очень похож на код возвратной рекурсии, который вы видели в разделе 3.2.

```
/* UVa 11450 - Wedding Shopping - Сверху вниз */
// предполагаем, что необходимые библиотечные файлы были включены
// этот код похож на код возвратной рекурсии
// фрагменты кода, специфичные для нисходящего DP,
// прокомментированы соответственно: "TOP-DOWN"

int M, C, price[25][25];                                  // price[g (<= 20)][model (<= 20)]
int memo[210][25];                      // TOP-DOWN: таблица memo[money (<= 200)][g (<= 20)]
int shop(int money, int g) {
  if (money < 0) return -1000000000;                   // решение не существует, возвращаем
                                                      // очень большое неотрицательное число
  if (g == C) return M - money;                     // куплен последний предмет одежды, конец
  // если закомментировать следующую строку, то нисходящее DP
  // превращается в возвратную рекурсию !!
  if (memo[money][g] != -1) return memo[money][g];                  // TOP-DOWN: мемоизация
  int ans = -1;                     // начните с цифры -ve, так как все цены неотрицательные
  for (int model = 1; model <= price[g][0]; model++)                // перебор всех моделей
    ans = max(ans, shop(money - price[g][model], g + 1));
  return memo[money][g] = ans; }                 // TOP-DOWN: мемоизация и возврат значения

int main() {                          // легко кодировать, если вы уже знакомы с этим методом
  int i, j, TC, score;
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
```

[1] «Простого» означает «без вариантов оптимизации, которые мы увидим позже в этом разделе и в разделе 8.3».

```
    for (i = 0; i < C; i++) {
      scanf("%d", &price[i][0]);                              // сохраняем K в price[i][0]
      for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);
    }
    memset(memo, -1, sizeof memo);                 // TOP-DOWN: инициализируем таблицу memo
    score = shop(M, 0);                                          // запускаем нисходящее DP
    if (score < 0) printf("no solution\n");
    else           printf("%d\n", score);
} } // return 0;
```

Мы хотим воспользоваться возможностью проиллюстрировать другой стиль, используемый при реализации решений DP (применим только для программистов, пишущих на C/C++). Вместо того чтобы часто обращаться к определенной ячейке в таблице memo, мы можем использовать локальную *ссылочную* переменную для хранения адреса памяти требуемой ячейки в таблице memo, как показано ниже. Эти два стиля кодирования не сильно различаются, и вам решать, какой стиль вы предпочитаете.

```
int shop(int money, int g) {
  if (money < 0) return -1000000000;                  // важен порядок > 1 простых случаев
  if (g == C) return M - money;                          // если мы дошли до этой строки,
                                                 // кол-во денег (money) не может быть < 0
  int &ans = memo[money][g];                                    // запомним адрес памяти
  if (ans != -1) return ans;
  for (int model = 1; model <= price[g][0]; model++)
    ans = max(ans, shop(money - price[g][model], g + 1));
  return ans;                        // ans (или memo[money][g]) обновляется непосредственно
}
```

Файл исходного кода: *ch3_02_UVa11450_td.cpp/java*

Подход 5: восходящее DP (результат: AC (Accepted) – принято)

Существует еще один способ реализации решения DP, часто называемый «восходящим» DP («снизу вверх»). На самом деле это «естественная форма» DP, так как DP изначально было известно как «табличный метод» (метод вычислений с использованием таблицы). Основные этапы построения решения восходящего DP:

1) определите необходимый набор параметров, которые однозначно описывают задачу (состояние). Этот шаг аналогичен тому, что мы уже обсуждали, когда рассматривали возвратную рекурсию и нисходящее DP;
2) если для представления состояний требуется N параметров, подготовьте N-мерную таблицу DP, выделив одну запись на каждое состояние. Это эквивалентно таблице memo в нисходящем DP. Тем не менее есть различия. В восходящем DP нам нужно инициализировать только некоторые ячейки таблицы DP известными начальными значениями (основные случаи). Напомним, что в нисходящем DP мы инициализируем таблицу memo полностью фиктивными значениями (обычно –1), чтобы указать, что мы еще не вычислили ее значения;

3) теперь, когда ячейки/состояния основного случая в таблице DP уже заполнены, определите ячейки/состояния, которые могут быть заполнены за следующие шаги (переходы). Повторяйте этот процесс, пока таблица DP не будет заполнена. Для восходящего DP эта часть обычно выполняется с помощью итераций с использованием циклов (более подробно об этом позже).

Для задачи UVa 11450 мы можем записать решение с использованием восходящего DP следующим образом: мы описываем состояние подзадачи с двумя параметрами: текущий предмет одежды `g` и текущее количество денег `money`. Эта формулировка состояния по существу эквивалентна состоянию в нисходящем DP выше, за исключением того, что мы изменили порядок, чтобы сделать `g` первым параметром (таким образом, значения `g` являются индексами строк таблицы DP, чтобы мы могли воспользоваться дружественным к кешу обходом строк в двумерном массиве, см. советы по ускорению в разделе 3.2.3). Затем мы инициализируем двумерную таблицу (булеву матрицу) `reachable[g][money]` размером 20×201. Первоначально только для ячеек/состояний, достижимых при покупке любой из моделей первого предмета одежды `g = 0`, устанавливается значение `true` (в первой строке). Возьмем тестовый пример *A*, приведенный выше, в качестве примера. На рис. 3.8 в верхней части лишь столбцы «20 – 6 = 14», «20 – 4 = 16» и «20 – 8 = 12» в строке 0 изначально имеют значение `true`.

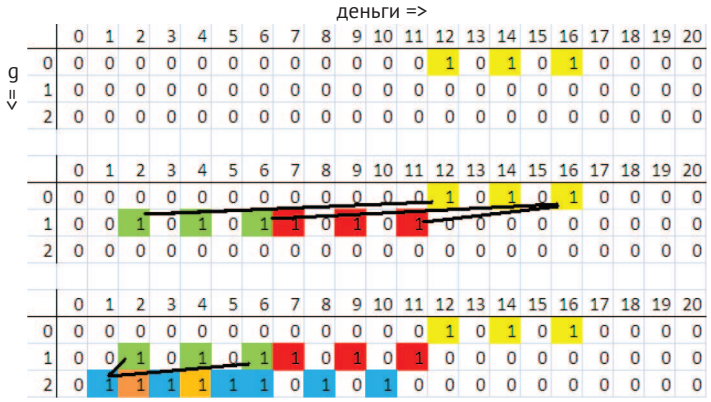


Рис. 3.8 ❖ Восходящее DP (столбцы с 21 по 200 не показаны)

Теперь мы переходим от второго предмета одежды `g = 1` (вторая строка) к последнему предмету одежды `g = C-1 = 3-1 = 2` (третья и последняя строка) в построчном порядке (строка за строкой). Если значение `reachable[g-1][money]` истинно («true»), то следующее состояние `reachable[g][money-p]`, где `p` – цена модели текущего предмета одежды `g`, также достижимо, пока значение второго параметра (`money`) неотрицательно. См. рис. 3.8 (среднюю часть), где `reachable[0][16]` распространяется на `reachable[1][16-5]` и `reachable[1][16-10]`, когда куплены модели с ценой 5 и 10 предмета одежды `g = 1` соответственно; `reachable[0][12]` распространяется на `reachable[1][12-10]`, когда покупается модель с ценой 10 предмета одежды `g = 1`, и т. д. Мы повторяем этот процесс запол-

нения таблицы строка за строкой, пока наконец не закончим заполнение последней строки[1].

Наконец, ответ может быть найден в последней строке, когда g = C-1. Найдите в этой строке состояние, которое является ближайшим к индексу 0 и достижимым. На рис. 3.8, внизу, ячейка таблицы reachable[2][1] дает нам ответ. Это означает, что мы можем достичь состояния (money = 1), купив некоторую комбинацию различных моделей одежды. На самом деле необходимый нам окончательный ответ – M - money, или в данном случае 20-1 = 19. Ответ «нет решения» будет выдан в том случае, если в последней строке нет достижимого состояния (где reachable[C-1][money] имеет значение «true»). Ниже мы приводим нашу реализацию для сравнения с версией нисходящего DP.

```
/* UVa 11450 - Wedding Shopping - Восходящее DP */
// предположим, что необходимые файлы библиотеки были включены

int main() {
  int g, money, k, TC, M, C;
  int price[25][25];                                  // цена[g (<= 20)][модель (<= 20)]
  bool reachable[25][210];                  // таблица reachable[g (<= 20)][money (<= 200)]
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (g = 0; g < C; g++) {
      scanf("%d", &price[g][0]);                          // мы сохраняем K в price[g][0]
      for (money = 1; money <= price[g][0]; money++)
        scanf("%d", &price[g][money]);
    }
    memset(reachable, false, sizeof reachable);                          // очистить все
    for (g = 1; g <= price[0][0]; g++)          // инициализация значений (основные случаи)
      if (M - price[0][g] >= 0)                              // для предотвращения выхода
                                                // значений индекса массива за пределы
        reachable[0][M - price[0][g]] = true;        // возьмем первый предмет одежды g = 0

    for (g = 1; g < C; g++)                         // для всех оставшихся предметов одежды
      for (money = 0; money < M; money++) if (reachable[g-1][money])
        for (k = 1; k <= price[g][0]; k++) if (money - price[g][k] >= 0)
          reachable[g][money - price[g][k]] = true;              // также теперь достижимо

    for (money = 0; money <= M && !reachable[C - 1][money]; money++);

    if (money == M + 1) printf("no solution\n");                 // в последней строке нет
                                                                       // включенных битов
    else                printf("%d\n", M - money);
  }
} // return 0;
```

Файл исходного кода: *ch3_03_UVa11450_bu.cpp/java*

[1] Позднее в разделе 4.7.1 мы обсудим подход динамического программирования как обход (неявного) направленного ациклического графа. Чтобы избежать ненужного «возврата» вдоль этого направленного ациклического графа, мы должны приходить в вершины графа в их топологическом порядке (см. раздел 4.2.5). Порядок, в котором мы заполняем таблицу DP, является топологической сортировкой основного неявного направленного ациклического графа.

Существует преимущество в написании решений DP методом «восходящего DP». Для задач, в которых нам нужна только последняя строка таблицы DP (или, в более общем случае, последний обновленный фрагмент всех состояний), чтобы определить решение, – и к таким задачам относится рассмотренная нами выше задача, – мы можем оптимизировать использование памяти в нашем решении DP, пожертвовав одним измерением в нашей таблице DP. Для более сложных задач DP с жесткими требованиями к памяти этот «прием экономии места» может оказаться полезным, хотя общая временная сложность представленного решения не меняется.

Давайте еще раз посмотрим на рис. 3.8. Нам нужно сохранить только две строки: текущую строку, которую мы обрабатываем, и предыдущую строку, которую мы обработали. Чтобы вычислить строку 1, нам нужно знать лишь столбцы в строке 0, для которых в функции `reachable` достижимо значение «true». Чтобы вычислить строку 2, нам также нужно знать только столбцы в строке 1, для которых в функции `reachable` достижимо значение «true». В общем, для вычисления строки `g` нам нужны лишь значения из предыдущей строки `g - 1`. Таким образом, вместо того чтобы хранить булеву матрицу `reachable[g][money]` размерностью 20×201, мы можем просто хранить `reachable[2][money]` размером 2×201. Мы можем использовать этот программный прием, чтобы ссылаться на одну строку как на «предыдущую» строку, а на другую строку – как на «текущую» (например, `prev = 0`, `cur = 1`), а затем поменять их местами (т. е. сейчас `prev = 1`, `cur = 0`), так как мы вычисляем значения в таблице DP восходящим методом, строка за строкой. Обратите внимание, что для рассматриваемой задачи экономия памяти не является значительной. Для более сложных задач DP, например когда вместо 20 моделей у вас могут быть тысячи моделей одежды, этот прием с экономией памяти может быть очень важен.

Нисходящее или восходящее DP

Хотя в обоих методах используются «таблицы», способ заполнения таблицы при восходящем DP отличается от метода заполнения таблицы memo при нисходящем DP. В нисходящем DP записи таблицы memo заполняются «по мере необходимости» посредством самой рекурсии. В восходящем DP мы использовали правильный «порядок заполнения таблицы DP», вычисляя значения таким образом, чтобы предыдущие значения, необходимые для обработки текущей ячейки таблицы, уже были получены. Этот порядок заполнения таблицы является топологическим порядком неявного направленного ациклического графа (более подробное объяснение приведено в разделе 4.7.1) в структуре рекурсии. Для большинства задач DP топологический порядок может быть достигнут просто с определением надлежащей последовательности некоторых (вложенных) циклов.

Для большинства задач DP оба стиля одинаково хороши, и решение об использовании определенного стиля DP является вопросом предпочтения. Однако для более сложных задач DP один из стилей может оказаться лучше другого. Чтобы помочь вам понять, какой стиль следует использовать при решении задач с применением DP, изучите сравнительную таблицу двух методов DP – восходящего и нисходящего (см. табл. 3.2).

Таблица 3.2. Сравнительная таблица нисходящего и восходящего методов DP

Нисходящее DP	Восходящее DP
Плюсы: 1. Это естественное преобразование из обычной рекурсии полного перебора 2. Переходит к выполнению подзадач только при необходимости (иногда это быстрее)	Плюсы: 1. Быстрее, если много подзадач выполняется по нескольку раз, так как нет затрат на рекурсивные вызовы 2. Может сэкономить место в памяти при использовании метода «экономии места»
Минусы: 1. Работает медленнее, если многие подзадачи выполняются повторно из-за накладных расходов на вызовы функций (это обычно не приводит к снижению баллов в соревнованиях по программированию) 2. Если имеется *M* состояний, требуется размер таблицы *O*(*M*), что может привести к ошибке превышения лимита памяти (MLE) для некоторых более сложных задач (кроме случаев, когда мы используем прием, показанный в разделе 8.3.4)	Минусы: 1. Для программистов, склонных к использованию рекурсии, этот стиль может быть неинтуитивным 2. Если имеется *M* состояний, восходящее DP проходит снизу вверх и заполняет значения всех этих *M* состояний

Представление оптимального решения

Во многих задачах DP требуется вывести только значение оптимального решения (например, UVa 11450 выше). Тем не менее многие участники олимпиад по программированию приходят в замешательство, когда они понимают, что необходимо вывести (распечатать) само оптимальное значение. Мы расскажем о двух известных способах сделать это.

Первый способ в основном используется в подходе восходящего DP (и все еще применим для нисходящего DP), где в каждом состоянии мы храним информацию о предшествующем состоянии. Если существует более одного оптимального предыдущего состояния и нам необходимо вывести все оптимальные решения, мы можем сохранить эти выводимые предыдущие состояния в виде списка. Как только мы получим оптимальное конечное состояние, мы можем выполнить рекурсивный возврат из оптимального конечного состояния и проследовать по оптимальным переходам, записанным в каждом состоянии, пока не достигнем одного из основных (начальных) случаев. Если в задаче требуется вывести все оптимальные решения, эта процедура поиска с возвратами выведет их все. Однако большинство авторов задач обычно задают дополнительные выходные критерии, чтобы выбранное оптимальное решение было уникальным (для упрощения процесса выставления оценки за решение задачи).

Пример: см. рис. 3.8, нижнюю часть. Оптимальное конечное состояние – `reachable[2][1]`. Предыдущим состоянием для этого оптимального конечного состояния является `reachable[1][2]`. Теперь вернемся к `reachable[1][2]`. Далее см. рис. 3.8, среднюю часть. Предыдущее состояние по отношению к `reachable[1][2]` – `reachable[0][12]`. Затем мы возвращаемся к `reachable[0][12]`. Поскольку это уже одно из начальных основных состояний (в первой строке), мы знаем, что

оптимальное решение: (20 → 12) = стоимость 8, затем (12 → 2) = стоимость 10, потом (2 → 1) = стоимость 1. Однако, как упоминалось ранее в описании задачи, эта задача может иметь несколько других оптимальных решений, например мы также можем следовать по пути: `reachable[2][1]` → `reachable[1][6]` → `reachable[0][16]`, который представляет другое оптимальное решение: (20 → 16) = стоимость 4, затем (16 → 6)) = стоимость 10, тогда (6 → 1) = стоимость 5.

Второй способ применим главным образом к подходу нисходящего DP, где мы используем рекурсию и мемоизацию для выполнения той же задачи. Внося изменения в код нисходящего DP, показанный в подходе 4 выше, мы добавим еще одну функцию `void print_shop(int money, int g)`, которая имеет ту же структуру, что и `int shop(int money, int g)`, за исключением того, что она использует сохраненные значения в таблице memo, чтобы восстановить решение. Пример реализации (который выводит только одно оптимальное решение) показан ниже:

```
void print_shop(int money, int g) {                          // эта функция возвращает void
  if (money < 0 || g == C) return;                 // одинаковые основные (начальные) случаи
  for (int model = 1; model <= price[g][0]; model++)              // какая модель выбрана?
    if (shop(money - price[g][model], g + 1) == memo[money][g]) {
      printf("%d%c", price[g][model], g == C-1 ? '\n' : '-');                      // эта
      print_shop(money - price[g][model], g + 1);              // рекурсия к этому состоянию
      break;                                             // не переходите к другим состояниям
}   }
```

Упражнение 3.5.1.1. Чтобы убедиться в том, что вы понимаете задачу UVa 11450, обсуждаемую в этом разделе, определите, какие выходные данные для тестового примера D приведены ниже.

Контрольный пример D, пусть $M = 25$, $C = 3$:

- стоимость трех моделей одежды $g = 0 \rightarrow 6\ 4\ 8$;
- стоимость двух моделей одежды $g = 1 \rightarrow 10\ 6$;
- стоимость четырех моделей одежды $g = 2 \rightarrow 7\ 3\ 1\ 5$.

Упражнение 3.5.1.2. Является ли следующее описание состояния: `shop(g, model)`, где `g` представляет текущий элемент одежды, а `model` – текущую модель, подходящим и исчерпывающим для задачи UVa 11450?

Упражнение 3.5.1.3. Добавьте прием, позволяющий экономить пространства в код, иллюстрирующий восходящее DP в разделе, где описан подход 5.

3.5.2. Классические примеры

Задача UVa 11450 – Wedding Shopping, описанная выше, является (относительно простой) неклассической задачей динамического программирования, в которой *мы сами* должны были придумать правильные состояния DP и переходы.

Однако существует много других *классических* задач с эффективными решениями DP, то есть такие задачи, для которых состояния DP и переходы хорошо известны. Поэтому каждый участник олимпиад по программированию, который хочет преуспеть в ICPC или IOI, должен изучить и освоить методы

решения таких классических задач DP. В этом разделе мы перечисляем шесть классических задач DP и приводим их решения. Примечание: как только вы усвоите основу этих решений DP, попробуйте выполнить упражнения по программированию, в которых встречаются различные *вариации*.

1. Максимальная сумма диапазона 1D (Max 1D Range Sum)

Упрощенная формулировка задачи UVa 507 – Jill Rides Again: дан целочисленный массив A, содержащий $n \leqslant 20K$ ненулевых целых чисел, определите максимальную сумму 1D-диапазона A. Другими словами, найдите максимальную сумму диапазона (выполните запрос суммы диапазона, RSQ), ограниченного двумя индексами `i` и `j` в промежутке `[0..n-1]`, то есть вычислите следующее значение: `A[i] + A[i + 1] + A[i + 2] + ... + A[j]` (также см. раздел 2.4 .3 и 2.4.4).

Алгоритм полного перебора, который перебирает все возможные пары `i` и `j` (число таких комбинаций $O(n^2)$), вычисляет требуемое значение `RSQ(i, j)` за $O(n)$ и, наконец, выбирает максимальное значение, имеет общую временную сложность $O(n^3)$. При верхнем ограничении n, равном $20K$, такое решение получит вердикт жюри «превышение лимита времени» (TLE).

В разделе 2.4.4 мы обсудили следующую стратегию DP: предварительная обработка массива A путем вычисления `A[i] += A[i-1]` $\forall$ `i` $\in$ `[1..n-1]` таким образом, чтобы элемент массива `A[i]` содержал сумму целых чисел в подмассиве `A[0..i]`. Теперь мы можем вычислить `RSQ(i, j)` за время $O(1)$: `RSQ(0, j) = A[j]` и `RSQ(i, j) = A[j] - A[i-1]` $\forall$`i> 0`. Таким образом, приведенный выше алгоритм полного поиска можно заставить работать в $O(n^2)$. Для $n \leqslant 20K$ это все еще слишком медленно (TLE). Тем не менее этот метод может оказаться полезным в других случаях (см. использование этого способа нахождения максимальной суммы диапазона (1D) в разделе 8.4.2).

Есть лучший алгоритм для решения этой задачи. Ниже показана основная часть алгоритма Джея Кадана с временной сложностью $O(n)$ (может рассматриваться как «жадный» или DP-алгоритм) для решения данной задачи.

```
// внутри int main()
  int n = 9, A[] = { 4, -5, 4, -3, 4, 4, -4, 4, -5 };                  // исходный массив A
  int sum = 0, ans = 0;                          // важно, и должно быть инициализировано в 0
  for (int i = 0; i < n; i++) {                     // последовательный просмотр данных, O(n)
    sum += A[i];                               // мы увеличиваем эту сумму с нарастающим итогом
                                                          // с использованием "жадного" метода
    ans = max(ans, sum);                             // мы сохраняем максимальный RSQ в целом
    if (sum < 0) sum = 0;                                      // но сбрасываем текущую сумму
  }                                          // если она когда-либо примет значение меньше 0
  printf("Max 1D Range Sum = %d\n", ans);
```

Файл исходного кода: *ch3_04_Max1DRangeSum.cpp/java*

Ключевая идея алгоритма Кадана состоит в том, чтобы сохранять текущую сумму целых чисел и устанавливать ее в 0, используя «жадный» метод, если текущая сумма становится меньше 0. Такая стратегия выбрана потому, что «перезапуск» с 0 всегда лучше, чем продолжение работы алгоритма с отрицательной промежуточной суммы. Алгоритм Кадана необходимо использовать для решения задачи UVa 507 при $n \leqslant 20K$.

Обратите внимание, что мы также можем рассматривать этот алгоритм Кадана как решение DP. На каждом шаге у нас есть два варианта: мы можем использовать ранее накопленную максимальную сумму или начать новый диапазон. Таким образом, переменная DP `dp(i)` представляет собой максимальную сумму диапазона целых чисел, который заканчивается элементом `A[i]`. И окончательный ответ является максимальным по всем значениям `dp(i)`, где `i` ∈ `[0..n-1]`. Если допустимы диапазоны нулевой длины, то 0 также должен рассматриваться как возможный ответ. Вышеприведенная реализация является эффективной версией, которая использует прием экономии места, рассмотренный ранее.

2. Максимальная сумма диапазона 2D (Max 2D Range Sum)

Сокращенная формулировка условий задачи UVa 108 – Maximum Sum: пусть задана квадратная матрица целых чисел `A` размерностью $n \times n$ ($1 \leqslant n \leqslant 100$), где каждое целое число варьируется в пределах `[-127..127]`. Найдите подматрицу `A` с максимальной суммой. Например: матрица 4×4 ($n = 4$) в табл. 3.3A (ниже) имеет подматрицу 3×2 в левом нижнем углу с максимальной суммой 9 + 2 – 4 + 1 – 1 + 8 = 15.

Таблица 3.3. UVa 108 – Maximum Sum

A					B					C				
	0	-2	-7	0		0	-2	-9	-9		0	-2	-9	-9
	9	2	-6	2		9	9	-4	2		9	9	-4	2
	-4	1	-4	1		5	6	-11	-8		5	6	-11	-8
	-1	8	0	-2		4	13	-4	-3		4	13	-4	-3

Попытка решения этой задачи «в лоб» с использованием метода полного перебора, как показано ниже, не работает, поскольку выполняется за $O(n^6)$. Для самого большого объема тестовых данных, где $n = 100$, алгоритм с временной сложностью $O(n^6)$ – слишком медленный.

```
maxSubRect = -127*100*100;                  // минимально возможное значение для этой задачи
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++)             // начальная координата
  for (int k = i; k < n; k++) for (int l = j; l < n; l++) {          // конечная координата
    subRect = 0;                            // суммировать элементы в этом подпрямоугольнике
    for (int a = i; a <= k; a++) for (int b = j; b <= l; b++)
      subRect += A[a][b];
    maxSubRect = max(maxSubRect, subRect); }                                    // это ответ
```

Решение для задачи о максимальной сумме 1D-диапазона (в предыдущем подразделе) может быть расширено до двух (или более) измерений при условии правильного применения принципа включения-исключения. Единственное отличие состоит в том, что хотя мы имели дело с перекрывающимися поддиапазонами в Max 1D Range Sum, мы будем иметь дело с перекрывающимися подматрицами в Max 2D Range Sum. Мы можем превратить входную матрицу $n \times n$ в *матрицу накопленной суммы* размерностью $n \times n$, где `A[i][j]` больше не содержит свои собственные значения, а содержит сумму всех элементов в подматрице, индексы которой изменяются от `(0, 0)` до `(i, j)`. Это может быть сделано одновременно с чтением входных данных, и при этом такое решение будет от-

рабатывать за $O(n^2)$. Приведенный ниже код превращает исходную квадратную матрицу (см. рис. 3.9.A) в матрицу накопленной суммы (см. рис. 3.9B).

```
scanf("%d", &n);                                  // размерность исходной квадратной матрицы
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("%d", &A[i][j]);
  if (i > 0) A[i][j] += A[i - 1][j];                      // при возможности добавим сверху
  if (j > 0) A[i][j] += A[i][j - 1];                       // при возможности добавим слева
  if (i > 0 && j > 0) A[i][j] -= A[i - 1][j - 1];           // избегаем двойного вычисления
}                                                           // принцип включения-выключения
```

С помощью матрицы суммы мы можем определить сумму любой подматрицы, содержащей элементы в диапазоне от `(i, j)` до `(k, l)` за $O(1)$, используя приведенный ниже код. Например, давайте вычислим сумму от `(1, 2)` до `(3, 3)`. Разобьем эту сумму на четыре части и вычислим `A[3][3] - A[0][3] - A[3][1] + A[0][1] = -3 - 13 - (-9) + (-2) = -9`, как показано в табл. 3.3C. С этой постановкой задачи для подхода DP, с временной сложностью $O(1)$, задача Max 2D Range Sum может быть решена за $O(n^4)$. Для самого большого тестового примера UVa 108 с $n = 100$ это все еще достаточно быстро.

```
maxSubRect = -127*100*100;                  // минимально возможное значение для этой задачи
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++)            // начальная координата
  for (int k = i; k < n; k++) for (int l = j; l < n; l++) {         // конечная координата
    subRect = A[k][l];                    // сумма всех элементов от (0, 0) до (k, l): O(1)
    if (i > 0) subRect -= A[i - 1][l];                                              // O(1)
    if (j > 0) subRect -= A[k][j - 1];                                              // O(1)
    if (i > 0 && j > 0) subRect += A[i - 1][j - 1];                                 // O(1)
    maxSubRect = max(maxSubRect, subRect); }                                  // это ответ
```

Файл исходного кода: *ch3_05_UVa108.cpp/java*

Из этих двух примеров – задач на нахождение максимальной суммы в диапазоне для случаев 1D и 2D – мы видим, что не для каждой задачи, где нужно исследовать диапазоны значений, требуется дерево сегментов или дерево Фенвика, как обсуждалось в разделе 2.4.3 или 2.4.4. Задачи, связанные с диапазонами статических входных данных, часто решаются с помощью методов DP. Стоит также упомянуть, что для решения задач, связанных с диапазонами, очень естественно использовать восходящие методы DP, поскольку операнд уже является одномерным или двумерным массивом. Мы все еще можем написать рекурсивное нисходящее решение для задачи на диапазоны, но это решение не столь органично.

3. Наибольшая возрастающая подпоследовательность (LIS)

Пусть задана последовательность `{A[0], A[1], ..., A[n-1]}`; определите ее *наибольшую возрастающую подпоследовательность НВП* (Longest Increasing Subsequence, LIS)[1]. Обратите внимание, что эти «подпоследовательности» не

[1] Существуют и другие варианты этой задачи, в числе которых – нахождение максимальной убывающей подпоследовательности и максимальной невозрастающей/неубывающей подпоследовательности. Задачу нахождения увеличивающихся под-

обязательно являются соприкасающимися множествами. Пример: $n = 8$, $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$. Возрастающей подпоследовательностью с максимальным числом элементов, равным 4, в данном примере будет последовательность $\{-7, 2, 3, 8\}$.

Индекс	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

Рис. 3.9 ❖ Наибольшая возрастающая подпоследовательность

Как упоминалось в разделе 3.1, простой полный перебор, который строит все возможные подпоследовательности, чтобы найти самую длинную возрастающую последовательность, работает слишком медленно, поскольку существует $O(2^n)$ возможных подпоследовательностей. Вместо того чтобы перепробовать все возможные подпоследовательности, мы можем рассмотреть другой подход к решению задачи. Мы можем описать состояние этой задачи с помощью лишь одного параметра: *i*. Пусть `LIS(i)` будет максимальная возрастающая подпоследовательность, последним элементом которой будет элемент с индексом `i`. Мы знаем, что `LIS(0) = 1`, поскольку первое число в последовательности `A` само является подпоследовательностью. Для `i ≥ 1` вычисление `LIS(i)` немного сложнее. Нам нужно найти индекс `j` такой, что `j < i` и `A[j] < A[i]`, а `LIS(j)` является наибольшим. Найдя этот индекс `j`, мы узнаем, что `LIS(i) = 1 + LIS(j)`. Мы можем записать это формально следующим образом:

1) `LIS(0) = 1` // начальный случай;
2) `LIS(i) = max (LIS(j) + 1), ∀j ∈ [0..i-1]` и `A[j] <A[i]` // рекурсивный случай, на один больше предыдущего лучшего решения, заканчивающегося на значении индекса `j`, для всех `j <i`.

Ответом является наибольшее значение `LIS(k)` `∀k ∈ [0..n-1]`.
Теперь давайте посмотрим, как работает этот алгоритм (также см. рис. 3.10):

- `LIS(0)` равно 1, первое число в `A` = {−7}, начальный вариант.
- `LIS(1)` равно 2, поскольку мы можем расширить подпоследовательность `LIS(0)` = {−7}, добавив в нее следующий элемент {10}, чтобы сформировать возрастающую подпоследовательность {−7, 10} длины 2.
  Наилучшим значением `j` для `i = 1` является `j = 0`.
- `LIS(2)` равно 2, так как мы можем расширить подпоследовательность `LIS(0)` = {−7}, добавив в нее следующий элемент {9}, чтобы сформировать возрастающую подпоследовательность {−7, 9} длины 2.
  Мы не можем далее расширить подпоследовательность `LIS(1)` = {−7, 10}, добавив в нее элемент {9}, так как полученная при этом подпоследовательность не будет являться возрастающей.
  Наилучшим `j` для `i = 2` является `j = 0`.

последовательностей можно смоделировать как направленный ациклический граф (Directed Acyclic Graph, DAG), и нахождение LIS эквивалентно поиску самых длинных путей в направленном ациклическом графе (см. раздел 4.7.1).

- LIS(3) равно 2, так как мы можем расширить подпоследовательность LIS(0) = {–7}, добавив в нее элемент {2}, чтобы сформировать возрастающую подпоследовательность {–7, 2} длины 2.
  Мы не можем далее расширить подпоследовательность LIS(1) = {–7, 10}, добавив в нее элемент {2}, так как полученная при этом подпоследовательность не будет являться возрастающей.
  Мы также не можем расширить подпоследовательность LIS(2) = {–7, 9}, добавив в нее элемент {2}, так как полученная при этом подпоследовательность не будет являться возрастающей.
  Наилучшим j для i = 3 является j = 0.
- LIS(4) равно 3, так как мы можем расширить подпоследовательность LIS(3) = {–7, 2}, добавив в нее элемент {3}, чтобы сформировать возрастающую подпоследовательность {–7, 2, 3}.
  Это лучший выбор из всех возможных.
  Наилучшим j для i = 4 является j = 3.
- LIS(5) равно 4, так как мы можем расширить подпоследовательность LIS(4) = {–7, 2, 3}, добавив в нее элемент {8}, чтобы сформировать возрастающую подпоследовательность {–7, 2, 3, 8}.
  Это лучший выбор из всех возможных.
  Наилучшим j для i = 5 является j = 4.
- LIS(6) равно 4, так как мы можем расширить подпоследовательность LIS(4) = {–7, 2, 3}, добавив в нее элемент {8}, чтобы сформировать возрастающую подпоследовательность {–7, 2, 3, 8}.
  Это лучший выбор из всех возможных.
  Наилучшим j для i = 6 является j = 4.
- LIS(7) равно 2, так как мы можем расширить подпоследовательность LIS(0) = {–7}, добавив в нее элемент {1}, чтобы сформировать возрастающую подпоследовательность {–7, 1}.
  Это лучший выбор среди возможных.
  Наилучшим j для i = 7 является j = 0.
- Ответами на вопрос задачи являются LIS(5) или LIS(6); для обоих этих случаев значения (длины LIS) равны 4.
  Обратите внимание, что индекс k, для которого LIS(k) является самым высоким, может находиться где угодно в интервале [0..n-1].

Очевидно, что в задаче нахождения наибольшей возрастающей подпоследовательности существует много перекрывающихся подзадач, поскольку для вычисления LIS(i) нам нужно вычислить LIS(j) ∀j ∈ [0..i-1]. Однако есть только n различных состояний; индексы LIS заканчиваются на индексе i, ∀i ∈ [0..n-1]. Поскольку нам нужно вычислить каждое состояние с помощью цикла с временной сложностью $O(n)$, этот алгоритм DP выполняется за $O(n^2)$.

При необходимости решение (решения) задачи НВП можно восстановить, сохранив информацию о предыдущем элементе (стрелки на рис. 3.10) и проследив за стрелками от индекса k, которые содержат наибольшее значение LIS(k). Например, LIS(5) является оптимальным конечным состоянием. Внимательно рассмотрите рис. 3.10. Мы можем проследить за стрелками следующим образом: LIS(5) → LIS(4) → LIS(3) → LIS(0), поэтому оптимальным решением (мы

читаем индексы в обратном направлении) является последовательность, индексы элементов которой равны {0, 3, 4, 5}, или последовательность {–7, 2, 3, 8}.

Задача нахождения наибольшей возрастающей подпоследовательности также может быть решена с использованием комбинации чувствительного к выходным данным «жадного» алгоритма с временной сложностью $O(n \log k)$ и подхода «разделяй и властвуй» (D&C) (где k – длина LIS) вместо $O(n^2)$, работая с массивом, который *всегда сортируется*, и, следовательно, для него можно использовать двоичный поиск. Пусть массив `L` будет таким массивом, что `L(i)` представляет наименьшее конечное значение из всех LIS длины – `i`, найденных до сих пор. Хотя это определение немного сложнее, легко увидеть, что оно всегда упорядочено – `L(i-1)` всегда будет меньше, чем `L(i)`, так как предпоследний элемент любой LIS (длины – `i`) меньше, чем его последний элемент. Таким образом, мы можем использовать массив двоичного поиска `L`, чтобы определить самую длинную возможную подпоследовательность, которую мы можем создать, добавив текущий элемент `A[i]`, – просто найдем индекс последнего элемента в `L`, который будет меньше, чем `A[i]`. Используя тот же пример, мы будем обновлять массив `L` шаг за шагом, используя следующий алгоритм:

- первоначально, при `A[0] = -7`, мы имеем `L = {-7}`;
- мы можем добавить `A[1] = 10` к `L[1]`, тогда у нас появляется LIS длины 2, `L = {-7, 10}`;
- для `A[2] = 9` мы заменим `L[1]`, чтобы у нас было «лучшее» окончание LIS длины 2: `L = {-7, 9}`.
  Это «жадный» алгоритм. Сохраняя LIS с меньшим конечным значением, мы максимально расширяем наши возможности для дальнейшего расширения LIS будущими значениями из начального массива;
- для `A[3] = 2` мы заменим `L[1]`, чтобы получить «еще лучшее» окончание LIS длины 2: `L = {-7, 2}`;
- мы добавляем `A[4] = 3` в `L[2]`, чтобы у нас была более длинная LIS, `L = {-7, 2, 3}`;
- мы добавляем `A[5] = 8` в `L[3]`, чтобы у нас была более длинная LIS, `L = {-7, 2, 3, 8}`;
- для `A[6] = 8` ничего не изменяется, так как `L[3] = 8`.
  `L = {-7, 2, 3, 8}` остается без изменений;
- для `A[7] = 1` мы улучшаем `L[1]`, получая в результате `L = {-7, 1, 3, 8}`.
  Это показывает, что массив `L` не является LIS для `A`. Данный шаг важен, так как в будущем могут быть более длинные подпоследовательности, которые могут расширить подпоследовательность длины –2 при `L[1] = 1`. Например, рассмотрим следующий контрольный пример: `A = {-7, 10, 9, 2, 3, 8, 8, 1, 2, 3, 4}`. Длина LIS для этого теста составляет 5;
- ответ задачи – наибольшая длина отсортированного массива L в конце процесса.

Файл исходного кода: *ch3_06_LIS.cpp/java*

---

4. Задача о рюкзаке (Рюкзак 0–1) (сумма подмножества)

Задача[1]: пусть дано n предметов, каждый из которых имеет ценность Vi и вес Wi, $\forall i \in [0..n-1]$, и максимальный размер рюкзака S; вычислим максимальную ценность предметов, которые мы можем нести в рюкзаке, если мы можем либо[2] взять конкретный предмет, либо не брать его (отсюда и термин 0–1 для решения «не брать»/«взять»).

Пример: $n = 4$, $V = \{100, 70, 50, 10\}$, $W = \{10, 4, 6, 12\}$, $S = 12$.

Если мы выберем предмет 0 с весом 10 и ценностью 100, мы не сможем взять другой предмет. Это не оптимальное решение.

Если мы выберем предмет 3 с весом 12 и ценностью 10, мы не сможем взять другой предмет. Это не оптимальное решение.

Если мы выберем предметы 1 и 2, у этих предметов будет общий вес 10 и общая ценность 120. Это максимум.

Решение: используйте эти повторяющиеся операции полного перебора `val(id, remW)`, где `id` – это индекс текущего элемента, который нужно рассмотреть, а `remW` – вес, оставшийся в рюкзаке после размещения в нем выбранных предметов:

1) `val(id, 0) = 0` // если `remW = 0`, мы больше ничего не можем взять;
2) `val(n, remW) = 0` // если `id = n`, мы рассмотрели все элементы;
3) если `W[id]> remW`, у нас нет другого выбора, кроме как не брать этот предмет

   ```
   val(id, remW) = val(id + 1, remW);
   ```

4) если `W[id] ≤ remW` , у нас есть два варианта: не брать или взять этот предмет; мы берем максимум

   ```
   val(id, remW) = max(val(id + 1, remW), V[id] + val(id + 1, remW - W[id])).
   ```

Ответ можно найти, взяв `value(0, S)`. Обратите внимание на перекрывающиеся подзадачи в этой задаче Рюкзак 0–1. Пример: после того как мы взяли предмет 0 и не взяли предметы 1–2, мы достигаем состояния `(3, 2)` – третьего предмета (`id = 3`) с двумя оставшимися единицами веса (`remW = 2`). Если мы не берем предмет 0 и берем предметы 1–2, мы также достигаем того же состояния `(3, 2)`. Хотя есть перекрывающиеся подзадачи, существует только $O(nS)$ возможных различных состояний (поскольку `id` может варьироваться между `[0..n-1]` и `remW` может варьироваться между `[0..S]`). Мы можем вычислить каждое из этих состояний за время $O(1)$, таким образом, общая временная сложность[3] этого решения DP составляет $O(nS)$.

Примечание. Нисходящее решение DP для данной задачи часто работает быстрее, чем восходящее. Это связано с тем, что не все состояния мы фактиче-

---

[1] Эту задачу также называют задачей о сумме подмножеств. Формулировка задачи о сумме подмножеств: если задан набор целых чисел и целое число S, существует ли (непустое) подмножество с суммой, равной S?

[2] Есть и другие разновидности этой задачи, например задача о рюкзаке с использованием дробных значений, для решения которой существует «жадный» алгоритм.

[3] Если значение S так велико, что $NS >> 1M$, решение этой задачи методом динамического программирования невозможно, даже с использованием приемов экономии места!

ски проходим, и, следовательно, задействованные критические состояния DP на самом деле являются лишь (очень маленьким) подмножеством всего пространства состояний. Помните: нисходящее решение DP проходит только *требуемые* состояния, тогда как восходящее решение DP проходит *все различные состояния*.

Обе версии представлены в нашей библиотеке исходного кода.

Файл исходного кода: *ch3_07_UVa10130.cpp/java*

5. Размен монет (Coin Change, CC) – общий случай

Задача: пусть имеется некоторое число центов V и список номиналов для n монет, т. е. у нас есть `coinValue[i]` (в центах) для типов монет `i` ∈ `[0..n-1]`. Какое минимальное количество монет мы должны использовать для набора суммы V? Предположим, что у нас неограниченный запас монет любого типа (см. также раздел 3.4.1).

Пример 1: $V = 10$, $n = 2$, `coinValue` = {1, 5}.

Мы можем использовать:

A) десять одноцентовых монет = 10 × 1 = 10. Общее количество использованных монет = 10;
B) одну монету достоинством 5 центов + пять монет достоинством 1 цент = 1 × 5 + 5 × 1 = 10. Общее количество использованных монет = 6;
C) две монеты достоинством 5 центов = 2 × 5 = 10. Общее количество использованных монет = 2 → оптимальное решение.

Мы можем использовать «жадный» алгоритм, если номиналы монет подходят (см. раздел 3.4.1).

Решение для приведенного выше примера 1 можно получить, используя «жадный» алгоритм. Однако для общих случаев мы должны использовать DP (см. пример 2 ниже).

Пример 2: $V = 7$, $n = 4$, `coinValue` = {1, 3, 4, 5}.

Применение «жадного» алгоритма даст в результате решение, использующее 3 монеты: 5 + 1 + 1 = 7, но на самом деле оптимальное решение – 2 монеты (сумма $V = 7$ набирается из монет достоинством 4 + 3).

Решение: используйте рекуррентные соотношения при полном переборе для `change(value)`, где `value` – это оставшееся количество центов, которое мы должны набрать из монет указанного достоинства:

1) `change(0)` = 0 // нам нужно 0 монет для получения 0 центов;
2) `change(<0)` = ∞ // на практике мы можем вернуть большое положительное значение;
3) `change(value)` = `1 + min(change(value - coinValue[i]))` ∀`i` ∈ `[0..n-1]`.

Ответом является возвращаемое значение `change(V)`.

По рис. 3.10 видно, что:

- `change(0)` = 0 и `change (<0)` = ∞: это базовые (граничные) случаи;
- `change(1)` = 1, из `1 + change(1-1)`, поскольку для `1 + change(1-5)` не существует ответа (возвращает ∞);

- change(2) = 2, из 1 + change(2-1), так как для 1 + change(2-5) также не существует ответа (возвращает ∞);
- ...то же самое для change(3) и change(4);
- change(5) = 1, из 1 + change(5-5) = 1 монета, меньше чем 1 + change(5-1) = 5 монет;
- ...и так до change(10).

<0	0	1	2	3	4	5	6	7	8	9	10
∞	0	1	2	3	4	1	2	3	4	5	2

V = 10, N = 2, coinValue= {1, 5}

Рис. 3.10 ❖ Размен монет

Ответом является возвращаемое значение change(V), для данного примера – change(10) = 2.

Мы можем видеть, что в этой задаче об изменении монеты есть много пересекающихся подзадач (например, для определения change (10) и change (6) требуется значение change (5)). Однако в задаче существует только $O(V)$ возможных различных состояний (так как значение может варьироваться в пределах [0..V]). Поскольку нам нужно попробовать n типов монет для каждого состояния, общая временная сложность этого решения DP составляет $O(nV)$.

Один из вариантов этой задачи состоит в подсчете количества возможных (канонических) способов получения суммы V центов с использованием списка номиналов монет n. Для примера 1, приведенного выше, ответ 3: {1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, 5 + 1 + 1 + 1 + 1 + 1, 5 + 5}.

Решение: используйте рекуррентные соотношения при полном переборе для ways(type, value), где value – то же, что и в решении выше, но теперь у нас есть еще один параметр для индекса типа монеты, который мы сейчас рассматриваем. Этот второй параметр type важен, так как данное решение рассматривает типы монет последовательно. Как только мы решим не брать определенный тип монет, мы не должны рассматривать его снова, чтобы избежать двойного счета:

1) ways(type, 0) = 1 // один способ, ничего не использовать;
2) ways(type, <0) = 0 // решения нет, мы не можем набрать отрицательное значение заданной суммы;
3) ways(n, value) = 0 // решения нет, мы рассмотрели все типы монет ∈ [0..n-1];
4) ways(type, value) = ways(type + 1, value) + // если мы не используем этот тип монеты,
   ways(type, value - coinValue[type]) // плюс, если мы используем этот тип монеты.

Есть только $O(nV)$ возможных различных состояний. Поскольку каждое состояние может быть вычислено за $O(1)$, общая временная сложность[1]

[1] Если V так велико, что $nV >> 1M$, решение этой задачи методом динамического программирования невозможно, даже с использованием приемов экономии места!

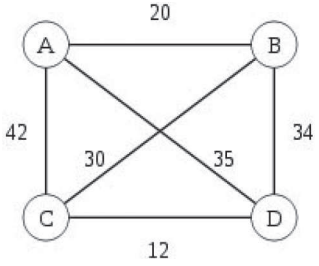
этого решения DP составляет $O(nV)$. Ответом для задачи является значение `ways(0, V)`. Примечание: если значения монет не изменились, есть много запросов с разными V, то мы можем *не* очищать таблицу memo. Поэтому мы запускаем этот алгоритм с временной сложностью $O(nV)$ один раз и просто выполняем поиск $O(1)$ для последующих запросов.

Файл исходного кода (для этого варианта задачи о размене монет):
ch3_08_UVa674.cpp/java

6. Задача о коммивояжере (Traveling Salesman Problem, TSP)

Задача: пусть дано n городов и попарные расстояния между ними в виде матрицы `dist` с размерностью $n \times n$, вычислите стоимость путешествия по маршруту[1], который начинается в любом городе s, проходит через все остальные $n - 1$ городов (каждый город посещается ровно один раз) и, наконец, возвращается обратно в город s, с которого началось путешествие.

Пример: в графе, показанном на рис. 3.11, имеется $n = 4$ города. Поэтому у нас 4! = 24 возможных маршрута (число перестановок множества из 4 городов). Один из кратчайших маршрутов – `A-B-C-D-A`, стоимостью 20 + 30 + 12 + 35 = 97 (обратите внимание, что у задачи может быть более одного оптимального решения).



	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

Рис. 3.11 ❖ Полный граф

Решение задачи TSP методом «грубой силы» (итеративное или рекурсивное), которое перебирает все $O((n - 1)!)$ возможных маршрутов (располагаем первый город в вершине A, чтобы воспользоваться симметрией), эффективно только тогда, когда значение n не превосходит 12, так как 11! ≈ 40 млн. Когда $n > 12$, решение методом «грубой силы» на олимпиаде по программированию получит вердикт жюри «превышение лимита времени» (TLE). Однако если мы имеем несколько тестовых примеров, ограничение для решения задачи о коммивояжере методом «грубой силы», вероятно, лишь $n \leqslant 11$.

Мы можем решать задачу о коммивояжере с использованием динамического программирования, так как расчеты подмаршрутов явно перекрываются:

[1] Такое путешествие называется гамильтоновым маршрутом, представляющим собой цикл в неориентированном графе, который посещает каждую вершину ровно один раз, а также возвращается в начальную вершину.

например, маршрут $A - B - C - (n - 3)$ других городов, который в конечном итоге возвращается в A, явно перекрывается с маршрутом $A - C - B -$ те же $(n - 3)$ других городов, который также возвращается в A. Если мы сможем избежать повторных вычислений, вычисляя длины таких субмаршрутов, мы тем самым сэкономим много времени. Тем не менее отдельное состояние в задаче о коммивояжере зависит от двух параметров: последнего города, который посетил коммивояжер / вершины графа `pos`, а также от того, что мы, возможно, не встречали ранее при решении задач, – подмножества посещенных городов (вершин).

Есть много способов представить множество. Однако, поскольку мы собираемся передать эту информацию о множестве в качестве параметра рекурсивной функции (если используется нисходящий DP), используемое нами представление должно быть легким и эффективным. В разделе 2.2 мы представили жизнеспособный вариант для этого: битовая маска. Если у нас есть n городов, мы используем двоичное целое число длины n. Если бит i равен «1» (включен), мы говорим, что элемент (город) i принадлежит множеству (он был посещен); и, соответственно, элемент i не принадлежит множеству (и не был посещен), если соответствующий бит установлен в «0» (выкл.) Например: `mask` = 18_{10} = 10010_2 подразумевает, что элементы (города) {1, 4} принадлежат[1] множеству (и были посещены). Напомним, что для проверки, включен ли бит i, мы можем использовать выражение `mask & (1 << i)`. Чтобы установить бит i, мы можем использовать выражение `mask |= (1 << i)`.

Решение: используйте рекуррентные соотношения при полном переборе для `tsp (pos, mask)`:

1) `tsp(pos, 2^n - 1) = dist[pos][0]` // все города посещены, возврат в начальный город // *Примечание*: `mask = (1 << n) - 1` или $2^n - 1$ подразумевает, что все n бит в `mask` включены;
2) `tsp (pos, mask) = min (dist[pos][nxt] + tsp (nxt, mask | (1 << nxt)))` // ∀ `nxt` ∈ `[0..n-1]`, `nxt! = pos`, и `(mask & (1 << nxt))` равно '0' (соответствующий бит «выключен») // Мы перебираем все возможные оставшиеся города, которые раньше не посещались, на каждом шаге.

Есть только $O(n \times 2^n)$ различных состояний, потому что есть n городов, и мы помним до 2^n других городов, которые посетили на каждом маршруте. Каждое состояние может быть вычислено в $O(n)$, лучше таким образом, чтобы общая временная сложность этого решения DP составляла $O(2^n \times n^2)$. Это позволяет нам решать данную задачу при значениях[2] $n \approx 16$ (так как $162 \times 216 \approx 17M$). Это не является значительным улучшением по сравнению с решением методом «грубой силы»; но если задача о коммивояжере, предложенная на олимпиаде по программированию, имеет объем входных данных $11 \leqslant n \leqslant 16$, тогда следует

---

[1] Помните, что для `mask` индексы начинаются с 0 и отсчитываются справа.

[2] Поскольку задачи по программированию обычно требуют точных решений, представленное здесь решение задачи о коммивояжере с использованием динамического программирования уже является одним из лучших решений. В реальной жизни эту задачу часто приходится решать для тысяч городов. Чтобы решить более масштабные задачи такого типа, у нас есть неточные подходы, подобные представленным в [26].

использовать приемы DP, а не метод «грубой силы». Ответом для задачи является значение `tsp(0, 1)`: мы начинаем с города 0 (мы можем начать с любой вершины; но самый простой вариант – выбрать вершину 0) и устанавливаем для нее `mask = 1`, чтобы никогда повторно не посещать город 0.

Обычно при решении задачи о коммивояжере с использованием DP на олимпиадах по программированию требуется некоторая предварительная обработка графов для генерации матрицы расстояний `dist` перед запуском решения DP. Эти варианты решений обсуждаются в разделе 8.4.3.

Решения DP, которые включают (небольшой) набор логических значений в качестве одного из параметров, более известны как DP с использованием битовой маски. Более сложные проблемы динамического программирования, связанные с этим способом, обсуждаются в разделах 8.3 и 9.2.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/rectree.html

Файл исходного кода: *ch3_09_UVa10496.cpp/java*

Упражнение 3.5.2.1. Решение задачи о максимальной сумме 2D-диапазона (Max 2D Range Sum) имеет временную сложность $O(n^4)$. На самом деле существует решение $O(n^3)$, которое объединяет решение DP для задачи о максимальной сумме 1D-диапазона (Max Range 1D Sum) и использует ту же идею, которая была предложена в алгоритме Кадана, для другого измерения. Решите задачу UVa 108, используя решение с временной сложностью $O(n^3)$.

Упражнение 3.5.2.2. Решение для запроса минимальной суммы диапазона `Range Minimum Query(i, j)` для одномерных массивов, обсуждаемое в разделе 2.4.3, использует дерево сегментов. Это излишне, если данный массив является статическим и неизменным во всех запросах. Используйте метод DP для поиска ответа на запрос `RMQ(i, j)`, работающего с быстротой $O(n \log n)$ при предварительной обработке и $O(1)$ при выдаче ответа на запрос.

Упражнение 3.5.2.3. Решите задачу о нахождении наибольшей возрастающей подпоследовательности (LIS), используя решение, работающее с быстротой $O(n \log k)$, а *также* восстановите одну из LIS.

Упражнение 3.5.2.4. Можем ли мы использовать метод итеративного полного перебора, который перебирает все возможные подмножества n элементов, как обсуждалось в разделе 3.2.1, для решения задачи о рюкзаке (Рюкзак 0–1)? Каковы ограничения этого решения, если таковые имеются?

Упражнение 3.5.2.5*. Предположим, мы добавили еще один параметр к классической задаче о рюкзаке (Рюкзак 0–1). Пусть K_i обозначает количество экземпляров предмета i для использования в задаче. Пример: $n = 2$, $V = \{100, 70\}$, $W = \{5, 4\}$, $K = \{2, 3\}$, $S = 17$ означает, что есть два экземпляра предмета 0 с весом 5 и ценностью 100 и есть три экземпляра предмета 1 с весом 4 и ценностью 70. Оптимальное решение для этого примера – взять один из предметов 0 и три

предмета 1 с общим весом 17 и общей ценностью 310. Решить новый вариант задачи, предполагая что $1 \leqslant n \leqslant 500$, $1 \leqslant S \leqslant 2000$, $n \leqslant \sum_i^{n-1} = 0\ K_i \leqslant 40\,000$. *Подсказка*: каждое целое число может быть записано как сумма степеней 2.

Упражнение 3.5.2.6*. Решение задачи о коммивояжере с использованием динамического программирования, показанное в этом разделе, все еще может быть немного улучшено, чтобы сделать возможным решение задачи при ограничении $n = 17$ на олимпиаде по программированию. Покажите, какие незначительные изменения необходимо выполнить, чтобы это стало возможным. Подсказка: учтите симметрию.

Упражнение 3.5.2.7*. В дополнение к незначительному изменению, о котором шла речь в упражнении 3.5.2.5*, какие еще изменения необходимы в решении задачи о коммивояжере с использованием динамического программирования, чтобы было возможно обрабатывать набор входных данных с $n = 18$ (или даже $n = 19$), но с гораздо меньшим количеством тестовых примеров)?

3.5.3. Неклассические примеры

Хотя задачи, решаемые с использованием динамического программирования, – наиболее популярная разновидность задач, чаще всего встречавшаяся на недавних олимпиадах по программированию, классические задачи, предполагающие использование динамического программирования *в чистом виде*, обычно никогда не появляются в числе задач, предлагаемых для решения на олимпиадах ICPC или IOI. Мы изучаем их, чтобы понять, что такое динамическое программирование, но мы должны научиться решать многие другие неклассические задачи, решаемые этим способом (которые могут стать классическими в ближайшем будущем), и развивать наши «навыки DP» в процессе их решения. В этом подразделе мы обсудим еще два неклассических примера в добавление к задаче UVa 11450 – Wedding Shopping, которую мы подробно обсудили ранее. Мы также выбрали несколько простых неклассических задач DP в качестве упражнений. После того как вы решите большинство из этих задач, можете взяться за более сложные в других разделах этой книги, например в разделах 4.7.1, 5.4, 5.6, 6.5, 8.3, 9.2, 9.21 и т. д.

1. UVa 10943 – How do you add?

Сокращенная формулировка условий задачи: для заданного целого числа n сколькими способами можно дополнить K неотрицательных целых чисел, меньших или равных n, до n? *Ограничения:* $1 \leqslant n, K \leqslant 100$. *Пример:* для $n = 20$ и $K = 2$ существует 21 способ: 0 + 20, 1 + 19, 2 + 18, 3 + 17, ..., 20 + 0.

Математически число способов может быть выражено как ${}^{(n+k-1)}C_{(k-1)}$ (см. раздел 5.4.2, где говорится о биномиальных коэффициентах). Мы будем использовать эту простую задачу, чтобы еще раз проиллюстрировать принципы динамического программирования, которые мы обсуждали в данном разделе, особенно процесс получения соответствующих состояний для задачи и получения правильных переходов из одного состояния в другое с учетом граничных случаев.

Во-первых, мы должны определить параметры этой задачи, которые необходимо выбрать для представления различных состояний. В этой задаче есть только два параметра, n и K.

Следовательно, есть только четыре возможные комбинации:

1) если мы не выберем ни один из этих параметров, то не сможем представить состояние. Этот вариант мы отбрасываем;
2) если мы выбираем только *n, то не знаем, сколько чисел* $\leqslant n$ было использовано;
3) если мы выберем только K, то мы не знаем целевой суммы n;
4) поэтому состояние данной задачи должно быть представлено парой (или кортежем) (n, K).
   Порядок выбранных параметров не имеет значения, то есть пара (K, n) также подходит.

Далее мы должны определить граничный (базовый) случай (или случаи). Оказывается, эта задача очень проста, когда $K = 1$. Каким бы ни было n, есть только один способ добавить ровно одно число, меньшее или равное n, чтобы получить n: использовать само n. Для этой задачи нет другого основного случая.

Для общего случая мы имеем следующую рекурсивную формулировку, которую не так сложно вывести: в состоянии (n, K), где $K > 1$, мы можем разбить n на одно число $X \in$ `[0..n]` и $n - X$, то есть $n = X + (n - X)$. Проделав это, мы приходим к подзадаче $(n - X, K - 1)$, т. е. если задано число $n - X$, сколько существует способов дополнить $K - 1$ чисел, меньших или равных $n - X$, до числа $n - X$? Затем мы можем суммировать все эти способы.

Эти идеи можно записать в виде следующих способов повторения полного поиска (n, K):

1) `ways(n, 1)` = 1 // мы можем использовать только 1 число, чтобы добавить n – само число n;
2) `ways(n, K)` = $\sum_{X=0}^{n}$ `ways(n - X, K - 1)` // рекурсивно суммируем все возможные способы.

В этой задаче есть перекрывающиеся подзадачи. Например, контрольный пример $n = 1, K = 3$ имеет перекрывающиеся подзадачи: состояние $(n = 0, K = 1)$ достигается дважды (см. рис. 4.39 в разделе 4.7.1). Однако существует только $n \times K$ возможных состояний (n, K). Стоимость вычисления каждого состояния – $O(n)$. Таким образом, общая временная сложность составляет $O(n^2 \times K)$. Поскольку $1 \leqslant n, K \leqslant 100$, подобный вариант решения подходит. Ответом является возвращаемое значение `ways(n, K)`.

Обратите внимание, что для этой задачи на самом деле нужен только результат по модулю $1M$ (то есть последние шесть цифр ответа). См. раздел 5.5.8, где обсуждаются операции арифметики по модулю.

Файл исходного кода: *ch3_10_UVa10943.cpp/java*

2. UVa 10003 – Cutting Sticks

Сокращенная формулировка условий задачи: брусок длиной $1 \leqslant l \leqslant 1000$ необходимо разрезать на несколько частей $1 \leqslant n \leqslant 50$ (даны координаты разрезов,

лежащие в диапазоне [0..*l*]). Стоимость разреза определяется длиной нарезаемого бруска. Ваша задача – найти такой вариант резки, чтобы общие затраты на разрезание бруска на части были минимальными.

Пример: *l* = 100, *n* = 3 и координаты разрезов: A = {25, 50, 75} (значения уже отсортированы).

Если мы начнем резать палку слева направо, то понесем затраты = 225:

1) первый разрез – в точке с координатой 25, общие затраты на данный момент = 100;
2) второй разрез – в точке с координатой 50, общие затраты на данный момент = 100 + 75 = 175;
3) третий разрез – в точке с координатой 75, окончательные общие затраты = 175 + 50 = 225.

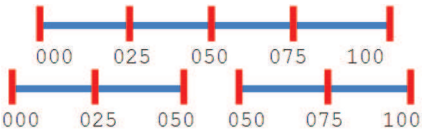


Рис. 3.12 ❖ Иллюстрация разрезания бруска

Тем не менее ответ для оптимального случая – 200:

1) первый разрез – в точке с координатой 50, общие затраты на данный момент = 100 (этот разрез показан на рис. 3.12);
2) второй разрез – в точке с координатой 25, общие затраты на данный момент = 100 + 50 = 150;
3) третий разрез – в точке с координатой 75, окончательные общие затраты = 150 + 50 = 200.

Как мы решаем эту задачу? Сначала рассмотрим подход с использованием алгоритма полного перебора.

Попробуйте все возможные точки разреза. Перед этим мы должны выбрать подходящее определение состояния для задачи: (промежуточные) бруски. Мы можем описать брусок с двумя конечными точками: `left` (левой) и `right` (правой). Однако эти два значения могут быть очень большими, что может впоследствии усложнить решение, когда мы захотим запомнить их значения.

Мы можем воспользоваться тем фактом, что есть только *n* + 1 меньших брусков после разрезания оригинального бруска *n* раз. Конечные точки каждого меньшего бруска могут быть описаны начальной точкой 0, координатами точки разрезания и *l*.

Поэтому мы добавим еще две координаты: `A` = {0, исходное `A` и *l*}, чтобы мы могли обозначать брусок индексами его конечных точек в `A`.

Затем мы можем использовать повторяющиеся операции `cut(left, right)`, где `left/right` – это левый/правый индексы бруска, имеющегося в данный момент, относительно `A`. Первоначально брусок имеет индексы `left` = 0 и `right` = *n* + 1, то есть мы имеем брусок с длиной [0..*l*]:

1) `cut(i-1, i) = 0, ∀i ∈ [1..n+1]` // если `left + 1 = right`, где `left` и `right` – индексы в `A`, то у нас есть сегмент бруска, который не нужно разрезать дальше;

2) `cut(left, right) = min(cut(left, i) + cut(i, right) + (A[right]-A[left])) ∀i ∈ [left+1..right-1]` // пробуем все возможные точки разреза и выбираем наилучший вариант.
   Стоимость одного разреза – это длина бруска в настоящий момент, записанная в виде `(A[right] - A[left])`.
   Ответом является возвращаемое значение `cut(0, n+1)`.

Теперь давайте проанализируем временную сложность этой задачи. Первоначально у нас есть n вариантов для точек разрезания бруска.

Как только мы режем брусок в определенной точке, у нас остается $n - 1$ вариантов выбора второй точки. Это повторяется, пока у нас не останется ноль точек для резания бруска. Перебор всех возможных точек разреза, таким образом, приводит к алгоритму с временной сложностью $O(n!)$. Это неприемлемо для $1 \leqslant n \leqslant 50$.

Однако для этой задачи существуют перекрывающиеся подзадачи. Например, на рис. 3.13 выше операция разрезания бруска с индексом 2 (точка разрезания = 50) создает два состояния: `(0, 2)` и `(2, 4)`. Такое же состояние `(2, 4)` также может быть достигнуто путем операции разрезания с индексом 1 (точка разрезания = 25), а затем операции разрезания с индексом 2 (точка разрезания = 50). Таким образом, пространство поиска для данной задачи на самом деле не так велико. Существуют только $(n + 2) \times (n + 2)$ возможных левых/правых индексов, или $O(n^2)$ различных состояний, которые необходимо запомнить.

Время, необходимое для вычисления одного состояния, составляет $O(n)$. Таким образом, общая временная сложность (нисходящего DP) составляет $O(n^3)$. При $n \leqslant 50$ это приемлемое решение.

Файл исходного кода: *ch3_11_UVa10003.cpp/java*

Упражнение 3.5.3.1*. Почти весь исходный код, показанный в этом разделе (нахождение наибольшей возрастающей подпоследовательности, размен монет, задача о коммивояжере и UVa 10003 – Cutting Sticks (Разрезание бруска)), написан с использованием нисходящего DP, в соответствии с предпочтениями авторов этой книги. Перепишите исходный код решения этих задач, используя вариант восходящего DP.

Упражнение 3.5.3.2*. Найдите решение задачи о разрезании бруска с временной сложностью $O(n^2)$. Подсказка: используйте ускорение при помощи алгоритма Кнута–Яо, применяя следующее свойство: каждый повтор в рекурсии удовлетворяет неравенству четырехугольника (см. [2]).

Замечания о задачах на динамическое программирование на олимпиадах по программированию

Основные (и в т. ч. «жадные» алгоритмы) способы решения задач с использованием динамического программирования подробно освещаются в популярных учебниках по алгоритмам, например Introduction to Algorithms («Введение

в алгоритмы») [7], Algorithm Design («Разработка алгоритмов») [38] и Algorithm («Алгоритмы») [8].

В этом разделе мы обсудили шесть классических задач, решаемых с помощью динамического программирования, и их решения. Краткое резюме приведено в табл. 3.4. Это классические задачи DP; если они появятся на проводимых в будущем олимпиадах по программированию, то, скорее всего, только как часть более серьезных и сложных задач.

Таблица 3.4. Перечень классических задач динамического программирования в этом разделе

	1D RSQ	**2D RSQ**	**НВП**	**Рюкзак**	**Размен монет**	**Задача о коммивояжере**
Состояние	`(i)`	`(i,j)`	`(i)`	`(id, remW)`	`(v)`	`(pos, mask)`
Пространственная сложность	$O(n)$	$O(n^2)$	$O(n)$	$O(nS)$	$O(V)$	$O(n2^n)$
Переходы	подмассив	подматрица	все $j < i$	брать / не брать	все n монет	все n городов
Временная сложность	$O(1)$	$O(1)$	$O(n^2)$	$O(nS)$	$O(nV)$	$O(2^n n^2)$

Чтобы справиться с возрастающей сложностью задач и повышающимися требованиями к креативности при их решении, необходимой для полноценного овладения этими методами (особенно для неклассического DP), мы рекомендуем вам также прочитать учебные пособия по алгоритмам TopCoder [30] и попробовать решить более поздние варианты задач, предлагаемых на конкурсах по программированию.

В этой книге мы снова вернемся к DP в нескольких случаях: алгоритм Флойда–Уоршелла (раздел 4.5), DP на (неявном) направленном ациклическом графе (раздел 4.7.1), выравнивание строк (редактирование расстояния), наибольшая общая подпоследовательность (НОП), другие варианты применения DP в алгоритмах работы со строками (раздел 6.5), более продвинутые случаи использования DP (раздел 8.3) и несколько тем по DP в главе 9.

В прошлом (1990-е годы) участник, хорошо владевший методами DP, мог стать «королем олимпиад по программированию», поскольку задачи DP обычно были чем-то вроде «решающего забега» на соревнованиях. Теперь владение приемами DP является *базовым* требованием! Вы не можете преуспеть на олимпиадах по программированию без этих знаний. Однако мы должны постоянно напоминать читателям данной книги, чтобы они не думали, что знают DP, если они запоминают только решения классических задач DP! Попробуйте овладеть искусством решения задач с использованием DP: научитесь определять состояния (таблица DP), которые могут однозначно и эффективно представлять подзадачи, а также способы заполнения этой таблицы рекурсивным (сверху вниз, или нисходящим способом) или итеративным способом (снизу вверх, или восходящим способом).

Нет лучшего пути освоить эти подходы к решению задач, чем решение реальных задач по программированию! Здесь мы приведем несколько примеров. Как только вы ознакомитесь с примерами, приведенными в этом

разделе, переходите к изучению новых задач динамического программирования, которые начали появляться на недавних олимпиадах по программированию.

Задачи по программированию, решаемые с помощью метода динамического программирования

- Максимальная сумма диапазона 1D (Max 1D Range Sum)
  1. UVa 00507 – Jill Rides Again (стандартная задача)
  2. **UVa 00787 – Maximum Sub… *** (произведение максимального диапазона 1D, будьте осторожны со значением 0, используйте Java BigInteger, см. раздел 5.3)
  3. **UVa 10684 – The Jackpot *** (стандартная задача; легко решаемая с помощью примера исходного кода)
  4. ***UVa 10755 – Garbage Heap **** (комбинация максимальной суммы двумерного диапазона (2D) в двух из трех измерений – см. ниже – и максимальной суммы одномерного диапазона (1D) с использованием алгоритма Кадана для третьего измерения)
     Больше примеров см. в разделе 8.4.
- Максимальная сумма диапазона 2D (Max 2D Range Sum)
  1. **UVa 00108 – Maximum Sum *** (обсуждается в этом разделе, с примером исходного кода)
  2. UVa 00836 – Largest Submatrix (конвертируйте «0» в -INF)
  3. UVa 00983 – Localized Summing for… (максимальная сумма диапазона 2D, получить подматрицу)
  4. UVa 10074 – Take the Land (стандартная задача)
  5. UVa 10667 – Largest Block (стандартная задача)
  6. **UVa 10827 – Maximum Sum on… *** (скопировать матрицу $n \times n$ в матрицу $n \times 2n$; затем эта задача снова становится стандартной)
  7. ***UVa 11951 – Area **** (используйте long long; максимальная сумма двумерного диапазона; по возможности сокращайте пространство поиска)
- Наибольшая возрастающая подпоследовательность (НВП)
  1. UVa 00111 – History Grading (будьте осторожны с системой ранжирования)
  2. UVa 00231 – Testing the Catcher (решение очевидно)
  3. UVa 00437 – The Tower of Babylon (можно смоделировать как LIS)
  4. **UVa 00481 – What Goes Up? *** (используйте решение LIS с временной сложностью $O(n \log k)$; решение для печати; см. наш пример исходного кода)
  5. UVa 00497 – Strategic Defense Initiative (решение должно быть напечатано)
  6. UVa 01196 – Tiling Up Blocks (LA 2815, Гаосюн'03; отсортируйте все блоки по возрастанию `L[i]`, тогда мы получим классическую задачу LIS)

7. UVa 10131 – Is Bigger Smarter? (сортировка слонов на основе снижения IQ; LIS на увеличение веса)
8. UVa 10534 – Wavio Sequence (необходимо дважды использовать алгоритм LIS с временной сложностью $O(n \log k)$)
9. *UVa 11368 – Nested Dolls* (сортировка в одном измерении, LIS в другом)
10. **UVa 11456 – Trainsorting** * (max(LIS(i) + LDS(i) – 1), ∀i ∈ [0..n – 1])
11. **UVa 11790 – Murcia's Skyline** * (комбинация LIS + LDS, взвешенная)

- Задача о рюкзаке (Рюкзак 0–1) (сумма подмножества)

1. UVa 00562 – Dividing Coins (используйте одномерную таблицу)
2. UVa 00990 – Diving For Gold (необходимо распечатать решение)
3. UVa 01213 – Sum of Different Primes (LA 3619, Йокогама'06, дополненная задача о рюкзаке 0–1, используйте три параметра: (id, remN, remK) сверх привычных (id, remN))
4. UVa 10130 – SuperSale (обсуждается в этом разделе с примером исходного кода)
5. UVa 10261 – Ferry Loading (состояние: текущая машина, левая полоса, правая полоса)
6. **UVa 10616 – Divisible Group Sum** * (во входных данных могут оказаться отрицательные числа, примените long long)
7. UVa 10664 – Luggage (сумма подмножества)
8. **UVa 10819 – Trouble of 13-Dots** * (задача о рюкзаке 0–1 с особенностями, присущими кредитным картам)
9. *UVa 11003 – Boxes* (попробуйте все, максимальный вес от 0 до $max(weight[i]+capacity[i])$, $\forall i \in [0..n-1]$; если известен максимальный вес, сколько ящиков может быть уложено?)
10. UVa 11341 – Term Strategy (s: id, h_learned, h_left; t: учить модуль программы «id» в течение 1 часа или же пропустить этот модуль)
11. ***UVa 11566 – Let's Yum Cha*** * (проблема с формулировкой на английском языке, на самом деле просто вариант задачи о рюкзаке: удвойте каждую сумму dim и добавьте один параметр, чтобы проверить, не заказали ли мы слишком много блюд)
12. UVa 11658 – Best Coalition (s: id, share; t: формировать / не формировать альянс с id)

- Размен монет (Coin Change, CC)

1. UVa 00147 – Dollars (решается аналогично UVa 357 и UVa 674)
2. UVa 00166 – Making Change (два варианта рамена монет в одной задаче)
3. **UVa 00357 – Let Me Count The Ways** * (решается аналогично UVa 147/674)
4. UVa 00674 – Coin Change (обсуждается в этом разделе с примером исходного кода)
5. **UVa 10306 – e-Coins** * (вариант: каждая монета состоит из двух компонентов)

6. UVa 10313 – Pay the Price (измененная задача про размен монет + задача о сумме диапазона 1D, решаемая с использованием динамического программирования)
7. UVa 11137 – Ingenuous Cubrency (используйте long long)
8. **UVa 11517 – Exact Change** * (вариант задачи о размене монет)

- Задача о коммивояжере (Traveling Salesman Problem, TSP)
  1. **UVa 00216 – Getting in Line** * (TSP, все еще решается с помощью возвратной рекурсии)
  2. **UVa 10496 – Collecting Beepers** * (обсуждается в этом разделе, приводится пример исходного кода; фактически, поскольку $n \leqslant 11$, эта задача все еще решается с помощью возвратной рекурсии и сокращения набора рассматриваемых вариантов)
  3. **UVa 11284 – Shopping Trip** * (требуется предварительная обработка кратчайших путей; вариант задачи о коммивояжере, где мы можем вернуться домой раньше; нам просто нужно немного изменить повтор решения DP в задаче о коммивояжере: для каждого состояния у нас есть еще один вариант – вернуться домой раньше)
     См. другие примеры в разделах 8.4.3 и 9.2.

- Неклассические задачи (более легкие)
  1. UVa 00116 – Unidirectional TSP (аналог UVa 10337)
  2. *UVa 00196 – Spreadsheet* (обратите внимание, что зависимости ячеек являются ациклическими; поэтому мы можем запоминать прямое (или непрямое) значение каждой ячейки)
  3. *UVa 01261 – String Popping* (LA 4844, Тэджон'10, простая задача, использующая возвратную рекурсию; но мы используем `set<string>` для предотвращения двойной проверки одного и того же состояния (подстроки))
  4. UVa 10003 – Cutting Sticks (обсуждается в этом разделе с примером исходного кода)
  5. UVa 10036 – Divisibility (необходимо использовать метод смещения, поскольку значение может быть отрицательным)
  6. *UVa 10086 – Test the Rods* (s: idx, rem1, rem2; на каком объекте мы находимся сейчас; до 30 объектов; оставшиеся бруски, которые должны быть проверены в NCPC, и оставшиеся бруски, которые должны быть проверены в BCEW; t: для каждого объекта мы разделили количество брусков: x брусков предназначили для испытаний в NCPC и $m[i] - x$ брусков – для испытаний в BCEW; распечатайте решение)
  7. **UVa 10337 – Flight Planner** * (динамическое программирование; задача на нахождение кратчайших путей для направленного ациклического графа)
  8. UVa 10400 – Game ShowMath (решается с помощью возвратной рекурсии и сокращения набора рассматриваемых вариантов)
  9. *UVa 10446 – The Marriage Interview* (немного измените данную рекурсивную функцию, используйте мемоизацию)

10. UVa 10465 – Homer Simpson (одномерная таблица DP)
11. *UVa 10520 – Determine it* (просто напишите данную формулу в виде нисходящего DP с использованием мемоизации)
12. *UVa 10688 – The Poor Giant* (обратите внимание, что образец данных в постановке задачи немного неправильный, он должен быть: 1 + (1 + 3) + (1 + 3) + (1 + 3) = 1 + 4 + 4 + 4 = 13, а не 14, как указано в описании задачи; в противном случае используем простое DP)
13. **UVa 10721 – Bar Codes *** (s: n, k; t: переберите все значения от 1 до *m*)
14. UVa 10910 – Mark's Distribution (двумерная таблица DP)
15. UVa 10912 – Simple Minded Hashing (s: len, last, sum; t: попробуйте следующий символ)
16. **UVa 10943 – How do you add? *** (обсуждается в этом разделе с примером исходного кода; s: n, k; t: попробуйте все возможные точки разбиения; альтернативное решение заключается в использовании математической формулы замкнутой формы: $C(n + k - 1, k - 1)$, которая также решается с помощью динамического программирования, см. раздел 5.4)
17. *UVa 10980 – Lowest Price in Town* (простая задача)
18. *UVa 11026 – A Grouping Problem* (DP, идея, аналогичная биномиальной теореме, см. раздел 5.4)
19. UVa 11407 – Squares (можно использовать мемоизацию)
20. UVa 11420 – Chest of Drawers (s: prev, id, numlck; запереть/отпереть этот сундук)
21. UVa 11450 – Wedding Shopping (подробно обсуждается в этом разделе с примером исходного кода)
22. UVa 11703 – sqrt log sin (можно использовать мемоизацию)

- Другие классические задачи, решаемые с помощью динамического программирования, в этой книге
  1. Алгоритм Флойда–Уоршелла для нахождения кратчайших путей для всех пар вершин (см. раздел 4.5)
  2. Выравнивание строк (редактирование расстояния) (см. раздел 6.5)
  3. Самая длинная общая подпоследовательность (см. раздел 6.5)
  4. Умножение цепочки матриц (см. раздел 9.20)
  5. Максимальное (взвешенное) независимое множество (на дереве, см. раздел 9.22)

- Дополнительные задачи по программированию, решаемые с помощью динамического программирования

  Также см. разделы 4.7.1, 5.4, 5.6, 6.5, 8.3, 8.4 и части главы 9, в которых приведены дополнительные упражнения, связанные с динамическим программированием.

3.6. Решения упражнений, не помеченных звездочкой

Упражнение 3.2.1.1. Решение позволяет избежать оператора деления, чтобы мы работали только с целыми числами!

Если вместо этого мы проведем итерацию по abcde, при вычислении мы можем получить нецелочисленный результат: fghij = abcde / N.

Упражнение 3.2.1.2. Мы также получим AC как 10! ≈ 3 млн, это примерно соответствует алгоритму, приведенному в разделе 3.2.1.

Упражнение 3.2.2.1. Измените функцию возвратной рекурсии, чтобы она была похожа на приведенный ниже код:

```
void backtrack(int c) {
  if (c == 8 && row[b] == a) {                  // решение-кандидат, (a, b) имеется 1 ферзь
    printf("%2d %d", ++lineCounter, row[0] + 1);
    for (int j = 1; j < 8; j++) printf(" %d", row[j] + 1);
    printf("\n"); }
  for (int r = 0; r < 8; r++)                       // попробуйте все возможные горизонтали
    if (col == b && r != a) continue;                              // ДОБАВЬТЕ ЭТУ СТРОЧКУ
    if (place(r, c)) {                   // если вы можете разместить ферзя в этой позиции,
                                                        // на этих столбике и горизонтали
      row[c] = r; backtrack(c + 1);               // поместите этого ферзя здесь и повторите
}   }
```

Упражнение 3.3.1.1. Эта задача может быть решена без использования двоичного поиска. Смоделируйте путешествие один раз. Нам просто нужно найти наибольшую потребность в топливе за всю поездку и сделать топливный бак достаточным для того, чтобы вместить данное количество топлива.

Упражнение 3.5.1.1. Предмет одежды $g = 0$, возьмите третью модель (стоимость 8); предмет одежды $g = 1$, возьмите первую модель (стоимость 10); предмет одежды $g = 2$, возьмите первую модель (стоимость 7); потраченные деньги = 25.

Денег не осталось. Тестовый пример C также решается с помощью «жадного» алгоритма.

Упражнение 3.5.1.2. Нет, эта концепция состояния не работает. Нам нужно знать, сколько денег у нас осталось на каждую подзадачу, чтобы мы могли определить, достаточно ли у нас денег для покупки определенной модели текущего предмета одежды.

Упражнение 3.5.1.3. Измененный код восходящего DP показан ниже:

```
#include <cstdio>
#include <cstring>
using namespace std;

int main() {
  int g, money, k, TC, M, C, cur;
  int price[25][25];
  bool reachable[2][210];           // доступная таблица[ТОЛЬКО ДВЕ СТРОКИ][money (<= 200)]
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (g = 0; g < C; g++) {
      scanf("%d", &price[g][0]);
      for (money = 1; money <= price[g][0]; money++)
```

```
      scanf("%d", &price[g][money]);
    }
    memset(reachable, false, sizeof reachable);
    for (g = 1; g <= price[0][0]; g++)
      if (M - price[0][g] >= 0)
        reachable[0][M - price[0][g]] = true;

    cur = 1;                                                  // мы начинаем с этой строки
    for (g = 1; g < C; g++) {
      memset(reachable[cur], false, sizeof reachable[cur]);          // сброс строки
      for (money = 0; money < M; money++) if (reachable[!cur][money])
        for (k = 1; k <= price[g][0]; k++) if (money - price[g][k] >= 0)
          reachable[cur][money - price[g][k]] = true;
      cur = !cur;                                  // ВАЖНЫЙ ПРИЕМ: переверните две строки
    }

    for (money = 0; money <= M && !reachable[!cur][money]; money++);

    if (money == M + 1) printf("no solution\n");   // в последней строчке не нашлось ответа
    else                printf("%d\n", M - money);
} } // return 0;
```

Упражнение 3.5.2.1. Решение $O(n^3)$ для задачи о максимальной сумме 2D-диапазона (Max 2D Range Sum) показано ниже:

```
scanf("%d", &n);                                  // размерность входной квадратной матрицы
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("%d", &A[i][j]);
  if (j > 0) A[i][j] += A[i][j - 1];               // просто добавьте столбцы этой строки i
}

maxSubRect = -127*100*100;                 // наименьшее возможное значение для этой задачи
for (int l = 0; l < n; l++) for (int r = l; r < n; r++) {
  subRect = 0;
  for (int row = 0; row < n; row++) {
    // максимальная сумма диапазона 1D для столбцов этой строки i
    if (l > 0) subRect += A[row][r] - A[row][l - 1];
    else       subRect += A[row][r];

    // алгоритм Кадана для строк
    if (subRect < 0) subRect = 0;               // "жадный", начните заново, если сумма < 0
    maxSubRect = max(maxSubRect, subRect);
} }
```

Упражнение 3.5.2.2. Решение приведено в разделе 9.33.

Упражнение 3.5.2.3. Решение приведено ранее в `ch3_06_LIS.cpp/java`.

Упражнение 3.5.2.4. Метод итеративного полного перебора для генерации и проверки всех возможных подмножеств размера n выполняется за $O(n \times 2^n)$. Это допустимо для $n \leqslant 20$, но слишком медленно, когда $n > 20$. Решение DP, приведенное в разделе 3.5.2, выполняется за $O(n \times S)$. Если S не так велико, допустимо гораздо большее значение n, чем просто 20 предметов.

3.7. Примечания к главе 3

Многие задачи, предлагаемые на соревнованиях ICPC или IOI, решаются с помощью методов (см. раздел 8.4), использующих комбинации описанных стратегий решения задач. Если бы нам нужно было выбрать только одну главу в этой книге, которую участники олимпиад по программированию должны действительно хорошо освоить, мы бы выбрали эту.

В табл. 3.5 мы сравниваем четыре метода решения задач и их прогнозируемые результаты для различных типов задач. Из табл. 3.5 и списка упражнений в этом разделе вы увидите, что задач, использующих полный перебор и DP, *гораздо больше*, чем задач, использующих стратегию «разделяй и властвуй» (D&C) и «жадные» алгоритмы. Поэтому мы рекомендуем читателям сконцентрироваться на улучшении навыков решения задач с помощью полного перебора и динамического программирования.

Таблица 3.5. Сравнение методов решения задач (только приблизительное, общее правило)

	Задача на полный перебор	**Задачи на «разделяй и властвуй» (D&C)**	**Задачи на «жадные» алгоритмы**	**Задачи на динамическое программирование**
Полный перебор	AC	TLE/AC	TLE/AC	TLE/AC
«Разделяй и властвуй» (D&C)	WA	AC	WA	WA
«Жадные» алгоритмы	WA	WA	AC	WA
Динамическое программирование	MLE/TLE/AC	MLE/TLE/AC	MLE/TLE/AC	AC
Частотность	Высокая	(Очень) Низкая	Низкая	Высокая

Мы завершим эту главу, отметив, что для некоторых реальных задач, особенно тех, которые классифицируются как NP-сложные [7], многие из подходов, обсуждаемых в данном разделе, не будут работать. Например, задача о рюкзаке (Рюкзак 0–1), которая имеет сложность $O(nS)$ при использовании методов динамического программирования, будет работать слишком медленно, если S велико; задача о коммивояжере, имеющая сложность $O(2^n \times n^2)$ DP, будет работать слишком медленно при n больше 18 (см. **упражнение 3.5.2.7***). Для таких задач мы можем прибегнуть к эвристическому подходу или методам локального поиска, таким как поиск с запретами [26, 25], генетические алгоритмы, алгоритм муравьиной колонии, симуляции восстановления, лучевому поиску и т. д. Однако все эти эвристические методы поиска находятся за рамками программы IOI [20], а также не широко используются в ICPC.

Таблица 3.6. Статистические данные, относящиеся к главе 3

Параметр	Первое издание	Второе издание	Третье издание
Число страниц	32	32 (+0 %)	52 (+63 %)
Письменные упражнения	7	16 (+129 %)	11 + 10* = 21 (+31 %)
Задачи по программированию	109	194 (+78 %)	245 (+26 %)

Распределение количества упражнений по программированию по разделам этой главы показано ниже.

Таблица 3.7. Распределение количества упражнений по программированию по разделам главы 3

Раздел	Название	Число заданий	% в главе	% в книге
3.2	Полный перебор	112	45 %	7 %
3.3	Разделяй и властвуй	23	9 %	1 %
3.4	«Жадные» алгоритмы	45	18 %	3 %
3.5	Динамическое программирование	67	27 %	4 %

Глава 4

Графы

Любых двух людей на земле в среднем разделяет шесть рукопожатий.

– Стэнли Милгрэм, исследование «Мир тесен» в 1969 году [64]

4.1. Общий обзор и мотивация

Многие реальные проблемы могут рассматриваться как задачи из области теории графов. У некоторых из них есть эффективные решения. У некоторых их еще нет. В этой относительно большой главе с большим количеством рисунков мы обсуждаем задачи из области теории графов, которые обычно появляются на олимпиадах по программированию, алгоритмы их решения и практическую реализацию этих алгоритмов. Мы затрагиваем темы, начиная от базовых обходов графов, минимальных остовных деревьев, кратчайших путей из одной вершины / для всех пар вершин, потоков, и обсуждаем графы со специальными свойствами.

При написании данной главы мы предполагаем, что читатели уже знакомы с терминологией графов, перечисленной в табл. 4.1. Если вы столкнулись с каким-либо незнакомым термином, пожалуйста, прочитайте справочную литературу, например [7, 58] (или просмотрите интернет), и найдите тот термин, который вам непонятен.

Таблица 4.1. Список важных терминов теории графов

Вершины/Узлы Взвешенный/ Невзвешенный граф Путь Собственный простой цикл Ориентированный ациклический граф (DAG, Directed Acyclic Graph)	Ребро Ориентированный/ Неориентированный граф Цикл Кратные ребра Дерево/Лес	Множество V; Размер множества $\|V\|$ Разреженный граф Изолированная вершина Мультиграф Эйлеров граф	Множество E; Размер множества $\|E\|$ Плотность Достижимость Простой граф Двудольный граф	Граф G (V, E) Входящая/ исходящая степень Связный граф Подграф Полный граф

Мы также предполагаем, что читатели изучали различные способы представления информации о графах, которые обсуждались ранее в разделе 2.4.1. То есть мы будем напрямую использовать такие термины, как матрица смежности, список смежности, список ребер и неявный граф, не переопределяя их. Пожалуйста, вернитесь к разделу 2.4.1, если вы незнакомы с этими структурами данных графа.

Наше исследование задач на графы в недавних региональных олимпиадах ACM ICPC (Азия) показало, что в наборе задач ICPC есть по крайней мере одна (и, возможно, более) задача, имеющая отношение к графам. Однако, поскольку спектр задач на графы очень велик, каждая отдельная задача на графы имеет малую вероятность появления. Таким образом, на вопрос «на каких задачах мы должны сосредоточиться?» нет четкого ответа. Если вы хотите преуспеть в ACM ICPC, у вас нет иного выбора, кроме как изучить и освоить все эти материалы.

Программа IOI [20] ограничивает задачи подмножеством материалов, упомянутых в этой главе. Это логично, поскольку учащиеся старших классов, участвующие в IOI, вряд ли будут хорошо разбираться в многочисленных алгоритмах для решения конкретных задач. Чтобы помочь читателям, желающим принять участие в IOI, мы упомянем, находится ли определенный раздел этой главы за пределами программы.

4.2. Обход графа

4.2.1. Поиск в глубину (Depth First Search, DFS)

Поиск в глубину – сокращенно DFS – это простой алгоритм обхода графа.

Начиная с выделенной исходной вершины, DFS будет проходить по графу «сначала в глубину». Каждый раз, когда DFS достигает точки ветвления (вершины с несколькими соседями), он выбирает одну из непосещенных соседних вершин и посещает эту соседнюю вершину. DFS повторяет этот процесс и идет глубже, пока не достигнет вершины, у которой нет непосещенных соседей. Когда это происходит, DFS будет «возвращаться назад» и исследовать другую непосещенную соседнюю вершину(ы), если таковая(ые) имее(ю)тся.

Подобное поведение обхода графа может быть легко реализовано с помощью рекурсивного кода, приведенного ниже. Наша реализация DFS использует глобальный вектор целых чисел `vi dfs_num`, чтобы различать состояние каждой вершины. Для простейшей реализации DFS мы применяем только `vi dfs_num`, чтобы различать «непосещенные» (мы используем постоянное значение `UNVISITED = -1`) и «посещенные» (мы используем другое постоянное значение `VISITED = 1`). Первоначально все значения в `dfs_num` устанавливаются как «непосещенные». Позже мы будем использовать `vi dfs_num` для других целей. Вызов `dfs (u)` запускает DFS из вершины *u*, помечает вершину *u* как «посещенную», а затем DFS рекурсивно посещает каждого непосещенного соседа *v* из *u* (т. е. ребро *u* – *v* существует в графе и `dfs_num[v] == UNVISITED`).

```
typedef pair<int, int> ii;                  // В данной главе мы будем часто использовать эти
typedef vector<ii> vii;      // три ярлыка для типов данных. Они могут выглядеть загадочно,
```

```
typedef vector<int> vi;                    // но они полезны в олимпиадном программировании
vi dfs_num;                // глобальная переменная, изначально для всех элементов установлены
                                                                        // значения UNVISITED
void dfs(int u) {              // DFS для обычного использования: как алгоритм обхода графа
  dfs_num[u] = VISITED;                    // важно: мы помечаем эту вершину как посещенную
  for (int j = 0; j < (int)AdjList[u].size(); j++) {                    // структура данных
                                                                  // по умолчанию: AdjList
    ii v = AdjList[u][j];                                        // v - это пара (сосед, вес)
    if (dfs_num[v.first] == UNVISITED)             // важная проверка, чтобы избежать цикла
      dfs(v.first);                  // рекурсивно посещает непосещенных соседей вершины u
} }                 // для простого обхода графа мы игнорируем вес, сохраненный в v.second
```

Временная сложность этой реализации DFS зависит от используемой структуры данных графа. В графе, где имеется V вершин и E ребер, DFS завершает работу за время $O(V + E)$ и $O(V^2)$, если граф хранится как список смежности и матрица смежности соответственно (см. упражнение 4.2.2.2).

На примере графа, показанном на рис. 4.1, функция `dfs(0)`, вызывающая DFS из начальной вершины $u = 0$, запустит такую последовательность посещения вершин: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. Это метод обхода «сначала в глубину», т. е. DFS отправляется в самую глубокую вершину из начальной вершины перед попыткой прохода другой ветви (в данном случае ее нет).

Обратите внимание, что эта последовательность посещения очень сильно зависит от того, как мы упорядочиваем соседей вершины[1], т. е. последовательность $0 \rightarrow 1 \rightarrow 3 \rightarrow 2$ (возврат к 3) $\rightarrow 4$ также возможна.

Также обратите внимание, что один вызов `dfs(u)` будет посещать те, и только те вершины графа, которые связаны с вершиной u. Вот почему вершины 5, 6, 7 и 8 на рис. 4.1 остаются непосещенными после вызова `dfs(0)`.

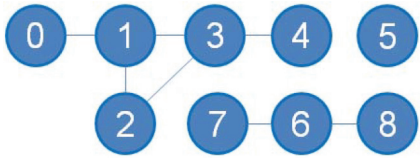


Рис. 4.1 ❖ Пример графа

Код DFS, показанный здесь, очень похож на код возвратной рекурсии, показанный ранее в разделе 3.2. Если мы сравним псевдокод типичного кода возвратной рекурсии (воспроизведенный ниже) с кодом DFS, показанным выше, то увидим, что основным отличием является пометка посещенных вершин (изменение их состояний). Возврат (автоматически) отменяет пометку посещенных вершин (сбросит состояние в предыдущее), когда рекурсия вернется назад, в исходную точку, чтобы разрешить повторное посещение этих вершин (и изменение состояний) из другой ветви. Не посещая повторно вершины общего

[1] Для простоты мы обычно упорядочиваем вершины по их номерам; например, на рис. 4.1 для вершины 1 соседними вершинами будут {0, 2, 3}, при заданном порядке.

графа (посредством проверок dfs_num), DFS завершает работу за время $O(V + E)$, но временная сложность возвратной рекурсии является экспоненциальной.

```
void backtrack(state) {
  if (достигли конечного или недопустимого состояния)     // нам нужно условие останова или
    return;               // сокращения пути, чтобы избежать зацикливания и ускорить поиск
  for each neighbor of this state                          // перебираем все перестановки
    backtrack(neighbor);
}
```

Пример применения: UVa 11902 – Dominator

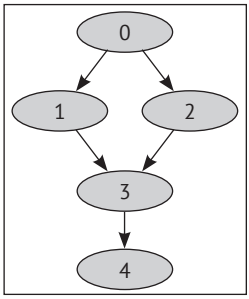


Рис. 4.2 ❖ UVa 11902

Сокращенная формулировка условий задачи: вершина X доминирует над вершиной Y, если каждый путь от начальной вершины *a* (вершины 0 для данной задачи) до Y должен проходить через X. Если вершина Y недостижима из начальной вершины, то у Y нет никакой доминирующей вершины. Каждая вершина, достижимая из начальной вершины, доминирует над собой. Например, на графе, показанном на рис. 4.2, вершина 3 доминирует над вершиной 4, поскольку все пути от вершины 0 до вершины 4 должны проходить через вершину 3. Вершина 1 не доминирует над вершиной 3, так как существует путь 0–2–3, который не включает вершину 1. Наша задача: для заданного графа определить доминирующие вершины для каждой из вершин.

Это задача о проверке достижимости из начальной вершины (вершины 0). Поскольку исходный граф для этой задачи мал ($V < 100$), мы можем позволить себе использовать следующий алгоритм $O(V \times V^2 = V^3)$. Запустите `dfs(0)` на исходном графе, чтобы записать вершины, достижимые из вершины 0. Затем, чтобы проверить, для каких вершин вершина X является доминирующей, мы (временно) отключаем все исходящие ребра вершины X и повторно запускаем `dfs(0)`. Теперь для вершины Y вершина X не является доминирующей, если `dfs(0)` изначально не может достичь вершины Y или `dfs(0)` может достичь вершины Y даже после того, как все исходящие ребра вершины X (временно) отключены. В противном случае вершина доминирует над вершиной Y. Повторим этот процесс $\forall X \in [0...V-1]$.

Советы: нам не нужно физически удалять вершину *X* из исходного графа. Мы можем просто добавить инструкцию в нашу подпрограмму DFS, чтобы остановить обход, если он достигает вершины X.

4.2.2. Поиск в ширину (Breadth First Search, BFS)

Поиск в ширину – сокращенно BFS – это еще один алгоритм обхода графа. Начиная с выделенной исходной вершины, BFS будет обходить граф «в ширину». Таким образом, BFS будет посещать вершины, которые являются прямыми соседями исходной вершины (первый слой), соседями прямых соседей (второй слой) и т. д., слой за слоем.

BFS начинается с добавления исходной вершины *s* в очередь, затем обрабатывает очередь следующим образом: вынимает самую первую вершину *u* из

очереди, ставит в очередь все непосещенные соседние вершины *u* (обычно соседние вершины упорядочены на основе их номеров вершин) и отмечает их как посещенные. В порядке организованной таким образом очереди BFS будет посещать вершины *s* и все вершины в компоненте связности, который содержит *s*, слой за слоем. Алгоритм BFS также выполняется за $O(V + E)$ и $O(V^2)$ на графе, представленном как список смежности и матрица смежности соответственно (опять же, см. упражнение 4.2.2.2).

Реализация BFS проста, если мы применяем библиотеки C++ STL или Java API. Мы используем очередь, чтобы упорядочить последовательность посещения вершин, и `vector<int>` (или `vi`), чтобы записать, была посещена данная вершина или нет, что в то же время также записывает расстояние (номер слоя) каждой вершины от исходной вершины. Эта функция вычисления расстояния позже будет использоваться для решения особого случая задачи нахождения кратчайших путей из одной исходной вершины (см. разделы 4.4 и 8.2.3).

```
// внутри int main()---нет рекурсии
  vi d(V, INF); d[s] = 0;                  // расстояние от исходной вершины s до s равно 0
  queue<int> q; q.push(s);                                  // начинаем с исходной вершины

  while (!q.empty()) {
    int u = q.front(); q.pop();                                  // очередь: слой за слоем!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                // для каждой вершины, являющейся соседней для u
      if (d[v.first] == INF) {                    // если v.first непосещенная + достижимая
        d[v.first] = d[u] + 1;           // установите d[v.first] != INF, чтобы отметить ее
        q.push(v.first);                // добавьте в очередь v.first для следующей итерации
  } } }
```

Если мы запустим BFS из вершины 5 (т. е. исходной вершины *s* = 5) на связном неориентированном графе, показанном на рис. 4.3, то посетим вершины в следующем порядке:

```
Слой 0: посещаем вершину 5
Слой 1: посещаем вершину 1, посещаем вершину 6, посещаем вершину 10
Слой 2: посещаем вершину 0, посещаем вершину 2, посещаем вершину 11, посещаем вершину 9
Слой 3: посещаем вершину 4, посещаем вершину 3, посещаем вершину 12, посещаем вершину 8
Слой 4: посещаем вершину 7
```

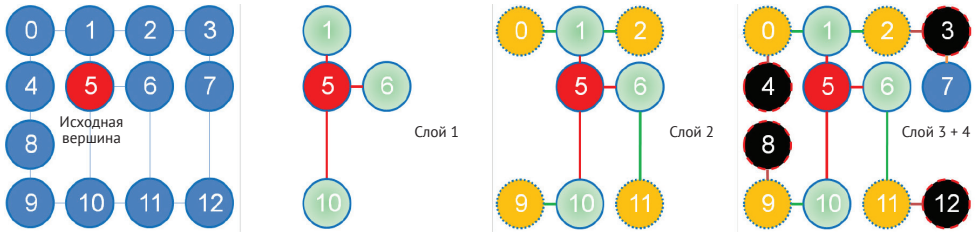


Рис. 4.3 ❖ Пример динамического отображения BFS

Упражнение 4.2.2.1. Чтобы показать, что как DFS, так и BFS могут использоваться для посещения всех вершин, которые достижимы из исходной вершины, решите задачу UVa 11902 – Dominator, используя BFS.

Упражнение 4.2.2.2. Почему DFS и BFS выполняются за $O(V + E)$, если граф хранится в виде списка смежности, и становятся медленнее (выполняются за $O(V^2)$), если граф хранится в виде матрицы смежности? Дополнительный вопрос: какова временная сложность DFS и BFS, если граф хранится в виде списка ребер? Что мы должны делать, если в условиях задачи исходный граф задан в виде списка ребер и мы хотим найти эффективное решение, чтобы обойти весь этот граф?

4.2.3. Поиск компонент связности (неориентированный граф)

Методы DFS и BFS полезны не только для обхода графа. Они могут быть использованы для решения многих других задач, связанных с графами. Первые несколько задач, приведенных ниже, можно решить с помощью DFS или BFS, хотя некоторые из последних среди перечисленных задач больше подходят только для DFS.

Тот факт, что единственный вызов `dfs(u)` (или `bfs(u)`) будет посещать лишь те вершины, которые действительно связаны с u, может быть использован для поиска (и подсчета количества) компонент связности в неориентированном графе (см. далее в разделе 4.2.9 для аналогичной задачи на ориентированном графе). Мы можем просто использовать следующий код для перезапуска DFS (или BFS) из одной из оставшихся непосещенных вершин, чтобы найти следующую компоненту связности. Этот процесс повторяется до тех пор, пока не будут посещены все вершины; общая временная сложность данного метода составляет $O(V + E)$.

```
// внутри int main()--- это решение DFS
  numCC = 0;
  dfs_num.assign(V, UNVISITED);                  // устанавливаем для всех вершин состояние
                                                                  // "непосещенные" (UNVISITED)
  for (int i = 0; i < V; i++)                  // для каждой вершины i в диапазоне [0..V-1]
    if (dfs_num[i] == UNVISITED)                     // если вершина i еще не была посещена
      printf("CC %d:", ++numCC), dfs(i), printf("\n");                 // здесь три строки!

// Для графа на рис. 4.1, рассматриваемого в качестве примера,
// выходные данные выглядят следующим образом:
// CC 1: 0 1 2 3 4
// CC 2: 5
// CC 3: 6 7 8
```

Упражнение 4.2.3.1. UVa 459 – Graph Connectivity – это в основном задача поиска компонент связности неориентированного графа. Решите эту задачу, используя приведенное выше решение DFS.

Однако мы также можем использовать структуру данных системы непересекающихся множеств (Union-Find Disjoint Sets) (см. раздел 2.4.2) или BFS (см. раздел 4.2.2) для решения этой задачи с графом. Каким образом мы можем это сделать?

4.2.4. Закрашивание – Маркировка/раскрашивание компонент связности

DFS (или BFS) можно использовать не только для поиска (и подсчета количества) компонент связности (СК). Здесь мы покажем, как простой способ, имеющий временную сложность $O(V+E)$ `dfs(u)` (или `bfs(u)`), можно использовать для маркировки (другой термин, используемый в информатике, – «раскрасить») и подсчета размера каждого компонента. Этот вариант более известен как «заливка» или «закрашивание» и, как правило, выполняется на неявных графах (обычно на двумерных сетках).

```
int dr[] = {1,1,0,-1,-1,-1, 0, 1};                    // метод для изучения неявной 2D-сетки
int dc[] = {0,1,1, 1, 0,-1,-1,-1};                            // S,SE,E,NE,N,NW,W,SW соседи

int floodfill(int r, int c, char c1, char c2) {                    // возвращает размер СК
  if (r < 0 || r >= R || c < 0 || c >= C) return 0;                  // за пределами сетки
  if (grid[r][c] != c1) return 0;                                     // не имеет цвета c1
  int ans = 1;                // добавляет 1 к ans, потому что вершина (r, c) имеет цвет c1
  grid[r][c] = c2;                        // теперь перекрашивает вершину (r, c) в c2, чтобы
                                                                    // избежать зацикливания!
  for (int d = 0; d < 8; d++)
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);
  return ans;                                          // код корректен благодаря dr[] и dc[]
}
```

Пример использования: UVa 469 – Wetlands of Florida

Давайте рассмотрим пример ниже (UVa 469 – Wetlands of Florida). Неявный граф – это двумерная сетка, в которой вершинами являются ячейки сетки, а ребрами выступают соединения между ячейкой и окружающими ее ячейками, обозначенными как S/SE/E/NE/N/NW/W/SW. «W» обозначает клетку с заболоченной землей, а «L» обозначает клетку с сухой почвой.

Заболоченная зона определяется как соседствующие между собой ячейки, помеченные буквой «W». Мы можем пометить (и одновременно посчитать размер) заболоченную область с помощью заливки. В приведенном ниже примере показано выполнение заливки со 2-й строки, 1-го столбца (значения индекса начинаются с 0), символ «W» заменяется на «.».

Мы хотим сделать замечание о том, что на сайте архива задач университета Вальядолида (UVa) размещено большое количество задач на закрашивание [47], например UVa 1103 – Ancient Messages (Надписи на древних языках); эта задача предлагалась на финальных соревнованиях на кубок мира ICPC в 2011 году. Для освоения подобной техники читателям полезно попытаться решить задачи из упражнений по программированию в этом разделе.

```
// внутри int main()
  // считывание сетки как глобального массива 2D + считывание координат запроса (row, col)
  printf("%d\n", floodfill(row, col, 'W', '.'));    // подсчет размера заболоченной области
                                                                    // возвращаемый ответ: is 12
// LLLLLLLLL         LLLLLLLLL
// LLWWLLWLL         LL..LLWLL                                  // размер компоненты связности
// LWWLLLLLL (R2,C1) L..LLLLLL                                      // (связанные ячейки 'W')
```

```
// LWWWLWWLL         L...L..LL          // вместе с одной ячейкой 'W' (строка 2, столбец 1)
                                                                    // составляют 12
// LLLWWWLLL ======> LLL...LLL
// LLLLLLLLL         LLLLLLLLL                        // Заметьте, что все связанные 'W'
// LLLWWLLWL         LLLWWLLWL                      // заменены на '.' после раскрашивания
// LLWLWLLLL         LLWLWLLLL
// LLLLLLLLL         LLLLLLLLL
```

4.2.5. Топологическая сортировка (направленный ациклический граф)

Топологическая сортировка направленного ациклического графа (Directed Acyclic Graph, DAG) – это линейное упорядочение вершин в DAG, так что если в DAG существует ребро ($u \rightarrow v$), то вершина u предшествует вершине v. Каждый DAG имеет, по крайней мере, один и, *возможно*, несколько вариантов топологической сортировки.

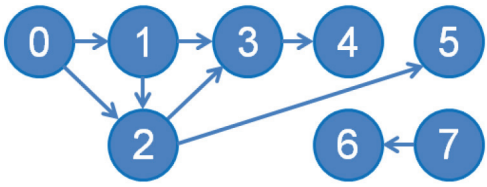


Рис. 4.4 ❖ Пример DAG

Одно из применений топологической сортировки состоит в том, чтобы найти возможную последовательность курсов в рамках программы, которую студент университета должен пройти для получения диплома. Каждый курс имеет определенные предварительные требования, которые необходимы для успешного прохождения курса и обязательно должны выполняться. Эти предварительные требования никогда не бывают циклическими, поэтому их можно смоделировать как DAG. Топологическая сортировка графа предварительных требований к курсу программы дает студенту линейный список курсов, которые нужно проходить один за другим, выполняя все предварительные требования.

Существует несколько алгоритмов топологической сортировки. Самый простой способ – немного изменить реализацию DFS, приведенную выше, в разделе 4.2.1.

```
vi ts;       // глобальный вектор для хранения топологической сортировки в обратном порядке

void dfs2(int u) {                        // другое имя функции по сравнению с исходным dfs
  dfs_num[u] = VISITED;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      dfs2(v.first);
  }
```

```
  ts.push_back(u); }                                  // вот и все, это единственное изменение
// внутри int main()
  ts.clear();
  memset(dfs_num, UNVISITED, sizeof dfs_num);
  for (int i = 0; i < V; i++)                          // эта часть совпадает с поиском СК
    if (dfs_num[i] == UNVISITED)
      dfs2(i);
                  // альтернативный способ, вызов: reverse(ts.begin(), ts.end()); сначала
  for (int i = (int)ts.size() - 1; i >= 0; i--)          // читать в обратном направлении
    printf(" %d", ts[i]);
  printf("\n");

// Для примера графа на рис. 4.4 выходные данные выглядят следующим образом:
// 7 6 0 1 2 5 3 4 (помните, что может быть >= 1 допустимого варианта топологической
// сортировки)
```

В `dfs2(u)` мы добавляем *u* в конец списка (вектора) исследованных вершин только после посещения всех поддеревьев ниже *u* в остовном дереве DFS[1]. Мы добавляем *u* в конец этого вектора, потому что `vector` в C++ STL (`Vector` в Java) поддерживает *эффективную вставку* $O(1)$ только с конца. Список будет вестись в обратном порядке, но мы можем обойти эту проблему, изменив порядок печати выводимых элементов при выводе результатов. Этот простой алгоритм поиска (действительной) топологической сортировки принадлежит Роберту Андре Тарьяну. Он имеет временную сложность $O(V + E)$, что соответствует временной сложности DFS, поскольку использует оригинальный алгоритм DFS с добавлением одной постоянной операции.

В завершение обсуждения топологической сортировки мы покажем еще один алгоритм: алгоритм Кана [36]. Он выглядит как «модифицированный BFS» . В некоторых задачах, например UVa 11060 – Beverages, требуется использовать этот алгоритм Кана для топологической сортировки, а не применять алгоритм на основе DFS, показанный ранее.

```
поставить вершины с нулевой входящей степенью в очередь Q (очередь с приоритетом)
while (Q не пуста) {
  vertex u = Q.dequeue();поместить вершину u в список топологической сортировки;
  удалить вершину u и все исходящие ребра из этой вершины;
  если такое удаление приводит к тому, что вершина v имеет нулевую входящую степень
    Q.enqueue(v); }
```

Упражнение 4.2.5.1. Почему добавления вершины `u` в конец `vi`, то есть `ts.push_back(u)` в стандартном коде DFS, достаточно, чтобы помочь нам найти топологический вид DAG?

Упражнение 4.2.5.2. Можете ли вы определить другую структуру данных, которая поддерживает эффективную вставку за $O(1)$ спереди, чтобы нам не приходилось изменять порядок содержимого `vi ts`?

Упражнение 4.2.5.3. Что произойдет, если мы запустим код топологической сортировки выше для графа, не являющегося DAG?

---

[1] Остовное дерево DFS более подробно обсуждается в разделе 4.2.7.

Упражнение 4.2.5.4. Код топологической сортировки, показанный выше, может генерировать лишь *один* правильный топологический порядок вершин DAG. Что нам делать, если мы хотим вывести все правильные топологические порядки вершин DAG?

4.2.6. Проверка двудольности графа

Двудольный граф имеет важные свойства, которые мы рассмотрим позже в разделе 4.7.4. В этом подразделе мы просто хотим проверить, является ли граф двудольным (или двухцветным) для решения таких задач, как UVa 10004 – Bicoloring. Мы можем использовать BFS или DFS для этой проверки, но мы считаем способ BFS более естественным. Модифицированный код BFS, представленный ниже, начинается с окрашивания исходной вершины (первый слой) значением 0, далее он закрашивает прямых соседей исходной вершины (второй слой) значением 1, снова закрашивает соседей прямых соседей (третий слой) значением 0 и т. д., чередуя значение 0 и значение 1 как единственные два действительных цвета. Если мы столкнемся с каким-либо нарушением (нарушениями) на этом пути – ребром с двумя конечными точками, имеющими одинаковый цвет, – то можем заключить, что данный граф не является двудольным графом.

```
// внутри int main()
  queue<int> q; q.push(s);
  vi color(V, INF); color[s] = 0;
  bool isBipartite = true;  // добавление одного логического флага, начальное значение true
  while (!q.empty() & isBipartite) {              // аналогично оригинальной процедуре BFS
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (color[v.first] == INF) {      // однако, вместо того чтобы записывать расстояние,
        color[v.first] = 1 - color[u];            // мы просто записываем два цвета {0, 1}
        q.push(v.first); }
      else if (color[v.first] == color[u]) {          // u & v.first имеют одинаковый цвет
        isBipartite = false; break; } } }                       // мы имеем конфликт цвета
```

Упражнение 4.2.6.1*. Видоизмените приведенный пример – реализуйте проверку графа на то, является ли он двудольным, с использованием DFS.

Упражнение 4.2.6.2*. Пусть доказано, что *простой* граф с V вершинами является двудольным графом.
Каково максимально возможное количество ребер у этого графа?

Упражнение 4.2.6.3. Докажите (или опровергните) следующее утверждение: «У двудольного графа нет нечетного цикла».

4.2.7. Проверка свойств ребер графа через остовное дерево DFS

При запуске DFS на связном графе генерируется *остовное дерево* DFS[1] (или *остовной лес*[2], если граф несвязный). С помощью еще одного состояния вершины: `EXPLORED = 2` (посещено, но *еще не завершено*) поверх `VISITED` (посещено и завершено) – мы можем использовать это остовное дерево (или лес) DFS для классификации ребер графа по трем типам:

1) ребро дерева: ребро, пройденное DFS, то есть ребро, исходящее из вершины, которая в данный момент находится в состоянии `EXPLORED`, и входящее в вершину с состоянием `UNVISITED`;
2) обратное ребро: ребро, являющееся частью цикла, т. е. ребро, исходящее из вершины с текущим состоянием `EXPLORED`, и входящее в вершину, также имеющую состояние `EXPLORED`. Это важный момент для данного алгоритма. Обратите внимание, что мы обычно не считаем двунаправленные ребра имеющими «цикл» (нам нужно помнить `dfs_parent`, чтобы различать это, см. код, приведенный ниже);
3) прямые/перекрестные ребра, исходящие из вершины, имеющей состояние `EXPLORED`, и входящие в вершину с состоянием `VISITED`. Эти два типа ребер обычно не тестируются в олимпиадных задачах по программированию.

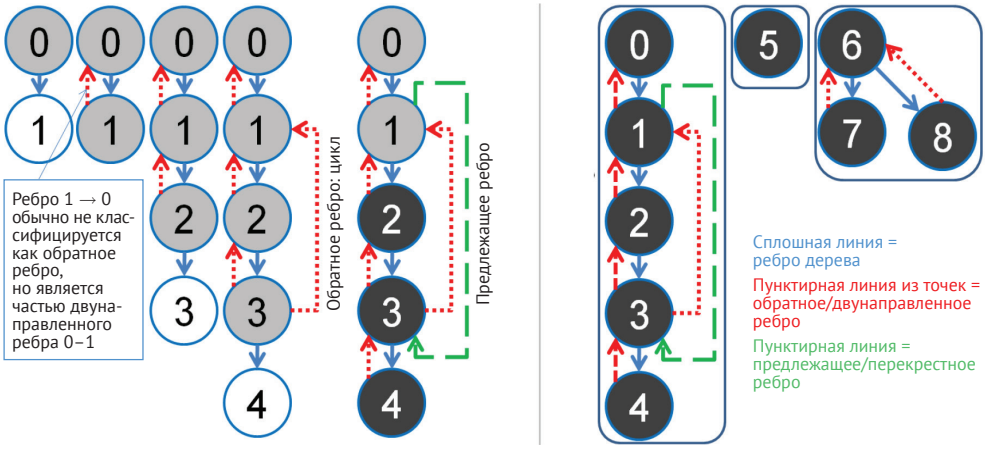


Рис. 4.5 ❖ Динамическое отображение DFS при запуске на графе, показанном на рис. 4.1

---

[1] Остовное дерево связного графа G – это дерево, которое охватывает (покрывает) все вершины группы G, но использует только подмножество ребер группы G.

[2] Несвязный граф G имеет несколько связных компонент. Каждая компонента имеет свое собственное остовное поддерево. Все остовные поддеревья G, по одному от каждого компонента, образуют то, что мы называем остовным лесом.

На рис. 4.5 показано воспроизведение (слева направо) вызова `dfs(0)` (оно дано более подробно), затем `dfs(5)` и, наконец, `dfs(6)` на примере графа, изображенного на рис. 4.1. Мы можем видеть, что 1 → 2 → 3 → 1 является (истинным) циклом, и мы классифицируем ребро (3 → 1) как обратное ребро, тогда как 0 → 1 → 0 не является циклом, это просто двунаправленное ребро (0–1). Код для этого варианта DFS приведен ниже.

```
void graphCheck(int u) {                          // DFS для проверки свойств ребер графа
  dfs_num[u] = EXPLORED;                        // раскрашиваем u как EXPLORED, а не VISITED
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {              // Ребро дерева, EXPLORED->UNVISITED
      dfs_parent[v.first] = u;             // родителем этих потомков является этот элемент
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == EXPLORED) {                        // EXPLORED->EXPLORED
      if (v.first == dfs_parent[u])                       // чтобы различать эти два случая
        printf("Двунаправленное (%d, %d)-(%d, %d)\n", u, v.first, v.first, u);
      else              // наиболее распространенное применение: проверка цикличности графа
        printf("Обратное ребро (%d, %d) (Цикл)\n", u, v.first);
    }
    else if (dfs_num[v.first] == VISITED)                            // EXPLORED->VISITED
      printf("Предлежащее/перекрестное ребро (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = VISITED;                // после рекурсии, раскрашиваем u как VISITED (DONE)
}

// внутри int main()
  dfs_num.assign(V, UNVISITED);
  dfs_parent.assign(V, 0);                                                  // новый вектор
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED)
      printf("Component %d:\n", ++numComp), graphCheck(i);                 // 2 строки в 1!

// Для графа, приведенного на рис. 4.1, выходные данные будут:
// Компонента 1:
//  Двунаправленное (1, 0) - (0, 1)
//  Двунаправленное (2, 1) - (1, 2)
//  Обратное ребро (3, 1) (Цикл)
//  Двунаправленное (3, 2) - (2, 3)
//  Двунаправленное (4, 3) - (3, 4)
//  Предлежащее/перекрестное ребро (1, 3)
// Компонента 2:
// Компонента 3:
//  Двунаправленное (7, 6) - (6, 7)
//  Двунаправленное (8, 6) - (6, 8)
```

Упражнение 4.2.7.1. Выполните проверку свойства ребер графа, представленного на рис. 4.9. Предположим, что вы запускаете DFS из вершины 0. Сколько обратных ребер вы можете найти на этот раз?

4.2.8. Нахождение точек сочленения и мостов (неориентированный граф)

Задача: пусть дана дорожная карта (неориентированный граф) со стоимостью перекрытия для всех перекрестков (вершин графа) и дорог (ребер графа). Перекройте либо один перекресток, либо одну дорогу, так что дорожная сеть будет перерезана (отключена), причем это будет сделано наименее затратным способом. Это задача поиска точки сочленения (перекрестка) с наименьшей стоимостью или моста (дороги) с наименьшей стоимостью на неориентированном графе (дорожной карте).

Точка сочленения определяется как *вершина* графа G, удаление которой (при этом также удаляются все ребра, соединенные с этой вершиной) разъединяет G. Граф, в котором не существует ни одной точки сочленения, называется *двусвязным*. Точно так же *мост* определяется как ребро графа G, удаление которого разъединяет G. Эти две задачи обычно определяются для неориентированных графов (они более сложны для ориентированных графов, и для их решения требуется другой алгоритм, см. [35]).

Наивный алгоритм поиска точек сочленения заключается в следующем (его можно изменить, чтобы найти мосты):

1) запустите DFS (или BFS) с временной сложностью $O(V+E)$, чтобы подсчитать количество компонентов связности (СК) исходного графа. Обычно входные данные представляют собой связный граф, поэтому проверка чаще дает нам один компонент связности;
2) для каждой вершины $v \in V$ // $O(V)$:
   a) вырежите (удалите) вершину v и ее инцидентные ребра;
   b) запустите $O(V+E)$ DFS (или BFS) и посмотрите, увеличивается ли число компонентов связности;
   c) если число компонентов связности увеличивается, v является точкой сочленения. Восстановите v и соединенные с ней ребра графа.

Этот наивный алгоритм запускает DFS (или BFS) $O(V)$ раз, поэтому он будет выполняться за время $O(V \times (V+E)) = O(V^2 + VE)$. Но это не лучший алгоритм, так как мы можем просто запустить DFS, выполняющийся за время $O(V+E)$, один раз, чтобы определить все точки сочленения и мосты.

Этот вариант решения задачи, использующий DFS, благодаря Джону Эдварду Хопкрофту и Роберту Андре Тарьяну (см. [63] и задачу 22.2 в [7]) является еще одним вариантом применения кода DFS, приведенного ранее.

Теперь мы имеем два числовых значения: `dfs_num(u)` и `dfs_low(u)`. Здесь `dfs_num(u)` сохраняет счетчик итераций при первом посещении вершины u (не только для того, чтобы отличить `UNVISITED` от `EXPLORED/VISITED`). Другое числовое значение `dfs_low(u)` хранит самое маленькое значение `dfs_num`, достижимое из текущего остовного поддерева u в DFS. Вначале, когда вершина u посещается впервые, `dfs_low(u) = dfs_num(u)`. Тогда значение `dfs_low(u)` может быть уменьшено только при наличии цикла (обратное ребро существует). Обратите внимание, что мы не обновляем `dfs_low(u)` на основании найденного обратного ребра (u, v), если v является прямым родителем u.

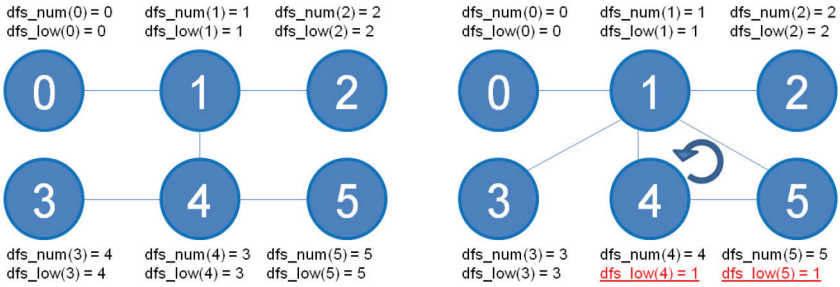


Рис. 4.6 ❖ Добавление двух дополнительных атрибутов DFS: `dfs_num` и `dfs_low`

Обратимся к рис. 4.6 для ясности. На обоих графах мы запускаем вариант DFS из вершины 0. Предположим, что для графа на рис. 4.6 (слева) последовательность обхода будет 0 (на итерации 0) → 1 (1) → 2 (2) (возврат к 1) → 4 (3) → 3 (4) (возврат к 4) → 5 (5). Убедитесь, что эти счетчики итераций правильно отображаются в `dfs_num`. Поскольку в этом графе нет обратного ребра, все `dfs_low` = `dfs_num`.

Предположим, что для графа на рис. 4.6 (справа) последовательность обхода будет 0 (на итерации 0) → 1 (1) → 2 (2) (возврат к 1) → 3 (3) (возврат к 1) → 4 (4) → 5 (5). На этом этапе в остовном дереве DFS существует важное обратное ребро, которое формирует цикл, – это ребро 5–1, которое является частью цикла 1–4–5–1. Это позволяет из вершин 1, 4 и 5 достигать вершины 1 (при этом `dfs_num` принимает значение 1). Таким образом, все значения `dfs_low` {1, 4, 5} равны 1.

Когда мы находимся в вершине *u*, у которой имеется соседняя вершина *v*, и `dfs_low(v)` ≥ `dfs_num(u)`, то вершина *u* является точкой сочленения. Это утверждение верно в силу того, что условие `dfs_low(v)` не меньше, чем `dfs_num(u)`, подразумевает, что не существует обратного ребра от вершины *v*, которое позволяло бы достичь другой вершины *w* с меньшим значением `dfs_num(w)`, чем `dfs_num(u)`. Вершина *w* с более низким `dfs_num(w)`, чем вершина *u* с `dfs_num(u)`, подразумевает, что *w* является предком *u* в остовном дереве DFS. Это означает, что для достижения предка(ов) *u* из *v* необходимо пройти через вершину *u*. Следовательно, удаление вершины *u* приведет к разъединению графа.

Однако есть один **особый случай**: корень остовного дерева DFS (вершина, выбранная в качестве начальной при вызове процедуры DFS) является точкой сочленения, только если в остовном дереве DFS имеется несколько дочерних элементов (тривиальный случай, который не определяется этим алгоритмом).

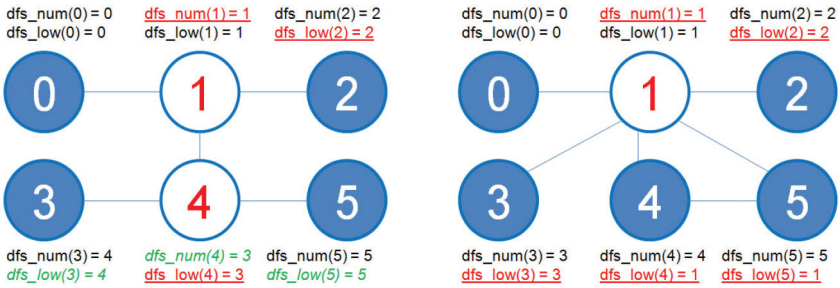


Рис. 4.7 ❖ Поиск точек сочленения с использованием `dfs_num` и `dfs_low`

Просмотрите рис. 4.7, чтобы более подробно разобрать задачу. На графе, изображенном на рис. 4.7 (слева), вершины 1 и 4 являются точками сочленения, потому что, например, в ребре 1–2 мы видим, что `dfs_low(2)` ≥ `dfs_num(1)`, и в ребре 4–5 мы также видим, что `dfs_low(5)` ≥ `dfs_num(4)`. Для графа, изображенного на рис. 4.7 (справа), только вершина 1 является точкой сочленения, потому что, например, на ребре 1–5 `dfs_low(5)` ≥ `dfs_num(1)`.

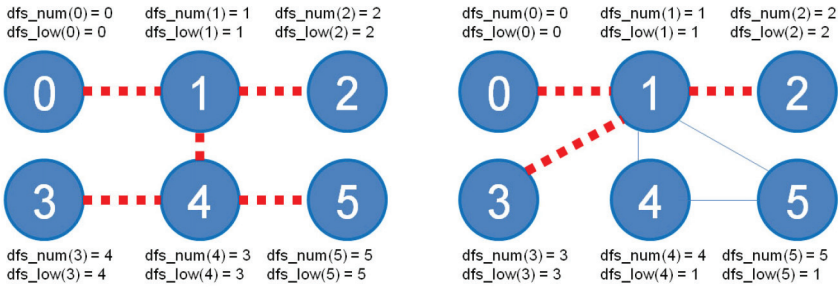


Рис. 4.8 ❖ Поиск мостов, также с использованием `dfs_num` и `dfs_low`

Поиск мостов аналогичен разобранному примеру. Если `dfs_low(v)> dfs_num(u)`, то ребро *u*–*v* – мост (обратите внимание, что мы удаляем в условии проверки для нахождения мостов проверку на равенство «=»). На рис. 4.8 почти все ребра являются мостами для левого и правого графов. Только ребра 1–4, 4–5 и 5–1 не являются мостами на правом графе (они фактически образуют цикл). Это происходит потому, что, например, для ребра 4–5 у нас есть `dfs_low(5)` ≤ `dfs_num(4)`, т. е. даже если ребро 4–5 удалено, мы точно знаем, что из вершины 5 все еще можно достичь вершины 1 через другой путь, который обходит вершину 4, так как `dfs_low(5) = 1` (этот другой путь фактически является ребром 5–1). Код, реализующий поиск мостов, приведен ниже:

```
void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;                  // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {                                  // ребро дерева
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;                  // частный случай, если u - корень

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])                              // для точки сочленения
        articulation_vertex[u] = true;                 // сначала сохраните эту информацию
      if (dfs_low[v.first] > dfs_num[u])                                        // для моста
        printf(" Ребро (%d, %d) является мостом\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);                // обновляем dfs_low[u]
    }
    else if (v.first != dfs_parent[u])                  // обратное ребро и непрямой цикл
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);                // обновляем dfs_low[u]
} }
```

```
// внутри int main()
  dfsNumberCounter = 0; dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0);
  dfs_parent.assign(V, 0); articulation_vertex.assign(V, 0);
  printf("Bridges:\n");
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED) {
      dfsRoot = i; rootChildren = 0; articulationPointAndBridge(i);
      articulation_vertex[dfsRoot] = (rootChildren > 1); }               // частный случай
  printf("Точки сочленения:\n");
  for (int i = 0; i < V; i++)
    if (articulation_vertex[i])
    printf(" Вершина %d\n", i);
```

Упражнение 4.2.8.1. Изучите граф на рис. 4.1, не используя алгоритм, приведенный выше.

Какие вершины являются точками сочленения, а какие – мостами?

Далее запустите алгоритм и убедитесь, что вычисленные значения `dfs_num` и `dfs_low` каждой вершины графа на рис. 4.1 можно использовать для определения тех же точек сочленения и мостов, найденных вручную!

4.2.9. Нахождение компонент сильной связности (ориентированный граф)

Еще одно применение DFS – найти компоненты сильной связности в ориентированном графе, как, например, в задаче UVa 11838 – Come and Go. Эта задача отличается от задачи поиска компонентов связности в неориентированном графе. На рис. 4.9 показан граф, аналогичный графу на рис. 4.1, за исключением одной детали: его ребра имеют направление. Хотя граф на рис. 4.9 выглядит так, как будто он имеет один компонент *связности*, на самом деле этот компонент не является *сильно связанным* компонентом. В ориентированных графах нас больше интересует понятие *компонент сильной связности* (Strongly Connected Component, SCC).

SCC определяется следующим образом: если мы выберем любую пару вершин u и v в SCC, то сможем найти путь от u до v, и наоборот. На самом деле на рис. 4.9 есть три компонента сильной связности, которые выделены тремя рамками: {0}, {1, 3, 2} и {4, 5, 7, 6}. Примечание: если эти компоненты сильной связности свернуты (заменены большими вершинами), они образуют ориентированный ациклический граф (см. также раздел 8.4.3).

Существует по крайней мере два известных алгоритма для поиска SCC: алгоритм Косараджу, объяснение которого приводится в [7], и алгоритм Тарьяна [63]. В этом разделе мы рассматриваем версию Тарьяна, поскольку она естественно вытекает из нашего предыдущего обсуждения способа нахождения точек сочленения и мостов, следующего логике алгоритма Тарьяна. Мы обсудим алгоритм Косараджу позже, в разделе 9.17.

Основная идея алгоритма состоит в том, что SCC образуют поддеревья в остовном дереве DFS (сравните исходный ориентированный граф и остовное

дерево DFS на рис. 4.9). Помимо вычисления значений `dfs_num(u)` и `dfs_low(u)` для каждой вершины, мы также добавляем вершину *u* в конец стека *S* (здесь стек реализован как вектор) и отслеживаем вершины, которые в настоящее время исследуются с помощью `vi visited`. Условие обновления `dfs_low(u)` немного отличается от предыдущего алгоритма DFS для поиска точек сочленения и мостов. Здесь включены только вершины, в данный момент имеющие флаг `visited` (часть текущего SCC), которые могут обновить значение `dfs_low(u)`. Теперь, если у нас есть вершина *u* в этом остовном дереве DFS, у которой `dfs_low(u) = dfs_num(u)`, мы можем заключить, что *u* является корнем (началом) SCC (наблюдаем вершину 0, 1 и 4 на рис. 4.9), и члены этих SCC идентифицируются путем выталкивания текущего содержимого стека *S*, пока мы снова не достигнем вершины *u* (корня) SCC.

На рис. 4.9 содержимое *S* равно {0, 1, 3, 2, 4, 5, 7, 6}, когда вершина 4 идентифицируется как корень SCC (`dfs_low(4) = dfs_num(4) = 4`) , поэтому мы вставляем элементы в *S* один за другим, пока не достигнем вершины 4 и не получим SCC: {6, 7, 5, 4}. Далее, содержимое *S* равно {0, 1, 3, 2}, когда вершина 1 идентифицируется как другой корень другого SCC (`dfs_low(1) = dfs_num(1) = 1`), поэтому мы добавляем элементы в *S* один за другим, пока не достигнем вершины 1 и не получим SCC: {2, 3, 1}. Наконец, у нас есть последний SCC только с одним членом: {0}.

Приведенный ниже код исследует ориентированный граф и сообщает о существующих у этого графа компонентах сильной связности. Этот код в основном является небольшой модификацией стандартного кода DFS. Рекурсивная часть аналогична стандартной DFS, а часть, выполняющая вывод данных SCC, будет выполняться $O(V)$ раз, поскольку каждая вершина будет принадлежать только одному SCC и, таким образом, будет выводиться только один раз. В целом этот алгоритм все еще будет работать, укладываясь во временную оценку $O(V + E)$.

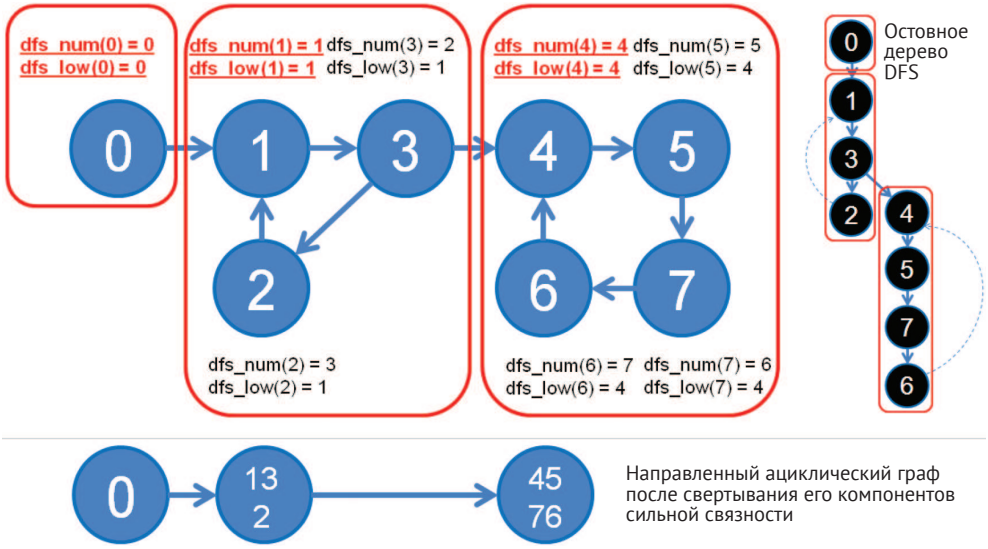


Рис. 4.9 ❖ Пример ориентированного графа и его компонентов сильной связности

```
vi dfs_num, dfs_low, S, visited;                                  // глобальные переменные

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;                  // dfs_low[u] <= dfs_num[u]
  S.push_back(u);                    // хранит u в векторе, основанном на порядке посещения
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      tarjanSCC(v.first);
    if (visited[v.first])                                        // условие обновления значения
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); }

  if (dfs_low[u] == dfs_num[u]) {                              // если это корень (начало) SCC
    printf("SCC %d:", ++numSCC);                    // эта часть выполняется после рекурсии
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break; }
    printf("\n");
} }

// внутри int main()
  dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0); visited.assign(V, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED)
      tarjanSCC(i);
```

Файл исходного кода: *ch4_01_dfs.cpp/java; ch4_02_UVa469.cpp/java*

Упражнение 4.2.9.1. Докажите (или опровергните) следующее утверждение: «Если две вершины находятся в одном и том же компоненте сильной связности, то между ними не существует пути, который когда-либо выйдет за пределы вышеуказанной компоненты».

Упражнение 4.2.9.2*. Напишите код, который принимает на вход ориентированный граф, а затем преобразует его в ориентированный ациклический граф (DAG), свертывая SCC (например, рис. 4.9, сверху вниз).
См. раздел 8.4.3, где приведен пример приложения.

Замечания о задачах на обход графов на олимпиадах по программированию

Примечательно, что простые алгоритмы обхода DFS и BFS имеют так много интересных вариантов, которые могут быть использованы для решения различных задач, использующих графы, где требуется выполнить обход графа. На олимпиаде ICPC могут быть предложены задачи на применение любого из этих вариантов. В соревнованиях IOI могут появляться творческие задачи, связанные с обходом графа.

Использование DFS (или BFS) для поиска компонентов связности в неориентированном графе само по себе редко встречается в формулировках задач, хотя его разновидность – раскрашивание – является одним из наиболее часто встречавшихся типов задач на прошедших олимпиадах по программированию. Тем не менее мы считаем, что количество (новых) задач на раскрашивание уменьшается.

Топологическая сортировка редко используется сама по себе, но это полезный этап предварительной обработки данных для «DP на (неявном) DAG», см. раздел 4.7.1. Простейшую версию кода топологической сортировки очень легко запомнить, так как это всего лишь простой вариант DFS. Альтернативный алгоритм Кана («модифицированный BFS», который ставит в очередь только вершины с числом входящих компонентов 0) также весьма прост.

Эффективные решения для проверки двудольного графа, выполнимые за время $O(V + E)$, проверки свойств ребер графа и нахождения точек сочленения/мостов также полезно знать, но, как видно из статистики архива задач университета Вальядолида (UVa) (и итогов проведения недавних региональных олимпиад ICPC в Азии), в настоящее время они используются в небольшом числе задач.

Знание алгоритма Тарьяна (нахождение компонентов сильной связности) может оказаться полезным для решения современных задач, когда одна из подзадач включает в себя ориентированные графы, которые «требуют преобразования» в DAG путем сокращения циклов (см. раздел 8.4.3). Библиотечный код, приведенный в этой книге, возможно, следует принести на олимпиаду по программированию, где позволяется иметь копию распечатанного библиотечного кода (например, ICPC). Однако на олимпиаде IOI это вряд ли пригодится: тема «Компоненты сильной связности» в настоящее время исключена из программы IOI 2009 [20].

Следует заметить, что многие задачи на графы, обсуждаемые в этом разделе, решаются с помощью DFS или BFS. Лично мы считаем, что многие из них легче решить с помощью рекурсивного и менее требовательного к памяти DFS. Обычно мы не используем BFS в задачах на простой обход графа, но будем использовать его для решения задач нахождения кратчайших путей из одной исходной вершины для невзвешенного графа (см. раздел 4.4). В табл. 4.2 приведено важное сравнение этих двух популярных алгоритмов обхода графа.

Таблица 4.2. Сравнительная таблица для алгоритмов обхода графа

	$O(V + E)$ DFS	**$O(V + E)$ BFS**
За	*Обычно* использует меньше памяти Может найти точки сочленения, мосты, SCC	Используется для нахождения кратчайших путей из одной исходной вершины (для невзвешенного графа)
Против	Не применима для нахождения кратчайших путей из одной исходной вершины для невзвешенного графа	*Обычно* использует больше памяти
Код	Чуть проще писать код	Написание кода занимает чуть больше времени

Анимированная иллюстрация алгоритмов DFS/BFS и некоторые из их вариантов приведены по ссылке URL ниже. Перейдите по данной ссылке, чтобы лучше понять, как работают эти алгоритмы.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/dfsbfs.html

Задачи по программированию, связанные с обходом графа

- Простой обход графа
  1. UVa 00118 – Mutant Flatworld Explorers (обход неявного графа)
  2. UVa 00168 – Theseus and the… (матрица смежности, анализ, обход)
  3. UVa 00280 – Vertex (граф; тест на достижимость вершины путем обхода графа)
  4. *UVa 00318 – Domino Effect* (обход; будьте осторожны с крайними случаями)
  5. UVa 00614 – Mapping the Route (обход неявного графа)
  6. UVa 00824 – Coast Tracker (обход неявного графа)
  7. UVa 10113 – Exchange Rates (просто обход графа, однако используются дроби и функция gcd, см. соответствующие разделы главы 5)
  8. UVa 10116 – Robot Motion (обход неявного графа)
  9. UVa 10377 – Maze Traversal (обход неявного графа)
  10. UVa 10687 – Monitoring the Amazon (построение графа, геометрия, достижимость)
  11. **UVa 11831 – Sticker Collector… *** (неявный граф; порядок ввода – «NSEW»)
  12. UVa 11902 – Dominator (отключайте вершины одну за одной по очереди; проверьте, изменяется ли достижимость из вершины 0)
  13. **UVa 11906 – Knight in a War Grid *** (DFS/BFS для определения достижимости, несколько хитрых случаев; будьте осторожны, когда M = 0 || N = 0 || M = N)
  14. *UVa 12376 – As Long as I Learn, I Live* (симуляция «жадного» алгоритма обхода DAG)
  15. ***UVa 12442 – Forwarding Emails **** (модифицированный DFS, частный случай графа)
  16. *UVa 12582 – Wedding of Sultan* (приведен обход графа DFS, подсчитайте степень каждой вершины)
  17. IOI 2011 – Tropical Garden (обход графа; DFS; используется цикл)
- Закрашивание / Поиск компонентов связности
  1. UVa 00260 – Il Gioco dell'X (шесть соседей на каждую клетку)
  2. UVa 00352 – The Seasonal War (найти количество компонентов связности (СК))
  3. UVa 00459 – Graph Connectivity (также решается с помощью «union find»)
  4. UVa 00469 – Wetlands of Florida (вычислите размер СК; обсуждается в этом разделе)

5. UVa 00572 – Oil Deposits (подсчитать количество СК, решается аналогично UVa 352)
6. UVa 00657 – The Die is Cast (здесь есть три «цвета»)
7. *UVa 00722 – Lakes* (вычислите размер СК)
8. *UVa 00758 – The Same Game* (закрашивание ++)
9. UVa 00776 – Monkeys in a Regular… (обозначьте СК индексами, отформатируйте выходные данные)
10. UVa 00782 – Countour Painting (замените символ « » на «#» в сетке)
11. UVa 00784 – Maze Exploration (очень похоже на UVa 782)
12. UVa 00785 – Grid Coloring (также очень похоже на UVa 782)
13. UVa 00852 – Deciding victory in Go (интересная настольная игра «го»)
14. UVa 00871 – Counting Cells in a Blob (определите размер наибольшего СК)
15. ***UVa 01103 – Ancient Messages*** * (LA 5130, финальные соревнования чемпионата мира, Орландо'11; главный совет: каждый иероглиф имеет уникальный номер белого связанного компонента; затем это упражнение по реализации для анализа входных данных и выполнения заливки для определения количества белых СК внутри каждого черного иероглифа)
16. UVa 10336 – Rank the Languages (считайте и ранжируйте СК с одинаковым цветом)
17. UVa 10707 – 2D-Nim (проверьте изоморфизм графов; утомительная задача; определение компонентов связности)
18. UVa 10946 – You want what filled? (найдите СК и оцените их по размеру)
19. **UVa 11094 – Continents** * (сложная задача на закрашивание, используется прокрутка)
20. UVa 11110 – Equidivisions (закрашивание + соответствие заданным ограничениям)
21. UVa 11244 – Counting Stars (количество СК)
22. UVa 11470 – Square Sums (вы можете закрашивать слой за слоем; однако есть и другой способ решения этой задачи, например путем поиска структур)
23. UVa 11518 – Dominos 2 (в отличие от UVa 11504, мы рассматриваем компоненты сильной связности как простые компоненты связности)
24. UVa 11561 – Getting Gold (раскрашивание с дополнительным ограничением блокировки)
25. UVa 11749 – Poor Trade Advisor (найдите самый большой СК с самой высокой средней годовой прибылью)
26. **UVa 11953 – Battleships** * (интересная задача, решается с помощью закрашивания)

- Топологическая сортировка
  1. UVa 00124 – Following Orders (используйте рекурсию для создания топологической сортировки)
  2. UVa 00200 – Rare Order (топологическая сортировка)

3. **UVa 00872 – Ordering** * (аналогично UVa 124, используйте возвратную рекурсию)
4. **UVa 10305 – Ordering Tasks** * (используйте алгоритм топологической сортировки, приведенный в этом разделе)
5. **UVa 11060 – Beverages** * (необходимо использовать алгоритм Кана для топологической сортировки («модифицированный BFS»))
6. UVa 11686 – Pick up sticks (топологическая сортировка + проверка цикла)
   Также см.: задачи DP на (неявных) DAG (см. раздел 4.7.1).

- Проверка двудольных графов
  1. **UVa 10004 – Bicoloring** * (проверка графа, является ли он двудольным графом)
  2. UVa 10505 – Montesco vs Capuleto (двудольный граф, вычислить максимум (слева, справа))
  3. **UVa 11080 – Place the Guards** * (проверка графа, является ли он двудольным, некоторые сложные случаи)
  4. **UVa 11396 – Claw Decomposition** * (это просто проверка графа, является ли он двудольным)

- Нахождение точек сочленения / мостов
  1. **UVa 00315 – Network** * (поиск точек сочленения)
  2. UVa 00610 – Street Directions (нахождение мостов)
  3. **UVa 00796 – Critical Links** * (нахождение мостов)
  4. UVa 10199 – Tourist Guide (поиск точек сочленения)
  5. ***UVa 10765 – Doves and Bombs*** * (поиск точек сочленения)

- Поиск компонент сильной связности (SCC)
  1. **UVa 00247 – Calling Circles** * (решение SCC + вывод ответа)
  2. UVa 01229 – Sub-dictionary (LA 4099, Иран'07, идентифицируйте SCC графа; эти найденные таким образом вершины и вершины, из которых имеется путь к ним (например, необходимо понимать эти слова), являются ответами на вопрос задачи)
  3. UVa 10731 – Test (SCC + вывод ответа)
  4. **UVa 11504 – Dominos** * (интересная задача: считать |(всех) SCC| без входящего ребра из вершины, выходящей за пределы SCC)
  5. UVa 11709 – Trust Groups (найти число SCC)
  6. UVa 11770 – Lighting Away (аналогично UVa 11504)
  7. **UVa 11838 – Come and Go** * (проверьте, является ли граф сильно связным)

4.3. Минимальное остовное дерево

4.3.1. Обзор

Задача: для заданного связного, ненаправленного и взвешенного графа G (см. крайний левый граф на рис. 4.10) выберите подмножество ребер $E' \in G$ так,

чтобы граф G оставался связным, а общий вес выбранных ребер E' был минимальным.

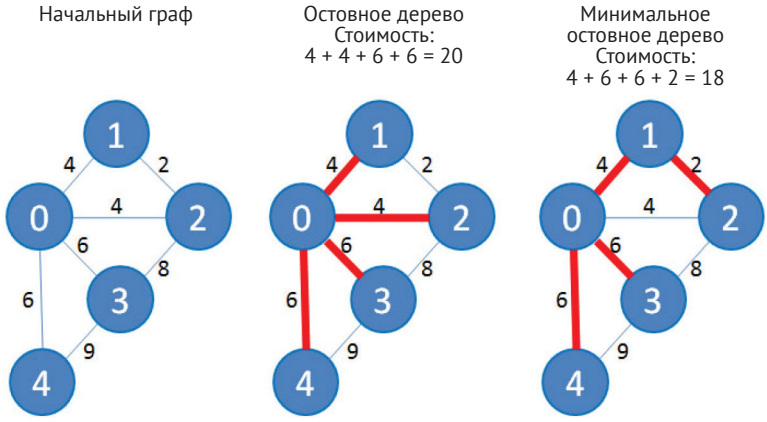


Рис. 4.10 ❖ Пример задачи построения минимального остовного дерева

Чтобы удовлетворить критериям связности, нам нужно, по крайней мере, $V - 1$ ребер, которые образуют дерево, и это дерево должно охватывать (покрывать) все $V \in G$ – т. е. остовное дерево. В G может быть несколько остовных деревьев; т. е. если обратиться к рис. 4.10, это средний и правый примеры. Остовные деревья DFS и BFS, которые мы изучили в предыдущем разделе 4.2, также удовлетворяют данному критерию. Среди множества остовных деревьев есть несколько (по крайней мере одно), которые удовлетворяют критериям минимального веса. Эта задача называется задачей построения минимального остовного дерева (Minimum Spanning Tree, MST) и имеет много практических применений. Например, мы можем смоделировать задачу построения дорожной сети в отдаленных деревнях как задачу MST. Вершины графа – деревни. Ребра – потенциальные дороги, которые могут быть построены между этими деревнями. Стоимость строительства дороги, соединяющей деревни i и j, равна весу ребра (i, j). Таким образом, MST этого графа – это дорожная сеть с минимальными затратами, которая соединяет все эти деревни. В архиве задач университета Вальядолида (UVa) [47] у нас есть несколько задач на построение MST, например: UVa 908, 1174, 1208, 10034, 11631 и др.

Эта задача построения MST может быть решена с помощью нескольких известных алгоритмов, а именно алгоритмов Прима и Краскала. Оба являются «жадными» алгоритмами и рассматриваются во многих учебниках по информатике и программированию [7, 58, 40, 60, 42, 1, 38, 8]. Вес MST, создаваемый этими двумя алгоритмами, уникален; однако может существовать несколько остовных деревьев с одинаковым весом MST.

4.3.2. Алгоритм Краскала

Алгоритм Джозефа Бернарда *Краскала*-младшего сначала сортирует ребра E по неубыванию веса. Это можно легко сделать, сохранив ребра в структуре данных

EdgeList (см. раздел 2.4.1), а затем отсортировав ребра по неубыванию веса. Затем алгоритм Краскала «жадно» пытается добавить каждое ребро в MST, если такое добавление не образует цикл. Эта проверка цикла может быть легко выполнена с использованием системы непересекающихся множеств (Union-Find Disjoint Sets), как показано в разделе 2.4.2. Код, реализующий этот алгоритм, короткий (потому что мы выделили код реализации системы непересекающихся множеств в отдельный класс). Общее время выполнения данного алгоритма составляет O(сортировка + попытка добавить каждое ребро × стоимость операций Union-Find) = $O(E \log E + E \times (\approx 1)) = O(E \log E) = O(E \log V^2) = O(2 \times E \log V) = O(E \log V)$.

```
// внутри int main()
  vector< pair<int, ii> > EdgeList;                          // (вес, две вершины) ребра
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);                  // считываем три параметра: (u, v, w)
    EdgeList.push_back(make_pair(w, ii(u, v))); }                              // (w, u, v)
  sort(EdgeList.begin(), EdgeList.end());          // сортировка по весу ребра O(E log E)
                          // Примечание: объект pair имеет встроенную функцию сравнения
  int mst_cost = 0;
  UnionFind UF(V);                             // все V изначально непересекающихся множеств
  for (int i = 0; i < E; i++) {                                  // для каждого ребра, O(E)
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second)) {              // проверка
      mst_cost += front.first;                                  // добавляем вес e в MST
      UF.unionSet(front.second.first, front.second.second);                // связываем их
  } }                                     // замечание: время работы UFDS очень маленькое
  // Примечание: число непересекающихся множеств должно в конечном счете
  // быть 1 для действительного MST
  printf("Стоимость MST = %d (алгоритм Краскала)\n", mst_cost);
```

На рис. 4.11 показано пошаговое выполнение алгоритма Краскала на графе, изображенном на рис. 4.10 в крайнем левом углу. Обратите внимание, что результирующее минимальное остовное дерево (MST) не является уникальным.

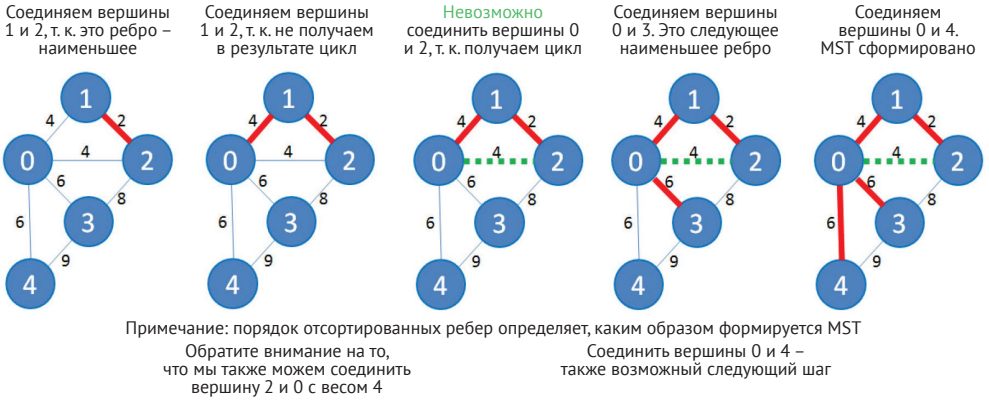


Рис. 4.11 ❖ Динамическое изображение алгоритма Краскала для задачи MST

Упражнение 4.3.2.1. Приведенный выше код останавливается только после обработки последнего ребра в EdgeList. Во многих случаях мы можем остановить выполнение алгоритма Краскала раньше. Измените код, чтобы реализовать это.

Упражнение 4.3.2.2*. Можете ли вы решить задачу MST с помощью решения, работающего *быстрее*, чем $O(E \log V)$, если входной граф гарантированно будет иметь веса ребер, которые лежат в небольшом целочисленном диапазоне [0..100]? Является ли потенциальное ускорение значительным?

4.3.3. Алгоритм Прима

Алгоритм Роберта Клэя *Прима* сначала берет начальную вершину (для простоты мы возьмем вершину 0), помечает ее как «выбранную» и помещает пару параметров в приоритезированную очередь. Эта пара параметров – вес w и другая вершина u ребра $0 \rightarrow u$, которая еще не «выбрана». Эти пары сортируются в очереди с приоритетами по увеличению их веса, а в случае равенства весов – по увеличению номера вершины. Затем алгоритм Прима «жадно» выбирает с начала очереди с приоритетами пару (w, u), которая имеет минимальный вес w, если конечная вершина этого ребра, u, не была выбрана ранее. Это делается для предотвращения появления цикла. Если эта пара (w, u) удовлетворяет всем перечисленным условиям, то вес w добавляется к стоимости MST, u помечается как «выбранная», и пара (w', v) каждого ребра $u \rightarrow v$ с весом w', который соответствует u, помещается в приоритезированную очередь, если v не была выбрана ранее. Этот процесс повторяется до тех пор, пока приоритезированная очередь не станет пустой. Длина кода примерно такая же, как у алгоритма Краскала, и также выполняется в O (однократная обработка каждого ребра × стоимость постановки в очередь / удаления из очереди) = $O(E \times \log E) = O(E \log V)$.

```
vi taken;                                  // глобальный логический флаг, чтобы избежать цикла
priority_queue<ii> pq;                     // приоритезированная очередь, чтобы помочь выбрать
                                                                  // более короткие ребра
                                // Примечание: по умолчанию для C++ STL priority_queue - это
                                                        // невозрастающая пирамида (max heap)
void process(int vtx) {        // мы используем знак -ve, чтобы изменить порядок сортировки
  taken[vtx] = 1;
  for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
    ii v = AdjList[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
} }                             // сортируем по (увел.) веса, потом по (увел.) идентификатора

// внутри int main()---предположим, что граф хранится в AdjList, pq пуст
  taken.assign(V, 0);                                // ни одна вершина не выбирается в начале
  process(0);           // выберите вершину 0 и обработайте все ребра, входящие в вершину 0
  mst_cost = 0;
  while (!pq.empty()) {         // повторяйте, пока не будут выбраны V вершин (E=V-1 ребро)
    ii front = pq.top(); pq.pop();
    u = -front.second, w = -front.first;                      // снова инвертируйте id и вес
    if (!taken[u])                                          // мы еще не связали эту вершину
```

```
    mst_cost += w, process(u);          // берем u, обрабатываем все ребра, входящие в u
}                                        // каждое ребро находится в pq только один раз!
printf("Стоимость MST = %d (алгоритм Прима)\n", mst_cost);
```

На рис. 4.12 показано пошаговое выполнение алгоритма Прима на графе, изображенном на рис. 4.10 (в крайнем левом углу). Сравните рис. 4.12 с рис. 4.11, чтобы понять сходство и различия между алгоритмами Краскала и Прима.

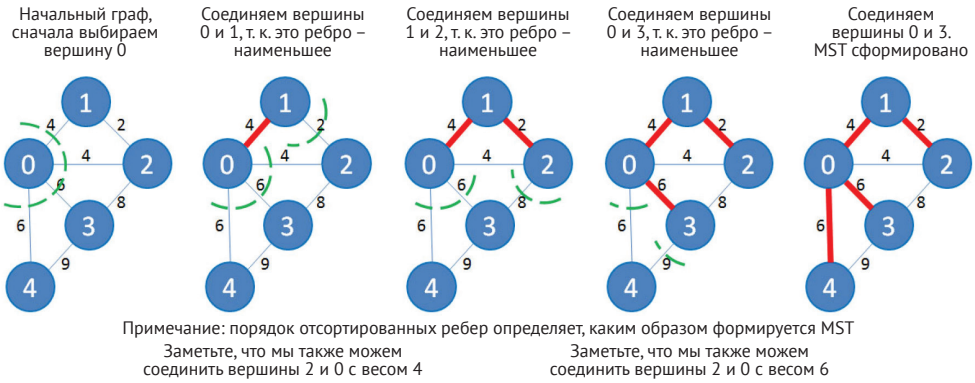


Рис. 4.12 ❖ Динамическое изображение алгоритма Прима для графа, приведенного на рис. 4.10 (слева)

Инструментальное средство визуализации: www.comp.nus.edu.sg/~stevenha/visualization/mst.html

Файл исходного кода: *ch4_03_kruskal_prim.cpp/java*

4.3.4. Другие варианты применения

Варианты основной задачи MST представляют особый интерес. В этом разделе мы рассмотрим некоторые из них.

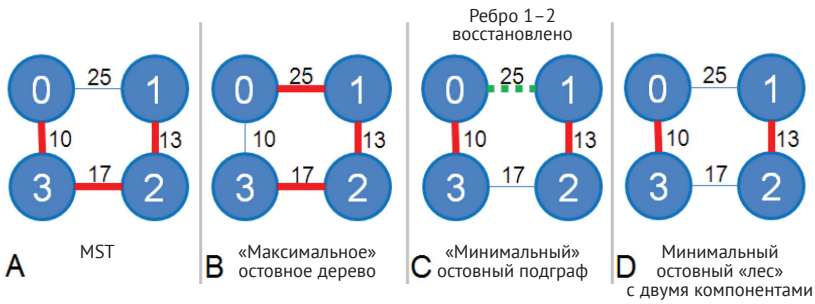


Рис. 4.13 ❖ Слева направо: MST, «максимальное» остовное дерево, «минимальный» остовный подграф, минимальный остовный «лес»

«Максимальное» остовное дерево

Это простой вариант, где мы хотим получить максимальное вместо минимального остовного дерева, например UVa 1234 – RACING (обратите внимание, что условия этой задачи сформулированы так, что она не выглядит как задача нахождения минимального остовного дерева, MST). На рис. 4.13B мы видим пример максимального остовного дерева.

Сравните его с соответствующим MST (рис. 4.13A).

Решение для этого варианта очень простое: немного изменив алгоритм Краскала, теперь мы просто сортируем ребра по *невозрастанию* веса.

«Минимальный» остовный подграф

В этом варианте мы не начинаем с чистого листа. Некоторые ребра в данном графе восстановлены и должны быть включены в решение как его часть, например UVa 10147 – Highways.

Для начала заметим, что эти ребра, взятые по умолчанию, могут формировать структуру, не являющуюся деревом. Наша задача – продолжить выбор оставшихся ребер (при необходимости), чтобы связать граф с наименьшими затратами. Результирующий охватывающий подграф может не являться деревом; и даже если он является деревом, он может не быть MST. Вот почему мы заключаем термин «минимальный» в кавычки и используем термин «подграф», а не «дерево». На рис. 4.13C мы видим пример, когда одно ребро 0–1 уже зафиксировано. Фактическое значение веса MST равно 10 + 13 + 17 = 40, при этом в MST отсутствует ребро 0–1 (рис. 4.13.A). Однако правильное решение для этого примера (25) + 10 + 13 = 48, в котором используется ребро 0–1.

Решение для данного варианта простое. После выбора и включения в MST всех фиксированных ребер и их стоимости мы продолжаем запускать алгоритм Краскала на оставшихся свободных ребрах, пока у нас не получится остовный подграф (или остовное дерево).

Минимальный «остовный лес»

В этом варианте мы хотим сформировать лес из K компонентов связности (K поддеревьев) таким способом, чтобы стоимость результирующего дерева была минимальной; при этом K заранее заданы в описании задачи, например UVa 10369 – Arctic Networks. На рис. 4.13A мы видим, что вес MST для этого графа будет 10 + 13 + 17 = 40. Но если мы довольны связующим лесом с двумя компонентами связности, то решением будет являться просто 10 + 13 = 23 на рис. 4.13D. То есть мы опускаем ребро 2–3 с весом 17, которое соединило бы эти две компоненты в одно остовное дерево, если это ребро было бы выбрано.

Получить минимальный остовный лес просто. Запустите алгоритм Краскала в обычном режиме, но как только количество подключенных компонент станет равным требуемому заранее определенному числу K, мы сможем завершить работу алгоритма.

Второе лучшее остовное дерево

Иногда альтернативные решения важны. В контексте поиска MST нам может потребоваться не только MST, но и второе лучшее остовное дерево (Spanning Tree, ST), если MST не работает, например UVa 10600 – ACM contest and blackout. На рис. 4.14 показано MST (слева) и второе лучшее ST (справа). Мы можем видеть, что второе лучшее ST на самом деле является MST с разницей всего в два ребра, то есть одно ребро берется из MST, а другое хордовое ребро[1] добавляется в MST. Здесь ребро 3–4 вынимается, а ребро 1–4 добавляется в граф.

Решением для этого варианта является модифицированный алгоритм Краскала: отсортируйте ребра за время $O(E \log E) = O(E \log V)$, затем найдите MST, используя алгоритм Краскала, за время $O(E)$. Затем для каждого ребра в MST (в MST есть не более $V - 1$ ребер) временно пометьте его так, чтобы его нельзя было выбрать, потом попробуйте снова найти MST за $O(E)$, но теперь исключая это *помеченное* ребро. Обратите внимание, что нам не нужно повторно сортировать ребра в этой точке. Лучшее остовное дерево, найденное в результате этого процесса, является вторым лучшим остовным деревом. На рис. 4.15 этот алгоритм показан для данного графа. В целом алгоритм работает за время O (однократная сортировка ребер + построение исходного MST + поиск второго лучшего ST) = $O(E \log V + E + VE) = O(VE)$.

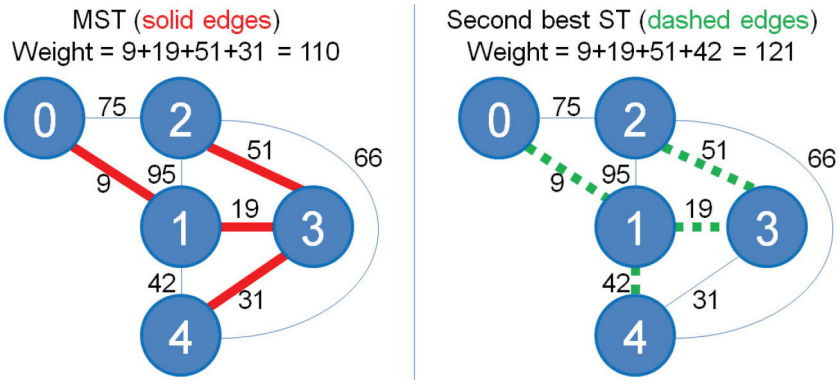


Рис. 4.14 ❖ Второе лучшее остовное дерево (из UVa 10600 [47])

Минимакс (и максимин)

Задача минимакса – это задача нахождения минимума максимального веса ребер среди всех возможных путей между двумя вершинами i и j. Стоимость пути от i до j определяется максимальным весом ребер вдоль этого пути. Среди всех этих возможных путей от i до j выберите тот, у которого максимальный вес ребра минимален. Обратная задача (о максиминном пути) определяется аналогично.

---

[1] Хордовое ребро определяется как ребро в графе G, которое не выбрано в MST графа G.

Задача минимакса между вершинами *i* и *j* может быть решена путем моделирования ее как задачи MST. Обоснованием такой модели является тот факт, что в решении задачи предпочитается путь с малым весом отдельных ребер, даже если путь длиннее с точки зрения количества задействованных вершин/ребер, тогда наличие MST (с использованием алгоритма Краскала или Прима) данного взвешенного графа является правильным шагом. MST является связным, таким образом обеспечивая путь между любой парой вершин. Итак, решение задачи минимакса – это нахождение максимального веса ребер вдоль уникального пути между вершинами *i* и *j* в этом MST.

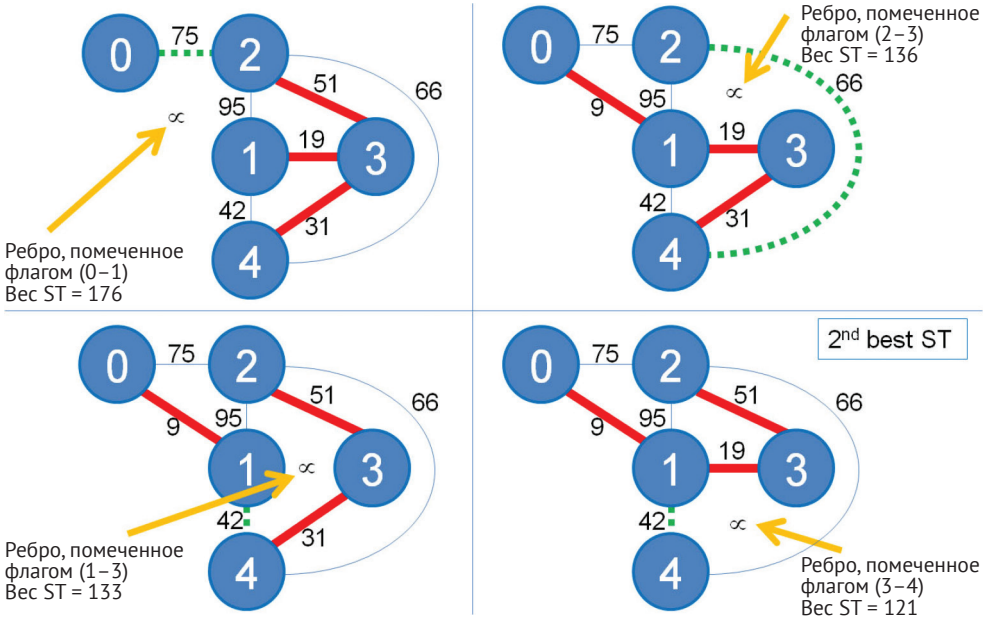


Рис. 4.15 ❖ Поиск второго лучшего остовного дерева MST

Общая временная сложность для данного решения равна O(построение MST + один обход по результирующему дереву). Поскольку в дереве $E = V - 1$, любой обход по дереву является просто $O(V)$. Таким образом, сложность этого подхода составляет $O(E \log V + V) = O(E \log V)$.

На рис. 4.16 (слева) показан тестовый пример для UVa 10048 – Audiophobia. У нас есть граф с 7 вершинами и 9 ребрами. Шесть выбранных ребер MST показаны сплошными жирными линиями на рис. 4.16 (справа). Теперь, если нас попросят найти минимаксный путь между вершинами 0 и 6 на рис. 4.16 (справа), мы просто пройдем MST от вершины 0 до 6. Будет только один путь, путь 0–2–5–3–6. Максимальный вес ребра, найденный вдоль пути, равен минимальной стоимости: 80 (из-за ребра 5–3).

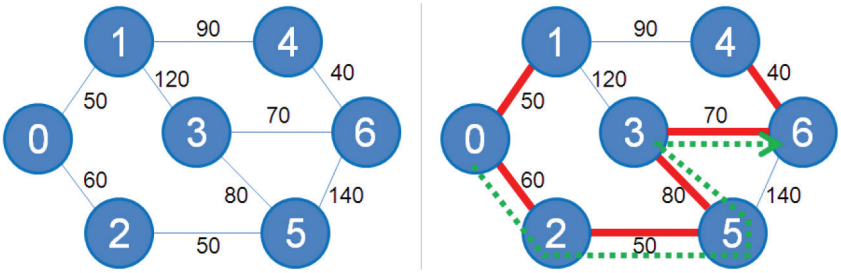


Рис. 4.16 ❖ Минимакс (UVa 10048 [47])

Упражнение 4.3.4.1. Решите пять вариантов задачи MST, описанных выше, используя алгоритм Прима. Какой вариант (какие варианты) являе(ю)тся / не являе(ю)тся дружественным(и) для алгоритма Прима?

Упражнение 4.3.4.2*. Существуют лучшие решения для задачи построения второго лучшего ST, показанной выше. Решите эту задачу с помощью решения, которое лучше, чем $O(VE)$. Подсказки: вы можете решить это упражнение либо с использованием наименьшего общего предка (Lowest Common Ancestor, LCA), либо с использованием системы непересекающихся множеств (Union-Find Disjoint Sets).

Замечания о задачах на построение минимального остовного дерева на олимпиадах по программированию

Чтобы решить многие задачи на построение MST на современных олимпиадах по программированию, мы можем использовать только алгоритм Краскала и пропустить алгоритм Прима (или другие алгоритмы построения MST). Алгоритм Краскала – по нашему мнению, лучший алгоритм для решения задач, связанных с MST, на олимпиадах по программированию. Это легко понять и хорошо увязывается со структурой данных системы непересекающихся множеств (Union-Find Disjoint Sets) (см. раздел 2.4.2), которая используется для проверки циклов. Однако поскольку мы любим выбирать между различными вариантами решений, то также включили в книгу обсуждение другого популярного алгоритма для построения MST – алгоритма Прима.

По умолчанию (и наиболее распространенное) использование алгоритма Краскала (или Прима) заключается в решении задачи минимального ST (UVa 908, 1174, 1208, 11631), но возможен и простой вариант «максимального» ST (UVa 1234, 10842). Обратите внимание, что в большинстве (если не все) задач, связанных с MST, на олимпиадах по программированию требуется определить только *уникальную* стоимость MST, а не само MST. Так происходит потому, что могут быть разные MST с одинаковыми минимальными затратами – обычно бывает слишком сложно написать специальную программу проверки, чтобы оценить правильность результата решения на основании таких неуникальных выходных данных.

Другие варианты MST, обсуждаемые в этой книге, такие как «минимальный» остовный подграф (UVa 10147, 10397), минимальный «остовный лес» (UVa 1216,

10369), второе лучшее остовное дерево (UVa 10462, 10600), минимакс/максимин (UVa 534) 544, 10048, 10099), на самом деле встречаются редко.

В настоящее время более общей тенденцией для задач, связанных с построением MST, является то, что авторы задачи формулируют условия задач, связанных с MST, так, что не ясно, что эта задача на самом деле является задачей на построение MST (например, UVa 1216, 1234, 1235). Однако как только участники поймут это, задача может оказаться легкой.

Обратите внимание, что существуют более сложные задачи на построение MST, для решения которых может потребоваться более сложный алгоритм, например задача об ориентированном дереве, дерево Штейнера, MST с ограничением степени, *k*-MST и т. д.

Задачи по программированию, связанные с построением минимального остовного дерева

- Стандартные задачи
  1. UVa 00908 – Re-connecting… (стандартная задача построения MST)
  2. *UVa 01174 – IP-TV* (LA 3988, Юго-Западная Европа'07, MST, классика, просто нужен код для сопоставления названий городов с индексами)
  3. UVa 01208 – Oreon (LA 3171, Манила'06, MST)
  4. UVa 01235 – Anti Brute Force Lock (LA 4138, Джакарта'08, основной проблемой является построение MST)
  5. UVa 10034 – Freckles (стандартная задача построения MST)
  6. **UVa 11228 – Transportation System** * (разделите выходные данные на короткие и длинные ребра)
  7. **UVa 11631 – Dark Roads** * (вычислите вес ((все ребра графа) – (все ребра MST)))
  8. UVa 11710 – Expensive Subway (выведите «Impossible», если граф все еще не является связным после запуска MST)
  9. UVa 11733 – Airports (сохраняйте стоимость при каждом обновлении)
  10. **UVa 11747 – Heavy Cycle Edges** * (суммируйте веса самых тяжелых ребер на цикле)
  11. UVa 11857 – Driving Range (определите вес последнего ребра, добавленного к MST)
  12. IOI 2003 – Trail Maintenance (используйте эффективное инкрементальное MST)
- Варианты
  1. UVa 00534 – Frogger (минимакс, также решается с помощью алгоритма Флойда–Уоршелла)
  2. UVa 00544 – Heavy Cargo (максимин, также решается с помощью алгоритма Флойда–Уоршелла)
  3. *UVa 01160 – X-Plosives* (подсчитайте количество невыбранных ребер, алгоритм Краскала)
  4. UVa 01216 – The Bug Sensor Problem (LA 3678, Гаосюн'06, минимальный «остовный лес»)

5. UVa 01234 – RACING (LA 4110, Сингапур'07, максимальное остовное дерево)
6. **UVa 10048 – Audiophobia** * (минимакс, см. обсуждение выше)
7. UVa 10099 – Tourist Guide (максимин, также решается с помощью алгоритма Флойда–Уоршелла)
8. UVa 10147 – Highways (минимальный остовный подграф)
9. **UVa 10369 – Arctic Networks** * (минимальный «остовный лес»)
10. UVa 10397 – Connect the Campus («минимальный» остовный подграф)
11. UVa 10462 – Is There A Second… (второе лучшее остовное дерево)
12. **UVa 10600 – ACM Contest and…** * (второе лучшее остовное дерево)
13. UVa 10842 – Traffic Flow (найдите минимальное взвешенное ребро в «максимальном» остовном дереве)

Известные авторы алгоритмов

Роберт Андре Тарьян (род. 1948) – американский ученый, специалист в области теории вычислительных систем. Он изобрел несколько важных алгоритмов на графах. Наиболее важным в контексте олимпиадного программирования является алгоритм нахождения **компонент сильной связности** в ориентированном графе и алгоритм поиска **мостов и точек сочленения** в неориентированном графе (обсуждается в разделе 4.2 вместе с другими вариантами DFS, изобретенными им и его коллегой [63]). Он также изобрел алгоритм поиска наименьшего общего предка в дереве (алгоритм Тарьяна поиска наименьшего общего предка), изобрел структуру данных сплей-дерево (splay tree) и проанализировал временную сложность структуры данных системы непересекающихся множеств (Union-Find Disjoint Sets) (см. раздел 2.4.2).

Джон Эдвард Хопкрофт (род. 1939) – американский ученый, специалист в области теории вычислительных систем. Он профессор компьютерных наук в Корнелльском университете. За фундаментальные достижения в разработке и анализе алгоритмов и структур данных Хопкрофт получил премию Тьюринга – самую престижную награду в этой области, которая часто в разговорах упоминается как «Нобелевская премия по вычислительной технике» (совместно с Робертом Андре Тарьяном в 1986 году). Наряду с этой совместной работой с Тарьяном над плоскими графами (и некоторыми другими алгоритмами на графах, такими как **поиск точек сочленения / мостов с использованием DFS**) он также известен как автор **алгоритма Хопкрофта–Карпа** для поиска максимального паросочетания в двудольных графах, изобретенного вместе с Ричардом Мэннингом Карпом [28] (см. раздел 9.12).

Джозеф Бернард Краскал-младший (1928–2010) был американским ученым, специалистом в области теории вычислительных систем. Его самая известная работа, связанная с олимпиадным программированием, – это алгоритм Краскала для построения минимального остовного дерева (MST) взвешенного графа. У MST есть интересные применения в области построения и ценообразования сетей связи.

Роберт Клэй Прим (род. 1921) – американский математик и специалист в области теории вычислительных систем. В 1957 году в Bell Laboratories он разработал алгоритм Прима для решения задачи MST. Прим знал Краскала, поскольку они вместе работали в Bell Laboratories. Алгоритм Прима впервые был открыт в 1930 году Войцехом Ярником и позже заново изобретен Примом. Таким образом, алгоритм Прима иногда также называют алгоритмом Ярника–Прима.

Войцех Ярник (1897–1970) был чешским математиком. Он разработал алгоритм, теперь известный как алгоритм Прима. В эпоху быстрой и повсеместной публикации научных результатов, в наши дни, алгоритм Прима назывался бы алгоритмом Ярника, а не алгоритмом Прима.

Эдсгер Вайб Дейкстра (1930–2002) был голландским ученым, специалистом в области теории вычислительных систем. Одним из его известных вкладов в информатику является алгоритм поиска кратчайших путей, известный как **алгоритм Дейкстры** [10]. Ему очень не нравился оператор «GOTO», и, будучи его ярым противником, он повлиял на широкое осуждение «GOTO» и его замену на структурированные управляющие конструкции. Одна из его знаменитых компьютерных фраз: «two or more, use a for» (два или больше – используй «for»).

Ричард Эрнест Беллман (1920–1984) был американским ученым, специализировавшимся в области прикладной математики. Помимо изобретения **алгоритма Форда–Беллмана** для поиска кратчайших путей в графах, содержащих ребра с отрицательным весом (и, возможно, цикл с отрицательным весом), Ричард Беллман более известен своим изобретением метода динамического программирования в 1953 году.

Лестер Рэндольф Форд-младший (род. 1927) – американский математик, специализирующийся на задачах сетевого потока. В статье Форда, опубликованной в 1956 году совместно с Фалкерсоном, о задаче определения максимального потока в транспортной сети и о методе Форда–Фалкерсона для ее решения была сформулирована теорема о максимальном потоке (максимальный поток равен минимальному сечению).

Дельберт Рэй Фалкерсон (1924–1976) был математиком, разработавшим **метод Форда–Фалкерсона** – алгоритм для решения задачи вычисления максимального потока в сетях. В 1956 году он опубликовал свою статью о методе Форда–Фалкерсона совместно с Лестером Р. Фордом.

4.4. Нахождение кратчайших путей из заданной вершины во все остальные (Single – Source Shortest Paths, SSSP)

4.4.1. Обзор

Задача: для заданного *взвешенного* графа G и исходной вершины *s* каковы кратчайшие пути из *s* во все остальные вершины *G*?

Эта задача называется задачей о поиске *кратчайших путей из заданной вершины* (Single-Source Shortest Paths, SSSP)[1] во все остальные вершины на взвешенном графе (Single-Source Shortest Paths, SSSP). Эта классическая задача в теории графов имеет множество реальных вариантов применения. Например, мы можем смоделировать город, в котором живем, в виде графа. Вершины – это дорожные развязки (перекрестки). Ребра графа – дороги. Время, затраченное на прохождение одной дороги, – это вес соответствующего ребра. В настоящее время вы находитесь на определенном перекрестке. Каково самое короткое время, необходимое, чтобы достичь другого заданного перекрестка?

Существуют эффективные алгоритмы для решения этой задачи SSSP. Если граф невзвешенный (или все ребра имеют одинаковый или постоянный вес), мы можем использовать эффективный алгоритм с временной сложностью $O(V + E)$ BFS, приведенный ранее в разделе 4.2.2. Для общего случая взвешенного графа BFS не работает правильно, и мы должны использовать такие алгоритмы, как алгоритм Дейкстры с временной сложностью $O((V + E)\log V)$ или алгоритм Форда–Беллмана с временной сложностью $O(VE)$. Это разные алгоритмы, их обсуждение приведено ниже.

Упражнение 4.4.1.1*. Докажите, что кратчайший путь между двумя вершинами i и j в графе G, не имеющий цикла с отрицательным весом, должен быть простым (ациклическим).

Упражнение 4.4.1.2*. Докажите, что отрезки кратчайших путей от u до v также являются кратчайшими.

4.4.2. SSSP на невзвешенном графе

Вернемся к разделу 4.2.2. Тот факт, что BFS посещает вершины графа слой за слоем из исходной вершины (см. рис. 4.3), делает BFS естественным выбором для решения задач SSSP на невзвешенных графах. В невзвешенном графе расстояние между двумя соседними вершинами, связанными с ребром, составляет просто одну единицу. Следовательно, количество слоев вершины, которое мы видели в разделе 4.2.2, – это как раз кратчайшая длина пути от начальной вершины до этой вершины.

Например, на рис. 4.3 кратчайший путь от вершины 5 до вершины 7 равен 4, поскольку 7 находится в четвертом слое в последовательности обхода вершин BFS, начиная с вершины 5.

В некоторых задачах программирования требуется, чтобы мы воспроизвели фактический кратчайший путь, а не только нашли длину кратчайшего пути. Например, на рис. 4.3 кратчайший путь от 5 до 7 равен $5 \rightarrow 1 \rightarrow 2 \rightarrow 3$

---

[1] Эта общая задача SSSP также может быть использована для решения следующих вариантов задач: 1) задача нахождения кратчайшего пути (Single-Pair, SP или Single-Source Single-Destination), где заданы начальная и конечная вершины, и 2) задача SP с одной точкой назначения, в которой мы просто меняем роль начальной/конечной вершин.

→ 7. Восстановить последовательность переходов этого пути просто, если мы сохраняем остовное дерево (фактически это дерево BFS) кратчайшего пути[1]. Это легко сделать, используя вектор целых чисел `vi p`. Каждая вершина *v* помнит своего родителя *u* (`p[v] = u`) в остовном дереве кратчайшего пути. В этом примере вершина 7 запоминает 3 как своего родителя, вершина 3 запоминает 2, вершина 2 запоминает 1, вершина 1 запоминает 5 (начальную вершину). Чтобы восстановить фактический кратчайший путь, мы можем выполнить простую рекурсию от последней вершины 7 до тех пор, пока не достигнем начальной вершины 5. Измененный код BFS (просмотрите комментарии) относительно прост:

```
void printPath(int u) {                                  // извлечение информации из 'vi p'
  if (u == s) { printf("%d", s); return; }         // базовый случай, в начальной вершине s
  printPath(p[u]);    // рекурсивно: чтобы установить формат выходных данных: s -> ... -> t
  printf(" %d", u); }

// внутри int main()
  vi dist(V, INF); dist[s] = 0;           // расстояние от начальной вершины s до s равно 0
  queue<int> q; q.push(s);
  vi p;                                  // дополнение: предшественник / родительский вектор
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] == INF) {
        dist[v.first] = dist[u] + 1;
        p[v.first] = u;                  // дополнение: родителем вершины v.first является u
      q.push(v.first);
  } } }
  printPath(t), printf("\n");                // дополнение: вызываем printPath из вершины t
```

Файл исходного кода: *ch4_04_bfs.cpp/java*

Мы хотели бы отметить, что в последнее время задачи, связанные с BFS, предлагавшиеся для решения на олимпиадах по программированию, уже не формулируются как простые задачи SSSP, а пишутся гораздо более креативно.

Возможные варианты таких задач: BFS на неявном графе (2D-сетка: UVa 10653 или 3D-сетка: UVa 532), BFS с выводом фактического кратчайшего пути (UVa 11049), BFS на графе с некоторыми заблокированными вершинами (UVa 10977), BFS с несколькими исходными вершинами (UVa 11101, 11624), BFS и единственной конечной вершиной – такие задачи решаются путем взаимозамены начальной и конечной вершин (UVa 11513), BFS с нетривиальными состояниями (UVa 10150) – больше таких задач приведено в разделе 8.2.3 – и т. д. Поскольку существует много интересных вариантов BFS, мы рекомендуем читателям постараться решить как можно больше задач по программированию, приведенных в этом разделе.

---

[1] Восстановление кратчайшего пути не показано в следующих двух подразделах (алгоритмы Дейкстры/Форда–Беллмана), но их идея аналогична показанной здесь (и с воспроизведением решения DP в разделе 3.5.1).

Упражнение 4.4.2.1. Мы можем запустить BFS из > 1 исходной вершины. Мы называем этот вариант задачей о кратчайших путях из нескольких исходных вершин (Multi-Sources Shortest PathsMSSP) на невзвешенных графах. Попробуйте решить UVA 11101 и 11624, чтобы получить представление о MSSP на невзвешенном графе. Наивное решение – вызов BFS несколько раз. Если имеется k возможных источников, такое решение будет иметь временную сложность $O(k \times (V + E))$. Вы можете предложить вариант лучше?

Упражнение 4.4.2.2. Предложите простое улучшение приведенного выше кода BFS, если вас попросят решить задачу «Кратчайший путь из исходной вершины к *единственной конечной вершине*» (Single-Source *Single-Destination* Shortest Path) на невзвешенном графе. Заданы исходная *и* конечная вершины.

Упражнение 4.4.2.3. Объясните, почему мы можем использовать BFS для решения задачи SSSP на взвешенном графе, где *все ребра* имеют одинаковый вес C.

Упражнение 4.4.2.4*. Дана карта с навигационной сеткой с размерами R×C, как показано ниже. Определите кратчайший путь от любой ячейки, помеченной как «A», до любой ячейки, помеченной как «B». Вы можете прокладывать путь только через ячейки, помеченные символом «.», в направлении NESW (считаются за *одну* единицу), и ячейки, обозначенные алфавитом «A»–«Z» (считаются за *ноль* единиц)! Вы можете предложить решение с временной сложностью $O(R \times C)$?

```
.....................CCCC. // Для данного тестового примера ответ: 13 единиц
AAAAA................CCCC. // Решение: двигайтесь сначала на восток от
AAAAA.AAA............CCCC. // самой правой 'A' до самой левой 'C' в этом ряду
AAAAAAAAA....###....CCCC. // затем на юг от самой правой 'C' в этом ряду
AAAAAAAAA..............   // вниз
AAAAAAAAA..............   // к
.......DD..............BB // самой левой 'B' в этом ряду
```

4.4.3. SSSP на взвешенном графе

Если заданный граф является взвешенным, BFS не работает. Это связано с тем, что может быть «более длинный» путь (с точки зрения количества вершин и ребер, участвующих в пути), но он имеет меньший общий вес, чем «более короткий» путь, найденный BFS. Например, на рис. 4.17 кратчайший путь от исходной вершины 2 до вершины 3 идет не через прямое ребро 2 → 3 с весом 7, которое обычно находится, когда мы применяем BFS, а по «обходному» пути: 2 → 1 → 3 с меньшим общим весом 2 + 3 = 5.

Чтобы решить задачу SSSP на взвешенном графе, мы используем «жадный» алгоритм Эдсгера В. *Дейкстры*. Есть несколько способов реализовать этот классический алгоритм. Фактически оригинальная статья Дейкстры, описывающая этот алгоритм [10], не описывает конкретную реализацию.

Многие другие ученые-специалисты в области теории вычислительных систем предложили варианты реализации, основанные на оригинальной работе

Дейкстры. Здесь мы приводим один из самых простых вариантов реализации, который использует `priority_queue` в C++ STL (или `PriorityQueue` в Java). Это делается для того, чтобы минимизировать длину кода – необходимая вещь в олимпиадном программировании.

Данный вариант реализации алгоритма Дейкстры поддерживает **приоритетную очередь** `pq`, в которой хранятся пары значений, содержащие информацию о вершинах. Первый и второй элементы пары – это расстояние вершины от источника и номер вершины соответственно. Эта `pq` сортируется по увеличению расстояния от начальной вершины и, в случае равенства, по номеру вершины. Данная реализация отличается от другой реализации алгоритма Дейкстры, которая использует функцию двоичной кучи, не поддерживающуюся во встроенных библиотеках[1].

Эта `pq` изначально содержит только один элемент: базовый случай (0, *s*), который является верным для исходной вершины. Затем этот вариант реализации алгоритма Дейкстры повторяет следующий процесс до тех пор, пока очередь `pq` не станет пустой: он «жадно» извлекает информацию о паре вершин (*d*, *u*) из начала `pq`. Если расстояние до *u* от начальной вершины, записанной в параметре *d*, больше, чем `dist[u]`, мы игнорируем *u*; в противном случае мы обрабатываем *u*. Основание для этой специальной проверки показано ниже.

Когда этот алгоритм обрабатывает *u*, он пытается выполнить операцию `relax`[2] для всех соседей *v* вершины *u*. Каждый раз, когда он выполняет операцию `relax` на ребре $u \rightarrow v$, он помещает пару значений (более новое / более короткое расстояние до *v* от исходной вершины, *v*) внутрь `pq` и оставляет младшую пару значений (более старое / большее расстояние до *v* от исходной вершины, *v*) внутри `pq`. Это называется «ленивым удалением» (Lazy Deletion) и приводит к существованию более одного экземпляра одной и той же вершины в `pq` на разных расстояниях от источника. Вот почему у нас ранее была предусмотрена проверка, чтобы обработать только пару параметров с информацией о вершине, *первой извлеченной из очереди*, которая имеет правильное / более короткое расстояние (другие экземпляры той же вершины будут иметь устаревшее / более длинное расстояние). Код реализации алгоритма приведен ниже и очень похож на код BFS и код реализации алгоритма Прима, показанный в разделах 4.2.2 и 4.3.3 соответственно.

```
vi dist(V, INF); dist[s] = 0;                          // INF = 1B во избежание переполнения
 priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
 while (!pq.empty()) {                                                    // основной цикл
   ii front = pq.top(); pq.pop();               // "жадный": получить непосещенную вершину
                                                                  // с наикратчайшим путем
   int d = front.first, u = front.second;
   if (d > dist[u]) continue;                                  // это очень важная проверка
   for (int j = 0; j < (int)AdjList[u].size(); j++) {
```

---

[1] Для обычной реализации алгоритма Дейкстры (например, см. [7, 38, 8]) требуется операция `heapDecreaseKey` в двоичной куче DS, которая не поддерживается встроенной реализацией очереди приоритетов в C++ STL или Java API. Вариант реализации алгоритма Дейкстры, рассмотренный в этом разделе, использует только две основные операции приоритетной очереди: `enqueue` и `dequeue`.

[2] Операция `relax(u, v, w_u_v)` устанавливает `dist[v] = min(dist[v], dist[u] + w_u_v)`.

```
    ii v = AdjList[u][j];                                  // все исходящие вершины из u
    if (dist[u] + v.second < dist[v.first]) {
      dist[v.first] = dist[u] + v.second;                          // операция relax
      pq.push(ii(dist[v.first], v.first));
} } }      // этот вариант может привести к дублированию элементов в приоритетной очереди
```

Файл исходного кода: *ch4_05_dijkstra.cpp/java*

На рис. 4.17 мы показываем пошаговый пример запуска этого варианта реализации алгоритма Дейкстры на небольшом графе и $s = 2$. Обратите внимание на содержимое `pq` на каждом шаге.

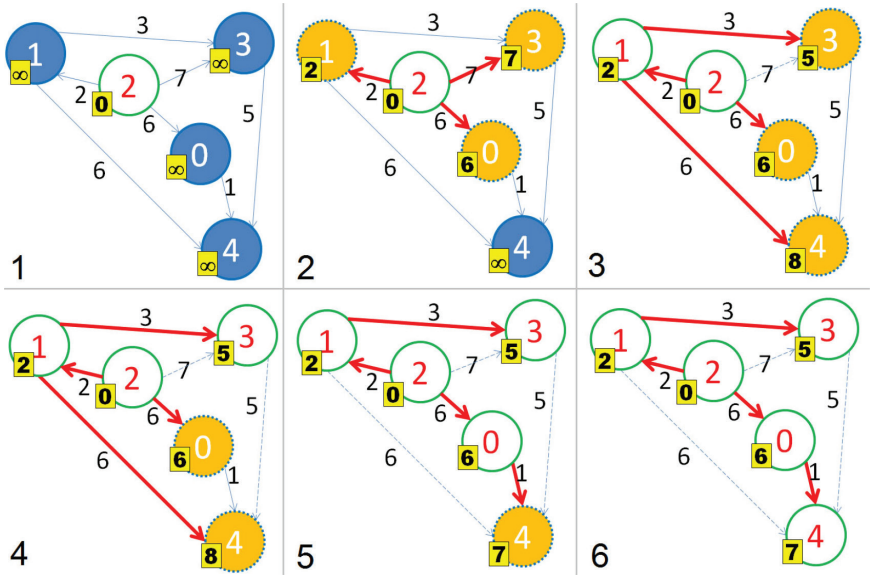


Рис. 4.17 ❖ Воспроизведение шагов алгоритма Дейкстры на взвешенном графе (из UVa 341 [47])

1. Вначале `dist[s] = dist[2] = 0`, `priority_queue pq` содержит пару значений {(0,2)}.
2. Исключаем пару значений с информацией о вершинах (0,2) из `pq`. Выполняем операцию `relax` на ребрах, инцидентных вершине 2, чтобы получить `dist[0] = 6`, `dist[1] = 2` и `dist[3] = 7`. Теперь `pq` содержит три пары значений {(2,1), (6,0), (7,3)}.
3. Среди необработанных пар значений в `pq` пара значений (2,1) находится в начале `pq`. Мы удаляем (2,1) и выполняем операцию `relax` на ребрах, инцидентных вершине 1, чтобы получить `dist[3] = min(dist[3], dist[1] + weight(1,3)) = min(7, 2 + 3) = 5` и `dist[4] = 8`. Теперь `pq` содержит три пары значений {(5,3), (6,0), (7,3), (8,4)}. Обратите внимание, что у нас есть *две пары* значений, относящиеся к вершине 3 в нашей очереди `pq`, с увеличением расстояния от исходной вершины *s*. Мы не сразу удаляем нижнюю пару (7,3) из `pq` и полагаемся на будущие итерации работающего алгоритма

Дейкстры, чтобы выбрать правильный вариант с минимальным расстоянием позже, то есть пару значений (5,3). Это называется «ленивым удалением».

4. Мы удаляем из очереди пару (5,3) и пытаемся выполнить операцию `relax(3,4,5)`, то есть 5 + 5 = 10. Но `dist[4] = 8` (что следует из пути 2–1–4), поэтому `dist[4]` оставляем без изменений. Теперь `pq` содержит три пары значений {(6,0), (7,3), (8,4)}.
5. Удаляем пару значений (6,0) из очереди и выполняем `relax(0,4,1)`, при этом `dist[4] = 7` (более короткий путь из 2 в 4, уже 2–0–4 вместо 2–1–4). Теперь `pq` содержит три пары значений {(7,3), (7,4), (8,4)}, при этом имеется две пары значений, относящихся к вершине 4. Это еще один случай «ленивого удаления».
6. Теперь пару значений (7,3) можно игнорировать, так как мы знаем, что его `d > dist[3]` (т. е. 7 > 5). Только на итерации 6 выполняется фактическое удаление младшей пары (7,3) (а не на итерации 3, выполненной ранее). Поскольку окончательное удаление пары (7,3) отложено до итерации 6, младшая пара значений (7,3) находится в легкой позиции для стандартного удаления с временной сложностью $O(\log n)$, выполняемой на неубывающей пирамиде (min heap): а именно в корне неубывающей пирамиды, то есть в начале приоритетной очереди.
7. Затем пара значений (7,4) обрабатывается, как и раньше, но ничего не меняется. Теперь `pq` содержит только {(8,4)}.
8. Наконец, пара значений (8,4) снова игнорируется, так как `d > dist[4]` (т. е. 8 > 7). Этот вариант реализации алгоритма Дейкстры останавливается на данном шаге, так как очередь `pq` теперь пуста.

Пример использования: UVa 11367 – Full Tank?

Сокращенная формулировка условий задачи: пусть задана *длина* связного взвешенного графа, в котором хранится длина дороги между E парами городов i и j ($1 \leqslant V \leqslant 1000$, $0 \leqslant E \leqslant 10\,000$), цена $p[i]$ топлива в каждом городе i, и вместимость топливного бака c автомобиля ($1 \leqslant c \leqslant 100$). Нужно определить самую дешевую стоимость поездки от начального города s до конечного города e с использованием автомобиля с запасом топлива c. Все автомобили используют одну единицу топлива на единицу расстояния и начинают путешествие с пустого топливного бака.

На примере этой задачи мы хотим обсудить важность моделирования графа. Явно заданный граф в этой задаче является взвешенным графом дорожной сети. Однако мы не можем решить эту задачу только с помощью данного графа. Это связано с тем, что для определения состояния[1] в данной задаче требуется не только текущее местоположение (город), но и уровень топлива в этом месте. В противном случае мы не сможем определить, достаточно ли в баке автомобиля топлива для поездки по определенной дороге (поскольку мы не можем заправиться в середине дороги). Поэтому используем пару значений для представления состояния: (местоположение, топливо), и, таким образом, общее количество вершин модифицированного графа многократно увеличи-

[1] Напомним: состояние – это подмножество параметров задачи, с помощью которых можно кратко описать задачу.

вается: с 1000 вершин до 1000 × 100 = 100 000 вершин. Мы назовем измененный граф «Состояние–Расстояние».

На графе «Состояние–Расстояние» начальная вершина – это состояние (*s*, 0) (в начальном городе *s* с пустым топливным баком), а конечные вершины – это состояния (*e*, *любое*) – в конечном городе *e* с любым уровнем топлива, находящимся в интервале между [0..*c*]. В графе «Состояние–Расстояние» есть два типа ребер: 0 – взвешенное ребро, которое идет от вершины (x, $fuel_x$) к вершине (y, $fuel_x$ – длина(x, y)), если у автомобиля достаточно топлива, чтобы проехать от вершины графа x к вершине y, и $p[x]$ – взвешенное ребро, которое идет от вершины (x, $fuel_x$) к вершине (x, $fuel_x + 1$), если автомобиль может заправиться в вершине x одной единицей топлива (обратите внимание, что уровень топлива не может превышать емкость топливного бака c). Теперь запуск алгоритма Дейкстры на этом графе Состояние–Расстояние дает нам решение данной задачи (также см. раздел 8.2.3, где приводится дальнейшее обсуждение).

Упражнение 4.4.3.1. Модифицированный вариант реализации алгоритма Дейкстры, приведенный выше, может отличаться от того, что вы узнали из других книг (например, [7, 38, 8]). Проанализируйте, работает ли этот вариант за время, не превышающее $O((V + E)\log V)$ на различных типах взвешенных графов (см. также следующее упражнение 4.4.3.2*)?

Упражнение 4.4.3.2*. Построить граф с отрицательными весами ребер, но без цикла с отрицательным весом, который может значительно замедлить эту реализацию алгоритма Дейкстры.

Упражнение 4.4.3.3. Единственная причина, по которой этот вариант допускает дублирование вершин в приоритетной очереди, заключается в том, что он может использовать встроенную библиотеку с реализацией приоритетной очереди в неизмененном виде. Существует еще один альтернативный вариант реализации, который также требует минимального объема кода. Он использует `set`. Реализуйте этот вариант.

Упражнение 4.4.3.4. Приведенный выше исходный код использует `priority_queue< ii, vector<ii>, greater<ii> > pq`, чтобы сортировать пары целых чисел, увеличивая расстояние от источника *s*. Как мы можем достичь того же эффекта без определения оператора сравнения для `priority_queue`?

Подсказка: мы использовали аналогичный прием с реализацией алгоритма Краскала в разделе 4.3.2.

Упражнение 4.4.3.5. В упражнении 4.4.2.2 мы нашли способ ускорить решение задачи нахождения кратчайших путей, если нам заданы как начальная, так и конечная вершины. Можно ли использовать один и тот же прием ускорения для всех видов взвешенных графов?

Упражнение 4.4.3.6. Моделирование графа для UVa 11367, приведенного выше, преобразует задачу SSSP на взвешенном графе в задачу SSSP на взвешенном графе Состояние–Расстояние. Можем ли мы решить эту задачу с помощью метода DP? Если можем, то почему? Если не можем, то почему нет? Подсказка: прочитайте раздел 4.7.1.

4.4.4. SSSP на графе, имеющем цикл с отрицательным весом

Если в графе есть ребра отрицательного веса , типичная реализация алгоритма Дейкстры (например, [7, 38, 8]) может дать неправильный ответ. Однако вариант реализации алгоритма Дейкстры, показанный в разделе 4.4.3 выше, будет работать нормально, хотя и медленнее. Проверьте это на примере графа на рис. 4.18.

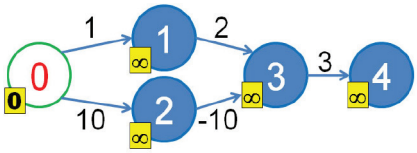


Рис. 4.18 ❖ Вес -ve

Это связано с тем, что вариант реализации алгоритма Дейкстры будет продолжать добавлять новую пару значений с информацией о вершинах графа в приоритетную очередь каждый раз, когда он выполняет операцию `relax`. Таким образом, если граф не имеет цикла отрицательного веса, вариант будет продолжать распространять эту пару значений с информацией о кратчайшем пути до тех пор, пока не будет больше возможна операция `relax` (что означает, что все кратчайшие пути из начальной вершины были найдены). Однако когда у нас имеется граф с циклом отрицательного веса, такой вариант – если он реализован, как показано в разделе 4.4.3 выше, – приведет к бесконечному циклу.

Пример: см. граф на рис. 4.19. Путь 1–2–1 – это цикл отрицательного веса. Вес этого цикла составляет 15 + (–42) = –27.

Чтобы решить задачу SSSP при потенциальном наличии цикла(ов) с отрицательным весом, необходимо использовать более общий (но более медленный) алгоритм Форда–Беллмана. Этот алгоритм был изобретен Ричардом Эрнестом Беллманом (пионером техники DP) и Лестером Рэндольфом Фордом-младшим (тем самым, кто изобрел метод Форда–Фалкерсона, о котором будет рассказано в разделе 4.6.2). Основная идея данного алгоритма проста: выполнить операцию `relax` для всех E ребер (в произвольном порядке) $V - 1$ раз.

Изначально `dist[s] = 0`. Если мы выполним операцию `relax` для ребра s → u, то `dist[u]` будет иметь правильное значение. Если затем выполнить операцию `relax` для ребра $u \rightarrow v$, то `dist[v]` также будет иметь правильное значение. Если мы выполним операцию `relax` для всех E ребер $V - 1$ раз, то должен быть правильно вычислен кратчайший путь от начальной вершины до самой дальней вершины (который будет простым путем с $V - 1$ ребрами). Основная часть кода реализации алгоритма Форда–Беллмана проще, чем код BFS и код реализации алгоритма Дейкстры:

```
vi dist(V, INF); dist[s] = 0;
 for (int i = 0; i < V - 1; i++)              // выполняем relax для всех E ребер V-1 раз
   for (int u = 0; u < V; u++)                        // эти два цикла = O(E), итого O(VE)
     for (int j = 0; j < (int)AdjList[u].size(); j++) {
```

```
      ii v = AdjList[u][j];                              // запишем SP здесь, если нужно
      dist[v.first] = min(dist[v.first], dist[u] + v.second);          // операция relax
    }
```

Временная сложность алгоритма Форда–Беллмана составляет $O(V^3)$, если граф хранится в виде матрицы смежности, или $O(VE)$, если граф хранится в виде списка смежности. Это объясняется тем, что если мы используем матрицу смежности, нам необходимо время $O(V^2)$, чтобы перебрать все ребра в нашем графе. В обоих случаях эта реализация будет работать намного медленнее, чем реализация алгоритма Дейкстры. Однако алгоритм Форда–Беллмана гарантирует отсутствие бесконечных циклов, даже если в данном графе есть цикл отрицательного веса. Фактически алгоритм Форда–Беллмана может быть использован для обнаружения наличия цикла отрицательного веса (например, в задаче UVa 558 – Wormholes), хотя такая задача SSSP является недостаточно определенной).

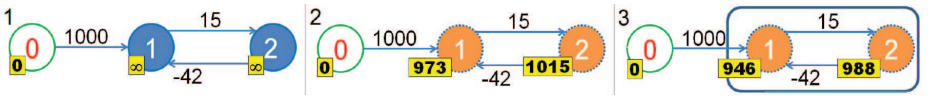


Рис. 4.19 ❖ Алгоритм Беллмана–Форда может обнаружить наличие цикла с отрицательным весом (на примере задачи UVa 558 [47])

В упражнении 4.4.4.1 мы доказываем, что после выполнения операции `relax` для всех E ребер $V - 1$ раз задача SSSP должна была быть решена, то есть мы не можем больше выполнять операцию `relax` для ребра. Как следствие: если мы все еще можем выполнить операцию `relax` для ребра, в нашем взвешенном графе должен быть отрицательный цикл.

Например, на рис. 4.19 (слева) мы видим простой граф с циклом отрицательного веса. После 1 прохода `dist[1] = 973` и `dist[2] = 1015` (посередине). После прохождения $V - 1 = 2$ проходов `dist[1] = 946` и `dist[2] = 988` (справа). Поскольку существует цикл с отрицательным весом, мы все еще можем сделать это снова (и снова), то есть мы все еще можем выполнить операцию `relax` для `dist[2] = 946 + 15 = 961`. Это ниже, чем текущее значение `dist[2] = 988`. Наличие цикла отрицательного веса приводит к тому, что для вершин, достижимых из этого цикла, нет информации о корректных путях. Это происходит потому, что можно просто пройти этот цикл с отрицательным весом бесконечное число раз, чтобы все вершины, достижимые из этого цикла, получили длину кратчайшего пути, равную минус бесконечности. Код для проверки на цикл отрицательного веса прост:

```
// после выполнения алгоритма O(VE) Беллмана-Форда, показанного выше
  bool hasNegativeCycle = false;
  for (int u = 0; u < V; u++)                          // еще один проход для проверки
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] > dist[u] + v.second)          // если это до сих пор возможно
        hasNegativeCycle = true;          // значит, цикл с отрицательным весом существует!
    }
  printf("Цикл с отрицательным весом существует? %s\n", hasNegativeCycle ? "Да" : "Нет");
```

На олимпиадах по программированию медленная работа алгоритма Форда–Беллмана и его функции обнаружения цикла с отрицательным весом приводит к тому, что он используется только для решения задачи SSSP на небольшом графе, где не гарантировано отсутствие цикла отрицательного веса.

Упражнение 4.4.4.1. Почему, просто выполнив операцию `relax` для всех E ребер нашего взвешенного графа $V - 1$ раз, мы получим правильный ответ для решения задачи SSSP? Докажите это.

Упражнение 4.4.4.2. Временная сложность $O(VE)$ (для худшего случая) на практике слишком велика. В большинстве случаев мы можем остановить работу алгоритма Форда–Беллмана намного раньше. Предложите простое улучшение приведенного выше кода, чтобы заставить алгоритм Форда–Беллмана работать быстрее, чем $O(VE)$.

Упражнение 4.4.4.3*. Известное улучшение для алгоритма Форда–Беллмана (особенно среди китайских программистов) – это ускоренный алгоритм поиска кратчайшего пути (Shortest Path Faster Algorithm, SPFA). Изучите материал из раздела 9.30.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/sssp.html

Файл исходного кода: *ch4_06_bellman_ford.cpp/java*

Задачи по программированию, связанные с кратчайшими путями из одной вершины

- На невзвешенном графе: BFS, более простые
  1. UVa 00336 – A Node Too Far (обсуждается в этом разделе)
  2. UVa 00383 – Shipping Routes (простой вариант SSSP, решается с помощью BFS, используйте код для сопоставления (mapper))
  3. *UVa 00388 – Galactic Import* (ключевая идея: мы хотим минимизировать перемещения планет, потому что каждое использованное ребро уменьшает значение на 5 %)
  4. **UVa 00429 – Word Transformation** * (каждое слово является вершиной, соедините два слова ребром, если они отличаются на одну букву)
  5. UVa 00627 – The Net (также распечатайте путь, см. обсуждение в этом разделе)
  6. UVa 00762 – We Ship Cheap (простая задача SSSP, решается с помощью BFS, используйте код для сопоставления (mapper))
  7. **UVa 00924 – Spreading the News** * (распространение похоже на прохождение BFS)
  8. *UVa 01148 – The mysterious X network* (LA 3502, Юго-Западная Европа'05, единственная начальная вершина, одна конечная вершина, задача о поиске кратчайшего пути, но исключая конечные точки)

9. UVa 10009 – All Roads Lead Where? (простая задача SSSP, решается с помощью BFS)
10. UVa 10422 – Knights in FEN (решается с помощью BFS)
11. UVa 10610 – Gopher and Hawks (решается с помощью BFS)
12. **UVa 10653 – Bombs; NO they... *** (эффективная реализация BFS)
13. UVa 10959 – The Party, Part I (SSSP от начальной вершины 0 до остальных)

- На невзвешенном графе: BFS, более сложные
  1. **UVa 00314 – Robot *** (состояние: (позиция, направление), преобразование входных данных графа)
  2. UVa 00532 – Dungeon Master (3-D BFS)
  3. *UVa 00859 – Chinese Checkers* (BFS)
  4. *UVa 00949 – Getaway* (интересное использование структур данных графа)
  5. UVa 10044 – Erdos numbers (входной синтаксический анализ проблематичен; если у вас возникли трудности с этим, см. раздел 6.2)
  6. UVa 10067 – Playing with Wheels (неявный граф в постановке задачи)
  7. UVa 10150 – Doublets (состояние BFS – строка!)
  8. UVa 10977 – Enchanted Forest (BFS с заблокированными состояниями)
  9. UVa 11049 – Basic Wall Maze (некоторые запрещенные переходы, необходимо распечатать путь)
  10. **UVa 11101 – Mall Mania *** (BFS с несколькими начальными вершинами из m1, получить минимум на границе m2)
  11. UVa 11352 – Crazy King (сначала отфильтруйте граф, после чего он станет SSSP)
  12. UVa 11624 – Fire (BFS с несколькими исходными вершинами)
  13. UVa 11792 – Krochanska is Here (будьте осторожны с «особой станцией»)
  14. **UVa 12160 – Unlock the Lock *** (LA 4408, Куала-Лумпур'08, Вершины = Числа; свяжите два числа с ребром, если мы можем использовать нажатие кнопки, чтобы преобразовать одно в другое; используйте BFS, чтобы получить ответ)

- На взвешенном графе: алгоритм Дейкстры, более простые
  1. **UVa 00929 – Number Maze *** (на двумерном лабиринте / неявном графе)
  2. ***UVa 01112 – Mice and Maze **** (LA 2425, Юго-Западная Европа'01, запустите алгоритм Дейкстры из конечной вершины)
  3. UVa 10389 – Subway (используйте базовые знания геометрии для построения взвешенного графа, после запустите алгоритм Дейкстры)
  4. **UVa 10986 – Sending email *** (непосредственное применение алгоритма Дейкстры)

- На взвешенном графе: алгоритм Дейкстры, более сложные
  1. UVa 01202 – Finding Nemo (LA 3133, Пекин'04, SSSP, алгоритм Дейкстры на сетке: рассматривайте каждую ячейку как вершину; идея проста, но с реализацией следует быть осторожнее)

2. UVa 10166 – Travel (эту задачу можно смоделировать как задачу поиска кратчайшего пути)
3. *UVa 10187 – From Dusk Till Dawn* (особые случаи: начало маршрута = пункт назначения: 0 литров; город, из которого начинается путешествие, или пункт назначения не найден, или до города, являющегося конечной целью путешествия, невозможно добраться из города, где путешествие начинается: маршрут отсутствует; остальные: алгоритм Дейкстры)
4. UVa 10278 – Fire Station (алгоритм Дейкстры: исследование маршрута от пожарных станций до всех перекрестков; требуется сокращение лишних вариантов, чтобы удовлетворить ограничениям по времени)
5. UVa 10356 – Rough Roads (мы можем добавить один дополнительный параметр для каждой вершины: добираемся ли мы до этой вершины, используя цикл, или же нет; затем запустите алгоритм Дейкстры, чтобы решить задачу SSSP на этом видоизмененном графе)
6. UVa 10603 – Fill (состояние: (a, b, c), начальная вершина: (0, 0, c), шесть возможных переходов)
7. **UVa 10801 – Lift Hopping *** (тщательно смоделируйте граф!)
8. *UVa 10967 – The Great Escape* (смоделируйте граф; найдите кратчайший путь)
9. *UVa 11338 – Minefield* (кажется, что тестовых данных меньше, чем указано в описании задачи ($n \leqslant 10\,000$); мы используем цикл $O(n^2)$ для построения взвешенного графа и запускаем алгоритм Дейкстры, не получая вердикт жюри «превышение лимита времени» (TLE))
10. UVa 11367 – Full Tank? (решение обсуждалось выше в этом разделе)
11. UVa 11377 – Airport Setup (тщательно смоделируйте граф: ребро, соединяющее один город и другой город, не имеющий аэропорта, имеет граничный вес 1. Ребро, соединяющее один город и другой город, имеющий аэропорт, имеет граничный вес 0. Запустите алгоритм Дейкстры из начальной вершины. Если город пункта отправки и город пункта назначения совпадают, и при этом этот город не имеет аэропорта, ответ должен быть 0)
12. **UVa 11492 – Babel *** (моделирование графа; каждое слово является вершиной; соедините две вершины ребром, если они имеют общий язык и имеют различный первый символ; соедините исходную вершину со всеми словами, принадлежащими начальному языку; соедините все слова, которые принадлежат конечному языку для получения конечной вершины, мы можем перевести вес вершины в вес ребра, затем решите задачу SSSP, начиная с исходной вершины и заканчивая конечной вершиной
13. UVa 11833 – Route Change (остановка алгоритма Дейкстры на служебном маршруте плюс некоторые изменения алгоритма)
14. ***UVa 12047 – Highest Paid Toll **** (умное использование алгоритма Дейкстры; запуск алгоритма Дейкстры, начиная от исходной точки и от места назначения; перепробуйте все ребра (u, v), если $dist[source][u] + weight(u, v) + dist[v][пункт\ назначения] \leqslant p$, запишите наибольший найденный вес ребра)

15. *UVa 12144 – Almost Shortest Path* (алгоритм Дейкстры; необходимо хранить несколько предшественников)
16. IOI 2011 – Crocodile (можно смоделировать как задачу SSSP)

- SSSP на графе с циклом отрицательного веса (алгоритм Форда–Беллмана)
  1. **UVa 00558 – Wormholes** * (проверка наличия цикла отрицательного веса)
  2. ***UVa 10449 – Traffic*** * (найдите путь минимального веса, который может быть отрицательным; будьте осторожны: ∞ + отрицательный вес меньше, чем ∞!)
  3. **UVa 10557 – XYZZY** * (проверьте «положительный» цикл (цикл с положительным весом), проверьте связность!)
  4. UVa 11280 – Flying to Fredericton (модифицированный алгоритм Форда–Беллмана)

4.5. Кратчайшие пути между всеми вершинами

4.5.1. Обзор

Задача: для заданного связного взвешенного графа G с $V \leqslant 100$ и двумя вершинами s и d найдите максимально возможное значение `dist[s][i] + dist[i][d]` для всех возможных $i \in [0...V-1]$. Это ключевая идея для решения задачи UVa 11463 – Commandos. Тем не менее каков наилучший способ реализации кода решения этой задачи?

Для решения данной задачи необходимо найти кратчайший путь из всех возможных исходных точек (всех возможных вершин) G. Мы можем выполнить V вызовов алгоритма Дейкстры, который мы изучили ранее (см. раздел 4.4.3). Однако можем ли мы решить эту задачу, написав более короткий код? Ответ: да. Если данный взвешенный граф имеет $V \leqslant 400$, то существует другой алгоритм, который гораздо *проще* написать.

Загрузите небольшой граф в матрицу смежности и затем запустите следующий код, состоящий из четырех строк, с тремя вложенными циклами, приведенный ниже. Когда он завершится, `AdjMat[i][j]` будет содержать кратчайшее расстояние между двумя парами вершин i и j в G. Первоначальная задача (UVa 11463, формулировка которой приведена выше) теперь становится легкой.

```
// внутри int main()
  // предусловие: AdjMat[i][j] содержит вес ребра (i, j)
  // или INF (1B), если такого ребра не существует
  // AdjMat - массив 32-битовых целых чисел со знаком
  for (int k = 0; k < V; k++)                          // помните, что порядок цикла k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMat[i][j] = min(AdjMat[i][j], AdjMat[i][k] + AdjMat[k][j]);
```

Файл исходного кода: *ch4_07_floyd_warshall.cpp/java*

Этот алгоритм называется алгоритмом Флойда–Уоршелла, он изобретен Робертом В. Флойдом [19] и Стивеном Уоршеллом [70]. Алгоритм Флойда–Уоршелла – это алгоритм DP, который имеет временную сложность $O(V^3)$ благодаря трем вложенным циклам[1]. Следовательно, в условиях олимпиады по программированию он может использоваться только для графов с $V \leqslant 400$. В общем, алгоритм Флойда–Уоршелла решает еще одну классическую задачу на графе: задачу о нахождении кратчайших расстояний между всеми вершинами (All-Pairs Shortest Paths, APSP). Этот алгоритм (для небольших графов) дает преимущество по сравнению с многократным вызовом алгоритма SSSP:

1) V вызовов $O((V + E)\log V)$ алгоритма Дейкстры = $O(V^3 \log V)$, если $E = O(V^2)$;
2) V вызовов $O(VE)$ алгоритма Форда–Беллмана = $O(V^4)$, если $E = O(V^2)$.

На олимпиадах по программированию главная привлекательность алгоритма Флойда–Уоршелла – это в основном скорость его реализации: только четыре короткие строчки. Если заданный граф невелик ($V \leqslant 400$), не стесняйтесь использовать этот алгоритм – даже если вам нужно только решение задачи SSSP.

Упражнение 4.5.1.1. Есть ли какая-либо конкретная причина, по которой для `AdjMat[i][j]` должно быть установлено значение 1B (10^9), чтобы указать, что между «*i*» и «*j*» нет границы? Почему мы не используем $2^{31} - 1$ (MAX_INT)?

Упражнение 4.5.1.2. В разделе 4.4.4 мы различаем граф с ребрами отрицательного веса, но без цикла отрицательного веса, и граф, имеющий цикл отрицательного веса. Будет ли этот короткий алгоритм Флойда–Уоршелла работать на графе с отрицательным весом и/или на графе, имеющем цикл отрицательного веса? Проведите эксперимент!

4.5.2. Объяснение алгоритма DP Флойда–Уоршелла

Мы предоставляем этот раздел для читателей, которым интересно узнать, почему работает алгоритм Флойда–Уоршелла. Данный раздел можно пропустить, если вы просто хотите использовать этот алгоритм как таковой.

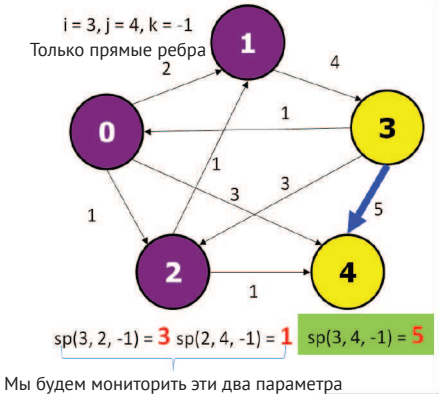
Тем не менее изучение данного раздела может еще больше укрепить ваши навыки DP. Обратите внимание, что существуют задачи на графах, которые еще не имеют классического алгоритма и должны решаться с помощью методов DP (см. раздел 4.7.1).

Основная идея алгоритма Флойда–Уоршелла состоит в том, чтобы постепенно разрешать использование промежуточных вершин (`vertex[0..k]`) для формирования кратчайших путей. Обозначим кратчайший путь от вершины *i* к вершине *j*, используя только промежуточные вершины `[0..k]` в качестве `sp(i, j, k)`. Пусть вершины помечены от 0 до $V - 1$. Мы начинаем с прямых ребер только тогда, когда $k = -1$, то есть `sp(i, j, -1)` = вес ребра (*i*, *j*). Затем мы находим кратчайшие пути между любыми двумя вершинами с помощью ограниченных промежуточных вершин из вершины [0..*k*]. На рис. 4.20 мы хотим

---

[1] Алгоритм Флойда–Уоршелла должен использовать матрицу смежности, чтобы получить доступ к весу ребра (*i*, *j*) за время $O(1)$.

найти `sp(3,4,4)` – кратчайший путь от вершины 3 до вершины 4, используя любую промежуточную вершину в графе (вершина `[0..4]`). Возможный кратчайший путь – это путь 3–0–2–4 со стоимостью 3. Но как достичь этого решения? Мы знаем, что, используя только прямые ребра, `sp(3,4, -1)` = 5, как показано на рис. 4.20. Решение для `sp(3,4,4)` в конечном итоге будет достигнуто из `sp(3,2,2)` + `sp(2,4,2)`. Но при использовании только прямых ребер `sp(3,2, -1)` + `sp(2,4, -1)` = 3 + 1 = 4, что по-прежнему > 3.



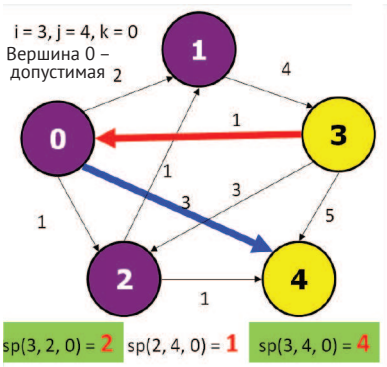
Текущее содержимое матрицы смежности D при $k = -1$

k = -1	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	5
4	∞	∞	∞	∞	0

Рис. 4.20 ❖ Объяснение алгоритма Флойда–Уоршелла 1

Затем алгоритм Флойда–Уоршелла шаг за шагом устанавливает $k = 0$, потом $k = 1$, $k = 2$... до $k = V - 1$.

Когда мы берем $k = 0$, то есть вершина 0 теперь может использоваться как промежуточная вершина, тогда `sp(3,4,0)` уменьшается как `sp(3,4,0)` = `sp(3,0, -1)` + `sp(0,4, -1)` = 1 + 3 = 4, как показано на рис. 4.21. Обратите внимание, что при $k = 0$ `sp(3,2,0)` – что нам понадобится позже – также снизится с 3 до `sp(3,0, -1)` + `sp(0,2, -1)` = 1 + 1 = 2. Алгоритм Флойда–Уоршелла будет обрабатывать `sp(i, j, 0)` для всех остальных пар, рассматривая только вершину 0 как промежуточную вершину, но есть лишь еще одно изменение: `sp(3,1,0)` от ∞ до 3.

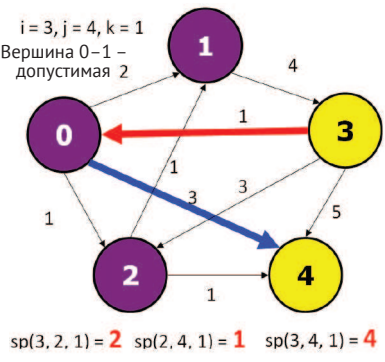


Текущее содержимое матрицы смежности D при $k = 0$

k = 0	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

Рис. 4.21 ❖ Объяснение алгоритма Флойда–Уоршелла 2

Когда мы берем $k = 1$, то есть вершины 0 и 1 теперь можно использовать в качестве промежуточных вершин, тогда получается, что нет изменений ни в `sp(3,2,1)`, ни в `sp(2,4,1)`, ни в `sp(3,4,1)`. Однако два других значения изменяются: `sp(0,3,1)` и `sp(2,3,1)`, как показано на рис. 4.22, но эти два значения не влияют на окончательное вычисление кратчайшего пути между вершинами 3 и 4.

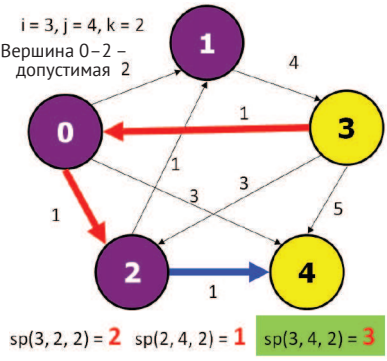


Текущее содержимое матрицы смежности D при $k = 1$

k = 1	0	1	2	3	4
0	0	2	1	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

Рис. 4.22 ❖ Объяснение алгоритма Флойда–Уоршелла 3

Если мы возьмем $k = 2$, то есть вершины 0, 1 и 2 теперь можно использовать в качестве промежуточных вершин, тогда значение `sp(3,4,2)` снова уменьшается, так как `sp(3,4,2) = sp(3, 2,2) + sp(2,4,2) = 2 + 1 = 3`, как показано на рис. 4.23. Алгоритм Флойда–Уоршелла повторяет этот процесс для $k = 3$ и, наконец, $k = 4$, но значение `sp(3,4,4)` остается равным 3, и это окончательный ответ.



Текущее содержимое матрицы смежности D при $k = 2$

k = 2	0	1	2	3	4
0	0	2	1	6	2
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

Рис. 4.23 ❖ Объяснение алгоритма Флойда–Уоршелла 4

Формально мы определяем повторения DP алгоритма Флойда–Уоршелла следующим образом. Пусть $D^k_{i,j}$ будет кратчайшим расстоянием от i до j только для [0..k] промежуточных вершин. Тогда мы можем определить работу алгоритма Флойда–Уоршелла так:

$D^{-1}_{i,j}$ = *вес*(*i*, *j*). Это основной случай, когда мы не используем промежуточные вершины.
$D^{k}_{i,j} = min(D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j})$ = *min*(не используем вершину *k*, используем вершину *k*), для $k \geqslant 0$.

Эта постановка задачи DP должна работать с данными графа уровень за уровнем (путем увеличения *k*). Чтобы добавить запись в таблице *k*, мы используем записи в таблице *k* – 1. Например, чтобы вычислить $D^{2}_{3,4}$ (строка 3, столбец 4, в таблице *k* = 2, индекс начинается с 0), мы вычисляем минимум $D^{1}_{3,4}$ или сумму $D^{1}_{3,2} + D^{1}_{2,4}$ (см. табл. 4.3). Наивная реализация – использование трехмерной матрицы `D[k][i][j]` размерности $O(V^3)$. Однако, поскольку для вычисления уровня *k* нам нужно знать только значения из уровня *k* – 1, мы можем отбросить размерность *k* и вычислять `D[i][j]` «на лету» (прием, позволяющий экономить место, обсуждаемый в разделе 3.5.1). Таким образом, алгоритму Флойда – Уоршелла нужно только $O(V^2)$ памяти, хотя он все еще работает на $O(V^3)$.

Таблица 4.3. Таблица DP для алгоритма Флойда–Уоршелла

k = 1

	k = 1	0	1	2	3	4
	0	0	2	1	6	3
	1	∞	0	∞	4	∞
k	**2**	∞	1	0	5	**1**
i	**3**	1	3	**2**	0	**4**
	4	∞	∞	∞	∞	0

k = 2

	k = 2	0	1	2	3	4
	0	0	2	1	6	2
	1	∞	0	∞	4	∞
	2	∞	1	0	5	1
i	**3**	1	3	2	0	**3**
	4	∞	∞	∞	∞	0

4.5.3. Другие применения

Основная цель алгоритма Флойда–Уоршелла – решить задачу о нахождении кратчайших расстояний между всеми вершинами. Тем не менее алгоритм Флойда–Уоршелла часто используется и в других задачах, если граф небольшой. Здесь мы перечисляем несколько вариантов задач, которые также могут быть решены с использованием данного алгоритма.

Решение задачи SSSP на небольшом взвешенном графе

Если у нас есть информация о кратчайших расстояниях между всеми вершинами (APSP), нам также известно решение задачи о кратчайших путях из одной вершины до любой другой вершины графа (SSSP). Если данный взвешенный граф имеет сравнительно небольшое число вершин $V \leqslant 400$, может быть полезно с точки зрения времени написания кода использовать четырехстрочный код алгоритма Флойда–Уоршелла, а не более длинный алгоритм Дейкстры.

Вывод кратчайших путей

Распространенная проблема, с которой сталкиваются программисты, использующие четырехстрочный код алгоритма Флойда–Уоршелла, не понимая, как

это работает, – это когда их просят также вывести кратчайшие пути. В алгоритмах BFS / Дейкстры / Форда–Беллмана нам просто нужно запомнить дерево кратчайших путей, используя одномерный массив `vi p` для хранения родительской информации для каждой вершины. В алгоритме Флойда–Уоршелла мы должны хранить двумерную исходную матрицу. Измененный код алгоритма показан ниже.

```
// внутри int main()
  // пусть p - двумерная исходная матрица, где p[i][j] - последняя вершина перед j
  // на кратчайшем пути из i в j, т. е. i -> ... -> p[i][j] -> j
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      p[i][j] = i;                                        // инициализируем исходную матрицу
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)         // на этот раз нам нужно использовать оператор if
        if (AdjMat[i][k] + AdjMat[k][j] < AdjMat[i][j]) {
          AdjMat[i][j] = AdjMat[i][k] + AdjMat[k][j];
          p[i][j] = p[k][j];                                  // обновляем исходную матрицу
        }
//--------------------------------------------------------------------------
// когда нам нужно распечатать самые короткие пути, мы можем вызвать метод,
// приведенный ниже:
void printPath(int i, int j) {
  if (i != j) printPath(i, p[i][j]);
  printf(" %d", j);
}
```

Транзитивное замыкание (алгоритм Уоршелла)

Стивен Уоршелл [70] разработал алгоритм для решения задачи *транзитивного замыкания*: для заданного графа необходимо определить, связана ли вершина *i* с вершиной *j* прямо или косвенно. Этот вариант использует логические побитовые операторы, которые (намного) быстрее, чем арифметические операторы. Первоначально `AdjMat[i][j]` содержит значение 1 (`true`), если вершина *i* напрямую связана с вершиной *j*, и 0 (`false`) в противном случае. После запуска алгоритма Уоршелла, имеющего временную сложность $O(V^3)$, приведенного ниже, мы можем проверить, связаны ли любые две вершины *i* и *j* прямо либо косвенно, проверяя значения `AdjMat[i][j]`.

```
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMat[i][j] |= (AdjMat[i][k] & AdjMat[k][j]);
```

Минимакс и максимин (повторное рассмотрение задачи)

Ранее в разделе 4.3.4 мы обсуждали задачу нахождения минимаксного (и максиминного) пути. Ниже показано решение этой задачи с использованием алгоритма Флойда–Уоршелла. Сначала инициализируйте `AdjMat[i][j]` как вес ребра (`i, j`). Это минимаксная стоимость по умолчанию для двух вершин, связанных напрямую.

Для пары i-j без прямого ребра установим AdjMat[i][j] = INF. Затем мы перебираем все возможные промежуточные вершины *k*. Минимальная стоимость AdjMat[i][j] является минимумом либо самой себя, либо максимума между AdjMat[i][k] и AdjMat[k][j]. Однако этот подход можно использовать только в том случае, если входной граф достаточно мал ($V \leqslant 400$).

```
for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      AdjMat[i][j] = min(AdjMat[i][j], max(AdjMat[i][k], AdjMat[k][j]));
```

Нахождение (самого дешевого / отрицательного) цикла

В разделе 4.4.4 мы видели, как работа алгоритма Форда–Беллмана завершается после $O(VE)$ шагов независимо от типа входного графа (поскольку он выполняет операцию relax для всех E ребер не более $V - 1$ раз) и как алгоритм Форда–Беллмана можно использовать для проверки, имеет ли данный граф цикл отрицательного веса. Алгоритм Флойда–Уоршелла также завершается после $O(V^3)$ шагов независимо от типа входного графа. Это позволяет использовать алгоритм Флойда–Уоршелла для определения того, имеет ли (сравнительно небольшой) граф цикл отрицательного веса, и даже находить имеющий минимальную стоимость цикл (с неотрицательным весом) среди всех возможных циклов (обход графа).

Для этого мы изначально установили для *главной диагонали* матрицы смежности очень большое значение, т. е. AdjMat[i][i] = INF (1B). Затем мы запускаем алгоритм Флойда–Уоршелла, имеющий временную сложность $O(V^3)$. Далее мы проверим значение AdjMat[i][i], которое теперь выражает вес кратчайшего цикла, который начинается с вершины i, проходит до $V - 1$ промежуточных вершин и снова возвращается в i. Если значения AdjMat[i][i] больше не равны INF для любого i ∈ [0..V-1], то у нас есть цикл. Наименьший неотрицательный элемент AdjMat[i][i] ∀i ∈ [0..V-1] – самый дешевый цикл. Если AdjMat[i][i] < 0 для любого i ∈ [0..V-1], то у нас есть цикл отрицательного веса, потому что если мы возьмем этот цикл еще раз, то получим еще более короткий «кратчайший» путь.

Определение диаметра графа

Диаметр графа определяется как максимальное расстояние по кратчайшему пути между любой парой вершин этого графа. Чтобы найти диаметр графа, мы сначала находим кратчайший путь между каждой парой вершин (т. е. решаем задачу APSP). Максимальное найденное расстояние – это диаметр графа. Задача UVa 1056 – Degrees of Separation, которая предлагалась на финальных соревнованиях на кубке мира ICPC в 2006 году, – это задача на определение диаметра графа. Чтобы решить ее, мы можем сначала запустить алгоритм Флойда–Уоршелла с временной сложностью $O(V^3)$, чтобы получить необходимую нам информацию, которая содержится в решении задачи APSP. Затем можем узнать, каков диаметр графа, найдя максимальное значение в AdjMat за время $O(V^2)$. Однако мы можем сделать это только для небольшого графа с $V \leqslant 400$.

Нахождение компонент сильной связности ориентированного графа

В разделе 4.2.1 мы узнали, как алгоритм Тарьяна с временной сложностью $O(V+E)$ может использоваться для поиска компонент сильной связности (SCCs) ориентированного графа. Тем не менее код алгоритма довольно длинный. Если входной граф имеет небольшой размер (например, как в задачах UVa 247 – Calling Circles, UVa 1229 – Sub-dictionary, UVa 10731 – Test), мы также можем найти компоненты сильной связности графа за время, не превышающее $O(V^3)$, используя алгоритм Уоршелла поиска транзитивного замыкания, а затем выполнить следующую проверку: чтобы найти все элементы компоненты сильной связности, которые содержат вершину `i`, проверьте все остальные вершины `j` ∈ [`0..V-1`]. Если выражение `AdjMat[i][j] && AdjMat[j][i]` имеет значение `true`, то вершины `i` и `j` принадлежат одной и той же компоненте сильной связности.

Упражнение 4.5.3.1. Как найти транзитивное замыкание графа с $V \leqslant 1000$, $E \leqslant 100\,000$? Предположим, что существует только Q ($1 \leqslant Q \leqslant 100$) запросов транзитивного замыкания для этой задачи, позволяющих ответить на следующий вопрос: связана вершина *u* с вершиной *v* прямо или косвенно? Что, если входной граф ориентирован? Упрощает ли задачу это свойство направленности?

Упражнение 4.5.3.2*. Решите задачу о нахождении *максимального* пути с помощью алгоритма Флойда–Уоршелла.

Упражнение 4.5.3.3. Арбитраж – это обмен одной валюты на другую с надеждой воспользоваться небольшими различиями в курсах обмена между несколькими валютами для получения прибыли. Например, в задаче UVa 436 – Arbitrage II: если за 1,0 доллар США (USD) можно купить 0,5 британского фунта (GBP), за 1,0 GBP можно купить 10,0 французского франка (FRF[1]) и за 1,0 FRF можно купить 0,21 USD, то арбитражный трейдер может начать с 1,0 долл. США и покупки $1{,}0 \times 0{,}5 \times 10{,}0 \times 0{,}21 = 1{,}05$ долл. Таким образом, прибыль составит 5 %. Эта проблема на самом деле является проблемой поиска прибыльного цикла. Это похоже на задачу нахождения цикла с помощью алгоритма Флойда–Уоршелла, показанную в этом разделе. Решите задачу об арбитраже, используя алгоритм Флойда–Уоршелла.

Замечания о задачах поиска кратчайших путей на олимпиадах по программированию

Все три алгоритма, обсуждаемых в двух предыдущих разделах: Дейкстры, Форда–Беллмана и Флойда–Уоршелла, – используются при решении задач поиска кратчайших путей (SSSP или APSP) на взвешенных графах для *общего случая*. Среди этих трех алгоритмов алгоритм Форда–Беллмана с временной сложностью $O(VE)$ редко используется на олимпиадах по программированию из-за своей высокой сложности. Его полезно применять, только если автор задачи

---

[1] В настоящий момент (2013) Франция фактически использует евро в качестве своей валюты.

дает граф «разумного размера» с циклом отрицательного веса. Для общих случаев (модифицированный нами) вариант реализации алгоритма Дейкстры с временной сложностью $O((V+E)\log V)$ является лучшим решением проблемы SSSP для взвешенного графа «разумного размера» без цикла отрицательного веса. Однако когда этот граф мал ($V \leqslant 400$) – что случается не так редко, – как следует из материала, изложенного в данном разделе, $O(V^3)$ алгоритм Флойда–Уоршелла – наилучший путь решения задач.

Одна из возможных причин того, почему алгоритм Флойда–Уоршелла весьма популярен на олимпиадах по программированию, заключается в том, что иногда автор задачи включает поиск кратчайших путей как промежуточный этап в решении основной, (гораздо) более сложной задачи. Чтобы сделать задачу по-прежнему выполнимой за время, отведенное на соревнования, автор задачи намеренно устанавливает небольшой размер входного файла, чтобы подзадача по кратчайшим путям была решена с помощью четырех строчек кода, реализующих алгоритм Флойда–Уоршелла (например, UVa 10171, 10793, 11463). Неконкурентоспособный программист выберет более длинный путь для решения этой подзадачи.

Согласно нашему опыту, многие задачи на нахождение кратчайших путей не являются задачами на взвешенных графах, для решения которых нужно использовать алгоритмы Дейкстры или Флойда–Уоршелла. Если вы посмотрите на упражнения по программированию, перечисленные в разделе 4.4 (и позже в разделе 8.2), то увидите, что многие из них являются задачами на невзвешенных графах, которые разрешимы с помощью метода поиска в ширину (BFS) (см. раздел 4.4.2).

Мы также наблюдаем, что чаще всего задачи поиска кратчайших путей включают тщательное моделирование графов (UVa 10067, 10801, 11367, 11492, 12160) – это становится трендом последних лет. Поэтому, чтобы преуспеть в соревнованиях по программированию, убедитесь, что у вас есть этот навык: способность быстро и безошибочно определять тип графа на основании условий задачи. В этой главе мы показали несколько примеров таких навыков моделирования графов, которые, мы надеемся, вы сможете оценить и в конечном итоге овладеть ими.

В разделе 4.7.1 мы рассмотрим некоторые задачи нахождения кратчайших путей в направленном ациклическом графе (Directed Acyclic Graph, DAG). Этот важный вариант решаем с помощью универсальной техники динамического программирования (DP), которая обсуждалась в разделе 3.5. В данном разделе мы также представим другой взгляд на метод DP как на «алгоритм на DAG».

Мы представляем сравнительный анализ использования алгоритмов для решения задач SSSP/APSP на олимпиадах по программированию в табл. 4.4, чтобы помочь читателям решить, какой алгоритм выбрать в зависимости от различных параметров графа. Используются следующие термины: «Наилучший» → наиболее подходящий алгоритм; «OK» → правильный алгоритм, но не лучший; «Плохой» → (очень) медленный алгоритм; «WA» → неверный алгоритм и «Избыточный» → правильный алгоритм, но его использование избыточно и потому нецелесообразно.

Таблица 4.4. Сравнительная таблица для алгоритмов SSSP/APSP

Критерий (параметр графа)	**BFS** ***O*(*V* + *E*)**	**Алгоритм Дейкстры** ***O*((*V* + *E*)log *V*)**	**Алгоритм Форда–Беллмана** ***O*(*VE*)**	**Алгоритм Флойда–Уоршелла** ***O*(V^3)**
Максимальный размер	$V, E \leqslant 10M$	$V, E \leqslant 300K$	$VE \leqslant 10M$	$V \leqslant 400$
Невзвешенный	Наилучший	OK	Плохой	Плохой (как правило)
Взвешенный	WA	Наилучший	OK	Плохой (как правило)
С отрицательным весом	WA	Наш вариант – OK	OK	Плохой (как правило)
Имеющий цикл отрицательного веса	Не обнаруживает	Не обнаруживает	Обнаруживает	Обнаруживает
Граф малого размера	WA (если взвешенный)	Избыточный	Избыточный	Наилучший

Задачи по программированию, связанные с применением алгоритма Флойда–Уоршелла

- Стандартное применение алгоритма Флойда–Уоршелла (для задач APSP или SSSP на небольшом графе)
  1. UVa 00341 – Non-Stop Travel (граф небольшого размера)
  2. UVa 00423 – MPI Maelstrom (граф небольшого размера)
  3. UVa 00567 – Risk (простой случай SSSP, решается с помощью BFS, но мы имеем граф небольшого размера, поэтому задачу легче решить с помощью алгоритма Флойда–Уоршелла)
  4. **UVa 00821 – Page Hopping *** (LA 5221, финальные соревнования на кубок мира, Орландо'00, одна из самых «легких» задач финальных соревнований олимпиады ICPC)
  5. UVa 01233 – USHER (LA 4109, Сингапур'07, алгоритм Флойда–Уоршелла можно использовать для определения цикла с минимальной стоимостью на графе; максимальный размер входного графа составляет $p \leqslant 500$, однако при этом такое решение задачи все же не получает вердикт «TLE» согласно «Онлайн-арбитру» университета Вальядолида (UVa))
  6. UVa 01247 – Interstar Transport (LA 4524, Синьчжу'09, APSP, алгоритм Флойда–Уоршелла, немного измененный, чтобы предпочесть кратчайший путь с наименьшим количеством промежуточных вершин)
  7. **UVa 10171 – Meeting Prof. Miguel *** (легко решается с информацией о кратчайших расстояниях между всеми вершинами (APSP))
  8. *UVa 10354 – Avoiding Your Boss* (найдите кратчайшие пути вашего начальника, удалите ребра, связанные с кратчайшими путями вашего начальника, перезапустите поиск кратчайших путей от дома до рынка)

9. *UVa 10525 – New to Bangladesh?* (используйте две матрицы смежности: время и длину; используйте модифицированный алгоритм Флойда–Уоршелла)
10. UVa 10724 – Road Construction (добавление одного ребра меняет «несколько вещей»)
11. UVa 10793 – The Orc Attack (алгоритм Флойда–Уоршелла упрощает эту задачу)
12. UVa 10803 – Thunder Mountain (граф небольшого размера)
13. UVa 10947 – Bear with me, again… (граф небольшого размера)
14. UVa 11015 – 05-32 Rendezvous (граф небольшого размера)
15. **UVa 11463 – Commandos *** (легко решается, используя информацию о кратчайших расстояниях между всеми вершинами (APSP))
16. *UVa 12319 – Edgetown's Traffic Jams* (алгоритм Флойда–Уоршелла, повторить дважды и сравнить)

- Другие варианты применения алгоритма Флойда–Уоршелла

1. **UVa 00104 – Arbitrage *** (небольшая задача об арбитраже, решаемая с помощью алгоритма Флойда–Уоршелла)
2. UVa 00125 – Numbering Paths (модифицированный алгоритм Флойда–Уоршелла)
3. UVa 00186 – Trip Routing (небольшой граф, необходимо вывести маршрут)
4. *UVa 00274 – Cat and Mouse* (вариант задачи поиска транзитивного замыкания)
5. UVa 00436 – Arbitrage (II) (еще одна задача об арбитраже)
6. **UVa 00334 – Identifying Concurrent… *** (транзитивное замыкание ++)
7. UVa 00869 – Airline Comparison (запустите алгоритм Флойда 2x, сравните матрицы смежности)
8. *UVa 00925 – No more prerequisites…* (транзитивное замыкание ++)
9. ***UVa 01056 – Degrees of Separation **** (LA 3569, финальные соревнования на кубок мира, Сан-Антонио'06, диаметр небольшого графа)
10. *UVa 01198 – Geodetic Set Problem* (LA 2818, Гаосюн'03, транзитивное замыкание ++)
11. UVa 11047 – The Scrooge Co Problem (необходимо вывести найденный путь; особый случай: если начальная точка = пункт назначения, выведите дважды)

Известные авторы алгоритмов

Роберт В. Флойд (1936–2001) был выдающимся американским ученым, специалистом в области теории вычислительных систем. Вклад Флойда включает разработку алгоритма Флойда [19], который эффективно находит все кратчайшие пути на графе. Флойд работал в тесном контакте с Дональдом Эрвином Кнутом, в частности он был основным рецензентом основополагающей кни-

ги Кнута «Искусство программирования». Его публикации цитируются в этой книге чаще всех остальных.

Стивен Уоршелл (1935–2006) был специалистом по вычислительной технике, который изобрел алгоритм транзитивного замыкания, теперь известный как алгоритм Уоршелла [70]. Этот алгоритм был позже назван алгоритмом Флойда–Уоршелла, поскольку Флойд и Уоршелл независимо друг от друга изобрели аналогичный по своей сути алгоритм.

Джек Р. Эдмондс (род. 1934) – математик. Он и Ричард Карп изобрели алгоритм Эдмондса–Карпа для вычисления максимального потока в сети с временной сложностью $O(VE^2)$ [14]. Он также изобрел алгоритм для MST на ориентированных графах (задача об ориентированном дереве). Этот алгоритм был предложен независимо сначала Чу и Лю (1965), а затем Эдмондсом (1967) – так называемый **алгоритм Чу–Лю/Эдмондса** [6]. Тем не менее его наиболее важным вкладом, вероятно, является **алгоритм нахождения паросочетаний / срезания цветка** (**алгоритм Эдмондса**) – одна из наиболее цитируемых статей по информатике [13].

Ричард Мэннинг Карп (род. 1935) – ученый, специалист в области теории вычислительных систем. Он сделал много важных открытий в области компьютерных наук, в частности в области комбинаторных алгоритмов. В 1971 году он и Эдмондс опубликовали алгоритм Эдмондса–Карпа для решения задачи о максимальном потоке [14]. В 1973 году он и Джон Хопкрофт опубликовали алгоритм Хопкрофта–Карпа, который до сих пор остается самым быстрым из известных методов нахождения максимального по мощности паросочетания на двудольном графе [28].

4.6. Поток

4.6.1. Обзор

Задача: представьте себе связный (целочисленный) взвешенный и направленный граф[1] как сеть труб, где ребра – это трубы, а вершины – точки разъединения. Каждое ребро имеет вес, равный пропускной способности трубы. Есть также две специальные вершины: источник s и сток t. Каков максимальный поток (скорость потока) от источника s к стоку t на этом графе (представьте, что вода течет в сети труб, мы хотим знать максимальный объем воды в единицу времени, который может проходить через эту сеть труб)? Эта задача называется задачей о максимальном потоке (часто сокращенно называемой просто максимальным потоком), одной из задач в целом семействе задач, связанных с потоком в сетях. Задача о максимальном потоке проиллюстрирована на рис. 4.24.

---

[1] Взвешенное ребро в неориентированном графе может быть преобразовано в два направленных ребра с одинаковым весом.

Рис. 4.24 ❖ Иллюстрация к задаче о максимальном потоке (UVa 820 [47] – финальные соревнования на кубок мира ICPC 2000, задача E)

4.6.2. Метод Форда–Фалкерсона

Одним из решений для задачи о максимальном потоке является метод Форда–Фалкерсона, изобретенный тем же Лестером Рэндольфом *Фордом-младшим*, который изобрел алгоритм Форда–Беллмана, и Дельбертом Рэем *Фалкерсоном*.

```
построим направленный остаточный граф с пропускной способностью ребер = исходным весам графа
mf = 0                    // это итерационный алгоритм, mf означает max_flow (макс. поток)
while (если существует увеличивающий путь p из s в t) {
  // p - путь из s в t, проходящий через +ve вершин в остаточном графе
  увеличивающийся/исходящий поток f по пути p (s -> ... -> i -> j -> ... t)
    1. найти f, минимальный вес ребра вдоль пути p
    2. уменьшить пропускную способность прямых ребер (например, i -> j) вдоль пути p на f
    3. увеличить пропускную способность обратных ребер (например, j -> i) вдоль пути p на f
  mf += f                 // мы можем пропускать поток размером f из s в t, увеличить mf
}
output mf                                                  // это максимальная величина потока
```

Метод Форда–Фалкерсона – это итеративный алгоритм, который многократно находит увеличивающую цепь p: путь от источника s к стоку t, который проходит через ребра, имеющие положительный вес, в остаточной сети[1]. После на-

[1] Мы используем название «остаточная сеть», потому что первоначально вес каждого ребра `res[i][j]` совпадает с исходной пропускной способностью ребра `(i, j)` в исходном графе. Если это ребро `(i, j)` используется увеличивающей цепью и поток проходит через это ребро с весом `f` ≤ `res[i][j]` (поток не может превышать эту пропускную

хождения увеличивающей цепи p, для которой f является минимальным весом ребра вдоль пути p (ребро, являющееся узким местом на этом пути), метод Форда–Фалкерсона выполняет два важных шага: уменьшение/увеличение пропускной способности прямых ($i \to j$) / обратных ($j \to i$) ребер вдоль пути p на f соответственно. Метод Форда–Фалкерсона будет повторять эти действия до тех пор, пока больше не будет существовать увеличивающей цепи от источника s к стоку t, что подразумевает, что общий поток до этих пор является максимальным потоком. Теперь, когда вы поняли эти пояснения, снова рассмотрите рис. 4.24.

Причина уменьшения емкости прямого ребра очевидна. Направляя поток через увеличивающую цепь p, мы уменьшим оставшиеся (остаточные) пропускные способности (прямых) ребер, используемых в p. Причина увеличения пропускной способности обратных ребер может быть не столь очевидна, но этот шаг важен для правильной работы метода Форда–Фалкерсона. Увеличивая пропускную способность обратного ребра ($j \to i$), метод Форда–Фалкерсона позволяет *будущей итерации* (*потоку*) отменять (частично) измененную пропускную способность прямого ребра ($i \to j$), которая была некорректно использована некоторым потоком (c) на более ранней итерации.

В приведенном выше псевдокоде есть несколько различных способов найти увеличивающую цепь s–t. В этом разделе мы рассмотрим два способа: с использованием поиска в глубину (Depth First Search, DFS) и с использованием поиска в ширину (Breadth First Search, BFS).

Метод Форда–Фалкерсона, реализованный с применением DFS, имеет временную сложность $O(|f^*|E)$, где $|f^*|$ – значение максимального потока `mf`. Это объясняется тем, что у нас может быть такой же граф, как тот, что показан на рис. 4.26. В этом случае мы можем столкнуться с ситуацией, когда две увеличивающие цепи: $s \to a \to b \to t$ и $s \to b \to a \to t$ – только уменьшают пропускную способность (прямых[1]) ребер вдоль пути на 1. В худшем случае это повторяется $|f^*|$ раз (что составляет 200 раз для графа, приведенного на рис. 4.25). Поскольку временная сложность DFS в графе потока[2] равна $O(E)$, общая временная сложность метода составляет $O(|f^*|E)$. Это не подходит для олимпиад по программированию, так как автор задачи может выбрать очень большое значение $|f^*|$.

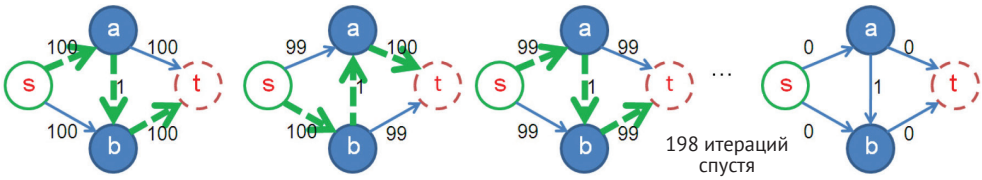


Рис. 4.25 ❖ Метод Форда–Фалкерсона, реализованный с использованием DFS, может работать медленно

способность), то оставшаяся (или остаточная) пропускная способность ребра `(i, j)` будет равна `res[i][j] - f`.

[1] Обратите внимание, что после направления потока через вершины $s \to a \to b \to t$ прямое ребро $a \to b$ заменяется обратным ребром $b \to a$ и т. д. Если этого не сделать, то максимальное значение потока составит всего 1 + 99 + 99 = 199, а не 200 (неверно).

[2] Число ребер в графе потока должно быть $E \geqslant V - 1$, чтобы обеспечить $\exists \geqslant 1$ потока $s - t$. Это подразумевает, что и DFS, и BFS – с использованием списка смежности – выполняются за $O(E)$ вместо $O(V + E)$.

4.6.3. Алгоритм Эдмондса–Карпа

Лучшей реализацией метода Форда–Фалкерсона является использование BFS для нахождения кратчайшего пути с точки зрения количества слоев/отрезков между s и t. Этот алгоритм был открыт Джеком *Эдмондсом* и Ричардом Мэннингом *Карпом* и назван алгоритмом Эдмондса–Карпа [14]. Он выполняется за $O(VE^2)$, поскольку можно доказать, что после $O(VE)$ BFS-итераций все увеличивающие цепи уже будут найдены и пройдены. Заинтересованные читатели могут обратиться к литературе, например к [14, 7], чтобы узнать больше об этом доказательстве. Поскольку BFS на графе потока выполняется за $O(E)$, общая временная сложность составляет $O(VE^2)$. В примере, приведенном на рис. 4.26, в алгоритме Эдмондса–Карпа будут использоваться только два пути s–t: $s \rightarrow a \rightarrow t$ (два отрезка, величина исходящего потока 100 единиц) и $s \rightarrow b \rightarrow t$ (два отрезка, величина исходящего потока 100 единиц). Таким образом, этот алгоритм не попадает в ловушку, когда при маршрутизации потока будут использоваться более длинные пути (три отрезка): $s \rightarrow a \rightarrow b \rightarrow t$ (или $s \rightarrow b \rightarrow a \rightarrow t$).

Первый опыт реализации алгоритма Эдмондса–Карпа может оказаться трудным для начинающих программистов. В этом разделе мы представляем нашу самую простую реализацию алгоритма Эдмондса Карпа, который использует только матрицу смежности `res`, имеющую размерность $O(V^2)$, для хранения остаточной пропускной способности каждого ребра. Эта версия алгоритма, которая выполняется за $O(VE)$ BFS-итераций $\times$ $O(V^2)$ для каждой итерации BFS из-за размерности матрицы смежности = $O(V^3E)$, работает достаточно быстро, чтобы решить *некоторые* задачи о максимальном потоке (на графах небольшого размера).

```
int res[MAX_V][MAX_V], mf, f, s, t;                              // глобальные переменные
vi p;                                          // p хранит остовное дерево BFS с вершиной в s

void augment(int v, int minEdge) {          // обход остовного дерева BFS в направлении s->t
  if (v == s) { f = minEdge; return; }       // записываем minEdge в глобальную переменную f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v]));
                         res[p[v]][v] -= f; res[v][p[v]] += f; } }

// внутри int main(): инициализируйте 'res', 's', и 't' соответствующими значениями
  mf = 0;                                                  // mf означает max_flow (макс. поток)
  while (1) {                                 // алгоритм Эдмондса-Карпа, выполняется в O(VE^2)
                                                       // (итоговая временная сложность O(V^3 E)
    f = 0;
    // запустите BFS, сравните с оригинальным BFS, приведенным в разделе 4.2.2
    vi dist(MAX_V, INF); dist[s] = 0; queue<int> q; q.push(s);
    p.assign(MAX_V, -1);                               // сохраните дерево обхода BFS из s в t
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;              // немедленно останавливаем BFS, если мы уже достигли t
      for (int v = 0; v < MAX_V; v++)              // примечание: эта часть работает медленно
        if (res[u][v] > 0 && dist[v] == INF)
          dist[v] = dist[u] + 1, q.push(v), p[v] = u;                      // 3 строчки в 1!
    }
    augment(t, INF);                       // находим минимальный вес ребра 'f' на этом пути,
                                                                    // если таковой имеется
    if (f == 0) break;          // мы больше не можем маршрутизировать поток ('f' = 0); конец
```

```
    mf += f;    // мы все еще можем маршрутизировать поток; увеличиваем максимальный поток!
  }
  printf("%d\n", mf);                                    // это максимальное значение потока
```

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/maxflow.html

Файл исходного кода: *ch4_08_edmonds_karp.cpp/java*

Упражнение 4.6.3.1. Прежде чем продолжить чтение, ответьте на следующий вопрос на рис. 4.26.

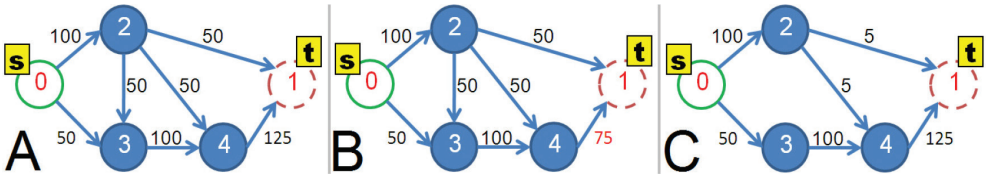


Рис. 4.26 ❖ Каково значение максимального потока этих трех остаточных сетей?

Упражнение 4.6.3.2. Основной недостаток простой реализации, код которой приводится в этом разделе, состоит в том, что при перечислении соседей вершины вместо $O(k)$ мы берем $O(V)$ (где k – число соседей этой вершины). Другой (но не существенный) недостаток заключается в том, что нам также не нужно значение `vi dist`, поскольку нам достаточно `bitset` (указывающего, была уже посещена вершина или же нет). Измените приведенный выше код, реализующий алгоритм Эдмондса–Карпа, чтобы его временная сложность не превышала $O(VE^2)$.

Упражнение 4.6.3.3*. Еще лучшая реализация алгоритма Эдмондса Карпа состоит в том, чтобы не использовать матрицу смежности $O(V^2)$ для хранения остаточной пропускной способности каждого ребра. Лучший способ – сохранять как исходную пропускную способность, так и фактический поток (а не только остаток) для каждого ребра в виде модифицированного списка смежности + ребер с временной сложностью $O(V + E)$. Таким образом, у нас есть три параметра для каждого ребра: исходная пропускная способность ребра, поток в этом ребре в настоящий момент времени, – и мы можем вывести остаточную пропускную способность ребра как разность между исходной пропускной способностью и потоком для этого ребра. Теперь реализуйте данный способ. Как эффективно справиться с обратным потоком?

4.6.4. Моделирование графа потока – часть I

С учетом приведенного выше кода, реализующего алгоритм Эдмондса–Карпа, решение (основной/стандартной) задачи о сетевом потоке, особенно задачи о максимальном потоке, упростилось. Теперь необходимо сделать следующее:

1) классифицировать задачу: необходимо убедиться в том, что задача действительно является задачей о сетевом потоке (вы будете справляться с этим гораздо лучше и быстрее, когда приобретете опыт, решив множество задач о сетевом потоке);
2) построить соответствующий граф потока (т. е. если вы используете наш код, приведенный ранее, инициируйте остаточную матрицу `res` и установите соответствующие значения для «s» и «t»);
3) запустить код, реализующий алгоритм Эдмондса–Карпа, на этом графе потока.

В этом подразделе мы показываем пример моделирования (остаточного) графа потока UVa 259 – Software Allocation[1]. Сокращенная формулировка условия задачи: у вас имеется не более 26 приложений/программ (маркированных от «A» до «Z»), не более 10 компьютеров (пронумерованных от 0 до 9), число людей, которые подали заявки на использование каждого приложения в некоторый день (положительное целое число, состоящее из одной цифры, или [1..9]), список компьютеров, на которых может работать определенное приложение, и условие, что на каждом компьютере можно запускать только одно приложение в этот день. Ваша задача состоит в том, чтобы определить, можно ли распределить (то есть сопоставить) приложения по компьютерам, и, если это возможно, найти удовлетворяющее условиям задачи распределение. Если это невозможно, просто выведите восклицательный знак «!».

Один вариант (двудольной) остаточной сети показан на рис. 4.27. Индексируем вершины [0..37], поскольку существует 26 + 10 + 2 особые вершины (источник и сток) = 38 вершин. Для источника *s* определим вершину индекса 0, для 26 возможных приложений определим значения индексов [1..26], для 10 компьютеров определим значения индексов [27..36], и, наконец, для стока *t* определим значение индекса 37.

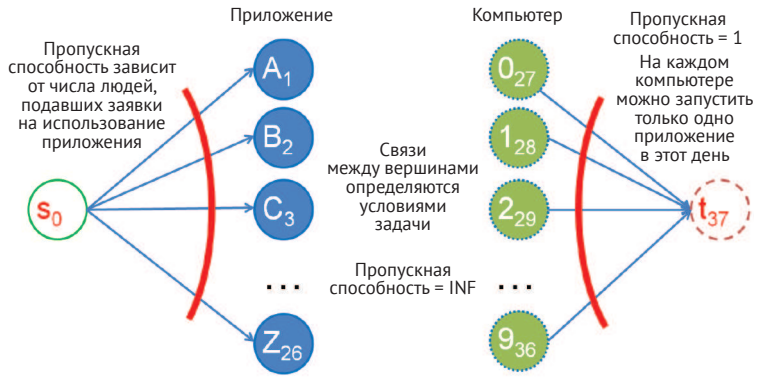


Рис. 4.27 ❖ Остаточная сеть для задачи UVa 259 [47]

[1] На самом деле эта задача имеет небольшой размер входных данных (у нас только 26 + 10 = 36 вершин плюс еще 2: источник и сток), благодаря чему она по-прежнему может быть решена с помощью возвратной рекурсии (см. раздел 3.2). Данная задача – задача о назначениях, или «задача о распределении», или (специальная) задача о паросочетании в двудольном графе с учетом пропускной способности.

Затем мы связываем приложения с компьютерами, как указано в постановке задачи. Мы соединяем источник *s* со всеми приложениями и соединяем все компьютеры со стоком *t*. Все ребра в этом графе потока являются направленными ребрами. Проблема заключается в том, что может существовать более одного пользователя, загружающего конкретное приложение *A* в определенный день (допустим, число таких пользователей *X*). Таким образом, мы задаем вес (пропускную способность) ребра, исходящего из источника *s* и входящего в конкретное приложение от *A* до *X*. В условиях задачи также сказано, что каждый компьютер может использоваться только один раз. Таким образом, мы устанавливаем вес ребра, исходящего из каждого компьютера *B* и входящего в сток *t*, равным 1. Вес ребер, ведущих от приложений к компьютерам, равняется ∞. При таких условиях, если существует поток от приложения *A* к компьютеру *B* и, наконец, к стоку *t*, этот поток соответствует одному сопоставлению между этим конкретным приложением *A* и компьютером *B*.

Как только мы построим этот граф потока, можем использовать на нем нашу реализацию алгоритма Эдмондса–Карпа, показанную ранее, чтобы получить значение максимального потока `mf`. Если `mf` равно количеству заявок, поданных в этот день, то у нас есть решение: если у нас есть *X* пользователей, подавших заявки на приложение *A*, то алгоритм Эдмондса–Карпа должен найти *X* различных путей (т. е. сопоставлений) от вершины *A* до стока *t* (для других приложений сопоставление выполняется аналогично).

Фактические связи приложение → компьютер можно найти, просто проверив обратные ребра, ведущие от компьютеров (вершины 27–36) к приложениям (вершины 1–26). Обратное ребро (компьютер → приложение) в остаточной матрице `res` будет содержать значение +1, если соответствующее прямое ребро (приложение → компьютер) включено в те пути, которые влияют на максимальный поток `mf`. Именно поэтому мы запускаем граф потока с направленными ребрами только от приложений к компьютерам.

Упражнение 4.6.4.1. Почему мы используем значение ∞ в качестве весов (пропускной способности) направленных ребер от приложений к компьютерам? Можем ли мы использовать значение пропускной способности 1 вместо ∞?

Упражнение 4.6.4.2*. Можно ли решить такую задачу о назначениях (задачу о паросочетании в двудольном графе с учетом пропускной способности) с помощью стандартного алгоритма нахождения максимального по мощности паросочетания на двудольном графе (Max Cardinality Bipartite Matching, MCBM), показанного далее в разделе 4.7.4? Если это возможно, определите, какое решение является лучшим.

4.6.5. Другие разновидности задач, использующих поток

Есть несколько других интересных применений / вариантов задач, связанных с потоком в сети. Здесь мы обсудим три примера, в то время как некоторые другие будут изложены в разделе 4.7.4 (двудольный граф), а также в разделах 9.13, 9.22 и 9.23. Обратите внимание, что некоторые приемы, показанные здесь, также могут быть применимы к другим задачам на графах.

Минимальный разрез графа

Определим *разрез* s–t как C = (S-, T-компонент) как разбиение $V \in G$ таким образом, что источник $s \in S$-компонент и сток $t \in T$-компонент. Давайте также определим реберный разрез C как множество $\{(u, v) \in E \mid u \in S$-компонент, $v \in T$-компонент$\}$ таким образом, что если все ребра в реберном разрезе C удалить, то максимальный поток от s к t равен 0 (то есть s и t отключены). Стоимость s–t разреза C определяется суммой пропускных способностей ребер в реберном разрезе C. Задача о минимальном разрезе, часто сокращенно обозначаемом как «Min Cut», заключается в минимизации пропускной способности разрезов s–t. Эта задача является более общей, чем задача о нахождении мостов (см. раздел 4.2.1), поскольку в подобном случае мы можем разрезать более одного ребра, и мы хотим сделать это с наименьшими затратами. Как и в случае решения задачи с нахождением мостов, задача о минимальном разрезе применяется в решении задач о «диверсиях» в сетях: например, задача UVa 10480 – Sabotage является одной из задач о минимальном разрезе.

Решение простое: побочный продукт вычисления максимального потока (Max Flow) – минимальный разрез (Min Cut)! Давайте снова посмотрим на рис. 4.24.D. После остановки алгоритма Max Flow мы опять запускаем обход графов (DFS/BFS) из источника s. Все вершины, достижимые из источника s, использующие ребра положительного веса в остаточном графе, принадлежат S-компоненту (то есть вершины 0 и 2). Все остальные недостижимые вершины принадлежат T-компоненту (то есть вершины 1 и 3). Все ребра, соединяющие S-компонент с T-компонентом, относятся к разрезу C (ребро 0–3 (пропускная способность 30 / поток 30 / остаточная пропускная способность 0), 2–3 (5/5/0) и 2–1 (25/25/0) в данном случае). Значение Min Cut составляет 30 + 5 + 25 = 60 = величине максимального потока `mf`. Это минимальное значение из всех возможных значений для всех разрезов s–t.

Более одного источника / Более одного стока

Иногда в графе может иметься более одного источника и/или более одного стока. Однако этот вариант не сложнее, чем исходная задача потока с одним источником и одним стоком. Создайте супериcточник *ss* и суперсток *st*. Соедините вершину *ss* со всеми s с помощью ребер с бесконечной пропускной способностью, также соедините все t с *st* при помощи ребер с бесконечной пропускной способностью, затем запустите алгоритм Эдмондса–Карпа, как обычно. Обратите внимание, что мы встречали этот вариант в упражнении 4.4.2.1.

Пропускные способности вершин

У нас также может быть вариант потока в сети, когда пропускная способность определяется не только на ребрах, но и на вершинах. Чтобы решить этот вариант задачи, мы можем использовать технику расщепления вершин, которая (к сожалению) удваивает количество вершин в графе. Взвешенный граф со взвешенными вершинами можно преобразовать в более привычный без взвешенных вершин, разделив каждую взвешенную вершину v на v_{in} и v_{out}, переназначив входящие/исходящие ребра на v_{in}/v_{out} соответственно и, наконец, определив вес исходной вершины v как вес ребра $v_{in} \rightarrow v_{out}$ (см. рис. 4.28).

Теперь, когда все веса определены на ребрах, мы можем запустить алгоритм Эдмондса–Карпа как обычно.

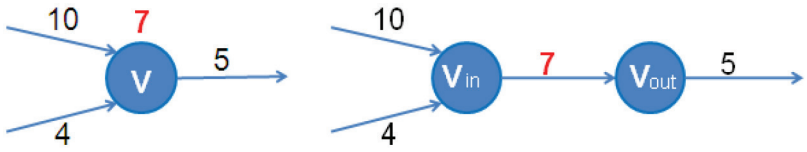


Рис. 4.28 ❖ Техника расщепления вершин

4.6.6. Моделирование графа потока – часть II

Самая сложная часть решения задачи о потоке – это моделирование графа потока (при условии что у нас уже есть заранее написанный код для нахождения максимального потока). В разделе 4.6.4 мы видели один пример моделирования графа для решения задачи о назначениях, или задачи о паросочетании в двудольном графе с учетом пропускной способности. Здесь мы представляем другой (более сложный) случай моделирования графа потока для решения задачи UVa 11380 – Down Went The Titanic. Прежде чем вы продолжите читать, нам хотелось бы дать вам один совет: пожалуйста, не только запомните решение, но и постарайтесь понять ключевые шаги для построения требуемого графа потока.

На рис. 4.29 у нас есть четыре небольших тестовых примера для задачи UVa 11380. Дана небольшая 2D-сетка, содержащая следующие пять символов, приведенных в табл. 4.5. Вы хотите поместить как можно больше «*» (человек) в безопасные места: «#» (большое бревно). Сплошные и пунктирные стрелки на рис. 4.30 обозначают ответ.

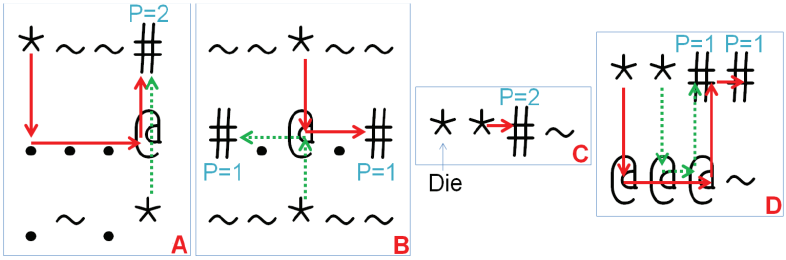


Рис. 4.29 ❖ Некоторые тестовые примеры для задачи UVa 11380

Таблица 4.4: Символы, используемые в задаче UVa 11380

Символ	Значение	Число случаев использования
*	Люди, оставшиеся на плавучей льдине	1
~	Ледяная вода	0
.	Плавучая льдина	1
@	Большой айсберг	∞
#	Большое бревно	∞

Для моделирования графа потока мы делаем следующее. На рис. 4.30A мы сначала соединяем вместе ячейки таблицы, не содержащие символ «~», с большой пропускной способностью (для этой задачи достаточно большим значением будет 1000). В результате мы получим описание возможных движений на сетке. На рис. 4.30B мы установили пропускную способность вершин ячеек «*» и «.» равной 1, чтобы указать, что они могут использоваться *только один раз*. Затем мы устанавливаем большое значение (1000) пропускной способности для вершин «@» и «#», чтобы указать, что они могут использоваться несколько раз. На рис. 4.30C мы создаем вершину-источник *s* и вершину-сток *t*. Источник *s* соединен со всеми ячейками «*» в сетке с пропускной способностью 1, чтобы указать, что нужно сохранить одного человека. Все ячейки «#» в сетке соединены со стоком *t* с пропускной способностью P, чтобы указать, что большое бревно может использоваться P раз. Теперь требуемый ответ – число выживших – соответствует максимальному значению потока между источником *s* и стоком *t* этого графа потока. Поскольку граф потока использует пропускные способности вершин, нам необходимо применять метод разделения вершин, обсуждавшийся ранее.

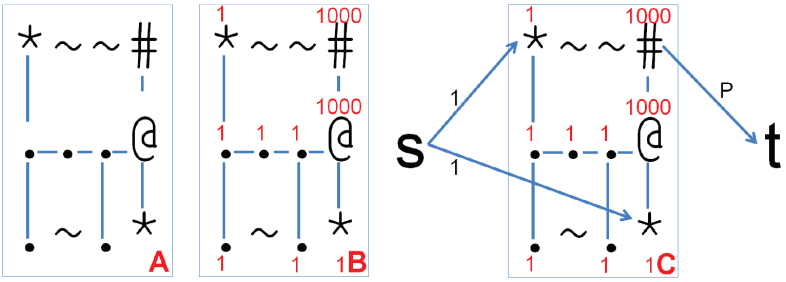


Рис. 4.30 ❖ Моделирование графа потока

Упражнение 4.6.6.1*. Достаточно ли быстр алгоритм Эдмондса–Карпа (временная сложность $O(VE^2)$) для вычисления максимального значения потока на максимально возможном графе для задачи UVa 11380 при следующих условиях: сетка 30×30 и $P = 10$? Почему?

Замечания о задачах на вычисление потока на олимпиадах по программированию

По состоянию на 2013 год, когда на олимпиаде по программированию появляется задача сетевого (обычно максимального) потока, это, как правило, является одной из тех задач, которые требуют мастерства при их решении. На соревнованиях ICPC многие интересные графовые задачи написаны таким образом, что они на первый взгляд не похожи на задачи о сетевом потоке. Самое сложное для участника олимпиады по программированию – это осознать, что такая задача на самом деле является задачей о сетевом потоке, и правильно смоделировать граф потока. Это ключевой навык, который нужно освоить на практике.

Чтобы не тратить драгоценное время соревнования на написание относительно длинного кода библиотеки для вычисления максимального потока, мы предлагаем следующий подход: пусть в команде, участвующей в ICPC, один из членов команды приложит значительные усилия, подготовив хороший код для вычисления максимального потока (как один из возможных вариантов – реализовав алгоритм Диница, см. раздел 9.7), и практикуется в решении различных задач, связанных с вычислением сетевого потока. Эти задачи доступны во многих онлайн-сборниках задач и архивах задач, и мы рекомендуем этому члену команды практиковаться, чтобы набраться опыта в решении задач о сетевом потоке и их разновидностей. Упражнения в этом разделе содержат несколько простых значений на вычисление максимального потока, задач о паросочетании в двудольном графе с учетом пропускной способности (или задач о распределении), задач о минимальном разрезе и задач о сетевом потоке, связанных с пропускной способностью вершин. Постарайтесь выполнить как можно больше упражнений и решить как можно больше задач.

В разделе 4.7.4 мы рассмотрим классическую задачу нахождения максимального по мощности паросочетания на двудольном графе (Max Cardinality Bipartite Matching, MCBM) и увидим, что эта задача также может быть решена с помощью вычисления максимального потока (Max Flow). Позже в главе 9 мы увидим некоторые более сложные задачи, связанные с сетевым потоком, например более быстрый алгоритм вычисления максимального потока (раздел 9.7), задачи о нахождении независимого и реберно не пересекающегося пути (раздел 9.13), задачи о независимом множестве *максимального веса* на двудольном графе (раздел 9.22) и задачи о максимальном потоке с минимальной стоимостью (раздел 9.23).

В соревнованиях IOI задачи о сетевом потоке (и их варианты) в настоящее время находятся за пределами программы 2009 года [20]. Таким образом, участники IOI могут пропустить этот раздел. Тем не менее мы считаем, что для участников IOI было бы неплохо изучить этот более сложный материал «заблаговременно», чтобы улучшить свои навыки в решении задач на графы.

Задачи по программированию, связанные с вычислением сетевого потока

- Стандартные задачи на нахождение максимального потока (алгоритм Эдмондса–Карпа)
  1. **UVa 00259 – Software Allocation** * (обсуждается в этом разделе)
  2. **UVa 00820 – Internet Bandwidth** * (LA 5220, финальные соревнования на кубок мира, Орландо'00, базовая задача на нахождение максимального потока, обсуждается в этом разделе)
  3. UVa 10092 – The Problem with the… (задача о назначениях, сопоставление с учетом пропускной способности, решается аналогично UVa 259)
  4. UVa 10511 – Councilling (сопоставление, максимальный поток, необходимо распечатать назначения)
  5. *UVa 10779 – Collectors Problem* (неочевидное моделирование максимального потока; основная идея состоит в том, чтобы построить

граф потока таким образом, чтобы каждая увеличивающая цепь соответствовала серии обменов стикерами, у которых имеются дубликаты, начиная с Боба, раздающего один из его дубликатов, и заканчивая тем, что он получает новую наклейку; повторяйте эту процедуру, пока данный обмен станет невозможным)

6. UVa 11045 – My T-Shirt Suits Me (задача назначениях; но на самом деле входное ограничение достаточно мало, чтобы было возможно использовать возвратную рекурсию)
7. ***UVa 11167 – Monkeys in the Emei... **** (моделирование максимального потока; на графе потока много ребер; поэтому лучше сжимать ребра пропускной способности – 1, когда это возможно; используйте алгоритм Диница с временной сложностью $O(V^2E)$ для нахождения максимального потока, чтобы большое количество ребер не снижало производительность вашего решения)
8. UVa 11418 – Clever Naming Patterns (двуслойное сопоставление, может быть, проще использовать решение задачи о максимальном потоке)

- Другие задачи
  1. UVa 10330 – Power Transmission (задача о максимальном потоке с пропускными способностями вершин)
  2. UVa 10480 – Sabotage (несложная задача о минимальном разрезе)
  3. ***UVa 11380 – Down Went The Titanic **** (обсуждается в этом разделе)
  4. **UVa 11506 – Angry Programmer *** (задача о минимальном разрезе с пропускными способностями вершин)
  5. ***UVa 12125 – March of the Penguins **** (моделирование максимального потока с пропускными способностями вершин; другая интересная задача, аналогичная по уровню с задачей UVa 11380)

4.7. Специальные графы

Некоторые базовые задачи на графах имеют более простые / быстрые полиномиальные алгоритмы, если исходный граф относится к категории *специальных графов*. Основываясь на нашем опыте, мы определили следующие специальные графы, которые обычно предлагаются на олимпиадах по программированию: ориентированный ациклический граф, дерево, эйлеров граф и двудольный граф. Авторы задачи могут заставить участников использовать специальные алгоритмы для этих специальных графов, увеличив размер входных данных таким образом, что при использовании правильного алгоритма для общего случая задача не будет считаться решенной, получив вердикт «превышение лимита времени» (TLE) (см. [21]).

В этом разделе мы обсудим подходы к решению некоторых популярных задач из области теории графов для этих специальных графов (см. рис. 4.31). Многие из этих задач обсуждались ранее для общих случаев. Обратите внимание, что на момент написания данной книги двудольные графы (раздел 4.7.4) все еще не входят в программу IOI [20].

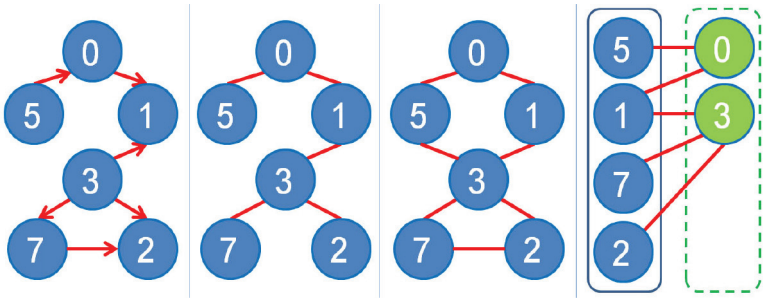


Рис. 4.31 ❖ Специальные графы (слева направо): ориентированный ациклический граф, дерево, эйлеров граф, двудольный граф

4.7.1. Направленный ациклический граф

Направленный ациклический граф (Directed Acyclic Graph, DAG) – это специальный граф, обладающий следующими характеристиками: направленный и не имеющий цикла. В направленном ациклическом графе циклы отсутствуют по определению. Благодаря этому задачи, которые можно смоделировать как DAG, могут быть решены с помощью методов динамического программирования (DP) (см. раздел 3.5). В конце концов, переходы в DP должны быть ациклическими. Мы можем рассматривать состояния DP как вершины неявного DAG, а ациклические переходы между состояниями DP – как направленные ребра этого неявного DAG. Топологическая сортировка данного DAG (см. раздел 4.2.1) позволяет обрабатывать каждую вложенную подзадачу (подграф DAG) только один раз.

(Одна исходная вершина) Кратчайшие / самые длинные пути на DAG

Задача о кратчайших путях из одной вершины до любой другой вершины графа (SSSP) становится намного проще, если данный граф является DAG. Это связано с тем, что DAG имеет хотя бы одну топологическую сортировку! Мы можем использовать алгоритм топологической сортировки с временной сложностью $O(V + E)$, описанный в разделе 4.2.1, чтобы найти одну такую топологическую сортировку, а затем ослабить исходящие ребра этих вершин в соответствии с данным порядком. Топологическая сортировка гарантирует, что если у нас есть вершина b, у которой есть входящее ребро из вершины a, то для вершины b выполняется операция `relax` *после* того, как вершина a получит правильное значение кратчайшего расстояния. Таким образом, значения наименьших расстояний будут получены для всего графа в целом только за один проход, при последовательном просмотре данных (временная сложность составит $O(V + E)$)! В этом заключается суть метода динамического программирования, позволяющего избежать повторного вычисления вложенной подзадачи, описанной ранее в разделе 3.5. Когда мы используем восходящий метод DP, мы, в сущности, заполняем таблицу DP, используя топологическую сортировку основного неявного направленного ациклического графа повторений DP, лежащего в основе этого метода.

Задача о самых длинных путях (с одной исходной вершиной)[1] – это задача нахождения самых длинных (простых[2]) путей от начальной вершины *s* до других вершин. Вариант решения этой задачи NP-полон на графе в общем случае[3]. Однако задача снова становится простой, если у графа нет цикла, что верно для DAG. Для решения задачи о самых длинных путях на DAG[4] потребуется лишь внести несколько незначительных изменений в решение методом DP для решения задачи SSSP на DAG, приведенное выше. Первое из этих изменений – умножить все веса ребер на –1 и запустить то же самое решение SSSP, что и представленное выше. Наконец, инвертируйте полученные значения, чтобы получить фактические результаты.

У задачи о самом длинном пути на DAG есть применение в области планирования проектов, как, например, показано в задаче UVa 452 – Project Scheduling. Мы можем смоделировать зависимость подпроекта как DAG, а время, необходимое для завершения подпроекта, как *вес вершины*. Наименьшее возможное время для завершения всего проекта определяется самым длинным путем в этом DAG (т. е. *критическим* путем), который начинается с любой вершины (подпроекта) с входящей степенью, равной 0. См. рис. 4.32, где показан пример с 6 подпроектами, их оценочными временами завершения и зависимостями. Самый длинный путь 0 → 1 → 2 → 4 → 5, занимающий 16 единиц времени, определяет минимальный срок для завершения всего проекта. Чтобы проект завершился в этот срок, все подпроекты, расположенные на самом длинном (критическом) пути, должны завершиться вовремя.

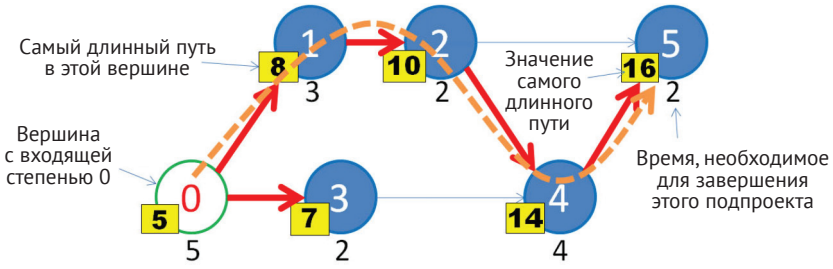


Рис. 4.32 ❖ Самый длинный путь в этом DAG

---

[1] На самом деле это может быть несколько источников, так как мы можем начать с любой вершины с входящей степенью 0.

[2] Для общего случая графа с положительными взвешенными ребрами задача о самом длинном пути плохо определена, поскольку можно взять положительный цикл (цикл положительного веса) и использовать этот цикл для построения бесконечно длинного пути. Это та же самая проблема, что и проблема, возникающая с отрицательным циклом (циклом отрицательного веса) в задаче нахождения кратчайшего пути. Вот почему для графа общего вида мы используем термин «задача о самом длинном *простом* пути». Все пути в DAG просты по определению, поэтому в этом особом случае мы можем просто использовать термин «задача о самом длинном пути».

[3] В приведенном варианте решения этой задачи спрашивается, имеет ли исследуемый граф общего вида простой путь с полным весом $\geqslant k$.

[4] Задача о наибольшей возрастающей подпоследовательности (LIS), приведенная в разделе 3.5.2, также может быть смоделирована как поиск самых длинных путей в неявном DAG.

Подсчет путей в DAG

Задача (UVa 988 – Many paths, one destination): в жизни у каждого есть много путей, которые можно выбрать, что приведет ведущих ко множеству разных жизненных сценариев. Перечислите, сколько разных жизненных сценариев можно прожить, учитывая определенный набор вариантов в каждый момент времени. Один из них содержит список событий и ряд вариантов, которые можно выбрать для каждого события. Цель состоит в том, чтобы подсчитать, сколько способов пройти от события, с которого все началось (рождение, индекс 0), до события, когда у человека больше нет выбора (то есть смерть, индекс n).

Ясно, что основной граф приведенной выше задачи – DAG, поскольку можно двигаться вперед во времени, но невозможно возвращаться назад. Количество таких путей может быть легко найдено путем вычисления одной (любой) топологической сортировки с временной сложностью $O(V + E)$ (в этой задаче вершина 0/рождение всегда будет первой в топологической сортировке, а вершина n/смерть всегда будет оставаться последней). Мы начинаем с того, что устанавливаем значение `num_paths[0] = 1`. Затем обрабатываем оставшиеся вершины одну за другой в соответствии с топологической сортировкой. При обработке вершины `u` мы обновляем каждую ее соседнюю вершину, `v`, устанавливая `num_paths[v] += num_paths[u]`. После $O(V + E)$ таких шагов мы узнаем количество путей, выражаемое как `num_paths[n]`. На рис. 4.33 показан пример, в котором определено девять событий и в конечном счете шесть различных возможных сценариев жизни.

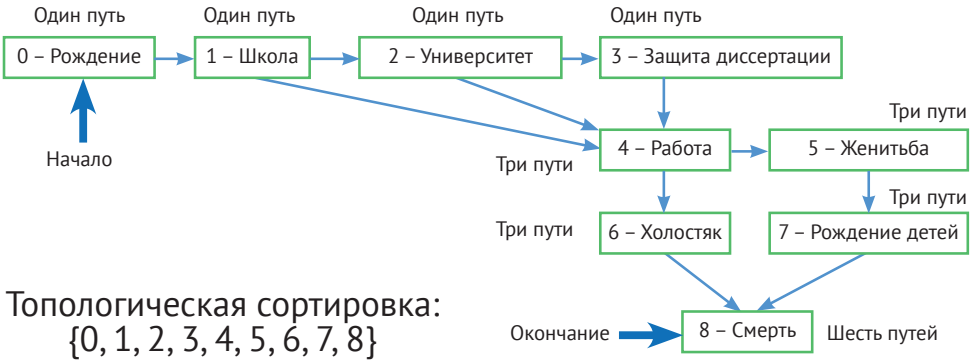


Рис. 4.33 ❖ Пример подсчета путей в DAG – снизу вверх

Реализации, использующие восходящий и нисходящий методы

Прежде чем продолжить, мы хотим отметить, что все три решения для нахождения кратчайших / самых длинных путей и подсчета числа путей на/в DAG, приведенных выше, – это метод восходящего DP. Мы начинаем с известного базового(ых) случая(ев) (исходной(ых) вершины/вершин), а затем используем топологическую сортировку DAG для распространения правильных данных на соседние вершины, не возвращаясь при этом к предыдущим шагам.

В разделе 3.5 мы видели, что существует также реализация нисходящего DP. Рассматривая задачу UVa 988 в качестве примера, мы также можем написать

решение DP следующим образом: пусть `numPaths(i)` будет количеством путей, начинающихся с вершины *i* до конечной вершины *n*. Мы можем написать решение, используя следующие рекуррентные соотношения для полного перебора:

1) `numPaths(n) = 1 // в конечной вершине n существует лишь один возможный путь;`
2) `numPaths(i)` = $\sum_j$ `numPaths(j), ∀j, где j - вершина, смежная с i.`

Чтобы избежать повторных вычислений, мы запоминаем количество путей для каждой вершины *i*. Есть $O(V)$ различных вершин (состояний), и каждая вершина обрабатывается только один раз. Имеется $O(E)$ ребер, и каждое ребро также посещается не более одного раза. Следовательно, временная сложность этого подхода нисходящего DP (когда мы спускаемся по дереву сверху вниз) равна $O(V + E)$ так же, как подход восходящего DP, показанный ранее. На рис. 4.34 показан аналогичный DAG, но значения вычисляются, начиная с конечной вершины и доходя до начальной вершины (следуйте по пунктирным стрелкам назад). Сравните рис. 4.34 с предыдущим рис. 4.33, где значения вычисляются, начиная с начальной вершины и заканчивая конечной вершиной.

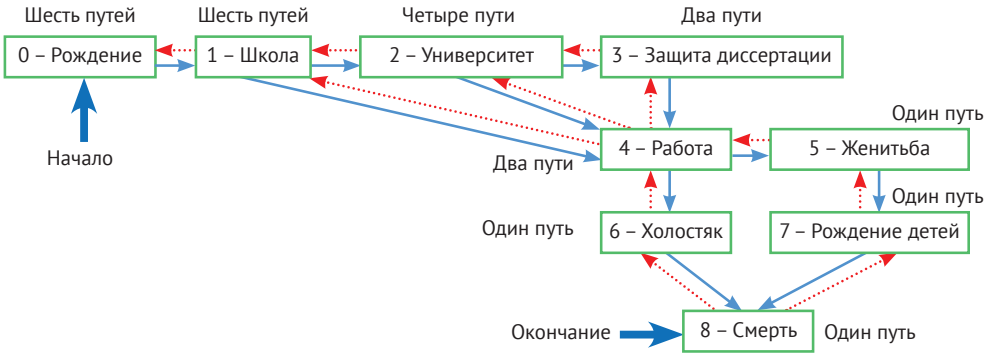


Рис. 4.34 ❖ Пример подсчета путей в DAG – нисходящий метод

Преобразование графа общего вида в DAG

Иногда исходный граф, приведенный в постановке задачи, не является явным DAG. Однако после дальнейшего исследования данный граф может быть смоделирован как DAG, если мы добавим один (или более) параметр(ов). После того как вы получите DAG, следующим шагом будет применение метода динамического программирования (нисходящего или восходящего). Мы проиллюстрируем эту концепцию на двух примерах.

1. SPOJ 0101: Fishmonger

Сокращенная формулировка условия задачи: пусть задано некоторое количество городов $3 \leqslant n \leqslant 50$, доступное время $1 \leqslant t \leqslant 1000$ и две матрицы $n \times n$ (одна содержит время в пути, а другая – пошлины за проезд между двумя городами). Выберите маршрут из города-порта (вершина 0) таким образом, чтобы торговец рыбой заплатил как можно меньше сборов, чтобы прибыть в город, где имеется рынок (вершина $n - 1$), за время, не превышающее определенное зна-

чение *t*. Торговец рыбой не должен посещать все города. В качестве выходных данных выведите два параметра: общее количество фактически взимаемых сборов и фактическое время в пути. См. рис. 4.35 (слева), где показан исходный граф для этой задачи.

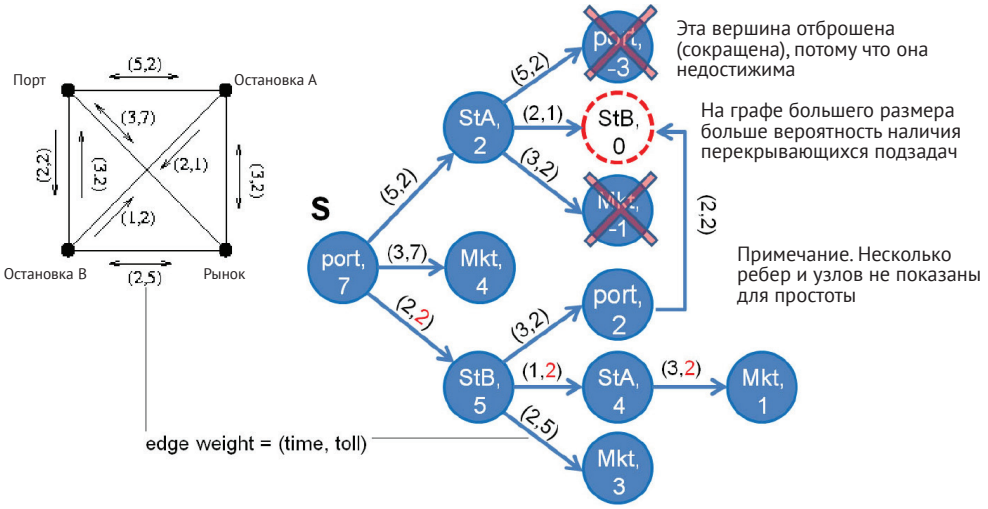


Рис. 4.35 ❖ Исходный граф общего вида (слева) превращается в DAG

Обратите внимание, что в этой задаче есть два потенциально противоречивых требования. Первым требованием является минимизация платы за проезд по маршруту. Второе требование – обеспечить прибытие торговца рыбой в рыночный город в назначенное время, что может привести к тому, что он заплатит более высокие пошлины в некоторой части пути. Второе требование – жесткое ограничение для этой задачи. Мы должны его удовлетворить, в противном случае решение будет отсутствовать.

«Жадный» алгоритм SSSP, такой как алгоритм Дейкстры (см. раздел 4.4.3) – в чистом виде, – не может использоваться для этой задачи. Выбор пути с наименьшим временем прохождения, чтобы помочь торговцу рыбой добраться до рыночного города $n - 1$, за время $\leqslant t$, может не привести к наименьшим возможным потерям. Выбор пути с самыми низкими пошлинами не гарантирует, что торговец рыбой прибывает в рыночный город $n - 1$ за время $\leqslant t$. Эти два требования не являются независимыми!

Однако если мы добавим параметр `t_left` (оставшееся время) для каждой вершины, то данный граф превратится в DAG, как показано на рис. 4.35 (справа). Начнем с вершины (`port`, `t`) в полученном DAG. Каждый раз, когда торговец рыбой перемещается из текущего города в другой город `X`, мы перемещаемся в измененную вершину (`X`, `t – travelTime[cur][X]`) в DAG через ребро с весом `toll[cur][X]`. Поскольку время сокращается, мы никогда не столкнемся с ситуацией цикла на нашем графе. Затем мы можем использовать это (нисходящее) повторение DP: `go(cur,t_left)`, чтобы найти кратчайший путь (с точки зрения общего количества оплаченных дорожных сборов) для этого DAG. Ответ можно

найти, вызвав функцию `go(0,t)`. Код на C++ для реализации `go(cur, t_left)` приведен ниже:

```
ii go(int cur, int t_left) {                    // возвращает пару (tollpaid, timeneeded)
  if (t_left < 0) return ii(INF, INF);             // недопустимое состояние, отбрасываем
  if (cur == n - 1) return ii(0, 0);         // торговец на рынке, tollpaid=0, timeneeded=0
  if (memo[cur][t_left] != ii(-1, -1)) return memo[cur][t_left];
  ii ans = ii(INF, INF);
  for (int X = 0; X < n; X++) if (cur != X) {                      // едем в другой город
    ii nextCity = go(X, t_left - travelTime[cur][X]);                  // рекурсивный шаг
    if (nextCity.first + toll[cur][X] < ans.first) {      // выбираем минимальную стоимость
      ans.first = nextCity.first + toll[cur][X];
      ans.second = nextCity.second + travelTime[cur][X];
  } }
  return memo[cur][t_left] = ans; }                      // сохраняем ответ в таблице memo
```

Обратите внимание, что при использовании нисходящего метода DP нам не нужно явно создавать DAG и вычислять топологическую сортировку. Рекурсия сделает эти шаги за нас. Есть только $O(nt)$ различных состояний (обратите внимание, что таблица memo хранит пары). Каждое состояние может быть найдено за $O(n)$. Таким образом, общая временная сложность составляет $O(n^2t)$ – этот метод можно использовать.

2. Минимальное покрытие вершин (на дереве)

Древовидная структура данных также является ациклической структурой данных. Но, в отличие от DAG, в дереве нет пересекающихся поддеревьев. Таким образом, нет смысла использовать технику динамического программирования (DP) на стандартном дереве. Однако, аналогично приведенному выше примеру задачи о торговце рыбой, некоторые деревья на олимпиадах по программированию превращаются в DAG, если мы закрепляем один (или более) параметр(ов) за каждой вершиной дерева. Тогда решение обычно состоит в том, чтобы запустить DP на получающемся DAG. Такие задачи (неудачно с точки зрения строгости терминологии[1]) называются задачами «DP на дереве» в олимпиадном программировании.

Примером задачи DP на дереве является задача нахождения минимального вершинного покрытия (MVC) на дереве. В этой задаче нам нужно выбрать наименьший возможный набор вершин $C \in V$, чтобы каждое ребро дерева попадало по крайней мере в одну вершину множества C. Для примера дерева, показанного на рис. 4.36 (слева), решение такой задачи – выбрать только вершину 1, потому что все ребра 1–2, 1–3, 1–4 соединены с вершиной 1.

Теперь есть только две возможности для каждой вершины. Либо она выбрана, либо нет. Закрепляя этот статус «выбрана или не выбрана» за каждой вершиной, мы преобразуем исходное дерево в DAG (см. рис. 4.36 (справа)). У каждой вершины теперь есть пара параметров (номер вершины, логический флаг выбран / нет).

---

[1] Мы упоминали, что нет смысла использовать DP на дереве. Но термин «DP на дереве», который фактически относится к «DP на неявном DAG», уже является устоявшимся термином в сообществе программистов-«олимпиадников».

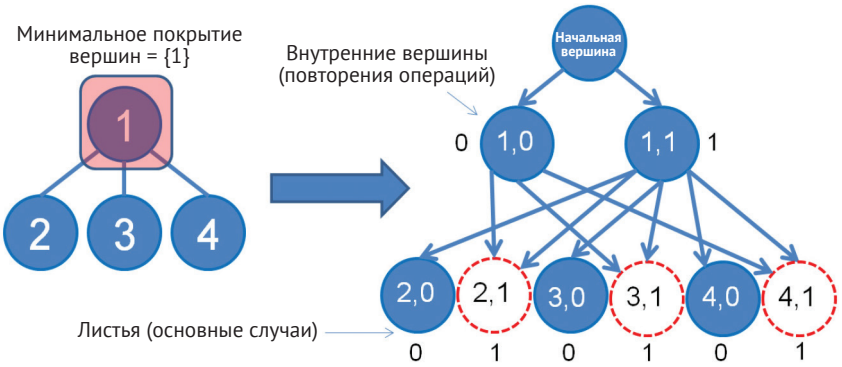


Рис. 4.36 ❖ Данный граф общего вида / дерево (слева) преобразуется в DAG

Неявные ребра определяются по следующим правилам: 1) если текущая вершина не выбрана, то мы должны выбрать всех ее потомков, чтобы получить правильное решение; 2) если текущая вершина выбрана, то мы выбираем лучшее между выбором или невыбором ее потомков. Теперь мы можем записать эту рекурсию нисходящего DP: `MVC(v, flag)`. Ответ можно найти, взяв `min(MVC(root, false), MVC(root, true))`. Обратите внимание на наличие перекрывающихся подзадач (обозначенных пунктирными кружками) в DAG. Однако, поскольку существует только $2 \times V$ состояний и каждая вершина имеет не более двух входящих ребер, это решение DP имеет временную сложность $O(V)$.

```
int MVC(int v, int flag) {                                  // Минимальное вершинное покрытие
  int ans = 0;
  if (memo[v][flag] != -1) return memo[v][flag];                          // нисходящее DP
  else if (leaf[v])            // leaf[v] истинно, если v - лист, в противном случае - нет
    ans = flag;                                                          // 1/0 = выбрана/нет
  else if (flag == 0) {          // если v не будет выбрана, мы должны выбрать ее потомков
    ans = 0;              // Примечание: "Потомки" - это список смежности, который содержит
                          // направленную версию дерева (родитель указывает на своих
                          // потомков но потомки не указывают на родителей)
    for (int j = 0; j < (int)Children[v].size(); j++)
      ans += MVC(Children[v][j], 1);
  }
  else if (flag == 1) {                       // если v выбрана, берем минимальное значение
    ans = 1;                                        // выбираем или не выбираем ее потомков
    for (int j = 0; j < (int)Children[v].size(); j++)
      ans += min(MVC(Children.[v][j], 1), MVC(Children[v][j], 0));
  }
  return memo[v][flag] = ans;
}
```

Раздел 3.5 – повторение

Здесь мы хотим еще раз подчеркнуть читателям тесную связь между методами DP, показанными в разделе 3.5, и алгоритмами на DAG. Обратите внимание, что все упражнения по программированию, связанные с нахождением кратчайших / самых длинных путей и подсчета числа путей в DAG (или на графе

общего вида, который преобразуется в DAG с помощью некоторого моделирования/преобразования графа), также могут быть отнесены к категории DP. Часто, когда у нас возникает задача с решением методом DP, которое «минимизирует это», «максимизирует это» или «подсчитывает что-то», такое решение DP фактически вычисляет самый короткий путь, самый длинный путь или подсчитывает количество путей на/в (обычно неявной) DP-рекурсии на DAG этой задачи соответственно.

Теперь мы приглашаем читателей вернуться к некоторым из задач DP, с которыми мы встречались ранее в разделе 3.5, рассмотрев их с этой, вероятно новой, точки зрения (рассмотрение DP как алгоритмов на DAG обычно не встречается в других учебниках по информатике). Для начала мы вернемся к классической задаче размена монет. На рис. 4.37 показан тот же пример, который использовался в разделе 3.5.2. Существует $n = 2$ достоинства монет: {1, 5}. Сумма, которую нужно набрать, $V = 10$. Мы можем смоделировать каждую вершину как текущее значение. Каждая вершина v имеет $n = 2$ невзвешенных ребра, которые входят в вершину $v - 1$ и $v - 5$ в этом тестовом примере, если только это не приводит к отрицательному индексу. Обратите внимание, что граф в рассматриваемом примере является DAG, и некоторые состояния (выделены пунктирными окружностями) перекрываются (имеют более одного входящего ребра). Теперь мы можем решить эту задачу, найдя кратчайший путь на этом DAG от исходной вершины $V = 10$ до конечной вершины $V = 0$. Самый простой способ топологической сортировки – это обработка вершин в обратном порядке, т. е. {10, 9, 8, ..., 1, 0}, что является допустимым. Мы определенно можем использовать кратчайшие пути $O(V + E)$ в решении DAG. Однако, поскольку граф невзвешенный, мы также можем использовать $O(V + E)$ BFS для решения данной проблемы (использование алгоритма Дейкстры тоже возможно, но излишне). Путь: 10 → 5 → 0 – самый короткий путь с общим весом = 2 (иными словами, нужно две монеты). Примечание: в этом тестовом примере «жадный» алгоритм для размена монет также выберет тот же путь: 10 → 5 → 0.

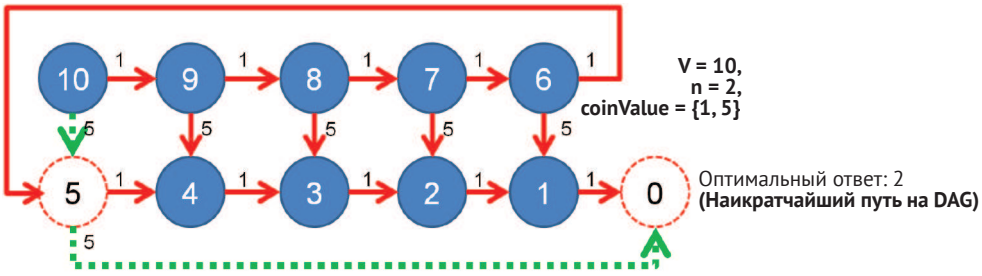


Рис. 4.37 ❖ Размен монет как нахождение кратчайших путей на DAG

Теперь давайте вернемся к классической задаче о рюкзаке (Рюкзак 0–1). На этот раз мы используем следующие данные для тестового примера: $n = 5$, $V = \{4, 2, 10, 1, 2\}$, $W = \{12, 1, 4, 1, 2\}$, $S = 15$. Мы можем смоделировать каждую вершину в виде пары значений (`id, remW`). Каждая вершина имеет по крайней мере одно ребро, исходящее из (`id, remW`) и входящее в (`id + 1, remW`), которое соответствует тому случаю, когда предмет с номером `id`, не берется в рюкзак. Некоторые вершины имеют ребро, исходящее из (`id, remW`) и входящее в (`id + 1, remW-W[id]`), если `W[id] ≤ remW`, что соответствует взятию предмета с номером `id`. На рис. 4.38 показаны некоторые фрагменты вычислений на DAG для обычной задачи о рюкзаке (Рюкзак 0–1) для приведенного выше контрольного примера. Обратите внимание, что некоторые состояния могут посещаться несколькими путями (перекрывающаяся подзадача выделена пунктирной окружностью). Теперь мы можем решить эту задачу, найдя самый длинный путь на этом DAG от исходной вершины (0, 15) до целевой вершины (5, любой id). Ответом является следующий путь: (0, 15) → (1, 15) → (2, 14) → (3, 10) → (4, 9) → (5, 7) с весом 0 + 2 + 10 + 1 + 2 = 15.

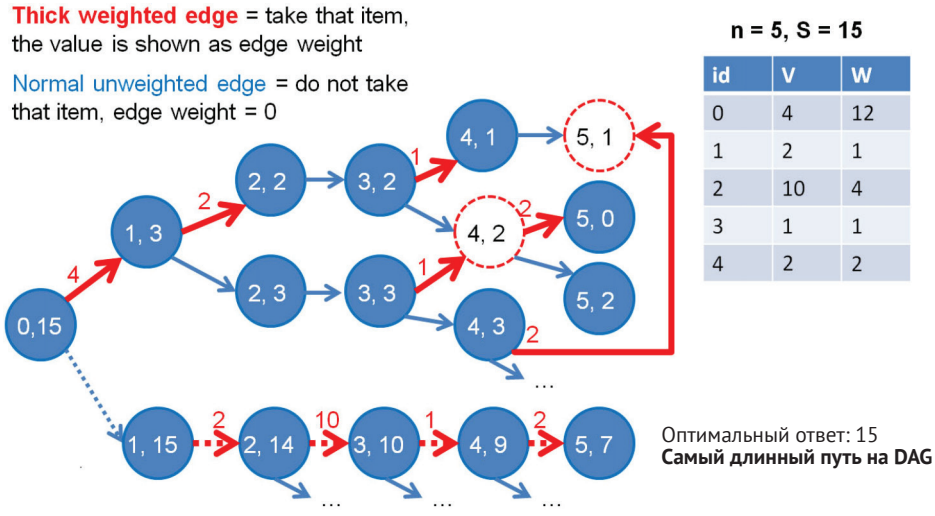


Рис. 4.38 ❖ Задача «Рюкзак 0–1» как задача о нахождении самых длинных путей на DAG

Рассмотрим еще один пример: решение задачи UVa 10943 – How do you add?, которая обсуждается в разделе 3.5.3. Если мы нарисуем граф для тестового примера для этой задачи: $n = 3$, $K = 4$, то у нас получится DAG, как показано на рис. 4.39. Есть перекрывающиеся подзадачи, выделенные пунктирными кругами. Если мы посчитаем количество путей в этом DAG, то действительно найдем ответ: 20 путей.

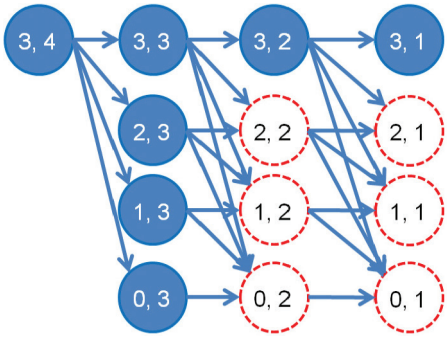


Рис. 4.39 ❖ Решение задачи UVa 10943 как подсчет путей в DAG

Упражнение 4.7.1.1*. Нарисуйте DAG для некоторых тестовых примеров других классических задач DP, не упомянутых выше: задачи коммивояжера (TSP) ≈ кратчайшие пути на неявном DAG, задачи о наибольшей возрастающей подпоследовательности (LIS) ≈ самые длинные пути на неявном DAG, вариант «подсчет сдачи» (метод подсчета количества возможных способов получения значения V центов с использованием списка номиналов N монет) – подсчет числа возможных путей в DAG и т. д.

4.7.2. Дерево

Дерево – это специальный граф со следующими характеристиками: у него есть $E = V - 1$ (любой алгоритм $O(V + E)$ для дерева – это $O(V)$), у него отсутствуют циклы, он является связным, и в нем существует один уникальный путь между любыми двумя вершинами.

Обход дерева

В разделах 4.2.1 и 4.2.2 мы рассмотрели алгоритмы DFS и BFS для обхода графа общего вида с временной сложностью $O(V + E)$. Если данный граф является *корневым двоичным деревом*, существуют более простые алгоритмы обхода дерева, такие как обход в прямом порядке (pre-order traversal), ориентированный обход (in-order traversal) и обход в обратном порядке (post-order traversal) (примечание: обход по уровням (level-order traversal) по сути является BFS). В данном случае нет существенного ускорения во времени, так как эти алгоритмы обхода дерева также работают на $O(V)$, но код, реализующий их, более прост. Их псевдокод показан ниже:

```
pre-order(v)
  visit(v);
  pre-order(left(v));
  pre-order(right(v));
```

```
in-order(v)
  in-order(left(v));
  visit(v);
  in-order(right(v));
```

```
post-order(v)
  post-order(left(v));
  post-order(right(v));
  visit(v);
```

Нахождение точек сочленения и мостов на дереве

В разделе 4.2.1 мы познакомились с алгоритмом DFS Тарьяна для нахождения точек сочленения и мостов графа, имеющим временную сложность $O(V + E)$. Однако если данный граф является деревом, задача упрощается: все ребра дерева являются мостами, а все внутренние вершины (со степенью > 1) – точками сочленения. Этот алгоритм все еще имеет временную сложность $O(V)$, так как мы должны последовательно просмотреть дерево, чтобы посчитать количество внутренних вершин, но код, реализующий этот вариант, *проще*.

Кратчайшие пути из одного источника на взвешенном дереве

В разделах 4.4.3 и 4.4.4 мы рассмотрели два алгоритма общего назначения ($O((V + E)\log V)$ Дейкстры и $O(VE)$ Форда–Беллмана) для решения задачи о кратчайших путях из одной вершины до любой другой вершины графа (SSSP) на взвешенном графе. Но если данный граф является взвешенным деревом, задача SSSP становится проще: любой алгоритм обхода графа имеет временную сложность $O(V)$, т. е. как BFS, так и DFS может быть использован для решения этой задачи. Между любыми двумя вершинами дерева существует только один уникальный путь, поэтому мы просто обходим дерево, чтобы найти уникальный путь, соединяющий две вершины. Вес кратчайшего пути между этими двумя вершинами – это фактически сумма весов ребер этого уникального пути (например, от вершины 5 до вершины 3 на рис. 4.41A, уникальный путь 5 → 0 → 1 → 3 с весом 4 + 2 + 9 = 15).

Кратчайшие расстояния между всеми вершинами (APSP) на взвешенном дереве

В разделе 4.5 мы познакомились с алгоритмом общего назначения (алгоритмом Флойда–Уоршелла с временной сложностью $O(V^3)$) для решения задачи APSP на взвешенном графе. Однако если данный граф является взвешенным деревом, задача APSP упрощается: повторите SSSP для взвешенного дерева V раз, устанавливая каждую вершину как исходную, одну за другой по очереди. Общая временная сложность для этого случая составляет $O(V^2)$.

Диаметр взвешенного дерева

Для графа общего вида нам понадобится алгоритм Флойда–Уоршелла с временной сложностью $O(V^3)$, приведенный в разделе 4.5, а также еще одна проверка всех пар $O(V^2)$ для вычисления диаметра. Однако если данный граф является взвешенным деревом, задача упрощается. Нам нужны только два обхода $O(V)$: выполните DFS/BFS из любой вершины s, чтобы найти самую дальнюю вершину x (например, выполните поиск из вершины $s = 1$ до вершины $x = 2$ на рис. 4.40B1), затем выполните еще раз DFS/BFS из вершины x, чтобы получить истинную вершину y, наиболее удаленную от x. Длина уникального пути вдоль x до y является диаметром этого дерева (например, путь $x = 2 \rightarrow 3 \rightarrow 1 \rightarrow 0 \rightarrow y = 5$, длина которого равна 20, на рис. 4.40B2).

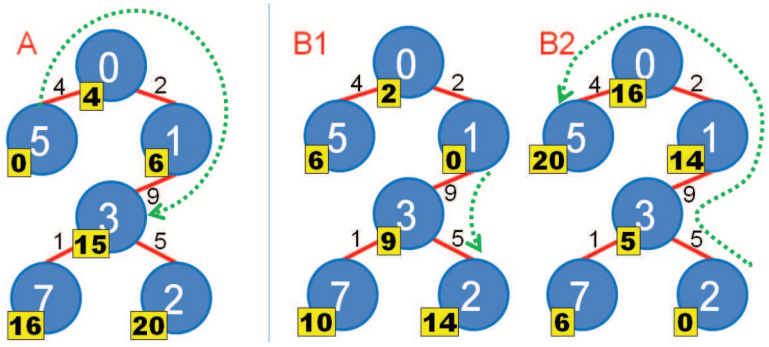


Рис. 4.40 ❖ A: SSSP (часть APSP); B1–B2: диаметр дерева

Упражнение 4.7.2.1*. Пусть задан обход в прямом порядке T корневого дерева двоичного поиска (BST), содержащий n вершин. Напишите рекурсивный псевдокод для вывода обхода в обратном порядке этого BST. Какова временная сложность вашего лучшего алгоритма?

Упражнение 4.7.2.2*. Существует еще более быстрое решение, чем $O(V^2)$, для задачи «Кратчайшие расстояния между всеми вершинами» на взвешенном дереве. Оно реализуется с использованием наименьшего общего предка (Lowest Common Ancestor, LCA). Какова реализация этого варианта?

4.7.3. Эйлеров граф

Путь Эйлера определяется как путь на графе, который проходит через каждое ребро графа ровно один раз. Аналогично обход/цикл Эйлера – это путь Эйлера, который начинается и заканчивается в одной и той же вершине. Граф, имеющий путь Эйлера, или обход Эйлера, называется эйлеровым графом[1].

Этот тип графов впервые был изучен Леонардом Эйлером при решении задачи о семи мостах Кенигсберга в 1736 году. Открытие Эйлера «положило начало» области теории графов!

Проверка эйлерова графа

Проверить, существует ли у связного неориентированного графа эйлеров цикл, несложно. Нам просто нужно проверить, все ли вершины имеют четные степени. Он тождествен существованию пути Эйлера, то есть ориентированный граф имеет путь Эйлера, если все вершины, кроме двух вершин, имеют четные степени. Этот путь Эйлера начнется с одной из этих вершин нечетной степени и закончится в другой[2]. Такая проверка может быть выполнена на

[1] Сравните это свойство с гамильтоновым путем/циклом в задаче TSP (см. раздел 3.5.2).

[2] Также возможен путь Эйлера на ориентированном графе: граф должен быть слабосвязным, иметь равные входящие/исходящие степени вершин, не более одной вершины с разностью (входящая степень – исходящая степень) = 1 и не более одной вершины с разностью (исходящая степень – входящая степень) = 1.

$O(V + E)$, обычно она выполняется одновременно с чтением входного графа. Вы можете попробовать выполнить эту проверку на двух графах, приведенных на рис. 4.41.

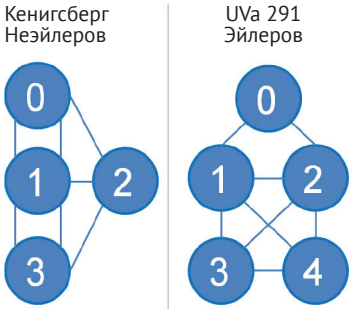


Рис. 4.41 ❖ Эйлеров граф

Вывод эйлерова цикла

В то время как проверить, является ли граф эйлеровым, легко, поиск фактического цикла/пути Эйлера потребует больших трудозатрат. Приведенный ниже код строит путь Эйлера. На вход алгоритма подается невзвешенный эйлеров граф, хранящийся в виде списка смежности, где вторым атрибутом в паре параметров, описывающих ребра, является логическое значение 1 (это ребро еще может использоваться при построении пути Эйлера) или 0 (это ребро больше не может использоваться).

```
list<int> cyc;                          // нам нужен список для быстрой вставки в середине

void EulerTour(list<int>::iterator i, int u) {
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (v.second) {                                 // если это ребро еще можно использовать
      v.second = 0;                               // делаем вес этого ребра равным 0 ('удалено')
      for (int k = 0; k < (int)AdjList[v.first].size(); k++) {
        ii uu = AdjList[v.first][k];                          // удалить двунаправленное ребро
        if (uu.first == u && uu.second) {
          uu.second = 0;
          break;
      } }
      EulerTour(cyc.insert(i, u), v.first);
} } }

// внутри int main()
  cyc.clear();
  EulerTour(cyc.begin(), A);                  // cyc содержит путь Эйлера, начинающийся с A
  for (list<int>::iterator it = cyc.begin(); it != cyc.end(); it++)
    printf("%d\n", *it);                                                    // путь Эйлера
```

4.7.4. Двудольный граф

Двудольный граф является специальным графом со следующими характеристиками: множество вершин V можно разбить на два непересекающихся множества V_1 и V_2, и все ребра в $(u, v) \in E$ имеют следующее свойство: $u \in V_1$ и $v \in V_2$. Вследствие этого свойства двудольный граф не имеет циклов нечетной длины (см. упражнение 4.2.6.3). Обратите внимание, что дерево также является двудольным графом!

Нахождение максимального по мощности паросочетания на двудольном графе (Max Cardinality Bipartite Matching, MCBM) и его решение с помощью вычисления максимального потока (Max Flow)

Задача (из открытого первого отборочного тура Top Coder 2009 года (TopCoder Open 2009 Qualifying 1) [31]): дан список чисел *N*, вернуть список всех элементов в *N*, которые могут быть успешно соединены с *N*[0], как часть *полного простого попарного соединения*, отсортированного в порядке возрастания. Полное простое попарное соединение означает, что каждый элемент *a* в *N* соединен с другим уникальным элементом *b* в *N*, так что *a* + *b* является простым.

Например: пусть дан список чисел *N* = {1, 4, 7, 10, 11, 12}, ответ: {4, 10}. Это связано с тем, что при соединении элемента *N*[0] = 1 с элементом 4 получается простая пара, а остальные четыре элемента также могут образовывать две простые пары (7 + 10 = 17 и 11 + 12 = 23). Аналогичная ситуация, когда элемент *N*[0] = 1 соединяется с элементом 10, то есть 1 + 10 = 11 является простой парой, и у нас также есть две другие простые пары (4 + 7 = 11 и 11 + 12 = 23). Мы не можем соединить *N*[0] = 1 с любым другим элементом в *N*. Например, если мы соединяем *N*[0] = 1 с 12, у нас есть простая пара, но не будет никакого способа соединить четыре оставшихся числа для формирования еще двух простых пар.

Ограничения: список *N* содержит четное количество элементов (`[2..50]`). Каждый элемент *N* будет лежать в интервале `[1..1000]`. Каждый элемент *N* будет уникальным числом в списке.

Хотя эта задача связана с простыми числами, она не является чисто математической задачей, так как элементы *N* займут не более 1 КБ – не так уж много простых чисел меньше 1000 (только 168 простых). Проблема в том, что мы не можем попарно соединить числа, используя полный перебор, так как для первой пары существует ${}^{50}C_2$ возможных комбинаций, для второй пары ${}^{48}C_2$... и т. д. до ${}^{2}C_2$ для последней пары. DP с использованием битовой маски (раздел 8.3.1) также не подходит, потому что число 2^{50} слишком велико.

Ключом к решению данной задачи является понимание того, что это соединение (сопоставление) выполняется на двудольном графе. Чтобы получить простое число, нам нужно сложить 1 нечетное + 1 четное, потому что 1 нечетное + 1 нечетное (или 1 четное + 1 четное) дает четное число (которое не является простым). Таким образом, мы можем разбить нечетные/четные числа на `set1`/`set2` и добавить ребро `i` → `j`, если `set1[i] + set2[j]` – простое число.

После того как мы построим этот двудольный граф, решение будет тривиальным: если размеры `set1` и `set2` различны, полное попарное сопоставление невозможно. В противном случае, если размер обоих множеств равен *n*/2, попробуйте сопоставить `set1[0]` с `set2[k]` для *k* = `[0..n / 2-1]` и найдите максимальное по мощности паросочетание на двудольном графе (MCBM) для остальных элементов (MCBM является одним из наиболее распространенных типов прикладных задач на двудольных графах). Если вы получите больше *n* / 2 – 1 соответствий, добавьте `set2[k]` во множество, содержащее ответ к задаче. Для приведенного контрольного примера ответом является {4, 10} (см. рис. 4.42, посередине).

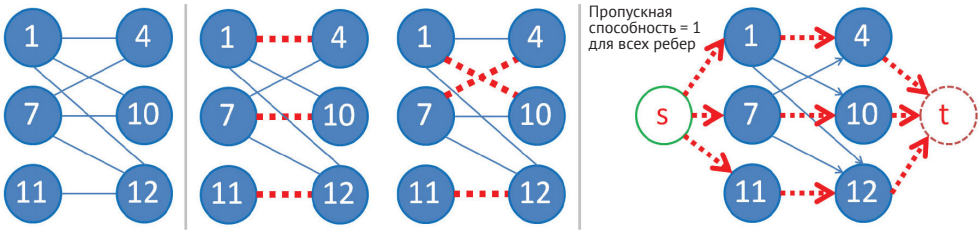


Рис. 4.42 ❖ Задача нахождения паросочетания на двудольном графе может быть сведена к задаче о максимальном потоке

Задача MCBM может быть сведена к задаче о максимальном потоке путем назначения фиктивной исходной вершины (источником) *s*, связанной со всеми вершинами в `set1`, и все вершины в `set2` соединяются с фиктивной конечной вершиной (стоком) *t*. Направление ребер ($s \to u$, $u \to v$, $v \to t$, где $u \in$ `set1` и $v \in$ `set2`). Устанавливая пропускные способности всех ребер в этом графе равными 1, мы заставляем каждую вершину во множестве 1 соответствовать не более чем одной вершине во множестве 2. Максимальный поток будет равен максимальному количеству соединений на исходном графе (см. рис. 4.42, справа, например).

Максимальное независимое множество и минимальное вершинное покрытие на двудольном графе (MCBM)

Независимое множество (Independent Set, IS) графа *G* является подмножеством вершин, таким, что никакие две вершины в подмножестве не образуют ребро *G*. Максимальное IS (MIS) есть IS, максимального размера. В двудольном графе размер MIS + MCBM = *V*. Иными словами: MIS = *V* – MCBM. На рис. 4.43B у нас имеется двудольный граф с двумя вершинами слева и тремя вершинами справа. MCBM равно 2 (на рисунке показано двумя пунктирными линиями), а MIS равно 5 – 2 = 3. Действительно, {3, 4, 5} являются элементами MIS этого двудольного графа.

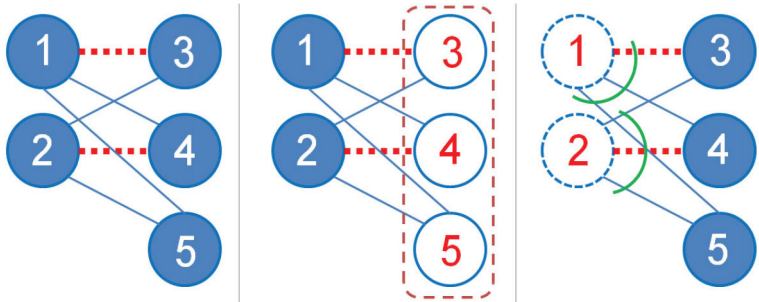


Рис. 4.43 ❖ Варианты MCBM

Вершинное покрытие графа G – это множество вершин C, такое, что каждое ребро графа G инцидентно хотя бы одной вершине во множестве вершин C. В двудольном графе число ребер в MCBM равно количеству вершин в минимальном покрытии (Min Vertex Cover, MVC) – это теорема венгерского математика Денеша Кенига. На рис. 4.43C у нас имеется тот же двудольный граф, что и ранее, для которого MCBM = 2. MVC также равен 2. Действительно, {1, 2} являются элементами MVC этого двудольного графа.

Отметим, что хотя сами значения MCBM/MIS/MVC являются уникальными, решения соответствующих задач могут быть неуникальными. Пример: на рис. 4.43A мы также можем связать {1, 4} и {2, 5}, при этом для обоих вариантов максимальное количество элементов будет одинаковым, равным 2.

Пример применения: UVa 12083 – Guardian of Decency

Сокращенная формулировка условия задачи: имеется $N \leqslant 500$ учеников (доступны данные об их росте, поле, предпочитаемом музыкальном стиле и любимом виде спорта). Определите, сколько учеников могут поехать на экскурсии, если учитель хочет, чтобы любая пара из двух учеников удовлетворяла хотя бы одному из этих четырех критериев, чтобы ни одна пара учеников не стала влюбленной парочкой: 1) их рост отличается более чем на 40 см; 2) они одного пола; 3) их предпочитаемые музыкальные стили различаются; 4) у них одинаковый любимый вид спорта (они, вероятно, болеют за разные команды, и это приведет к ссоре).

Во-первых, обратите внимание, что задача заключается в поиске максимального независимого множества, то есть у выбранных учеников не должно быть никаких шансов стать парой. Поиск независимых множеств – сложная задача для графа общего вида, поэтому давайте проверим, является ли наш граф специальным графом. Далее, обратите внимание, что из условий задачи следует, что у нас есть простой двудольный граф: пол студентов (ограничение номер два). Мы можем расположить учеников с левой стороны, а учениц – с правой. На этом этапе мы должны спросить: какими должны быть ребра этого двудольного графа? Чтобы найти ответ, обратимся к задаче о независимом множестве: мы проводим ребро между учеником i и ученицей j, если существует вероятность, что (i, j) могут стать влюбленной парочкой.

В контексте данной задачи: если у i и j разный пол, $и$ их рост отличается НЕ более чем на 40 см, $и$ предпочитаемый ими стиль музыки – ОДИН И ТОТ ЖЕ, $и$ их любимые виды спорта – РАЗНЫЕ, то эти двое, ученик i и ученица j, имеют высокую вероятность стать влюбленной парой. Учитель может выбрать только одного из них для поездки на экскурсию.

Теперь, когда у нас есть этот двудольный граф, мы можем запустить алгоритм MCBM и сообщить решение: N – MCBM. В этом примере мы еще раз подчеркнули важность наличия хороших навыков моделирования графов! Нет смысла знать алгоритм MCBM и уметь его реализовать в виде кода, если участник олимпиады по программированию не может найти двудольный граф на основании формулировки задачи.

Алгоритм удлиняющей цепи для нахождения максимального по мощности паросочетания

Есть лучший способ решить задачу нахождения максимального по мощности паросочетания (МСВМ) в олимпиадном программировании (с точки зрения времени реализации), чем идти путем решения задачи о максимальном потоке. Мы можем использовать специальный, простой в реализации алгоритм удлиняющей цепи с временной сложностью $O(VE)$. Реализовав его, мы сможем легко решить все задачи МСВМ, включая другие задачи, связанные с графами, использующие МСВМ, такие как нахождение максимального независимого множества на двудольном графе, минимальное вершинное покрытие на двудольном графе и минимальное покрытие путей (Min Path Cover) на DAG (см. раздел 9.24).

Удлиняющая цепь – это путь, который начинается со *свободной* (*не имеющей сочетания*) вершины в левом подграфе двудольного графа, затем продолжается на свободном ребре (теперь в правом подграфе), далее на ребре, имеющем сочетание (теперь снова в левом подграфе)... – на свободном ребре (теперь в правом подграфе) до тех пор, пока, наконец, не достигнет свободной вершины в правом подграфе двудольного графа. Лемма Клода Бержа, сформулированная в 1957 году, гласит, что число сочетаний M на графе G является максимальным (имеет максимально возможное число ребер) тогда и только тогда, когда на G больше не существует удлиняющей цепи. Этот алгоритм удлиняющей цепи является прямой реализацией леммы Бержа: найти, а затем устранить удлиняющие цепи.

Теперь давайте взглянем на простой двудольный граф на рис. 4.44 с вершинами n и m в левом и правом подграфах соответственно. Вершины в левом подграфе пронумерованы как `[1..n]`, а вершины в правом подграфе пронумерованы как `[n + 1..n + m]`. Этот алгоритм пытается найти, а затем устраняет удлиняющие цепи, начиная со свободных вершин в левом подграфе.

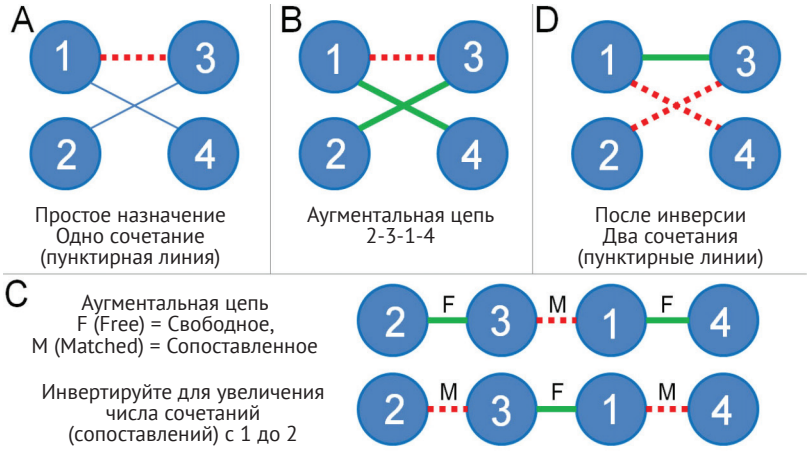


Рис. 4.44 ❖ Алгоритм аугментальной цепи

Мы начинаем со свободной вершины 1. На рис. 4.44A мы видим, что этот алгоритм «неправильно»[1] соединит вершину 1 с вершиной 3 (а не вершину 1 с вершиной 4), поскольку путь 1–3 уже является простой удлиняющей цепью. Как вершина 1, так и вершина 3 являются свободными вершинами. Соединяя вершину 1 и вершину 3, мы получаем наше первое соединение. Обратите внимание, что после того, как мы соединяем вершины 1 и 3, мы не можем найти другое соединение.

На следующей итерации (когда мы находимся в свободной вершине 2) этот алгоритм теперь демонстрирует свою полную силу, находя следующую удлиняющую цепь, которая начинается со свободной вершины 2 слева, идет к вершине 3 через свободное ребро (2–3), переходит к вершине 1 через имеющее сочетание ребро (3–1) и, наконец, снова переходит к вершине 4 через свободное ребро (1–4). И вершина 2, и вершина 4 являются свободными вершинами. Следовательно, удлиняющая цепь будет 2–3–1–4, как показано на рис. 4.44B и 4.44C.

Если мы инвертируем граничный статус для этой удлиняющей цепи, то есть от «свободного к сопоставленному» и от «сопоставленного к свободному», мы получим *еще одно сочетание*. См. рис. 4.44C, где мы инвертируем состояние ребер вдоль удлиняющей цепи 2–3–1–4. Обновленное сопоставление показано на рис. 4.44D.

Этот алгоритм будет продолжать выполнять данный процесс поиска удлиняющих цепей и устранения их, пока больше не найдется ни одной удлиняющей цепи. Поскольку алгоритм повторяет DFS-подобный код[2] с временной сложностью $O(E)$ V раз, он выполняется за время, эквивалентное $O(VE)$. Код, реализующий этот алгоритм, приведен ниже. Заметим, что это не лучший алгоритм поиска МСВМ. Позднее, в разделе 9.12, мы рассмотрим алгоритм Хопкрофта–Карпа, который может решить задачу поиска МСВМ за время, эквивалентное $O(\sqrt{V}E)$ [28].

Упражнение 4.7.4.1*. На рис. 4.42 (справа) проиллюстрирован найденный нами способ свести задачу МСВМ к задаче о максимальном потоке. Вопрос: должны ли ребра в потоковом графе быть направленными? Возможно ли использовать ненаправленные ребра в потоковом графе?

Упражнение 4.7.4.2*. Перечислите общие ключевые слова, которые можно использовать, чтобы помочь участникам на основании постановки задачи сделать вывод о том, что в решении будет использоваться двудольный граф. Например: четный – нечетный, мужской – женский и т. д.

Упражнение 4.7.4.3*. Предложите простое усовершенствование алгоритма поиска удлиняющих цепей, который может улучшить его производительность (временная сложность этого алгоритма в наихудшем случае составляет $O(VE)$) на (почти) полном двудольном графе.

---

[1] Мы предполагаем, что соседи вершины упорядочены по возрастанию номера вершины, то есть, начав из вершины 1, мы сначала посетим вершину 3, а затем вершину 4.

[2] Для упрощения анализа предположим, что $E > V$ в таких двудольных графах.

```
vi match, vis;                                                  // глобальные переменные

int Aug(int l) {                                  // возвращаем 1, если найдена  удлиняющая цепь
  if (vis[l]) return 0;                                        // в противном случае возвращаем 0
  vis[l] = 1;
  for (int j = 0; j < (int)AdjList[l].size(); j++) {
    int r = AdjList[l][j];                        // вес ребра не нужен -> vector<vi> AdjList
    if (match[r] == -1 || Aug(match[r])) {
      match[r] = l; return 1;                                   // найдено 1 сочетание
  } }
  return 0;                                                     // сочетаний нет
}
// внутри int main()
  // строим невзвешенный двудольный граф со множеством направленных ребер слева -> направо
  int MCBM = 0;
  match.assign(V, -1);                                // V - число вершин в двудольном графе
  for (int l = 0; l < n; l++) {                                 // n = размер левого подграфа
    vis.assign(n, 0);                                // устанавливаем в 0 перед каждой рекурсией
    MCBM += Aug(l);
  }
  printf("Найдено %d сочетаний\n", MCBM);
```

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/matching.html

Файл исходного кода: *ch4_09_mcbm.cpp/java*

Замечания о специальных графах на олимпиадах по программированию

Из четырех специальных графов, упомянутых в разделе 4.7, на олимпиадах по программированию чаще всего встречаются DAG и деревья, в особенности на олимпиадах IOI. *Нередко* динамическое программирование (DP) на DAG или на дереве встречается в условиях задач IOI. Поскольку эти варианты DP (как правило) имеют эффективные решения, для них характерен большой размер входных данных. Следующим по популярности специальным графом является двудольный граф. Этот специальный граф подходит для задач о сетевом потоке и нахождения паросочетаний на двудольном графе. Мы считаем, что участники должны освоить использование более простого алгоритма удлиняющей цепи для решения задачи нахождения максимального по мощности паросочетания (Max Cardinality Bipartite Matching, MCBM). В этом разделе мы видели, что многие задачи, связанные с графами, так или иначе сводятся к MCBM. Участники ICPC должны быть знакомы с двудольным графом, построенным поверх DAG и дерева. Участникам IOI не нужно беспокоиться о задачах на двудольные графы, поскольку они все еще не включены в программу IOI 2009 [20]. Другой специальный граф, обсуждаемый в этой главе, – эйлеров граф – в последнее время не так уж часто попадается участникам олимпиад по программированию. Есть и другие специальные графы, которые могут попасться вам

на олимпиаде, однако мы редко встречаемся с ними. Например, планарный граф; полный граф K_n; лес путей; звездный граф и т. д. Когда они попадаются вам, попробуйте использовать их специальные свойства, чтобы ускорить работу алгоритмов.

Известные авторы алгоритмов

Денеш Кениг (1884–1944) был венгерским математиком, специализировавшимся на теории графов и написавшим первый учебник по этой дисциплине. В 1931 году Кениг постулировал эквивалентность между задачей максимального паросочетания и задачей о минимальном вершинном покрытии в контексте двудольных графов, то есть он доказывает, что MCBM = MVC в двудольном графе.

Клод Берж (1926–2002) был французским математиком, признанным одним из современных основателей комбинаторики и теории графов. Его основной вклад, который включен в эту книгу, – это лемма Бержа, которая гласит, что число сочетаний *M* в графе *G* является максимальным тогда и только тогда, когда в *G* более нет удлиняющей цепи относительно *M*.

Задачи по программированию, связанные со специальными графами

- Нахождение кратчайших/наидлиннейших путей из заданной вершины во все остальные (Single – Source Shortest/Longest Paths) на DAG
  1. UVa 00103 – Stacking Boxes (самые длинные пути на DAG; возвратная рекурсия подходит как метод решения)
  2. **UVa 00452 – Project Scheduling *** (PERT; самые длинные пути на DAG; DP)
  3. UVa 10000 – Longest Paths (наидлиннейшие пути на DAG; возвратная рекурсия подходит как метод решения)
  4. UVa 10051 – Tower of Cubes (наидлиннейшие пути на DAG; DP)
  5. UVa 10259 – Hippity Hopscotch (наидлиннейшие пути на неявном DAG; DP)
  6. **UVa 10285 – Longest Run… *** (наидлиннейшие пути на неявном DAG; однако граф достаточно мал для использования возвратной рекурсии)
  7. **UVa 10350 – Liftless Eme *** (наикратчайшие пути; неявный DAG; DP) Также см. «Наибольшая возрастающая подпоследовательность» (раздел 3.5.3)
- Подсчет путей на DAG
  1. UVa 00825 – Walking on the Safe Side (подсчет путей в неявном DAG; DP)
  2. UVa 00926 – Walking Around Wisely (задача похожа на UVa 825)
  3. UVa 00986 – How Many? (подсчет путей в DAG; DP; s: x, y, lastmove, peaksfound; t: try NE/SE)
  4. **UVa 00988 – Many paths, one… *** (подсчет путей в DAG; DP)

5. **UVa 10401 – Injured Queen Problem** * (подсчет путей в неявном DAG; DP; s: col, row; t: next col, избегать двух или трех смежных горизонталей (строк))
6. UVa 10926 – How Many Dependencies? (подсчет путей в DAG; DP)
7. UVa 11067 – Little Red Riding Hood (задача похожа на UVa 825)
8. *UVa 11655 – Waterland* (подсчет путей на DAG и еще одна похожая задача: подсчет числа вершин, находящихся на этих путях)
9. **UVa 11957 – Checkers** * (подсчет путей на DAG; DP)

- Преобразование графа общего вида в DAG

1. UVa 00590 – Always on the Run (s: pos, day_left)
2. **UVa 00907 – Winterim Backpack…** * (s: pos, night_left)
3. UVa 00910 – TV Game (s: pos, move_left)
4. UVa 10201 – Adventures in Moving… (s: pos, fuel_left)
5. UVa 10543 – Traveling Politician (s: pos, given_speech)
6. UVa 10681 – Teobaldo's Trip (s: pos, day_left)
7. UVa 10702 – Traveling Salesman (s: pos, T_left)
8. UVa 10874 – Segments (s: ряд, налево/направо; t: идти налево/направо)
9. **UVa 10913 – Walking…** * (s: r, c, neg_left, stat; t: вниз / (влево / вправо))
10. UVa 11307 – Alternative Arborescence (минимальное количество цветов, максимум шесть цветов)
11. **UVa 11487 – Gathering Food** * (s: row, col, cur_food, len; t: 4 dirs)
12. UVa 11545 – Avoiding… (s: cPos, cTime, cWTime; t: двигаться вперед / отдыхать)
13. UVa 11782 – Optimal Cut (s: id, rem K; t: выбрать вершину / попробовать левое – правое поддерево)
14. SPOJ 0101 – Fishmonger (обсуждается в этом разделе)

- Дерево

1. UVa 00112 – Tree Summing (возвратная рекурсия)
2. UVa 00115 – Climbing Trees (обход дерева, наименьший общий предок)
3. UVa 00122 – Trees on the level (обход дерева)
4. UVa 00536 – Tree Recovery (восстановление дерева с помощью обхода в прямом порядке + симметричного обхода)
5. *UVa 00548 – Tree* (восстановление дерева с помощью симметричного обхода + обхода в обратном порядке)
6. UVa 00615 – Is It A Tree? (проверка свойства графа)
7. UVa 00699 – The Falling Leaves (обход в прямом порядке)
8. UVa 00712 – S-Trees (простой вариант обхода бинарного дерева)
9. UVa 00839 – Not so Mobile (можно рассматривать как задачу о рекурсии на дереве)
10. UVa 10308 – Roads in the North (диаметр дерева, обсуждается в этом разделе)
11. ***UVa 10459 – The Tree Root*** * (укажите диаметр этого дерева)
12. UVa 10701 – Pre, in и post (восстановление дерева с помощью обхода в прямом порядке + симметричного обхода)

13. ***UVa 10805 – Cockroach Escape…*** * (включает диаметр)
14. *UVa 11131 – Close Relatives* (прочитайте дерево; выполните два обхода в обратном порядке)
15. *UVa 11234 – Expressions* (преобразование обхода в обратном порядке в обход по уровням, двоичное дерево)
16. UVa 11615 – Family Tree (подсчет размера поддеревьев)
17. **UVa 11695 – Flight Planning** * (обрежьте наихудший край по диаметру дерева, свяжите два центра)
18. *UVa 12186 – Another Crisis* (входной граф – дерево)
19. *UVa 12347 – Binary Search Tree* (дан обход BST в прямом порядке, используйте свойство BST, чтобы получить BST, выведите результат обхода BST в обратном порядке)

Эйлеров граф

1. UVa 00117 – The Postal Worker… (путь Эйлера, стоимость пути)
2. UVa 00291 – The House of Santa… (путь Эйлера, граф небольшого размера, возвратная рекурсия)
3. **UVa 10054 – The Necklace** * (необходимо вывести путь Эйлера)
4. UVa 10129 – Play on Words (проверка, является ли граф графом Эйлера)
5. **UVa 10203 – Snow Clearing** * (основной граф – граф Эйлера)
6. **UVa 10596 – Morning Walk** * (проверка свойства графа Эйлера)

- Двудольный граф

1. *UVa 00663 – Sorting Slides* (попробуйте отклонить ребро, чтобы увидеть, изменится ли MCBM; это означает, что ребро должно использоваться)
2. UVa 00670 – The Dog Task (MCBM)
3. UVa 00753 – A Plug for Unix (изначально нестандартная задача сопоставления, но эта задача может быть сведена к простой задаче MCBM)
4. UVa 01194 – Machine Schedule (LA 2523, Пекин'02, минимальное вершинное покрытие / MVC)
5. UVa 10080 – Gopher II (MCBM)
6. **UVa 10349 – Antenna Placement** * (максимальное независимое множество: V – MCBM)
7. **UVa 11138 – Nuts and Bolts** * (задача на «чистое» MCBM; если у вас мало опыта в решении задач MCBM, лучше начать с этой задачи)
8. **UVa 11159 – Factors and Multiples** * (MIS, но это MCBM)
9. *UVa 11419 – SAM I AM* (MVC, теорема Кенига)
10. UVa 12083 – Guardian of Decency (LA 3415, Северо-Западная Европа'05, MIS)
11. UVa 12168 – Cat vs. Dog (LA 4288, Северо-Западная Европа'08, MIS)
12. Открытые отборочные соревнования турнира Top Coder 2009 (Top Coder Open 2009): пары простых чисел (задача обсуждается в этом разделе)

4.8. Решения упражнений, не помеченных звездочкой

Упражнение 4.2.2.1. Просто замените `dfs(0)` на `bfs` из источника `s` = 0.

Упражнение 4.2.2.2. При хранении графа в виде матрицы смежности, списка смежности и списка ребер потребуется $O(V)$, $O(k)$ и $O(E)$ времени для перечисления списка соседей вершины соответственно (примечание: k – это число фактических соседей вершины). Поскольку DFS и BFS просматривают все ребра, исходящие из каждой вершины, время исполнения кода зависит от скорости структуры данных графа при перечислении соседних элементов графа. Следовательно, временная сложность алгоритмов DFS и BFS составляет $O(V \times V = V^2)$, $O(\max(V, V\sum_{i=0}^{V-1} k_i) = V + E)$ и $O(V \times E = VE)$ для обхода графа, хранящегося в виде матрицы смежности, списка смежности и списка ребер соответственно. Поскольку список смежности является наиболее эффективной структурой данных для обхода графа, может быть полезно сначала преобразовать матрицу смежности или список ребер в список смежности (см. **упражнение 2.4.1.2***), прежде чем обходить граф.

Упражнение 4.2.3.1. Начните с вершин, не имеющих общих элементов. Для каждого `edge(u, v)` выполните операцию `unionSet(u, v)`. Состояние непересекающихся множеств после обработки всех ребер представляет компоненты связности. Решение BFS «тривиально»: просто замените `dfs(i)` на `bfs(i)`. Оба варианта отрабатывают за время $O(V + E)$.

Упражнение 4.2.5.1. Это своего рода «обход в прямом порядке» в терминологии обхода двоичного дерева. Функция `dfs2` посещает все дочерние элементы u перед добавлением вершины u в конце вектора `ts`. Это удовлетворяет свойству топологической сортировки.

Упражнение 4.2.5.2. Ответ – использовать связный список. Однако, поскольку в главе 2 мы сказали, что хотим избежать использования связного списка, мы решили использовать здесь `vi ts`.

Упражнение 4.2.5.3. Алгоритм по-прежнему завершается, но выведенный результат теперь не релевантен, так как не DAG не имеет топологической сортировки.

Упражнение 4.2.5.4. Для этого мы должны использовать возвратную рекурсию.

Упражнение 4.2.6.3. Доказательство от противного. Предположим, что двудольный граф имеет нечетный (нечетной длины) цикл. Пусть нечетный цикл содержит $2k + 1$ вершин для некоторого целого числа k, которое формирует этот путь: $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{2k-1} \rightarrow v_{2k} \rightarrow v_0$. Теперь мы можем поместить v_0 в левом множестве, v_1 в правом множестве, ..., v_{2k} снова в левом множестве, но тогда у нас есть ребро (v_{2k}, v_0), которого нет в левом множестве. Следовательно, это не цикл → противоречие. Таким образом, двудольный граф не имеет нечетного цикла. Это свойство может быть важно для решения некоторых задач, связанных с двудольным графом.

Упражнение 4.2.7.1. Два задних ребра: 2 → 1 и 6 → 4.

Упражнение 4.2.8.1. Точки сочленения: 1, 3 и 6; мосты: 0–1, 3–4, 6–7 и 6–8.

Упражнение 4.2.9.1. Доказательство от противного. Предположим, что существует путь из вершины *u* в *w* и из *w* к *v*, где *w* находится вне компонент сильной связности (SCC). Отсюда можно сделать вывод, что мы можем добраться из вершины *w* в любые вершины SCC и из любых вершин SCC в *w*. Следовательно, вершина *w* также должна быть в SCC. Противоречие. Таким образом, не существует пути между двумя вершинами в SCC, который когда-либо выходит за пределы SCC.

Упражнение 4.3.2.1. Мы можем остановиться, когда число непересекающихся множеств уже равно одному. Простая модификация: измените начало цикла MST, заменив строку `for (int i = 0; i < E; i++) {` следующей строкой: `for (int i = 0; i < E && disjointSetSize > 1; i++) {`.

В качестве альтернативы мы подсчитываем количество ребер, выбранных до сих пор. Как только оно достигнет $V - 1$, мы можем остановиться.

Упражнение 4.3.4.1. Мы обнаружили, что задачи о минимальном остовном дереве и втором лучшем остовном дереве сложнее решить с помощью алгоритма Прима.

Упражнение 4.4.2.1. Для этого варианта решение является простым. Просто поставьте в очередь все источники и установите значение `dist[s]` = 0 для всех источников перед запуском цикла BFS. Поскольку это всего лишь один вызов BFS, он выполняется в $O(V + E)$.

Упражнение 4.4.2.2. В начале цикла while, когда мы выделяем самую первую вершину из очереди, мы проверяем, является ли эта вершина конечной, т. е. «местом назначения». Если это так, мы прерываем на этом цикл. Наихудшая временная сложность в этом случае – все еще $O(V + E)$, но наш поиск BFS остановится раньше, если конечная вершина находится близко к исходной вершине.

Упражнение 4.4.2.3. Вы можете преобразовать этот взвешенный граф с постоянным весом в невзвешенный граф, заменив все веса ребер на единицы. Информация SSSP, полученная BFS, затем умножается на константу C, чтобы получить фактические ответы.

Упражнение 4.4.3.1. На положительно взвешенном графе – да. Каждая вершина будет пройдена и обработана только один раз. Каждый раз, когда обрабатывается вершина, мы пытаемся выполнить операцию relax для ее соседей. Из-за «ленивого удаления» у нас может быть не более $O(E)$ элементов в очереди с приоритетом в определенное время, но это все равно $O(\log E) = O(\log V^2) = O(2 \times \log V) = O(\log V)$ для каждой операции удаления из очереди или постановки в очередь. Таким образом, временная сложность остается на уровне $O((V + E)\log V)$. На графе с (несколькими) ребрами с отрицательным весом, но без цикла с отрицательным весом он работает медленнее из-за необходимости повторной обработки обработанных вершин, но значения для кратчайших путей верны (в отличие от реализации алгоритма Дейкстры, приведенной в [7]). Это показано в примере в разделе 4.4.4. В редких случаях эта реализация

алгоритма Дейкстры может работать очень медленно на некоторых графах, имеющих несколько ребер с отрицательным весом, хотя граф не имеет цикла отрицательного веса (см. **упражнение 4.4.3.2***). Если граф имеет цикл отрицательного веса, этот вариант реализации алгоритма Дейкстры будет иметь бесконечный цикл.

Упражнение 4.4.3.3. Используйте `set<ii>`. В этом наборе хранится отсортированная пара значений, содержащих информацию о вершинах, как показано в разделе 4.4.3. Вершина с минимальным расстоянием является первым элементом в (отсортированном) наборе. Чтобы обновить расстояние до определенной вершины от источника, мы ищем, а затем удаляем старую пару значений. Затем мы добавляем в набор новую пару значений. Поскольку мы обрабатываем каждую вершину и ребро один раз и каждое обращение к `set<ii>` выполняется за $O(\log V)$, общая временная сложность варианта реализации Дейкстры с использованием `set<ii>` по-прежнему составляет $O((V + E)\log V)$.

Упражнение 4.4.3.4. В разделе 2.3 мы показали способ превратить невозрастающую кучу (max heap) (по умолчанию реализованную для `priority_queue` в C++ STL) в неубывающую кучу (min heap) путем умножения ключей на –1.

Упражнение 4.4.3.5. Ответ, аналогичный упражнению 4.4.2.2, если данный взвешенный граф не имеет ребер с отрицательным весом. Существует вероятность неправильного ответа, если данный взвешенный граф имеет ребра с отрицательным весом.

Упражнение 4.4.3.6. Нет, мы не можем использовать DP. Моделирование состояний и переходов, описанное в разделе 4.4.3, создает граф пространства состояний, который не является DAG. Например, мы можем начать с состояния (s, 0), добавить 1 единицу топлива в вершине s, чтобы достичь состояния (s, 1), перейти к соседней вершине y – предположим, что это будет переход всего лишь на 1 единицу расстояния, – чтобы достичь состояния (y, 0), снова добавить 1 единицу топлива в вершине y, чтобы достичь состояния (y, 1), а затем вернуться обратно в состояние (s, 0) (цикл). Таким образом, эта задача является задачей поиска кратчайшего пути на взвешенном графе общего вида. Нам нужно использовать алгоритм Дейкстры.

Упражнение 4.4.4.1. Это потому, что изначально только вершина источника имеет правильную информацию о расстоянии. Затем каждый раз, когда мы выполняем операцию relax на всех E ребрах, мы гарантируем, что по крайней мере еще одна вершина с еще одним переходом (в терминах ребер, используемых в кратчайшем пути от источника) имеет правильную информацию о расстоянии. В упражнении 4.4.1.1 мы увидели, что кратчайший путь должен быть простым путем (имеет не более $E = V - 1$ ребер). Таким образом, после выполнения ($V - 1$) раз алгоритма Форда–Беллмана даже вершина с наибольшим количеством переходов будет иметь правильную информацию о расстоянии.

Упражнение 4.4.4.2. Добавьте логический флаг `modified = false` во внешний цикл (тот, который повторяет операцию relax для всех E ребер $V - 1$ раз). Если по крайней мере одна операция relax выполняется во внутренних циклах (та, которая обходит все E ребер), обновите значение `modified = true`. Немедленно

прервите самый внешний цикл, если флаг `modified` все еще имеет значение `false`, после того как все E ребер были пройдены. Если операция relax не выполнялась на итерации i внешнего цикла, на итерации $i + 1, i + 2, \ldots i = V - 1$ она также не будет выполнена.

Упражнение 4.5.1.1. Это потому, что мы добавим `AdjMat[i][k] + AdjMat[k][j]`, что приведет к переполнению, если оба значения `AdjMat[i][k]` и `AdjMat[k][j]` сопоставимы с MAX_INT, что в результате дает неправильный ответ.

Упражнение 4.5.1.2. Алгоритм Флойда–Уоршелла работает на графе с ребрами отрицательного веса. Для графа, имеющего цикл отрицательного веса, см. раздел 4.5.3 о поиске цикла отрицательного веса.

Упражнение 4.5.3.1. Запуск алгоритма Уоршелла непосредственно на графе с $V \leqslant 1000$ приведет к превышению лимита времени (TLE). Поскольку количество запросов невелико, мы можем позволить себе запускать DFS с временной сложностью $O(V + E)$ для каждого запроса, чтобы проверить, связаны ли вершины u и v каким-либо путем. Если входной граф направленный, мы можем сначала найти SCC для направленного графа за время, эквивалетное $O(V + E)$. Если u и v принадлежат одному и тому же SCC, то из u непременно можно попасть в v. Это можно проверить без дополнительных затрат. Если SCC, содержащий u, имеет направленное ребро к SCC, содержащему v, тогда из u также достигается v. Но проверку соединения между различными SCC реализовать гораздо сложнее, так что мы можем просто использовать обычный DFS, чтобы получить ответ.

Упражнение 4.5.3.3. В алгоритме Флойда–Уоршелла замените сложение умножением и установите главную диагональ равной 1,0. После того как мы запустили алгоритм Флойда–Уоршелла, мы проверяем, есть ли значения главной диагонали > 1,0.

Упражнение 4.6.3.1. A 150; B = 125; C = 60.

Упражнение 4.6.3.2. В обновленном коде, приведенном ниже, мы используем как список смежности (для быстрого перечисления соседей; не забудьте включить обратные ребра из-за обратного направления потока), так и матрицу смежности (для быстрого доступа к остаточной пропускной способности) одного и того же потокового графа, т. е. мы концентрируемся на улучшении следующей строки: `for (int v = 0; v < MAX_V; v++)`. Мы также заменяем `vi dist(MAX_V, INF)` на `bitset<MAX_V> visited`, чтобы немного ускорить код.

```
// внутри int main(), предполагается, что у нас имеется как (AdjMatrix), так и AdjList
  mf = 0;
  while (1) {                          // алгоритм Эдмондса-Карпа теперь действительно имеет
                                                              // временную сложность O(VE^2)
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;                    // мы заменяем vi dist на bitset!
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
```

```
    if (u == t) break;
    for (int j = 0; j < (int)AdjList[u].size(); j++) {     // здесь используется AdjList!
      int v = AdjList[u][j];                          // мы используем vector<vi> AdjList
      if (res[u][v] > 0 && !vis[v])
        vis[v] = true, q.push(v), p[v] = u;
    }
  }
  augment(t, INF);
  if (f == 0) break;
  mf += f;
}
```

Упражнение 4.6.4.1. Мы используем ∞ в качестве значения пропускной способности «средних направленных ребер» между левым и правым множествами двудольного графа для общей корректности моделирования этого потокового графа. Если пропускная способность от правого множества к стоку *t* не равна 1, как в UVa 259, мы получим неправильное значение максимального потока (Max Flow), если установим пропускную способность этих «средних направленных ребер» равной 1.

4.9. Примечания к главе 4

Мы завершаем эту относительно длинную главу тем, что отмечаем, что в данной главе приведено много алгоритмов и упоминается множество изобретателей алгоритмов – больше, чем в остальных главах этой книги. Вероятно, мы продолжим эту традицию в будущем – в книге будет *больше* алгоритмов на графах. Тем не менее мы должны предупредить участников, что на последних ICPC и IOI участников не просто просили решить задачи, связанные с простыми формами этих алгоритмов на графах. Новые задачи обычно требуют от участников творческого подхода при моделировании графа, объединении двух или более алгоритмов или использовании определенных алгоритмов с некоторыми продвинутыми структурами данных, например объединении алгоритма поиска самого длинного пути на DAG со структурой данных Segment Tree (дерево отрезков); использовании SCC для сокращения направленного графа и преобразовании графа в DAG перед тем, как решить актуальную задачу для DAG; и т. д. Эти более сложные разновидности задач, применяющих теорию графов, обсуждаются в разделе 8.4.

В этой главе, хотя она уже довольно длинная, все еще пропущено множество известных алгоритмов на графах и задач из области графов, которые могут встретиться на ICPC, а именно: нахождения *k* кратчайших путей, задача коммивояжера (см. раздел 9.2), алгоритм Чу–Лю/Эдмондса для решения задачи о минимизации издержек, алгоритм МСВМ Хопкрофта–Карпа (см. раздел 9.12), алгоритм «взвешенного» МСВМ Куна–Манкреса (венгерский алгоритм), алгоритм Эдмондса для графа общего вида и т. д. Мы приглашаем читателей просмотреть главу 9, где приведены некоторые из этих алгоритмов.

Если вы хотите увеличить свои шансы на победу в ACM ICPC, потратьте некоторое время на изучение дополнительных алгоритмов / задач на графы, по-

мимо[1] данной книги. Эти сложные задачи редко появляются в региональных олимпиадах, но если они все же включены в программу соревнований, они обычно становятся *решающими*. Сложные задачи на графы чаще появляются на уровне финальных мировых турниров ACM ICPC.

Тем не менее у нас есть хорошие новости для участников IOI. Мы считаем, что большинство материалов, связанных с графами, включенных в программу IOI, уже рассматриваются в этой главе. Вам необходимо освоить базовые алгоритмы, описанные здесь, а затем улучшить свои навыки решения задач, применяя эти базовые алгоритмы к задачам на графы, требующим творческого подхода, часто встречающимся в IOI.

Таблица 4.6. Статистические данные, относящиеся к главе 4

Параметр	Первое издание	Второе издание	Третье издание
Число страниц	35	49 (+40 %)	70 (+43 %)
Письменные упражнения	8	30 (+275 %)	30 + 20* = 50 (+63 %)
Задачи по программированию	173	230 (+33 %)	248 (+8 %)

Распределение количества упражнений по программированию по разделам этой главы показано ниже:

Таблица 4.7. Распределение количества упражнений по программированию по разделам главы 4

Раздел	Название	Число заданий	% в главе	% в книге
4.2	Обход графа	65	26 %	4 %
4.3	Минимальное остовное дерево	25	10 %	1 %
4.4	Нахождение кратчайших путей из заданной вершины во все остальные (Single – Source Shortest Paths, SSSP)	51	21 %	3 %
4.5	Кратчайшие расстояния между всеми вершинами	27	11 %	2 %
4.6	Сетевой поток	13	5 %	1 %
4.7	Специальные графы	67	27 %	4 %

[1] Заинтересованные читатели могут ознакомиться с работой Феликса [23], в которой обсуждается алгоритм нахождения максимального потока для больших графов, состоящих из 411 млн вершин и 31 млрд ребер.

Глава 5

Математика

Мы используем математику в повседневной жизни, чтобы предсказывать погоду, узнавать время, обращаться с деньгами.

Математика – это нечто больше, чем формулы или уравнения. Это логика, это рациональность, она использует наш ум, чтобы постичь величайшие тайны.

– Телесериал NUMB3RS

5.1. Общий обзор и мотивация

Появление математических задач на олимпиадах по программированию неудивительно, поскольку корни информатики глубоко уходят в область математики. Сам термин «компьютер» происходит от слова «вычислять», поскольку компьютер создан в первую очередь для того, чтобы помогать человеку оперировать с числами. Многие интересные проблемы из реальной жизни могут быть смоделированы как математические задачи, и в этой главе вы увидите множество таких примеров.

В последнее время программы соревнований ICPC (особенно в Азии) обычно включают одну или две математические задачи. Задачи, предлагаемые на соревнованиях IOI, обычно не содержат чисто математических задач, однако многие олимпиадные задачи IOI требуют математического понимания. Цель этой главы – подготовить участников к решению многих из этих математических задач.

Мы понимаем, что в разных странах на этапе доуниверситетского образования обучению математике уделяется различная степень внимания. Поэтому некоторые участники олимпиад по программированию хорошо знакомы с математическими терминами, приведенными в табл. 5.1, в то время как для остальных эти математические термины – темный лес. Участники олимпиад могут не понимать эти термины потому, что не изучали их прежде, либо потому, что до сих пор встречали этот термин лишь на своем родном языке. В этой главе мы хотим уравнять возможности читателей с разными уровнями погружения в эту тему и потому приводим множество общих математических терминов, определений, задач и алгоритмов, которые часто встречаются на олимпиадах по программированию.

Таблица 5.1. Перечень некоторых математических терминов, встречающихся в этой главе

Арифметическая прогрессия (Arithmetic Progression)	**Геометрическая прогрессия (Geometric Progression)**	**Полином (Polynomial)**
Алгебра (Algebra)	Логарифм/показатель степени (Logarithm/Power)	BigInteger (длинные числа)
Комбинаторика (Combinatorics)	Числа Фибоначчи (Fibonacci)	Золотое сечение (Golden Ratio)
Формула Бине (Binet's formula)	Теорема Цекендорфа (Zeckendorf's theorem)	Числа Каталана (Catalan Numbers)
Факториал (Factorial)	Беспорядок (перестановка без неподвижных точек) (Derangement)	Биномиальные коэффициенты (Binomial Coefficients)
Теория чисел (Number Theory)	Простое число (Prime Number)	Решето Эратосфена (Sieve of Eratosthenes)
Модифицированное решето Эратосфена (Modified Sieve)	Тест Миллера–Рабина (Miller-Rabin's)	Функция Эйлера («фи» Эйлера) (Euler Phi)
Наибольший общий делитель (Greatest Common Divisor)	Наименьшее общее кратное (Lowest Common Multiple)	Расширенный алгоритм Евклида (Extended Euclid)
Линейное диофантово уравнение (Linear Diophantine Equation)	Поиск цикла (Cycle-Finding)	Теория вероятностей (Probability Theory)
Теория игр (Game Theory)	Игра с нулевой суммой (Zero-Sum Game)	Дерево решений (Decision Tree)
Идеальная игра (Perfect Play)	Минимакс (Minimax)	Игра Ним (Nim Game)

5.2. Задачи Ad Hoc и математика

Мы начнем эту главу с легких тем: задач Ad Hoc про математические объекты. Это задачи, предлагаемые для решения на олимпиадах по программированию, которые требуют не более чем элементарных навыков программирования и некоторой фундаментальной математической подготовки. Поскольку к этой категории относится слишком много задач, мы делим их на подкатегории, как показано далее. Эти задачи не вошли в раздел 1.4, так как они являются задачами Ad Hoc «с математическим уклоном». Вы можете перейти от раздела 1.4 к этому разделу. Но помните, что многие из этих задач легче, чем остальные. Чтобы занять высокие места на олимпиаде по программированию, участники олимпиады также должны освоить материал из других разделов этой главы.

- **Самые простые задачи.** Решение этих задач состоит всего лишь из нескольких строк кода. Они послужат будущим участникам олимпиад для повышения уверенности в своих силах. Это задачи для тех, кто раньше никогда не решал олимпиадных задач.
- **Математическое моделирование (перебор).** Эти задачи можно решить путем математического моделирования. Обычно при их решении требуется использовать циклы в той или иной форме. Пример: пусть имеется множество S, состоящее из $1M$ миллиона случайных целых чи-

сел, и целое число *X*. Сколько целых чисел в *S* меньше *X*? Ответ: задача решается методом перебора: переберите все целые числа из множества *S* и посчитайте, сколько из них меньше *X*. Этот способ будет работать немного быстрее, чем если вы сначала отсортируете целые числа множества *S* с размерностью 1*M*. Если вам необходимо повторить различные (итеративные) методы полного перебора, см. раздел 3.2. Некоторые математические задачи, решаемые с помощью перебора, также приведены в разделе 3.2.

- **Поиск закономерности или формулы.** Для решения этих задач нужно внимательно прочитать условие задачи, чтобы выявить закономерность или вывести упрощенную формулу. Решение таких задач «в лоб» обычно приводит к превышению лимита времени (вердикту тестирующей системы «TLE»). Работающие решения обычно короткие, и в них не требуется использовать циклы или рекурсии. Пример: пусть *S* – бесконечное множество квадратов целых чисел, отсортированных по возрастанию: {1, 4, 9, 16, 25, ...}. Дано целое число *X* ($1 \leqslant X \leqslant 1017$. Определить, сколько целых чисел во множестве *S* меньше, чем *X? Ответ:* $\lfloor\sqrt{(X-1)}\rfloor$.
- **Решетки.** Эти задачи связаны с операциями над решетками. Решетка может быть сложной, но при этом подчиняться некоторым простым правилам. «Тривиальная» одномерная или двумерная решетка не относится к этому случаю. Решение обычно требует творческого подхода к поиску закономерностей для действий на решетке или к приведению заданной решетки к более простому варианту.
- **Системы счисления и последовательности.** В некоторых специальных математических задачах задаются определения известных (или специально придуманных) систем счисления или последовательностей, и наша задача состоит в том, чтобы сгенерировать либо число (или последовательность) в некотором диапазоне, либо *n*-й член последовательности, проверить, является ли данное число (последовательность) корректным с точки зрения определения или заданных условий, и т. д. Обычно ключом к решению подобных задач является точное следование описанию постановки задачи. Однако при решении некоторых более сложных задач требуется сначала упростить формулу. Некоторые известные примеры:
  1) числа Фибоначчи (раздел 5.4.1): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...;
  2) факториал (раздел 5.5.3): 1, 1, 2, 6, 24, 120, 720, 5040, 40 320, 362 880, ...;
  3) беспорядок (перестановка) (раздел 9.8): 1, 0, 1, 2, 9, 44, 265, 1854, 14 833, 133 496, ...;
  4) числа Каталана (раздел 5.4.3): 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...;
  5) арифметическая прогрессия: a_1, $(a_1 + d)$, $(a_1 + 2 \times d)$, $(a_1 + 3 \times d)$, ..., например 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ..., которая начинается с $a_1 = 1$ и имеет разность $d = 1$ между любыми двумя ее последовательными членами. Сумма первых *n* членов этого ряда арифметической прогрессии $S_n = (n/2) \times (2 \times a_1 + (n - 1) \times d)$;
  6) геометрическая прогрессия, например a_1, $a_1 \times r$, $a_1 \times r^2$, $a_1 \times r^3$, ... т. е. 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ..., которая начинается с $a_1 = 1$ и имеет общий знаменатель $r = 2$ между ее последовательными членами.

Сумма первых n членов этого геометрического ряда прогрессии $S_n = a \times (1 - r^n)/(1 - r)$.

- **Логарифмическая функция, экспонента, степенная функция.** Эти задачи связаны с неочевидным использованием функций log() и/или exp(). Несколько значимых примеров показаны в упражнениях ниже.
- **Полиномы.** Эти задачи включают в себя вычисления, связанные с полиномами, операцию взятия производной, умножение, деление и т. д. Мы можем реализовать многочлен, сохранив коэффициенты членов многочлена, отсортированные по их степеням (обычно в порядке убывания). Операции над полиномами обычно требуют особой внимательности при использовании циклов.
- **Представление чисел в разных системах счисления.** Это математические задачи, связанные с разными системами счисления; сюда не включаются стандартные задачи преобразования из одной системы счисления в другую, которые можно легко решить, используя Java BigInteger (см. раздел 5.3).
- **Другие задачи Ad Hoc.** Это другие математические и околоматематические задачи Ad Hoc, которые не относятся ни к одной из вышеперечисленных подкатегорий.

Мы предлагаем читателям – особенно тем, кто не обладает достаточным опытом в решении задач по математике, – начать подготовку с решения этих задач и решить, по крайней мере, две-три задачи из каждой подкатегории, в особенности те, которые мы пометили как **обязательные ***.

Упражнение 5.2.1. Какие возможности C/C ++/Java мы должны использовать для вычисления $\log_b(a)$ (логарифма по основанию b)?

Упражнение 5.2.2. Что возвращает `(int)floor(1 + log10((double)a))`?

Упражнение 5.2.3. Как вычислить $\sqrt[n]{a}$ (корень n-й степени из a) в C/C ++/Java?

Упражнение 5.2.4*. Изучите метод (Руффини–)Горнера для нахождения корней полиномиального уравнения $f(x) = 0$.

Упражнение 5.2.5*. Для $1 < a < 10$, $1 \leqslant n \leqslant 100\,000$, покажите, как вычислить значение $1 \times a + 2 \times a^2 + 3 \times a^3 + \ldots + n \times a^n$ эффективно, т. е. за время $O(\log n)$.

Примеры Ad Hoc задач с математическими объектами

- Простые задачи
  1. UVa 10055 – Hashmat the Brave Warrior (функция вычисления абсолютного значения; используйте long long)
  2. UVa 10071 – Back to High School… (очень просто: вывод $2 \times v \times t$)
  3. UVa 10281 – Average Speed (расстояние = скорость × прошедшее время)
  4. UVa 10469 – To Carry or not to Carry (очень просто, если вы используете XOR)
  5. **UVa 10773 – Back to Intermediate… *** (несколько хитрых случаев)

6. UVa 11614 – Etruscan Warriors Never... (найдите корни квадратного уравнения)
7. **UVa 11723 – Numbering Roads *** (простая математика)
8. UVa 11805 – Bafana Bafana (существует очень простая формула за $O(1)$)
9. **UVa 11875 – Brick Game *** (получить медиану отсортированного ввода)
10. *UVa 12149 – Feynman* (поиск закономерности; полные квадраты)
11. *UVa 12502 – Three Families* (сначала нужно понять хитрость с формулировкой)

- Математическое моделирование (перебор), более простые
  1. UVa 00100 – The 3n + 1 problem (выполните то, что требуется в условиях задачи; обратите внимание, что j может быть < i)
  2. UVa 00371 – Ackermann Functions (аналогично UVa 100)
  3. **UVa 00382 – Perfection *** (выполните пробное деление)
  4. UVa 00834 – Continued Fractions (выполните то, что требуется в условиях задачи)
  5. UVa 00906 – Rational Neighbor (вычисляйте c, начиная с $d = 1$ до $a/b < c/d$)
  6. **UVa 01225 – Digit Counting *** (N мало)
  7. UVa 10035 – Primary Arithmetic (подсчитать количество операций переноса)
  8. **UVa 10346 – Peter's Smoke *** (интересная задача на моделирование)
  9. UVa 10370 – Above Average (вычислите среднее, посмотрите, сколько значений находится выше среднего)
  10. UVa 10783 – Odd Sum (размер входных данных очень мал, задача решается простым перебором)
  11. UVa 10879 – Code Refactoring (просто используйте перебор)
  12. UVa 11150 – Cola (аналогично UVa 10346, будьте осторожны с граничными случаями!)
  13. UVa 11247 – Income Tax Hazard (используйте перебор, чтобы получить правильный ответ; в данном случае это самый надежный метод)
  14. UVa 11313 – Gourmet Games (задача решается по аналогии с UVa 10346)
  15. UVa 11689 – Soda Surpler (задача решается по аналогии с UVa 10346)
  16. UVa 11877 – The Coco-Cola Store (задача решается по аналогии с UVa 10346)
  17. UVa 11934 – Magic Formula (просто используйте перебор)
  18. *UVa 12290 – Counting Game* (ответ –1 невозможен)
  19. *UVa 12527 – Different Digits* (переберите все варианты, проверьте, не повторяются ли цифры)

- Математическое моделирование (перебор), более сложные
  1. *UVa 00493 – Rational Spiral* (моделирование спирального процесса)
  2. *UVa 00550 – Multiplying by Rotation* (перестановка последней цифры на первую позицию; попробуйте применять описанный метод ко всем цифрам по очереди, начиная с 1-й цифры)

3. **UVa 00616 – Coconuts, Revisited** * (используйте метод перебора до $\sqrt{n}$, выявите закономерность)
4. UVa 00697 – Jack and Jill (требуется определенное форматирование выходных данных и базовые знания физики)
5. UVa 00846 – Steps (используйте формулу суммы арифметической прогрессии)
6. *UVa 10025 – The? 1? 2?* … (сначала упростите формулу, используйте итеративный подход)
7. *UVa 10257 – Dick and Jane* (мы можем использовать перебор для нахождения целочисленных значений возраста кошки, собаки и черепахи; нужно применить опыт в решении подобных математических задач)
8. *UVa 10624 – Super Number* (возвратная рекурсия с проверкой делимости)
9. **UVa 11130 – Billiard bounces** * (используйте способ отражения бильярдного стола: отразите бильярдный стол слева направо (и/или сверху вниз), чтобы рассматривать только одну прямую линию вместо ломаных линий движения меняющего направление бильярдного шара)
10. **UVa 11254 – Consecutive Integers** * (используйте сумму арифметической прогрессии: $n = r/2 \times (2 \times a + r - 1)$ или $a = (2 \times n + r - r^2)/(2 \times r)$; если дано n, используйте перебор для всех значений r от $\sqrt{2n}$ до 1, остановите алгоритм при первом полученном действительном a)
11. UVa 11968 – In The Airport (вычисление среднего; напитки и пирожные; если сочетаются, выберите ближайшее меньшее значение)

Также см. некоторые математические задачи в разделе 3.2.

- Поиск закономерности или вывод формулы, более простые
  1. UVa 10014 – Simple calculations (выведите необходимую формулу)
  2. UVa 10170 – The Hotel with Infinite… (существует формула, которая занимает одну строчку)
  3. UVa 10499 – The Land of Justice (существует простая формула)
  4. UVa 10696 – f91 (элементарное упрощение формул)
  5. ***UVa 10751 – Chessboard*** * (оценка для $N = 1$ и $N = 2$; сначала выведите формулу для $N > 2$; подсказка: используйте диагональ везде, где только можно)
  6. **UVa 10940 – Throwing Cards Away II** * (найдите закономерность, используя перебор)
  7. UVa 11202 – The least possible effort (используйте симметрию и отражение)
  8. ***UVa 12004 – Bubble Sort*** * (возьмите малое значение n; найдите шаблон; используйте long long)
  9. *UVa 12027 – Very Big Perfect Squares* (прием с использованием sqrt)
- Поиск закономерности или вывод формулы, более сложные
  1. *UVa 00651 – Deck* (используйте приведенный пример входных/выходных данных для вывода простой формулы)

2. UVa 00913 – Joana and The Odd… (выведите краткие формулы)
3. **UVa 10161 – Ant on a Chessboard** * (использование функций sqrt, ceil…)
4. *UVa 10493 – Cats, with or without Hats* (дерево, выведите формулу)
5. UVa 10509 – R U Kidding Mr. … (есть только три разных случая)
6. UVa 10666 – The Eurocup is here (проанализируйте двоичное представление X)
7. UVa 10693 – Traffic Volume (вывод краткой формулы физики)
8. *UVa 10710 – Chinese Shuffle* (непростой вывод формулы; используйте modPow; см. раздел 5.3 или 9.21)
9. *UVa 10882 – Koerner's Pub* (принцип включения-исключения)
10. UVa 10970 – Big Chocolate (существует прямая формула, или используйте DP)
11. UVa 10994 – Simple Addition (упрощение формулы)
12. **UVa 11231 – Black and White Painting** * (существует формула $O(1)$)
13. *UVa 11246 – K-Multiple Free Set* (выведите формулу)
14. UVa 11296 – Counting Solutions to an… (существует простая формула)
15. *UVa 11298 – Dissecting a Hexagon* (простая математика; сначала найдите закономерность)
16. *UVa 11387 – The 3-Regular Graph* (невозможно для нечетного n или когда $n = 2$; если n кратно 4, рассмотрим полный граф K_4; если $n = 6 + k \times 4$, рассмотрим один регулярный компонент 3-й степени, состоящий из шести вершин, остальные будут составлять граф K_4, задача сведена к предыдущей)
17. *UVa 11393 – Tri-Isomorphism* (нарисуйте несколько графов K_n небольшого размера, получите шаблон)
18. **UVa 11718 – Fantasy of a Summation** * (преобразование циклов в замкнутую формулу, используйте `modPow` для вычисления результатов, см. разделы 5.3 и 9.21)

- Решетки и координаты
  1. **UVa 00264 – Count on Cantor** * (математика, решетки, правило последовательности)
  2. UVa 00808 – Bee Breeding (математика, решетки, задача аналогична UVa 10182)
  3. UVa 00880 – Cantor Fractions (математика, решетки, задача аналогична UVa 264)
  4. **UVa 10182 – Bee Maja** * (математика, решетки)
  5. **UVa 10233 – Dermuba Triangle** * (число элементов в одном ряду формирует арифметическую прогрессию; используйте гипотезу)
  6. UVa 10620 – A Flea on a Chessboard (просто смоделируйте прыжки)
  7. UVa 10642 – Can You Solve It? (задача, обратная UVA 264)
  8. *UVa 10964 – Strange Planet* (преобразовать координаты в (x, y), тогда эта задача – как раз о нахождении евклидова расстояния между двумя координатами)
  9. *SPOJ 3944 – Bee Walk* (задача, решаемая с использованием координатной сетки)

- Системы счисления или последовательности
  1. UVa 00136 – Ugly Numbers (используйте метод, аналогичный UVa 443)
  2. UVa 00138 – Street Numbers (формула арифметической прогрессии, предварительные вычисления)
  3. UVa 00413 – Up and Down Sequences (симуляция; операции с массивом)
  4. **UVa 00443 – Humble Numbers *** (переберите все $2^i \times 3^j \times 5^k \times 7^l$, отсортируйте результат)
  5. UVa 00640 – Self Numbers (восходящий метод DP, сгенерируйте числа, пометьте флагом один раз)
  6. UVa 00694 – The Collatz Sequence (задача, аналогичная UVa 100)
  7. UVa 00962 – Taxicab Numbers (предварительно вычислить ответ)
  8. UVa 00974 – Kaprekar Numbers (чисел Капрекара не так много)
  9. UVa 10006 – Carmichael Numbers (непростые числа, которые имеют три или более простых множителей)
  10. **UVa 10042 – Smith Numbers *** (разложение на простые множители, просуммируйте все цифры)
  11. *UVa 10049 – Self-describing Sequence* (операции сортировки первых 700*K* чисел самоописываемой последовательности достаточно, чтобы выйти за пределы > 2*G*, указанные в качестве ограничения входных данных)
  12. UVa 10101 – Bangla Numbers (внимательно следите за крайне запутанным условием задачи)
  13. **UVa 10408 – Farey Sequences *** (сначала сгенерируйте пары (i, j) таким образом, чтобы `gcd(i, j) = 1`, затем отсортируйте)
  14. UVa 10930 – A-Sequence (задачаAd Hoc; следуйте правилам, приведенным в постановке задачи)
  15. *UVa 11028 – Sum of Product* (это «последовательность дартс»)
  16. UVa 11063 – B2 Sequences (посмотрите, повторяется ли число, будьте осторожны с ve)
  17. **UVa 11461 – Square Numbers** (ответ $\sqrt{b} - \sqrt{a-1}$)
  18. UVa 11660 – Look-and-Say sequences (симуляция, прерывание после j-го символа)
  19. UVa 11970 – Lucky Numbers (квадратные числа, проверка делимости, перебор)
- Логарифм, экспонента, степенная функция
  1. UVa 00107 – The Cat in the Hat (используйте логарифм, степенную функцию)
  2. UVa 00113 – Power Of Cryptography (используйте `exp(ln(x)*y)`)
  3. UVa 00474 – Heads Tails Probability (это всего лишь упражнение на использование логарифмической и степенной функций (`log& pow`))
  4. UVa 00545 – Heads (используйте логарифмическую и степенную функции, задача аналогична UVa 474)
  5. **UVa 00701 – Archaeologist's Dilemma *** (используйте мемоизацию для подсчета количества цифр)

6. *UVa 01185 – BigNumber* (число разрядов факториала, используйте логарифмическую функцию для решения; $\log(n!) = \log(n \times (n-1) \ldots \times 1) = \log(n) + \log(n-1) + \ldots + \log(1)$)
7. **UVa 10916 – Factstone Benchmark *** (используйте логарифмическую и степенную функции)
8. *UVa 11384 – Help is needed for Dexter* (найти наименьшую степень двойки, образующую число, большее, чем n; можно легко решить с помощью $ceil(eps + \log_2(n))$)
9. *UVa 11556 – Best Compression Ever* (связано со степенью двойки, используйте long long)
10. UVa 11636 – Hello World (используйте логарифм)
11. UVa 11666 – Logarithms (найдите формулу)
12. UVa 11714 – Blind Sorting (используйте модель дерева решений, чтобы найти минимум и второй минимум; конечный вариант решения включает только логарифм)
13. **UVa 11847 – Cut the Silver Bar *** (существует математическая формула, алгоритм реализации которой будет работать за $O(1)$: $\log_2(n)$)
14. UVa 11986 – Save from Radiation ($\log_2(N+1)$; ручная проверка на точность)
15. *UVa 12416 – Excessive Space Remover* (ответ $\log_2$ от максимального числа последовательных пробелов в строке)

- Многочлены
  1. *UVa 00126 – The Errant Physicist* (умножение многочленов и утомительное форматирование выходных данных)
  2. UVa 00392 – Polynomial Showdown (следуйте инструкциям в условии задачи: форматирование вывода)
  3. **UVa 00498 – Polly the Polynomial *** (вычисление многочлена)
  4. *UVa 10215 – The Largest/Smallest Box* (два тривиальных случая для самых маленьких объемов коробки; выведите формулу для самого большого объема, которая содержит квадратное уравнение)
  5. **UVa 10268 – 498' *** (дифференцирование многочлена; правило Горнера)
  6. UVa 10302 – Summation of Polynomials (используйте long double)
  7. UVa 10326 – The Polynomial Equation (если известны корни полинома, восстановить полином; форматирование)
  8. **UVa 10586 – Polynomial Remains *** (деление; операции с коэффициентами)
  9. UVa 10719 – Quotient Polynomial (деление многочленов с остатком)
  10. *UVa 11692 – Rain Fall* (используйте алгебраические операции, чтобы вывести квадратное уравнение; решите его; рассмотрите особый случай, когда $H < L$)

- Представление чисел в разных системах счисления
  1. **UVa 00377 – Cowculations *** (операции в системе счисления с основанием 4)
  2. **UVa 00575 – Skew Binary *** (перевод из одной системы счисления в другую)

3. UVa 00636 – Squares (преобразование основания системы счисления до 99; невозможно использовать Java BigInteger, так как MAX RADIX ограничен 36)
4. UVa 10093 – An Easy Problem (попробуйте все значения)
5. UVa 10677 – Base Equality (попробуйте все значения от $r2$ до $r1$)
6. **UVa 10931 – Parity *** (преобразовать из десятичной в двоичную систему, подсчитать число единиц («1»))
7. UVa 11005 – Cheapest Base (попробуйте все возможные основания систем счисления от 2 до 36)
8. UVa 11121 – Base-2 (поищите в интернете термин «негабинарный» (negabinary))
9. *UVa 11398 – The Base-1 Number System* (просто изучите новые правила и реализуйте их)
10. *UVa 12602 – Nice Licence Plates* (простое преобразование чисел из одной системы счисления в другую)
11. *SPOJ 0739 – The Moronic Cowmpouter* (найти представление чисел в системе счисления с основанием –2)
12. IOI 2011 – Alphabets (используйте более компактную запись в системе счисления с основанием 26)

- Другие специальные задачи

1. UVa 00276 – Egyptian Multiplication (умножение египетских иероглифов)
2. UVa 00496 – Simply Subsets (операции над множествами)
3. *UVa 00613 – Numbers That Count* (проанализируйте число; определите тип; задача похожа на задачу нахождения цикла в разделе 5.7)
4. **UVa 10137 – The Trip *** (остерегайтесь потери точности)
5. UVa 10190 – Divide, But Not Quite… (смоделируйте процесс)
6. *UVa 11055 – Homogeneous Squares* (неклассическая задача, чтобы избежать решения перебором, необходима догадка, основанная на наблюдательности)
7. *UVa 11241 – Humidex* (самый сложный случай – вычисление точки росы при заданной температуре и значении индекса; выведите формулу, используя знания алгебры)
8. **UVa 11526 – H(n) *** (перебирайте до $\sqrt{n}$, найдите закономерность, избегайте превышения лимита времени (вердикта тестирующей системы «TLE»))
9. UVa 11715 – Car (моделирование физического процесса)
10. UVa 11816 – HST (простая математика, требуется точность)
11. ***UVa 12036 – Stable Grid **** (используйте принцип Дирихле)

5.3. Класс Java BigInteger

5.3.1. Основные функции

Когда промежуточный и/или конечный результат математической операции вычислений с целыми числами невозможно сохранить, применяя самый большой встроенный целочисленный тип данных, либо данная задача не может быть решена с помощью разложения числа на множители, равные степеням простых чисел (см. раздел 5.5.5) или арифметических операций по модулю (см. раздел 5.5.8), у нас остается только один выбор – использовать библиотеки BigInteger (aka bignum). Пример: вычислите точное значение 25! (факториал 25). Результат: 15 511 210 043 330 985 984 000 000 (26 цифр). Это явно слишком большой размер для типа данных `unsigned long long` в 64-разрядной версии C/C ++ (или `long` в Java).

Одним из способов реализации библиотеки BigInteger является хранение чисел BigInteger в виде (длинной) строки[1]. Например, мы можем хранить число 10^{21} в строке `num1` = «1 000 000 000 000 000 000 000», в то время как в 64-разрядной версии C/C ++ `unsigned long long` (или в `long` в Java) это уже приведет к переполнению памяти. Затем для обычных математических операций мы можем использовать метод «цифра за цифрой» для обработки двух операндов BigInteger. Например, если `num2` = «173», мы выполняем операцию `num1 + num2` следующим образом:

```
num1        = 1,000,000,000,000,000,000,000
num2        =                           173
            ------------------------------ +
num1 + num2 = 1,000,000,000,000,000,000,173
```

Мы также можем вычислить произведение этих чисел `num1 * num2`:

```
num1        =   1,000,000,000,000,000,000,000
num2        =                             173
               ------------------------------ *
                3,000,000,000,000,000,000,000
               70,000,000,000,000,000,000,00
              100,000,000,000,000,000,000,0
               ------------------------------ +
num1 * num2 = 173,000,000,000,000,000,000,000
```

---

[1] На самом деле во встроенных типах данных числа также хранятся в виде ограниченной строки битов в памяти компьютера. Например, 32-разрядный тип данных `int` хранит целое число как 32 бита двоичной строки. Метод хранения, используемый в BigInteger, – это всего лишь обобщение данного метода, который применяет десятичную форму представления (основание 10) и более длинную строку цифр. Примечание: класс Java BigInteger, вероятно, использует более эффективный метод, чем тот, который показан в этом разделе.

Сложение и вычитание – две самые простые операции в BigInteger. Умножение потребует немного больше усилий при написании программного кода, как видно из приведенного выше примера. Эффективная реализация операции деления и возведения целого числа в степень еще более сложна. В любом случае, написание кода для реализации этих библиотечных функций в C/C ++ прямо на соревновании может привести к ошибкам в коде даже в ситуации, когда правила соревнований разрешают приносить распечатанный текст такой библиотеки для C/C++[1]. К счастью, в Java есть класс BigInteger, который мы можем использовать для этой цели. По состоянию на 24 мая 2013 года в C ++ STL такой функции нет, поэтому рекомендуется применять Java для решения задач, в которых предполагается использовать BigInteger.

Класс Java BigInteger (мы сокращенно называем его BI) поддерживает следующие основные целочисленные операции: сложение – `add(BI)`, вычитание – `subtract(BI)`, умножение – `multiply(BI)`, возведение в степень – `pow(int exponent)`, деление – `divide(BI)`, вычисление остатка – `remainder(BI)`, остаток целочисленного деления – `mod(BI)` (отличается от `remainder(BI)`), деление и остаток – `divideAndRemainder(BI)` и некоторые другие интересные функции, которые будут обсуждаться далее. Все они занимают лишь одну строчку кода.

Однако нужно заметить, что все операции BigInteger выполняются медленнее, чем те же операции со стандартными 32/64-разрядными целочисленными типами данных. Полезное правило: если для решения математической задачи вы можете применять другой алгоритм, для которого требуется только встроенный целочисленный тип данных, используйте его вместо обращения к BigInteger.

Тем читателям, которые еще не использовали класс Java BigInteger, мы предлагаем изучить приведенный ниже короткий код на Java, который является решением для задачи UVa 10925 – Krakovia. Для решения этой задачи требуется выполнить операции сложения (для суммирования *N* больших счетов) и деления BigInteger (чтобы разделить большую сумму на *F* друзей). Обратите внимание, насколько короток и понятен код; сравните это с ситуацией, когда вам нужно писать собственную реализацию BigInteger.

```
import java.util.Scanner;                                    // внутри пакета java.util
import java.math.BigInteger;                                 // внутри пакета java.math

class Main {    /* UVa 10925 - Krakovia */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int caseNo = 1;
    while (true) {
      int N = sc.nextInt(), F = sc.nextInt();                      // N счетов, F друзей
      if (N == 0 && F == 0) break;
      BigInteger sum = BigInteger.ZERO;                // в BigInteger пароль выглядит так
      for (int i = 0; i < N; i++) {                       // суммирование N больших счетов
        BigInteger V = sc.nextBigInteger();          // для считывания следующего BigInteger!
        sum = sum.add(V);                                      // это сложение BigInteger
      }
```

[1] Хорошие новости для участников IOI. Для решения задач IOI обычно не требуется, чтобы участники использовали BigInteger.

```
      System.out.println("Счет #" + (caseNo++) + " сумма " + sum +
        ": каждый из друзей должен заплатить " + sum.divide(BigInteger.valueOf(F)));
      System.out.println();                    // предыдущая строка - деление BigInteger
    }                                              // разделим большую сумму на F друзей
  }
}
```

Файл исходного кода: *ch5_01_UVa10925.java*

Упражнение 5.3.1.1. Можем ли мы использовать встроенный тип данных для вычисления последней ненулевой цифры числа «25!»?

Упражнение 5.3.1.2. Проверьте, делится ли 25! на 9317. Можем ли мы использовать при проверке встроенный тип данных?

5.3.2. Дополнительные функции

Класс Java BigInteger имеет несколько дополнительных функций, которые могут быть более полезны на олимпиадах по программированию (с точки зрения сокращения длины кода) по сравнению с тем случаем, когда нам приходится самим реализовывать эти функции[1]. Класс Java BigInteger имеет целый набор встроенных функций: конструктор класса и функцию, преобразующую числа: `toString(int radix)`, очень хорошую (но вероятностную) функцию, проверяющую, является ли заданное число BigInteger простым или составным: `isProbablePrime(int sureness)`, подпрограмму нахождения наибольшего общего делителя (GCD): `gcd(BI)` и функцию возведения в степень по модулю: `modPow(BI exponent, BI m)`. Среди этих дополнительных функций наиболее полезной и часто используемой является функция, преобразующая число в строку; следующей по степени полезности и частоте применения является функция, проверяющая, является ли заданное число BigInteger простым или составным.

Эти дополнительные функции продемонстрированы в четырех примерах из онлайн-архива задач университета Вальядолида (UVa).

Преобразование чисел

См. пример ниже для UVa 10551 – Basic Remains. Пусть задано основание системы счисления b и два неотрицательных целых числа p и m – оба в системе счисления, имеющей основание b. Вычислите $p\%\ m$ и выведите результат в виде целого числа в системе счисления с основанием b.

Преобразование чисел на самом деле является не такой сложной математической задачей[2], но эту задачу можно решить еще проще с помощью клас-

---

[1] Примечание для программистов, предпочитающих писать на «чистом C/C ++»: хорошо быть программистом, владеющим несколькими языками программирования, – это дает возможность переходить на Java в случае, когда такой переход выгоднее.

[2] Например, чтобы преобразовать число 132, записанное в системе счисления с основанием 8 (восьмеричной системе), в систему счисления с основанием 2 (двоичную систему), мы можем использовать основание 10 (десятичное) в качестве промежуточного

са Java BigInteger. Мы можем создать и вывести на печать объект класса Java BigInteger в системе счисления с любым основанием, как показано ниже:

```
class Main { /* UVa 10551 - Basic Remains */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while (true) {
      int b = sc.nextInt();
      if (b == 0) break;                                  // конструктор специального класса
      BigInteger p = new BigInteger(sc.next(), b);                        // второй параметр
      BigInteger m = new BigInteger(sc.next(), b);          // основание системы счисления
      System.out.println((p.mod(m)).toString(b));                  // можно выводить число
                                                                  // в любой системе счисления
} } }
```

Файл исходного кода: *ch5_02_UVa10551.java*

Вероятностная проверка, является ли заданное число BigInteger простым или составным

Далее в разделе 5.5.1 мы обсудим алгоритм решета Эратосфена и детерминистский алгоритм проверки, является ли число простым или составным, который применим для многих олимпиадных задач. Тем не менее вам все же потребуется написать несколько строк кода на C/C ++ или Java. Если вам просто нужно проверить, является ли одно большое число (или несколько[1] больших чисел) простым или составным (например, как в задаче UVa 10235, рассматриваемой далее), есть альтернативный и более короткий подход, использующий функцию `isProbablePrime` в Java BigInteger, – это вероятностная функция проверки простых чисел, основанная на алгоритме Миллера–Рабина [44, 55]. У этой функции есть важный параметр – достоверность (`certainty`). Если эта функция возвращает «true», то вероятность того, что заданное число BigInteger является простым, превышает $1 - (1/2)^{certainty}$. Для типичных олимпиадных задач значение `certainty` = 10 должно быть достаточным, поскольку $1 - (1/2)^{10} =$ 0,9990234375 составляет ≈ 1,0. Обратите внимание, что использование большего значения достоверности, очевидно, уменьшает вероятность получения неправильного ответа (WA), но в то же время оно замедляет работу вашей программы и, таким образом, увеличивает риск превышения лимита времени (TLE). Попробуйте выполнить **упражнение 5.3.2.3***, чтобы убедиться на собственном опыте.

```
class Main { /* UVa 10235 - Simply Emirp */
  public static void main(String[] args) {
```

шага преобразования: $(132)_8$ равно $1 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 = 64 + 24 + 2 = (90)_{10}$ и $(90)_{10}$ равно $90 \rightarrow 45(0) \rightarrow 22(1) \rightarrow 11(0) \rightarrow 5(1) \rightarrow 2(1) \rightarrow 1(0) \rightarrow 0\ (1) = (1011010)_2$ (то есть делим наше число на 2 до тех пор, пока не получим остаток 0, а затем читаем остатки от деления справа налево (т. е. «задом наперед»).

[1] Обратите внимание, что если ваша цель – получить список из первых нескольких миллионов простых чисел, алгоритм решета Эратосфена, приведенный в разделе 5.5.1, должен работать быстрее, чем несколько миллионов вызовов функции `isProbablePrime`.

```
    Scanner sc = new Scanner(System.in);
    while (sc.hasNext()) {
      int N = sc.nextInt();
      BigInteger BN = BigInteger.valueOf(N);
      String R = new StringBuffer(BN.toString()).reverse().toString();
      int RN = Integer.parseInt(R);
      BigInteger BRN = BigInteger.valueOf(RN);
      System.out.printf("%d is ", N);
      if (!BN.isProbablePrime(10))                          // значения certainty-10 is
                                                  // в большинстве случаев достаточно
        System.out.println("не является простым (not prime).");
      else if (N != RN && BRN.isProbablePrime(10))
        System.out.println("эмирп (emirp).");
      else
        System.out.println("простое (prime).");
} } }
```

Файл исходного кода: *ch5_03_UVa10235.java*

Наибольший общий делитель (Greatest Common Divisor, GCD)

Ниже приведен пример кода для решения задачи UVa 10814 – Simplifying Fractions. В задаче требуется привести дробь к канонической форме, разделив числитель и знаменатель на их наибольший общий делитель.

Также см. раздел 5.5.2 для более подробной информации о наибольшем общем делителе.

```
class Main { /* UVa 10814 - Simplifying Fractions */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int N = sc.nextInt();
    while (N-- > 0) {                  // в отличие от C/C++, мы используем > 0 в (N --> 0)
      BigInteger p = sc.nextBigInteger();
      String ch = sc.next();                    // игнорируем знак деления во входных данных
      BigInteger q = sc.nextBigInteger();
      BigInteger gcd_pq = p.gcd(q);                                          // wow :)
      System.out.println(p.divide(gcd_pq) + " / " + q.divide(gcd_pq));
} } }
```

Арифметика по модулю

Ниже приведен пример кода для решения задачи UVa 1230 (LA 4104) – MODEX, который вычисляет значение $x^y \pmod n$. Также просмотрите разделы 5.5.8 и 9.21, чтобы увидеть, каким образом выполняется вычисление функции `modPow`.

```
class Main { /* UVa 1230 (LA 4104) - MODEX */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int c = sc.nextInt();
    while (c-- > 0) {
      BigInteger x = BigInteger.valueOf(sc.nextInt());          // valueOf преобразует
      BigInteger y = BigInteger.valueOf(sc.nextInt());          // простое целое число
```

```
      BigInteger n = BigInteger.valueOf(sc.nextInt());                    // в BigInteger
      System.out.println(x.modPow(y, n));                          // она есть в библиотеке!
} } }
```

Файл исходного кода: *ch5_05_UVa1230.java*

Упражнение 5.3.2.1. Попробуйте решить задачу UVa 389, используя функцию Java BigInteger, с которой вы познакомились в этом разделе. Можете ли вы сделать это, не превышая лимит времени (т. е. не получив вердикт тестирующей системы «TLE»)? Если нет, существует ли (несколько) лучший способ решения этой задачи?

Упражнение 5.3.2.2*. По состоянию на 2013 год на олимпиадах по программированию все еще довольно редко встречаются задачи, включающие операции с десятичными числами с *произвольной точностью* (не обязательно целыми числами). На сегодня мы нашли только две задачи, предлагаемые в архиве задач университета Вальядолида (UVa), где требуется использовать эту функцию: UVa 10464 и UVa 11821. Попробуйте решить эти две задачи, используя другую библиотеку: класс Java BigDecimal. Изучите следующий дополнительный материал: http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html.

Упражнение 5.3.2.3*. Напишите программу на Java, чтобы *эмпирически* определить минимальное значение параметра `certainty`, такое, чтобы наша программа могла работать быстро, и для диапазона значений входного параметра [2…10*M*] – типичного диапазона для олимпиадных задач – случайно не определялся неправильный тип числа, т. е. составное число не определялось как простое при использовании `isProbablePrime(certainty)`. Поскольку `isProbablePrime` использует вероятностный алгоритм, вы должны повторить свой эксперимент несколько раз для каждого значения параметра `certainty`. Достаточно ли `certainty` = 5? А `certainty` = 10? А `certainty` = 1000?

Упражнение 5.3.2.4*. Изучите и реализуйте алгоритм проверки простых чисел Миллера–Рабина (см. [44, 55]) на тот случай, если вам потребуется реализовать его на C/C ++.

Задачи по программированию, связанные с использованием BigInteger, не упоминаемые в других разделах[1]

- Основные функции
  1. UVa 00424 – Integer Inquiry (сложение чисел BigInteger)
  2. UVa 00465 – Overflow (BigInteger добавить/умножить, сравнить с $2^{31} - 1$)
  3. UVa 00619 – Numerically Speaking (BigInteger)
  4. **UVa 00713 – Adding Reversed …*** (BigInteger + StringBuffer reverse ())

[1] Стоит отметить, что в других разделах этой главы (а также в других главах) есть много иных задач по программированию, в которых также используется BigInteger.

5. UVa 00748 – Exponentiation (возведение в степень чисел BigInteger)
6. UVa 01226 – Numerical surprises (LA 3997, Дананг'07, деление по модулю)
7. UVa 10013 – Super long sums (сложение чисел BigInteger)
8. UVa 10083 – Division (BigInteger + теория чисел)
9. UVa 10106 – Product (умножение чисел BigInteger)
10. UVa 10198 – Counting (рекурсия, BigInteger)
11. *UVa 10430 – Dear GOD* (BigInteger, сначала выведите формулу)
12. *UVa 10433 – Automorphic Numbers* (BigInteger, операторы возведения в степень, вычитания, деления по модулю)
13. UVa 10494 – If We Were a Child Again (деление BigInteger)
14. UVa 10519 – Really Strange (рекурсия, BigInteger)
15. **UVa 10523 – Very Easy *** (сложение, умножение и возведение в степень чисел BigInteger)
16. UVa 10669 – Three powers (BigInteger для 3^n, двоичное представление множества)
17. UVa 10925 – Krakovia (BigInteger, сложение и деление)
18. *UVa 10992 – The Ghost of Programmers* (размер входных данных до 50 цифр)
19. UVa 11448 – Who said crisis? (Вычитание BigInteger)
20. *UVa 11664 – Langton's Ant* (простое моделирование с использованием BigInteger)
21. UVa 11830 – Contract revision (используйте строковое представление BigInteger)
22. **UVa 11879 – Multiple of 17 *** (BigInteger, операции деления по модулю, деления, вычитания, равенства)
23. *UVa 12143 – Stopping Doom's Day* (LA 4209, Дакка'08, упрощение формул – сложная часть; использование BigInteger – легкая часть)
24. *UVa 12459 – Bees' ancestors* (нарисуйте дерево предков, чтобы увидеть закономерность)

- Дополнительная функция: преобразование чисел между системами счисления
  1. **UVa 00290 – Palindroms ←→ ...** (кроме преобразований чисел, используются палиндромы)
  2. **UVa 00343 – What Base Is This? *** (попробуйте все возможные пары оснований)
  3. UVa 00355 – The Bases Are Loaded (преобразование чисел из одной системы счисления в другую)
  4. **UVa 00389 – Basically Speaking *** (используйте класс Integer в Java)
  5. UVa 00446 – Kibbles 'n' Bits 'n' Bits... (преобразование чисел из одной системы счисления в другую)
  6. UVa 10473 – Simple Base Conversion (из десятичной системы счисления в шестнадцатеричную и обратно; если вы реализуете эту возможность на C/C ++, то можете использовать `strtol`)
  7. **UVa 10551 – Basic Remains *** (включает операции по модулю с BigInteger)

8. UVa 11185 – Ternary (преобразование чисел из десятичной системы в систему с основанием 3)
9. *UVa 11952 – Arithmetic* (проверяйте только преобразование из системы счисления с основанием 2 в систему счисления с основанием 18; особый случай – основание, равное 1)

- Дополнительная функция: проверка, является ли заданное число простым или составным
  1. *UVa 00960 – Gaussian Primes* (формулировка условий этой задачи – из области теории чисел)
  2. **UVa 01210 – Sum of Consecutive…** * (LA 3399, Токио'05, простая задача)
  3. **UVa 10235 – Simply Emirp** * (анализ: составное / простое / простое-«палиндром»; простое-«палиндром» определяется как простое число, которое при чтении его цифр в обратном порядке также остается простым числом)
  4. UVa 10924 – Prime Words (проверьте, является ли сумма буквенных значений простым числом)
  5. **UVa 11287 – Pseudoprime Numbers** * (выведите «да», если `!IsPrime(p) + a.modPow(p, p) = a`; используйте Java BigInteger)
  6. *UVa 12542 – Prime Substring* (HatYai12, используйте перебор, также используйте isProbablePrime для проверки, является ли число простым)

- Дополнительные функции: другие
  1. **UVa 01230 – MODEX** * (LA 4104, Сингапур'07, операция modPow с BigInteger)
  2. *UVa 10023 – Square root* (код, реализующий метод Ньютона, использование BigInteger)
  3. UVa 10193 – All You Need Is Love (преобразуйте две двоичные строки S1 и S2 в строку в десятичной системе и проверьте, выполняется ли условие gcd(s1, s2) > 1)
  4. UVa 10464 – Big Big Real Numbers (задачу можно решить, используя класс Java BigDecimal)
  5. **UVa 10814 – Simplifying Fractions** * (используйте BigInteger, gcd)
  6. **UVa 11821 – High-Precision Number** * (используйте класс Java BigDecimal)

Известные авторы алгоритмов

Гари Ли Миллер – профессор информатики университета Карнеги-Меллона. Он является первым изобретателем алгоритма Миллера–Рабина.

Майкл Озер Рабин (род. 1931) – израильский ученый, специалист в области теории вычислительных систем. Он усовершенствовал идею Миллера и изобрел алгоритм проверки, является ли заданное число простым, – алгоритм Миллера–Рабина (тест Миллера–Рабина). Вместе с Ричардом Мэннингом Карпом он также изобрел алгоритм Рабина–Карпа – алгоритм сравнения строк.

5.4. Комбинаторика

Комбинаторика – это раздел дискретной математики[1], касающийся изучения счетных дискретных структур. В олимпиадном программировании задачи, связанные с комбинаторикой, обычно начинаются с вопроса «Сколько [объектов]», «Подсчитайте [объекты]» и т. п., хотя многие авторы задач предпочитают не раскрывать суть вопроса сразу в названии задач. Код решения таких задач обычно достаточно короткий, но для получения формулы (как правило, рекурсивной) потребуется определенный уровень математической подготовки, а также терпение.

Если вы принимаете участие в соревнованиях ICPC[2], то, встретившись с подобной задачей, попросите одного члена команды, который силен в математике, вывести формулу, в то время как двое других членов команды сосредоточатся на других задачах. Быстро напишите код для реализации формулы (как правило, короткой), как только эта формула будет выведена – прервите для этого любого члена команды, который в данный момент использует компьютер. Также полезно запомнить/выучить общие формулы, такие как формулы, связанные с числами Фибоначчи (см. раздел 5.4.1), биномиальные коэффициенты (см. раздел 5.4.2) и числа Каталана (см. раздел 5.4.3).

Для некоторых из формул комбинаторики могут существовать перекрывающиеся подзадачи, что означает необходимость использования динамического программирования (см. раздел 3.5). Некоторые результаты вычислений также могут быть большими числами, и в этом случае может понадобиться использовать BigInteger (см. раздел 5.3).

5.4.1. Числа Фибоначчи

Числа Леонардо *Фибоначчи* определяются как $fib(0) = 0$, $fib(1) = 1$, а для $n \geqslant 2$ $fib(n) = fib(n - 1) + fib(n - 2)$. Эта формула задает следующую широко известную последовательность: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 и т. д. Эта последовательность иногда появляется в олимпиадных задачах по программированию, где вообще не упоминается слово «Фибоначчи», например в некоторых задачах, приведенных в этом разделе (UVa 900, 10334, 10450, 10497, 10862 и т. д.).

Обычно мы выводим числа Фибоначчи с помощью «тривиального» способа DP со сложностью по времени $O(n)$, а не реализуем приведенную формулу «в лоб» с помощью рекурсии (поскольку подобное решение будет работать очень медленно). Однако решение с использованием DP со сложностью по времени $O(n)$ не является самым быстрым для всех случаев. Далее в разделе 9.21 мы покажем способ вычисления n-го числа Фибоначчи (где n велико) за время $O(\log n)$, используя перемножение матриц. Примечание: существует метод *an-*

---

[1] Дискретная математика – это изучение структур, которые являются дискретными (как, например, целые числа {0, 1, 2, ...}, графы/деревья (вершины и ребра), логические величины (истина/ложь)), а не непрерывными (как, например, действительные числа).

[2] Обратите внимание, что задачи из области чистой комбинаторики редко встречаются в программе IOI (такая задача может быть частью более крупной задачи).

проксимации с временной сложностью $O(1)$ для получения n-го числа Фибоначчи. Мы можем вычислить ближайшее целое число $(\phi^n - (-\phi)^{-n})/\sqrt{5}$ (формула Бине), где ф (золотое сечение) равно $((1 + \sqrt{5})/2) \approx 1.618$. Однако эта формула не дает точный результат для больших чисел Фибоначчи.

Последовательность чисел Фибоначчи растет очень быстро, поэтому некоторые задачи, связанные с числами Фибоначчи, нужно решать, используя библиотеку Java BigInteger (см. раздел 5.3).

Числа Фибоначчи имеют много интересных свойств. Одной из них является теорема Цекендорфа: каждое положительное целое число можно единственным образом представить в виде суммы одного или нескольких различных чисел Фибоначчи так, чтобы в этом представлении не оказалось двух соседних чисел из последовательности Фибоначчи. Для любого заданного натурального числа представление, удовлетворяющее теореме Цекендорфа, можно найти с помощью «жадного» алгоритма: выбирайте максимально возможное число Фибоначчи на каждом шаге. Например: 100 = 89 + 8 + 3; 77 = 55 + 21 + 1, 18 = 13 + 5 и т. д.

Другим интересным свойством является период Пизано, в котором последняя / последние две / последние три / последние четыре цифры числа Фибоначчи повторяются с периодичностью 60/300/1500/15000 соответственно.

Упражнение 5.4.1.1. Попробуйте использовать формулу Бине $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$ при малых n и посмотрите, действительно ли эта формула выдает $fib(7) = 13$, $fib(9) = 34$, $fib(11) = 89$. Теперь напишите простую программу, чтобы определить первое значение n, такое, что фактическое значение $fib(n)$ отличается от результата этой аппроксимации. Насколько это значение n обычного использования формулы Бине в соревнованиях по программированию?

5.4.2. Биномиальные коэффициенты

Другая классическая задача комбинаторики заключается в нахождении *коэффициентов* полинома, полученного при раскрытии степени бинома[1]. Эти коэффициенты также представляют собой количество способов, которыми из n элементов возможно составить набор, состоящий из k элементов, обычно это записывается как $C(n, k)$, или C_n^k. Например, $(x + y)^3 = 1x^3 + 3x^2y + 3xy^2 + 1y^3$. Числа {1, 3, 3, 1} являются биномиальными коэффициентами для $n = 3$, при этом $k = \{0, 1, 2, 3\}$. Или, другими словами, количество способов, которыми из $n = 3$ элементов можно выбрать $k = \{0, 1, 2, 3\}$ элементов за один раз, составляет {1, 3, 3, 1} соответственно.

Мы можем вычислить значение $C(n, k)$, используя следующую формулу: $C(n, k) = n!/((n - k)! \times k!)$. Однако вычислить значение $C(n, k)$ может оказаться непросто, когда n и/или k – большие числа. Существует несколько приемов, упрощающих вычисления, например: уменьшить k (если $k > n - k$, то мы переопределим $k = n - k$), потому что $C_n^k = C_n^{(n-k)}$; во время промежуточных вычислений мы сначала делим числа, а затем умножаем их на следующее число.

---

[1] Бином является частным случаем полинома, который имеет только два члена.

Или же можно использовать BigInteger (в крайнем случае, поскольку операции с BigInteger выполняются медленно).

Если нам нужно вычислить *много*, *но не все значения* $C(n, k)$ для разных n и k, лучше использовать нисходящий метод динамического программирования. Мы можем записать $C(n, k)$, как показано ниже, и использовать двумерную таблицу memo, чтобы избежать повторных вычислений.

$C(n, 0) = C(n, n) = 1$ // тривиальные случаи.
$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$ // выбираем или не выбираем элемент, $n > k > 0$.

Однако если нам нужно получить все значения $C(n, k)$ от $n = 0$ до определенного значения n, тогда может быть полезно построить *треугольник Паскаля*, треугольный массив биномиальных коэффициентов. Самые крайние левые и самые крайние правые записи в каждой строке всегда равны 1. Внутренние значения представляют собой сумму двух значений непосредственно над ней, как показано для строки $n = 4$ ниже. Это, по сути, другая версия решения, использующая восходящий метод DP (решение с использованием нисходящего метода записано выше).

```
n = 0             1
n = 1           1   1
n = 2         1   2   1
n = 3       1   3   3   1 <- как показано выше
             \ / \ / \ /
n = 4     1   4   6   4   1 ... и т. д.
```

Упражнение 5.4.2.1. В задачах, где применяются биномиальные коэффициенты $C(n, k)$, часто используется значение $k = 2$. Покажите, что $C(n, 2) = O(n^2)$.

5.4.3. Числа Каталана

Во-первых, давайте определим n-е число Каталана – записанное с использованием приведенного выше обозначения биномиальных коэффициентов C_n^k – следующим образом: $Cat(n) = (C^n_{(2\times n)})/(n + 1)$; $Cat(0) = 1$. Далее мы увидим, для чего нам могут понадобиться эти числа.

Если нас попросят вычислить значения $Cat(n)$ для *нескольких* значений n, то, возможно, лучше использовать восходящий метод динамического программирования. Если мы знаем $Cat(n)$, мы можем вычислить $Cat(n + 1)$, используя формулу, приведенную ниже.

$$Cat(n) = \frac{2n!}{n!\times n!\times(n+1)}.$$

$$Cat(n+1) = \frac{(2\times(n+1))!}{(n+1)!\times(n+1)!\times((n+1)+1)} = \frac{(2n+2)\times(2n+1)\times 2n!}{(n+1)\times n!\times(n+1)\times n!\times(n+2)} =$$

$$= \frac{(2n+2)\times(2n+1)\times\ldots[2n!]}{(n+2)\times(n+1)\times\ldots[n!\times n!\times(n+1)]}.$$

Следовательно, $Cat(n+1) = \frac{(2n+2)\times(2n+1)}{(n+2)\times(n+1)} \times Cat(n)$.

В качестве альтернативы мы можем задать $m = n + 1$ таким образом, чтобы: $Cat(m) = \frac{2m\times(2m-1)}{(m+1)\times m} \times Cat(m-1)$.

Числа Каталана встречаются в различных задачах на комбинаторику. Здесь мы перечислим некоторые из наиболее интересных (есть еще несколько, см. **упражнение 5.4.4.8***). Во всех приведенных ниже примерах используется $n = 3$, и $Cat(3) = (C^3_{(2\times3)})/(3+1) = (C^3_6)/4 = 20/4 = 5$.

1. $Cat(n)$ вычисляет количество различных двоичных деревьев с n вершинами, например для $n = 3$:

2. $Cat(n)$ подсчитывает количество выражений, содержащих n пар скобок, расставленных правильно, например для $n = 3$ имеем: () () (), () (()), (()) (), ((())) и (() ()).
3. $Cat(n)$ подсчитывает количество различных способов, которыми $n + 1$ множителей можно заключить в скобки, например для $n = 3$ и $3 + 1 = 4$ множителей {a, b, c, d} имеем: (ab)(cd), a(b(cd)), ((ab)c)d, (a(bc))(d) и a((bc)d).
4. $Cat(n)$ подсчитывает количество способов триангуляции выпуклого многоугольника (см. раздел 7.3) с $n + 2$ сторонами. См. рис. 5.1, слева.
5. $Cat(n)$ подсчитывает количество монотонных путей по краям сетки $n\times n$, которые не проходят выше диагонали. Монотонный путь – это путь, который начинается в левом нижнем углу, заканчивается в правом верхнем углу и полностью состоит из ребер, направленных вправо или вверх. См. рис. 5.1, справа; также см. раздел 4.7.1.

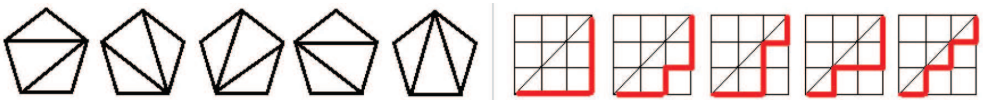


Рис. 5.1 ❖ Слева: триангуляция выпуклого многоугольника, справа: монотонные пути

Замечания о задачах, использующих комбинаторику, на олимпиадах по программированию

Есть много других задач на комбинаторику, которые также могут появляться на олимпиадах по программированию; однако они встречаются не так часто, как задачи, использующие числа Фибоначчи, биномиальные коэффициенты или числа Каталана. Некоторые наиболее интересные из них приведены в разделе 9.8.

В онлайн-соревнованиях по программированию, где участник может получить доступ к интернету, есть еще один полезный прием. Сначала сгене-

рируйте выходные данные для небольших примеров, а затем найдите эту последовательность в OEIS (онлайн-энциклопедии целочисленных последовательностей), размещенной на сайте http://oeis.org/. Если вам повезет, OEIS может сообщить вам название последовательности и/или подсказать нужную общую формулу для больших значений членов последовательности.

Есть еще множество других приемов и формул, однако их слишком много, чтобы обсуждать их все в этой книге. Мы завершаем данный раздел задачами для проверки / дальнейшего улучшения ваших навыков решения задач из области комбинаторики. Примечание: число задач, приведенных в этом разделе, составляет ≈ 15 % от общего числа задач в этой главе.

Упражнение 5.4.4.1. Подсчитайте количество различных возможных результатов, если вы бросите два шестигранных кубика и подбросите две двусторонние монеты.

Упражнение 5.4.4.2. Сколько существует способов сформировать трехзначное число из набора цифр {0, 1, 2, …, 9}, если каждая цифра может встречаться в сформированном числе только один раз? Обратите внимание, что 0 не может использоваться как начальная цифра.

Упражнение 5.4.4.3. Предположим, у вас есть шестибуквенное слово «FACTOR». Если мы возьмем три буквы из слова «FACTOR», у нас может получиться другое английское слово из трех букв, например «ACT», «CAT», «ROT» и т. д. Какое максимальное количество различных трехбуквенных слов может быть составлено из букв, составляющих слово «FACTOR»? При решении задачи вам не нужно принимать во внимание то, является трехбуквенное слово словарным английским словом или же нет.

Упражнение 5.4.4.4. Предположим, у вас есть пятибуквенное слово «BOBBY». Если мы перегруппируем буквы, то можем получить другое слово, например «BBBOY», «YOBBB» и т. д. Сколько существует возможных вариантов таких перестановок?

Упражнение 5.4.4.5. Решите задачу UVa 11401 – Triangle Counting. Ее можно кратко сформулировать следующим образом: «Имеется n стержней длиной 1, 2, …, n; выберите любые три из них и постройте треугольник. Сколько разных треугольников вы можете построить (используйте неравенство треугольника, см. раздел 7.2)? ($3 \leqslant n \leqslant 1M$)». Обратите внимание, что два треугольника будут считаться разными, если они имеют хотя бы одну пару сторон разной длины. Если вам повезет, вы можете потратить всего несколько минут, чтобы найти ключ к решению. В противном случае эта задача может оказаться нерешенной до конца соревнований, что, безусловно, отрицательно отразится на результате вашей команды.

Упражнение 5.4.4.6*. Изучите следующие темы: лемма Бернсайда, числа Стирлинга.

Упражнение 5.4.4.7*. Пусть n – произвольное большое целое число. В каком из следующих случаев сложнее всего разложить число на множители (см. раздел 5.5.4): $fib(n)$, $C(n, k)$ (предположим, что $k = n/2$) или $Cat(n)$? Почему?

Упражнение 5.4.4.8*. Числа Каталана *Cat*(*n*) появляются в некоторых других интересных задачах, помимо тех, которые приведены в этом разделе. Исследуйте другие применения этих чисел.

Задачи по программированию, связанные с комбинаторикой

- Числа Фибоначчи
  1. UVa 00495 – Fibonacci Freeze (очень легко решается с использованием Java BigInteger)
  2. UVa 00580 – Critical Mass (эта задача использует последовательность чисел Трибоначчи; числа Трибоначчи являются обобщением чисел Фибоначчи; их последовательность определяется так: $T_1 = 1$, $T_2 = 1$, $T_3 = 2$ и $T_n = T_{n-1} + T_{n-2} + T_{n-3}$ для $n \geqslant 4$)
  3. **UVa 00763 – Fibinary Numbers *** (представление Цекендорфа; «жадный» алгоритм, используйте Java BigInteger)
  4. **UVa 00900 – Brick Wall Patterns** (комбинаторика, модель укладки ≈ Фибоначчи)
  5. UVa 00948 – Fibonaccimal Base (представление Цекендорфа, «жадный» алгоритм)
  6. UVa 01258 – Nowhere Money (LA 4721, Пхукет'09, вариант чисел Фибоначчи, представление Цекендорфа, «жадный» алгоритм)
  7. UVa 10183 – How many Fibs? (получить количество чисел Фибоначчи при их генерации; BigInteger)
  8. **UVa 10334 – Ray Through Glasses *** (комбинаторика, Java BigInteger)
  9. **UVa 10450 – World Cup Noise** (комбинаторика, шаблон ≈ Фибоначчи)
  10. **UVa 10497 – Sweet Child Make Trouble** (модель ≈ Фибоначчи)
  11. UVa 10579 – Fibonacci Numbers (задача очень легко решается с помощью Java BigInteger)
  12. **UVa 10689 – Yet Another Number... *** (задача решается легко, если вы знаете период Пизано (для чисел Фибоначчи))
  13. UVa 10862 – Connect the Cable Wires (закономерность похожа на Фибоначчи)
  14. UVa 11000 – Bee (комбинаторика, закономерность похожа на числа Фибоначчи)
  15. *UVa 11089 – Fi-binary Number* (список фи-двоичных чисел составляется с помощью теоремы Цекендорфа)
  16. UVa 11161 – Help My Brother (II) (Фибоначчи + медиана)
  17. UVa 11780 – Miles 2 Km (задача с использованием чисел Фибоначчи)
- Биномиальные коэффициенты
  1. UVa 00326 – Extrapolation using a... (таблица разностей)
  2. UVa 00369 – Combinations (будьте осторожны, может возникнуть ошибка переполнения)
  3. UVa 00485 – Pascal Triangle of Death (биномиальные коэффициенты + BigInteger)

4. UVa 00530 – Binomial Showdown (работа с вещественными числами (тип double); оптимизация вычислений)
5. *UVa 00911 – Multinomial Coefficients* (для этого есть формула, результат = $n!/(z_1! \times z_2! \times z_3! \times \ldots \times z_k!)$)
6. UVa 10105 – Polynomial Coefficients ($n!/(n_1! \times n_2! \times \ldots \times n_k!)$; однако вывод формулы сложен)
7. **UVa 10219 – Find the Ways *** (сосчитать длину C_n^k; BigInteger)
8. UVa 10375 – Choose and Divide (главная задача – избежать переполнения)
9. *UVa 10532 – Combination, Once Again* (видоизмененная задача, использующая биномиальные коэффициенты)
10. ***UVa 10541 – Stripe **** (хорошая задача на комбинаторику; подсчитайте, сколько белых клеток, с помощью $N_{white} = N$ – сумма всех K целых чисел; представьте, что у нас есть еще одна белая клетка в самом начале, теперь мы можем дать ответ, разместив черные полосы после K из $N_{white} + 1$ белых или ${}_{N_{white}+1}C_K$ (используйте Java BigInteger); однако если $K > N_{white} + 1$, ответ равен 0)
11. **UVa 11955 – Binomial Theorem *** (простое применение теоремы; DP)

- Числа Каталана
  1. **UVa 00991 – Safe Salutations *** (числа Каталана)
  2. **UVa 10007 – Count the Trees *** (ответ: $Cat(n) \times n!$; BigInteger)
  3. UVa 10223 – How Many Nodes? (предпросчитайте ответы, поскольку есть только 19 чисел Каталана $< 2^{32} - 1$)
  4. UVa 10303 – How Many Trees (сгенерируйте $Cat(n)$, как показано в этом разделе, используйте Java BigInteger)
  5. ***UVa 10312 – Expression Bracketing **** (число двойных скобок может быть подсчитано с помощью $Cat(n)$; общее количество скобок может быть вычислено с использованием *суперчисел Каталана* (см. Super Catalan Numbers))
  6. *UVa 10643 – Facing Problems With…* ($Cat(n)$ – подзадача общей задачи)

- Другие задачи, более простые
  1. UVa 11115 – Uncle Jack (N^D, используйте Java BigInteger)
  2. **UVa 11310 – Delivery Debacle *** (требуется DP: пусть `dp[i]` – количество способов упаковки тортов для коробки `2×i`. Обратите внимание, что можно использовать два «L-образных» торта для получения формы 2×3)
  3. **UVa 11401 – Triangle Counting *** (определите закономерность, написание кода легко)
  4. UVa 11480 – Jimmy's Balls (попробуйте все r, но существует более простая формула)
  5. **UVa 11597 – Spanning Subtree *** (используйте знания теории графов, ответ тривиален)
  6. UVa 11609 – Teams ($N \times 2^{N-1}$, используйте Java BigInteger для вычисления modPow)

7. *UVa 12463 – Little Nephew* (учитывайте парность носков и обуви, чтобы облегчить решение задачи)

- Другие задачи, более сложные
  1. *UVa 01224 – Tile Code* (вывести формулу из небольших тестовых сценариев)
  2. UVa 10079 – Pizza Cutting (получим однострочную формулу)
  3. UVa 10359 – Tiling (выведите формулу, используйте Java BigInteger)
  4. UVa 10733 – The Colored Cubes (лемма Бернсайда)
  5. **UVa 10784 – Diagonal *** (количество диагоналей в многоугольнике с n вершинами = $n*(n-3)/2$, используйте эту формулу для решения задачи)
  6. UVa 10790 – How Many Points of… (используйте формулу арифметической прогрессии)
  7. UVa 10918 – Tri Tiling (здесь есть два связанных друг с другом повторения)
  8. **UVa 11069 – A Graph Problem *** (используйте динамическое программирование)
  9. UVa 11204 – Musical Instruments (имеет значение только первый выбор)
  10. *UVa 11270 – Tiling Dominoes* (последовательность A004003 в OEIS)
  11. ***UVa 11538 – Chess Queen **** (считать по горизонталям, вертикалям и диагоналям)
  12. UVa 11554 – Hapless Hedonism (решается аналогично UVa 11401)
  13. *UVa 12022 – Ordering T-shirts* (количество способов, которыми n спортсменов могут оцениваться на соревнованиях, с учетом возможных связей, см. http://oeis.org/A000670)

Известные авторы алгоритмов

Леонардо Фибоначчи (также известный как Леонардо Пизанский) (1170–1250) был итальянским математиком. Он опубликовал книгу под названием *Liber Abaci* (Книга абака, или Книга о счете), в которой обсуждалась задача роста популяции кроликов на основе идеализированных предположений. Решением задачи была последовательность чисел, теперь известных как числа Фибоначчи.

Эдуард Цекендорф (1901–1983) был бельгийским математиком. Он наиболее известен своей работой над числами Фибоначчи и, в частности, доказательством теоремы Цекендорфа.

Жак Филипп Мари Бине (1786–1856) был французским математиком. Он внес значительный вклад в теорию чисел. Формула Бине, выражающая в явном виде числа Фибоначчи, названа в его честь, хотя эта формула была известна ранее.

Блез Паскаль (1623–1662) был французским математиком. Одним из его знаменитых изобретений, обсуждаемых в этой книге, является треугольник биномиальных коэффициентов Паскаля.

Эжен Шарль Каталан (1814–1894) был французским и бельгийским математиком. Именно он ввел числа Каталана для решения задачи из области комбинаторики.

Эратосфен Киренский (≈ 300–200 лет до нашей эры) был греческим математиком. Он изобрел географию, измерил окружность земли и изобрел простой алгоритм для генерации простых чисел, которые мы обсудим в этой книге.

Леонард Эйлер (1707–1783) был швейцарским математиком. Его изобретения, упомянутые в этой книге, – это функция Эйлера («фи») и путь/цикл Эйлера (в теории графов).

Кристиан Гольдбах (1690–1764) был немецким математиком. Сегодня его помнят благодаря гипотезе Гольдбаха, которую он широко обсуждал с Леонардом Эйлером.

Диофант Александрийский (≈ 200–300 гг. н. э.) был греческим математиком родом из Александрии. Он много занимался алгеброй. Одна из его работ – линейные диофантовы уравнения.

5.5. Теория чисел

Очень важно освоить как можно большее количество тем в области теории чисел, поскольку некоторые математические задачи становятся легкими (или более легкими), если вы знаете теорию, лежащую в их основе. В противном случае ваше решение не принесет вам удачи: либо использование простого перебора приведет вас к превышению лимита времени (вердикту жюри TLE), либо вы просто не сможете работать со входными данными большого размера без предварительной обработки.

5.5.1. Простые числа

Натуральное число, не меньшее 2: {2, 3, 4, 5, 6, 7, …}, – рассматривается как *простое* число, если оно делится только на 1 или на само себя. Первое и единственное четное простое число равно 2. Следующие простые числа: 3, 5, 7, 11, 13, 17, 19, 23, 29, …; простых чисел бесконечно много (доказательство в [56]). В диапазоне чисел [0…100] существует 25 простых чисел, в диапазоне [0…1000] – 168 простых чисел, в диапазоне [0…7919] – 1000 простых чисел, в диапазоне [0…10 000] – 1229 простых чисел и т. д. Несколько примеров больших простых чисел[1]: 104 729, 1 299 709, 15 485 863, 179 424 673, 2 147 483 647, 32 416 190 071, 112 272 535 095 293, 48 112 959 837 082 048 697 и т. д.

Простые числа являются важной темой в теории чисел и используются во многих задачах программирования[2]. В этом разделе мы обсудим алгоритмы с простыми числами.

---

[1] Список больших случайных простых чисел может пригодиться для тестирования, так как это числа, которые трудны для алгоритмов, таких как проверка, является ли число простым, или алгоритмы факторизации.

[2] В реальной жизни в криптографии используются большие простые числа, потому что трудно разложить число xy на множители $x \times y$, когда оба они взаимно простые.

Оптимизированная функция проверки, является ли число простым

Первый алгоритм, представленный в этом разделе, предназначен для проверки, является ли заданное натуральное число N простым, то есть `bool isPrime(N)`. Наиболее наивной версией этой проверки является проверка, использующая определение простого числа, то есть проверка, делится ли N на число $\in$ `[2..N-1]` без остатка. Такая проверка работает, но время ее выполнения составляет $O(N)$ – с точки зрения количества операций деления. Это не лучший способ с точки зрения производительности, и можно выполнить несколько улучшений данного алгоритма.

Первое значительное улучшение: необходимо проверить, делится ли N на какое-либо число $\in$ `[2..`$\sqrt{N}$`]`, то есть мы останавливаемся, когда *делитель больше, чем* $\sqrt{N}$. Обоснование: если N делится на d, то $N = d \times N/d$. Если N/d меньше, чем d, то раньше уже нашлось бы такое число N/d (или простой множитель числа N/d), на которое N делилось бы без остатка. Следовательно, *d и N/d не могут быть больше, чем* $\sqrt{N}$*, одновременно*. Этот улучшенный вариант имеет временную сложность $O(\sqrt{N})$, что уже намного быстрее, чем в предыдущем случае; однако производительность проверки, является ли заданное число простым, все еще может быть улучшена. Следующее улучшение ускорит алгоритм вдвое.

Второе улучшение: мы будем проверять, делится ли N на число $\in$ [3, 5, 7, .., $\sqrt{N}$], т. е. проверим нечетные числа только до $\sqrt{N}$. Это связано с тем, что существует только одно четное простое число (число 2), которое можно проверить отдельно. Этот вариант имеет временную сложность $O(\sqrt{N}/2)$, что равняется $O(\sqrt{N})$.

Третье улучшение[1], которого уже достаточно, чтобы использовать его[2] для решения олимпиадных задач, заключается в проверке делимости N на *простые делители* $\leqslant \sqrt{N}$. Это объясняется следующим образом: если N не делится без остатка на простое число X, то нет смысла проверять, делится ли N на числа, кратные X, или же нет. Этот способ работает быстрее, чем $O(\sqrt{N})$, его временная сложность составляет примерно $O(\#primes \leqslant \sqrt{N})$. Например, в интервале чисел [1...$\sqrt{10^6}$] есть 500 нечетных чисел, но в этом же интервале только 168 простых. Теорема о простых числах [56] гласит, что число простых чисел, меньших или равных M, обозначаемых через $\pi(M)$, ограничено $O(M/(\ln(M) - 1))$. Следовательно, сложность этой функции проверки, является ли заданное число простым, составляет около $O(\sqrt{N}/\ln(\sqrt{N}))$. Код, реализующий эту проверку, приведен ниже.

Решето Эратосфена: составление списка простых чисел

Если мы хотим сгенерировать список простых чисел в диапазоне [0...N], существует лучший алгоритм, чем проверка каждого числа из данного диапазона, является оно простым числом или нет. Алгоритм, который называется «решето Эратосфена», изобретен Эратосфеном Александрийским.

---

[1] Это похоже на рекурсию – проверка, является ли число простым числом, с использованием другого (меньшего) простого числа. Но в следующем разделе объясняется причина, почему подобный метод проверки эффективен.

[2] См. также раздел 5.3.2, где рассказывается о методе Миллера–Рабина – вероятностной функции проверки чисел на принадлежность ко множеству простых чисел – с использованием Java BigInteger.

В самом начале этот алгоритм «решета» устанавливает для всех чисел в заданном диапазоне атрибуты «вероятно, простое», но для чисел 0 и 1 он устанавливает значение атрибута «не простое». Затем он берет 2 (как простое число) и вычеркивает в заданном диапазоне все числа, кратные 2[1], начиная с $2 \times 2 = 4$, 6, 8, 10, …, до тех пор, пока не найдется число, кратное 2, которое будет больше N. Далее алгоритм берет следующее невычеркнутое число 3 (как простое число) и вычеркивает все числа, кратные 3, начиная с $3 \times 3 = 9$, 12, 15, … Затем он берет 5 и вычеркивает все кратные 5, начиная с $5 \times 5 = 25$, 30, 35, … и т. д. После этого все числа, которые остались невычеркнутыми в диапазоне $[0…N]$, являются простыми числами. Этот алгоритм выполняет приблизительно ($N \times (1/2 + 1/3 + 1/5 + 1/7 + … + 1/$последнее простое число в диапазоне $\leqslant N$)) операций. Используя формулу «суммы величин, обратных значениям простых чисел до n», мы получаем сложность по времени примерно $O(N \log \log N)$.

Поскольку создание списка простых чисел $\leqslant 10K$ с использованием «решета» выполняется быстро (наш код, приведенный ниже, может работать с числами до 10^7, удовлетворяя принятым на олимпиадах ограничениям по времени), мы решили оставить «решето» для малых значений простых чисел и предназначить оптимизированную функцию проверки, является ли число простым, для проверки больших чисел – см. предыдущее обсуждение. Код выглядит следующим образом:

```
#include <bitset>        // компактная библиотека STL для "решета" лучше, чем vector<bool>!
ll _sieve_size;                    // ll определена следующим образом: typedef long long ll;
bitset<10000010> bs;                  // 10^7 должно быть достаточно для большинства случаев
vi primes;                             // компактный список простых чисел в виде vector<int>

void sieve(ll upperbound) {                      // создаем список простых чисел в диапазоне
                                                                          // [0…верхняя граница]
  _sieve_size = upperbound + 1;    // добавляем 1, чтобы включить верхнюю границу диапазона
  bs.set();                                                     // устанавливаем все биты в 1
  bs[0] = bs[1] = 0;                                          // за исключением индексов 0 и 1
  for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
    // вычеркиваем кратные i, начиная с i * i!
    for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
    primes.push_back((int)i);          // добавляем это простое число в список простых чисел
} }                                                             // вызываем этот метод в main

bool isPrime(ll N) {  // достаточно хороший детерминированный способ проверки простых чисел
  if (N <= _sieve_size) return bs[N];              // O(1) для малых значений простых чисел
  for (int i = 0; i < (int)primes.size(); i++)
    if (N % primes[i] == 0) return false;
  return true;              // это занимает больше времени, если N – большое простое число.
}                     // Примечание: работает только для N <= (последнее простое число в vi
                                                                          // "простых чисел")^2

// внутри int main()
  sieve(10000000);      // может достигать 10^7 (для вычисления требуется несколько секунд)
```

[1] Обычная реализация должна начинаться с $2 \times i$, а не $i \times i$, но разница не так уж велика.

```
printf("%d\n", isPrime(2147483647));                    // 10-разрядное простое число
printf("%d\n", isPrime(136117223861LL));              // непростое число, 104729*1299709
```

Файл исходного кода: *ch5_06_primes.cpp/java*

5.5.2. Наибольший общий делитель и наименьшее общее кратное

Наибольший общий делитель (Greatest Common Divisor, GCD) двух целых чисел: a, b, обозначаемый $gcd(a, b)$, является наибольшим положительным целым числом d таким, что $d|a$ и $d|b$ (где $x|y$ означает, что y делится на x без остатка). Примеры наибольших общих делителей: $gcd(4, 8) = 4$, $gcd(6, 9) = 3$, $gcd(20, 12) = 4$. Одним из практических применений GCD является упрощение дробей (см. UVa 10814 в разделе 5.3.2), например: $6/9 = (6/gcd(6,9))/(9/gcd\ (6,9)) = (6/3)/(9/3) = 2/3$.

Найти GCD для двух целых чисел – простая задача, в которой используется алгоритм Евклида – эффективный алгоритм, использующий стратегию «разделяй и властвуй» [56, 7], который может быть реализован в виде однострочного кода (см. ниже). Таким образом, поиск GCD для двух целых чисел, как правило, не является основной проблемой в олимпиадной задаче по программированию, связанной с математикой, а является лишь частью общего решения.

GCD тесно связан с наименьшим общим кратным (Least Common Multiple, LCM). *Наименьшее общее кратное* двух целых чисел (a, b), обозначаемое $lcm(a, b)$, определяется как наименьшее положительное целое число l, такое что $a|l$ и $b|l$. Примеры LCM: $lcm(4, 8) = 8$, $lcm(6, 9) = 18$, $lcm(20, 12) = 60$. Было показано (см. [56]), что: $lcm(a, b) = a \times b/gcd(a, b)$. Поиск LCM также может быть реализован в виде однострочного кода (см. ниже).

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
int lcm(int a, int b) { return a * (b / gcd(a, b)); }
```

GCD более двух чисел, например $gcd(a, b, c)$, равен $gcd(a, gcd(b, c))$ и т. д., и аналогичное правило существует для LCM. Оба алгоритма GCD и LCM имеют сложность по времени $O(\log_{10} n)$, где $n = \max(a, b)$.

Упражнение 5.5.2.1. Формула для LCM имеет вид $lcm(a, b) = a \times b/gcd(a, b)$, но почему вместо нее мы используем $a \times (b/\text{gcd}(a, b))$? Подсказка: попробуйте взять $a = 1\,000\,000\,000$ и $b = 8$, используя 32-разрядные знаковые целые числа.

5.5.3. Факториал

Факториал n, то есть $n!$ или $fac(n)$, определяется как 1, если $n = 0$, и $n \times fac(n - 1)$, если $n > 0$. Однако обычно удобнее работать с итерационным вариантом формулы, то есть $fac(n) = 2 \times 3 \times \ldots \times (n - 1) \times n$ (цикл от 2 до n, пропускаем 1). Значение функции $fac(n)$ растет очень быстро. Мы все еще можем использовать `long long` в C/C++ (`long` в Java) для вычисления факториалов чисел вплоть до $fac(20)$. Кроме того, нам может потребоваться использовать библиотеку Java BigInteger для

точных, но медленных вычислений (см. раздел 5.3), работать с простыми множителями факториала (см. раздел 5.5.5) или получить промежуточные и окончательные результаты деления по модулю меньшего числа (см. раздел 5.5.8).

5.5.4. Нахождение простых множителей с помощью оптимизированных операций пробных разложений на множители

Из теории чисел мы знаем, что простое число N имеет только два простых множителя – 1 и само себя, – но составное число N, то есть число, не являющееся простым, может быть однозначно выражено как произведение его простых множителей. То есть простые числа являются мультипликативными «строительными блоками» целых чисел (основная теорема арифметики). Например, $N = 1200 = 2 \times 2 \times 2 \times 2 \times 3 \times 5 \times 5 = 2^4 \times 3 \times 5^2$ (последняя форма называется разложением на множители, равные степени простого числа, или факторизацией по степеням простых чисел).

Наивный алгоритм генерирует список простых чисел (например, используя алгоритм «решета») и проверяет, на какие простые числа целое число N делится без остатка (при этом во время работы алгоритма число N не меняется). Этот алгоритм можно улучшить.

Улучшенный алгоритм использует подход «разделяй и властвуй». Целое число N может быть выражено как $N = PF \times N'$, где PF – простой множитель, а N' – другое число, которое представляет собой N/PF, т. е. мы можем уменьшить N, «убрав» его простой множитель PF. Мы можем продолжать делать это до тех пор, пока в конце концов не придем к значению $N = 1$. Чтобы еще больше ускорить процесс, мы используем свойство делимости чисел, согласно которому у любого числа не существует делителя больше, чем $\sqrt{N}$, поэтому мы повторяем процесс поиска простых множителей до тех пор, пока $PF \leqslant \sqrt{N}$. Остановка алгоритма на $\sqrt{N}$ представляет собой особый случай: если (текущее значение $PF)^2 > N$ и при этом N все еще не равно 1, то N является последним простым множителем. Код, приведенный ниже, принимает на вход целое число N и возвращает список простых множителей.

В худшем случае – когда N простое – этот простой алгоритм факторизации с пробным разложением требует проверки всех меньших простых чисел вплоть до $\sqrt{N}$; математически это можно выразить как $O(\pi(\sqrt{N})) = O(\sqrt{N}/\ln\sqrt{N})$ – см. пример разложения большого составного числа 136 117 223 861 на два больших простых множителя 104 729 × 1 299 709 в коде, приведенном ниже. Однако если даны составные числа со множеством небольших простых множителей, этот алгоритм работает достаточно быстро – например, факторизация числа 142 391 208 960, разложением которого является $2^{10} \times 3^4 \times 5 \times 7^4 \times 11 \times 13$.

```
vi primeFactors(ll N) {                // помните: vi - это vector<int>, ll - это long long
  vi factors;
  ll PF_idx = 0, PF = primes[PF_idx];                    // простые числа были заполнены
                                                         // с использованием "решета"
  while (PF * PF <= N) {               // останавливаемся на sqrt(N); N может стать меньше
    while (N % PF == 0) { N /= PF; factors.push_back(PF); }                  // убираем PF
```

```
    PF = primes[++PF_idx];                          // рассматриваем только простые числа!
  }
  if (N != 1) factors.push_back(N);              // частный случай, если N - простое число
  return factors;             // если N не укладывается в 32-битное целое число и является
                                                                          // простым числом
}                                          // тогда 'factors' придется заменить на vector<ll>
// внутри int main(), предполагая, что sieve(1000000) было вызвано ранее
  vi r = primeFactors(2147483647);                          // самое медленное вычисление,
                                                          // 2 147 483 647 является простым
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("> %d\n", *i);

  r = primeFactors(136117223861LL);                      // медленное, 104 729 * 1 299 709
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("# %d\n", *i);

  r = primeFactors(142391208960LL);                 // более быстрое, 2^10*3^4*5*7^4*11*13
  for (vi::iterator i = r.begin(); i != r.end(); i++) printf("! %d\n", *i);
```

Упражнение 5.5.4.1. Изучите приведенный выше код. Какое значение(я) N может вызвать ошибку этого кода? Предполагается, что `vi 'primes'` содержит список простых чисел с наибольшим простым числом 9 999 991 (чуть меньше 10 млн).

Упражнение 5.5.4.2. Джон Поллард изобрел лучший алгоритм для целочисленной факторизации. Изучите и реализуйте алгоритм Полларда (как оригинальный, так и усовершенствованную версию, предложенную Ричардом П. Брентом), см. [52, 3].

5.5.5. Работа с простыми множителями

Помимо использования «медленной» функции Java BigInteger (см. раздел 5.3), мы можем выполнять точные *промежуточные вычисления* для больших целых чисел, оперируя разложением действительных целых чисел на *простые множители*, а не самими этими числами. Следовательно, для некоторых нетривиальных задач, использующих теорию чисел, мы должны работать с простыми множителями целых чисел, подаваемых на вход, даже если наша главная цель заключается совсем не в поиске простых чисел. В конце концов, простые множители являются «строительными блоками» целых чисел. Давайте рассмотрим пример: UVa 10139 – Factovisors.

Краткая формулировка условий задачи UVa 10139: «Является ли число m множителем $n!$ (т. е. делится ли $n!$ на m)? ($0 \leqslant n, m \leqslant 2^{31}-1$)». В предыдущем разделе 5.5.3 мы упоминали, что при использовании встроенных типов данных самый большой факториал, который мы можем точно вычислить, равен 20!. В разделе 5.3 мы показываем, что можем выполнять вычисления с большими целыми числами с помощью функции Java BigInteger. Тем не менее точное вычисление значений $n!$ для больших n выполняется очень медленно. Эта проблема решается с помощью нахождения простых множителей обоих чисел – $n!$ и m. Мы разложим m на простые множители и посмотрим, содержатся ли эти простые множители в $n!$. Например, при $n = 6$ у нас есть 6! – число, которое выражается в виде произведения простых чисел и их степеней: $6! = 2 \times 3 \times 4 \times 5 \times 6 =$

$2 \times 3 \times (2^2) \times 5 \times (2 \times 3) = 2^4 \times 3^2 \times 5$. Число $m_1 = 9 = 3^2$ входит в состав простых множителей 6! – вы можете убедиться, что 3^2 находится среди простых множителей числа 6!; таким образом, число 6! делится на $m_1 = 9$. Однако $m_2 = 27 = 3^3$ не содержится среди простых множителей n! – обратите внимание, что наибольшая степень числа 3 при факторизации числа 6! – только 3^2; таким образом, число 6! *не делится* на $m_2 = 27$.

Упражнение 5.5.5.1. Определите GCD и LCM для чисел ($2^6 \times 3^3 \times 97^1$, $2^5 \times 5^2 \times 11^2$).

5.5.6. Функции, использующие простые множители

Существуют и другие известные функции, относящиеся к теории чисел, которые используют простые множители. Информация о них приведена далее в этом разделе. Все они имеют одинаковую временную сложность при разложении на простые множители с помощью метода пробного разложения, приведенного выше. Заинтересованные читатели могут изучить дополнительные материалы (например, см. [56], главу 7 «Мультипликативные функции»).

1. `numPF(N)`: подсчет числа простых множителей `N`
   Простая настройка алгоритма пробного разложения для нахождения простых множителей, приведенного выше.

```
ll numPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    while (N % PF == 0) { N /= PF; ans++; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}
```

2. `numDiffPF(N)`: подсчет количества *различных* простых множителей `N`
3. `sumPF(N)`: сумма простых множителей `N`
4. `numDiv(N)`: подсчет количества *делителей* `N`
   Делитель целого числа N определяется как целое число, на которое число N делится без остатка.
   Если число $N = a^i \times b^j \times \ldots \times c^k$, то у числа N имеется $(i + 1) \times (j + 1) \times \ldots \times (k + 1)$ делителей.
   Например: $N = 60 = 22 \times 31 \times 51$ имеет $(2 + 1) \times (1 + 1) \times (1 + 1) = 3 \times 2 \times 2 = 12$ делителей. Вот эти 12 делителей: {1, **2**, **3**, 4, **5**, 6, 10, 12, 15, 20, 30, 60}. Простые множители выделены в числе этих 12 делителей **подчеркиванием**. Заметьте, что число делителей N больше, чем число его простых множителей.

```
ll numDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;                  // начнем с ans = 1
  while (PF * PF <= N) {
    ll power = 0;                                               // сосчитаем степень
```

```
    while (N % PF == 0) { N /= PF; power++; }
    ans *= (power + 1);                                  // в соответствии с формулой
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= 2;                              // (у последнего множителя pow = 1,
                                                     // добавляем 1 к этому значению)
  return ans;
}
```

5. `sumDiv(N)`: делителей N

   В предыдущем примере мы показали, что $N = 60$ имеет 12 делителей. Сумма этих делителей составляет 168.

   Эта сумма также может быть получена с помощью простых множителей. Если число $N = a^i \times b^j \times \ldots \times c^k$, то сумма делителей N равна $((a^{i+1} - 1)/(a - 1)) \times ((b^{j+1} - 1)/(b - 1)) \times \ldots \times ((c^{k+1} - 1)/(c - 1))$. Давайте попробуем вычислить это значение. $N = 60 = 22 \times 31 \times 51$, `sumDiv(60)` $= (2^{2+1} - 1)/(2-1) \times ((3^{1+1} - 1)/(3-1)) \times ((5^{1+1} - 1)/(5-1)) = (7 \times 8 \times 24)/(1 \times 2 \times 4) = 168$.

```
ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;                  // начнем с ans = 1
  while (PF * PF <= N) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);          // последнее
  return ans;
}
```

6. `EulerPhi(N)`: подсчет количества натуральных чисел < N, которые являются взаимно простыми по отношению к N.

   Напомним: два целых числа a и b называются взаимно простыми, если $gcd(a, b) = 1$; например 25 и 42. Наивный алгоритм для подсчета количества натуральных чисел < N, которые взаимно просты с N, начинает со значения `counter = 0`, далее проходит цикл по $i \in [1..N-1]$ и увеличивает счетчик, если $gcd(i, N) = 1$. Это работает слишком медленно для больших N.

   Лучшим вариантом решения этой задачи является функция Эйлера «фи» (тотиент) $\phi(N) = N \times \prod_{PF}(1 - (1/PF))$, где PF – простой множитель N.

   Например, $N = 36 = 2^2 \times 3^2$. $\phi(36) = 36 \times (1 - (1/2)) \times (1 - (1/3) = 12$. Это те самые 12 целых положительных чисел, которые являются взаимно простыми по отношению к 36: {1, 5, 7, 11, 13, 17, 19 , 23, 25, 29, 31, 35}.

```
ll EulerPhi(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = N;                  // начинаем с ans = N
  while (PF * PF <= N) {
    if (N % PF == 0) ans -= ans / PF;              // подсчитываем только уникальные
                                                               // значения множителей
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
```

```
    }
   if (N != 1) ans -= ans / N;                                  // последний множитель
    return ans;
  }
```

Упражнение 5.5.6.1. Реализуйте `numDiffPF(N)` и `sumPF(N)`.
Подсказка: обе функции похожи на `numPF(N)`.

5.5.7. Модифицированное «решето»

Если число различных простых множителей должно быть определено для *нескольких* (или *ряда*) целых чисел, то есть лучшее решение, чем многократный вызов `numDiffPF(N)`, как показано в разделе 5.5.6 выше. Лучшим решением является модифицированный алгоритм «решета». Вместо того чтобы находить простые множители и затем вычислять требуемые значения, мы начинаем с простых чисел и изменяем значения их кратных. Краткий модифицированный код «решета» приведен ниже:

```
memset(numDiffPF, 0, sizeof numDiffPF);
for (int i = 2; i < MAX_N; i++)
  if (numDiffPF[i] == 0)                                        // i - простое число
    for (int j = i; j < MAX_N; j += i)
      numDiffPF[j]++;                                  // увеличиваем значения кратных i
```

Использовать этот модифицированный алгоритм «решета» предпочтительнее, нежели отдельные вызовы `numDiffPF(N)`, если задан широкий диапазон чисел. Однако если нам просто нужно вычислить количество различных простых множителей для одного большого целого числа *N*, может быть быстрее просто использовать `numDiffPF(N)`.

Упражнение 5.5.7.1. Функция `EulerPhi(N)`, о которой говорилось в разделе 5.5.6, также может быть переписана таким образом, что для вычисления ее значений будет использоваться модифицированное «решето». Напишите код, реализующий этот способ.

Упражнение 5.5.7.2*. Можем ли мы написать код, который будет использовать модифицированное «решето» для вычисления других функций, перечисленных в разделе 5.5.6 выше (т. е. кроме `numDiffPF(N)` и `EulerPhi(N)`), не увеличивая временную сложность «решета»? Если да, напишите код, реализующий этот способ. Если нет, объясните, почему это невозможно.

5.5.8. Арифметические операции по модулю

Некоторые математические вычисления в задачах программирования могут в конечном итоге иметь своим результатом очень большие положительные (или очень малые отрицательные) промежуточные или конечные результаты, выходящие за пределы диапазона самого большого встроенного целочислен-

ного типа данных (в настоящее время это 64-разрядное `long long` в C ++ или `long` в Java). В разделе 5.3 мы показали способ точных вычислений, оперирующих с большими целыми числами. В разделе 5.5.5 мы показали другой способ работы с большими целыми числами с использованием его простых множителей. Для некоторых других задач нас интересует только результат деления по модулю – (обычно простое) число, чтобы промежуточные или конечные результаты всегда «укладывались» в диапазон встроенных целочисленных типов данных. В этом подразделе мы обсудим эти виды задач.

Например, в задаче UVa 10176 – Ocean Deep! Make it shallow!! нас просят преобразовать длинное двоичное число (до 100 цифр) в десятичное. Быстрые вычисления показывают, что наибольшее возможное число – $2^{100} - 1$, что выходит за пределы диапазона 64-разрядных целых чисел. Тем не менее в задаче спрашивается, делится ли результат на 131 071 (простое число). Итак, то, что нам нужно сделать, – это преобразовать двоичное число в десятичное цифру за цифрой, выполняя операцию взятия по модулю 131 071 для получения промежуточного результата. Если конечный результат равен 0, то *фактическое двоичное число* (которое мы никогда не вычисляем полностью) делится на 131 071.

Упражнение 5.5.8.1. Какие утверждения верны? Примечание: «%» является символом деления по модулю.

1. `(a + b - c) % s = ((a % s) + (b % s) - (c % s) + s) % s`
2. `(a * b) % s = (a % s) * (b % s)`
3. `(a * b) % s = ((a % s) * (b % s)) % s`
4. `(a / b) % s = ((a % s) / (b % s)) % s`
5. `(ab) % s = ((ab/2 % s) * (ab/2 % s)) % s`; предположим, что b – четное.

5.5.9. Расширенный алгоритм Евклида: решение линейного диофантова уравнения

Задача: предположим, домохозяйка покупает яблоки и апельсины, уплачивая в итоге 8.39 сингапурского доллара. Яблоко стоит 25 центов. Апельсин стоит 18 центов. Сколько фруктов каждого вида она покупает?

Для решения этой задачи можно составить линейное уравнение с двумя переменными: $25x + 18y = 839$. Поскольку мы знаем, что x и y должны быть целыми числами, это линейное уравнение называется линейным диофантовым уравнением. Мы можем решить линейное диофантово уравнение с двумя переменными, даже если у нас есть только одно уравнение! Вот алгоритм решения.

Пусть a и b – целые числа и $d = gcd(a, b)$. Уравнение $ax + by = c$ не имеет целочисленных решений, если не выполняется условие $d|c$. Но если $d|c$, то существует бесконечно много целочисленных решений. Первое решение (x_0, y_0) может быть найдено с использованием расширенного алгоритма Евклида, приведенного ниже, а остальное можно получить из $x = x_0 + (b/d)n$, $y = y_0 - (a/d)n$, где n – целое число. Олимпиадные задачи по программированию обычно имеют дополнительные ограничения, чтобы сделать результат конечным (и уникальным).

```
// храним x, y и d как глобальные переменные
void extendedEuclid(int a, int b) {
  if (b == 0) { x = 1; y = 0; d = a; return; }                    // основной случай
  extendedEuclid(b, a % b);                                  // аналогично оригинальному gcd
  int x1 = y;
  int y1 = x - (a / b) * y;
  x = x1;
  y = y1;
}
```

Используя функцию `extendedEuclid`, мы можем решить нашу задачу, приведенную выше.

Линейное диофантово уравнение с двумя переменными: $25x + 18y = 839$.

$a = 25$, $b = 18$

`extendedEuclid(25, 18)` дает нам $x = -5$, $y = 7$, $d = 1$; или $25 \times (-5) + 18 \times 7 = 1$.

Умножим левую и правую части уравнения, приведенного выше, на $839/gcd(25, 18) = 839$:

$25 \times (-4195) + 18 \times 5873 = 839$.

Таким образом, $x = -4195 + (18/1)n$ и $y = 5873 - (25/1)n$.

Поскольку нам нужно иметь неотрицательные x и y (неотрицательное количество яблок и апельсинов), у нас есть еще два дополнительных ограничения:

$-4195 + 18n \geqslant 0$ и $5873 - 25n \geqslant 0$, или

$4195/18 \leqslant n \leqslant 5873/25$, или

$233.05 \leqslant n \leqslant 234.92$.

Единственное возможное целое число для n теперь составляет только 234. Таким образом, единственным решением является $x = -4195 + 18 \times 234 = 17$ и $y = 5873 - 25 \times 234 = 23$, то есть 17 яблок (по 25 центов каждое) и 23 апельсина (по 18 центов каждый) в общей сложности составляют 8,39 сингапурского доллара.

Замечания о задачах из теории чисел на олимпиадах по программированию

Есть множество других задач, связанных с теорией чисел, которые мы не можем подробно рассматривать в этой книге. Исходя из нашего опыта, задачи из теории чисел часто появляются на олимпиадах ICPC, особенно в Азии. Поэтому для команды будет хорошей стратегией, если один из ее членов специально изучит теорию чисел по списку литературы, приведенному в этой книге, однако не ограничиваясь только этим списком.

Задачи по программированию, связанные с теорией чисел

- Простые числа
  1. UVa 00406 – Prime Cuts («решето», возьмите средние)
  2. **UVa 00543 – Goldbach's Conjecture *** («решето»; полный поиск; гипотеза Христиана Гольдбаха (дополненная Леонардом Эйлером): каждое четное число $\geqslant 4$ может быть выражено как сумма двух простых чисел)

3. UVa 00686 – Goldbach's Conjecture (II) (задача решается аналогично UVa 543)
4. UVa 00897 – Anagrammatic Primes («решето»; просто нужно проверить перестановки цифр)
5. UVa 00914 – Jumping Champion («решето»; обратите особое внимание на случаи L и $U < 2$)
6. **UVa 10140 – Prime Distance** * («решето», последовательный просмотр данных)
7. UVa 10168 – Summation of Four Primes (возвратная рекурсия и сокращение лишних вариантов)
8. UVa 10311 – Goldbach and Euler (анализ случая, перебор, см. UVa 543)
9. **UVa 10394 – Twin Primes** * («решето»; проверьте, являются ли p и $p + 2$ простыми числами; если да, то они являются близнецами; предварительно вычислите результат).
10. UVa 10490 – Mr. Azad and his Son (специальная задача; предварительно вычислите результаты для включения их в ответ)
11. UVa 10650 – Determinate Prime («решето»; найти три последовательных простых числа, находящихся на равном расстоянии друг от друга)
12. **UVa 10852 – Less Prime** («решето»; $p = 1$, найти первое простое число $\geqslant n/2 + 1$)
13. UVa 10948 – The Primary Problem (гипотеза Гольдбаха, см. UVa 543)
14. UVa 11752 – Super Powers (попробуйте основание: от 2 до $\sqrt[4]{2^{64}}$, степень составного числа, сортировка)

- Наибольший общий делитель и наименьшее общее кратное

1. UVa 00106 – Fermat vs. Pythagoras (перебор; используйте GCD, чтобы получить взаимно простые тройки)
2. UVa 00332 – Rational Numbers from… (используйте GCD для сокращения дроби)
3. UVa 00408 – Uniform Generator (задача поиска цикла с более простым решением; это хороший вариант, если `step < mod` и `GCD(step, mod) == 1`)
4. UVa 00412 – Pi (перебор GCD, чтобы найти элементы без общего делителя)
5. **UVa 10407 – Simple Division** * (вычтите из элементов множества s $s[0]$, найдите gcd)
6. **UVa 10892 – LCM Cardinality** * (количество пар делителей N: (m, n) таких, что $gcd(m, n) = 1$)
7. UVa 11388 – GCD LCM (понять связь между GCD и LCM)
8. UVa 11417 – GCD (перебор, размер входных данных небольшой)
9. *UVa 11774 – Doom's Day* (найдите закономерность, используя gcd на небольшом объеме с тестовыми примерами)
10. **UVa 11827 – Maximum GCD** * (GCD многих чисел, малый размер входных данных)
11. *UVa 12068 – Harmonic Mean* (применяется дробь; используйте LCM и GCD)

- Факториал
  1. **UVa 00324 – Factorial Frequencies *** (считайте цифры $n!$ до 366!)
  2. UVa 00568 – Just the Facts (можно использовать Java BigInteger, работает медленно, но все же получает вердикт жюри «AC»)
  3. **UVa 00623 – 500 (factorial) *** (легко решается с помощью Java BigInteger)
  4. UVa 10220 – I Love Big Numbers (используйте Java BigInteger; предварительные вычисления)
  5. UVa 10323 – Factorial. You Must… (переполнение: n>13/-odd n; отрицательное переполнение: n<8/-even n; PS: фактически факториал отрицательного числа не определен)
  6. **UVa 10338 – Mischievous Children *** (используйте long long, чтобы хранить числа до 20! включительно)
- Нахождение простых множителей
  1. **UVa 00516 – Prime Land *** (задача, связанная с факторизацией – разложением на множители, являющиеся степенями простых чисел)
  2. **UVa 00583 – Prime Factors *** (основная теорема арифметики)
  3. UVa 10392 – Factoring Large Numbers (перечислите простые числа, являющиеся делителями чисел из входных данных)
  4. **UVa 11466 – Largest Prime Divisor *** (используйте эффективную реализацию «решета» для получения наибольших простых множителей разложения)
- Работа с простыми множителями
  1. UVa 00160 – Factors and Factorials (предварительные вычисления: подсчет результатов разложения на простые множители числа 100! (будет < 100))
  2. UVa 00993 – Product of digits (найдите делители от 9 до 1)
  3. UVa 10061 – How many zeros & how… (в десятичном формате, «10» с одним нулем обусловлено множителями 2×5)
  4. **UVa 10139 – Factovisors *** (обсуждается в этом разделе)
  5. UVa 10484 – Divisibility of Factors (простые множители факториала)
  6. UVa 10527 – Persistent Numbers (задача решается аналогично UVa 993)
  7. UVa 10622 – Perfect P-th Power (получить GCD всех простых степеней, особый случай, если x будет отрицательным)
  8. **UVa 10680 – LCM *** (используйте простые коэффициенты `[1..N]`, чтобы получить LCM(1, 2, …, N)).
  9. UVa 10780 – Again Prime? No time (задача, похожая на UVa 10139, но все же отличающаяся от нее)
  10. UVa 10791 – Minimum Sum LCM (проанализировать простые множители N)
  11. UVa 11347 – Multifactorials (разложение по степеням простых множителей; `numDiv(N)`)
  12. UVa 11395 – Sigma Function (подсказка: квадратное число, умноженное на степени двух, то есть $2^k \times i^2$ для $k \geqslant 0$, $i \geqslant 1$, имеет нечетную сумму делителей)

13. **UVa 11889 – Benefit** * (LCM с разложением по степеням простых множителей)

- Функции, использующие простые множители
  1. **UVa 00294 – Divisors** * (`numDiv(N)`)
  2. UVa 00884 – Factorial Factors (`numPF(N)`; предварительные вычисления)
  3. UVa 01246 – Find Terrorists (LA 4340, Амрита'08, `numDiv(N)`)
  4. **UVa 10179 – Irreducible Basic…** * (функция `EulerPhi(N)`)
  5. UVa 10299 – Relatives (`EulerPhi(N)`)
  6. UVa 10820 – Send A Table (`a[i] = a[i - 1] + 2 * EulerPhi(i)`)
  7. *UVa 10958 – How Many Solutions?* (`2 * numDiv(n * m * p * p) - 1`)
  8. UVa 11064 – Number Theory (`N - EulerPhi(N) - numDiv(N)`)
  9. UVa 11086 – Composite Prime (найдите числа *N*, для которых `numPF(N) == 2`)
  10. UVa 11226 – Reaching the fix-point (`sumPF(N)`; получить длину; DP)
  11. *UVa 11353 – A Different kind of Sorting* (`numPF(N)`; модифицированная сортировка)
  12. **UVa 11728 – Alternate Task** * (`sumDiv(N)`)
  13. *UVa 12005 – Find Solutions* (`numDiv(4N-3)`)

- Модифицированное решето
  1. **UVa 10699 – Count the Factors** * (`numDiffPF(N)` для диапазона *N*)
  2. **UVa 10738 – Riemann vs. Mertens** * (`numDiffPF(N)` для диапазона *N*)
  3. ***UVa 10990 – Another New Function*** * (модифицированное «решето» для вычисления диапазона значений функции Эйлера («фи»); используйте DP для вычисления значений глубины «фи»; и наконец, применяйте DP для вычисления максимальной суммы диапазона 1D для вывода ответа)
  4. UVa 11327 – Enumerating Rational… (предварительные вычисления: `EulerPhi(N)`)
  5. *UVa 12043 – Divisors* (`sumDiv(N)` и `numDiv(N)`; «метод грубой силы»)

- Арифметические операции по модулю
  1. UVa 00128 – Software CRC ($(a \times b)mods = ((amods) * (bmods))mods$)
  2. **UVa 00374 – Big Mod** * (решается с помощью Java BigInteger modPow; или напишите свой собственный код, см. раздел 9.21)
  3. UVa 10127 – Ones (отсутствие множителей 2 и 5 означает, что нет конечного нуля)
  4. UVa 10174 – Couple-Bachelor-Spinster… (номера чисел – «старых дев» (Spinster) отсутствуют в выходном файле)
  5. **UVa 10176 – Ocean Deep; Make it…** * (обсуждается в этом разделе)
  6. **UVa 10212 – The Last Non-zero Digit** * (существует решение с использованием арифметики по модулю: умножьте числа от *N* до $N - M + 1$; многократно используйте /10, чтобы отбросить конечный ноль (нули), а затем выполните операцию %(1 млрд), чтобы запомнить только последние несколько (максимум 9) ненулевых цифр)

7. UVa 10489 – Boxes of Chocolates (сохраняйте рабочие значения небольшими с помощью арифметики по модулю)
8. *UVa 11029 – Leading and Trailing* (комбинация логарифмического приема для получения первых трех цифр и приема «big mod» для получения трех последних цифр)

- Расширенный алгоритм Евклида
  1. **UVa 10090 – Marbles *** (используйте метод решения линейного диофантова уравнения)
  2. **UVa 10104 – Euclid Problem *** (чистое применение расширенного алгоритма Евклида)
  3. UVa 10633 – Rare Easy Problem (эту задачу можно решить с помощью линейного диофантова уравнения; пусть $C = N - M$ (заданные входные данные), $N = 10a + b$ (N состоит не менее чем из двух цифр, последняя цифра b)), и $M = a$; теперь эта задача становится задачей поиска решения линейного диофантова уравнения: $9a + b = C$)
  4. **UVa 10673 – Play with Floor and Ceil *** (применение расширенного алгоритма Евклида)

- Другие задачи теории чисел
  1. UVa 00547 – DDF (задача на нахождение самой длинной последовательности неповторяющихся чисел)
  2. UVa 00756 – Biorhythms (китайская теорема об остатке)
  3. **UVa 10110 – Light, more light *** (проверьте, является ли n квадратом целого числа)
  4. UVa 10922 – 2 the 9s (проверка делимости на 9)
  5. UVa 10929 – You can say 11 (проверка делимости на 11)
  6. UVa 11042 – Complex, difficult and… (анализ случая; только четыре возможных варианта выходных данных)
  7. **UVa 11344 – The Huge One *** (прочитайте M как строку, используйте свойства делимости для [1…12])
  8. **UVa 11371 – Number Theory for… *** (дана стратегия решения)

Известные авторы алгоритмов

Джон Поллард (род. 1941) – британский математик, который изобрел алгоритмы для факторизации больших чисел (ро-алгоритм Полларда) и для вычисления дискретных логарифмов (не обсуждаемых в этой книге).

Ричард Пирс Брент (род. 1946) – австралийский математик и специалист в области теории вычислительных систем. Его исследовательские интересы: теория чисел (в частности, факторизация чисел), генераторы случайных чисел, архитектура вычислительных систем и анализ алгоритмов. Он является изобретателем или соизобретателем различных математических алгоритмов. В этой книге мы обсудим алгоритм нахождения цикла Брента (см. **упражнение 5.7.1***) и улучшенный ро-алгоритм Полларда, предложенный Брентом (см. **упражнение 5.5.4.2*** и раздел 9.26).

5.6. Теория вероятностей

Теория вероятностей – раздел математики, занимающийся анализом случайных явлений. Хотя такое событие, как отдельное (честное) подбрасывание монеты, является случайным, последовательность случайных событий будет демонстрировать определенные статистические закономерности, если событие повторяется много раз. Эти закономерности можно изучать и прогнозировать. Вероятность выпадения «решки» равна 1/2 (аналогично вероятности выпадения «орла»). Поэтому если мы (честно) подбрасываем монету n раз, то ожидаем, что увидим «решку» $n/2$ раз. Далее мы перечислим основные способы, которыми на олимпиадах по программированию могут быть решены задачи, связанные с теорией вероятности.

- Формула, полученная аналитическим путем. Для решения этих задач нужно вывести искомую формулу (обычно временная сложность таких формул составляет $O(1)$). Например, давайте обсудим, как получить решение задачи UVa 10491 – Cows and Cars, которая формулируется как описание игры, основанной на американском телешоу: «Проблема Монти Холла»[1].

  Вам сообщают число дверей, за которыми находятся коровы (NCOWS), число дверей, за которыми находятся автомобили (NCARS), и количество дверей (за которыми находятся коровы) NSHOW, которые открывает вам ведущий. Теперь вам нужно определить вероятность выигрыша автомобиля, предполагая, что вы всегда будете изменять свой выбор, выбирая другую неоткрытую дверь.

  Первый шаг к решению задачи – понять, что есть два способа получить автомобиль. Либо вы сначала выбираете дверь, за которой находится корова, а затем, изменив свой выбор, откроете дверь, за которой будет находиться автомобиль, либо сначала выбираете дверь, за которой находится автомобиль, и затем выбираете другую дверь, за которой находится другой автомобиль.

  Вероятность каждого случая можно вычислить, как показано ниже. В первом случае вероятность выбрать дверь, за которой находится корова, при первом «ходе» игрока составляет (NCOWS / (NCOWS+NCARS)). Тогда вероятность выбора другой двери, за которой находится автомобиль, равна (NCARS / (NCARS+NCOWS–NSHOW–1)). Перемножим эти два значения, чтобы получить вероятность для первого случая. Значение –1 означает выбор двери, которую вы уже выбирали, так как вы не можете выбрать ее повторно.

  Вероятность для второго случая можно вычислить аналогичным образом. Вероятность выбрать дверь, за которой находится автомобиль, при

[1] Это забавная головоломка, связанная с теорией вероятности. Читателям, которые ранее не сталкивались с этой задачей, рекомендуется найти в интернете и прочитать про нее. В исходной задаче NCOWS = 2, NCARS = 1 и NSHOW = 1. Вероятность выигрыша для случая, если вы остановитесь на первоначальным выборе, составляет 1/3, а вероятность выигрыша, если вы выберете другую еще не открытую дверь, составляет 2/3, и поэтому всегда выгодно выбирать другую дверь.

первом «ходе» игрока (NCARS / (NCARS+NCOWS)). Тогда вероятность выбора другой двери, за которой также находится автомобиль, равна ((NCARS–1) / (NCARS+NCOWS–NSHOW–1)). Так же как и в предыдущем случае, –1 означает выбор двери, за которой находится автомобиль и которую вы уже выбрали.

Просуммируйте значения вероятностей этих двух случаев, чтобы получить окончательный ответ.

- Исследование пространства поиска (выборки) для подсчета количества событий (обычный подсчет событий достаточно непрост; для этого мы можем использовать знания комбинаторики – см. раздел 5.4, полный перебор – см. раздел 3.2 или динамическое программирование – см. раздел 3.5) и вычисления числа событий в пространстве счетной выборки (обычно его намного проще посчитать). Примеры:
  - UVa 12024 – Hats – это задача про n людей, которые сдали свои n шляп в гардеробе во время некоторого мероприятия. Когда мероприятие заканчивается, эти n человек забирают свои шляпы. Некоторые берут чужую шляпу. Вычислите вероятность события, когда все берут чужие шляпы.

    Эту задачу можно решить с помощью перебора и предварительного расчета: переберите все $n!$ перестановок и посмотрите, сколько раз требуемые события появляются в $n!$ вариантах (вы можете использовать «метод грубой силы», потому что n в этой задаче невелико ($n \leqslant 12$)). Однако более математически подкованный участник может вместо этого использовать эту формулу, вычисляемую через динамическое программирование: $A_n = (n - 1) \times (A_{n-1} + A_{n-2})$;
  - UVa 10759 – Dice Throwing, краткое описание условий задачи: мы бросаем n обычных игральных костей. Какова вероятность того, что сумма значений, выпавших на всех брошенных кубиках, равна по крайней мере x?

    (Ограничения: $1 \leqslant n \leqslant 24$, $0 \leqslant x < 150$.)

    Пространство выборки (знаменатель значения вероятности) очень просто вычислить. Это 6^n.

    Число событий вычислить немного сложнее. Мы будем использовать (простой) метод DP, потому что здесь имеется много пересекающихся подзадач. Состояние будет описываться параметрами (*dice_left*, *score*), где *dice_left* хранит оставшееся число бросков костей, которые мы все еще можем сделать (начиная с n), а *score* подсчитывает накопленный счет (начиная с 0). DP использовать можно, поскольку для этой задачи существует только $24 \times (24 \times 6) = 3456$ различных состояний.

    Когда *dice_left* = 0, мы возвращаем 1 (событие), если *score* $\geqslant x$, в противном случае возвращаем 0; когда *dice_left* > 0, мы пытаемся бросить еще одну кость. Результат v для этой кости может быть одним из шести значений, и мы переходим в состояние (*dice_left* – 1, *score* + v). Мы суммируем все события.

    Последнее требование заключается в том, что мы должны использовать gcd (см. раздел 5.5.2), чтобы упростить дробь, характеризующую вероятность. В некоторых других задачах нас могут попросить вывес-

ти корректное значение вероятности события с определенным числом цифр после десятичной запятой.

Задачи по программированию, связанные с теорией вероятностей

1. *UVa 00542 – France '98* (использование подхода «разделяй и властвуй»)
2. UVa 10056 – What is the Probability? (получить аналитическую формулу)
3. *UVa 10218 – Let's Dance* (вероятности и биномиальные коэффициенты)
4. UVa 10238 – Throw the Dice (аналогично UVa 10759; используйте Java BigInteger)
5. UVa 10328 – Coin Toss (DP, 1D-состояние, Java BigInteger)
6. **UVa 10491 – Cows and Cars *** (обсуждается в этом разделе)
7. **UVa 10759 – Dice Throwing *** (обсуждается в этом разделе)
8. *UVa 10777 – God, Save me* (ожидаемое значение)
9. *UVa 11021 – Tribbles* (вероятность)
10. ***UVa 11176 – Winning Streak **** (DP, s: (n_left, max_streak), где n_left – количество оставшихся игр, а max_streak хранит максимальное количество выигрышей подряд; t: вероятность проиграть эту игру или выиграть следующие W = [1...n_left] игр и проиграть (W + 1)-ю игру; особый случай, если W = n_left)
11. UVa 11181 – Probability (bar) Given (итеративный подход, «метод грубой силы», переберите все возможности)
12. *UVa 11346 – Probability* (немного геометрии)
13. UVa 11500 – Vampires (задача о разорении игрока)
14. UVa 11628 – Another lottery (p[i] = ticket[i] / суммарная величина; используйте gcd для упрощения дроби)
15. UVa 12024 – Hats (обсуждается в этом разделе)
16. *UVa 12114 – Bachelor Arithmetic* (простая вероятность)
17. *UVa 12457 – Tennis contest* (простая задача с ожидаемой ценностью; используйте DP)
18. *UVa 12461 – Airplane* («метод грубой силы», используйте малое n, чтобы убедиться, что ответ очень прост)

5.7. Поиск цикла

Для данной функции $f : S \to S$ (которая отображает натуральное число из конечного множества S на другое натуральное число из того же конечного множества S) и начального значения $x_0 \in N$ последовательность **значений итерированной функции** $\{x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots x_i = f(x_{i-1}), \ldots\}$ должна в конечном итоге включать в себя одно и то же значение дважды, т. е. $\exists i \neq j$ такое, что $x_i = x_j$. После того как это произойдет ($x_i = x_j$), эта последовательность должна повторить цикл значений от x_i до x_{j-1}. Пусть μ (начало цикла) будет наименьшим индексом i, а λ (длина цикла) – наименьшим натуральным числом, таким, что $x_\mu = x_{\mu+\lambda}$. Задача поиска цикла определяется как задача нахождения μ и λ, если заданы $f(x)$ и x_0.

Например, в задаче UVa 00350 – Pseudo-Random Numbers нам дана функция генератора псевдослучайных чисел $f(x) = (Z \times x + I)\%M$ и $x_0 = L$, и мы хотим выяснить длину последовательной цепочки неповторяющихся чисел – длину последовательности сгенерированных чисел, перед тем как числа в ней начнут повторяться (то есть λ). Хороший генератор псевдослучайных чисел должен иметь большое значение λ, в противном случае сгенерированные числа не будут выглядеть «случайными».

Давайте рассмотрим эту задачу, используя тестовый пример, в котором $Z = 7, I = 5, M = 12, L = 4$, таким образом, $f(x) = (7 \times x + 5)\%12$ и $x_0 = 4$. Последовательность значений итерированной функции: {4, 9, 8, 1, 0, 5, 4, 9, 8, 1, 0, 5, ...}. Мы имеем $\mu = 0$ и $\lambda = 6$ при $x_0 = x_\mu + \lambda = x_{0+6} = x_6 = 4$. Последовательность значений итерированной функции циклически повторяется начиная с индекса 6 и далее.

Для другого тестового случая $Z = 3$, $I = 1$, $M = 4$, $L = 7$, мы имеем $f(x) = (3 \times x + 1)\%4$ и $x_0 = 7$. Последовательность значений итерированной функции для этого случая: {7 , 2, 3, 2, 3, ...}. На этот раз мы имеем $\mu = 1$ и $\lambda = 2$.

5.7.1. Решение(я), использующее(ие) эффективные структуры данных

Простой алгоритм, который будет работать для *многих вариантов* этой задачи на поиск цикла, использует эффективную структуру данных для хранения пары параметров с информацией о том, что число x_i встречалось на итерации i в последовательности значений итерированных функций. Затем для x_j, которое встречается позже ($j > i$), мы проверяем, хранится ли x_j в структуре данных. Если это так, то это означает, что $x_j = x_i$, $\mu = i$, $\lambda = j - i$. Этот алгоритм имеет временную сложность $O((\mu + \lambda) \times DS\_cost)$, где *DS_cost* – это стоимость одной операции структуры данных (вставка/поиск). Для этого алгоритма потребуется как минимум $O(\mu + \lambda)$ места для хранения прошлых значений.

Для многих задач поиска циклов с довольно большим S (и, вероятно, большим $\mu + \lambda$) мы можем использовать пространство размером $O(\mu + \lambda)$ в структуре `map` в C ++ STL/Java `TreeMap` для хранения/проверки индексов итерации предыдущих значений за время $O(\log(\mu + \lambda))$. Но если нам просто нужно остановить алгоритм при обнаружении первого повторяющегося числа, мы можем вместо этого использовать `set` в C ++ STL/Java `TreeSet`.

Для других задач поиска циклов с относительно небольшим S (и, вероятно, небольшим $\mu + \lambda$) мы можем использовать таблицу прямой адресации, занимающую пространство $O(|S|)$, для хранения/проверки индексов итерации предыдущих значений за время $O(1)$. Здесь мы жертвуем пространством в памяти ради увеличения скорости выполнения.

5.7.2. Алгоритм поиска цикла, реализованный Флойдом

Существует лучший алгоритм, реализованный Флойдом и называемый алгоритмом поиска цикла, который имеет временную сложность $O(\mu + \lambda)$ и пространственную сложность $O(1)$ – намного меньше, чем простые версии, приведенные выше. Этот алгоритм также называют алгоритмом «черепахи (Ч)

и зайца (З)». Он состоит из трех частей, которые мы рассмотрим ниже на примере задачи UVa 350, условия которой обсуждались выше, для случая $Z = 3$, $I = 1$, $M = 4$, $L = 7$.

Эффективный способ обнаружить цикл: найти kλ

Заметим, что для любого $i \geqslant \mu$, $x_i = x_{i+k\lambda}$, где $k > 0$, например в табл. 5.2 $x_1 = x_{1+1\times2} = x_3 = x_{1+2\times2} = x_5 = 2$ и т. д. Если мы установим $k\lambda = i$, то получим $x_i = x_{i+i} = x_{2i}$. Алгоритм поиска цикла, предложенный Флойдом, использует это допущение.

Таблица 5.2. Часть 1: нахождение kλ, f(x) = (3×x + 1)%4, x_0 = 7

Шаг	x_0 7	x_1 2	x_2 3	x_3 2	x_4 3	x_5 2	x_6 3
Начало	ЧЗ						
1		Ч	З				
2			Ч	З			

Алгоритм обнаружения цикла, предложенный Флойдом, поддерживает два указателя, называемых «черепаха» (медленный) на x_i и «заяц» (самый быстрый, который продолжает прыгать) на x_{2i}. Первоначально оба указателя находятся в x_0. На каждом шаге алгоритма «черепаха» перемещается на один шаг вправо, а «заяц» перемещается на два шага вправо[1] в нашей заданной последовательности. Затем алгоритм сравнивает значения последовательности в этих двух указателях. Наименьшее значение индекса $i > 0$, при котором «черепаха» и «заяц» указывают на равные значения, является значением $k\lambda$ (кратным λ). Мы определим фактическое λ из $k\lambda$, используя два следующих шага. В табл. 5.2, при $i = 2$, мы имеем $x_2 = x_4 = x_{2+\underline{2}} = x_{2+k\lambda} = 3$. Итак, $k\lambda = 2$. В этом примере мы увидим ниже, что k в конечном итоге равно 1, поэтому $\lambda = 2$.

Нахождение μ

Затем мы возвращаем «зайца» обратно в x_0 и оставляем «черепаху» в ее текущем положении. Теперь мы перемещаем оба указателя вправо по одному шагу за один раз, тем самым сохраняя интервал $k\lambda$ между двумя указателями. Когда «черепаха» и «заяц» указывают на одно и то же значение, мы только что нашли первое повторение чисел последовательности длины $k\lambda$. Поскольку $k\lambda$ кратно λ, должно быть верно, что $x_\mu = x_{\mu+k\lambda}$. В первый раз, когда мы сталкиваемся с первым повторением чисел последовательности длины $k\lambda$, то получаем значение μ. Для примера, приведенного в табл. 5.3, мы находим, что $\mu = 1$.

Таблица 5.3. Часть 2: нахождение μ

Шаг	x_0 7	x_1 2	x_2 3	x_3 2	x_4 3	x_5 2	x_6 3
1	З		Ч				
2		З		Ч			

[1] Для перехода вправо на один шаг от x_i мы используем $x_i = f(x_i)$. Чтобы перейти на два шага вправо от x_i, мы используем $x_i = f(f(x_i))$.

Нахождение λ

Как только мы получим μ, мы оставим черепаху в ее текущем положении и поместим зайца рядом с ней. Теперь мы последовательно перемещаем зайца вправо. Заяц будет указывать на то же значение, на которое будет указывать и черепаха, в первый раз после λ шагов. В табл. 5.4, после того как заяц переместится один раз, $x_3 = x_{3+2} = x_5 = 2$. Итак, λ = 2.

Таблица 5.4. Часть 3: нахождение λ

Шаг	x_0 7	x_1 2	x_2 3	x_3 2	x_4 3	x_5 2	x_6 3
1				Ч	З		
2				Ч		З	

Поэтому мы выдаем ответ: μ = 1 и λ = 2 для $f(x) = (3 \times x + 1)\%4$ и $x_0 = 7$. Общая временная сложность этого алгоритма составляет $O(\mu + \lambda)$.

Реализация

Рабочая реализация этого алгоритма на C/C ++ (с комментариями) приведена ниже:

```
ii floydCycleFinding(int x0) {                    // функция int f(int x) определена ранее
  // 1-я часть: найти k*mu, скорость зайца в 2 раза больше скорости черепахи
  int tortoise = f(x0), hare = f(f(x0));                        // f(x0) - узел рядом с x0
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
  // 2-я часть: найти mu, заяц и черепаха движутся с одинаковой скоростью
  int mu = 0; hare = x0;
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu++;}
  // 3-я часть: найти lambda, заяц движется, черепаха стоит на месте
  int lambda = 1; hare = f(tortoise);
  while (tortoise != hare) { hare = f(hare); lambda++; }
  return ii(mu, lambda);
}
```

Файл исходного кода: *ch5_07_UVa350.cpp/java*

Упражнение 5.7.1*. Ричард П. Брент изобрел улучшенную версию алгоритма поиска циклов, реализованного Флойдом, который был показан выше. Изучите и реализуйте алгоритм Брента [3].

Задачи по программированию, связанные с поиском цикла

1. UVa 00202 – Repeating Decimals (расширяйте последовательность цифра за цифрой, пока цифры не начнут повторяться)
2. UVa 00275 – Expanding Fractions (аналогично UVa 202, за исключением формата выходных данных)
3. UVa 00350 – Pseudo-Random Numbers (обсуждается в этом разделе)

4. UVa 00944 – Happy Numbers (аналогично UVa 10591)
5. UVa 10162 – Last Digit (цикл после 100 шагов, используйте Java BigInteger для чтения входных данных, предварительные вычисления)
6. UVa 10515 – Powers et al (сосредоточьтесь на последней цифре)
7. UVa 10591 – Happy Number (эта последовательность «в конечном итоге периодическая»)
8. UVa 11036 – Eventually periodic… (поиск цикла, оценка `f` (в пользовательской нотации) с использованием `stack` – также см. раздел 9.27)
9. UVa 11053 – Flavius Josephus… * (нахождение цикла, ответ: $N - \lambda$)
10. UVa 11549 – Calculator Conundrum (повторять возведение в квадрат с ограниченными цифрами до тех пор, пока результирующая последовательность цифр не зациклится; то есть алгоритм обнаружения цикла, изобретенный Флойдом, используется только для обнаружения цикла, мы не используем значение μ или λ; вместо этого мы отслеживаем наибольшее значение итерированной функции, найденное до того, как встретился какой-либо цикл)
11. UVa 11634 – Generate random… * (используйте таблицу с прямой адресацией размером 10*K*, извлекайте цифры; хитрость программирования состоит в том, чтобы возвести в квадрат 4 цифры «a» и получить в результате средние 4 цифры `a = (a * a / 100) % 10000`)
12. UVa 12464 – Professor Lazy, Ph.D. (хотя n может быть очень большим, закономерность на самом деле циклическая; найдите длину цикла l и выполните операцию деления по модулю n с l)

5.8. Теория игр

Теория игр – это математическая модель стратегического положения (не обязательно игр в общем значении слова «игры»), в которых успех игрока при выборе определенного варианта зависит от выбора *других* игроков. Многие задачи программирования, связанные с теорией игр, классифицируются как игры с нулевой суммой – это математический способ сказать, что если один игрок выигрывает, то другой обязательно проигрывает. Например, игра в крестики-нолики (UVa 10111), шахматы, различные числовые игры / игры с целыми числами (например, UVa 847, 10368, 10578, 10891, 11489) и другие (UVa 10165, 10404, 11311) являются играми, в которых два игрока делают ходы поочередно (обычно идеально, т. е. выбирают наилучший ход), и в игре может быть только один победитель.

Распространенный вопрос, задаваемый в олимпиадных задачах по программированию, связанных с теорией игр, заключается в том, имеет ли выигрышный ход (преимущество в игре) игрок, делающий первый ход в конкурентной игре для двух игроков, если предположить, что оба игрока делают **идеальные ходы**, т. е. каждый игрок всегда выбирает наиболее оптимальный для него вариант.

5.8.1. Дерево решений

Одним из решений является написание рекурсивного кода для исследования *дерева решений* игры (или дерева игр). Если в задаче не имеется перекрывающихся подзадач, то для ее решения подходит возвратная рекурсия; в противном случае необходимо использовать динамическое программирование. Каждая вершина описывает текущего игрока и текущее состояние игры. Каждая вершина связана со всеми остальными вершинами, достижимыми из этой вершины в соответствии с правилами игры. Корневая вершина описывает начального игрока и начальное состояние игры. Если игровое состояние в листовой вершине является выигрышным, то это означает выигрыш для текущего игрока (и проигрыш для его соперника в игре). Во внутренней вершине текущий игрок выбирает вершину, которая обеспечивает наибольший выигрыш (или, если выигрыш невозможен, выбирайте вершину с наименьшими потерями). Такой подход называется *минимаксной стратегией*.

Например, в задаче UVa 10368 – Euclid's Game есть два игрока: Стэн (игрок 0) и Олли (игрок 1). Состояние игры – тройка целых чисел (*id*, *a*, *b*). Игрок с текущим идентификатором (т. е. делающий текущий ход) может вычесть любое положительное кратное меньшего из двух чисел, целого числа *b*, из большего из двух чисел, целого числа *a*, при условии что число, полученное в результате этого вычитания, должно быть неотрицательным. Мы постоянно должны следить за выполнением условия $a \geqslant b$. Стэн и Олли делают ходы попеременно, пока один игрок не сможет вычесть число, кратное меньшему числу, из большего, чтобы получить в результате 0, и тем самым выиграть. Первый игрок – Стэн. Дерево решений для игры с начальным состоянием $id = 0$, $a = 34$ и $b = 12$ показано ниже на рис. 5.2.

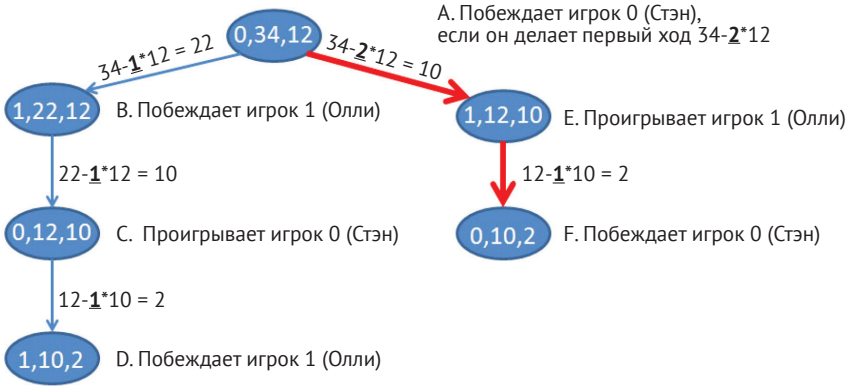


Рис. 5.2 ❖ Дерево решений для варианта «Игры Евклида»

Давайте рассмотрим, что происходит на рис. 5.2. В корне (начальное состояние) мы имеем тройку параметров (0, 34, 12). В этот момент игрок 0 (Стэн) имеет два варианта: либо взять разность $a - b = 34 - 12 = 22$ и перейти к вершине (1, 22, 12) (левая ветвь), либо вычесть $a - 2 \times b = 24 - 2 \times 12 = 10$ и перейти к вершине (1, 12, 10) (правая ветвь). Мы рекурсивно пробуем оба варианта.

Начнем с левой ветви. В вершине (1, 22, 12) (рис. 5.2.B) у текущего игрока 1 (Олли) нет другого выбора, кроме как вычесть $a - b = 22 - 12 = 10$. Теперь мы находимся в вершине (0, 12, 10) (рис. 5.2.C). Опять же, у Стэна есть только один выбор – вычитать $a - b = 12 - 10 = 2$. Теперь мы находимся в листовой вершине (1, 10, 2) (рис. 5.2.D). У Олли есть несколько вариантов, но Олли может определенно выиграть, сделав ход $a - 5 \times b = 10 - 5 \times 2 = 0$, и это означает, что вершина (0, 12, 10) является проигрышным состоянием для Стэна, и вершина (1, 22, 12) – это выигрышное состояние для Олли.

Теперь мы исследуем правую ветвь. В вершине (1, 12, 10) (рис. 5.2.E) у текущего игрока 1 (Олли) нет иного выбора, кроме как вычесть $a - b = 12 - 10 = 2$. Теперь мы находимся в конечной вершине (0, 10, 2) (рис. 5.2.F). У Стэна есть несколько вариантов, но Стэн может определенно выиграть, сделав ход $a - 5 \times b = 10 - 5 \times 2 = 0$, и это означает, что вершина (1, 12, 10) является проигрышным состоянием для Олли.

Поэтому, чтобы игрок 0 (Стэн) выиграл эту игру, Стэн должен сначала выбрать $a - 2 \times b = 34 - 2 \times 12$, так как это выигрышный ход для Стэна (рис. 5.2.A).

С точки зрения реализации, первый целочисленный идентификатор *id* в тройке параметров можно отбросить, поскольку мы знаем, что вершины с глубинами 0 (корень), 2, 4, … – всегда ходы Стэна, а вершины с глубинами 1, 3, 5, … – всегда ходы Олли. Этот целочисленный идентификатор используется на рис. 5.2 для упрощения объяснения.

5.8.2. Знание математики и ускорение решения

Не все задачи из области теории игр можно решить путем изучения всего дерева решений в игре, особенно если размер дерева велик. Если задача связана с числами, нам, возможно, придется применить некоторые математические подходы, чтобы ускорить вычисления.

Например, в задаче UVa 847 – A Multiplication Game есть два игрока: это опять Стэн (игрок 0) и Олли (игрок 1). Состояние игры[1] является целым числом p. Текущий игрок может умножить p на любое число в пределах от 2 до 9. Стэн и Олли делают ходы попеременно, пока один игрок не сможет умножить p на число от 2 до 9 так, что будет выполнено условие $p \geqslant n$ (n – целевое число), и тем самым выиграет. Первый игрок – Стэн, он начинает с $p = 1$.

На рис. 5.3 показан пример этой игры с $n = 17$. Первоначально у игрока 0 есть до 8 вариантов выбора (для умножения $p = 1$ на [2…9]). Тем не менее все эти восемь состояний являются выигрышными состояниями игрока 1, так как игрок 1 всегда может умножить текущее значение p на [2…9], чтобы получить $p \geqslant 17$ (см. рис. 5.3B). Поэтому игрок 0 наверняка проиграет (см. рис. 5.3A).

При $1 < n < 4\,294\,967\,295$ результирующее дерево решений для самого большого тестового примера может быть чрезвычайно большим. Это связано с тем, что каждая вершина в этом дереве решений имеет огромный коэффициент ветвления, равный 8 (поскольку существует возможность выбора любого из восьми чисел, от 2 до 9). Для данного случая невозможно исследовать дерево решений.

---

[1] На этот раз мы опускаем идентификатор игрока. Однако этот идентификатор все еще показан на рис. 5.3 для ясности.

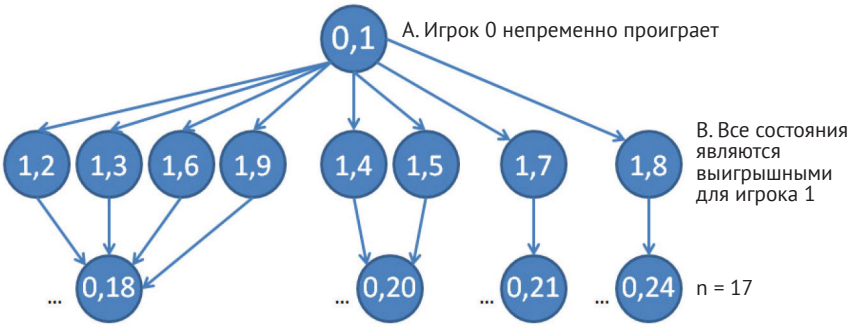


Рис. 5.3 ❖ Частичное дерево решений для варианта «Игры с умножением»

Оказывается, оптимальной стратегией победы Стэна является умножение p на 9 (максимально возможное), в то время как Олли всегда умножает p на 2 (минимально возможное). Такое понимание оптимизации может быть получено путем наблюдения закономерности, найденной в выходных данных других вариантов этой задачи гораздо меньшего объема. Обратите внимание, что опытные математики могут сначала доказать этот результат, полученный из наблюдений, прежде чем писать код, реализующий решение задачи.

5.8.3. Игра Ним

Существует специальная игра, о которой стоит упомянуть, поскольку она может появиться на олимпиаде по программированию: игра Ним[1]. В игре Ним два игрока по очереди убирают предметы из разных куч. Делая очередной ход, игрок должен удалить *хотя бы один предмет* и может удалить *любое количество предметов*, при условии что все они находятся в одной куче. Начальным состоянием игры является количество объектов n_i в каждой из k куч: $\{n_1, n_2, ..., n_k\}$. Есть хорошее решение для этой игры. Чтобы первый игрок (игрок, делающий начальный ход) выиграл, значение $n_1 \wedge n_2 \wedge ... \wedge n_k$ должно быть ненулевым (здесь $\wedge$ – битовый оператор xor (исключающее или)). Доказательство этого утверждения можно найти в статьях, посвященных теории игр.

Задачи по программированию, связанные с теорией игр

1. UVa 00847 – A Multiplication Game (смоделируйте идеальную игру, обсуждавшуюся выше)
2. **UVa 10111 – Find the Winning... *** (крестики-нолики, минимакс, возврат)
3. UVa 10165 – Stone Game (игра Ним, применение теоремы Шпрага–Гранди)
4. UVa 10368 – Euclid's Game (минимакс, возвратная рекурсия, обсуждается в этом разделе)

[1] Общие случаи игр для двух игроков входят в программу IOI [20], однако игра Ним в нее не входит.

5. UVa 10404 – Bachet's Game (игра для двух игроков, динамическое программирование)
6. UVa 10578 – The Game of 31 (возвратная рекурсия; попробуйте все варианты; посмотрите, кто победит в игре)
7. **UVa 11311 – Exclusively Edible *** (теория игр, сводимая к игре Ним; мы можем рассмотреть игру, в которую играют Гензель и Гретель, как игру Ним, где есть четыре кучи – кусочки торта слева/снизу/справа/над карамельным топингом; взять сумму Ним этих четырех значений, и если она равна 0, Гензель проигрывает).
8. **UVa 11489 – Integer Game *** (теория игр, сводимая к простой математике)
9. *UVa 12293 – Box Game* (проанализируйте игровое дерево более мелких вариантов, чтобы использовать математику для решения этой задачи)
10. *UVa 12469 – Stones* (игра, динамическое программирование, отбрасывание неподходящих вариантов)

5.9. Решения упражнений, не помеченных звездочкой

Упражнение 5.2.1. Библиотека `<cmath>` в C/C ++ имеет две функции: `log` (логарифм по основанию *e*) и `log10` (по основанию 10); однако в `Java.lang.Math` есть только `log` (логарифм по основанию *e*). Чтобы вычислить $\log_b(a)$ (логарифм по основанию *b*), мы используем тот факт, что $\log_b(a) = \log(a)/\log(b)$.

Упражнение 5.2.2. `(int)floor(1 + log10((double)a))` возвращает количество цифр в десятичном числе *a*. Чтобы посчитать количество цифр, используя функцию логарифма по другому основанию *b*, мы можем использовать аналогичную формулу: `(int)floor(1 + log10((double)a) / log10((double)b))`.

Упражнение 5.2.3. $\sqrt[n]{a}$ можно переписать как $a^{1/n}$. Затем мы можем использовать встроенную формулу, такую как `pow((double)a, 1.0 / (double)n)` or `exp(log((double)a) * 1.0 / (double)n)`.

Упражнение 5.3.1.1. Возможно; выполняйте промежуточные операции по **модулю** 10^6. Продолжайте вычленять конечные нули (после умножения $n!$ на $(n + 1)!$ не добавляется ни одного нуля или же добавляется несколько нулей).

Упражнение 5.3.1.2. Возможно. $9317 = 7 \times 11^3$. Мы также представляем 25! в виде произведения его простых множителей. Затем мы проверяем, есть ли среди них один множитель 7 (да, есть) и три множителя 11 (к сожалению, нет). Итак, 25! не делится на 9317. Альтернативный подход: использовать арифметику по модулю (см. раздел 5.5.8).

Упражнение 5.3.2.1. Для преобразования 32-разрядных целых чисел используйте `parseInt(String s, int radix)` и `toString(int i, int radix)` в классе Java **Integer** (этот способ работает быстрее). Вы также можете использовать `BufferedReader` и `BufferedWriter` для ввода/вывода данных (см. раздел 3.2.3).

Упражнение 5.4.1.1. Формула Бине для чисел Фибоначчи $fib(n) = (\phi^n - (-\phi)^{-n})/\sqrt{5}$ должна выдавать коректный результат для больших *n*. Но так как тип дан-

ных двойной точности ограничен, у нас появляются расхождения для больших n. Эта формула верна вплоть до fib(75), если реализована с использованием стандартного типа данных double в компьютерной программе. К сожалению, этого слишком мало, чтобы данный подход можно было использовать в типичных олимпиадных задачах по программированию, связанных с числами Фибоначчи.

Упражнение 5.4.2.1. $C(n, 2) = (n!)/((n-2)! \times 2!) = (n \times (n-1) \times (n-2)!)/((n-2)! \times 2) = (n \times (n-1))/2 = 0{,}5n^2 - 0{,}5n = O(n^2)$.

Упражнение 5.4.4.1. Основной принцип подсчета: если есть m способов сделать что-либо одно и n способов сделать что-то другое, то существует $m \times n$ способов сделать оба варианта. Таким образом, ответ на это упражнение: $6 \times 6 \times 2 \times 2 = 6^2 \times 2^2 = 36 \times 4 = 144$ различных возможных результата.

Упражнение 5.4.4.2. См. выше. Ответ: $9 \times 9 \times 8 = 648$. Первоначально есть 9 вариантов (1–9), затем есть еще 9 вариантов (1–9 минус 1, плюс 0), потом, наконец, есть только 8 вариантов.

Упражнение 5.4.4.3. Перестановка – это размещение элементов без повторений, где порядок элементов важен. Формула имеет вид ${}_nP_r = (n!)/((n-r)!)$, поэтому ответ на это упражнение: $6!/(6-3)! = 6 \times 5 \times 4 = 120$ трехбуквенных слов.

Упражнение 5.4.4.4. Формула для подсчета различных перестановок: $(n!)/((n_1)! \times (n_2)! \times \ldots \times (n_k)!)$ где n_i – частота каждой уникальной буквы i и $n_1 + n_2 + \ldots + n_k = n$. Ответ для этого упражнения: $(5!)/(3! \times 1! \times 1!) = 120/6 = 20$, потому что в начальном слове есть три буквы «B», 1 буква «O» и 1 буква «Y».

Упражнение 5.4.4.5. Ответы для нескольких небольших значений n = 3, 4, 5, 6, 7, 8, 9 и 10 равны 0, 1, 3, 7, 13, 22, 34 и 50 соответственно. Вы можете сгенерировать эти числа, сначала используя перебор. Затем найдите закономерность и используйте ее.

Упражнение 5.5.2.1. Следующий порядок выполнения операций – умножение $a \times b$ перед делением результата на $gcd(a, b)$ – будет иметь более высокую вероятность ошибки переполнения на олимпиаде по программированию, чем порядок выполнения операций $a \times (b/gcd(a, b))$. В приведенном примере у нас есть a = 1 000 000 000 и b = 8. LCM равен 1 000 000 000, что должно соответствовать 32-разрядным целым числам со знаком, и может быть вычислен без ошибки только с помощью применения порядка операций $a \times (b/gcd(a, b))$.

Упражнение 5.5.4.1. Поскольку наибольшее простое число в `vi 'primes'` равно 9 999 991, этот код может обрабатывать $N \leqslant 9\,999\,991^2 = 99\,999\,820\,000\,081 \approx 9 \times 10^{13}$. Если наименьший простой множитель N больше, чем 9 999 991, например $N = 1\,010\,189\,899^2 = 1\,020\,483\,632\,041\,630\,201 \approx 1 \times 10^{18}$ (это все еще в пределах 64-разрядного целого числа со знаком), этот код завершится сбоем или выдаст неправильный результат. Если мы решим отказаться от применения `vi 'primes'` и использовать PF = 3, 5, 7, … (со специальной проверкой для случая PF = 2), тогда у нас получится более медленный код, и новый верхний предел для N теперь равен N с наименьшим простым множителем в пределах до $2^{63} - 1$. Однако если заданы такие входные данные, нам нужно использовать алгоритмы, упомянутые в **упражнении 5.5.4.2*** и в разделе 9.26.

Упражнение 5.5.4.2. См. раздел 9.26.

Упражнение 5.5.5.1. GCD(A, B) можно получить, взяв наименьшую степень общих простых множителей A и B. LCM (A, B) можно получить, взяв наибольшую степень всех простых множителей A и B. Итак, GCD ($2^6 \times 3^3 \times 97^1$, $2^5 \times 5^2 \times 11^2$) = 2^5 = 32 и LCM ($2^6 \times 3^3 \times 97^1$, $2^5 \times 5^2 \times 11^2$) = $2^6 \times 3^3 \times 5^2 \times 11^2 \times 97^1$ = 507 038 400.

Упражнение 5.5.6.1.

```
ll numDiffPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    if (N % PF == 0) ans++;                                  // сосчитаем это pf один раз
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}

ll sumPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    while (N % PF == 0) { N /= PF; ans += PF; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans += N;
  return ans;
}
```

Упражнение 5.5.7.1. Модифицированный код «решета» для вычисления функции Эйлера до 106 показан ниже:

```
for (int i = 1; i <= 1000000; i++) EulerPhi[i] = i;
for (int i = 2; i <= 1000000; i++)
  if (EulerPhi[i] == i)                                      // i является простым числом
    for (int j = i; j <= 1000000; j += i)
      EulerPhi[j] = (EulerPhi[j] / i) * (i - 1);
```

Упражнение 5.5.8.1. Утверждения 2 и 4 неверны. Остальные три верны.

5.10. Примечания к главе 5

Эта глава значительно расширилась с момента выхода первого издания данной книги. Однако, даже выпустив третье издание, мы все больше осознаем, что существует еще много математических задач и алгоритмов, которые не обсуждались в этой главе, например:

- есть множество редких задач и формул комбинаторики, которые еще не обсуждались: лемма Бернсайда, числа Стирлинга и т. д.;

- существуют другие теоремы и гипотезы, которые не могут обсуждаться последовательно, одна за другой: например, функция Кармайкла, гипотеза Римана, проверка простоты числа Ферма (основанная на малой теореме Ферма), китайская теорема об остатках, теорема Шпрага–Гранди и т. д.;
- мы лишь кратко упомянули алгоритм нахождения цикла, предложенный Брентом (который немного быстрее, чем версия Флойда), в **упражнении 5.7.1***;
- (вычислительная) геометрия также является разделом математики, но поскольку для этой темы мы выделили специальную главу, то оставим обсуждение задач на геометрию для главы 7;
- позже, в главе 9, мы кратко обсудим еще несколько алгоритмов, связанных с математикой, например метод Гаусса – метод решения системы линейных уравнений (раздел 9.9), возведение матрицы в степень и его применение (раздел 9.21), ро-алгоритм Полларда (раздел 9.26), постфиксный калькулятор и преобразование выражений (раздел 9.27) и римские цифры (раздел 9.28).

Математическая тема поистине неисчерпаема. Это неудивительно, поскольку сотни лет назад люди исследовали различные математические проблемы. Некоторые из них обсуждаются в этой главе, многие другие нет, и все же на олимпиаде будет предложено решить одну или две такие задачи. Чтобы показать хорошие результаты на ICPC, неплохо иметь хотя бы одного сильного математика в вашей команде ICPC, чтобы он мог быстро решить эти математические задачи. Хорошее знание математики важно также и для участников IOI. Несмотря на то что количество тем, требующих изучения, в IOI меньше, чем в ICPC, многие задачи IOI требуют определенного «математического мастерства».

Мы заканчиваем эту главу перечислением источников информации, которые могут быть интересны некоторым читателям: прочтите книги по теории чисел, например [56], просмотрите математические темы в mathworld.wolfram.com или Википедии и, конечно, попробуйте решить множество задач по программированию, связанных с математикой, например в http://projecteuler.net [17] и https://brilliant.org [4].

Таблица 5.5. Статистические данные, относящиеся к главе 5

Параметр	Первое издание	Второе издание	Третье издание
Число страниц	17	29 (+71 %)	41 (+41 %)
Письменные упражнения	–	19	20 + 10* = 30 (+58 %)
Задачи по программированию	175	296 (+69 %)	369 (+25 %)

Распределение количества упражнений по программированию по разделам этой главы показано ниже.

Таблица 5.6. Распределение количества упражнений по программированию по разделам главы 4

Раздел	Название	Число заданий	% в главе	% в книге
5.2	Специальные математические задачи	144	39 %	9 %
5.3	Класс Java BigInteger	45	10 %	1 %
5.4	Комбинаторика	54	15 %	3 %
5.5	Теория чисел	86	23 %	5 %
5.6	Теория вероятностей	18	5 %	1 %
5.7	Поиск цикла	13	3 %	1 %
5.8	Теория игр	10	3 %	1 %

Рис. 5.4 ❖ Слева направо: Стивен, Ранальд, Хьюберт, Вэй Лян, Бернард, Зи Чун

Глава 6

Обработка строк

Геном человека содержит приблизительно 3.2 млрд пар оснований.

– Проект «Геном человека»

6.1. Обзор и мотивация

В этой главе мы представляем еще одну тему, которая в настоящее время тестируется на международных студенческих олимпиадах по программированию (ICPC), хотя и появляется не так часто[1], как математические задачи и задачи с использованием графов, – это задачи обработки строк (string processing). Задачи обработки строк весьма часто возникают при исследованиях в области биоинформатики. Поскольку строки (например, строки ДНК), с которыми работают исследователи, обычно (очень) длинные, возникает необходимость в эффективных структурах данных и алгоритмах, специально предназначенных для обработки строк. Некоторые такие задачи представлены как конкурсные задания на международных студенческих олимпиадах по программированию (ICPC). Тщательно изучив материал этой главы, участники олимпиад по программированию улучшат свои шансы на успешное решение задач обработки строк.

Задачи обработки строк также появляются на международных олимпиадах по программированию для школьников (IOI), но обычно они не требуют слишком сложных структур данных и алгоритмов из-за ограничений программы IOI[2] [20]. Кроме того, формат ввода и вывода строк в задачах олимпиад по программированию для школьников, как правило, упрощен[3]. Это исключает необходимость утомительного рутинного кодирования процедур синтаксического

---

[1] Одна из вероятных причин: при вводе строк труднее правильно выполнить синтаксический анализ (парсинг – parsing), а выводимые строки труднее правильно отформатировать, поэтому операции ввода/вывода строк менее предпочтительны, чем более точные и определенные операции ввода/вывода целых чисел.

[2] Здесь syllabus – это документ (см. https://ioinformatics.org/files/ioi-syllabus-2019.pdf), содержащий полный перечень знаний, которые могут потребоваться для решения задач международной олимпиады школьников по информатике IOI.

[3] На олимпиадах по программированию для школьников IOI 2010–2012 гг. участникам предлагалось реализовать функции вместо кодирования подпрограмм ввода/вывода.

анализа ввода и форматирования вывода, часто встречающихся в заданиях на международных студенческих олимпиадах по программированию (ICPC). Задания школьных олимпиад, требующие обработки строк, обычно остаются разрешимыми при помощи принципов и способов решения задач, рассмотренных в главе 3. Для участников IOI вполне достаточно бегло просмотреть все разделы этой главы, за исключением раздела 6.5, в котором рассматривается обработка строк с применением динамического программирования. Тем не менее мы надеемся, что этот материал в будущем может оказаться полезным для участников олимпиад по программированию для школьников при изучении некоторых более продвинутых тем за пределами школьной программы.

Эта глава имеет следующую структуру: сначала приводится обзор основных приемов и принципов обработки строк и достаточно длинный список специализированных задач обработки строк, которые можно решить с применением этих основных приемов и принципов. Несмотря на то что специализированные задачи обработки строк составляют большинство задач, рассматриваемых в этой главе, необходимо особо отметить, что на последних олимпиадах ACM ICPC (а также IOI) в заданиях обычно не требовались простые решения по обработке строк, за исключением «утешительных» задач, которые большинство команд (участников) вполне способны решить. Более важны разделы, в которых рассматриваются задачи сравнения (поиска совпадений) строк (раздел 6.4), задачи обработки строк, решаемые с применением динамического программирования (раздел 6.5), и, наконец, подробное обсуждение задач, в которых необходима обработка действительно весьма длинных строк (раздел 6.6). В последнем разделе рассматриваются эффективные структуры данных для строк: суффиксный бор (бор, луч, нагруженное дерево – suffix trie), суффиксное дерево (suffix tree) и суффиксный массив (suffix array).

6.2. Основные приемы и принципы обработки строк

Эта глава начинается с краткого рассмотрения нескольких основных приемов и принципов обработки строк, которыми обязан овладеть каждый программист, участвующий в олимпиадах по программированию. В этом разделе приводится ряд небольших задач, которые читатель должен решить поочередно, не пропуская ни одной задачи. Можно использовать любой из следующих языков программирования: C, C++ или Java. Лучше всего попытаться найти самую короткую, наиболее эффективную реализацию решения каждой задачи. Затем сравните свои реализации с нашими (см. раздел решений в конце главы). Если для вас не стала откровением любая из предложенных нами реализаций (или вы даже разработали более простую и эффективную реализацию), то вы уже вполне готовы для решения задач обработки строк различной сложности. Продолжайте изучение следующих разделов. В противном случае рекомендуем потратить некоторое время на тщательное изучение наших реализаций.

1. Дано: текстовый файл, содержащий только символы английского алфавита `[A-Za-z]`, цифры `[0-9]`, пробелы и точку («.»). Написать программу, считывающую содержимое этого файла построчно (по одной строке) до тех пор, пока не встретится строка, которая начинается с семи точек

('.......'). Объединить все считанные строки в одну длинную строку T. При объединении двух строк необходимо вставить один пробел между последним словом предыдущей строки и первым словом текущей строки. В одной строке может быть до 30 символов, а в каждом блоке ввода должно быть не более 10 строк. В конце каждой строки нет хвостовых пробелов, и каждая строка заканчивается символом перехода на новую строку (newline). Примечание: пример исходного текстового файла *ch6.txt* показан в формате исходного кода после пункта 1.d перед задачей 2.

a. Вы знаете, как сохранять строки средствами предпочитаемого вами языка программирования?
b. Как считывать заданный входной текст по одной строке?
c. Как объединить две строки в одну укрупненную строку?
d. Как проверить, что строка начинается с последовательности символов '.......', чтобы остановить процесс чтения входных данных?

```
I love CS3233 Competitive
Programming. i also love
AlGoRiThM
…….you must stop after reading this line as it starts with 7 dots
after the first input block, there will be one looooooooooong line…
```

2. Предположим, что имеется одна длинная строка T. Необходимо проверить, можно ли найти другую строку P в этой строке T. Требуется вывести все индексы (номера позиций в строке), в которых P встречается в T, или -1, если строка P не найдена в строке T. Например, если строка T = "I love CS3233 Competitive Programming. i also love AlGoRiThM", а строка P = 'I', то будет выведен только индекс {0} (при начале индексации с 0). Если считаются различными буквы в верхнем 'I' и нижнем 'i' регистрах, то символ 'i' с индексом {39} не является частью вывода. Если строка P = 'love', то выводятся индексы {2, 46}. Если строка P = 'book', то выводится {-1}.
   a. Как найти первое вхождение (искомой) подстроки в исходной строке (если оно существует)?
      Есть ли необходимость в реализации алгоритма поиска (совпадений) строки (например, алгоритма Кнута–Морриса–Пратта, рассматриваемого в разделе 6.4, и т. п.) или можно воспользоваться лишь библиотечными функциями?
   b. Как найти следующее вхождение (или несколько вхождений) (искомой) подстроки в исходной строке (если оно (они) существует)?
3. Предположим, что необходимо выполнить некоторый простой анализ символов в строке T, а также преобразовать каждый символ в строке T в нижний регистр. Сущность требуемого анализа: сколько цифр, гласных [aeiouAEIOU] и согласных (прочих букв алфавита, не являющихся гласными) содержится в строке T? Можно ли выполнить такой анализ за время $O(n)$, где n – длина (в символах) строки T?
4. Далее необходимо разделить эту длинную строку T на лексемы (tokens) (подстроки) и сохранить эти лексемы в массиве строк с именем tokens. В этой небольшой задаче разделителями лексем являются пробелы и точки (то есть выполняется разделение предложений на слова). Напри-

мер, если разделить на лексемы предлагаемую строку `T` (в которой символы уже преобразованы в нижний регистр), то получим следующий набор лексем: `tokens = {'i', 'love', 'cs3233', 'competitive', 'programming', 'i', 'also', 'love', 'algorithm'}`. Затем необходимо отсортировать этот массив строк лексикографически[1], а потом найти лексикографически наименьшую строку. То есть после сортировки массив выглядит так: `tokens = {'algorithm', 'also', 'competitive', 'cs3233', 'i', 'i', 'love', 'love', 'programming',}`. Таким образом, лексикографически наименьшей строкой в этом примере является `'algorithm'`.

   a. Как разделить строку на лексемы?
   b. Как сохранить полученные лексемы (более короткие строки) в массиве строк?
   c. Как лексикографически отсортировать массив строк?

5. Теперь необходимо определить, какое слово чаще всего встречается в строке `T`. Для ответа на этот вопрос необходимо подсчитать частоту появления каждого слова. В примере со строкой `T` выводимым ответом будет `'i'` и/или `'love'`, так как оба слова встречаются дважды. Какая структура данных должна использоваться в этой задаче?
6. В предложенном здесь текстовом файле имеется еще одна строка после строки, которая начинается с `('.......')`, но длина этой последней строки не ограничена. Ваша задача – подсчитать количество символов, содержащихся в этой последней строке. Как считать строку, если ее длина заранее неизвестна?

Файлы задач и исходного кода: *ch06_01_basic_string.html/cpp/java*

Известные авторы алгоритмов

Доналд Эрвин Кнут (Donald Ervin Knuth) (родился в 1938 г.) – ученый-информатик, почетный профессор Стэнфордского университета (Stanford University). Автор широко известной серии книг по информатике «Искусство программирования» («The Art of Computer Programming»). Кнут был назван «отцом анализа алгоритмов». Кроме того, Кнут также является автором TEX, системы компьютерной подготовки и верстки текстов, используемой для этой книги.

Джеймс Хайрем Моррис (James Hiram Morris) (родился в 1941 г.) – профессор информатики. Соавтор алгоритма Кнута–Морриса–Пратта для поиска строк.

Вон Роналд Пратт (Vaughan Ronald Pratt) (родился в 1944 г.) – почетный профессор Стэнфордского университета (Stanford University). Является одним из первопроходцев в области информатики. Внес существенный вклад в фундаментальные области информатики: алгоритмы поиска, алгоритмы сортировки и алгоритмы проверки чисел на простоту. Также является соавтором алгоритма Кнута–Морриса–Пратта для поиска строк.

---

[1] По существу, этот порядок сортировки очень похож на используемый в обычных словарях.

Сол Нидлман (Saul Needleman) и **Кристиан Д. Вунш** (Christian D. Wunsch) совместно опубликовали в 1970 году алгоритм выравнивания двух строк (последовательностей) с применением динамического программирования, который рассматривается в этой книге.

Темпл Феррис Смит (Temple Ferris Smith) – профессор биомедицинской инженерии, оказал помощь в разработке алгоритма Смита–Ватермана, предложенного Майклом Ватерманом в 1981 году. Алгоритм Смита–Ватермана служит основой для сравнения (локального выравнивания) многочисленных последовательностей, идентифицируя сегмент по схожести максимальной локальной последовательности. Этот алгоритм применяется для идентификации схожих сегментов ДНК, РНК и протеинов.

Майкл Спенсер Ватерман (Michael Spencer Waterman) – профессор университета Южной Калифорнии (University of Southern California). Ватерман является одним из основателей и действующих лидеров в области вычислительной биологии (computational biology). Его работы внесли существенный вклад в разработку наиболее широко применяемых инструментальных средств в этой области науки. В частности, алгоритм Смита–Ватермана (разработанный совместно с Т. Ф. Смитом) является основой многих программ сравнения (выравнивания) последовательностей.

6.3. Специализированные задачи обработки строк

Теперь рассмотрим не менее важную тему: специализированные задачи обработки строк. Это задачи олимпиад по программированию с использованием строк. Для решения таких задач требуются только базовые навыки программирования и, возможно, некоторые навыки практического применения простых методов обработки строк, рассмотренных выше в разделе 6.2. Необходимо лишь внимательно прочитать требования в условии задачи и написать код короткого (как правило) решения. Ниже приведен список таких специализированных задач обработки строк с краткими советами и рекомендациями по их решению. Эти задания по программированию делятся на следующие подкатегории:

- шифрование/кодирование/декодирование/расшифрование.
  Каждый пользователь желает, чтобы его личные цифровые средства обмена информацией были защищены. То есть сообщения (строки) могли бы прочитать только выбранные пользователем получатели. Для этой цели было разработано множество методов шифрования, и ряд этих методов (не самых сложных) в конечном итоге стал основой для специализированных задач обработки строк на олимпиадах по программированию, при этом в каждой задаче определены собственные правила кодирования/декодирования. Множество таких заданий содержится в репозитории онлайнового арбитра UVa [47]. Таким образом, эту категорию задач можно разделить еще на две подкатегории: более простые и более сложные версии. Попытайтесь решить хотя бы некоторые из них, особенно задачи, выделенные полужирным шрифтом и помеченные

звездочкой *, как обязательные к решению. При решении этих задач весьма полезными будут любые, даже базовые знания в области компьютерной безопасности и криптографии;

- подсчет частоты (появления символов).
  В этой группе задач участникам предлагается подсчитать частоту появления буквы (символа) (легкая версия, можно использовать таблицу прямой адресации) или слова (сложная версия, при ее решении используется либо сбалансированное дерево бинарного поиска, например структура C++ STL `map` или Java `TreeMap`, либо хеш-таблица). Некоторые из таких задач действительно связаны с криптографией (то есть с предыдущей подкатегорией);
- синтаксический анализ (парсинг) входных данных.
  Эта группа задач не предназначена для школьных олимпиад по программированию (IOI), так как в заданиях, не выходящих за рамки школьного курса, формулировки условий должны быть настолько простыми, насколько это возможно. Диапазон задач синтаксического анализа: от более простых заданий, которые могут быть решены с помощью итеративного парсера-анализатора, до более сложных, в которых используются некоторые грамматические правила, требующие применения метода рекурсивного спуска или класса Java `String/Pattern`;
- задания, разрешимые с применением класса Java `String/Pattern` (регулярное выражение).
  Некоторые (хотя и редко встречающиеся) задачи обработки строк можно решить с помощью однострочного[1] кода, в котором используются `matches(String regex)`, `replaceAll(String regex, String replacement)` и/или другие полезные функции класса Java `String`. Для получения возможности реализации такого решения необходимо в полной мере освоить концепцию регулярных выражений (regular expression – regex). Здесь мы не будем подробно рассматривать регулярные выражения, но приведем два примера их практического использования.

  1. В задании UVa 325 – Identifying Legal Pascal Real Constants предлагается определить, является ли заданная строка ввода допустимой (корректной) константой типа real языка Pascal. Предположим, что строка сохранена в объекте `String s`, тогда следующая строка кода Java представляет требуемое решение:

     ```
     s.matches("[-+]?\\d+(\\.\\d+([eE][-+]?\\d+)?|[eE][-+]?\\d+)")
     ```

  2. В задании UVa 494 – Kindergarten Counting Game предлагается подсчитать, сколько слов содержится в заданной строке. Здесь слово определяется как непрерывная последовательность букв (в верхнем и/или нижнем регистре). Предположим, что строка сохранена в объекте `String s`, тогда следующая строка кода Java представляет требуемое решение:

     ```
     s.replaceAll("[^a-zA-Z]+", " ").trim().split(" ").length
     ```

[1] Эти задачи можно решить и без применения регулярных выражений, но исходный код может оказаться более длинным.

- форматирование вывода.
  Это еще одна группа задач, которые не предназначены для школьных олимпиад по программированию (IOI). В этом случае проблемным становится вывод. В наборах для студенческих олимпиад по программированию (ICPC) такие задания используются как «разогревающие» или как «задачи, расходующие время» участников. Улучшайте навыки кодирования, решая подобные задачи с максимальной возможной скоростью, поскольку этот тип задач может оказать определяющее влияние на штрафное время каждой команды;
- сравнение строк.
  В этой группе задач участникам предлагается сравнить строки по разнообразным критериям. Эта подкатегория похожа на задачи поиска совпадений в строках (string matching), рассматриваемых в следующем разделе, но в задачах сравнения строк в основном используются функции типа `strcmp`;
- просто специализированные задачи обработки строк.
  Это все прочие задачи, связанные с обработкой строк, которые невозможно классифицировать как принадлежащие к одной из перечисленных выше подкатегорий.

Задания по программированию, связанные со специализированной обработкой строк

- Шифрование/кодирование/декодирование/расшифровка, более простые задания

1. UVa 00245 – Uncompress (использование заданного алгоритма)
2. UVa 00306 – Cipher (можно сделать решение более быстрым, если исключить цикл)
3. UVa 00444 – Encoder and Decoder (каждый символ отображается в две или три цифры)
4. UVa 00458 – The Decoder (сдвиг ASCII-значения каждого символа на –7)
5. UVa 00483 – Word Scramble (последовательное считывание по одному символу слева направо)
6. UVa 00492 – Pig Latin (специализированная задача, аналогична заданию UVa 483)
7. UVa 00641 – Do the Untwist (реверсировать заданную формулу и выполнить имитацию)
8. UVa 00739 – Soundex Indexing (простая задача преобразования)
9. UVa 00795 – Sandorf's Cipher (необходимо подготовить «механизм инверсного отображения»)
10. UVa 00865 – Substitution Cypher (простая подстановка / отображение символов)
11. UVa 10019 – Funny Encryption Method (несложная задача, необходимо найти шаблон)
12. UVa 10222 – Decode the Mad Man (простой механизм декодирования)

13. **UVa 10851 – 2D Hieroglyphs… *** (игнорировать границу; интерпретировать «\/» как 1/0; начать считывание снизу)
14. **UVa 10878 – Decode the Tape *** (интерпретировать пробел/«o» как 0/1, далее преобразование из двоичной в десятичную систему)
15. UVa 10896 – Known Plaintext Attack (перебор всех возможных ключей; использовать токенизатор)
16. UVa 10921 – Find the Telephone (простая задача преобразования)
17. UVa 11220 – Decoding the message (следовать инструкциям, описанным в задаче)
18. **UVa 11278 – One-Handed Typist *** (отображение раскладки клавиш QWERTY в раскладку DVORAK)
19. UVa 11541 – Decoding (последовательное считывание по одному символу и выполнение имитации)
20. UVa 11716 – Digital Fortress (простое шифрование)
21. UVa 11787 – Numeral Hieroglyphs (следовать описанию условий задачи)
22. UVa 11946 – Code Number (специализированная задача)

- Шифрование/кодирование/декодирование/расшифровка, более сложные задания

1. UVa 00213 – Message Decoding (расшифровать сообщение)
2. UVa 00468 – Key to Success (установить отображение по частоте появления букв)
3. **UVa 00554 – Caesar Cypher *** (перебор всех сдвигов; форматирование выходных данных)
4. *UVa 00632 – Compression (II)* (имитация процесса, использование сортировки)
5. *UVa 00726 – Decode* (частотное шифрование)
6. UVa 00740 – Baudot Data… (простая имитация процесса)
7. UVa 00741 – Burrows Wheeler Decoder (имитация процесса)
8. UVa 00850 – Crypt Kicker II (атака на основе открытого текста, нетривиальные тестовые варианты)
9. UVa 00856 – The Vigenère Cipher (три вложенных цикла: по одному для каждой цифры)
10. **UVa 11385 – Da Vinci Code *** (обработка строк + последовательность Фибоначчи)
11. **UVa 11697 – Playfair Cipher *** (следовать описанию условий задачи, несколько скучная тривиальная задача)

- Подсчет частоты (появления символов)

1. UVa 00499 – What's The Frequency… (использовать одномерный массив для подсчета частоты (вхождения символов))
2. UVa 00895 – Word Problem (получить частоту вхождения заданной буквы в каждом слове, сравнить со строкой головоломки)
3. **UVa 00902 – Password Search *** (последовательное считывание по одному символу; подсчет частоты слов)
4. UVa 10008 – What's Cryptanalysis? (подсчет частоты символов)

5. UVa 10062 – Tell me the frequencies (подсчет частот ASCII-символов)
6. **UVa 10252 – Common Permutation** * (подсчет частоты для каждого алфавитного символа)
7. UVa 10293 – Word Length and Frequency (простая задача)
8. UVa 10374 – Election (использовать структуру `map` для подсчета частоты)
9. UVa 10420 – List of Conquests (подсчет частоты слов, использовать структуру `map`)
10. UVa 10625 – GNU = GNU's Not Unix (суммирование частоты *n* раз)
11. UVa 10789 – Prime Frequency (проверка: является ли частота вхождения какой-либо буквы простым числом)
12. **UVa 11203 – Can you decide it…** * (описание задачи выглядит запутанным, но в действительности это простая задача)
13. UVa 11577 – Letter Frequency (простая задача)

- Синтаксический анализ (парсинг) входных данных (не рекурсивный)

1. UVa 00271 – Simply Syntax (проверка грамматики, построчный просмотр (сканирование))
2. UVa 00327 – Evaluating Simple C… (реализация может быть нетривиальной)
3. UVa 00391 – Mark-up (использовать флаги, утомительный рутинный синтаксический разбор)
4. UVa 00397 – Equation Elation (итеративное выполнение следующей операции)
5. UVa 00442 – Matrix Chain Multiplication (использовать свойства последовательного умножения матриц)
6. UVa 00486 – English-Number Translator (синтаксический разбор)
7. UVa 00537 – Artificial Intelligence? (простая формула; синтаксический разбор сложный)
8. UVa 01200 – A DP Problem (LA 2972, Tehran03, токенизация линейного уравнения)
9. **UVa 10906 – Strange Integration** * (синтаксический разбор формы Бэкуса–Наура (BNF), итеративное решение)
10. UVa 11148 – Moliu Fractions (извлечение целых чисел, простых/смешанных дробей из строки; применение алгоритма нахождения наибольшего общего делителя – см. раздел 5.5.2)
11. **UVa 11357 – Ensuring Truth** * (описание задачи выглядит несколько устрашающе – задача выполнимости булевых формул (SAT); наличие грамматики форм Бэкуса–Наура (BNF) наводит на мысль об использовании синтаксического разбора методом рекурсивного спуска; но только один элемент (clause) требует подтверждения выполнимости для получения результата TRUE; выполнимость этого элемента можно обеспечить, если для всех переменных в этом элементе их инверсные аналоги также не находятся в этом элементе; после этого мы получаем гораздо более простую задачу)
12. **UVa 11878 – Homework Checker** * (синтаксический разбор математического выражения)

13. *UVa 12543 – Longest Word* (LA6150, HatYai12, итеративный синтаксический анализатор)

- Синтаксический анализ (парсинг) входных данных (рекурсивный)
  1. UVa 00384 – Slurpys (рекурсивная проверка грамматики)
  2. UVa 00464 – Sentence/Phrase Generator (генерация выходных данных на основе заданной грамматики BNF (форма Бэкуса–Наура))
  3. UVa 00620 – Cellular Structure (рекурсивная проверка грамматики)
  4. **UVa 00622 – Grammar Evaluation** * (рекурсивная проверка/оценка грамматики BNF)
  5. UVa 00743 – The MTM Machine (рекурсивная проверка грамматики)
  6. **UVa 10854 – Number of Paths** * (рекурсивный синтаксический разбор плюс подсчет)
  7. UVa 11070 – The Good Old Times (рекурсивная грамматическая оценка)
  8. **UVa 11291 – Smeech** * (синтаксический анализ методом рекурсивного спуска)

- Задания, разрешимые с применением класса Java `String`/`Pattern` (регулярное выражение)
  1. **UVa 00325 – Identifying Legal…** * (см. приведенное выше решение на языке Java)
  2. **UVa 00494 – Kindergarten Counting…** * (см. приведенное выше решение на языке Java)
  3. UVa 00576 – Haiku Review (синтаксический разбор, грамматика)
  4. **UVa 10058 – Jimmi's Riddles** * (задание разрешимо с помощью регулярных выражений Java)

- Форматирование вывода
  1. UVa 00110 – Meta-loopless sort (в действительности это специализированная задача сортировки)
  2. UVa 00159 – Word Crosses (рутинная утомительная задача форматирования вывода)
  3. UVa 00320 – Border (требует применения метода заливки)
  4. UVa 00330 – Inventory Maintenance (использовать структуру `map` как вспомогательное средство)
  5. UVa 00338 – Long Multiplication (рутинная задача)
  6. UVa 00373 – Romulan Spelling (проверка сочетания букв «g перед p», специализированная задача)
  7. UVa 00426 – Fifth Bank of… (токенизация; сортировка; переформатирование выходных данных)
  8. UVa 00445 – Marvelous Mazes (имитация, форматирование выходных данных)
  9. **UVa 00488 – Triangle Wave** * (использовать несколько циклов)
  10. UVa 00490 – Rotating Sentences (обработка двумерного массива, форматирование выходных данных)
  11. UVa 00570 – Stats (использовать структуру `map` как вспомогательное средство)

12. UVa 00645 – File Mapping (использовать рекурсию для имитации структуры каталога, это поможет правильно отформатировать выходные данные)
13. UVa 00890 – Maze (II) (имитация, выполнение предписанных шагов, рутинная задача)
14. UVa 01219 – Team Arrangement (LA 3791, Tehran06)
15. UVa 10333 – The Tower of ASCII (задача, действительно расходующая много времени)
16. UVa 10500 – Robot Maps (симуляция, форматирование выходных данных)
17. UVa 10761 – Broken Keyboard (сложности с форматированием выходных данных; необходимо учесть, что «END» является частью входных данных)
18. **UVa 10800 – Not That Kind of Graph** * (рутинная задача)
19. UVa 10875 – Big Math (простая и рутинная задача)
20. UVa 10894 – Save Hridoy (как быстро вы сможете решить эту задачу?)
21. UVa 11074 – Draw Grid (форматирование выходных данных)
22. UVa 11482 – Building a Triangular… (рутинная задача)
23. UVa 11965 – Extra Spaces (заменить смежные пространства одним объединенным пространством)
24. **UVa 12155 – ASCII Diamondi** * (использовать корректную обработку индекса)
25. UVa 12364 – In Braille (обработка двумерного массива, проверка всех возможных цифр [0..9])

- Сравнение строк
  1. UVa 00409 – Excuses, Excuses (токенизация и сравнение со списком извинений/оправданий)
  2. **UVa 00644 – Immediate Decodability** * (использовать метод грубой силы)
  3. UVa 00671 – Spell Checker (сравнение строк)
  4. UVa 00912 – Live From Mars (имитация, поиск с заменой)
  5. **UVa 11048 – Automatic Correction…** * (гибкое сравнение строк с учетом использования словаря)
  6. **UVa 11056 – Formula 1** * (сортировка, сравнение строк без учета регистра букв)
  7. UVa 11233 – Deli Deli (сравнение строк)
  8. UVa 11713 – Abstract Names (модифицированное сравнение строк)
  9. UVa 11734 – Big Number of Teams… (модифицированное сравнение строк)

- Просто специализированные задачи обработки строк
  1. UVa 00153 – Permalex (найти формулу для этой задачи, задание аналогично заданию UVa 941)
  2. UVa 00263 – Number Chains (сортировка цифр, преобразование в целые числа, цикл проверки)
  3. UVa 00892 – Finding words (простая задача обработки строк)

4. **UVa 00941 – Permutations** * (формула для получения *n*-й перестановки)
5. UVa 01215 – String Cutting (LA 3669, Hanoi06)
6. UVa 01239 – Greatest K-Palindrome… (LA 4144, Jakarta08, метод полного перебора)
7. UVa 10115 – Automatic Editing (просто сделать, что требуется, использовать строки)
8. UVa 10126 – Zipf's Law (сортировка слов для упрощения решения этой задачи)
9. UVa 10197 – Learning Portuguese (обязательное и чрезвычайно точное выполнение указаний в описании условий задачи)
10. UVa 10361 – Automatic Poetry (считывание, токенизация, обработка в соответствии с требованиями)
11. UVa 10391 – Compound Words (это в большей степени задача правильного выбора структуры данных)
12. **UVa 10393 – The One-Handed Typist** * (следовать указаниям в описании условия задачи)
13. UVa 10508 – Word Morphing (количество слов = количество букв + 1)
14. UVa 10679 – I Love Strings (тестирование слабых элементов данных; простая проверка: является ли подстрока T префиксом строки S – насчитывается (AC), когда это не так)
15. **UVa 11452 – Dancing the Cheeky…** * (периодичность подстрок в строках, небольшой объем входных данных, BF)
16. UVa 11483 – Code Creator (очевидное решение, использование «экранирующего символа» (escape))
17. UVa 11839 – Optical Reader (некорректно/недопустимо, если помечено 0 или > 1 альтернативных вариантов)
18. UVa 11962 – DNA II (поиск формулы; аналогично заданию UVa 941; основание 4)
19. UVa 12243 – Flowers Flourish… (простая задача токенизации строк)
20. UVa 12414 – Calculating Yuan Fen (задача на применение метода грубой силы при обработке строк)

6.4. Поиск совпадений в строках

Поиск совпадений в строках (string matching) (также называемый просто поиском в строках (string searching)[1]) – это задача поиска начального индекса (или нескольких индексов) (под)строки (которую называют шаблоном (pattern) `P`) в более длинной строке (называемой текстом `T`). Пример: предположим, что имеется строка `T = 'STEVEN EVENT'`. Если `P = 'EVE'`, то ответом будет индекс 2 и ин-

[1] Мы имеем дело с такой задачей поиска совпадений в строках почти каждый раз, когда читаем/редактируем текст с помощью компьютера. Вспомните, сколько раз вы нажимали комбинацию клавиш **Ctrl**+**F** (стандартная комбинация клавиш в Windows и некоторых других ОС для вызова «функции поиска») в обычных программах обработки и редактирования текста, в веб-браузерах и т. д.

декс 7 (при индексации, начинающейся с 0). Если P = 'EVENT', то ответом будет только индекс 7. Если P = 'EVENING', то правильного ответа нет (не найдено ни одного совпадения, и в таких случаях обычно возвращается -1 или NULL).

6.4.1. Решения с использованием библиотечных функций

Для большинства простых («чистых») задач поиска совпадений в более или менее коротких строках можно воспользоваться библиотекой обработки строк для выбранного языка программирования. Это функция strstr в языке C <string.h>, find в языке C++ <string>, index0f в классе String языка Java. Рекомендуем повторно обратиться к разделу 6.2, мини-задача 2, где рассматриваются подобные решения с использованием библиотечных функций.

6.4.2. Алгоритм Кнута–Морриса–Пратта

В задании 7 раздела 1.2.3 приведено упражнение, требующее поиска всех вхождений подстроки *P* (длины *m*) в (длинной) строке *T* (длины *n*), если такие вхождения существуют. Фрагмент кода, приведенный ниже с комментариями, представляет простейшую реализацию алгоритма поиска совпадений в строках.

```
void naiveMatching()
{
  for( int i=0; i < n; i++ ) {          // перебор всех предположительно начальных индексов
    bool found = true;
    for( int j=0; j < m && found; j++ )               // используется логический флаг found
      if( i+j >= n || P[j] != T[i+j] )                      // если обнаружено несовпадение
        found = false;       // игнорировать этот символ, сдвинуть начальный индекс i на +1
    if( found )                                      // то есть если P[0..m-1] == T[i..i+m-1]
      printf( "P is found at index %d in T\n", i );
  }
}
```

Этот простейший алгоритм может в среднем выполняться за время $O(n)$, если применяется к обычному (естественному) тексту, например к абзацам этой книги, но время выполнения может стать равным $O(nm)$ при наихудшем варианте входных данных в заданиях на олимпиадах по программированию, например: T = 'AAAAAAAAAB' (десять букв 'A', затем одна буква 'B') и P = 'AAAAB'. Этот простейший алгоритм будет постоянно обнаруживать несовпадение на последнем символе шаблона P, затем проверять следующий начальный индекс, который отличается лишь на +1 от индекса предыдущей попытки. Это неэффективный способ. К сожалению, опытный автор задачи всегда включает такой наихудший тестовый вариант в свой секретный набор тестовых данных.

В 1977 году Кнут (Knuth), Моррис (Morris) и Пратт (Pratt) (отсюда и название алгоритма КМП (KMP)) разработали усовершенствованный алгоритм поиска совпадений в строках, который использует информацию, полученную по результатам сравнений предыдущих символов, особенно тех, которые совпадают. Алгоритм Кнута–Морриса–Пратта (КМП) никогда не сравнивает повторно тот символ в строке T, который совпал с символом в строке P. Тем не менее

КМП работает аналогично приведенному выше простейшему алгоритму, если первый символ шаблона (образца) P и текущий символ в строке T не совпадают. В приведенном выше примере[1], сравнение P[j] и T[i], начиная от i=0 до 13 при j=0 (первый символ шаблона P), не отличается от простейшего алгоритма.

```
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
    0123456789012
              1
    ^ the first character of P mismatch with T[i] from index i = 0 to 13
    KMP has to shift the starting index i by +1, as with naive matching.
... at i = 14 and j = 0 ...
   (^ первый символ P не совпадает с T[i], начиная с индекса i=0 до 13
    КМП должен сдвинуть начальный индекс i на +1, как и в простейшем алгоритме.
... далее по индексу i = 14 и j = 0 ...)
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                SEVENTY SEVEN
                   0123456789012
                             1
                              ^ then mismatch at index i = 25 and j = 11
                             (^ затем обнаружено несовпадение по индексу i = 25 и j = 11)
```

Здесь обнаружено 11 совпадающих символов, начиная с индекса $i = 14$ до 24, но несовпадение встречается по индексу $i = 25$ ($j = 11$). Простейший алгоритм поиска совпадений в строках неэффективно перезапустит процесс поиска с индекса $i = 15$, но алгоритм КМП может возобновить процесс поиска с индекса $i = 25$. Причина в том, что совпадающие символы перед несовпадением – это подстрока 'SEVENTY SEV'. 'SEV' (подстрока с длиной 3) выглядит и как правильный суффикс, и как правильный префикс подстроки 'SEVENTY SEV'. Эту подстроку 'SEV' также называют границей (border) подстроки 'SEVENTY SEV'. Можно без какого-либо риска пропустить индекс от $i = 14$ до 21: 'SEVENTY ' в подстроке 'SEVENTY SEV', так как здесь снова будет обнаружено несовпадение, но при этом нельзя исключать возможность того, что следующее совпадение начинается со второй подстроки 'SEV'. Поэтому алгоритм КМП возвращает j обратно к значению индекса 3, пропуская $11 - 3 = 8$ символов подстроки 'SEVENTY ' (обратите внимание на хвостовой пробел), в то время как i остается на индексе 25. Это главное отличие алгоритма КМП от рассмотренного выше простейшего алгоритма.

```
... at i = 25 and j = 3 (This makes KMP efficient) ...
(... с i = 25 и j = 3 (Это обеспечивает эффективность алгоритма КМП) ...)
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
```

[1] Предложение в строке T, приведенной ниже, предназначено только для демонстрационных целей. Оно некорректно грамматически.

```
P =                          SEVENTY SEVEN
                             0123456789012
                                      1
                                ^ then immediate mismatch at index i = 25, j = 3
                               (^ затем сразу же несовпадение по индексу i = 25, j = 3)
```

На этот раз префикс *P* перед несовпадением символов является подстрокой ‘SEV’, но отсутствует граница (border), поэтому алгоритм КМП восстанавливает значение *j* обратно к 0 (другими словами, «перезапускает» шаблон поиска совпадения *P* опять с его начального символа).

```
... mismatches from i = 25 to i = 29... then matches from i = 30 to i = 4
(... несовпадения с i = 25 до i = 29... затем совпадения с i = 30 по i = 4)
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                               SEVENTY SEVEN
                                  0123456789012
                                            1
```

Это совпадение, то есть шаблон *P* = 'SEVENTY SEVEN' найден по индексу *i* = 30. После этого алгоритм КМП знает, что 'SEVENTY SEVEN' содержит подстроку 'SEVEN' (с длиной 5) как границу (border), поэтому алгоритм переустанавливает значение *j* снова равным 5, эффективно пропуская 13 – 5 = 8 символов строки 'SEVENTY ' (особо отметим, что учитывается и хвостовой пробел), и сразу же возобновляет поиск с индекса *i* = 43 и обнаруживает следующее совпадение. Это эффективный способ.

```
... at i = 43 and j = 5, we have matches from i = 43 to i = 50 ...
So P = 'SEVENTY SEVEN' is found again at index i = 38.
(... по индексу i = 43 и j = 5 найдены совпадения с i = 43 по i = 50 ...
Значит, шаблон P = 'SEVENTY SEVEN' найден снова по индексу i = 38.)
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                                     SEVENTY SEVEN
                                        0123456789012
                                                  1
```

Чтобы обеспечить такое ускорение поиска, алгоритм КМП должен предварительно обработать строку шаблона и сформировать «таблицу возобновления поиска» (reset table) `b` (back). Если задана строка шаблона P = 'SEVENTY SEVEN', то таблица *b* будет выглядеть следующим образом:

```
                        1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3
P =  S E V E N T Y   S E V E N
b = -1 0 0 0 0 0 0 0 0 0 1 2 3 4 5
```

Это означает, что если несовпадение обнаружено по индексу `j = 11` (см. пример выше), то есть после нахождения соответствий для подстроки 'SEVENTY SEV', то становится ясно, что мы должны повторно попытаться найти совпадение `P`, начиная с индекса `j = b[11] = 3`, – алгоритм КМП теперь предпола-

гает, что было найдено совпадение только первых трех символов подстроки 'SEVENTY SEV', а именно 'SEV', поскольку следующее совпадение может начинаться с этого префикса 'SEV'. Относительно короткая реализация алгоритма Кнута–Морриса–Пратта с комментариями приведена ниже. Временная сложность этой реализации равна $O(n + m)$.

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N];                                  // T = текст, P = образец (шаблон)
int b[MAX_N], n, m;  // b = таблица возврата (back), n = длина строки T, m = длина строки P

void kmpPreprocess()            // эта функция вызывается перед вызовом функции kmpSearch()
{
  int i = 0, j = -1; b[0] = -1;                                          // начальные значения
  while (i < m) {                                  // предварительная обработка строки шаблона
    while (j >= 0 && P[i] != P[j]) j = b[j];               // несовпадение, восстановить j,
                                                                         // используя таблицу b
     i++; j++;                       // при совпадении увеличить на 1 значения обоих индексов
    b[i] = j;                 // проверить i = 8, 9, 10, 11, 12, 13 при j = 0, 1, 2, 3, 4, 5
  }                          // для примера с шаблоном P = "SEVENTY SEVEN", приведенным выше
}

void kmpSearch()                          // эта функция аналогична функции kmpPreprocess(),
                                                                  // но работает со строкой T
{
  int i = 0, j = 0;                                                      // начальные значения
  while (i < n) {                                                          // поиск в строке T
    while (j >= 0 && T[i] != P[j]) j = b[j];               // несовпадение, восстановить j,
                                                                         // используя таблицу b
    i++; j++;                       // при совпадении увеличить на 1 значения обоих индексов
    if (j == m) {                                                  // совпадение найдено при j == m
      printf("P is found at index %d in T\n", i - j);
      j = b[j];               // подготовить значение j для следующего возможного совпадения
    }
  }
}
```

Файл исходного кода: *ch6_02_kmp.cpp/java*

Упражнение 6.4.1*. Выполнить функцию `kmpPreprocess()` для шаблона `P = 'ABABA'` и вывести (показать) таблицу возврата (переустановки) `b`.

Упражнение 6.4.2*. Выполнить функцию `kmpSearch()` для шаблона `P = 'ABABA'` и строки `T = 'ACABAABABDABABA'`. Объяснить, как выглядит процесс поиска по алгоритму Кнута–Морриса–Пратта на этом примере.

6.4.3. Поиск совпадений в строках на двумерной сетке

Задачу поиска совпадений в строках можно также поместить в двумерное пространство. Пусть задана двумерная сетка/массив символов (вместо хорошо знакомого одномерного массива символов), необходимо найти все вхождения

шаблона P в этой двумерной сетке. В зависимости от требований к задаче направление поиска может быть ориентировано по четырем или восьми основным сторонам света, а кроме того, шаблон должен быть найден либо в одной непрерывной строке, либо может быть распределен между строками (часть шаблона находится в одной строке, часть – в следующей и т. д.). Рассмотрим следующий пример.

```
abcdefghigg    // Из задания UVa 10010 - Where's Waldorf?
hebkWaldork    // Можно вести поиск в 8 направлениях, требуемый шаблон
ftyawAldorm    // 'WALDORF' выделен заглавными буквами в этой сетке
ftsimrLqsrc
byoarbeDeyv    // Сможете ли вы найти шаблоны 'BAMBI' и 'BETTY'?
klcbqwikOmk
strebgadhRb    // Сможете ли вы найти шаблон 'DAGBERT' в этой строке?
yuiqlxcnbjF
```

Решением для такой задачи поиска совпадений в строках на двумерной сетке обычно является метод рекурсивного поиска с возвратами (recursive backtracking) (см. раздел 3.2.2). Причина в том, что, в отличие от поиска в одномерном массиве, где движение всегда выполняется вправо, в каждой точке (строка, столбец) двумерной сетки/массива имеется более одного варианта выбора направления поиска.

Для ускорения такого процесса поиска с возвратами обычно применяется следующая простая стратегия отсечения: как только глубина рекурсии превышает длину шаблона *P*, можно сразу же отсечь ту ветвь рекурсивного поиска. Этот метод также называют поиском с ограниченной глубиной (depth-limited search) (см. раздел 8.2.5).

Задания по программированию, связанные с поиском совпадений в строках

- Стандартные
  1. UVa 00455 – Periodic String (поиск шаблона *s* в строке *s* + *s*)
  2. *UVa 00886 – Named Extension Dialing* (преобразование первой буквы заданного имени и всех букв фамилии в цифры; затем выполнение специализированного типа поиска совпадений в строках, при котором необходимо найти совпадение с началом в префиксе строки)
  3. **UVa 10298 – Power Strings *** (поиск шаблона *s* в строке *s* + *s*, аналогично заданию UVa 455)
  4. UVa 11362 – Phone List (сортировка строк, поиск совпадений)
  5. **UVa 11475 – Extend to Palindromes *** (использование «границы» (border) алгоритма КМП)
  6. **UVa 11576 – Scrolling Sign *** (измененный поиск совпадений в строках; полный поиск)
  7. UVa 11888 – Abnormal 89's (для проверки «алиндрома» (alindrome) найти развернутый шаблон *s* в строке *s* + *s*)
  8. *UVa 12467 – Secret word* (основная идея аналогична заданию UVa 11475 – если вы решили задание UVa 11475, то сможете решить и это задание)

- На двумерной сетке (в двумерном массиве)
  1. **UVa 00422 – Word Search Wonder *** (двумерная сетка/массив, поиск с возвратами)
  2. *UVa 00604 – The Boggle Game* (двумерная матрица, поиск с возвратами, сортировка и сравнение)
  3. *UVa 00736 – Lost in Space* (двумерная сетка/массив, немного измененный метод поиска совпадений в строках)
  4. **UVa 10010 – Where's Waldorf? *** (задание рассмотрено в этом разделе)
  5. **UVa 11283 – Playing Boggle *** (двумерная сетка/массив, поиск с возвратами, запрещено считать буквы дважды)

6.5. Обработка строк с применением динамического программирования

В этом разделе рассматриваются некоторые задачи обработки строк, которые можно решить с использованием метода динамического программирования (DP – dynamic programming), описанного в разделе 3.5. Первые две задачи (выравнивание строк и поиск наибольшей общей подпоследовательности/подстроки) являются классическими задачами, которые должны знать все программисты, участвующие в олимпиадах.

Важное замечание: для различных задач динамического программирования, связанных с обработкой строк, мы обычно работаем с целочисленными индексами строк, а не с самими строками (или подстроками). Передача подстрок как параметров в рекурсивные функции строго не рекомендуется, так как это чрезвычайно замедляет процесс обработки и затрудняет мемоизацию (запоминание).

6.5.1. Регулирование строк (редакционное расстояние)

Задача выравнивания строк (редакционное расстояние[1]) определяется следующим образом: отрегулировать (выровнять)[2] две строки A и B с максимальной оценкой регулирования (или минимальным количеством операций редактирования).

---

[1] Редакционное расстояние по-другому называют расстоянием Левенштейна. Одним из наиболее значимых практических приложений этого алгоритма является функция проверки правописания, часто включаемая в широко распространенные текстовые редакторы. Если пользователь неправильно ввел слово, например «пробелма», то «умный» текстовый редактор определит, что это слово имеет очень близкое редакционное расстояние до правильного слова «проблема», и сможет автоматически исправить опечатку.

[2] Регулирование или выравнивание (align) строк – это процесс вставки пробелов в строки A и B так, чтобы обе строки содержали одинаковое количество символов. Можно рассматривать процедуру «вставки пробелов в строку B» как «удаление соответствующих выравнивающих символов в строке A».

После регулирования (выравнивания) строк A и B существует несколько возможных отношений между символами A[i] и B[i]:

1) символы A[i] и B[i] совпадают, поэтому делать ничего не надо (предположим, что это дает нам два очка);
2) символы A[i] и B[i] не совпадают, поэтому выполняется замена A[i] на B[i] (допустим, что это действие отнимает одно очко);
3) вставляется пробел в позицию A[i] (это также отнимает одно очко);
4) удаляется буква/символ из позиции A[i] (отнимает одно очко).

Например (отметим, что для обозначения пробела здесь используется специальный символ подчеркивания «_»):

```
A = 'ACAATCC' -> 'A_CAATCC'          // Пример неоптимального регулирования/выравнивания
B = 'AGCATGC' -> 'AGCATGC_'          // Проверка оптимального варианта, приведенного ниже
                 2-22--2-                    // Оценка регулирования = 4*2 + 4*-1 = 4
```

Решение методом полного перебора, при котором выполняется перебор всех возможных вариантов выравнивания, приводит к превышению лимита времени (TLE) даже при средней длине строк A и/или B. Для решения этой задачи существует алгоритм Нидлмана–Вунша (Needleman-Wunsch) (с проходом снизу вверх) с применением динамического программирования [62]. Рассмотрим две строки A[1..n] и B[1..m]. Определим $V(i, j)$ как оценку оптимального регулирования префикса A[1..i] и подстроки B[1..j], а оценку score(C1,C2) как функцию, которая возвращает оценку, если символ C1 выровнен по символу C2.

Основные варианты:

$V(0, 0) = 0$ // нет оценки для совпадения двух пустых строк
$V(i, 0) = i \times score(A[i], \_)$ // удаление подстроки A[1..i] для урегулирования, $i > 0$
$V(0, j) = j \times score(\_, B[j])$ // вставка пробелов в B[1..j] для урегулирования, $j > 0$

Рекуррентные функции: для $i > 0$ и $j > 0$:

$V(i, j) = max(вариант1, вариант2, вариант3)$, где

$вариант1 = V(i - 1, j - 1) + score(A[i], B[j])$ // оценка совпадения // или несовпадения
$вариант2 = V(i - 1, j) + score(A[i], \_)$ // удаление A[i]
$вариант3 = V(i, j - 1) + score(\_, B[j])$ // вставка B[j]

Если говорить кратко, этот алгоритм динамического программирования сосредоточен на трех возможных вариантах для последней пары символов. Возможными вариантами могут быть совпадение/несовпадение, удаление или вставка. Несмотря на то что нам неизвестно, какой из трех вариантов является наилучшим, можно попробовать все возможные варианты, избегая при этом повторного вычисления (выполнения) перекрывающихся подзадач (это один из основных принципов динамического программирования).

```
A = 'xxx...xx'              A = 'xxx...xx'          A = 'xxx...x_'
           |                           |                       |
B = 'yyy...yy'              B = 'yyy...y_'          B = 'yyy...yy'
match/mismatch              delete                  insert
(совпадение/несовпадение)   удаление                вставка
```

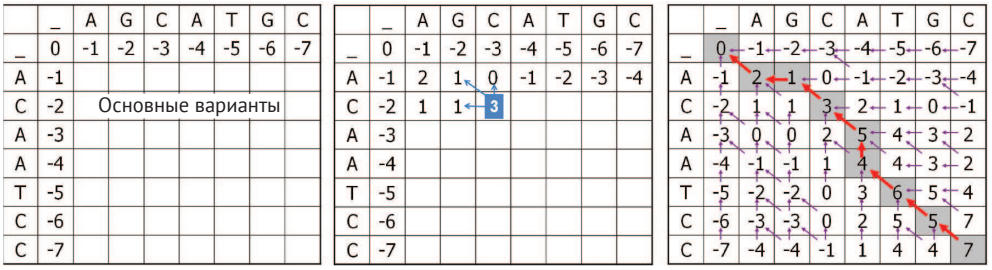


Рис. 6.1 ❖ Пример: `A = 'ACAATCC'` и `B = 'AGCATGC'` (оценка регулирования = 7)

При использовании простой функции оценки, в которой совпадение получает +2 очка, а несовпадение, а также операции вставки и удаления получают –1 очко, подробности процесса оценки регулирования/выравнивания строк `A = 'ACAATCC'` и `B = 'AGCATGC'` показаны на рис. 6.1. Изначально известны только основные варианты. Затем мы можем заполнять ячейки значениями строка за строкой слева направо. Для заполнения ячейки $V(i, j)$ при $i, j > 0$ необходимы только три других значения: $V(i-1, j-1)$, $V(i-1, j)$ и $V(i, j-1)$ – см. рис. 6.1, в середине, строка 2, столбец 3. Максимальная оценка процесса регулирования/выравнивания строк хранится в нижней правой ячейке (7 в рассматриваемом примере).

Для воспроизведения (оптимального) решения необходимо следовать по затененным (серым цветом) ячейкам, начиная с нижней правой ячейки. Решение для заданных строк `A` и `B` показано ниже. На рис. 6.1 диагональная стрелка обозначает совпадение или несовпадение (например, для последнего символа `..C`). Вертикальная стрелка обозначает удаление (например, из `..CAA..` в `..C_A..`). Горизонтальная стрелка обозначает вставку (например, от `A_C..` к `AGC..`).

```
A = 'A_CAAT[C]C'                          // Оптимальное регулирование/выравнивание
B = 'AGC_AT[G]C'                  // Оценка регулирования/выравнивания = 5*2 + 3*-1 = 7
```

Сложность по памяти (space complexity) этого алгоритма динамического программирования (с проходом снизу вверх) равна $O(nm)$, то есть размеру таблицы динамического программирования. Необходимо заполнить все ячейки этой таблицы за время $O(1)$ для каждой ячейки. Таким образом, временная сложность алгоритма равна $O(nm)$.

Файл исходного кода: *ch6_03_str_align.cpp/java*

Упражнение 6.5.1.1. Почему оценка (стоимость) совпадения установлена равной +2, а оценки (стоимости) замены, вставки и удаления установлены равными –1? Это какие-то магические числа? Будет ли успешно работать оценка +1 для совпадения? Можно ли установить другие значения для замены, вставки, удаления? Внимательно изучите алгоритм и найдите ответы на все поставленные вопросы.

Упражнение 6.5.1.2. В примере исходного кода – файл *ch6_03_str_align.cpp/java* – показана только оценка оптимального регулирования/выравнивания.

Измените этот исходный код так, чтобы он показывал действительное выравнивание.

Упражнение 6.5.1.3. Продемонстрируйте, как нужно использовать «прием с экономией пробелов» (space saving trick), описанный в разделе 3.5, для улучшения алгоритма динамического программирования Нидлмана–Вунша (с проходом снизу вверх). Какими будут сложности по памяти и по времени для этого нового решения? Каковы недостатки использования этого варианта алгоритма?

Упражнение 6.5.1.4. Задача регулирования/выравнивания строк, рассматриваемая в этом разделе, называется глобальной (обобщенной) задачей регулирования/выравнивания и выполняется за время $O(nm)$. Если в задание на олимпиаде по программированию введено ограничение только на d операций вставки или удаления, то можно получить более быстрый алгоритм. Найдите простой прием для алгоритма Нидлмана–Вунша, который позволит выполнять не более d операций вставки или удаления и будет выполняться быстрее.

Упражнение 6.5.1.5. Внимательно изучите усовершенствованный вариант алгоритма Нидлмана–Вунша (алгоритм Смита–Ватермана (Smith-Waterman) [62]) для решения локальной (local) задачи регулирования/выравнивания.

6.5.2. Поиск наибольшей общей подпоследовательности

Задача поиска наибольшей общей подпоследовательности (longest common subsequence – LCS) определяется следующим образом: заданы две строки A и B, найти самый длинный общий набор (подпоследовательность) символов, входящий в обе эти строки (символы не обязательно должны быть смежными). Например: строки `A = 'ACAATCC'` и `B = 'AGCATGC'` содержат наибольший набор символов (подпоследовательность) длиной 5: `'ACATC'`.

Задача поиска наибольшей общей подпоследовательности может быть сведена к задаче регулирования/выравнивания строк, рассмотренной в предыдущем разделе, поэтому можно использовать тот же алгоритм динамического программирования. Оценка (стоимость) для несовпадения устанавливается равной «минус бесконечности» (например, –1 млрд), оценка (стоимость) для операций вставки и удаления равна 0, а для совпадения 1. При таких условиях алгоритм Нидлмана–Вунша для регулирования/выравнивания строк никогда не рассматривает несовпадения.

Упражнение 6.5.2.1. Найти наибольшую общую подпоследовательность (LCS) для строк `A = 'apple'` и `B = 'people'`.

Упражнение 6.5.2.2. Задачу определения расстояния Хэмминга (Hamming distance), то есть задачу нахождения числа позиций, в которых соответствующие символы двух слов одинаковой длины различны, можно свести к задаче регулирования/выравнивания строк. Необходимо присвоить соответствующие оценки (стоимости) для соответствия, несоответствия, вставки и удаления,

чтобы можно было вычислить расстояние Хэмминга между двумя строками, используя алгоритм Нидлмана–Вунша.

Упражнение 6.5.2.3. Задача поиска наибольшей общей подпоследовательности может быть решена за время $O(n \log k)$, если все символы в строках различны, например если заданы две перестановки, как в задании UVa 10635. Найдите решение для этого варианта.

6.5.3. Неклассические задачи обработки строк с применением динамического программирования

UVa 11151 – Longest Palindrome

Палиндром – это строка, которая читается одинаково в обоих направлениях. Некоторые варианты задач нахождения палиндромов решаются с применением метода динамического программирования, например задание UVa 11151 – Longest Palindrome: задана строка с длиной не более $n = 1000$ символов, определить размер (в символах) самого длинного палиндрома, который можно составить из этой строки, удаляя ноль или больше символов. Примеры:

ADAM → ADA (длина 3, удалить символ «M»)
MADAM → MADAM (длина 5, без удаления символов)
NEVERODDOREVENING → NEVERODDOREVEN (длина 14, удалить символы «ING»)
RACEF1CARFAST → RACECAR (длина 7, удалить символы «F1» и «FAST»)

Решение методом динамического программирования: пусть $len(l, r)$ – длина самого длинного палиндрома, формируемого из строки `A[1..r]`.

Основные варианты:
Если $(l = r)$, то $len(l, r) = 1$. // палиндром нечетной длины
Если $(l + 1 = r)$, то $len(l, r) = 2$, если $(A[l] = A[r])$, иначе 1. // палиндром
// четной длины

Рекуррентные формулы:
Если $(A[l] = A[r])$, то $len(l, r) = 2 + len(l + 1, r - 1)$. // оба «поворотных» символа
// одинаковы
Иначе $len(l, r) = \max(len(l, r - 1), len(l + 1, r))$. // увеличение (индекса) на левой
// стороне или уменьшение на правой стороне

Решение методом динамического программирования имеет временную сложность $O(n^2)$.

Упражнение 6.5.3.1*. Можно ли воспользоваться решением задачи поиска наибольшей общей подпоследовательности, рассмотренным в разделе 6.5.2, для решения задания UVa 11151? Если можно, то как? Определить временную сложность такого решения.

Упражнение 6.5.3.2*. Предположим, что теперь необходимо найти самый длинный палиндром в заданной строке с длиной не более $n = 10\,000$ символов. В этот раз запрещено удалять какие-либо символы. Каким должно быть решение?

Задания по программированию, связанные с обработкой строк методом динамического программирования

- Классические
  1. UVa 00164 – String Computer (регулирование/выравнивание строк, редакционное расстояние)
  2. **UVa 00526 – Edit Distance *** (регулирование/выравнивание строк, редакционное расстояние)
  3. UVa 00531 – Compromise (задача поиска наибольшей общей подпоследовательности; вывод (печать) решения)
  4. UVa 01207 – AGTC (LA 3170, Manila06, классическая задача редактирования строк)
  5. UVa 10066 – The Twin Towers (задача поиска наибольшей общей подпоследовательности, но не для «строки»)
  6. UVa 10100 – Longest Match (задача поиска наибольшей общей подпоследовательности)
  7. **UVa 10192 – Vacation *** (задача поиска наибольшей общей подпоследовательности)
  8. UVa 10405 – Longest Common… (задача поиска наибольшей общей подпоследовательности)
  9. **UVa 10635 – Prince and Princess *** (задача поиска наибольшей общей подпоследовательности в двух перестановках)
  10. UVa 10739 – String to Palindrome (вариант определения редакционного расстояния)
- Неклассические
  1. *UVa 00257 – Palinwords* (стандартная задача динамического программирования для поиска палиндромов плюс проверки методом грубой силы)
  2. *UVa 10453 – Make Palindrome* (s: (L, R); t: ($L + 1$, $R - 1$), если $S[L] == S[R]$; или один плюс минимум из ($L + 1$, R) || (L, $R - 1$); также необходимо вывести (распечатать) требуемое решение)
  3. UVa 10617 – Again Palindrome (обработка индексов, в действительности обрабатывается не строка)
  4. **UVa 11022 – String Factoring *** (s: минимальный весовой коэффициент подстроки `[i..j]`)
  5. **UVa 11151 – Longest Palindrome *** (рассматривается в этом разделе)
  6. **UVa 11258 – String Partition *** (рассматривается в этом разделе)
  7. *UVa 11552 – Fewest Flops* (`dp(i,c)` = минимальное число фрагментов после рассмотрения первых i сегментов, заканчивающихся символом `c`)

Известные авторы алгоритмов

Уди Манбер (Udi Manber) – израильский ученый-информатик. Работает в компании Google в качестве одного из вице-президентов инженерной службы. Совместно с Джином Майерсом Манбер разработал структуру данных суффиксный массив (suffix array) в 1991 году.

Юджин «Джин» Уимберли Майерс, младший (Eugene «Gene» Wimberly Myers, Jr.) – американский ученый-информатик и биоинформатик, широко известный по разработке инструментального средства BLAST (Basic Local Alignment Search Tool) для научного анализа. Опубликованная им в 1990 году статья с описанием BLAST была процитирована более 24 000 раз и стала одной из наиболее часто цитируемых работ. Майерс также разработал структуру данных суффиксный массив (suffix array) совместно с Уди Манбером.

6.6. Суффиксный бор, суффиксное дерево, суффиксный массив

Префиксное дерево (бор, луч, нагруженное дерево – suffix trie), суффиксное дерево (suffix tree), суффиксный массив (suffix array) – это эффективные родственные структуры данных для строк. Эти темы не обсуждались в разделе 2.4, поскольку все три упомянутые структуры данных специально предназначены для строк.

6.6.1. Суффиксный бор и его приложения

Суффикс (suffix) i (или i-й суффикс) строки представляет собой «особый случай» подстроки, которая начинается с i-го символа строки и продолжается до самого последнего символа этой строки. Например, 2-м суффиксом строки 'STEVEN' является 'EVEN', а 4-м суффиксом той же строки 'STEVEN' является 'EN' (при индексации, начинающейся с 0).

Суффиксный бор (suffix trie)[1] набора строк S – это дерево всех возможных суффиксов строк в наборе S. Каждая метка ребра представляет символ. Каждая вершина представляет суффикс, обозначенный соответствующей меткой пути к ней: то есть это последовательность меток ребер от корня до этой вершины. Каждая вершина соединена с другими 26 вершинами (или 32 в случае использования кириллического алфавита, но соединения не обязательно должны существовать со всеми прочими вершинами; предполагается, что используются только буквы в верхнем регистре латинского, русского или какого-либо другого алфавита) в соответствии с суффиксами строк в наборе S. Общий префикс для двух суффиксов используется совместно. Каждая вершина имеет два логических флага. Первый флаг определяет, что существует суффикс в наборе строк S, завершающийся в этой вершине. Второй флаг определяет, что существует слово в наборе строк S, завершающееся в этой вершине. Пример: если задан

[1] Это не опечатка. Слово «TRIE» произошло от фразы «information reTRIEval» (извлечение информации).

набор строк S = {'CAR', 'CAT', 'RAT'}, то существуют следующие суффиксы {'CAR', 'AR', 'R', 'CAT', 'AT', 'T', 'RAT', 'AT', 'T'}. После сортировки и удаления повторяющихся элементов получаем: {'AR', 'AT', 'CAR', 'CAT', 'R', 'RAT', 'T'}. На рис. 6.2 показано префиксное дерево (Suffix Trie) с семью завершающими суффиксы вершинами (закрашенные кружки) и с тремя завершающими слова вершинами (закрашенные кружки с пометкой «в словаре»).

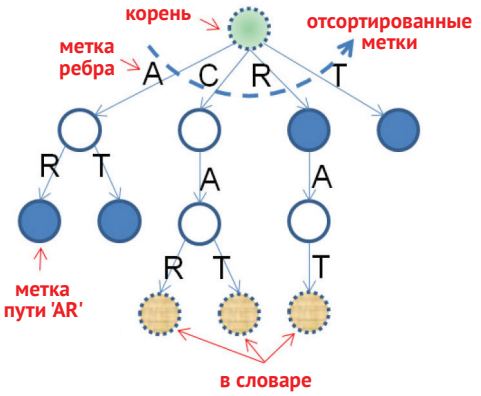


Рис. 6.2 ❖ Префиксное дерево (бор, луч, нагруженное дерево)

Префиксное дерево обычно используется как эффективная структура данных для словаря. Предполагая, что префиксное дерево для набора строк в конкретном словаре уже было построено, можно определить, существует ли строка запроса/шаблона P в этом словаре (префиксном дереве), за время $O(m)$, где m – длина строки P – это эффективный способ[1]. Поиск выполняется с помощью прохода по префиксному дереву, начиная с его корня. Например, если необходимо узнать, существует ли слово P = 'CAT' в префиксном дереве, показанном на рис. 6.2, можно начать с корневого узла, пройти по ребру с меткой 'C', затем по ребру с меткой 'A', наконец, по ребру с меткой 'T'. Поскольку для вершины, в которой мы в итоге оказались, флаг завершения слова имеет значение true (истина), то теперь нам известно, что слово 'CAT' имеется в этом словаре. Но при попытке поиска слова P = 'CAD' мы проходим следующий путь: корень → 'C' → 'A', а затем не обнаруживается ребро с меткой 'D', поэтому мы приходим к выводу о том, что слова 'CAD' нет в этом словаре.

Упражнение 6.6.1.1*. Реализовать структуру данных префиксное дерево (Suffix Trie), используя приведенное выше описание, то есть создать объект вершина с 26 (не более) упорядоченными ребрами, представляющими буквы от «A» до «Z», и двумя флагами, определяющими завершение суффикса/слова.

---

[1] Другой структурой данных для словаря является сбалансированное бинарное (двоичное) дерево поиска (balanced BST) – см. раздел 2.3. Его временная сложность равна $O(\log n \times m)$ для каждого запроса к словарю, где n – количество слов в этом словаре. Причина в том, что одна операция сравнения строк всегда имеет стоимость $O(m)$.

Разместить каждый суффикс каждой строки из набора *S* в префиксном дереве поочередно (один за другим). Проанализировать временную сложность этой стратегии формирования суффиксного бора и сравнить со стратегией формирования суффиксного массива, описанного в разделе 6.6.4. Также выполнить *O*(*m*) запросы для различных строк шаблонов *P*, начиная с корня и следуя по соответствующим меткам ребер.

6.6.2. Суффиксное дерево

Теперь вместо обработки нескольких коротких строк займемся обработкой одной (более) длинной строки. Рассмотрим строку `T = 'GATAGACA$'`. Последний символ `'$'` – это специальный завершающий символ, добавленный к исходной строке `'GATAGACA'`. ASCII-значение этого символа меньше, чем у любого символа в строке `T`. Такой завершающий символ гарантирует, что все суффиксы завершаются в вершинах-листьях.

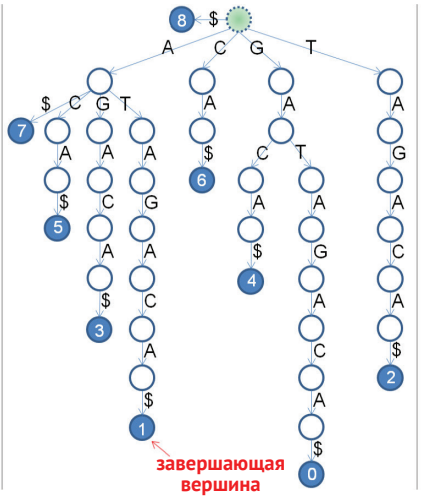
Префиксное дерево (Suffix Trie) для строки T показано на рис. 6.3 в середине. На этот раз в завершающей вершине хранится индекс суффикса, который завершается в этой вершине. На рис. 6.3 видно, что чем длиннее строка `T`, тем больше дублирующихся вершин будет присутствовать в префиксном дереве. Эта структура данных может стать неэффективной. Суффиксное дерево (Suffix Tree) для строки `T` – это префиксное дерево, в котором объединяются вершины, имеющие только одного потомка (по существу, это сжатие пути). При сравнении среднего и правого изображений на рис. 6.3 можно наглядно наблюдать этот процесс сжатия пути. Обратите внимание на метку ребра (edge label) и метку пути (path label) на рис. 6.3, справа. Здесь метка ребра может содержать более одного символа. Суффиксное дерево является гораздо более компактной структурой данных, чем префиксное дерево, поскольку содержит не более $2n$ вершин[1], следовательно, не более $2n - 1$ ребер. Поэтому вместо префиксного дерева мы будем использовать суффиксное дерево в следующих подразделах.

Для многих читателей нашей книги суффиксное дерево может оказаться новой незнакомой структурой данных. Поэтому в третьем издании книги мы добавили инструментальное средство визуализации для показа применения структуры суффиксного дерева для любой (но относительно короткой) входной строки T, определяемой самим читателем. Некоторые практические приложения суффиксного дерева рассматриваются в следующем подразделе 6.6.3 и также включены в комплект визуализации.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/suffixtree.html

[1] Существует не более *n* листьев для *n* суффиксов. Все промежуточные некорневые вершины всегда имеют разветвляющиеся продолжения, следовательно, возможно существование не более $n - 1$ таких вершин. В сумме имеем: *n* (листьев) + ($n - 1$) (промежуточных узлов) + 1 (корень) = $2n$ вершин.

i	Суффикс
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$



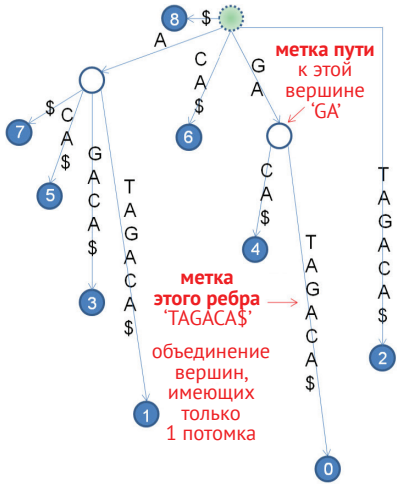


Рис. 6.3 ❖ Суффиксы, префиксное дерево и суффиксное дерево для строки `T = "GATAGACA$"`

Упражнение 6.6.2.1*. Построить (нарисовать) префиксное дерево и суффиксное дерево для строки `T = 'COMPETITIVE$'`. Совет: воспользуйтесь инструментальным средством визуализации, указанным выше.

Упражнение 6.6.2.2*. Заданы две вершины, представляющие два различных суффикса, например суффикс 1 и суффикс 5 на рис. 6.3, справа. Определить самый длинный общий префикс для этих суффиксов (это префикс 'A').

6.6.3. Практические приложения суффиксного дерева

Предположим, что суффиксное дерево для строки T уже сформировано, тогда мы можем воспользоваться им для следующих приложений (список не является полным).

Поиск совпадений в строках за время O(m + occ)

Используя суффиксное дерево, можно найти все (полностью совпадающие) вхождения строки шаблона P в строке T за время $O(m + occ)$, где m – длина строки шаблона P, а occ – суммарное количество вхождений P в T независимо от длины строки T. Если суффиксное дерево уже сформировано, то этот способ работает намного быстрее, чем алгоритмы поиска совпадений в строках, рассмотренные ранее в разделе 6.4.

При заданном существующем суффиксном дереве строки T нашей задачей является поиск в этом суффиксном дереве вершины x, метка пути к которой представляет собой строку шаблона P. Напомним, что в конечном итоге совпадение является общим префиксом для строки шаблона P и некоторых суффиксов строки T. Эта операция выполняется всего лишь за один проход от корня до листа суффиксного дерева строки T по искомым меткам ребер. Вершина

с меткой пути, идентичной шаблону *P*, является требуемой вершиной *x*. Далее индексы суффиксов, хранящиеся в завершающих вершинах (листьях) поддерева с корнем в вершине *x*, представляют собой искомые вхождения шаблона *P* в строке *T*.

Пример: в суффиксном дереве для строки `T = 'GATAGACA$'`, показанном на рис. 6.4, при заданном шаблоне `P = 'A'` можно просто выполнить проход от корня, двигаясь по ребру с меткой 'A', чтобы найти вершину *x* с меткой пути 'A'. Существует четыре вхождения[1] шаблона 'A' в поддереве с корнем в *x*. Это суффикс 7 `'A$'`, суффикс 5 `'ACA$'`, суффикс 3 `'AGACA$'` и суффикс 1 `'ATAGACA$'`.

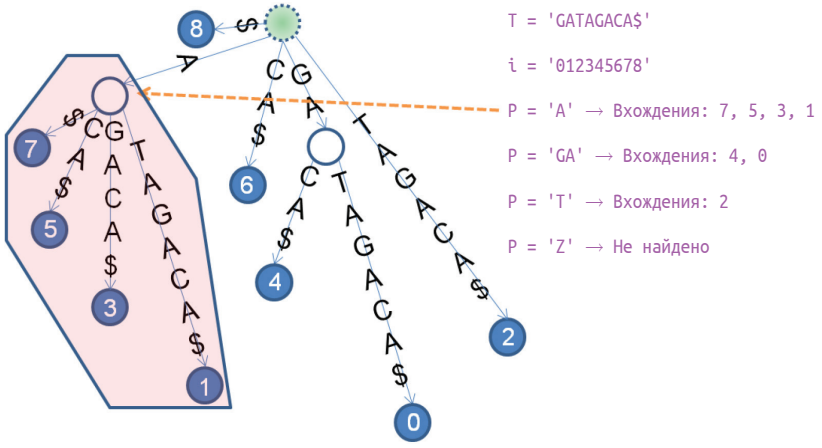


Рис. 6.4 ❖ Поиск совпадений в строке `T = 'GATAGACA$'` с различными строками шаблонов

Поиск самой длинной повторяющейся подстроки за время O(n)

Если задано суффиксное дерево для строки *T*, то можно также найти самую длинную повторяющуюся подстроку (Longest Repeated Substring – LRS[2]) в строке *T* эффективным способом. Задача поиска самой длинной повторяющейся строки – это задача поиска самой длинной подстроки, которая встречается в исходной строке не менее двух раз. Ответом (решением) этой задачи является метка пути самой глубокой внутренней вершины *x* в суффиксном дереве строки *T*. Вершина *x* может быть найдена линейным обходом вершин дерева по дереву. Из того факта, что *x* является внутренней вершиной, следует, что она представляет более одного суффикса строки *T* (то есть существует > 1 завершающей вершины в поддереве с корнем в *x*), и эти суффиксы совместно используют общий префикс (что подразумевает повторяющуюся подстроку). Из того

---

[1] Точнее, *occ* – это размер поддерева с корнем *x*, который может быть бóльшим (но не более, чем в два раза), чем действительное число (*occ*) завершающих вершин (листьев) в поддереве с корнем *x*.

[2] Эта задача имеет несколько интересных практических приложений: поиск припева в тексте песни (припев повторяется несколько раз), поиск (самых длинных) повторяющихся фраз в (длинной) речи политика и т. д.

факта, что *x* является самой глубокой внутренней вершиной (от корня), следует, что ее метка пути является самой длинной повторяющейся подстрокой.

Пример: в суффиксном дереве строки `T = 'GATAGACA$'` на рис. 6.5 самой длинной повторяющейся строкой является `'GA'`, так как это метка пути самой глубокой внутренней вершины *x* – `'GA'` повторяется дважды в строке `'GATAGACA$'`.

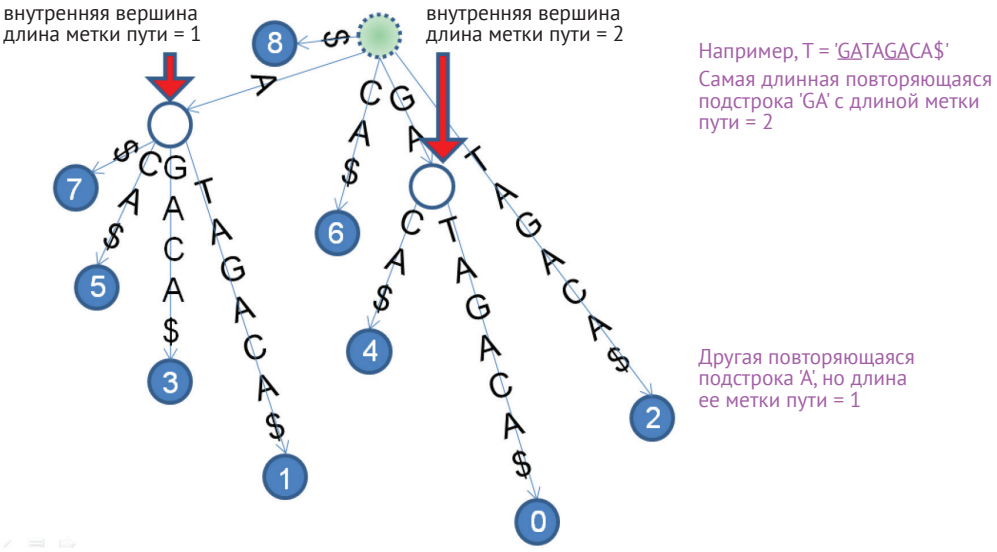


Рис. 6.5 ❖ Самая длинная повторяющаяся подстрока в строке `T = 'GATAGACA$'`

Поиск самой длинной общей подстроки за время O(n)

Задача поиска самой длинной общей подстроки (Longest Common Substring – LCS[1]) в двух и более строках может быть решена за линейное время[2] с помощью суффиксного дерева. Без ограничения общности рассмотрим вариант только лишь с двумя строками T_1 и T_2. Можно сформировать обобщенное суффиксное дерево, объединяющее суффиксные деревья для строк T_1 и T_2. Чтобы различать источник каждого суффикса, воспользуемся двумя различными символами, обозначающими завершающие вершины, по одному для каждой строки. Затем пометим внутренние вершины (internal vertices), которые имеют в своих поддеревьях вершины с различными завершающими символами. Суффиксы, представленные этими помеченными внутренними вершинами, совместно используют общий префикс и присутствуют в обеих строках T_1 и T_2. Таким образом, эти помеченные внутренние вершины представляют общие подстроки в строках T_1 и T_2. Поскольку нас интересует самая длинная общая подстрока,

---

[1] Следует отметить, что «подстрока» (substring) отличается от «подпоследовательности» (subsequence). Например, «BCE» является подпоследовательностью, но не подстрокой строки «ABCDEF», тогда как «BCD» (непрерывная) является и подпоследовательностью, и подстрокой строки «ABCDEF».

[2] Только если используется алгоритм создания суффиксного дерева за линейное время (этот аспект не обсуждается в данной книге, см. [65]).

в качестве результата выводится метка пути самой глубокой помеченной вершины.

Например, для строк T_1 = 'GATAGACA$' и T_2 = 'CATA#' самой длинной общей подстрокой является 'ATA' длиной 3. На рис. 6.6 можно видеть, что вершины с метками пути 'A', 'ATA', 'CA' и 'TA' имеют два различных завершающих символа (отметим, что вершина с меткой пути 'GA' не рассматривается как оба суффикса 'GACA$' и 'GATAGACA$' из строки T_1). Это общие подстроки в строках T_1 и T_2. Самая глубокая помеченная вершина 'ATA', и это самая длинная общая подстрока в строках T_1 и T_2.

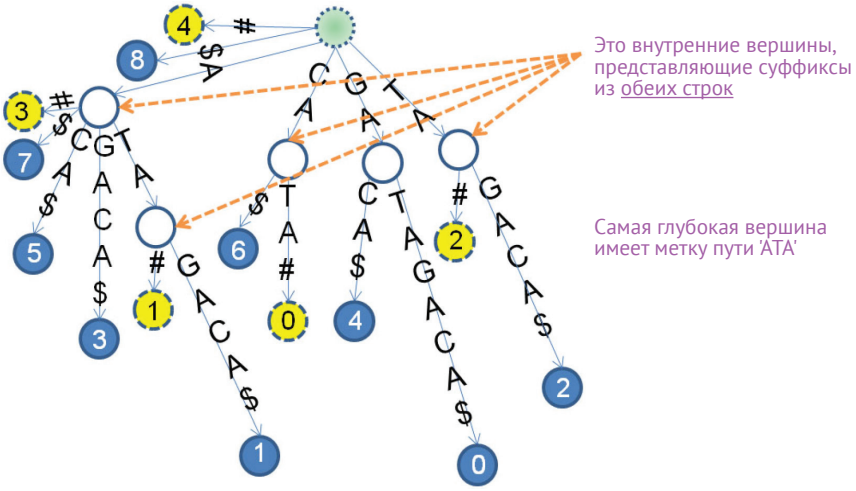


Рис. 6.6 ❖ Объединенное суффиксное дерево для строк T1 = 'GATAGACA$' и T2 = 'CATA#' и самая длинная общая подстрока в этих строках

Упражнение 6.6.3.1. Дано то же суффиксное дерево, что и на рис. 6.4. Найти шаблон P = 'CA' и шаблон P = 'CAT'.

Упражнение 6.6.3.2. Найти самую длинную повторяющуюся подстроку в строке T = 'CGACATTACATTA$'. Сначала постройте суффиксное дерево.

Упражнение 6.6.3.3*. Вместо поиска самой длинной повторяющейся подстроки теперь необходимо найти повторяющуюся подстроку, которая встречается чаще всего. Среди нескольких возможных кандидатов выберите самую длинную подстроку. Например, если T = 'DEFG1ABC2DEFG3ABC4ABC', то ответом будет подстрока 'ABC' длиной 3, которая встречается три раза (не подстрока 'BC' длиной 2 и не подстрока 'C' длиной 1, которые также встречаются три раза), а не подстрока 'DEFG', которая хотя и имеет длину 4, но встречается только два раза. Определите и опишите стратегию поиска решения.

Упражнение 6.6.3.4. Найти самую длинную общую подстроку в строках T_1 = 'STEVEN$' и T_2 = 'SEVEN#'.

Упражнение 6.6.3.5*. Подумайте, как обобщить этот метод для поиска самой длинной общей подстроки в более чем двух строках. Например, даны три строки T_1 = 'STEVEN$', T_2 = 'SEVEN#' и T_3 = 'EVE@'. Как определить, что самой длинной общей подстрокой в этих строках является 'EVE'?

Упражнение 6.6.3.6*. Модифицируйте решение таким образом, чтобы реализовать поиск самой длинной общей подстроки в k из n строк, где $k \leqslant n$. Например, заданы те же три строки T_1, T_2 и T_3, что и в предыдущем упражнении. Как определить, что самой длинной общей подстрокой для двух строк из заданных трех является 'EVEN'?

6.6.4. Суффиксный массив

В предыдущем подразделе рассматривалось несколько задач обработки строк, которые можно решить, если уже сформировано суффиксное дерево. Но эффективная реализация создания суффиксного дерева за линейное время (см. [65]) сложна, следовательно, ее применение в условиях олимпиады по программированию слишком рискованно. К счастью, следующая структура данных, которую мы будем рассматривать, – суффиксный массив (suffix array), разработанный Уди Манбером (Udi Manber) и Джином Майерсом (Gene Myers) [43], – обладает аналогичными функциональными свойствами и возможностями, как и суффиксное дерево, но при этом (намного) проще реализуется его создание и использование, особенно в условиях олимпиады по программированию. Таким образом, мы пропускаем описание формирования суффиксного дерева за время $O(n)$ [65], а вместо этого сосредоточим внимание на создании суффиксного массива $O(n \times \log n)$ [68], которое проще реализовать. Затем в следующем подразделе мы рассмотрим возможности применения суффиксного массива для решения задач, решения которых с использованием суффиксного дерева были показаны в предыдущем разделе.

По существу, суффиксный массив – это массив целых чисел, в котором хранится перестановка из n индексов отсортированных суффиксов. Например, рассмотрим все ту же строку T = 'GATAGACA$' при n = 9. Суффиксный массив для строки T – это перестановка целых чисел [0..n-1] = {8, 7, 5, 3, 1, 6, 4, 0, 2}, как показано на рис. 6.7. Таким образом, суффиксы в отсортированном порядке представлены как суффикс SA[0] = суффикс 8 = '$', суффикс SA[1] = суффикс 7 = 'A$', суффикс SA[2] = суффикс 5 = 'ACA$', …, наконец, суффикс SA[8] = суффикс 2 = 'TAGACA$'.

Суффиксное дерево и суффиксный массив тесно связаны. Как можно видеть на рис. 6.8, при проходе по суффиксному дереву завершающие вершины (листья) посещаются в том порядке, в котором они содержатся в суффиксном массиве. Внутренняя вершина (internal vertex) в суффиксном дереве соответствует диапазону (range) в суффиксном массиве (набору отсортированных суффиксов, которые совместно используют общий префикс). Завершающая вершина (terminating vertex) (всегда расположена в листе из-за использования завершающего символа строки) в суффиксном дереве соответствует отдельному ин-

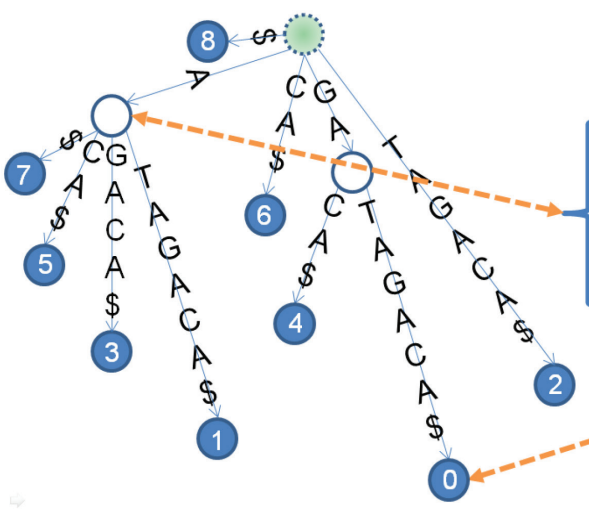
дексу (individual index) в суффиксном массиве (один индекс). Рекомендуется запомнить эти соответствия. Они будут весьма полезны при изучении следующего подраздела, в котором рассматриваются практические приложения суффиксного массива.

i	Суффикс
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

Сортировка →

i	SA[i]	Суффикс
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Рис. 6.7 ❖ Сортировка суффиксов для строки `T = 'GATAGACA$'`



i	SA[i]	Суффикс
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Рис. 6.8 ❖ Суффиксное дерево и суффиксный массив для строки `T = 'GATAGACA$'`

Суффиксный массив достаточно эффективен для многих трудных задач обработки строк, в том числе и длинных строк на олимпиадах по программированию. Здесь мы представляем два способа создания суффиксного массива для заданной строки `T[0..n-1]`. Первый способ очень прост, как показано ниже.

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

#define MAX_N 1010                                          // первый метод: O(n^2 log n)
char T[MAX_N];                  // этот простейший метод создания СМ не способен обработать
                                                            // более 1000 символов
int SA[MAX_N], i, n;                              // в условиях олимпиады по программированию

bool cmp( int a, int b ) { return strcmp( T + a, T + b ) < 0; }                    // O(n)

int main()
{
  n = (int)strlen(gets(T));          // считывание строки и немедленное вычисление ее длины
  for( int i=0; i < n; i++ ) SA[i] = i;      // начальное состояние СМ: {0, 1, 2, ..., n-1}
  sort( SA, SA + n, cmp );             // сортировка: O(n log n) * cmp: O(n) = O(n^2 log n)
  for( i=0; i < n; i++ )
    printf( "%2d\t%s\n", SA[i], T + SA[i] );
} // return 0;
```

Если применить этот простой код к строке `T = 'GATAGACA$'`, то будет выполнена сортировка всех суффиксов с помощью стандартных библиотечных функций сортировки и сравнения строк. В результате получим правильный суффиксный массив = {8, 7, 5, 3, 1, 6, 4, 0, 2}. Но этот метод более или менее эффективен только для олимпиадных задач при $n \leqslant 1000$. Общее время выполнения данного алгоритма равно $O(n^2 \log n)$, так как операция (библиотечная функция) `strcmp`, используемая для определения порядка двух (возможно, длинных) суффиксов, связана с чрезвычайно большими издержками (накладными расходами), дополняющими сложность $O(n)$ для сравнения одной пары суффиксов.

Более эффективным методом создания суффиксного массива является сортировка ранжирующих пар (ranking pairs) суффиксов за $O(\log_2 n)$ итераций из $k = 1, 2, 4, \ldots$, самой последней (ближайшей) степени 2, которая меньше n. На каждой итерации этот алгоритм создания суффиксного массива сортирует суффиксы на основе ранжирующей пары (`RA[SA[i]]`, `RA[SA[i]+k]`) для суффикса `SA[i]`. Этот алгоритм основан на описании, приведенном в [68]. Ниже показан пример выполнения данного алгоритма для строки `T = 'GATAGACA$'` и $n = 9$:

- сначала `SA[i] = i` и `RA[i]` = ASCII-значению `T[i]` $\forall i \in [0..n-1]$ (см. табл. 6.1, слева). На итерации $k = 1$ ранжирующей парой суффикса `SA[i]` является (`RA[SA[i]]`, `RA[SA[i]+1]`);

Таблица 6.1. Слева: перед сортировкой, справа: после сортировки; k = 1; показан начальный порядок отсортированных суффиксов

i	SA[*i*]	Суффикс	RA[SA[*i*]]	RA[SA[*i*]+1]
0	0	GATAGACA$	71 (G)	65 (A)
1	1	ATAGACA$	65 (A)	84 (T)
2	2	TAGACA$	84 (T)	65 (A)
3	3	AGACA$	65 (A)	71 (G)
4	4	GACA$	71 (G)	65 (A)
5	5	ACA$	65 (A)	67 (C)

i	SA[*i*]	Суффикс	RA[SA[*i*]]	RA[SA[*i*]+1]
0	8	$	36 ($)	00 (-)
1	7	A$	65 (A)	36 ($)
2	5	ACA$	65 (A)	67 (C)
3	3	AGACA$	65 (A)	71 (G)
4	1	ATAGACA$	65 (A)	84 (T)
5	6	CA$	67 (C)	65 (A)

Таблица 6.1 *(окончание)*

i	SA[*i*]	Суффикс	RA[SA[*i*]]	RA[SA[*i*]+1]
6	6	CA$	67 (C)	65 (A)
7	7	A$	65 (A)	36 ($)
8	8	$	36 ($)	00 (-)
Начальное ранжирование по RA[*i*] = ASCII-значению символа строки *T*[*i*] $ = 36, A = 65, C = 67, G = 71, T = 84				

i	SA[*i*]	Суффикс	RA[SA[*i*]]	RA[SA[*i*]+1]
6	0	GATAGA-CA$	71 (G)	65 (A)
7	4	GACA$	71 (G)	65 (A)
8	2	TAGACA$	84 (T)	65 (A)
Если SA[*i*]+*k* >= *n* (превышает длину строки *T*), то по умолчанию присваивается ранг 0 с меткой -				

Пример 1. Ранг суффикса 5 `'ACA$'` равен (`'A'`, `'C'`) = (65, 67).
Пример 2. Ранг суффикса 3 `'AGACA$'` равен (`'A'`, `'G'`) = (65, 71).
После сортировки таких ранжирующих пар формируется порядок суффиксов, показанный в табл. 6.1, справа, где суффикс 5 `'ACA$'` расположен перед суффиксом 3 `'AGACA$'`, и т. д.

- на итерации $k = 2$ ранжирующей парой для суффикса `SA[i]` является (`RA[SA[i]]`, `RA[SA[i]+2]`). Теперь эта ранжирующая пара получена при рассмотрении только первой и второй пар символов. Для получения новых ранжирующих пар нет необходимости повторно вычислять многие соотношения. Для первого суффикса, то есть суффикса 8 `'$'`, устанавливается новый ранг $r = 0$. Затем выполняется итерация от $i = [1..n - 1]$. Если ранжирующая пара суффикса SA[i] отличается от ранжирующей пары предыдущего суффикса SA[$i - 1$] в отсортированном порядке, то ранг увеличивается на единицу $r = r + 1$. В противном случае ранг остается прежним r (см. табл. 6.2, слева).

Таблица 6.2. Слева: до сортировки, справа: после сортировки; k = 2; суффиксы `'GATAGACA'` и `'GACA'` поменялись местами

i	SA[*i*]	Суффикс	RA[SA[*i*]]	RA[SA[*i*]+2]
0	8	$	0 ($-)	0 (--)
1	7	A$	1 (A$)	0 (--)
2	5	ACA$	2 (AC)	1 (A$)
3	3	AGACA$	3 (AG)	2 (AC)
4	1	ATAGACA$	4 (AT)	3 (AG)
5	6	CA$	5 (CA)	0 ($-)
6	0	GATAGACA$	6 (GA)	7 (TA)
7	4	GACA$	6 (GA)	5 (CA)
8	2	TAGACA$	7 (TA)	6 (GA)
$- (первому элементу) присвоен ранг 0, затем для $i = 1$ до $n - 1$ выполняется сравнение ранга пары текущей строки с предыдущей строкой				

i	SA[*i*]	Суффикс	RA[SA[*i*]]	RA[SA[*i*]+2]
0	8	$	0 ($-)	0 (--)
1	7	A$	1 (A$)	0 (--)
2	5	ACA$	2 (AC)	1 (A$)
3	3	AGACA$	3 (AG)	2 (AC)
4	1	ATAGACA$	4 (AT)	3 (AG)
5	6	CA$	5 (CA)	0 ($-)
6	4	GACA$	6 (GA)	5 (CA)
7	0	GATAGACA$	6 (GA)	7 (TA)
8	2	TAGACA$	7 (TA)	6 (GA)
Если SA[*i*] + *k* >= *n* (превышает длину строки *T*), то по умолчанию присваивается ранг 0 с меткой -				

Пример 1. В табл. 6.1, справа ранжирующей парой суффикса 7 `'A$'` является (65, 36), которая отличается от ранжирующей пары предыдущего суффикса 8 `'$-'` (36, 0). Следовательно, в табл. 6.2, слева суффиксу 7 присваивается новый ранг 1.

Пример 2. В табл. 6.1, справа ранжирующей парой суффикса 4 'GACA$' является (71, 65), совпадающая с ранжирующей парой предыдущего суффикса 0 'GATAGACA$', то есть тоже (71, 65). Таким образом, в табл. 6.2, слева поскольку суффиксу 0 присваивается новый ранг 6, то и суффиксу 4 также присваивается тот же новый ранг 6.

Сразу после обновления RA[SA[i]] $\forall i \in [0..n-1]$ также можно без затруднений определить значение RA[SA[i]+k]. В соответствии с описанием условий задачи если SA[i]+k $\geqslant n$, то по умолчанию присваивается ранг 0. Более подробное описание реализации всех аспектов на этом шаге см. в **упражнении 6.6.4.2***.

На этом этапе ранжирующей парой суффикса 0 'GATAGACA$' является (6, 7), а для суффикса 4 'GACA$' (6, 5). Эти два суффикса остаются неотсортированными, в то время как все прочие суффиксы уже расположены в правильном порядке. После следующего раунда сортировки порядок суффиксов показан в табл. 6.2, справа;

- на итерации $k = 4$ ранжирующей парой суффикса SA[i] является (RA[SA[i]], RA[SA[i]+4]). Теперь эта ранжирующая пара определяется при рассмотрении только первой и второй четверок символов. Здесь следует отметить, что предыдущие ранжирующие пары суффикса 4 (6, 5) и суффикса 0 (6, 7) в табл. 6.2, справа теперь различны. Таким образом, после изменения рангов все n суффиксов в табл. 6.3 получают различные ранги. Это легко подтвердить, если проверить равенство RA[SA[n-1]] == n-1. После установления данного факта мы имеем успешно сформированный суффиксный массив. Отметим, что основная работа по сортировке была выполнена только в нескольких первых итерациях, а обычно большого количества итераций не требуется.

Таблица 6.3. До и после последнего раунда сортировки; k = 4; изменений нет

i	SA[*i*]	Суффикс	RA[SA[*i*]]	RA[SA[*i*]+4]
0	8	$	0 ($---)	0 (----)
1	7	A$	1 (A$--)	0 (----)
2	5	ACA$	2 (ACA$)	0 (----)
3	3	AGACA$	3 (AGAC)	1 (A$--)
4	1	ATAGACA$	4 (ATAG)	2 (ACA$)
5	6	CA$	5 (CA$-)	0 (----)
6	4	GACA$	6 (GACA)	0 ($---)
7	0	GATAGACA$	7 (GATA)	6 (GACA)
8	2	TAGACA$	8 (TAGA)	5 (GA$-)
Теперь все суффиксы имеют различные ранги Работа завершена				

Для многих читателей нашей книги описанный выше метод создания суффиксного массива может оказаться новым незнакомым алгоритмом. Поэтому в третьем издании книги мы добавили инструментальное средство визуализации для показа всех этапов выполнения алгоритма создания суффиксного массива для любой (но относительно короткой) входной строки *T*, определяемой самим читателем. Некоторые практические приложения суффиксного массива

рассматриваются в следующем подразделе 6.6.5 и также включены в комплект визуализации.

Инструментальное средство визуализации:
www.comp.nus.edu.sg/~stevenha/visualization/suffixarray.html

Описанную выше процедуру сортировки ранжирующих пар можно реализовать с использованием (встроенных) функций сортировки $O(n \times \log n)$ стандартной библиотеки. Так как процесс сортировки повторяется не более $\log n$ раз, общая временная сложность алгоритма равна $O(\log n \times n \times \log n) = O(n \times \log^2 n)$. При такой временной сложности можно успешно обрабатывать строки длиной до $\approx 10K$ символов. Но поскольку сортировка выполняется только для пар небольших целых чисел, можно воспользоваться алгоритмом с линейным временем выполнения – двухпроходным алгоритмом поразрядной сортировки (Radix sort) (который выполняет внутренний вызов процедуры сортировки подсчетом – counting sort, – более подробно об этом см. раздел 9.32) для сокращения времени сортировки до $O(n)$. Так как процесс сортировки повторяется не более $\log n$ раз, общая временная сложность алгоритма равна $O(\log n \times n) = O(n \times \log n)$. Теперь можно обрабатывать строки длиной до $\approx 100K$ символов – это обычный размер строк на олимпиадах по программированию.

Ниже приведен код последнего варианта реализации $O(n \times \log n)$. Внимательно изучите код, чтобы понять, как он работает. Замечание только для участников студенческих олимпиад (ICPC): поскольку вам разрешено брать с собой печатные копии материалов в соревновательный зал, рекомендуем включить этот код в библиотеку вашей команды.

```
#define MAX_N 100010                                  // второй вариант реализации: O(n*log n)
char T[MAX_N];                                         // строка ввода, длина до 100K символов
int n;                                                                    // длина строки ввода
int RA[MAX_N], tempRA[MAX_N];                        // массив рангов и временный массив рангов
int SA[MAX_N], tempSA[MAX_N];              // суффиксный массив и временный суффиксный массив
int c[MAX_N];                               // для сортировки подсчетом /поразрядной сортировки

void countingSort(int k)                                                                  // O(n)
{
  int i, sum, maxi = max( 300, n );                     // до 255 ASCII-символов или длина n
  memset( c, 0, sizeof c );                                              // очистка таблицы частот
  for( i=0; i < n; i++ )         // подсчет частоты (появления) каждого целочисленного ранга
    c[i+k<n ? RA[i + k] : 0]++;
  for( i = sum = 0; i < maxi; i++ ) {
    int t = c[i]; c[i] = sum; sum += t;
  }
  for( i=0; i < n; i++ )                    // перемешать суффиксный массив, если необходимо
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for( i=0; i < n; i++ )                                  // обновление суффиксного массива SA
    SA[i] = tempSA[i];
}

void constructSA()                          // эта версия может обработать до 100 000 символов
```

```
{
  int i, k, r;
  for( i=0; i < n; i++ ) RA[i] = T[i];                          // начальное ранжирование
  for( i=0; i < n; i++ ) SA[i] = i;  // начальный суффиксный массив SA: {0, 1, 2, ..., n-1}
  for( k=1; k < n; k <<= 1 ) {                    // повторить процесс сортировки log n раз
    countingSort(k);                                  // собственно поразрядная сортировка:
                                                      // сортировка на основе 2-го элемента
    countingSort(0);               // затем (стабильная) сортировка на основе 1-го элемента
    tempRA[SA[0]] = r = 0;                          // изменение ранга; начать с ранга r = 0
    for( i=1; i < n; i++ )                                  // сравнение смежных суффиксов
      tempRA[SA[i]] =                     // если пары одинаковы => оставить тот же ранг r;
                                                          // иначе увеличить r на 1
       (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for( i=0; i < n; i++ )                                   // обновить массив рангов RA
      RA[i] = tempRA[i];
    if( RA[SA[n-1]] == n-1 ) break;                       // эффективный прием оптимизации
  }
}

int main()
{
  n = (int)strlen( gets(T) );          // строка ввода T в нормальном виде, без символа '$'
  T[n++] = '$';                                  // добавление завершающего символа '$'
  constructSA();
  for( int i=0; i < n; i++ )
    printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

Упражнение 6.6.4.1*. Описать (продемонстрировать) шаги вычисления суффиксного массива для строки `T = 'COMPETITIVE$'` с $n = 12$. Сколько итераций сортировки необходимо для получения суффиксного массива?

Совет: воспользуйтесь указанным выше инструментальным средством визуализации создания суффиксного массива.

Упражнение 6.6.4.2*. В приведенном выше коде создания суффиксного массива есть следующая строка:

```
(RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
```

Не приводит ли выполнение этого кода к выходу за границу массива в некоторых случаях?

То есть может ли значение `SA[i]+k` или `SA[i-1]+k` быть $\geqslant n$ и привести к аварийному завершению программы? Объясните свой ответ.

Упражнение 6.6.4.3*. Будет ли приведенный выше код создания суффиксного массива работать корректно, если в строке ввода T содержится пробел (ASCII-код = 32)?

Совет-подсказка: используемый по умолчанию завершающий символ $ имеет ASCII-код = 36.

6.6.5. Практические приложения суффиксного массива

Ранее уже отмечалось, что суффиксный массив тесно связан с суффиксным деревом. В этом подразделе мы покажем, что с помощью суффиксного массива (который проще сформировать) можно решить задачи обработки строк, рассмотренные в разделе 6.6.3, которые ранее решались с использованием суффиксного дерева.

Поиск совпадений в строках $O(m \times \log n)$

После создания суффиксного массива для строки T можно выполнить поиск строки шаблона P (длиной m) в строке T (длиной n) за время $O(m \times \log n)$. Множитель $\log n$ свидетельствует о замедлении поиска по сравнению с версией суффиксного дерева, но на практике это вполне приемлемо. Сложность $O(m \times \log n)$ определяется из того факта, что можно выполнить две операции бинарного поиска $O(\log n)$ в отсортированных суффиксах, а кроме того, можно выполнить до $O(m)$ операций сравнения суффиксов[1]. Первая/вторая операция бинарного поиска предназначена для определения нижней/верхней границы соответственно. Эта нижняя/верхняя граница представляет собой (соответственно) наименьшее/наибольшее значение `i`, такое, что префикс суффикса `SA[i]` совпадает со строкой шаблона `P`. Все суффиксы между нижней и верхней границами являются вхождениями строки шаблона `P` в строку `T`. Реализация этого метода показана ниже.

```
ii stringMatching()                              // поиск совпадений в строках за O(m*log n)
{
  int lo = 0, hi = n-1, mid = lo;                       // допустимое совпадение = [0..n-1]
  while( lo < hi ) {                                              // поиск нижней границы
    mid = (lo + hi) / 2;                     // округление с недостатком (в меньшую сторону)
    int res = strncmp( T + SA[mid], P, m );             // попытка найти P в суффиксе 'mid'
    if( res >= 0 )  hi = mid;              // отсечение верхней половины (внимание: знак >=)
    else            lo = mid;                      // отсечение нижней половины, включая mid
  }
  if( strncmp( T+SA[lo], P, m ) != 0 )  return ii( -1, -1 );  // если совпадение не найдено
  ii ans;  ans.first = lo;
  lo = 0;  hi = n-1;  mid = lo;
  while( lo < hi ) {            // если нижняя граница найдена, нужно найти верхнюю границу
    mid = (lo + hi) / 2;
    int res = strncmp( T + SA[mid], P, m );
    if( res > 0 )  hi = mid;                                    // отсечение верхней половины
    else           lo = mid + 1;                // отсечение нижней половины, включая mid
  }                               // (обратите внимание на выбранную ветвь, когда res == 0)
  if( strncmp( T+SA[hi], P, m ) != 0 )  hi--;                               // особый случай
  ans.second = hi;
  return ans;
}  // возврат нижней/верхней границы как первого/второго элемента (соответственно) пары ans

int main()
```

[1] Это вполне достижимая производительность при использовании функции `strncmp` для сравнения только первых m символов в обоих суффиксах.

```
{
  n = (int)strlen( gets(T) ); // ввод строки T в обычном виде, без завершающего символа '$'
  T[n++] = '$';                                          // добавление завершающего символа
  constructSA();
  for( int i=0; i < n; i++ )
    printf( "%2d\t%s\n", SA[i], T+SA[i] );

  while( m = (int)strlen( gets(P) ), m ) {    // остановить процесс, если P - пустая строка
    ii pos = stringMatching();
    if( pos.first != -1 && pos.second != -1 ) {
      printf( "%s found, SA[%d..%d] of %s\n", pos.first, pos.second, T );
      printf( "They are:\n" );
      for( int i = pos.first; i <= pos.second; i++ )
        printf( "  %s\n", T + SA[i] );
    } else  printf( "%s is not found in %s\n", P, T );
  }
}      // return 0;
```

Пример выполнения этого алгоритма поиска совпадений в строках с использованием суффиксного массива для строки `T = 'GATAGACA$'` и шаблона `P = 'GA'` показан ниже в таблице на рис. 6.9.

Процесс начинается с поиска нижней границы. Текущий диапазон `i = [0..8]`, следовательно, срединное значение `i = 4`. Сравниваются первые два символа суффикса `SA[4]`, то есть `'ATAGACA$'` с шаблоном `P = 'GA'`. Так как `P = 'GA'` больше, поиск продолжается в диапазоне `i = [5..8]`. Далее сравниваются первые два символа суффикса `SA[6]`, то есть `'GACA$'` с шаблоном `P = 'GA'`. Найдено совпадение. Поскольку в настоящий момент определяется нижняя граница, мы не останавливаемся здесь, а продолжаем поиск в диапазоне `i = [5..6]`. `P = 'GA'` больше, чем суффикс `SA[5]`, то есть `'CA$'`. Здесь процесс останавливается. Индекс `i = 6` является нижней границей, таким образом, суффикс `SA[6]` `'GACA$'` соответствует первому вхождению шаблона `P = 'GA'` в качестве префикса для суффикса в списке отсортированных суффиксов.

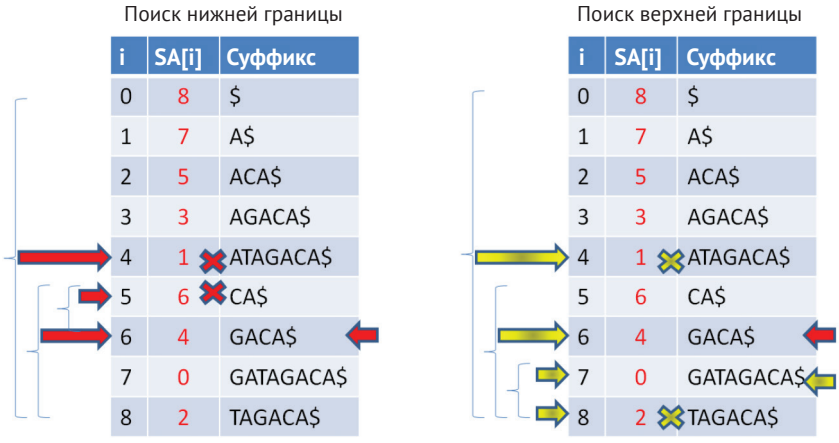


Рис. 6.9 ❖ Поиск совпадений в строках с использованием суффиксного массива

Далее выполняется поиск верхней границы. Первый шаг точно такой же, как и при поиске нижней границы. Но на втором шаге обнаруживается совпадение суффикса `SA[6] 'GACA$'` с шаблоном `P = 'GA'`. Поскольку сейчас идет поиск верхней границы, процесс продолжается в диапазоне `i = [7..8]`. Еще одно совпадение найдено при сравнении суффикса `SA[7] 'GATAGACA$'` с шаблоном `P = 'GA'`. Здесь процесс останавливается. Индекс `i = 7` является верхней границей в данном примере, таким образом, суффикс `SA[7] 'GATAGACA$'` – это последнее вхождение шаблона `P = 'GA'` качестве префикса для суффикса в списке отсортированных суффиксов.

Поиск самого длинного общего префикса O(n)

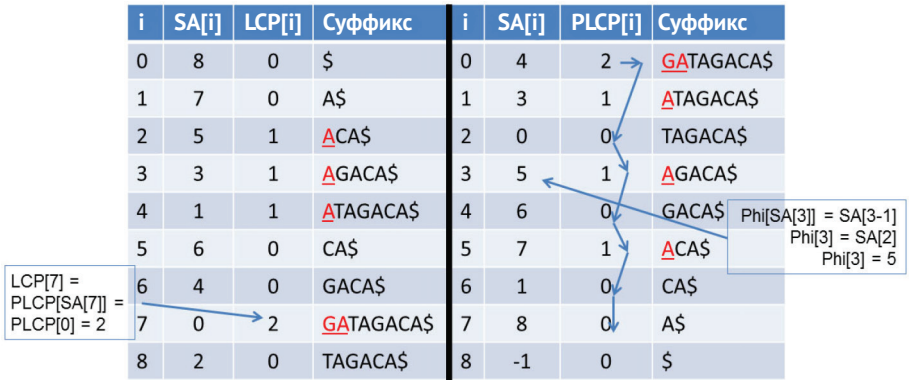
Если задан суффиксный массив для строки *T*, то можно вычислить самый длинный общий префикс (Longest Common Prefix – LCP) между последовательными суффиксами в упорядоченном суффиксном массиве. По определению LCP[0] = 0, так как суффикс SA[0] – это первый суффикс в упорядоченном суффиксном массиве без каких-либо других суффиксов, предшествующих первому. Для $i > 0$ LCP[i] = длине общего префикса между суффиксами SA[i] и SA[$i - 1$]. См. таблицу на рис. 6.10, слева. Можно вычислить LCP непосредственно по определению, используя код, приведенный ниже. Но это медленный способ, поскольку значение L может увеличиваться до $O(n^2)$. Это лишает смысла цель создания суффиксного массива за время $O(n \times \log n)$, как показано в разделе 6.8.

```
void computeLCP_slow()
{
  LCP[0] = 0;                                                  // значение по умолчанию
  for( int i=1; i < n; i++ ) {                              // вычисление LCP по определению
    int L = 0;                                          // необходимо всегда сбрасывать L в 0
    while( T[SA[i]+L] == T[SA[i-1]+L] )  L++;                  // тот же L-й символ, L++
    LCP[i] = L;
  }
}
```

Более эффективное решение с использованием теоремы о самом длинном общем префиксе с перестановкой (Permuted Longest-Common-Prefix – PLCP) [37] описано ниже. Основная идея проста: легче вычислять LCP для суффиксов в исходном порядке, нежели в лексикографическом. На рис. 6.10 в правой таблице указана исходная позиция в упорядоченном списке суффиксов для строки `T = 'GATAGACA$'`. Очевидно, что столбец `PLCP[i]` формирует шаблон: блок уменьшения на 1 (2 → 1 → 0); увеличение на 1; снова блок уменьшения на 1 (1 → 0); снова увеличение на 1; снова блок уменьшения на 1 (1 → 0) и т. д.

Теорема PLCP утверждает, что общее количество операций увеличения (и уменьшения) не превышает $O(n)$. Описанный выше подход и обеспечение сложности $O(n)$ используются в коде, приведенном ниже.

Сначала вычисляется `Phi[i]`, где хранится индекс суффикса, предшествующего суффиксу `SA[i]` в упорядоченном суффиксном массиве. По определению `Phi[SA[0]] = -1`, то есть для суффикса `SA[0]` не существует предшествующего суффикса. Рекомендуется уделить немного времени для проверки правильности значений в столбце `Phi[i]` в правой таблице на рис. 6.10. Например, `Phi[SA[3]] = SA[3-1]`, поэтому `Phi[3] = SA[2] = 5`.



i	SA[i]	LCP[i]	Суффикс
0	8	0	$
1	7	0	A$
2	5	1	ACA$
3	3	1	AGACA$
4	1	1	ATAGACA$
5	6	0	CA$
6	4	0	GACA$
7	0	2	GATAGACA$
8	2	0	TAGACA$

i	SA[i]	PLCP[i]	Суффикс
0	4	2	GATAGACA$
1	3	1	ATAGACA$
2	0	0	TAGACA$
3	5	1	AGACA$
4	6	0	GACA$
5	7	1	ACA$
6	1	0	CA$
7	8	0	A$
8	-1	0	$

Рис. 6.10 ❖ Вычисление самого длинного общего префикса (LCP) с использованием суффиксного массива для строки `T = 'GATAGACA$'`

Теперь, используя `Phi[i]`, можно вычислить перемещаемый (permuted) LCP (самый длинный общий префикс). Несколько первых шагов выполнения этого алгоритма подробно описаны ниже. При `i = 0` имеем `Phi[0] = 4`. Это означает, что суффиксу 0 `'GATAGACA$'` предшествует суффикс 4 `'GACA$'` в упорядоченном суффиксном массиве. Первые два символа (`L = 2`) этих двух суффиксов совпадают, поэтому `PLCP[0] = 2`.

При `i = 1` известно, что как минимум `L-1 = 1` символ может совпадать, так как следующий суффикс в упорядоченной позиции будет иметь на один начальный символ меньше, чем текущий суффикс. Получаем `Phi[1] = 3`. Это означает, что суффиксу 1 `'ATAGACA$'` предшествует суффикс 3 `'AGACA$'` в упорядоченном суффиксном массиве. Мы видим, что в этих двух суффиксах действительно имеется совпадение по меньшей мере 1 символа (то есть мы не начинаем с `L = 0`, как в функции `computeLCP_slow()`, показанной выше, следовательно, этот метод более эффективен). Поскольку дальнейшее продвижение невозможно, получаем `PLCP[1] = 1`.

Процесс продолжается до $i = n - 1$ с исключением особого случая, когда Phi[i] = −1. Поскольку теорема PLCP утверждает, что L будет увеличиваться/уменьшаться не более n раз, этот этап выполняется за приблизительное время $O(n)$. Наконец, после завершения формирования массива PLCP можно поместить перемещаемый LCP обратно в правильную позицию. Показанный ниже код относительно короток.

```
void computeLCP()
{
  int i, L;
  Phi[SA[0]] = -1;                                          // значение по умолчанию
  for( i=1; i < n; i++ )                                  // вычисление Phi за время O(n)
    Phi[SA[i]] = SA[i-1];          // запомнить, какой суффикс следует за текущим суффиксом
  for( i = L = 0; i < n; i++ ) {              // вычисление перемещаемого LCP за время O(n)
    if( Phi[i] == -1 ) { PLCP[i] = 0; continue; }                    // особый случай
    while( T[i+L] == T[Phi[i]+L] )  L++;                 // L увеличивается максимум n раз
    PLCP[i] = L;
    L = max( L-1, 0 );                                     // L уменьшается максимум n раз
```

```
  }
  for( i=0; i < n; i++ )                                  // вычисление LCP за время O(n)
    LCP[i] = PLCP[SA[i]];                 // поместить перемещаемый LCP в правильную позицию
}
```

Поиск самой длинной повторяющейся подстроки O(n)

Если суффиксный массив уже вычислен за время $O(n \times \log n)$, а самый длинный общий префикс (LCP) между смежными суффиксами в упорядоченном суффиксном массиве определен за время $O(n)$, то можно найти длину самой длинной повторяющейся подстроки (LRS) в строке T за время $O(n)$.

Длина самой длинной повторяющейся подстроки – это просто наибольшее число в массиве LCP. В левой таблице на рис. 6.10 это соответствует суффиксному массиву, а LCP в строке `T = 'GATAGACA$'` – это наибольшее число 2 по индексу `i = 7`. Первые 2 символа соответствующего суффикса `SA[7]` (суффикс `0`) `'GA'`. Это и есть самая длинная повторяющаяся подстрока в строке `T`.

Поиск самой длинной общей подстроки O(n)

Без ущерба для обобщенной картины рассмотрим вариант только лишь с двумя строками. Используем пример с теми же строками, что и ранее в разделе о суффиксном дереве: T_1 = `'GATAGACA$'` и T_2 = `'CATA#'`. Для решения задачи о поиске самой длинной общей подстроки (LCS) с использованием суффиксного массива сначала необходимо объединить обе строки (отметим, что завершающие символы в этих строках должны быть различными) для получения строки `T = 'GATAGACA$CATA#'`. Затем вычисляется суффиксный массив и массив LCP для строки `T`, как показано в табл. 6.4.

***Таблица 6.4. Суффиксный массив, самый длинный общий префикс (LCP) и владелец* `T = 'GATAGACA$'`**

i	SA[*i*]	LCP[*i*]	Владелец	Суффикс
0	13	0	2	#
1	8	0	1	$CATA#
2	12	0	2	A#
3	7	1	1	A$CATA#
4	5	1	1	ACA$CATA#
5	3	1	1	AGACA$CATA#
6	10	1	2	ATA#
7	1	3	1	ATAGACA$CATA#
8	6	0	1	CA$CATA#
9	9	2	2	CATA#
10	4	0	1	GACA$CATA#
11	0	2	1	GATAGACA$CATA#
12	11	0	2	TA#
13	2	2	1	TAGACA$CATA#

Потом выполняется проход по смежным индексам за время $O(n)$. Если два смежных индекса принадлежат различным «владельцам» (это можно лег-

ко проверить[1], например можно определить, принадлежит ли суффикс `SA[i]` строке T_1, проверяя условие `SA[i]` < длины строки T_1), то просматривается массив LCP и проверяется, можно ли увеличить (на единицу) максимальный LCP, найденный к настоящему моменту. После одного прохода $O(n)$ появляется возможность определения самой длинной общей подстроки (LCS). В табл. 6.4 это происходит при `i = 7`, так как суффикс `SA[7]` = суффикс `1 = 'ATAGACA$CATA#'` (принадлежащий строке T_1) и предшествующий ему суффикс – это `SA[6]` = суффикс `10 = 'ATA#'` (принадлежащий строке T_2) имеют общий префикс длиной 3 `'ATA'`. Это и есть самая длинная общая подстрока (LCS).

Этот раздел и глава в целом завершаются указанием на доступность написанного нами исходного кода. Рекомендуем уделить некоторое время на глубокое изучение (и понимание) данного кода, который может оказаться весьма нетривиальным для тех, кто ранее не был знаком с суффиксным массивом.

Файл исходного кода: *ch6_04_sa.cpp/java*

Упражнение 6.6.5.1*. Попробуйте предложить некоторые возможные улучшения кода функции `stringMatching()`, приведенного в этом разделе.

Упражнение 6.6.5.2*. Сравните алгоритм Кнута–Морриса–Пратта (КМП) из раздела 6.4 с методом поиска совпадений в строках с использованием суффиксного массива. В каких случаях предпочтительнее использовать суффиксный массив для решения задач поиска совпадений в строках, а в каких лучше применять КМП или функции обработки строк из стандартной библиотеки?

Упражнение 6.6.5.3*. Обязательно решите все упражнения из раздела о приложениях суффиксного дерева, то есть **упражнения 6.6.3.1–6.6.3.6***, но с использованием суффиксного массива.

Задания по программированию, связанные с использованием суффиксного массива[2]

1. UVa 00719 – Glass Beads (минимальное лексикографическое «вращение» (перестановка)[3]; построить суффиксный массив $O(n \times \log n)$)
2. **UVa 00760 – DNA Sequencing *** (самая длинная общая подстрока в двух строках)

---

[1] Для трех и более строк такая проверка потребует большего количества операторов `if`.

[2] Вы можете попытаться решить эти задачи с использованием суффиксного дерева, но тогда вам придется самостоятельно научиться писать код алгоритма создания суффиксного дерева. Перечисленные здесь задания по программированию разрешимы с использованием суффиксного массива. Кроме того, рекомендуем обратить особое внимание на использование в наших примерах кода функции `gets()` для считывания строк ввода. Если вы пользуетесь функцией `scanf("%s")` или `getline()`, то не забывайте учитывать различия в символах конца строки в различных ОС DOS/Windows/Unix.

[3] Эту задачу можно решить, если объединить исходную строку с самой собой, сформировать суффиксный массив, затем найти в упорядоченном отсортированном суффиксном массиве первый суффикс, длина которого больше или равна n.

3. UVa 01223 – Editor (LA 3901, Seoul07, самая длинная повторяющаяся подстрока (или алгоритм КМП))
4. *UVa 01254 – Top 10* (LA 4657, Jakarta09, суффиксный массив + дерево отрезков)
5. **UVa 11107 – Life Forms *** (самая длинная общая подстрока > ½ исходных строк)
6. **UVa 11512 – GATTACA *** (самая длинная повторяющаяся подстрока)
7. SPOJ 6409 – Suffix Array (автор задачи: Феликс Халим (Felix Halim))
8. IOI 2008 – Type Printer (поиск в глубину с проходом по префиксному дереву (suffix trie))

6.7. Решения упражнений, не помеченных звездочкой

Решения на языке C упражнений из раздела 6.2

Упражнение 6.2.1.

a. Строка хранится как массив символов, завершающийся null-символом, например `char str[30*10+50], line[30+50];`. Это правильный практический прием для объявления массива с несколько большим размером, чем действительно требуется, чтобы избежать ошибки типа «выход из диапазона за границу массива» (off-by-one).
b. Для считывания строк ввода поочередно (по одной строке) мы используем[1] функцию `gets(line);` или `fgets(line, 40, stdin);` из стандартной библиотеки `string.h` (или `cstring`). Следует отметить, что функция `scanf("%s", line);` здесь неприменима, так как считывает только первое слово строки.
c. Сначала для создания пустой строки применяется функция `strcpy(str, "");`, затем считанные строки `line` объединяются в более длинную строку с помощью функции `strcat(str, line);`. Если текущая строка не является последней, то добавляется пробел к концу строки `str` с помощью функции `strcat(str, " ");`, чтобы последнее слово в этой строке не было случайно объединено с первым словом следующей строки.
d. Считывание строк ввода останавливается, когда `strncmp(line, ".....", 7) == 0`. Отметим, что функция `strncmp` сравнивает только первые *n* символов.

Упражнение 6.2.2.

a. Для поиска подстроки в относительно короткой строке (стандартная задача поиска совпадений в строках) можно просто воспользоваться библиотечной функцией. Можно использовать функцию `p = strstr(str, substr);`. Значением `p` будет `NULL`, если подстрока `substr` не найдена в строке `str`.
b. Если в строке `str` содержится несколько экземпляров подстроки `substr`, то можно использовать прием `p = strstr(str+pos, substr);`. Изначально `pos = 0`, то есть поиск выполняется с первого символа строки `str`. После

[1] Замечание: в действительности функция `gets()` небезопасна, поскольку не выполняет проверку размера входной строки на соответствие заданному диапазону.

обнаружения первого вхождения подстроки `substr` в строке `str` снова вычисляется выражение `p = strstr(str+pos, substr);`, но на этот раз `pos` является индексом (позицией) текущего вхождения подстроки `substr` в строке `str` с обязательным увеличением на единицу, чтобы найти следующее вхождение. Этот процесс повторяется до тех пор, пока не будет выполнено условие `p == NULL`. Это решение на языке C требует хорошего понимания системы адресации памяти в массиве языка C.

Упражнение 6.2.3. Во многих задачах обработки строк требуется итеративный последовательный проход по всем символам в строке `str`. Если строка `str` содержит *n* символов, то для такого прохода требуется время *O*(*n*). В обоих языках C/C++ можно воспользоваться функциями стандартной библиотеки `tolower(ch)` и `toupper(ch)`, объявленными в заголовочном файле *ctype.h*, для преобразования символа в нижний или верхний регистр соответственно. В стандартной библиотеке также имеются функции `isalpha(ch)` и `isdigit(ch)`, позволяющие узнать, является символ алфавитным `[A-Za-z]` или цифровым соответственно. Чтобы проверить, является ли символ гласной буквой, можно применить простой метод с подготовкой строки `vowel = "aeiou"`, затем проверять, является ли символ одним из пяти символов в строке `vowel`. Чтобы проверить, является ли символ согласным, нужно просто проверить, является ли символ алфавитным, но не гласным.

Упражнение 6.2.4. Объединенные решения на языках C и C++:

a) одним из самых простых способов токенизации строки является использование функции `strtok(str, delimiters);` в языке C;
b) затем полученные в результате токены (элементы) можно сохранить в структуре данных C++ `vector<string> tokens;`
c) можно воспользоваться реализацией алгоритма сортировки из библиотеки C++ STL `algorithm::sort` для сортировки вектора токенов `vector <string> tokens`. При необходимости можно преобразовать строку C++ `string` обратно в строку языка C, используя для этого метод `str.c_str()`.

Упражнение 6.2.5. См. решение на языке C++.

Упражнение 6.2.6. Считывать строку ввода символ за символом и вести инкрементальный подсчет символов, проверять наличие символа `'\n'`, обозначающего конец строки. Предварительное создание буфера фиксированного размера не является удачной идеей, так как автор задачи может определить неимоверно длинную строку, чтобы нарушить работу вашего кода.

Решения на языке C++ из раздела 6.2

Упражнение 6.2.1.

a. Можно воспользоваться классом `string`.
b. Можно использовать метод `cin.getline()` из библиотеки `string`.
c. Можно напрямую применить оператор `'+'` для объединения строк.
d. Можно напрямую применить оператор `'=='` для сравнения двух строк.

Упражнение 6.2.2.

a. Можно воспользоваться функцией (методом) `find` из класса `string`.

b. Тот же принцип, что и в решении на языке C. Можно устанавливать значение смещения (по строке) во втором параметре функции `find` из класса `string`.

Упражнение 6.2.3–4. Те же решения, что и на языке C.

Упражнение 6.2.5. Можно воспользоваться структурой данных C++ STL `map<string, int>` для сохранения и отслеживания частоты вхождений каждого слова. При каждом обнаружении нового токена (то есть строки) увеличивается на единицу соответствующая частота вхождения этого токена. В конце выполняется проход по всем токенам и определяется токен с максимальной частотой вхождений.

Упражнение 6.2.6. То же решение, что и на языке C.

Решения на языке Java из раздела 6.2

Упражнение 6.2.1.

a. Можно воспользоваться классом `String`, `StringBuffer` или `StringBuilder` (последний класс быстрее, чем `StringBuffer`).
b. Можно использовать метод `nextLine` из класса Java `Scanner`. Для ускорения ввода/вывода можно рассмотреть использование метода `readLine` из класса Java `BufferedReader`.
c. Можно воспользоваться методом `append` из класса `StringBuilder`. Не следует объединять строки языка Java с помощью оператора `'+'`, так как класс Java `String` является неизменяемым, поэтому такая операция связана с (очень) большими издержками (накладными расходами).
d. Можно использовать метод `equals` из класса Java `String`.

Упражнение 6.2.2.

a. Можно использовать метод `index0f` из класса `String`.
b. Тот же принцип, что и в решении на языке C. Можно устанавливать значение смещения (по строке) во втором параметре функции `index0f` из класса `String`.

Упражнение 6.2.3. Использовать классы Java `StringBuilder` и `Character` для этих операций.

Упражнение 6.2.4.

a. Можно использовать класс Java `StringTokenizer` или метод `split` из класса `String`.
b. Можно воспользоваться классом Java `Vector`, сформированным из строк `String`.
c. Можно использовать реализацию алгоритма сортировки Java `Collections.sort`.

Упражнение 6.2.5. Тот же принцип, что и в решении на языке C++. Можно использовать структуру данных Java `TreeMap<String, Integer>`.

Упражнение 6.2.6. Необходимо использовать метод `read` из класса Java `BufferedReader`.

Решения упражнений из других разделов

Упражнение 6.5.1.1. Другая схема оценки приведет к иному (глобальному) регулированию (применению редакционного расстояния). Если поставлена задача регулирования строк, то необходимо внимательно прочитать условия задачи, чтобы понять, какова требуемая цена (оценка) совпадения, несовпадения, операций вставки и удаления. После этого необходимо соответствующим образом адаптировать алгоритм.

Упражнение 6.5.1.2. Необходимо сохранять информацию о предшественниках (стрелки) во время вычисления по методу динамического программирования. Затем следовать по стрелкам, используя рекурсивный поиск с возвратом. См. раздел 3.5.1.

Упражнение 6.5.1.3. Для решения методом динамического программирования необходима только ссылка на предыдущую строку, поэтому можно воспользоваться приемом экономии памяти, используя лишь две строки: текущую и предыдущую. Новая сложность по памяти $O(\min(n, m))$, то есть нужно поместить строку меньшей длины в строку 2, чтобы каждая строка содержала меньшее количество столбцов (то есть использовала меньший объем памяти). Временная сложность этого решения остается равной $O(mn)$. Единственным недостатком данного метода, как и любого другого метода, использующего подобный прием экономии памяти, является невозможность воспроизведения оптимального решения. Если необходимо предъявить действительно оптимальное решение, то от приема экономии памяти придется отказаться. См. раздел 3.5.1.

Упражнение 6.5.1.4. Просто сосредоточьтесь на главной диагонали с шириной d. Сделав это, можно ускорить алгоритм Нидлмана–Вунша до $O(dn)$.

Упражнение 6.5.1.5. Снова используется алгоритм Кадана (Kadane's algorithm) (см. задачу о поиске максимальной суммы непрерывного подмассива в разделе 3.5.2).

Упражнение 6.5.2.1. `'pple'`.

Упражнение 6.5.2.2. Установить оценку для совпадения match = 0, несовпадения mismatch = 1, для операций вставки (insert) и удаления (delete) = «минус бесконечность». Но это решение неэффективно и не вполне естественно, так как мы можем просто воспользоваться алгоритмом $O(\min(n, m))$ для просмотра обеих строк 1 и 2 и подсчета в них различающихся символов.

Упражнение 6.5.2.3. Сводится к решению задачи поиска наибольшей увеличивающейся подпоследовательности (LIS) $O(n \times \log k)$. Здесь процесс сведения к задаче LIS не показан. Изобразите графически условия задачи и определите, как свести эту задачу к задаче LIS.

Упражнение 6.6.3.1. Шаблон 'CA' найден, шаблон 'CAT' не найден.

Упражнение 6.6.3.2. 'ACATTA'.

Упражнение 6.6.3.4. 'EVEN'.

6.8. Примечания к главе

При написании разделов о регулировании строк (редакционном расстоянии), самой длинной общей подпоследовательности, префиксного дерева, суффиксного дерева и суффиксного массива использовались материалы, любезно предоставленные адъюнкт-профессором Кеном Сун Вин Кином (A/P Sung Wing Kin, Ken) из Школы информатики (School of Computing) Национального университета Сингапура (National University of Singapore). Исходный материал был изложен в более теоретическом стиле, поэтому мы переработали его для изложения в стиле этой книги по олимпиадному программированию.

Раздел об основных приемах обработки строк (раздел 6.2) и о специализированных задачах обработки строк появился как обобщение нашего практического опыта по применению методов и решению задач, связанных с обработкой строк. Количество упражнений по программированию, приведенных в этом разделе, составляет около трех четвертей от всех прочих задач обработки строк, рассматриваемых в данной главе. Мы понимаем, что эти задачи не являются типичными заданиями студенческих (ICPC)/школьных (IOI) олимпиад, но они остаются неплохими упражнениями для улучшения ваших навыков программирования.

В разделе 6.4 рассматриваются решения с использованием библиотечных функций и один быстрый алгоритм (алгоритм Кнута–Морриса–Пратта, КМП) для решения задачи поиска совпадений в строках. Реализация алгоритма КМП будет полезной, если необходимо изменить основные требования к задаче поиска совпадений в строках, но сохранить высокую производительность. Мы уверены, что алгоритм КМП достаточно быстр для решения типичных олимпиадных задач поиска строки шаблона в длинной строке. Экспериментируя, мы пришли к такому выводу: реализация алгоритма КМП, представленная в этой книге, немного быстрее, чем встроенные библиотечные функции C `strstr`, C++ `string.find` и Java `String.index0f`. Если во время олимпиады требуется еще более быстрый алгоритм поиска совпадений в строках для одной более длинной строки и гораздо большего количества запросов, то мы рекомендуем воспользоваться суффиксным массивом, рассмотренным в разделе 6.8. Существуют и другие алгоритмы поиска совпадений в строках, которые здесь не рассматривались, например алгоритм Бойера–Мура (Boyer-Moor), Рабина–Карпа (Rabin-Karp), Ахо–Корасик (Aho-Corasick), конечные автоматы (Finite State Automata) и т. д. Заинтересованные читатели могут изучить эти алгоритмы самостоятельно.

Дополнением к обсуждению решений данного типа задач стало описание неклассических задач динамического программирования (DP) для строк в разделе 6.5. Мы не без оснований предполагаем, что классические задачи такого типа будут редко предлагаться на современных олимпиадах по программированию.

Главным побудительным мотивом для практической реализации суффиксного массива (раздел 6.6) стала статья *Suffix arrays – a programming contest approach* («Суффиксные массивы – методика применения на олимпиадах по программированию») [68]. Мы объединили и адаптировали многие примеры из этой статьи с нашим способом реализации суффиксного массива. В третьем издании книги мы пересмотрели концепцию завершающего символа в суф-

фиксном дереве и суффиксном массиве, так как это упрощало обсуждение. Настоятельно рекомендуем решить все упражнения по программированию из раздела 6.6, хотя их пока не так много. Это важная структура данных, которая становится все более распространенной и широко применяется на практике.

По сравнению с первыми двумя изданиями книги эта глава увеличилась в объеме, так же, как и глава 5. Тем не менее здесь пока еще не рассматривались некоторые другие задачи обработки строк: методы хеширования (hashing techniques) для решения некоторых типов задач обработки строк, задача поиска самой короткой общей суперстроки (shortest common superstring), алгоритм преобразования Барроуза–Уилера (Burrows-Wheeler transformation), суффиксный автомат (suffix automaton), базисное дерево (radix tree) и т. д.

Таблица 6.5

Статистические характеристики	Первое издание	Второе издание	Третье издание
Количество страниц	10	24 (+140 %)	35 (+46 %)
Описанные задания	4	24 (+500 %)	17 + 16* = 33 (+38 %)
Упражнения по программированию	54	129 (+138 %)	164 (+27 %)

Количество упражнений по программированию в каждом разделе показано в табл. 6.6.

Таблица 6.6

Раздел	Название	Количество	% в главе	% в книге
6.3	Специализированные задачи обработки строк	126	77	8
6.4	Поиск совпадений в строках	13	8	1
6.5	Обработка строк с применением динамического программирования	17	10	1
6.6	Префиксное дерево / Суффиксное дерево / Суффиксный массив	8	5	≈1

Рис. 6.11 ❖ Слева д-р Билл Паучер, справа Стивен

Глава 7

(Вычислительная) Геометрия

«Не знающий геометрии да не войдет сюда».

– Надпись на входе
в Академию Платона в Афинах

7.1. Обзор и мотивация

Вычислительная[1] геометрия – это еще одна тема, которая часто появляется на олимпиадах по программированию. Почти все комплекты задач международных студенческих олимпиад по программированию (ICPC) содержат по меньшей мере одну геометрическую задачу. Если вам повезет, то потребуется геометрическое решение, которое вам уже было известно заранее. Обычно вы изображаете (строите) геометрический объект (или несколько геометрических объектов), затем выводите решение из некоторых хорошо известных геометрических формул. Но многие геометрические задачи являются вычислительными, то есть требуют применения определенного сложного алгоритма (или даже нескольких алгоритмов).

На международных олимпиадах по информатике для школьников (IOI) наличие заданий, связанных с геометрией, зависит от выбора Научного организационного комитета (Scientific Committee) в каждом году. В последние годы (2009–2012) в комплект заданий на международных олимпиадах по информатике для школьников не входили чисто геометрические задачи. Но раньше [67] на каждой олимпиаде IOI предлагалось одно или два задания, связанных с геометрией.

Мы обратили внимание на то, что такие задачи, связанные с геометрией, обычно не пытаются решить в начальном интервале времени, отведенного участникам олимпиады, из стратегических соображений, поскольку решения задач, так или иначе связанных с геометрией, имеют более низкую вероятность получения оценки «зачтено» (Accepted – AC) в рамках установленного лимита времени по сравнению с решениями других типов задач из комплекта заданий, например задач полного поиска или задач динамического програм-

---

[1] Мы особо отмечаем различие между чисто геометрическими задачами и задачами вычислительной геометрии. Чисто геометрические задачи обычно можно решить вручную (используя карандаш и бумагу). Задачи вычислительной геометрии, как правило, требуют выполнения некоторого алгоритма с использованием компьютера для получения решения.

мирования. Обычно при попытках решения геометрических задач возникают следующие проблемы:

- многие геометрические задачи содержат один, а чаще несколько каверзных крайних случаев, например: что, если линии являются вертикальными (бесконечный градиент)? что, если точки являются коллинеарными? что, если многоугольник невыпуклый? что, если выпуклая оболочка, определяемая множеством точек, является самим этим множеством точек? и т. д. Таким образом, как правило, лучше всего проверить геометрическое решение вашей команды на большом количестве крайних случаев, прежде чем отправлять решение в тестирующую систему;
- существует вероятность возникновения ошибок и погрешностей при вычислениях с плавающей точкой, которые даже при использовании «правильного» алгоритма приводят к получению оценки «неверный ответ» (Wrong Answer – WA);
- для решения геометрических задач обычно необходимо утомительное (зачастую неприятное рутинное) кодирование.

Эти причины заставляют участников соревнований считать, что драгоценные минуты следует тратить на решение других типов задач, а не на попытки решить геометрическую задачу, у которой меньше шансов на получение положительной оценки.

Еще одной не столь существенной причиной отказа от попыток решения геометрических задач является недостаточная подготовка участников соревнований:

- участники забывают некоторые важные основные формулы или не могут вывести требуемые (более сложные) формулы из основных формул;
- перед соревнованиями участники не подготовили тщательно проработанные библиотечные функции, поэтому их попытки написания кода таких функций в условиях ограниченного времени и соревновательного стресса приводят к ошибке, а в большинстве случаев – к нескольким ошибкам[1]. На международных студенческих олимпиадах по программированию самые лучшие команды обычно заполняют значительную часть своего «бумажного» справочника (который они могут взять с собой в зал, где проводится соревнование) огромным количеством геометрических формул и кодом соответствующих библиотечных функций.

Таким образом, главная цель этой главы – поощрить участников к увеличению количества попыток и, соответственно, зачтенных (AC) правильных решений геометрических задач на олимпиадах по программированию. Изучите эту главу, чтобы узнать некоторые основные принципы решения геометрических задач, предлагаемых на олимпиадах ICPC и IOI. В этой главе всего лишь два раздела.

---

[1] В качестве подтверждения: библиотечный код для обработки точек, линий, окружностей, треугольников и многоугольников, приведенный в этой главе, требует нескольких итерационных сеансов по исправлению ошибок, чтобы окончательно убедиться в том, что подавляющее большинство (обычно трудно обнаруживаемых) ошибок и особых случаев обработано корректно.

В разделе 7.2 представлены многие (все описать невозможно) геометрические термины и определения[1], а также основные формулы для нульмерных, одномерных, двумерных и трехмерных геометрических объектов, часто используемых в заданиях олимпиад по программированию. Этот раздел можно использовать как краткий справочник, когда участникам достается геометрическая задача, а они не уверены в необходимости применения какой-либо методики или забыли некоторые основные формулы.

В разделе 7.3 рассматриваются некоторые алгоритмы для двумерных многоугольников. Также представлено несколько эффективных предварительно написанных библиотечных подпрограмм, которые могут выделить успешные команды на фоне среднего уровня команд (участников). Это алгоритмы для проверки, является ли многоугольник выпуклым, для определения нахождения точки внутри или вне многоугольника, метод отсечения части многоугольника прямой линией, поиск выпуклой оболочки заданного множества точек и т. д.

Реализации формул и алгоритмов вычислительной геометрии, рассматриваемые в этой главе, используют следующие методики для повышения вероятности получения вердикта «зачтено» (AC):

1) выделены особые случаи, учитывая которые, можно найти и/или выбрать реализацию, сокращающую количество таких особых случаев;
2) мы пытаемся избежать выполнения операций с плавающей точкой (операций деления, извлечения квадратного корня и любых других операций, при которых возможно возникновение числовых погрешностей) и работаем с точными целочисленными значениями, когда это возможно (то есть с целочисленными операциями сложения, вычитания, умножения);
3) если действительно необходима обработка значений с плавающей точкой, то обязательно выполняется проверка равенства значений с плавающей точкой следующим способом: `fabs(a-b) < EPS`, где `EPS` – малое число[2], например `1e-9`, вместо простой проверки на равенство `a == b`. Если необходима проверка числа с плавающей точкой $x \geqslant 0$, то применяется правило `x > -EPS` (аналогично для проверки $x \leqslant 0$ применяется правило `x < EPS`).

7.2. Основные геометрические объекты и библиотечные функции для них

7.2.1. Нульмерные объекты: точки

1. Точка (point) – основной конструктивный элемент для создания геометрических объектов с более высокими степенями измерений. В двумер-

[1] В олимпиадах ICPC и IOI участвуют люди различных национальностей из разных стран. Но поскольку официальным языком олимпиад является английский, каждый участник должен хорошо знать геометрические термины не только на родном, но и на английском языке.

[2] Если не указано другое значение, то это число `1e-9` является значением по умолчанию для критерия `EPS`(ilon), используемого в данной главе.

ном евклидовом[1] пространстве точки обычно представлены с помощью структуры языка C/C++ (или класса языка Java) с двумя членами[2]: координатами *x* и *y*, отсчитываемыми от начала координат, то есть от (0,0). Если в описании задачи используются целочисленные координаты, то они определяются как целочисленные значения `int`, в противном случае применяются значения типа `double`. Для обобщения мы используем в данной книге версию структуры `struct point` с плавающей точкой. Конструкторы по умолчанию и конструкторы, определяемые пользователем, могут использоваться для (небольшого) упрощения кодирования в дальнейшем.

```
// struct point_i { int x, y; };                    // основная простейшая форма,
                                                      // минималистичный вариант
struct point_i {
  int x, y;        // везде, где это возможно, работаем только со структурой point_i
  point_i() { x = y = 0; }                            // конструктор по умолчанию
  point_i( int _x, int _y ) : x(_x), y(_y) {}         // конструктор, определенный
                                                                  // пользователем
};

struct point {                  // используется, если без вещественных чисел нельзя
  double x, y;
  point() { x = y = 0.0; }                            // конструктор по умолчанию
  point( double _x, double _y ) : x(_x), y(_y) {}     // конструктор, определенный
                                                                  // пользователем
};
```

2. Иногда необходима сортировка точек. Это легко сделать, если перегрузить оператор «меньше» внутри структуры `struct point` и воспользоваться библиотекой методов сортировки.

```
struct point {
  point() { x = y = 0.0; }
  point( double _x, double _y ) : x(_x), y(_y) {}
  bool operator < (point other) const {          // перегруженный оператор "меньше"
    if( fabs( x - other.x ) > EPS )                    // необходимо для сортировки
      return x < other.x;                         // первый критерий: по координате x
    return y < other.y; }                         // второй критерий: по координате y
};

// в функции int main() предполагаем, что уже существует заполненный
// вектор vector<point> P
  sort( P.begin(), P.end()      // здесь работает перегруженный оператор сравнения,
                                                          // определенный выше
```

3. Иногда необходимо определить, равны ли две точки. Такая операция легко реализуется с помощью перегрузки оператора «равно» в структуре `struct point`.

---

[1] В сокращенной упрощенной форме евклидово двумерное и трехмерное пространство обозначается аббревиатурами 2D и 3D соответственно, которые часто встречаются и в обычной жизни.

[2] Если добавить еще один член *z*, то получится представление точки в трехмерном евклидовом пространстве.

```
struct point {
  double x, y;
  point() { x = y = 0.0; }
  point( double _x, double _y ) : x(_x), y(_y) {}
  // используется EPS (1e-9) при проверке равенства двух точек с координатами
  // с плавающей точкой
  bool operator == (point other) const {
    return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); }
};

// в функции int main()
  point P1(0,0), P2(0,0), P3(0,1);
  printf( "%d\n", P1 == P2 );                                    // true - истина
  printf( "%d\n", P1 == P3 );                                     // false - ложь
```

4. Можно вычислить евклидово расстояние[1] между двумя точками, используя приведенную ниже функцию.

```
double dist( point p1, point p2 )                          // евклидово расстояние
{
                                  // hypot( dx, dy ) return sqrt( dx * dx + dy * dy )
  return hypot( p1.x - p2.x, p1.y - p2.y );     // возвращается значение типа double
}
```

5. Можно повернуть точку на угол θ[2] против часовой стрелки относительно начала координат (0,0), воспользовавшись для этого матрицей поворота, как показано на рис. 7.1.

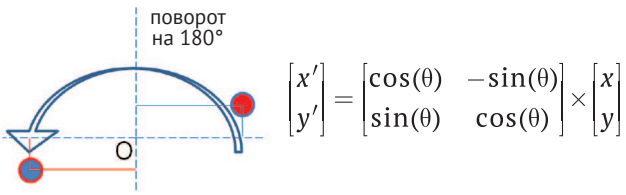


Рис. 7.1 ❖ Поворот точки (10,3) на 180° против часовой стрелки относительно начала координат (0,0)

```
// поворот точки p на угол theta градусов против часовой стрелки относительно
// начала координат (0,0)
point rotate( point p, double theta )
{
```

---

[1] Евклидово расстояние между двумя точками – это просто расстояние, которое можно измерить линейкой. С алгоритмической точки зрения это расстояние можно вычислить по формуле Пифагора, которую мы снова увидим в подразделе о треугольниках. Здесь мы просто используем библиотечную функцию.

[2] Обычно для людей более привычно работать с градусами, но многие математические функции в большинстве языков программирования (в частности, C/C++/Java) работают с радианами. Для преобразования величины угла в градусах в радианы необходимо умножить значение в градусах на (π/180.0). Для преобразования радианов в градусы необходимо умножить значение в радианах на (180.0/π).

```
  double rad = DEG_to_RAD( theta );                  // умножить theta на PI / 180.0
  return point( p.x * cos(rad) - p.y * sin(rad),
                p.x * sin(rad) + p.y * cos(rad) );
}
```

Упражнение 7.2.1.1. Вычислить евклидово расстояние между точками (2,2) и (6,5).

Упражнение 7.2.1.2. Выполнить поворот точки (10,3) на 90° против часовой стрелки относительно начала координат. Какими стали значения новых координат точки после поворота? (Их можно легко вычислить вручную.)

Упражнение 7.2.1.3. Выполнить поворот точки (10,3) на 90° против часовой стрелки относительно начала координат. Какими стали значения новых координат точки после поворота? (В этом случае вам потребуется калькулятор и матрица поворота.)

7.2.2. Одномерные объекты: прямые

1. Прямая (line) в двумерном евклидовом пространстве – это набор точек, координаты которых соответствуют заданному линейному уравнению $ax + by + c = 0$. Все последующие функции в этом подразделе предполагают, что в приведенном линейном уравнении коэффициент $b = 1$ для невертикальных прямых и $b = 0$ для вертикальных прямых, если не указано что-либо иное. Обычно прямые представлены структурой языка C/C++ (или классом языка Java) с тремя членами: коэффициентами a, b, c приведенного выше линейного уравнения.

   ```
   struct line { double a, b, c; };              // один из способов представления линии
   ```

2. Можно вывести требуемое линейное уравнение, если заданы, как минимум, две точки, через которые проходит соответствующая прямая, с помощью следующей функции.

   ```
   // ответ сохраняется в третьем параметре (переданном по ссылке)
   void pointsToLine( point p1, point p2, line &l )
   {
     if( fabs(p1.x - p2.x) < EPS ) {                    // вертикальная линия допустима
       l.a = 1.0; l.b = 0.0; l.c = -p1.x;                       // значения по умолчанию
     } else {
       l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
       l.b = 1.0;                                  // ВАЖНО: мы приводим значение b к 1.0
       l.c = -(double)(l.a * p1.x) - p1.y;
     }
   }
   ```

3. Можно проверить, являются ли две прямые параллельными, сравнивая их коэффициенты a и b на равенство. Также можно проверить две линии на совпадение, если эти линии являются параллельными и их коэффициенты c равны (то есть все три коэффициента a, b, c равны). Напомним,

что в приведенной выше реализации значение коэффициента *b* приводится к 0.0 для всех вертикальных прямых и на 1.0 для всех невертикальных прямых.

```
bool areParallel( line l1, line l2 )                // проверка коэффициентов a и b
{
  return (fabs(l1.a - l2.a) < EPS) && (fabs(l1.b - l2.b) < EPS);
}

bool areSame( line l1, line l2 )                  // также проверяется коэффициент c
{
  return areParallel( l1, l2 ) && (fabs(l1.c - l2.c) < EPS);
}
```

4. Если две прямые[1] не параллельны (следовательно, не совпадают), то они пересекаются в некоторой точке. Точку пересечения (*x*, *y*) можно найти, решив систему двух линейных алгебраических уравнений[2] с двумя неизвестными: $a_1x + b_1y + c_1 = 0$ и $a_2x + b_2y + c_2 = 0$.

```
// возвращает true (+ точку пересечения), если две линии пересекаются
bool areIntersect( line l1, line l2, point &p )
{
  if( areParallel( l1, l2 ) ) return false;                // линии не пересекаются
  // решение системы 2 линейных алгебраических уравнений с 2 неизвестными
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // особый случай: проверка линии на вертикальность, чтобы избежать деления на ноль
  if( fabs(l1.b) > EPS ) p.y = -(l1.a * p.x + l1.c);
  else                   p.y = -(l2.a * p.x + l2.c);
  return true;
}
```

5. Отрезок прямой линии (line segment) – это подмножество прямой с двумя конечными точками и конечной длиной.
6. Вектор (vector)[3] – это отрезок прямой линии (следовательно, вектор имеет две конечные точки и длину/модуль) с определенным направлением. Обычно[4] вектор представлен в виде структуры struct языка C/C++ (или класса языка Java) с двумя членами: *x* и *y* задают координаты вектора. При необходимости вектор может быть промасштабирован, то есть умножен на константу, путем умножения обеих координат на эту константу.
7. Можно перенести точку с помощью вектора, так как координаты вектора задают смещение точки по осям x и y.

```
struct vec                  // имя vec отличает геометрический вектор от типа vector
                                                                  // из библиотеки STL
```

---

[1] Чтобы избежать путаницы, необходимо различать пересечение прямых линий и пересечение *отрезков* прямых линий.

[2] См. раздел 9.9, где описано обобщенное решение для любой системы линейных уравнений.

[3] Не следует путать геометрический вектор со структурой данных C++ STL `vector` или классом Java `Vector`.

[4] Другой приемлемой стратегией проектирования является объединение `struct point` со структурой `struct vec`, поскольку они одинаковы.

```
{
  double x, y;
  vec( double _x, double _y ) : x(_x), y(_y) {}
};

vec toVec( point a, point b )              // преобразование 2 точек  в вектор a->b
{
  return vec( b.x - a.x, b.y - a.y );
}

vec scale( vec v, double s )            // неотрицательное значение s = [<1..1..>1]
{
  return vec( v.x * s, v.y * s );
}

point translate( point p, vec v )           // преобразование (перемещение) точки p
                                                                  // по вектору v
{
  return point( p.x + v.x, p.y + v.y );
}
```

8. Если задана точка p и прямая l (определенная двумя точками a и b), то можно вычислить минимальное расстояние от точки p до прямой l, сначала определяя положение точки c на прямой l как наиболее близкой точки к точке p (см. рис. 7.2, слева), а затем вычислить евклидово расстояние между точками p и c. Можно рассматривать точку c как точку a, перемещенную на вектор ab, масштабированный в u раз, то есть $c = a + u \times ab$. Для получения u применяется скалярная проекция вектора ap на вектор ab с использованием скалярного произведения (см. обозначенный пунктиром вектор $ac = u \times ab$ на рис. 7.2 слева). Сокращенная реализация этого решения показана ниже.

```
double dot( vec a, vec b ) { return (a.x * b.x + a.y * b.y); }

double norm_sq( vec v ) { return v.x * v.x + v.y * v.y; }

// возвращает расстояние от точки p до прямой, определенной двумя точками a и b
// (a и b обязательно должны быть различными точками)
// самая близкая точка сохраняется в 4-м параметре (переданном по ссылке)
double distToLine( point p, point a, point b, point &c )
{
  // вычисление по формуле: c = a + u * ab
  vec ap = toVec( a, p ), ab = toVec( a, b );
  double u = dot( ap, ab ) / norm_sq( ab );
  c = translate( a, scale( ab, u ) );          // преобразование (перемещение) a в c
  return dist( p, c );                  // евклидово расстояние между точками p и c
}
```

   Отметим, что это не единственный способ получения требуемого ответа. Предлагается выполнить **упражнение 7.2.2.10** для реализации другого способа.

9. Если вместо прямой задан отрезок (определяемый конечными точками a и b), то при вычислении минимального расстояния от точки p до отрезка ab также необходимо непременно рассмотреть два особых случая

расположения конечных точек *a* и *b* этого отрезка (см. рис. 7.2, среднее изображение). Реализация очень похожа на функцию distToLine, приведенную выше.

```
// возвращает расстояние от точки p до отрезка ab, определенного двумя точками a и b
// (допускается и особый случай a == b)
// самая близкая точка сохраняется в 4-м параметре (переданном по ссылке)
double distToLineSegment( point p, point a, point b, point &c )
{
  vec ap = toVec( a, p ), ab = toVec( a, b );
  double u = dot( ap, ab ) / norm_sq( ab );
  if( u < 0.0 ) {                                          // ближе к точке a
    c = point( a.x, a.y );
    return dist(p,a);                              // евклидово расстояние между p и a
  }
  if( u > 1.0 ) {                                          // ближе к точке b
    return dist(p,b);                              // евклидово расстояние между p и b
  return distToLine( p, a, b, c );                   // вычислить функцию distToLine,
                                                     // определенную выше
}
```

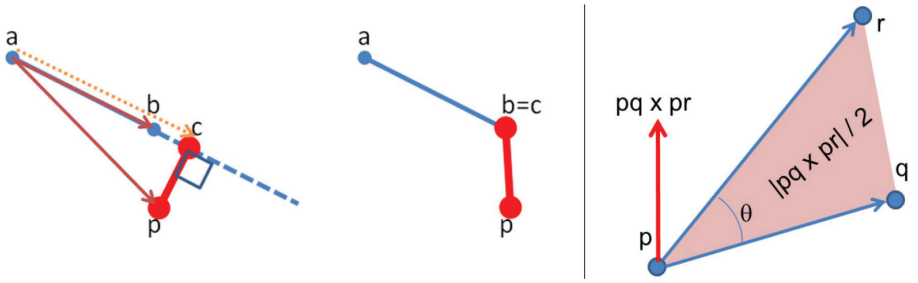


Рис. 7.2 ❖ Расстояние до прямой (слева) и до отрезка (в середине); векторное произведение (справа)

10. Можно вычислить угол aob по заданным трем точкам: *a*, *o*, *b*, – используя скалярное произведение[1]. Так как $oa \cdot ob = |oa| \times |ob| \times \cos(\theta)$, получаем $\arccos(oa \cdot ob/(|oa| \times |ob|))$.

```
double angle( point a, point o, point b )  // возвращает величину угла aob в радианах
{
  vec oa = toVector( o, a ), ob = toVector( o, b );
  return acos( dot(oa,ob) / sqrt( norm_sq(oa) * norm_sq(ob) ) );
}
```

11. Если задана линия, определяемая двумя точками *p* и *q*, то можно узнать, находится ли точка *r* слева или справа от этой линии или эти три точки *p*, *q*, *r* являются коллинеарными, то есть лежат на одной прямой. Это можно определить с помощью векторного произведения. Пусть *pq* и *pr* – два

[1] В вычисляемом выражении acos – это имя функции C/C++ для математической функции арккосинус arccos.

вектора, заданных с помощью этих трех точек. Векторное произведение $pq \times pr$ в трехмерном пространстве – это вектор, перпендикулярный обоим векторам pq и pr. Длина этого вектора равна площади параллелограмма, образованного исходными векторами[1]. Если длина полученного вектора положительна / равна нулю /отрицательна, то можно утверждать, что $p \rightarrow q \rightarrow r$ – поворот влево / все точки коллинеарны / поворот вправо соответственно (см. рис. 7.2, справа). Проверка на поворот влево более известна как тест на поворот против часовой стрелки (CCW – Counter Clockwise).

```
double cross( vec a, vec b ) { return a.x * b.y - a.y * b.x; }

// примечание: для приема коллинеарных точек необходимо изменить '> 0'
// возвращает true, если точка r находится слева от линии pq
bool ccw( point p, point q, point r )
{
  return cross( toVec(p,q), toVec(p,r) ) > 0;
}

// возвращает true, если точка r находится на линии pq
bool collinear( point p, point q, point r )
{
  return fabs( cross( toVec(p,q), toVec(p,r) ) ) < EPS;
}
```

Файл исходного кода: *ch7_01_points_lines.cpp/java*

Упражнение 7.2.2.1. Прямую линию также можно описать следующим математическим выражением: $y = mx + c$, где m – «градиент», или иначе – «угловой коэффициент» этой линии, а c – ордината точки пересечения этой линии с осью ординат.

Какая форма записи лучше (каноническая $ax + by + c = 0$ или с использованием углового коэффициента и ординаты точки пересечения $y = mx + c$)? Почему?

Упражнение 7.2.2.2. Вычислить коэффициенты уравнения прямой линии, которая проходит через две точки (2, 2) и (4, 3).

Упражнение 7.2.2.3. Вычислить коэффициенты уравнения прямой линии, которая проходит через две точки (2, 2) и (2, 4).

Упражнение 7.2.2.4. Предположим, что необходимо использовать другое уравнение прямой линии: $y = mx + c$. Продемонстрируйте, как вычисляются коэффициенты этого требуемого линейного уравнения, если заданы две точки, через которые проходит соответствующая прямая. Координаты точек принимаются равными (2, 2) и (2, 4), как в **упражнении 7.2.2.3**. Возникают ли при этом вычислении какие-либо проблемы?

[1] Следовательно, площадь треугольника pqr равна половине площади этого параллелограмма.

Упражнение 7.2.2.5. Уравнение прямой можно также определить, если задана одна точка и угловой коэффициент наклона этой прямой. Покажите, как определяется уравнение прямой, если задана точка и коэффициент угла наклона.

Упражнение 7.2.2.6. Переместить точку *c* (3, 2) по вектору *ab*, определенному двумя точками: *a* (2, 2) и *b* (4, 3). Вычислить новые координаты точки *c*.

Упражнение 7.2.2.7. Аналогично **упражнению 7.2.2.6**, но в этом случае длина вектора *ab* уменьшена наполовину. Вычислить новые координаты точки *c*.

Упражнение 7.2.2.8. Аналогично **упражнению 7.2.2.6**, но после перемещения повернуть полученную точку на 90° против часовой стрелки относительно начала координат. Вычислить итоговые координаты точки *c*.

Упражнение 7.2.2.9. Повернуть точку *c* (3, 2) на 90° против часовой стрелки относительно начала координат, затем переместить полученную точку по вектору *ab*, который определен в **упражнении 7.2.2.6**. Вычислить итоговые координаты точки *c*. Похож ли полученный результат на результат выполнения предыдущего **упражнения 7.2.2.8**? Какой вывод можно сделать из сравнения полученных результатов?

Упражнение 7.2.2.10. Повернуть точку *c* (3, 2) на 90° против часовой стрелки, но на этот раз относительно точки *p* (2, 1) (отметим, что точка *p* не является началом координат). Совет: необходимо переместить точку.

Упражнение 7.2.2.11. Можно вычислить положение точки *c* на прямой *l*, наиболее близкое к точке *p*, выполнив поиск другой прямой *l′*, которая перпендикулярна прямой *l* и проходит через точку *p*. Самая близкая точка с является точкой пересечения прямых *l* и *l′*. Но как построить прямую, перпендикулярную прямой *l*? Существуют ли особые случаи, заслуживающие внимания?

Упражнение 7.2.2.12. Заданы точка *p* и прямая *l* (определенная двумя точками *a* и *b*). Показать, как вычисляется положение точки *r*, полученной зеркальным отражением точки *p* относительно прямой *l*.

Упражнение 7.2.2.13. Заданы три точки: *a* (2, 2), *o* (2, 4), *b* (4, 3). Вычислить величину угла *aob* в градусах.

Упражнение 7.2.2.14. Определить, находится ли точка *r* (35, 30) слева, справа или на прямой, проходящей через две точки *p* (3, 7) и *q* (11, 13).

7.2.3. Двумерные объекты: окружности

1. Окружность (circle) с центром в точке с координатами (a, b) в двумерном евклидовом пространстве с радиусом r – это множество (геометрическое место) всех точек (x,y), таких, что $(x - a)^2 + (y - b)^2 = r^2$.
2. Для проверки нахождения точки внутри, снаружи или в точности на линии окружности можно воспользоваться приведенной ниже функцией. Для создания версии, работающей со значениями с плавающей точкой, внесите небольшие изменения в код этой функции.

```
int insideCircle( point_i p, point_i c, int r )                    // версия, работающая
                                                                   // с целочисленными значениями
{
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;          // только целочисленные значения
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;                       // внутри / на линии
                                                                   // окружности / снаружи
}
```

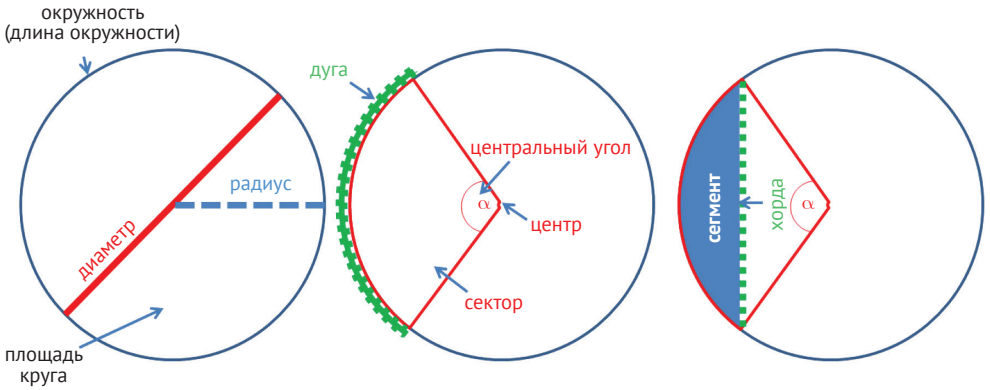


Рис. 7.3 ❖ Окружности

3. Константа пи (π) – это отношение длины любой окружности к ее диаметру. Чтобы избежать ошибок (погрешностей) при вычислениях, можно воспользоваться самым «безопасным» значением в условиях олимпиад по программированию, если постоянное число π не определено в условии задачи: `pi = acos(-1.0)` или `pi = 2 * acos(0.0)`.
4. Окружность радиуса r имеет диаметр $d = 2 \times r$ и длину окружности $c = 2 \times \pi \times r$.
5. Круг с радиусом r имеет площадь $A = \pi \times r^2$.
6. Дуга (arc) окружности определяется как непрерывная часть линии окружности с длиной c. Если задан центральный угол α (угол с вершиной в центре окружности, см. рис. 7.3, в середине) в градусах, то длину соответствующей дуги можно вычислить по формуле $\alpha/360.0 \times c$.
7. Хорда (chord) окружности определяется как отрезок прямой, конечные точки которого лежат на этой окружности[1]. Для окружности с радиусом r и центральным углом α в градусах (см. рис. 7.3, справа) существует соответствующая хорда с длиной, вычисляемой по формуле $\text{sqrt}(2 \times r^2 \times (1 - \cos(\alpha)))$. Эту формулу можно вывести из теоремы косинусов – см. формулировку данной теоремы в подразделе, в котором рассматриваются треугольники. Другой способ вычисления длины хорды при заданном радиусе r и центральном угле α – использование тригонометрических формул: $2 \times r \times \sin(\alpha/2)$. Тригонометрия также рассматривается в одном из следующих подразделов.

[1] Диаметр является самой длинной хордой любой окружности.

8. Сектор (sector) круга определяется как область круга, заключенная между двумя радиусами и дугой, ограниченной этими радиусами. Круг с площадью A и центральным углом α (в градусах; см. рис. 7.3, в середине) содержит соответствующий сектор с площадью $\alpha/360.0 \times A$.
9. Сегмент (segment) круга определяется как область круга, ограниченная хордой и дугой, лежащей между конечными точками этой хорды (см. рис. 7.3, справа). Площадь сегмента можно вычислить с помощью вычитания из площади соответствующего сектора площади равнобедренного треугольника со сторонами r, r и хорды.
10. Если заданы две точки на окружности ($p1$ и $p2$) и ее радиус r, можно определить положение центров ($c1$ и $c2$) двух окружностей, которые можно построить по этим данным (см. рис. 7.4). Исходный код приведен в **упражнении 7.2.3.1** ниже.

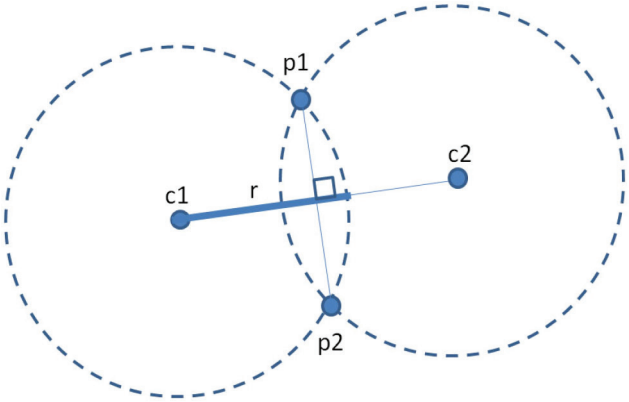


Рис. 7.4 ❖ Окружности, построенные по двум точкам и радиусу

Файл исходного кода: *ch7_02_circles.cpp/java*

Упражнение 7.2.3.1. Опишите, что вычисляется в исходном коде, приведенном ниже.

```
bool circle2PtsRad( point p1, point p2, double r, point &c )
{
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if( det < 0.0 ) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true;
}                       // для получения координат другого центра поменять местами p1 и p2
```

7.2.4. Двумерные объекты: треугольники

1. Треугольник (triangle) – это многоугольник с тремя вершинами и тремя сторонами (ребрами).
   Существует несколько типов треугольников:
   – равносторонний (equilateral): три стороны (ребра) равной длины и все внутренние углы равны 60°;
   – равнобедренный (isosceles): две стороны имеют одинаковую длину и два внутренних угла равны;
   – неравносторонний (scalene): все стороны имеют различную длину;
   – прямоугольный (right): величина одного из внутренних углов равна 90° (прямой угол).
2. Площадь треугольника с основанием b и высотой h определяется по формуле $A = 0.5 \times b \times h$.
3. Треугольник с тремя сторонами a, b, c имеет периметр $P = a + b + c$ и полупериметр $p = 0.5 \times P$.
4. Для треугольника с тремя сторонами a, b, c и полупериметром p площадь вычисляется по формуле $S = \text{sqrt}(p \times (p - a) \times (p - b) \times (p - c))$, которая называется формулой Герона.

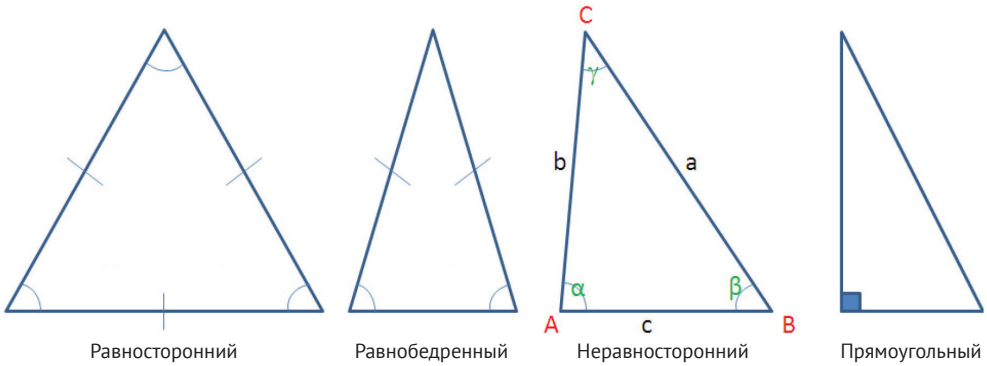


Рис. 7.5 ❖ Треугольники

5. В треугольник с площадью S и полупериметром p можно вписать окружность (inscribed circle/incircle) с радиусом = S/p.

```
double rInCircle( double ab, double bc, double ca )
{
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca));
}

double rInCircle( point a, point b, point c)
{
  return rInCircle( dist(a,b), dist(b,c), dist(c,a) );
}
```

6. Центр вписанной окружности является точкой пересечения биссектрис углов треугольника (см. рис. 7.6, слева). Центр вписанной окружности можно получить, если известны биссектрисы двух углов и можно найти точку их пересечения. Реализация приведена ниже.

```
// Предположение: требуемые функции обработки точек/линий уже написаны.
// Возвращается 1, если найден центр вписанной окружности inCircle, иначе
// возвращается 0.
// Если эта функция возвращает 1, то точка ctr является центром вписанной
// окружности, а радиус тот же самый, что и при вычислении функции rInCircle.
int inCircle( point p1, point p2, point p3, point &ctr, double &r )
{
  r = rInCircle( p1, p2, p3 );
  if( fabs(r) < EPS ) return 0;   // невозможно вычислить центр вписанной окружности

  line l1, l2;                    // для вычисления биссектрис двух внутренних углов
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine( p1, p, l1 );

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine( p2, p, l2 );

  areIntersect( l1, l2, ctr );            // вычисление точки пересечения биссектрис
  return 1;
}
```

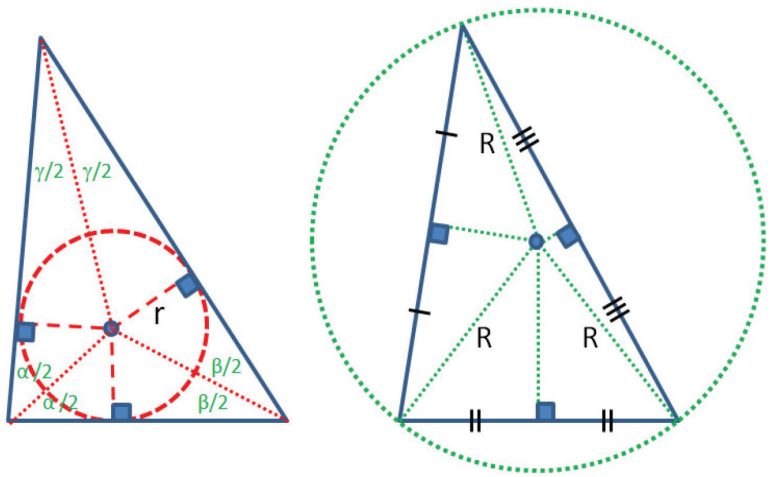


Рис. 7.6 ❖ Треугольник с вписанной и описанной окружностями

7. Вокруг треугольника с тремя сторонами a, b, c и площадью S можно описать окружность (circumscribed/circumcircle circle) с радиусом $R = a \times b \times c / (4 \times S)$.

```
double rCircumCircle( double ab, double bc, double ca )
{
```

```
  return ab * bc * ca / (4.0 * area(ab, bc, ca));
}

double rCircumCircle( point a, point b, point c )
{
  return rCircumCircle( dist(a,b), dist(b,c), dist(c,a) );
}
```

8. Центром описанной окружности является точка пересечения перпендикуляров к сторонам треугольника, проведенных через середины этих сторон (см. рис. 7.6, справа).
9. Для проверки того, что три отрезка длиной a, b, c могут образовать треугольник, можно просто выполнить проверку на неравенство треугольника: $(a + b > c)$ && $(a + c > b)$ && $(b + c > a)$.
   Если результат ложен (false), то эти три отрезка не могут сформировать треугольник.
   Если длины трех отрезков отсортированы, при этом a – наименьшая длина, а c – наибольшая длина, то можно выполнить только одну проверку $(a + b > c)$.
10. При изучении треугольника не следует забывать о тригонометрии, которая описывает отношения между сторонами и углами треугольника.
    В тригонометрии теорема косинусов (формула косинусов, правило косинусов) – это теорема, устанавливающая для любого треугольника отношение между длиной его сторон и косинусом одного из его углов. Рассмотрим неравносторонний треугольник на рис. 7.5 (второй справа). С учетом обозначений на рис. 7.5 имеем: $c^2 = a^2 + b^2 - 2 \times a \times b \times \cos(\gamma)$ или $\gamma = a\cos((a^2 + b^2 - c^2)/(2 \times a \times b))$. Для двух других углов α и β формула определяется точно так же.
11. В тригонометрии теорема синусов (формула синусов, правило синусов) – это равенство отношений сторон произвольного треугольника к синусам противолежащих углов. Еще раз обратимся к изображению неравностороннего треугольника на рис. 7.5. С учетом указанных обозначений и принимая R как радиус описанной окружности имеем: $a/\sin(\alpha) = b/\sin(\beta) = c/\sin(\gamma) = 2R$.
12. Теорема Пифагора является частным случаем теоремы косинусов. Теорема Пифагора применима только к прямоугольным треугольникам. Если угол γ прямой (его величина равна 90°, или $\pi/2$ радиан), то $\cos(\gamma) = 0$, следовательно, формула теоремы косинусов сводится к формуле $c^2 = a^2 + b^2$. Теорема Пифагора используется для вычисления евклидова расстояния между двумя точками, как было показано в одном из предыдущих разделов.
13. Пифагорова тройка (Pythagorean triple) – это тройка трех положительных целых чисел a, b, c, часто записываемая в виде (a, b, c), такая, что $a^2 + b^2 = c^2$. Общеизвестный пример: (3, 4, 5). Если (a, b, c) является пифагоровой тройкой, то ее свойство сохраняется для троек (ka, kb, kc), где k – любое положительное целое число. Пифагорова тройка соответствует целочисленным длинам трех сторон прямоугольного треугольника.

Файл исходного кода: *ch7_03_triangles.cpp/java*

Упражнение 7.2.4.1. Длины сторон a, b, c треугольника равны 2^{18}, 2^{18} и 2^{18}. Можно ли вычислить площадь этого треугольника по формуле Герона, приведенной в пункте 4 текущего раздела, без возникновения переполнения (предположим, что мы используем 64-битовые целые числа)? Что необходимо сделать, чтобы устранить эту проблему?

Упражнение 7.2.4.2*. Написать код для поиска центра описанной окружности (circumCircle) по трем точкам a, b, c. Структура этой функции аналогична структуре функции `inCircle`, представленной в данном разделе.

Упражнение 7.2.4.3*. Написать код для проверки, находится ли точка d внутри описанной окружности (circumCircle), построенной по трем точкам a, b, c.

7.2.5. Двумерные объекты: четырехугольники

1. Четырехугольник (quadrilateral, quadrangle) – это многоугольник с четырьмя сторонами (и четырьмя вершинами).
   Обобщенный термин «многоугольник» (polygon) более подробно рассматривается в следующем разделе 7.3.
   На рис. 7.7 показаны некоторые примеры (типы) четырехугольников.

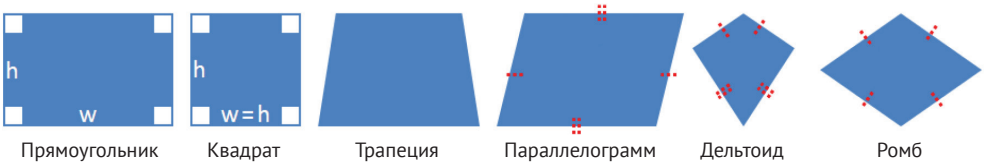


Рис. 7.7 ❖ Четырехугольники

2. Прямоугольник (rectangle) – это многоугольник с четырьмя сторонами, четырьмя вершинами и четырьмя прямыми углами.
3. Площадь прямоугольника с шириной w и высотой h вычисляется по формуле $S = w \times h$, а периметр – по формуле $P = 2 \times (w + h)$.
4. Квадрат (square) – частный случай прямоугольника, в котором $w = h$.
5. Трапеция (trapezium) – многоугольник с четырьмя сторонами, четырьмя вершинами и одной парой параллельных сторон. Если две непараллельные стороны имеют одинаковую длину, то это равнобедренная трапеция (isosceles trapezium).
6. Площадь трапеции с парой параллельных сторон длиной $w1$ и $w2$ и высотой h между этими двумя сторонами вычисляется по формуле $S = 0.5 \times (w1 + w2) \times h$.
7. Параллелограмм (parallelogram) – это многоугольник с четырьмя сторонами и четырьмя вершинами, в котором противолежащие стороны обязательно должны быть параллельными.
8. Дельтоид (kite) – четырехугольник, в котором две пары смежных сторон имеют одинаковую длину. Площадь дельтоида равна половине произведения длин диагоналей.

9. Ромб (rhombus) – частный случай параллелограмма, в котором каждая сторона имеет одинаковую длину. Кроме того, это частный случай дельтоида, в котором каждая сторона имеет одинаковую длину.

7.2.6. Замечания о трехмерных объектах

На олимпиадах по программированию задачи с использованием трехмерных объектов встречаются редко. Но если такая задача содержится в предложенном комплекте заданий, то она может быть одной из самых трудных. В приведенный ниже список заданий по программированию мы включили краткий список задач с использованием трехмерных объектов.

Задания по программированию, связанные с геометрией

- Точки и прямые линии
  1. UVa 00152 – Tree's a Crowd (предварительная сортировка точек в трехмерном пространстве)
  2. UVa 00191 – Intersection (пересечение отрезков прямых линий)
  3. UVa 00378 – Intersecting Lines (использование функций `areParallel`, `areSame`, `areIntersect`)
  4. UVa 00587 – There's treasure everywhere (вычисление евклидова расстояния `dist`)
  5. UVa 00833 – Water Falls (рекурсивная проверка, использование проверок на поворот против часовой стрелки `ccw`)
  6. UVa 00837 – Light and Transparencies (отрезки прямых, предварительная сортировка координат *x*)
  7. **UVa 00920 – Sunny Mountains *** (вычисление евклидова расстояния `dist`)
  8. UVa 01249 – Euclid (LA 4601, SoutheastUSA Regional 2009, вектор)
  9. UVa 10242 – Fourth Point (функция toVector; преобразование `translate` точек с помощью этого вектора)
  10. *UVa 10250 – The Other Two Trees* (вектор, поворот)
  11. **UVa 10263 – Railway *** (использование функции distToLineSegment)
  12. UVa 10357 – Playball (вычисление евклидова расстояния `dist`, простая физическая имитация)
  13. UVa 10466 – How Far? (вычисление евклидова расстояния `dist`)
  14. UVa 10585 – Center of symmetry (сортировка точек)
  15. *UVa 10832 – Yoyodyne Propulsion…* (евклидово расстояние в трехмерном пространстве; имитация)
  16. UVa 10865 – Brownie Points (точки и квадранты, простое задание)
  17. UVa 10902 – Pick-up sticks (пересечение отрезков прямых)
  18. **UVa 10927 – Bright Lights *** (сортировка точек по градиенту, евклидово расстояние)
  19. UVa 11068 – An Easy Task (два простых линейных уравнения с двумя неизвестными)
  20. UVa 11343 – Isolated Segments (пересечение отрезков прямых)
  21. UVa 11505 – Logo (вычисление евклидова расстояния `dist`)

22. UVa 11519 – Logo 2 (векторы и углы)
23. UVa 11894 – Genius MJ (приводится к операциям вращения и переноса точек)

- Окружности (только)
  1. *UVa 01388 – Graveyard* (предварительное разделение круга на n секторов, затем на $(n + m)$ секторов)
  2. **UVa 10005 – Packing polygons** * (полный поиск; использование функции `circle2PtsRad`, описанной в главе 7)
  3. UVa 10136 – Chocolate Chip Cookies (аналогично заданию UVa 10005)
  4. UVa 10180 – Rope Crisis in Ropeland (точка фигуры AB, ближайшая к началу координат; дуга)
  5. UVa 10209 – Is This Integration? (квадрат, дуги, аналогично заданию UVa 10589)
  6. UVa 10221 – Satellites (вычисление длины дуги и хорды окружности)
  7. *UVa 10283 – The Kissing Circles* (вывод требуемой формулы)
  8. UVa 10432 – Plygon Inside A Circle (площадь n-стороннего правильного многоугольника внутри окружности)
  9. UVa 10451 – Ancient… (вписанная/описанная окружность для n-стороннего правильного многоугольника)
  10. UVa 10573 – Geometry Paradox (здесь нет «невозможного» варианта)
  11. **UVa 10589 – Area** * (проверка: находится ли точка внутри области пересечения 4 кругов)
  12. UVa 10678 – The Grazing Cows * (площадь эллипса, обобщение формулы площади круга)
  13. UVa 12578 – 10:6:2 (площадь области, ограниченной прямоугольником и окружностью)

- Треугольники (в сочетании с окружностями)
  1. UVa 00121 – Pipe Fitters (использование теоремы Пифагора; сетка (грид-вычисления))
  2. UVa 00143 – Orchard Trees (подсчет целочисленных точек в треугольнике; проблема точности вычислений)
  3. UVa 00190 – Circle Through Three… (окружность, описанная вокруг треугольника)
  4. UVa 00375 – Inscribed Circles and… (окружность, вписанная в треугольник)
  5. UVa 00438 – The Circumference of… (окружность, описанная вокруг треугольника)
  6. UVa 10195 – The Knights Of The… (окружность, вписанная в треугольник, формула Герона)
  7. UVa 10210 – Romeo & Juliet (простая тригонометрия)
  8. UVa 10286 – The Trouble with a… (теорема синусов)
  9. UVa 10347 – Medians (заданы три медианы треугольника, найти его площадь)
  10. UVa 10387 – Billiard (расширяющаяся поверхность; тригонометрия)
  11. UVa 10522 – Height to Area (вывод требуемой формулы; использование формулы Герона)

12. **UVa 10577 – Bounding box** * (определение центра и радиуса внешней окружности по трем точкам, определение всех вершин, определение минимальной/максимальной координаты *x* и минимальной/максимальной координаты *y* искомого многоугольника)
13. *UVa 10792 – The Laurel-Hardy Story* (вывод тригонометрических формул)
14. UVa 10991 – Region (формула Герона, теорема косинусов, площадь сектора)
15. **UVa 11152 – Colourful…** * (окружность, вписанная в треугольник / описанная вокруг треугольника; формула Герона)
16. *UVa 11164 – Kingdom Division* (используются свойства треугольника)
17. *UVa 11281 – Triangular Pegs in…* (окружность минимальной длины, описанная вокруг нетупоугольного треугольника; если треугольник тупоугольный, то радиусы окружности минимальной длины – это наибольшая сторона треугольника)
18. *UVa 11326 – Laser Pointer* (тригонометрия, касательная, прием с отражением фигуры)
19. *UVa 11437 – Triangle Fun* (совет: 1/7)
20. UVa 11479 – Is this the easiest problem? (проверка свойств)
21. UVa 11579 – Triangle Trouble (сортировка; жадный алгоритм проверки: соответствуют ли три последовательные стороны неравенству треугольника, а если соответствуют, то проверка: не найден ли до сих пор наибольший треугольник)
22. UVa 11854 – Egypt (теорема Пифагора, пифагоровы тройки)
23. **UVa 11909 – Soya Milk** * (теорема синусов (или теорема тангенсов); два возможных случая)
24. UVa 11936 – The Lazy Lumberjacks (проверка: образуют ли три стороны допустимый корректный треугольник)

- Четырехугольники

1. UVa 00155 – All Squares (рекурсивные вычисления)
2. **UVa 00460 – Overlapping Rectangles** * (пересечение прямоугольников)
3. UVa 00476 – Points in Figures: … (аналогично заданиям UVa 477 и 478)
4. UVa 00477 – Points in Figures: … (аналогично заданиям UVa 476 и 478)
5. **UVa 11207 – The Easiest Way** * (разделение прямоугольника на четыре квадрата равного размера)
6. UVa 11345 – Rectangles (пересечение прямоугольников)
7. UVa 11455 – Behold My Quadrangle (проверка свойств)
8. UVa 11639 – Guard the Land (пересечение прямоугольников, использование массива флагов)
9. *UVa 11800 – Determine the Shape* (использование функции перебора перестановок next_permutation, чтобы облегчить перебор всех возможных 4! = 24 перестановок четырех точек; проверка: могут ли они подойти для определения квадрата, прямоугольника, ромба, параллелограмма, трапеции – в указанном порядке)
10. **UVa 11834 – Elevator** * (размещение (упаковка) двух окружностей в прямоугольнике)

11. *UVa 12256 – Making Quadrilaterals* (LA 5001, KualaLumpur 10, начать с трех сторон 1, 1, 1, затем четвертая сторона обязательно должна вычисляться как нарастающая сумма предыдущих трех сторон, чтобы определить прямую линию; процедура повторяется до получения *n*-й стороны)

- *Трехмерные* объекты
  1. **UVa 00737 – Gleaming the Cubes *** (куб и пересечение кубов)
  2. **UVa 00815 – Flooded *** (объем, жадный алгоритм, сортировка по высоте, имитация)
  3. **UVa 10297 – Beavergnaw *** (конусы, цилиндры, объемы)

Известные авторы алгоритмов

Пифагор Самосский (≈570–≈495 гг. до н. э.) – древнегреческий математик и философ, родился на острове Самос. Наиболее известен по теореме Пифагора, определяющей соотношения сторон прямоугольного треугольника.

Евклид Александрийский (≈325–≈265 гг. до н. э.) – древнегреческий математик, известный как «отец геометрии». Родился в городе Александрия. Наиболее важная работа Евклида по математике (особенно в области геометрии) – «Начала» (Elements). В «Началах» Евклид вывел основные принципы той дисциплины, которая сейчас называется евклидовой геометрией, из небольшого набора аксиом.

Герон Александрийский (≈10–≈75 гг.) – древнегреческий математик, родился в Александрии (Египет), в том же городе, что и Евклид. С его именем тесно связана формула вычисления площади треугольника по длинам его сторон.

Рональд Льюис Грэм (Грэхем) (род. в 1935 г.) – американский математик. В 1972 году разработал алгоритм Грэхема для поиска (построения) выпуклой оболочки по конечному множеству точек на плоскости. В настоящее время существует много вариантов и усовершенствований алгоритма поиска (построения) выпуклой оболочки.

7.3. Алгоритмы для многоугольников с использованием библиотечных функций

Многоугольник (polygon) – это плоская фигура, ограниченная замкнутым контуром (то есть контур начинается и заканчивается в одной и той же вершине), состоящим из конечной последовательности отрезков прямых линий. Эти отрезки называются сторонами, или ребрами, многоугольника. Точка соединения двух сторон (ребер) называется вершиной, или углом, многоугольника. Многоугольник является темой многих задач (вычислительной) геометрии, так как позволяет автору задачи представлять более реалистичные объекты по сравнению с рассматриваемыми в разделе 7.2.

7.3.1. Представление многоугольника

Стандартный способ представления многоугольника – простое перечисление его вершин в порядке обхода по часовой стрелке или против часовой стрелки, при этом первая вершина совпадает с последней вершиной (некоторые функции, рассматриваемые ниже в этом разделе, требуют обязательного соблюдения этого условия, см. **упражнение 7.3.4.1***). В этой книге по умолчанию принят порядок перечисления вершин против часовой стрелки. Многоугольник, построенный в результате выполнения приведенного ниже кода, показан на рис. 7.8, справа.

```
// 6 точек, ввод в направлении против часовой стрелки, индексация начинается с 0
vector<point> P;
P.push_back( point(1,1) );       // P0
P.push_back( point(3,3) );       // P1
P.push_back( point(9,1) );       // P2
P.push_back( point(12,4) );      // P3
P.push_back( point(9,7) );       // P4
P.push_back( point(1,7) );       // P5
P.push_back( P[0] );             // важно: замыкание контура в начальной вершине
```

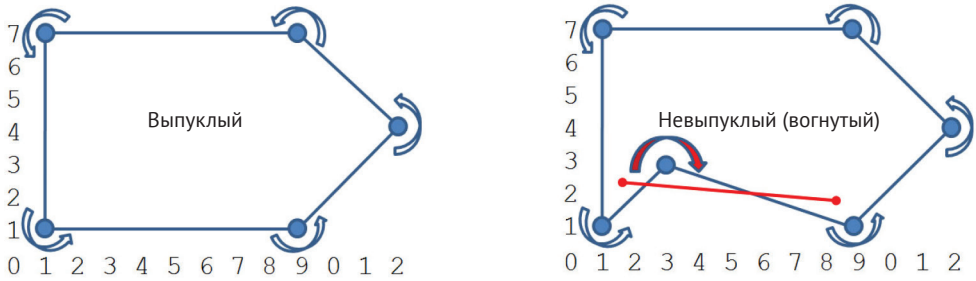


Рис. 7.8 ❖ Слева: выпуклый многоугольник, справа: невыпуклый (вогнутый) многоугольник

7.3.2. Периметр многоугольника

Периметр многоугольника (как выпуклого, так и невыпуклого) с n вершинами, заданными в определенном порядке (по часовой стрелке или против часовой стрелки), можно вычислить с помощью простой функции, приведенной ниже.

```
// Возвращает значение периметра, который является суммой евклидовых расстояний
// связанных между собой отрезков прямых линий (сторон многоугольника).
double perimeter( const vector<point> &P )
{
  double result = 0.0;
  for( int i=0; i < (int)P.size()-1; i++ )           // следует помнить, что P[0] = P[n-1]
    result += dist( P[i], P[i+1] );
  return result;
}
```

7.3.3. Площадь многоугольника

Площадь многоугольника S со знаком (как выпуклого, так и невыпуклого) с n вершинами, заданными в определенном порядке (по часовой стрелке или против часовой стрелки) можно вычислить с помощью определителя матрицы, показанного ниже. Эту формулу легко представить в виде библиотечного исходного кода:

$$A = \frac{1}{2} \times \begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \cdots & \cdots \\ x_{n-1} & y_{n-1} \end{vmatrix} = \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \cdots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \cdots - x_0 \times y_{n-1}).$$

```
// Возвращает площадь, равную половине значения определителя,
// содержащего координаты вершин.
double area( const vector<point> &P )
{
  double result = 0.0, x1, y1, x2, y2;
  for( int i=0; i < (int)P.size()-1; i++ ) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0;
}
```

7.3.4. Проверка многоугольника на выпуклость

Многоугольник является выпуклым (convex), если любой произвольный отрезок прямой линии, построенный внутри этого многугольника, не пересекает ни одну из его сторон. В противном случае многоугольник является невыпуклым (concave).

Но для проверки многоугольника на выпуклость существует вычислительный метод, более простой по сравнению с «попыткой проверить факт расположения всех возможных отрезков внутри многоугольника без пересечения его сторон». Можно просто проверить, выполняется ли во всех смежных тройках вершин многоугольника поворот сторон в одном и том же направлении (если вершины перечисляются в порядке против часовой стрелки, то все стороны в вершинах должны поворачиваться влево, то есть против часовой стрелки, а если вершины перечисляются в порядке по часовой стрелке, то все стороны в вершинах должны поворачиваться вправо, то есть по часовой стрелке). Если можно обнаружить хотя бы одну тройку вершин, для которых это условие не выполняется, то многоугольник не является выпуклым (см. рис. 7.8).

```
// Возвращает true, если для всех троек смежных вершин многоугольника P выполняется условие
// поворота сторон в одном и том же направлении.
```

```
bool isConvex( const vector<point> &P )
{
  int sz = (int)P.size();
  if( sz <= 3 ) return false;  // точка sz=2 или прямая sz=3 не являются выпуклыми фигурами
  bool isLeft = ccw( P[0], P[1], P[2] );                    // запомнить один результат
  for( int i=1; i < sz-1; i++ )               // затем сравнивать этот результат с другими
    if( ccw( P[i], P[i+1], P[(i+2) == sz ? 1 : i+2] ) != isLeft )
      return false;       // смена знака (направления поворота) -> многоугольник невыпуклый
  return true;                                                // многоугольник выпуклый
}
```

Упражнение 7.3.4.1*. Какую часть приведенного выше кода необходимо изменить, чтобы правильно обрабатывать тройки коллинеарных (лежащих на одной прямой) точек? Пример: многоугольник {(0,0), (2,0), (4,0), (2,2), (0,0)} должен быть определен как выпуклый.

Упражнение 7.3.4.2*. Если первая вершина не повторяется в перечислении как последняя вершина, то будут ли приведенные выше функции `perimeter`, `area` и `isConvex` работать правильно?

7.3.5. Проверка расположения точки внутри многоугольника

Еще одной часто выполняемой проверкой для многоугольника P является проверка расположения некоторой точки *pt* внутри или вне прямоугольника P. Приведенная ниже функция, в которой реализован «алгоритм определения порядка (индекса) точки относительно замкнутой кривой (winding number)», позволяет выполнить такую проверку для любого выпуклого или невыпуклого многоугольника. Принцип работы этой функции заключается в вычислении суммы углов между тремя точками: $\{P[i], pt, P[i+1]\}$, где $(P[i] - P[i+1])$ – смежные стороны многоугольника P. При этом определяются повороты влево (прибавление величины угла) и повороты вправо (вычитание величины угла) соответственно. Если итоговая сумма равна 2π (360°), то точка *pt* находится внутри многоугольника P (см. рис. 7.9).

```
// Возвращает true, если точка p находится внутри выпуклого/вогнутого многоугольника P.
bool inPolygon( point pt, const vector<point> &P )
{
  if( (int)P.size() == 0 ) return false;
  double sum = 0;      // предполагается, что первая вершина совпадает с последней вершиной
  for( int i=0; i < (int)P.size()-1; i++ ) {
    if( ccw( pt, P[i], P[i+1] ) )
      sum += angle( P[i], pt, P[i+1] );           // поворот влево / против часовой стрелки
    else sum -= angle( P[i], pt, P[i+1] );           // поворот вправо / по часовой стрелке
  }
  return fabs( fabs(sum) - 2*PI ) < EPS;
}
```

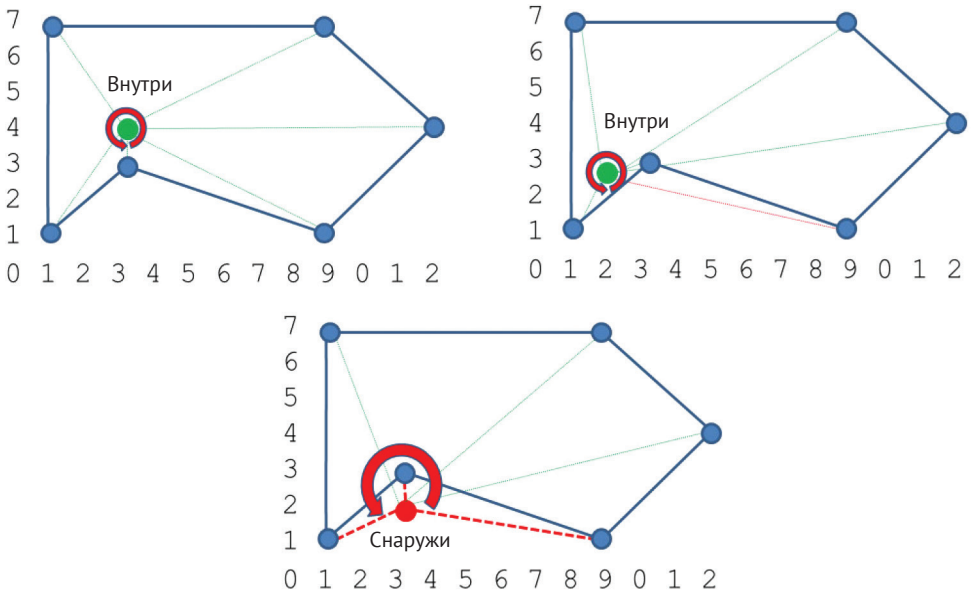


Рис. 7.9. Слева вверху: точка внутри многоугольника,
справа вверху: точка также внутри многоугольника, внизу: точка вне многоугольника

Упражнение 7.9.1*. Что произойдет с функцией `inPolygon`, если точка `pt` находится на одной из сторон многоугольника *P*, то есть `pt = P[0]` или `pt` находится на середине стороны между вершинами `P[0]` и `P[1]` и т. д.? Что необходимо сделать, чтобы исправить эту ситуацию?

Упражнение 7.9.2*. Рассмотрите все достоинства и недостатки следующих альтернативных методов проверки нахождения точки внутри многоугольника:

1) разделение выпуклого многоугольника на треугольники и проверка суммы площадей полученных треугольников на равенство площади выпуклого многоугольника;
2) алгоритм Ray Casting: построение луча из проверяемой точки в любом фиксированном направлении так, чтобы этот луч пересекал сторону (стороны) многоугольника. Если число пересечений нечетно/четно, то точка находится внутри/вне многоугольника соответственно.

7.3.6. Разделение многоугольника с помощью прямой линии

Еще одна заслуживающая внимания операция, которую можно выполнить с выпуклым многоугольником (см. **упражнение 7.3.6.2*** для невыпуклого многоугольника), – разделение его на два выпуклых многоугольника с помощью прямой линии, определяемой двумя точками *a* и *b*. Внимательно рассмотрите упражнения по программированию, предложенные в конце этого подраздела, в которых используется данная функция.

Основная идея приведенной ниже функции `cutPolygon` состоит в итеративном последовательном проходе по вершинам исходного многоугольника *Q*. Если прямая *ab* и вершина многоугольника *v* образуют поворот влево (что подразумевает расположение вершины *v* слева от прямой *ab*), то мы помещаем вершину *v* внутри нового многоугольника *P*. Как только мы находим сторону многоугольника, которая пересекается с прямой *ab*, мы используем эту точку пересечения как часть нового многоугольника *P* (см. рис. 7.10, слева, точка *C*). Затем мы пропускаем несколько следующих вершин исходного многоугольника *Q*, которые расположены справа от прямой *ab*. Рано или поздно мы обнаружим еще одну сторону исходного многоугольника, которая тоже пересекается с прямой *ab* (см. рис. 7.10, слева, точка *D*, которая оказалась одной из вершин исходного многоугольника *Q*). Далее мы продолжаем добавлять вершины исходного многоугольника *Q* в новый многоугольник *P*, поскольку теперь мы снова находимся слева от прямой *ab*. Операция останавливается, когда мы возвращаемся в начальную вершину и получаем итоговый многоугольник *P* (см. рис. 7.10, справа).

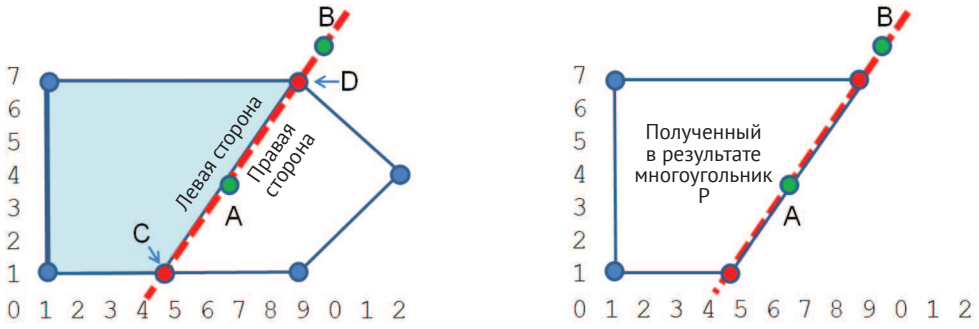


Рис. 7.10 ❖ Слева: перед разделением, справа: после разделения

```
// Отрезок прямой p-q пересекается с прямой линией A-B
point lineIntersectSeg( point p, point q, point A, point B )
{
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs( a * p.x + b * p.y + c );
  double v = fabs( a * q.x + b * q.y + c );
  return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v));
}

// Разделение многоугольника Q по прямой, проходящей через точки a -> b.
// (Примечание: последняя точка обязательно должна совпадать с первой точкой.)
vector<point> cutPolygon( point a, point b, const vector<point> &Q )
{
  vector<point> P;
  for( int i=0; i < (int)Q.size(); i++ ) {
    double left1 = cross( toVec(a,b), toVec(a, Q[i]) ), left2 = 0;
    if( i != (int)Q.size()-1 ) left2 = cross( toVec(a,b), toVec(a, Q[i+1]) );
```

```
    if( left1 < -EPS ) P.push_back( Q[i] );            // Q[i] находится слева от прямой ab
    if( left1 * left2 < -EPS )          // сторона (Q[i], Q[i+1]) пересекается с прямой ab
      P.push_back( lineIntersectSeg( Q[i], Q[i+1], a, b ) );
  }
  if( !P.empty() && !(P.back() == P.front()) )
    P.push_back( P.front() );                // сделать первую точку P = последней точке P
  return P;
}
```

Чтобы помочь читателям лучше понять описанные выше алгоритмы для многоугольников, мы создали инструмент визуального представления для третьего издания этой книги. Читатель может построить собственный многоугольник и предложить инструментальному средству наглядно представить описание выполнения алгоритма, рассмотренного в этом разделе.

Адрес инструментального средства визуального представления: www.comp.nus.edu.sg/~stevenha/visualization/polygon.html.

Упражнение 7.3.6.1. Функция `cutPolygon` возвращает только левую сторону исходного многоугольника Q после разделения его линией ab. Что нужно сделать, если необходимо возвращать правую сторону вместо левой?

Упражнение 7.3.6.2*. Что произойдет, если выполнить функцию `cutPolygon` для невыпуклого многоугольника?

7.3.7. Построение выпуклой оболочки множества точек

Выпуклая оболочка (convex hull) для некоторого множества точек P – это наименьший выпуклый многоугольник $CH(P)$, для которого каждая точка из множества P либо лежит на границе многоугольника $CH(P)$, либо находится в его внутренней области. Представьте, что точки – это гвозди, вбитые (не полностью) в двумерную плоскость, и у нас имеется достаточно длинная резиновая лента, которую можно растянуть так, чтобы окружить все гвозди. Если освободить эту ленту, то она попытается стянуться так, чтобы охватывать наименьшую возможную область. Эта область и есть выпуклая оболочка (convex hull) для данного множества точек/гвоздей (см. рис. 7.11). Поиск выпуклой оболочки для заданного множества точек имеет практическое применение в задачах упаковки (packing problems).

Так как каждая вершина многоугольника $CH(P)$ является вершиной из множества точек P, алгоритм поиска выпуклой оболочки по существу представляет собой алгоритм определения тех точек из множества P, которые должны быть выбраны как часть выпуклой оболочки. Существует несколько общеизвестных алгоритмов поиска выпуклой оболочки. В этом разделе мы выбрали для подробного рассмотрения алгоритм Роналда Грэхема (Грэма; Ronald Graham) со сложностью $O(n \times \log n)$.

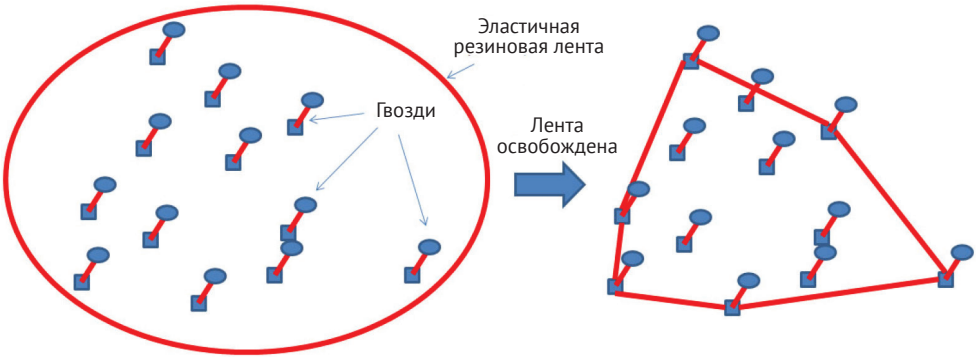


Рис. 7.11 ❖ Аналогия с резиновой лентой и множеством гвоздей для иллюстрации задачи о поиске выпуклой оболочки

Алгоритм Грэхема сначала сортирует все *n* точек множества *P*, при этом первая точка не должна повторяться как последняя точка (см. рис. 7.12A). Сортировка производится на основе величин углов по отношению к точке, которая определяется как центральная точка (pivot). В нашем примере в качестве центральной выбрана самая нижняя и самая правая точка из множества *P*. После сортировки по значениям углов относительно этой центральной точки можно видеть, что ребра-отрезки 0–1, 0–2, 0–3, ..., 0–10, 0–11 пронумерованы в порядке против часовой стрелки (см. точки с 1 по 11 относительно точки 0 на рис. 7.12B).

```
point pivot( 0, 0 );
bool angleCmp( point a, point b )                    // функция сортировки по величине угла
{
  if( collinear( pivot, a, b ) )                              // обработка особого случая
    return dist( pivot, a ) < dist( pivot, b );          // проверка, какая из точек ближе
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;                 // сравнение двух углов
}

vector<point> CH( vector<point> P )      // содержимое вектора P может быть переупорядочено
{
  int i, j, n = (int)P.size();
  if( n <= 3 ) {
    if( !(P[0] == P[n-1]) )  P.push_back( P[0] );              // случай совпадения первой
                                                                    // и последней точек
  return P;                              // особый случай: CH, собственно, и есть множество P
  }

  // Сначала найти P0 = точка с наименьшей координатой Y, а при равенстве координат –
  // точка с "самой правой" координатой X
  int P0 = 0;
  for( i=1; i < n; i++ )
    if( P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x) )
      P0 = i;
```

```
  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;              // обмен значений P[P0] и P[0]

  // Второй этап: сортировка точек по величине угла относительно центральной точки P0
  pivot = P[0];                       // использование этой глобальной переменной как ссылки
  sort( ++P.begin(), P.end, angleCmp );  // в процедуру сортировки не включается точка P[0]
// продолжение кода см. ниже
```

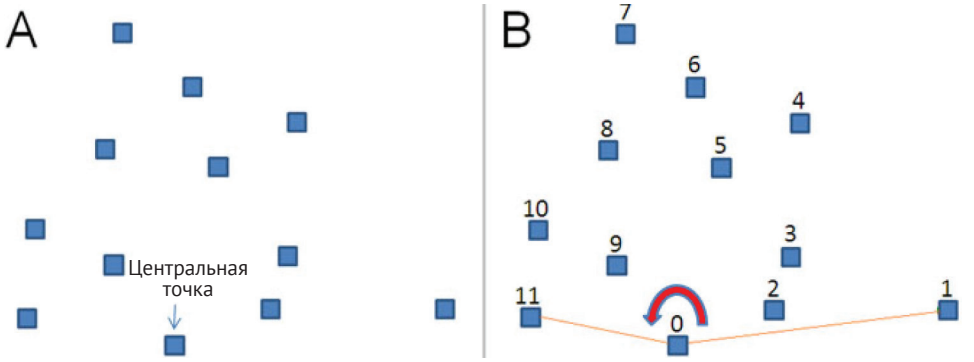


Рис. 7.12 ❖ Сортировка множества из 12 точек
по их углам относительно центральной точки (точка 0)

Затем этот алгоритм обрабатывает стек *S* точек-кандидатов. Каждая точка из множества *P* однократно помещается в стек *S*, а точки, которые не становятся частью выпуклой оболочки *CH*(*P*), в конечном итоге будут исключены из стека *S*. Алгоритм Грэхема постоянно следит за соблюдением неизменного условия (инварианта): три самые верхние точки в стеке *S* всегда обязательно должны выполнять поворот влево (это основное свойство выпуклого многоугольника).

Сначала мы помещаем в стек эти три точки: *N* – 1, 0 и 1. В нашем примере стек изначально содержит (дно) 11-0-1 (вершина). Эти точки всегда формируют поворот влево.

Теперь рассмотрим рис. 7.13C. Здесь мы пытаемся поместить в стек точку 2 и видим, что 0-1-2 формирует поворот влево, поэтому точка 2 принимается. Теперь стек *S* выглядит так: (дно) 11-0-1-2 (вершина).

Далее рассмотрим рис. 7.13D. При попытке поместить в стек точку 3 обнаруживается, что 1-2-3 – это поворот вправо. Значит, если мы принимаем точку перед точкой 3, то есть точку 2, то не получим выпуклый многоугольник. Поэтому необходимо удалить из стека точку 2. Теперь в стеке снова содержатся точки (дно) 11-0-1 (вершина). После этого выполняется повторная попытка помещения в стек точки 3. Тройка 0-1-3 – три текущие самые верхние точки в стеке S формируют поворот влево, поэтому точка 3 принимается. Теперь стек содержит точки (дно) 11-0-1-3 (вершина).

Этот процесс повторяется до тех пор, пока не будут обработаны все вершины (см. рис. 7.13E-F-G-H). После завершения работы алгоритма Грэхема все точки, оставшиеся в стеке S, являются точками выпуклой оболочки CH(P) (см. рис. 7.13H, стек содержит точки (дно) 11-0-1-4-7-10-11 (вершина)). Алгоритм Грэхема исключает все повороты вправо. Так как любые три последовательно

взятые вершины в стеке S всегда формируют повороты влево, мы получаем выпуклый многоугольник.

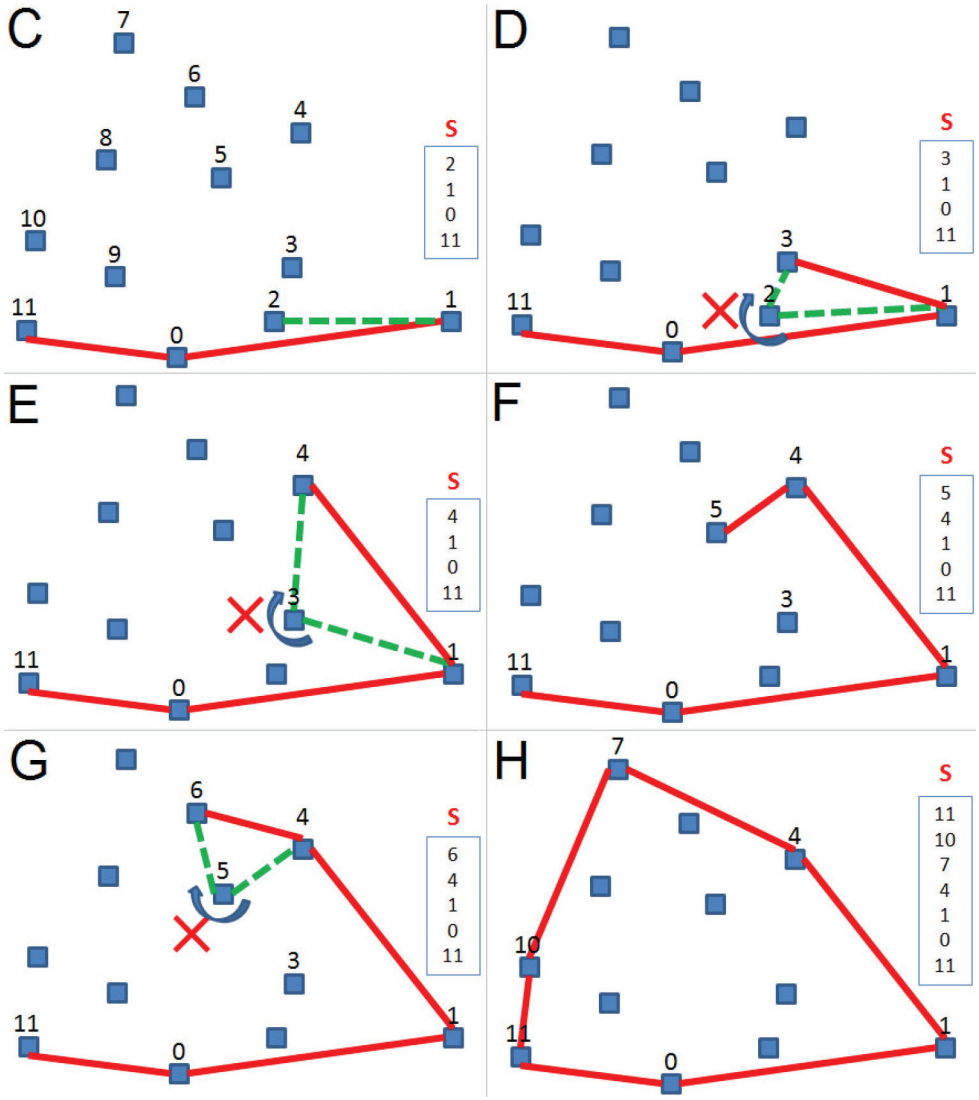


Рис. 7.13 ❖ Основная часть процесса выполнения алгоритма Грэхема

Реализация алгоритма Грэхема приведена ниже. Мы просто используем вектор `vector<point> S`, который ведет себя как стек, вместо настоящего стека `stack<point>`. Первая часть алгоритма Грэхема (поиск центральной точки) имеет сложность $O(n)$. Третья часть (проверки на поворот против часовой стрелки *ccw*) также имеет сложность $O(n)$. Этот вывод можно сделать на основе того факта, что каждая из n вершин может быть помещена в стек только один раз

и извлечена из стека однократно. Вторая часть (сортировка точек по величине угла относительно центральной точки $P[0]$) – самая громоздкая и требует времени $O(n \times \log n)$. Таким образом, общая временна́я сложность алгоритма Грэхема равна $O(n \times \log n)$.

```
// Продолжение исходного кода реализации алгоритма Грэхема
  // Третья часть: проверки на поворот против часовой стрелки ccw
  vector<point> S;
  S.push_back( P[n-1] ); S.push_back( P[0] ); S.push_back( P[1] );   // начальное состояние
                                                                                  // стека S
  i = 2;                                                  // далее проверяем остальные точки
  while( i < n ) {  // примечание: N обязательно должно быть >= 3, чтобы этот метод работал
    j = (int)S.size()-1;
    if( ccw( S[j-1], S[j], P[i] ) )  S.push_back( P[i++] );              // поворот влево,
                                                                    // точка принимается
    else S.pop_back();                              // или точки удаляются из стека до тех пор,
                                                                 // пока не найден поворот влево
  }
  return S;                           // возвращается конечный результат, сохраненный в стеке
}
```

В конце этого раздела и этой главы мы предлагаем вниманию читателей еще одно инструментальное средство визуализации, теперь это визуальное представление нескольких алгоритмов поиска выпуклой оболочки, в том числе и алгоритма Грэхема, алгоритма монотонных цепочек Эндрю (Andrew) (см. **упражнение 7.3.7.4***) и алгоритма March Джарвиса (Jarvis). Кроме того, мы предлагаем читателям применять приведенный выше исходный код для решения различных упражнений и заданий по программированию из этого раздела.

Сетевой адрес инструментального средства визуального представления алгоритмов: www.comp.nus.edu.sg/~stevenha/visualization/convexhull.html.

Файл исходного кода: *ch7_04_polygon.cpp/java*

Упражнение 7.3.7.1. Предположим, что задано пять точек: $P = \{(0,0), (1,0), (2,0), (2,2), (0,2)\}$. Выпуклая оболочка для этих пяти точек в действительности представлена самими этими точками (плюс одна, так как необходимо замыкание пути обхода в вершине (0, 0)). Но приведенная в этом разделе реализация алгоритма Грэхема удаляет точку (1, 0), так как точки (0, 0)-(1, 0)-(2, 0) коллинеарны (лежат на одной прямой). Какую часть реализации этого алгоритма необходимо изменить, чтобы обеспечить правильную обработку коллинеарных точек?

Упражнение 7.3.7.2. В функции `angleCmp` есть вызов функции `atan2`. Эта функция используется для сравнения величин двух углов, но что в действительности возвращает функция `atan2`? Проведите тщательное исследование.

Упражнение 7.3.7.3*. Протестировать приведенный выше код реализации алгоритма Грэхема `CH(P)` для перечисленных ниже особых нестандартных случаев. Какой будет выпуклая оболочка для следующих тестовых вариантов:

1) одна точка, например P_1 = {(0,0)}?
2) две точки (прямая), например P_2 = {(0,0), (1,0)}?
3) три точки (треугольник), например P_3 = {(0,0), (1,0), (1,1)}?
4) три коллинеарные точки, например P_4 = {(0,0), (1,0), (2,0)}?
5) четыре коллинеарные точки, например P_5 = {(0,0), (1,0), (2,0), (3,0)}?

Упражнение 7.3.7.4*. Приведенная выше реализация алгоритма Грэхема может быть неэффективной при больших значениях *n*, так как функция `atan2` повторно вычисляется при каждой операции сравнения углов (кроме того, проблемы возникают, когда углы близки к 90°). В действительности тот же основной принцип алгоритма Грэхема работает также и в случае, если входные данные отсортированы по координате *x* (а в случае их равенства – по координате *y*), а не по величинам углов. Тогда выпуклая оболочка вычисляется в два этапа с раздельным получением верхней и нижней частей этой оболочки. Это усовершенствование было разработано Э. М. Эндрю (A. M. Andrew) и известно как алгоритм монотонных цепочек Эндрю. Данный алгоритм обладает теми же основными свойствами, что и алгоритм Scan Грэхема, но исключает многозатратные операции сравнения величин углов [9]. Тщательно изучите этот алгоритм и напишите код его реализации.

Ниже приведен список заданий по программированию, так или иначе связанных с многоугольниками. Без предварительно написанного библиотечного кода, рассматриваемого в этом разделе, многие из этих заданий могут показаться «слишком трудными». При использовании предложенного библиотечного кода задания выглядят более простыми, поскольку теперь задачу можно разделить на несколько подзадач, решаемых с помощью библиотечных функций. Рекомендуем потратить некоторое время на их решение (по крайней мере, на попытки найти решение), особенно на решение заданий, помеченных звездочкой * и выделенных полужирным шрифтом (это «обязательные к выполнению» задания).

Задания по программированию, связанные с многоугольниками

1. UVa 00109 – Scud Busters (поиск выпуклой оболочки `CH`, проверка: точка внутри многоугольника `inPolygon`, площадь многоугольника `area`)
2. *UVa 00137 – Polygons* (пересечение выпуклых многоугольников, пересечение отрезков, точка внутри многоугольника `inPolygon`, выпуклая оболочка `CH`, площадь `area`, принцип включения-исключения)
3. UVa 00218 – Moth Eradication (поиск выпуклой оболочки `CH`, периметр многоугольника `perimeter`)
4. UVa 00361 – Cops and Robbers (проверка: точка внутри выпуклой оболочки `CH` из полицейских/воров; находится ли точка *pt* внутри выпуклой оболочки, затем окончательное определение треугольника, образованного тремя вершинами этой выпуклой оболочки, и этот треугольник содержит точку *pt*)
5. UVa 00478 – Points in Figures: ... (проверки: `inPolygon`/`inTriangle`; если заданный многоугольник *P* выпуклый, то есть другой способ провер-

ки расположения точки *pt* внутри или снаружи этого многоугольника *P*, помимо способа, описанного в текущем разделе; можно разделить многоугольник *P* на треугольники с точкой *pt* в качестве одной из их вершин, затем просуммировать площади этих треугольников: если сумма площадей равна площади многоугольника *P*, то точка *pt* находится внутри многоугольника *P*, если сумма площадей больше площади многоугольника *P*, то точка *pt* находится вне многоугольника *P*)

6. *UVa 00596 – The Incredible Hull* (выпуклая оболочка *CH*; форматирование вывода немного утомительно и рутинно)
7. UVa 00634 – Polygon (проверка: inPolygon, многоугольник может быть выпуклым или вогнутым)
8. UVa 00681 – Convex Hull Finging (собственно задача нахождения выпуклой оболочки *CH*)
9. UVa 00858 – Berry Picking (проверка пересечения прямой с многоугольником; сортировка; изменение отрезков)
10. **UVa 01111 – Trash Removal *** (LA 5138, World Finals Orlando11, выпуклая оболочка `CH`, расстояние от каждой стороны выпуклой оболочки `CH` (параллельной заданной стороне) до каждой вершины `CH`)
11. UVa 01206 – Boundary Points (LA 3169, Manila06, выпуклая оболочка `CH`)
12. *UVa 10002 – Center of Mass?* (центроид, центр выпуклой оболочки `CH`, площадь многоугольника `area`)
13. UVa 10060 – A Hole to Catch a Man (площадь многоугольника `area`)
14. UVa 10065 – Useless Tile Packers (поиск выпуклой оболочки `CH`, площадь многоугольника `area`)
15. UVa 10112 – Myacm Triangles (проверка: точка внутри/вне многоугольника/треугольника `inPolygon`/`inTriangle`, см. UVa 478)
16. UVa 10406 – Cutting tabletops (вектор, вращение `rotate`, преобразование `translate`, затем разделение `cutPolygon`)
17. **UVa 10652 – Board Wrapping *** (вращение `rotate`, преобразование `translate`, выпуклая оболочка `CH`, площадь `area`)
18. UVa 11096 – Nails (собственно классическая задача поиска выпуклой оболочки `CH`, рекомендуется начать с решения этого задания)
19. **UVa 11265 – The Sultan's Problem *** (разделение многоугольника `cutPolygon`, проверка: `inPolygon`, площадь `area`)
20. UVa 11447 – Reservoir Logs (площадь многоугольника `area`)
21. UVa 11473 – Campus Roads (периметр многоугольника `perimeter`)
22. UVa 11626 – Convex Hull (поиск выпуклой оболочки CH, будьте внимательны при обработке коллинеарных точек)

7.4. Решения упражнений, не помеченных звездочкой

Упражнение 7.2.1.1. 5.0.

Упражнение 7.2.1.2. (–3.0, 10.0).

Упражнение 7.2.1.3. (–0.674, 10.419).

Упражнение 7.2.2.1. Уравнение прямой линии $y = mx + c$ не может обработать все случаи: вертикальные прямые линии имеют «бесконечный» градиент/коэффициент угла наклона в этом уравнении, кроме того, проблемы возникают и при обработке «почти вертикальных» линий. Если используется это уравнение, то необходимо обрабатывать вертикальные прямые линии отдельно в исходном коде, а это снижает вероятность принятия (AC) решения на олимпиаде. К счастью, такой проблемы можно избежать, если воспользоваться более подходящим в данном случае уравнением $ax + by + c = 0$.

Упражнение 7.2.2.2. $-0.5*x + 1.0*y - 1.0 = 0.0$.

Упражнение 7.2.2.3. $1.0*x + 0.0*y - 2.0 = 0$. Если пользоваться уравнением $y = mx + c$, то вы получите в итоге $x = 2.0$, но не сможете представить в этой форме вертикальную прямую линию $y = ?$.

Упражнение 7.2.2.4. Даны две точки $(x1, y1)$ и $(x2, y2)$, коэффициент угла наклона можно вычислить по формуле $m = (y2 - y1)/(x2 - x1)$. Далее точку отсечения по оси y, то есть свободный член c, можно вычислить из уравнения прямой, подставляя значения координат точек (или одной точки) и ранее вычисленный градиент m. Исходный код приведен ниже. Обратите внимание на то, что необходимо сделать для отдельной затруднительной обработки вертикальных линий.

```
struct line2 { double m, c; };                  // другой способ представления прямой линии

int pointsToLine2( point p1, point p2, line2 &l )
{
  if( p1.x == p2.x ) {                                  // особый случай: вертикальная линия
    l.m = INF;                                          // l содержит m = INF, a c = x_value
    l.c = p1.x;                             // для обозначения вертикальной линии x = x_value
    return 0;  // возврат этого значения необходим для различения результата в особом случае
  }
  else {
    l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
    l.c = p1.y - l.m * p1.x;
    return 1;                           // l содержит m и c для уравнения прямой y = mx + c
  }
}
```

Упражнение 7.2.2.5.

```
// преобразование точки и градиента/коэффициента угла наклона в прямую линию
void pointSlopeToLine( point p, double m, line &l )
{
  l.a = -m;                                                              // всегда равно -m
  l.b = 1;                                                                // всегда равно 1
  l.c = -((l.a * p.x) + (l.b * p.y));                          // вычисление свободного члена
}
```

Упражнение 7.2.2.6. (5.0, 3.0).

Упражнение 7.2.2.7. (4.0, 2.5).

Упражнение 7.2.2.8. (–3.0, 5.0).

Упражнение 7.2.2.9. (0.0, 4.0). Результат отличается от результата, полученного в **упражнении 7.2.2.8**. Операция «перенос, затем поворот» отличается от операции «поворот, затем перенос». Будьте внимательны, определяя последовательность этих операций.

Упражнение 7.2.2.10. (1.0, 2.0). Если центр поворота не совпадает с началом координат, то необходимо сначала переместить исходную точку $c(3,2)$ по вектору, определенному как $-p$, то есть $(-2, -1)$ в точку c' $(1, 1)$. Затем выполняется поворот на 90° против часовой стрелки относительно начала координат для получения точки c'' $(-1, 1)$. Наконец, точка c'' перемещается по вектору p в точку $(1, 2)$ для получения окончательного ответа.

Упражнение 7.2.2.11. Решение (исходный код) приведено ниже.

```
void closestPoint( line 1, point p, point &ans )
{
  line perpendicular;                  // перпендикуляр к линии l, проходящий через точку p
  if( fabs(l.b) < EPS ) {                            // особый случай 1: вертикальная линия
    ans.x = -(l.c); ans.y = p.y; return
  }

  if( fabs(l.a) < EPS ) {                          // особый случай 2: горизонтальная линия
    ans.x = p.x; ans.y = -(l.c); return;
  }

  pointSlopeToLine( p, 1/l.a, perpendicular );                      // обычная прямая линия
  // пересечение прямой l этой перпендикулярной прямой
  // точка пересечения является самой близкой точкой
  areIntersect( l, perpendicular, ans );
}
```

Упражнение 7.2.2.12. Решение (исходный код) приведено ниже. Существуют и другие решения.

```
// Возвращает (по ссылке) отражение точки относительно прямой линии.
void reflectionPoint( line l, point p, point &ans )
{
  point b;
  closestPoint( l, p, b );                                  // аналогично функции distToLine
  vec v = toVector( p, b );                                  // создание вектора перемещения
  ans = translate( translate( p, v ), v );                   // перемещение точки p два раза
}
```

Упражнение 7.2.2.13. 63.43°.

Упражнение 7.2.2.14. Точка p $(3, 7)$ → точка q $(11, 13)$ → точка r $(35, 30)$ – формируется поворот вправо. Следовательно, точка r расположена справа от линии, проходящей через точки p и q.

Примечание: если точка r имеет координаты $(35, 31)$, то все три точки p, q, r лежат на одной прямой.

Упражнение 7.2.3.1. См. рис. 7.14.

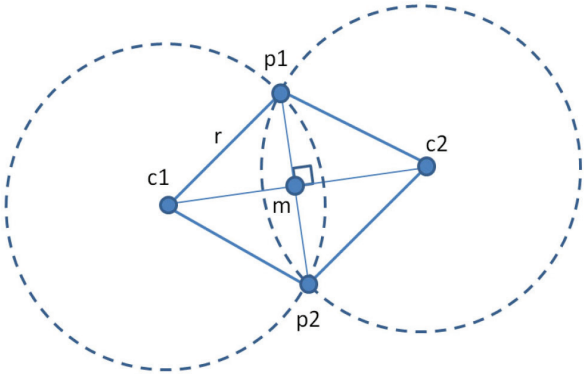


Рис. 7.14 ❖ Построение окружности с заданным радиусом, проходящей через две заданные точки

Пусть *c*1 и *c*2 – центры двух возможных окружностей, проходящих через две заданные точки *p*1 и *p*2, и задан радиус *r*. Четырехугольник *p*1-*c*2-*p*2-*c*1 является ромбом, так как все его четыре стороны равны. Пусть *m* – точка пересечения двух диагоналей этого ромба *p*1-*c*2-*p*2-*c*1. По свойствам ромба точка пересечения диагоналей *m* делит их пополам, и диагонали ромба перпендикулярны друг другу. Можно видеть, что *c*1 и *c*2 вычислимы с помощью масштабирования векторов *mp*1 и *mp*2 в соответствии с отношением (*mc*1/*mp*1) для получения величины, равной *mc*1, а затем повернуть точки *p*1 и *p*2 относительно точки *m* на 90°. В реализации, приведенной в **упражнении 7.2.3.1**, переменная *h* представляет половину отношения *mc*1/*mp*1 (с помощью карандаша и бумаги можно понять, почему *h* можно вычислить таким способом). В двух строках, вычисляющих координаты одного из центров, первые операнды операции сложения – это координаты *m*, а вторые операнды – это результаты масштабирования и поворота вектора *mp*2 относительно точки *m*.

Упражнение 7.2.4.1. Можно использовать тип данных `double`, который предоставляет больший диапазон значений. Но для дальнейшего снижения вероятности переполнения можно переписать формулу Герона в следующем виде: $A = \text{sqrt}(s) \times \text{sqrt}(s - a) \times \text{sqrt}(s - b) \times \text{sqrt}(s - c)$. Хотя при этом результат будет менее точным, поскольку функция вычисления квадратного корня sqrt вызывается 4 раза вместо одного.

Упражнение 7.3.6.1. Поменять местами точки *a* и *b* при вызове функции `cutPolygon(b, a, Q)`.

Упражнение 7.3.7.1. Изменить код функции `ccw` для обеспечения обработки точек, лежащих на одной прямой (коллинеарных точек).

Упражнение 7.3.7.2. Функция `atan2` вычисляет значение, обратное тангенсу *y*/*z*, используя знаки аргументов для правильного определения квадранта.

7.5. Замечания к главе

При написании этой главы использовались материалы, любезно предоставленные доктором Аланом Чен Холун (Dr. Alan Cheng Holun) из Школы информатики (School of Computing) национального университета Сингапура (National University of Singapore). Некоторые библиотечные функции взяты (с внесением небольших изменений) из библиотеки Игоря Навернюка (Igor Naverniouk): http://shygypsy.com/tools/.

По сравнению с первым изданием книги эта глава, как и главы 5 и 6, увеличилась почти вдвое относительно первоначального размера. Но приведенный здесь материал все еще остается неполным, особенно для участников студенческих олимпиад по программированию (ICPC). Если вы готовитесь к студенческой олимпиаде по программированию, то мы рекомендуем одному члену команды внимательно изучить содержимое данной главы. Этот член команды должен точно знать основные геометрические формулы и в совершенстве овладеть расширенными методами вычислительной геометрии, возможно, изучив для этого соответствующие главы следующих книг: [50, 9, 7]. Но теоретических знаний недостаточно, необходима также практическая тренировка для кодирования надежных и правильных геометрических решений, в которых полностью учитывается обработка вырожденных (особых) случаев и ошибки (погрешности) точности вычислений.

Другими методами вычислительной геометрии, которые не рассматривались в этой главе, являются: метод (алгоритм) заметающей прямой (plane sweep), пересечение других геометрических объектов, включая задачи пересечения прямых и отрезков, разнообразные решения по принципу «разделяй и властвуй» для нескольких классических геометрических задач: задачи о паре ближайших точек (closest pair problem), задачи о паре наиболее удаленных точек (furthest pair problem), алгоритм вращающихся кронциркулей (rotating calipers) и т. д. Некоторые из этих задач рассматриваются в главе 9.

Таблица 7.1

Статистические характеристики	Первое издание	Второе издание	Третье издание
Количество страниц	13	22 (+69 %)	29 (+32 %)
Описанные задания	–	20	22 + 9* = 31 (+55 %)
Упражнения по программированию	96	103 (+7 %)	96 (–7 %)

Количество[1] упражнений по программированию в каждом разделе показано в табл. 7.2.

Таблица 7.2

Раздел	Название	Количество	% в главе	% в книге
7.2	Основные геометрические объекты…	74	77	4
7.3	Алгоритмы для многоугольников…	22	23	1

[1] Небольшое уменьшение общего количества упражнений по программированию, несмотря на добавление нескольких новых задач, связано с тем, что некоторые упражнения перенесены в главу 8.

Рис. 7.15 ❖ Фото участников ACM ICPC World Finals, Tokyo 2007

Глава 8

Более сложные темы

«Гений – это один процент вдохновения
и девяносто девять процентов пота».

– Томас Алва Эдисон

8.1. Обзор и мотивация

Эта глава в основном является организационной. Ее первые два раздела содержат более сложный материал, расширяющий главы 3 и 4. В разделах 8.2 и 8.3 рассматриваются более сложные варианты и методы, объясняющие общие принципы решения двух наиболее широко известных классов задач: полный поиск и динамическое программирование. Изложение материала на таком уровне в предыдущих главах, возможно, стало бы отпугивающим фактором для некоторых новых читателей данной книги.

В разделе 8.4 рассматриваются сложные задачи, которые требуют применения более одного алгоритма и/или структуры данных. Этот материал может оказаться затруднительным для программистов-новичков, если они не уделили должного внимания предыдущим главам. Рекомендуется изучить различные (более простые) структуры данных и алгоритмы, прежде чем приступать к чтению текущей главы. Таким образом, лучше внимательно прочитать главы 1–7 до начала чтения раздела 8.4.

Кроме того, мы рекомендуем читателям избегать механического запоминания решений; гораздо более важно попытаться понять основные идеи и принципы этих решений, которые могут оказаться применимыми для других задач.

8.2. Более эффективные методы поиска

В разделе 3.2 рассматривались разнообразные (более простые) итеративные и рекурсивные (с возвратами) методы полного поиска. Но для некоторых задач посложнее требуются более изощренные решения, чтобы избежать превышения лимита времени (TLE). В этом разделе мы рассмотрим некоторые из этих методов с несколькими примерами.

8.2.1. Метод поиска с возвратами с применением битовой маски

В разделе 2.2 мы выяснили, что битовую маску можно использовать для моделирования небольшого набора логических значений. Операции с битовой маской чрезвычайно просты, следовательно, каждый раз, когда необходимо использовать небольшой набор логических значений, можно рассматривать применение метода битовой маски для ускорения решения задачи полного поиска. В данном подразделе рассматриваются два примера.

Еще раз о задаче расстановки N ферзей

В разделе 3.2.2 обсуждалось задание UVa 11195 – Another n-Queen Problem. Но даже после того, как мы усовершенствовали проверки левой и правой диагоналей, сохранив доступность каждой из n горизонталей и $2 \times n - 1$ левых/правых диагоналей в трех структурах `bitset`, мы все равно получили превышение лимита времени. Преобразование этих трех структур `bitset` в три битовые маски немного улучшает ситуацию, но превышение лимита времени остается.

К счастью, существует более эффективный способ проверок горизонталей, левых и правых диагоналей, описанный ниже. Этот метод решения[1] позволяет эффективно выполнять поиск с возвратами с применением битовой маски. Мы будем явно использовать три битовые маски `rw`, `ld` и `rd` для представления состояния поиска. Установленные (равные единице) биты в битовых масках `rw`, `ld` и `rd` указывают, какие горизонтали атакуются на следующей вертикали с учетом горизонтали, левой диагонали или правой диагонали, находящейся под атакой ранее размещенных ферзей, соответственно. Поскольку мы рассматриваем на каждом шаге одну вертикаль, существует только n возможных левых/правых диагоналей; следовательно, можно работать с тремя битовыми масками одной и той же длины в n бит (сравните с $2 \times n - 1$ битами для левых/правых диагоналей в ранее предлагаемом варианте решения в разделе 3.2.2).

Отметим, что хотя оба решения (из раздела 3.2.2 и приведенное здесь) используют одну и ту же структуру данных – три битовые маски, – описанный здесь вариант намного эффективнее. Это лишний раз подчеркивает необходимость рассмотрения возможностей решения задачи с различных точек зрения.

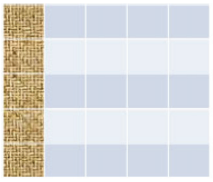
Сначала приведем короткий код предлагаемого поиска с возвратами с применением битовой маски для (обобщенной) задачи расстановки n ферзей при $n = 5$ и рассмотрим подробнее, как он работает.

```
int ans = 0, OK = (1 << 5) - 1;                              // тестирование для n = 5 ферзей

void backtrack( int rw, int ld, int rd )
{
  if( rw == OK ) { ans++; return; }                  // если все биты в rw установлены (=1)
  int pos = OK & (~(rw | ld | rd));             // все установленные биты (1) в pos доступны
  while( pos ) {                                              // этот цикл быстрее, чем O(n)
    int p = pos & -pos;   // выделение наименее значимого бита - это более быстрая операция
```

[1] Несмотря на то что предлагаемое здесь решение предназначено для данной задачи расстановки N ферзей, вероятнее всего, можно использовать части этого решения и для какой-нибудь другой задачи.

```
    pos -= p;                                          // сброс этого установленного бита
    backtrack( rw | p, (ld | p) << 1, (rd | p) >> 1 );                    // особый прием
  }
}

int main()
{
  backtrack( 0, 0, 0 );                                                  // начальный пункт
  printf( "%d\n", ans );                          // для n = 5 должен быть выведен ответ 10
}   // return 0;
```

Для $n = 5$ мы начинаем с состояния `(rw, ld, rd) = (0, 0, 0) = (00000, 00000, 00000)`$_2$. Это состояние показано на рис. 8.1. Переменная `OK = (1 << 5) -1` = $(11111)_2$ используется и для проверки условия завершения процедуры поиска, и для помощи при определении доступных горизонталей для конкретной вертикали. Операция `pos = OK & (~(rw | ld | rd))` объединяет информацию о том, какие горизонтали в следующей вертикали находятся под атакой ранее размещенных ферзей (с учетом атак по горизонтали, по левой или правой диагонали), инвертирует полученный результат и объединяет его с содержимым переменной `OK`, чтобы получить горизонтали, доступные для следующей вертикали. В начальном состоянии доступны все горизонтали на вертикали 0.



pos = 11111 & ~00000 = 11111 (p = 1)

Рис. 8.1 ❖ Задача о расстановке пяти ферзей: начальное состояние

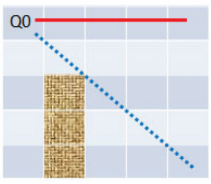
При полном поиске (рекурсивный поиск с возвратами) выполняются попытки с поочередным перебором всех возможных горизонталей (то есть всех установленных в единицу битов в переменной pos) на конкретной вертикали. Ранее в разделе 3.2.1 мы уже видели способ определения всех установленных битов в битовой маске за время $O(n)$.

```
for( p=0; p < n; p++ )                                                    // O(n)
  if( pos && (1 << p) )                    // если этот бит p установлен в переменной pos
    // обработка p
```

Но это не самый эффективный способ. При росте глубины рекурсивного поиска с возвратами остается все меньше горизонталей, доступных для выбора. Вместо попыток перебора всех n горизонталей можно ускорить выполнение приведенного выше цикла, просто проверяя все установленные биты в переменной pos. Предлагаемый ниже цикл выполняется за время $O(k)$.

```
while( pos ) {                  // O(k), где k - число установленных битов в переменной pos
  int p = pos & -pos;                          // определение наименее значимого бита в pos
  pos -= p;                                                               // сброс этого бита
  // обработка p
}
```

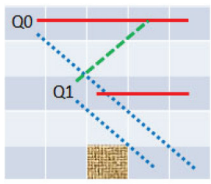
Вернемся к рассмотрению процесса поиска: для pos = $(11111)_2$ самый первый этап поиска начинается с p = pos & -pos = 1 или с горизонтали 0. После размещения первого ферзя (ферзь 0) на горизонтали 0 вертикали 0 эта горизонталь (0) перестает быть доступной для следующей вертикали 1, и этот факт немедленно фиксируется битовой операцией rw | p (а также операциями ld | p и rd | p). Именно в этом и заключается главное преимущество предлагаемого решения. Для левой/правой диагонали увеличивается/уменьшается количество горизонталей, находящихся под атакой при переходе на следующую вертикаль соответственно. Это поведение можно весьма эффективно зафиксировать с помощью операции сдвига влево/вправо: (ld | p) << 1 и (rd | p) >> 1. На рис. 8.2 можно видеть, что для следующей вертикали 1 горизонталь 1 недоступна, потому что ферзь 0 атакует ее по левой диагонали. Теперь для вертикали 1 остаются доступными только горизонтали 2, 3 и 4. Начинаем с горизонтали 2.




```
rw =            = 00001 (1)
ld = 00001 << 1 = 00010 (2)
rd = 00001 >> 1 = 00000|1 (0, наименее значимый бит удаляется)
---------------------- OR
                  00011 -> NEGATE -> 11100
pos = 11111 & 11100 = 11100 (p = 4)
```

Рис. 8.2 ❖ Задача о расстановке пяти ферзей: состояние после размещения первого ферзя

После размещения второго ферзя (ферзь 1) на горизонтали 2 в вертикали 1 горизонталь 0 (из-за размещения в ней ферзя 0) и занятая только что горизонталь 2 становятся недоступными для следующей вертикали 2. Операция сдвига влево для ограничения левой диагонали приводит к недоступности горизонтали 2 (из-за атаки ферзя 0) и (теперь) горизонтали 3 для следующей вертикали 2. Операция сдвига вправо для ограничения правой диагонали приводит к недоступности горизонтали 1 для следующей вертикали 2. Таким образом, только горизонталь 4 остается доступной для следующей вертикали 2, и мы вынуждены выбрать ее для размещения следующего ферзя (см. рис. 8.3).

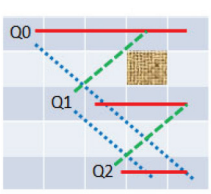



```
rw =            = 00101 (5)
ld = 00110 << 1 = 01100 (12)
rd = 00100 >> 1 = 00010 (2)
---------------------- OR
                  01111 -> NEGATE -> 10000
pos = 11111 & 10000 = 10000 (p = 16)
```

Рис. 8.3 ❖ Задача о расстановке пяти ферзей: состояние после размещения второго ферзя

После размещения третьего ферзя (ферзь 2) на горизонтали 4 в вертикали 2 горизонталь 0 (из-за атаки ферзя 0), горизонталь 2 (из-за атаки ферзя 1) и (теперь) горизонталь 4 становятся недоступными для следующей вертикали 3.

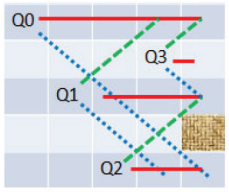
Операция сдвига влево для ограничения левой диагонали приводит к недоступности горизонтали 3 (из-за атаки ферзя 0) и горизонтали 4 (из-за атаки ферзя 1) для следующей вертикали 3 (горизонтали 5 нет – наиболее значимый бит в битовой маске `ld` не используется). Операция сдвига вправо для ограничения правой диагонали приводит к недоступности горизонтали 0 (из-за атаки ферзя 1) и (теперь) горизонтали 3 для следующей вертикали 3. Объединяя всю эту информацию, приходим к выводу: только горизонталь 1 доступна для следующей вертикали 3, поэтому именно там мы размещаем следующего ферзя (см. рис. 8.4).



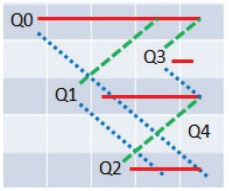

```
rw =            =  10101 (21)
ld = 11100 << 1 = 111000 (56, но наиболее значимый бит не используется)
rd = 10010 >> 1 =  01001 (9)
----------------------- OR
                  111101 -> NEGATE -> 000010
pos = 11111 & 000010 = 00010 (p = 2)
```

Рис. 8.4 ❖ Задача о расстановке пяти ферзей: состояние после размещения третьего ферзя

Аналогичное описание операций применимо для размещения четвертого и пятого ферзей (ферзи 3 и 4), как показано на рис. 8.5. Этот процесс можно продолжать для получения других 9 решений при $n = 5$.




```
rw =            =   10111 (23)
ld = 111010 << 1 = 1110100 (116, два наиболее значимых бита не используются)
rd =  01011 >> 1 =   00101 (5)
------------------------- OR
                  1110111 -> NEGATE -> 0001000
pos = 11111 & 0001000 = 01000 (p = 8)
```




```
rw =            =   11111 (получено одно решение)
```

Рис. 8.5 ❖ Задача о расстановке *N* ферзей: состояние после размещения четвертого и пятого ферзей

Используя описанный выше метод, можно решить задание UVa 11195. Необходимо лишь изменить приведенный выше код с учетом «плохих» клеток, которые также можно смоделировать в форме битовых масок. Попробуем выполнить приблизительный анализ наихудшего варианта для доски $n \times n$ без «плохих» клеток. Предположив, что при таком рекурсивном поиске с возвра-

тами с применением битовой маски на каждом шаге имеется приблизительно на две доступные горизонтали меньше, получим временную сложность $O(n!!)$, где $n!!$ – обозначение мультифакториала (двойного факториала). При $n = 14$ без «плохих» клеток решение рекурсивным поиском с возвратами из раздела 3.2.2 потребует до $O(14!) \approx 87\,178$ млн операций, тогда как для реализации предложенного в этом разделе рекурсивного поиска с возвратами с применением битовых масок потребуется лишь около $O(14!!) = 14 \times 12 \times 10 \times \ldots \times 2 = 645\,120$ операций.

Битовая матрица смежности

Задание UVa 11065 – Gentlemen Agreement – задача о джентльменском соглашении сводится к вычислению двух целых чисел: числа независимых наборов (Independent Set) и размера максимального независимого набора (Maximum Independent Set – MIS) (описание этой задачи см. в разделе 4.7.4) заданного произвольного графа с количеством вершин $V \leqslant 60$. Поиск максимального независимого набора (MIS) для обобщенного графа является NP-трудной задачей. Следовательно, поиск полиномиального алгоритма для решения этой задачи – дело безнадежное.

Один из вариантов решения – приведенный ниже усовершенствованный рекурсивный алгоритм поиска с возвратами. Состояние поиска представлено тройкой параметров (i, used, depth). Первый параметр i подразумевает, что мы можем рассматривать вершины в диапазоне $[i..V - 1]$ как включенные в независимый набор. Второй параметр used – это битовая маска с длиной V бит, которая определяет, какие вершины больше не являются доступными для использования в текущем независимом наборе, поскольку по крайней мере одна из их вершин-соседей уже была включена в этот независимый набор. Третий параметр depth сохраняет глубину рекурсии, которая также является размером текущего независимого набора.

Существует хитрый прием обработки битовой маски для этой задачи, который можно использовать для существенного ускорения ее решения. Отметим, что исходный граф имеет небольшой размер $V \leqslant 60$. Поэтому можно хранить исходный граф в виде матрицы смежности с размером не более $V \times V$ (для рассматриваемой здесь задачи для всех элементов на главной диагонали матрицы смежности устанавливается значение true (истина)). Но можно упаковать одну строку из V логических значений ($V \leqslant 60$) в битовую маску, используя для нее 64-битовое знаковое целое число.

С помощью такой компактной матрицы смежности AdjMat, которая содержит всего лишь V строк 64-битовых целых чисел, можно воспользоваться быстрыми операциями с битовой маской для реализации эффективной процедуры пометки соседних (смежных) вершин. Если мы решаем определить как свободную вершину i, то есть `(used & (1 << i)) == 0`, то при этом глубина depth увеличивается на единицу, а затем применяется операция с битовой маской O(1): `used | AdjMat[i]` для пометки всех соседних (смежных) вершин, включая и саму эту вершину i (напомним, что `AdjMat[i]` – это также битовая маска с длиной V бит, в которой i-й бит установлен).

Если все биты в маске used установлены (равны 1), это означает, что мы нашли еще один независимый набор. Мы также записываем наибольшее значе-

ние глубины depth во время текущего процесса, так как это размер максимального независимого набора исходного графа. Самые главные фрагменты кода реализации предлагаемого здесь решения показаны ниже.

```
void rec( int i, long long used, int depth )
{
  if( used == (1 << V) - 1 ) {                              // посещаются все пересечения
    nS++;                                                // еще один возможный независимый набор
    mxS = max( mxS, depth );                                          // размер этого набора
  }
  else {
    for( int j=i; j < V; j++ )
      if( !(used & (1 << j)) )          // если пересечение j пока еще не было использовано
        rec( j+1, used | AdjMat[j], depth+1 );                  // быстрая битовая операция
  }
}

// в функции int main()
  // более функциональный, использующий биты список смежных вершин
  // (для ускорения операций установки битов)
  for( int i=0; i < V; i++ )
    AdjMat[i] = (1 << i);        // установка самой вершины i (для каждой вершины в наборе)
  for( int i=0; i < E; i++ ) {
    scanf( "%d %d", &a, &b );
    AdjMat[a] |= (1 << b);
    AdjMat[b] |= (1 << a);
}
```

Упражнение 8.2.1.1*. Головоломка судоку – это еще одна *NP*-полная задача. Рекурсивный поиск с возвратами для поиска одного решения для стандартного 9×9 ($n = 3$) поля судоку можно ускорить, используя битовую маску. В каждую пустую ячейку (r, c) мы пытаемся методом поочередного перебора поместить число из диапазона $[1..n^2]$, если это допустимый ход. Проверки n^2 строк, n^2 столбцов и квадрата $n \times n$ можно выполнять с помощью трех битовых масок с длиной n^2 бит. Применяя этот метод, решите две похожие задачи: UVa 989 и UVa 10957.

8.2.2. Поиск с возвратами с интенсивным отсечением

Задача I – Robots on Ice, предложенная в финале мирового чемпионата ACM ICPC World Finals 2010, может рассматриваться как «надежный и жесткий тест стратегии отсечения». Задача описывается просто: задана доска $M \times N$ с тремя контрольными пунктами $\{A, B, C\}$, требуется найти гамильтонов цикл[1] с длиной $(M \times N)$ от координаты (0, 0) до координаты (0, 1). Этот гамильтонов цикл (путь) обязательно должен пройти три контрольных пункта A, B и C в одной

[1] Гамильтонов цикл (Hamiltonian path) – это путь в неориентированном (ненаправленном) графе, при котором вершины посещаются только по одному разу.

четвертой, половине и трех четвертых своего пути соответственно. Ограничения: $2 \leqslant M, N \leqslant 8$.

Пример: если задана доска 3×6 с контрольными пунктами A = (строка, столбец) = (2, 1), B = (2, 4) и C = (0, 4), как показано на рис. 8.6, то существуют два возможных пути.

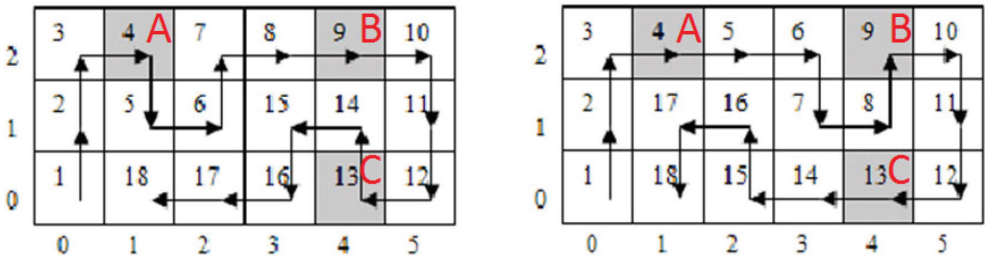


Рис. 8.6 ❖ Визуальное представление решения задания UVa 1098 – Robots on Ice

Простейший рекурсивный алгоритм поиска с возвратами приводит к превышению лимита времени (TLE), так как на каждом шаге имеется четыре варианта выбора, а максимальная длина пути равна $8 \times 8 = 64$ в предельном тестовом варианте. Попытка перебора всех 4^{64} возможных путей неприемлема. Для ускорения работы этого алгоритма необходимо отсечение пространства поиска, если процесс поиска:

1) выходит за пределы сетки (доски) M×N (очевидно);
2) не посещает соответствующий целевой контрольный пункт на 1/4, 1/2 или 3/4 пути – наличие этих трех контрольных пунктов в действительности сужает пространство поиска;
3) попадает в целевой контрольный пункт раньше, чем предписано условием;
4) не может достигнуть следующего контрольного пункта вовремя из текущей позиции;
5) не может перейти в следующий пункт с конкретными координатами, так как текущий частичный пройденный путь блокирует доступ к пункту с этими координатами. Это можно проверить простым поиском в глубину или в ширину (см. раздел 4.2). Сначала выполняется поиск в глубину или в ширину из целевого пункта с координатами (0, 1). Если на сетке (доске) M×N существуют пункты, которые недостижимы из (0, 1) и пока еще не были посещены текущим частичным путем, то этот текущий частичный путь можно отсечь.

Упражнение 8.2.2.1*. Пять пунктов стратегии отсечения, перечисленных выше в этом разделе, работают успешно, но в действительности этого недостаточно, чтобы уложиться в лимит времени, установленный для заданий LA 4793 и UVa 1098. Существует более быстрое решение этой задачи, использующее методику встречи в середине (см. раздел 8.2.4). Этот пример показывает, как

выбор устанавливаемого лимита времени может определять, какое из решений полного поиска считается достаточно быстрым. Предлагается тщательно изучить сущность методики встречи в середине в разделе 8.2.4 и применить ее для решения задачи Robots on Ice.

8.2.3. Поиск в пространстве состояний с применением поиска в ширину или алгоритма Дейкстры

В разделах 4.2.2 и 4.4.3 рассматривались два стандартных алгоритма на графах для решения задачи о поиске кратчайшего пути из одной вершины во все остальные (Single Source Shortest Paths – SSSP). Поиск в ширину (BFS) можно использовать, если граф невзвешенный, тогда как алгоритм Дейкстры следует использовать, если граф взвешенный. Задачи о поиске кратчайшего пути из одной вершины во все остальные (SSSP) остаются более простыми в том смысле, что в большинстве случаев мы можем с легкостью увидеть «граф» в описании задачи. Но это перестает быть справедливым для некоторых более трудных задач поиска в графе, рассматриваемых в этом разделе, в которых (обычно неявно подразумевается, что) графы не так-то просто увидеть, а состояние/набор вершин может быть сложным объектом. В этом случае вместо названия задачи о поиске кратчайшего пути из одной вершины во все остальные (SSSP) мы обычно обозначаем задачу как «поиск в пространстве состояний» (State Space Search).

Если состояние является сложным объектом, например парой (позиция, битовая маска), как в задании UVa 321 – The New Villa, четверкой (строка, столбец, направление, цвет), как в задании UVa 10047 – The Monocycle и т. д., то мы обычно не используем переменную `vector<int> dist` для хранения информации о расстоянии, как в стандартной реализации поиска в ширину или алгоритма Дейкстры. Причина в том, что состояния, подобные описанным выше, может быть не так-то просто преобразовать в целочисленные индексы. Одно из возможных решений – использование `map<VERTEX-TYPE, int> dist`. Этот прием добавляет (небольшой) множитель $\log V$ в оценку временной сложности алгоритма поиска в ширину и алгоритма Дейкстры. Но для сложного поиска в пространстве состояний эти дополнительные накладные расходы во время выполнения могут быть приемлемыми по сравнению с общей сложностью кодирования. В этом подразделе мы рассмотрим один пример такого сложного поиска в пространстве состояний.

Упражнение 8.2.3.1. Как хранить VERTEX-TYPE, если этот тип представлен парой, тройкой или четверкой информационных элементов, при использовании языков C++ и Java?

Упражнение 8.2.3.2. Тот же вопрос, что и в **упражнении 8.2.3.1**, но VERTEX-TYPE является еще более сложным объектом, например массивом.

Упражнение 8.2.3.3. Возможно ли свести рассматриваемую здесь задачу поиска в пространстве состояний к задаче максимизации?

UVa 11212 – Editing a Book

Сокращенное описание задачи: задано n абзацев, пронумерованных от 1 до n; необходимо разместить их в порядке возрастания номеров $1, 2, \ldots, n$. Используя буфер обмена (clipboard), вы можете нажимать клавиши **Ctrl**+**X** (вырезание) и **Ctrl**+**V** (вставка) несколько раз. Нельзя вырезать дважды перед вставкой, но можно вырезать несколько смежных абзацев за одну операцию, а потом вставить эти абзацы в том же порядке. Какое минимальное число операций (шагов) потребуется для решения данной задачи?

Пример 1: для сортировки последовательности {2, 4, (1), 5, 3, 6} вырезаем абзац (1) и вставляем его перед абзацем 2, чтобы получить {1, 2, 4, 5, (3), 6}. Затем вырезаем абзац (3) и вставляем его перед абзацем 4, чтобы получить {1, 2, 3, 4, 5, 6}. Ответ: два шага.

Пример 2: для сортировки последовательности {(3, 4, 5), 1, 2} вырезаем одновременно три абзаца (3, 4, 5) и вставляем их после абзаца 2, чтобы получить {1, 2, 3, 4, 5}. Задача решается за единственный шаг. Это не единственное решение, так как можно было бы поступить следующим образом: одновременно вырезать два абзаца (1, 2), вставить их перед абзацем 3 и получить требуемый результат {1, 2, 3, 4, 5} также за один шаг.

Грубая верхняя граница числа шагов, требуемых для переупорядочивания предложенных n абзацев, равна $O(k)$, где k – число абзацев, изначально находящихся на неправильных позициях. Это может послужить обоснованием использования следующего «простейшего» алгоритма (что неверно): вырезать один абзац, находящийся в неправильной позиции, и вставить его в правильную позицию. Но такой подход может оказаться не самым коротким способом решения.

Например, этот предлагаемый «простейший» алгоритм обработает последовательность {5, 4, 3, 2, 1} следующим образом: {(5), 4, 3, 2, 1} → {(4), 3, 2, 1, 5} → {(3), 2, 1, 4, 5} → {(2), 1, 3, 4, 5} → {1, 2, 3, 4, 5} – всего 4 операции вырезать-вставить. Это не оптимальный вариант, поскольку можно решить этот пример всего лишь за три шага: {5, 4, (3, 2), 1} → {3, (2, 5), 4, 1} → {3, 4, (1, 2), 5} → {1, 2, 3, 4, 5}.

Данная задача имеет огромное пространство поиска, даже для случая с относительно небольшим числом абзацев $n = 9$. Практически невозможно вычислить ответ вручную, например мы вряд ли начнем рисовать дерево рекурсии только для того, чтобы проверить и убедиться в том, что необходимо не менее четырех шагов для сортировки последовательности {5, 4, 9, 8, 7, 3, 2, 1, 6} и не менее пяти шагов для сортировки последовательности {9, 8, 7, 6, 5, 4, 3, 2, 1}.

Состоянием этой задачи является перестановка абзацев. Существует максимальное число $O(n!)$ перестановок абзацев. При максимальном значении $n = 9$ в условии задачи число перестановок равно 9!, или 362 880. Таким образом, количество вершин в графе пространства состояний в действительности не так уж велико.

Сложность этой задачи заключается в количестве ребер графа пространства состояний. Приняв перестановку длины n (вершина), получаем ${}_nC_2$ возможных вырезаемых точек (индекс $i, j \in [1..n]$) и n возможных вставляемых точек (индекс $k \in [1..(n - (j - i + 1))]$). Следовательно, для каждой из $O(n!)$ вершин существует приблизительно $O(n^3)$ ребер, соединенных с ними.

В действительности в задаче ставится вопрос о поиске кратчайшего пути из исходной вершины/состояния (исходная перестановка) в целевую вершину (полностью отсортированная перестановка) в этом невзвешенном, но огромном графе пространства состояний. Наихудший вариант поведения возникает в том случае, если мы выполняем один этап $O(V + E)$ поиска в ширину в этом графе пространства состояний, а его сложность равна $O(n! + (n!*n^3)) = O(n!*n^3)$. Для $n = 9$ потребуется 9! * 93 = 264 539 520 ≈ 265 млн операций. Такое решение, вероятнее всего, приведет к превышению лимита времени (TLE) (или, возможно, к превышению лимита памяти (MLE)).

Необходимо более эффективное решение, которое будет рассмотрено в следующем разделе 8.2.4.

8.2.4. Встреча в середине (двунаправленный поиск)

В некоторых задачах о поиске кратчайшего пути из одной вершины во все остальные (SSSP) (но обычно в задачах поиска в пространстве состояний) в сверхбольшом графе, когда нам известны две вершины: исходная вершина / состояние *s* и целевая вершина / состояние *t*, существует возможность существенно снизить временную сложность поиска, выполняя поиск с обоих направлений в надежде на то, что произойдет встреча в середине (meet in the middle). Мы проиллюстрируем эту методику, продолжая обсуждение трудной задачи UVa 11212.

Прежде чем продолжить, необходимо сделать замечание: метод встречи в середине не всегда обращается к двунаправленному поиску в ширину. Это задача выбора стратегии решения при «поиске с двух направлений / из двух частей», которая может возникнуть в другой форме в любой сложной задаче поиска, например см. **упражнение 3.2.1.4***.

UVa 11212 – Editing a Book, новый подход

Несмотря на то что наихудший вариант временной сложности для этой задачи при поиске в пространстве состояний неприемлем, наибольший возможный ответ для этой задачи невелик. При выполнении поиска в ширину для максимального тестового варианта $n = 9$ при поиске от целевого состояния *t* (полностью отсортированная перестановка {1, 2, …, 9}) до достижения всех прочих состояний обнаруживается, что для этой задачи максимальная глубина поиска в ширину при $n = 9$ равна 5 (после выполнения в течение нескольких минут – в условиях олимпиадного программирования это превышение лимита времени (TLE)).

Эта важная информация позволяет нам выполнить двунаправленный поиск в ширину, выбирая только проход с глубиной 2 с каждого направления. Хотя данная информация не является необходимым условием для выполнения двунаправленного поиска в ширину, она может помочь сократить пространство поиска.

Существует три возможных варианта, которые мы более подробно рассмотрим ниже.

Вариант 1: вершина *s* находится в пределах двух шагов от вершины *t* (см. рис. 8.7).

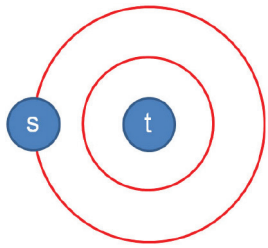


Рис. 8.7 ❖ Вариант 1: пример, в котором *s* находится на расстоянии двух шагов от *t*

Сначала выполняется поиск в ширину (максимальная глубина поиска в ширину = 2) от целевой вершины *t* для записи информации о расстоянии от вершины *t*: `dist_t`. Если исходная вершина *s* уже найдена, то есть `dist_t[s]` не содержит `INF`, то мы возвращаем это значение. Возможные ответы: 0 (если *s* = *t*), 1 или 2 шага.

Вариант 2: вершина *s* находится в пределах трех-четырех шагов от вершины *t* (см. рис. 8.8).

Если не удалось найти исходную вершину *s* после выполнения варианта 1, описанного выше, то есть `dist_t[s] = INF`, то становится понятно, что *s* расположена на более удаленном расстоянии от вершины *t*. Теперь поиск в ширину выполняется от исходной вершины *s* (также с максимальной глубиной поиска = 2) для записи информации о расстоянии от вершины *s*: `dist_s`. Если во время этого второго этапа поиска в ширину обнаруживается общая вершина *v* «в середине», то теперь нам известно, что вершина *v* расположена на расстоянии двух уровней (шагов) и от вершины *t*, и от вершины *s*. Следовательно, ответ: `dist_s[v] + dist_t[v]` шагов. Возможные ответы: 3 или 4 шага.

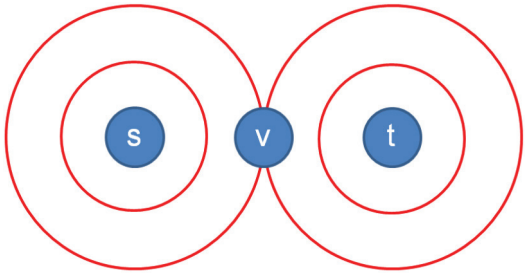


Рис. 8.8 ❖ Вариант 2: пример, в котором *s* находится на расстоянии четырех шагов от *t*

Вариант 3: вершина *s* находится в точности в пяти шагах от вершины *t* (см. рис. 8.9).

Если не найдена какая-либо общая вершина *v* после выполнения второго этапа поиска в ширину в варианте 2, описанном выше, то ответом является точное расстояние в 5 шагов, поскольку нам заранее известно, что вершины *s* и *t* всегда непременно должны быть достижимыми. Прекращение поиска на

глубине 2 позволяет нам не выполнять вычисления на глубине 3, которые занимают намного больше времени, чем вычисления с глубиной поиска 2.

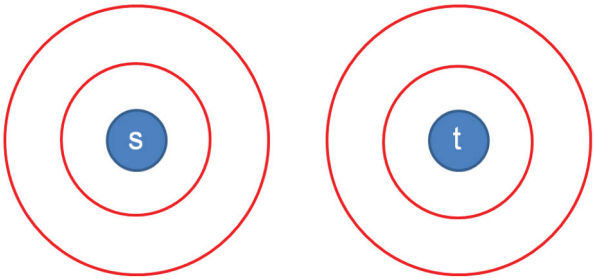


Рис. 8.9 ❖ Вариант 3: пример, в котором *s* находится на расстоянии пяти шагов от *t*

Здесь мы убедились в том, что при заданной перестановке длины n (вершина) существует около $O(n^3)$ ветвей в этом огромном графе пространства состояний. Но если просто выполнять каждый этап поиска в ширину с максимальной глубиной 2, то на каждом этапе выполняется всего лишь не более $O((n^3)^2) = O(n^6)$ операций. При $n = 9$ выполняется $9^6 = 531\,441$ операция (это больше, чем 9!, поскольку имеются некоторые перекрытия). Так как целевая вершина t не изменяется на протяжении всего процесса поиска в пространстве состояний, можно выполнить самый первый этап поиска в ширину от вершины t только один раз. Затем выполняется второй этап поиска в ширину от исходной вершины s для каждого запроса. Наша реализация поиска в ширину будет содержать дополнительный логарифмический множитель из-за использования табличной структуры данных (например, `map`) для хранения значений `dist_t` и `dist_s`. Такое решение является приемлемым (Accepted).

8.2.5. Поиск, основанный на имеющейся информации: A* и IDA*

*Основы поиска A**

Рис. 8.10 ❖ Головоломка 15 (пятнашки)

Алгоритмы полного поиска, которые рассматривались в главах 3 и 4, а также в предыдущих подразделах этого раздела текущей главы, являются «неинформированными», то есть все возможные состояния, достижимые из текущего состояния, одинаково хороши». В некоторых задачах доступна более подробная информация (отсюда и название «поиск, основанный на имеющейся информации»), поэтому можно воспользоваться интеллектуальным методом поиска A*, который применяет эвристики для «управления» направлением поиска.

Рассмотрим метод поиска A* на примере хорошо известной задачи о решении головоломки 15 (пятнашки). Головоломка состоит из 15 передвигаемых фишек, пронумерованных от 1 до 15. Фишки размещены в коробочке размером 4×4, то есть одно место остается свободным. Возможные действия ограничены перемещением на свободное место одной из фишек, расположенных рядом с этим свободным местом. Можно рассматривать эти действия с другой точки зрения: «перемещение свободной ячейки вправо, вверх, влево или вниз». Цель головоломки – передвигая фишки, расположить их в порядке возрастания номеров по рядам, как показано на рис. 8.10, то есть достичь «целевого» состояния.

Эта небольшая головоломка выглядит просто, но становится настоящей головной болью для различных алгоритмов поиска из-за огромного пространства поиска. Можно представить любое состояние данной головоломки, поместив числа на фишках строка за строкой слева направо в массив из 16 целых чисел. Для упрощения присвоим значение 0 свободной ячейке, тогда целевое состояние выглядит следующим образом: {1, 2, 3, …, 14, 15, 0}. Для любого произвольно взятого состояния может существовать до четырех достижимых состояний в зависимости от положения свободной ячейки. Если свободная ячейка находится в одном из четырех углов, то возможны две операции (перемещения). Если свободная ячейка находится в одной из восьми граничных неугловых позиций, то возможны три операции (перемещения). В четырех центральных позициях для свободной ячейки возможны четыре операции (перемещения). Это гигантское пространство поиска.

Но эти состояния не равноценны. Для решения данной задачи существует эффективная эвристика, которая может помочь в управлении алгоритмом поиска. По этой эвристике вычисляется сумма манхэттенских расстояний[1] между каждой фишкой (кроме свободной ячейки) в текущем состоянии и ее позицией в целевом состоянии. Эта эвристика дает нижнюю границу количества шагов для достижения целевого состояния. Объединяя стоимость достигнутого на текущий момент состояния (обозначенного как $g(s)$) и значение, полученное в результате применения эвристики (обозначенное как $h(s)$) для состояния s, мы получаем более качественную информацию о направлении следующего хода. Продемонстрируем применение предлагаемой эвристики на примере головоломки с начальным состоянием A, показанным ниже:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{\mathbf{0}} \\ 13 & 14 & 15 & 12 \end{bmatrix};$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & \underline{\mathbf{0}} \\ 9 & 10 & 11 & \underline{\mathbf{8}} \\ 13 & 14 & 15 & 12 \end{bmatrix}; \quad C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & \underline{\mathbf{0}} & \underline{\mathbf{11}} \\ 13 & 14 & 15 & 12 \end{bmatrix}; \quad D = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & \underline{\mathbf{12}} \\ 13 & 14 & 15 & \underline{\mathbf{0}} \end{bmatrix}.$$

[1] Манхэттенское расстояние (расстояние городских кварталов) между двумя точками – это сумма абсолютных разностей (модулей разностей) координат этих точек.

Стоимость начального состояния A $g(s) = 0$, так как пока еще не сделано ни одного хода. Для этого состояния A существуют три достижимых состояния $\{B, C, D\}$ со значениями $g(B) = g(C) = g(D) = 1$, то есть достижимых за один ход. Но эти три состояния не одинаково хороши:

1) если свободная ячейка 0 передвигается вверх, то значение эвристики $h(B) = 2$, так как для фишек 8 и 12 манхэттенское расстояние равно 1. Получаем $g(B) + h(B) = 1 + 2 = 3$;
2) если свободная ячейка 0 передвигается влево, то значение эвристики $h(C) = 2$, так как для фишек 11 и 12 манхэттенское расстояние равно 1. Получаем $g(C) + h(C) = 1 + 2 = 3$;
3) но если передвинуть свободную ячейку вниз, то получим значение эвристики $h(D) = 0$, так как все фишки находятся в правильных позициях. Таким образом, $g(D) + h(D) = 1 + 0 = 1$ – комбинация с минимальным значением.

Если рассматривать состояния в порядке возрастания значений $g(s) + h(s)$, то состояния с меньшей ожидаемой стоимостью будут оцениваться в первую очередь, как состояние D в приведенном выше примере – целевое состояние. Именно в этом заключается сущность алгоритма поиска A*.

Обычно такое упорядочение состояний реализуется с помощью очереди с приоритетами, что делает реализацию алгоритма поиска A* очень похожей на реализацию алгоритма Дейкстры, представленную в разделе 4.4. Отметим, что если установить для всех состояний $h(s) = 0$, то алгоритм A* сводится все к тому же алгоритму Дейкстры.

Поскольку эвристическая функция $h(s)$ никогда не переоценивает действительное расстояние до целевого состояния (поэтому ее также называют допустимой эвристической оценкой – admissible heuristic), этот алгоритм поиска A* является оптимальным. Самый сложный этап при решении задач с использованием алгоритма поиска A* – выбор правильной допустимой эвристической оценки.

*Ограничения алгоритма поиска A**

При использовании алгоритма поиска A* (а также алгоритмов Дейкстры и поиска в ширину на больших графах пространства состояний) с применением очередей (с приоритетами) главной проблемой становится слишком большая потребность в оперативной памяти, когда целевое состояние находится очень далеко от начального состояния. Для некоторых сложных задач поиска, возможно, потребуется применение вспомогательных методов, описанных ниже.

Поиск с ограниченной глубиной

В разделе 3.2.2 рассматривался рекурсивный алгоритм поиска с возвратами. Главная проблема чистого поиска с возвратами: возможное вовлечение в исследование чрезвычайно глубокого пути, который не приводит к решению, а возврат из этого пути происходит только после существенных затрат ценного времени выполнения.

Поиск с ограниченной глубиной (depth limited search – DLS) устанавливает предел глубины, с которой может быть выполнен возврат из пути. Если глуби-

на поиска превышает установленную, то этот алгоритм останавливает более глубокое проникновение по текущему исследуемому пути. Если предельная глубина оказывается равной глубине самого «мелкого» целевого состояния, то метод поиска с ограниченной глубиной работает быстрее, чем обобщенная подпрограмма поиска с возвратами. Но если предельная глубина слишком мала, то целевое состояние становится недостижимым. Если в условии задачи указано, что целевое состояние находится «не более чем в d шагах» от исходного состояния, то применение поиска с ограниченной глубиной вместо обобщенного метода поиска с возвратами вполне оправдано.

Поиск с итеративным углублением

Если поиск с ограниченной глубиной применяется неправильно, то целевое состояние становится недостижимым, хотя известно, что решение существует. Поиск с ограниченной глубиной обычно используется не как самостоятельный независимый метод, а как часть поиска с итеративным углублением (iterative deepening search – IDS).

Поиск с итеративным углублением вызывает метод поиска с ограниченной глубиной, постепенно увеличивая предельную глубину до тех пор, пока не будет найдено целевое состояние. Таким образом, поиск с итеративным углублением является полным и оптимальным. Поиск с итеративным углублением представляет собой превосходную стратегию, устраняющую проблемы с определением наилучшего значения предельной глубины, поскольку выполняет попытки с постепенным наращиванием всех возможных значений предельной глубины: сначала глубина 0 (собственно начальное состояние), затем глубина 1 (все состояния, в которые можно перейти за один шаг из начального состояния), потом глубина 2 и т. д. Действуя таким способом, алгоритм поиска с итеративным углублением объединяет преимущества более простого и менее требовательного к объему памяти алгоритма поиска в глубину и возможности алгоритма поиска в ширину, обеспечивающие посещение всех соседних состояний уровень за уровнем (см. табл. 4.2 в разделе 4.2).

Несмотря на то что алгоритм поиска с итеративным углублением вызывает процедуру поиска с ограниченной глубиной многократно, временная сложность остается равной $O(b^d)$, где b – множитель (степень) разветвления, а d – глубина самого «мелкого» целевого состояния. Обоснование: $O(b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + \ldots + (b^0 + b^1 + b^2 + \ldots + b^d)) \leqslant O(c \times b^d) = O(b^d)$.

Поиск с итеративным углублением A (IDA*)*

Для более быстрого решения задачи о пятнашках можно воспользоваться алгоритмом поиска с итеративным углублением A* (iterative deepening A* – IDA*), который по существу представляет собой поиск с итеративным углублением с применением модифицированного поиска с ограниченной глубиной. Поиск с итеративным углублением A* (IDA*) вызывает метод модифицированного поиска с ограниченной глубиной, чтобы выполнить попытки перехода во все соседние состояния в фиксированном порядке (то есть перемещать фишку 0 – свободную ячейку – вправо, затем вверх, потом влево, наконец, вниз – именно в этом порядке; здесь мы не используем очередь с приоритетами). Этот моди-

фицированный алгоритм поиска с ограниченной глубиной останавливается не в том случае, когда превышена предельная глубина, а когда значение $g(s) + h(s)$ превосходит наилучшее известное до сего момента решение. Алгоритм IDA* постепенно увеличивает предельную глубину до тех пор, пока не будет найдено целевое состояние.

Реализация алгоритма поиска с итеративным углублением A* не столь проста и очевидна, поэтому мы предлагаем читателям внимательно изучить исходный код, представленный на нашем вспомогательном веб-сайте.

Файл исходного кода: *ch8_01_UVa10181.cpp/java*

Упражнение 8.2.5.1*. Одним из самых сложных этапов решения задач поиска с использованием метода поиска A* является подбор правильной допустимой эвристической оценки для эффективного вычисления значений эвристики, и эту процедуру необходимо повторять много раз. Перечислите допустимые эвристические оценки, которые чаще всего используются при решении сложных задач поиска с применением алгоритма A*, и покажите, как нужно эффективно вычислять эти эвристические оценки. Одной из таких эвристик является махэттенское расстояние, рассматриваемое в этом разделе.

Упражнение 8.2.5.2*. Выполните задание UVa 11212 – Editing a Book, которое подробно рассматривалось в разделах 8.2.3–8.2.4, используя алгоритм поиска A* вместо двунаправленного поиска в ширину. Совет: сначала определите наиболее подходящую эвристику для этой задачи.

Задания по программированию, которые решаются с помощью усовершенствованных методов поиска

- Более трудные задачи поиска с возвратами
  1. *UVa 00131 – The Psychic Poker Player* (поиск с возвратами с использованием битовой маски 2^5, чтобы помочь с принятием решения: какую карту оставить на руках или обменять с самой верхней картой в колоде; использовать 5! для перемешивания (тасования) 5 карт на руках и получения наилучшего значения)
  2. *UVa 00710 – The Game* (поиск с возвратами с использованием мемоизации[1]/отсечения)
  3. *UVa 00711 – Dividing up* (перед поиском с возвратами сокращается пространство поиска)
  4. *UVa 00989 – Su Doku* (классическая головоломка судоку; такая задача NP-полна, но тем не менее разрешима с использованием поиска с отсечением; используется битовая маска для ускорения процедуры проверки доступных чисел)

[1] Мемоизация (запоминание) – кеширование результатов выполнения функций, чтобы избежать повторных вычислений и вызовов функций (специализированная методика оптимизации). – *Прим. перев.*

5. *UVa 01052 – Bit Compression* (LA 3565 – WorldFinals SanAntonio06, поиск с возвратами с использованием некоторой формы битовой маски)
6. **UVa 10309 – Turn the Lights Off** * (метод грубой силы для первого ряда 2^{10}; остальные этапы решения следуют из первого)
7. *UVa 10318 – Security Panel* (порядок не важен; поэтому можно пробовать нажимать кнопки в возрастающем порядке, ряд за рядом, столбец за столбцом; нажатие одной кнопки воздействует только на область 3×3 вокруг нее; следовательно, при нажатии кнопки (i, j) обязательно должен включиться световой индикатор (i – 1, j – 1) (в дальнейшем ни одна из кнопок не воздействует на этот световой индикатор); эту проверку можно использовать для отсечения при поиске с возвратами)
8. *UVa 10890 – Maze* (выглядит как задача динамического программирования, но состояние, включающее битовую маску, невозможно запомнить (мемоизировать); к счастью, размер сетки мал)
9. *UVa 10957 – So Doku Checker* (очень похоже на задание UVa 00989; если вы выполнили задание «Судоку», то можете немного изменить его код для решения этого задания)
10. **UVa 11195 – Another n-Queen Problem** * (см. упражнение 3.2.1.3* и описание в разделе 8.2.1)
11. **UVa 11065 – A Gentlemen's Agreement** * (независимый набор, битовая маска помогает ускорить решение; см. описание в разделе 8.2.1)
12. *UVa 11127 – Triple-Free Binary String* (свободные от утроений бинарные строки) (поиск с возвратами с использованием битовой маски)
13. *UVa 11464 – Even Parity* (метод грубой силы для первого ряда 2^{15}; остальные этапы решения следуют из первого)
14. *UVa 11471 – Arrange the Tiles* (сокращение пространства поиска с помощью группирования фишек одного типа; рекурсивный поиск с возвратами)

- Более трудные задачи поиска в пространстве состояний с использованием поиска в ширину или алгоритма Дейкстры
  1. UVa 00321 – The New Villa (*состояние*: (позиция, битовая маска 2^{10}), вывести путь)
  2. *UVa 00658 – It's not a Bug...* (*состояние*: битовая маска – вне зависимости от наличия или отсутствия ошибки используется алгоритм Дейкстры, так как граф пространства состояний является взвешенным)
  3. UVa 00928 – Eternal Truths (*состояние*: (строка, столбец, направление, шаг))
  4. **UVa 00985 – Round and Round...** * (4 поворота одинаковы и совпадают с нулевым (0) поворотом, *s*: (строка, столбец, поворот = [0..3]): найти самый короткий путь из состояния [1][1][0] в состояние [R][C][x], где $0 \leqslant x \leqslant 3$)
  5. *UVa 01057 – Routing* (LA 3570, World Finals San Antonio06, используется алгоритм Флойда–Уоршелла (Floyd-Warshall) для получения ин-

формации о кратчайшем пути между всеми парами вершин (APSP); затем исходная задача моделируется как другая задача поиска кратчайшего пути из одной вершины во все остальные (SSSP) во взвешенном графе, разрешимая с помощью алгоритма Дейкстры)

6. UVa 01251 – Repeated Substitution… (LA 4637, Tokyo09, задача поиска кратчайшего пути из одной вершины во все остальные (SSSP), разрешимая с применением алгоритма поиска в ширину)
7. UVa 01253 – Infected Land (LA 4645, Tokyo09, задача поиска кратчайшего пути из одной вершины во все остальные (SSSP), разрешимая с применением алгоритма поиска в ширину и со скучным моделированием состояний)
8. UVa 10047 – The Monocycle (*s*: (строка, столбец, направление, цвет); алгоритм поиска в ширину)
9. *UVa 10097 – The Color game* (*s*: (N1, N2); неявный невзвешенный граф; алгоритм поиска в ширину)
10. *UVa 10923 – Seven Seas* (*состояние*: (позиция корабля, позиции врагов, позиции препятствий, шаги, выполненные до сего момента); неявный невзвешенный граф; алгоритм Дейкстры)
11. **UVa 11198 – Dancing Digits *** (*состояние*: перестановка; алгоритм поиска в ширину; особые приемы при кодировании)
12. **UVa 11329 – Curious Fleas *** (*состояние*: битовая маска из 26 битов, 4 для описания позиции выхода из игры на сетке (доске) 4×4, 16 для описания ячеек, в которых находятся блохи, 6 для описания ячеек, смежных с позициями выхода из игры, в которых находятся блохи; используется структура данных map; кодирование скучное и утомительное)
13. UVa 11513 – 9 Puzzle (поменять местами исходную и целевую позиции)
14. UVa 11974 – Switch The Lights (алгоритм поиска в ширину в неявном невзвешенном графе)
15. UVa 12135 – Switch Bulbs (LA 4201, Dhaka08, аналогично заданию UVa 11974)

- Метод встречи в середине / алгоритм поиска A* / алгоритм поиска IDA*

1. UVa 00652 – Eight (классическая головоломка на перемещение восьми фишек, алгоритм IDA*)
2. **UVa 01098 – Robots on Ice *** (LA 4793, World Finals Harbin10, см. описание в разделе 8.2.2; но для этой задачи существует более быстрое решение с применением метода встречи в середине)
3. UVa 01217 – Route Planning (LA 3681, Kaohsiung06, решается с применением алгоритма A*/IDA*; тестовые данные с большой вероятностью содержат не более 15 контрольных пунктов, в которые уже включены начальный и конечный пункты этого маршрута)
4. **UVa 10181 – 15-Puzzle Problem *** (аналогично заданию UVa 00652, но здесь размер поля больше, поэтому можно воспользоваться алгоритмом IDA*)

5. UVa 11163 – Jaguar King (еще одна головоломка, разрешимая с помощью алгоритма IDA*)
6. **UVa 11212 – Editing a Book *** (метод встречи в середине, см. раздел 8.2.4)

- Также см. задачи полного поиска в разделе 8.4.

8.3. Более эффективные методы динамического программирования

В разделах 3.5, 4.7.1, 5.4, 5.6 и 6.5 была приведена вводная информация о методах динамического программирования (dynamic programming – DP), рассматривались некоторые классические задачи динамического программирования и их решения, а также некоторые начальные сведения, облегчающие понимание неклассических (не относящихся к типовым) задач динамического программирования. Существует несколько более эффективных методов динамического программирования, которые не рассматривались в перечисленных выше разделах. Некоторые из этих методов мы рассмотрим здесь.

8.3.1. Динамическое программирование с использованием битовой маски

Некоторые современные задачи динамического программирования требуют (небольшого) набора нулевых значений как одного из параметров состояния динамического программирования. Это еще один случай, когда метод применения битовой маски может оказаться полезным (также см. раздел 8.2.1). Этот метод пригоден для динамического программирования, поскольку целое число (которое представляет битовую маску) можно использовать как индекс таблицы динамического программирования. Мы уже рассматривали такой подход при обсуждении решения задачи коммивояжера методом динамического программирования (см. раздел 3.5.2). Здесь мы рассмотрим еще один пример.

UVa 10911 – Forming Quiz Teams

Для изучения сокращенного описания условия задачи и кода ее решения обратитесь к самой первой задаче в главе 1. Претенциозное название этой задачи: «совершенное паросочетание минимального веса в небольшом произвольном взвешенном графе». В общем случае эта задача является трудноразрешимой. Но если исходный размер невелик, скажем не более $M \leqslant 20$, то можно воспользоваться решением методом динамического программирования с применением битовой маски.

Решение методом динамического программирования с применением битовой маски для этой задачи простое. Состояние паросочетаний представлено битовой маской `bitmask`. Это решение мы продемонстрируем на небольшом примере с $M = 6$. Начинаем с состояния, в котором пока еще не найдено

ни одного паросочетания, то есть `bitmask = 000000`. Если элементы 0 и 2 образуют пару, то можно установить (в единицу) два бита (бит 0 и бит 2) одновременно с помощью простой битовой операции: `bitmask | (1 << 0) | (1 << 2)`, таким образом, устанавливается состояние `bitmask=000101`. Отметим, что нумерация индекса начинается с 0 и отсчитывается справа. Если из этого состояния определяется, что следующую пару образуют элементы 1 и 5, то состояние становится таким: `bitmask=100111`. Совершенное паросочетание найдено, когда состояние определено всеми единицами, то есть в нашем примере: `bitmask=111111`.

Несмотря на то что существует много способов достижения некоторого определенного состояния, количество различных состояний равно $O(2^M)$. Для каждого состояния записывается минимальный вес для всех предыдущих паросочетаний, которые обязательно должны быть определены при достижении этого текущего состояния. Необходимо найти совершенное паросочетание. Сначала мы находим один сброшенный (равный нулю) бит i, используя один цикл $O(M)$. Затем мы находим «наилучший» другой сброшенный бит j из диапазона $[i + 1..M - 1]$, используя еще один цикл $O(M)$ и рекурсивно сопоставляя i и j. Эта проверка опять же выполняется с помощью битовой операции, то есть проверяется условие `if( !(bitmask & (1 << i)))` и такое же условие для j. Время выполнения данного алгоритма $O(M \times 2^M)$. В задании UVa 10911 $M = 2N$ и $2 \leqslant N \leqslant 8$, решение методом динамического программирования с использованием битовой маски является выполнимым и приемлемым. Более подробную информацию вы получите, внимательно изучив исходный код в следующем файле.

Файл исходного кода: *ch8_02_UVa10911.cpp/java*

В этом подразделе мы показали, что динамическое программирование с применением битовой маски может использоваться для решения малых вариантов ($M \leqslant 20$) задачи поиска паросочетаний в обобщенном графе. Вообще говоря, метод битовой маски позволяет представить небольшой набор элементов не более ≈ 20. Упражнения по программированию в этом разделе содержат больше примеров использования битовой маски как одного из параметров состояния задачи динамического программирования.

Упражнение 8.3.1.1. Продемонстрировать требуемое решение с использованием динамического программирования с применением битовой маски, если приходится иметь дело с задачей поиска «паросочетания максимальной мощности в небольшом произвольном графе ($V \leqslant 18$)».

Упражнение 8.3.1.2*. Переписать код из файла *ch8_02_UVa10911.cpp/java* с использованием приема `LSOne`, описанного в разделе 8.2.1, для ускорения решения.

8.3.2. Некоторые общие параметры (динамического программирования)

После решения множества задач динамического программирования (включая рекурсивный поиск с возвратами без мемоизации) у участников олимпиад

развивается особое чувство, благодаря которому они определяют, какие параметры наиболее часто выбираются для представления состояний задач динамического программирования (или задач рекурсивного поиска с возвратами). Некоторые из таких параметров описаны ниже.

1. Параметр: индекс i в массиве, например $[x_0, x_1, \ldots, x_i, \ldots]$.
   Изменение состояния: расширение подмассива $[0..i]$ или $[i..n-1]$, проход по всем значениям i, использование или исключение элемента i и т. д.
   Пример: одномерная (1D) максимальная сумма, LIS, часть 0–1 Knapsack (упаковка рюкзака), задача коммивояжера (TSP) и т. д. (см. раздел 3.5.2).
2. Параметр: индексы (i, j) в двух массивах, например $[x_0, x_1, \ldots, x_i] + [y_0, y_1, \ldots, y_i]$.
   Изменение состояния: расширение i, j или обоих индексов и т. д.
   Пример: выравнивание строк / регулирование расстояния, LCS и т. д. (см. раздел 6.5).
3. Параметр: подмассив (i, j) массива $[\ldots, x_i, x_{i+1}, \ldots, x_j, \ldots]$.
   Изменение состояния: разделение массива (i, j) на подмассивы $(i, k) + (k + 1, j)$ или $(i, i + k) + (i + k + 1, j)$ и т. д.
   Пример: последовательное умножение матриц (см. раздел 9.20) и т. д.
4. Параметр: вершина (позиция) в (обычно неявном) направленном ациклическом графе.
   Изменение состояния: обработка соседей (смежных вершин) этой вершины и т. д.
   Пример: кратчайший / самый длинный / вычисляемый путь в направленном ациклическом графе и т. д. (раздел 4.7.1).
5. Параметр: параметр задач типа упаковки рюкзака.
   Изменение состояния: уменьшение (или увеличение) текущего значения до тех пор, пока не будет достигнуто значение ноль (или заданное пороговое значение) и т. д.
   Пример: 0–1 Knapsack (задача упаковки рюкзака), сумма подмножества, варианты размена монет и т. д. (см. раздел 3.5.2).
   Примечание: этот параметр неудобен для динамического программирования, если его диапазон слишком велик.
   См. советы и рекомендации в разделе 8.3.3, если значение этого параметра может стать отрицательным.
6. Параметр: небольшой набор значений (обычно используется метод битовой маски).
   Изменение состояния: установка (в единицу) или сброс (в ноль) одного элемента (или нескольких элементов) в наборе и т. д.
   Пример: задача коммивояжера, решаемая методом динамического программирования (DP-TSP) (см. раздел 3.5.2), динамическое программирование с применением битовой маски (см. раздел 8.3.1) и т. д.

Отметим, что более трудные задачи динамического программирования обычно объединяют два и более параметра для представления различных состояний. Рекомендуем решить дополнительные задачи динамического программирования, приведенные в этом разделе, чтобы усовершенствовать свои навыки в этой теме.

8.3.3. Обработка отрицательных значений параметров с использованием метода смещения

В редких случаях возможный диапазон параметров, используемых для представления состояния динамического программирования, может становиться отрицательным. Это создает проблемы при решении задач динамического программирования, поскольку значение параметра отображается в индекс таблицы динамического программирования. Предполагается, что индексы таблицы динамического программирования непременно должны быть неотрицательными. К счастью, эту проблему можно легко устранить, используя метод смещения, чтобы снова сделать неотрицательными все индексы. Мы рассмотрим данный метод подробнее на примере решения еще одной нестандартной задачи динамического программирования: произвольной расстановки скобок.

UVa 1238 – Free Parentheses (ACM ICPC Jakarta08, LA 4143)

Краткое описание задачи: задано простое арифметическое выражение, состоящее только из операций сложения и вычитания, например: `1 - 2 + 3 - 4 - 5`. Вы можете произвольно расставлять любое количество скобок в любых позициях этого выражения, но при этом выражение должно оставаться корректным. Сколько различных чисел можно получить с помощью различных вариантов расстановки скобок? Для простого выражения, приведенного выше, ответ: 6 чисел.

```
1 - 2 + 3 - 4 - 5   =  -7       1 - (2 + 3 - 4 - 5)   =  5
1 - (2 + 3) - 4 - 5 = -13       1 - 2 + 3 - (4 - 5)   =  3
1 - (2 + 3 - 4) - 5 =  -5       1 - (2 + 3) - (4 - 5) = -3
```

В этой задаче определены следующие ограничения: выражение состоит только из $2 \leqslant N \leqslant 30$ неотрицательных чисел, меньших 100 и разделенных операторами сложения и вычитания. Перед первым и после последнего числа нет никаких операторов.

Для решения этой задачи необходимо сделать три очевидных замечания:

1) надо поместить открывающую скобку после знака «–» (оператор вычитания), чтобы изменить смысл последующих операторов «+» и «–»;
2) если уже размещено X открывающих скобок, то можно разместить только X закрывающих скобок – эту информацию необходимо хранить для правильной обработки подзадач;
3) максимальное значение выражения 100 + 100 + … + 100 (100 повторяется 30 раз) = 3000, а минимальное значение выражения 0 – 100 – 100 – … – 100 (единственный 0, за которым следует 29 операций вычитания 100) = –2900 – эту информацию также необходимо сохранить, как мы увидим ниже.

Для решения данной задачи с использованием динамического программирования необходимо определить, какой набор параметров задачи представляет различные состояния. Проще всего определить следующие два параметра динамического программирования:

1) `idx` – текущая обрабатываемая позиция – необходимо знать, где мы находимся;

2) open – количество открытых скобок, чтобы можно было сформировать корректное выражение[1].

Но этих двух параметров пока еще недостаточно для полной и однозначной идентификации состояния. Например, следующее частичное выражение «1 – 1 + 1 – 1…» имеет индекс idx=3 (индексы 0, 1, 2, 3 уже обработаны) и open=0 (невозможно/запрещено разместить где-либо закрывающую скобку), а в сумме дает результат 0. Далее, выражение «1 – (1 + 1 – 1)…» также имеет индекс idx=3, open=0, и его результатом тоже является 0. А вот выражение «1 – (1 + 1) – 1…» имеет индекс idx=3, open=0, но его результат равен –2. Поэтому описанные выше два параметра не способны однозначно идентифицировать состояние. Необходим еще один параметр, для того чтобы различать состояния выражений, а именно значение val. Умение определять правильный набор параметров для представления различных состояний необходимо развивать постоянно, чтобы успешно решать задачи динамического программирования. Код решения и его описание (в комментариях) приведены ниже.

```
void rec( int idx, int open, int val )
{
  if( visited[idx][open][val+3000] )        // это состояние было достигнуто в самом начале
    return;    // применен прием +3000 для преобразования отрицательных индексов в диапазон
[200..6000],   // так как отрицательные индексы неудобны для доступа к статическому массиву
  visited[idx][open][val+3000] = true;            // отметить это состояние как достигнутое

  if( idx == N )                   // последнее число, текущее значение - одно из возможных
    used[val+3000] = true, return;                                    // результат выражения

  int nval = val + num[idx] * sig[idx] * ((open % 2 == 0) ? 1 : -1);
  if( sig[idx] == -1 )      // вариант 1: размещение открывающей скобки, только если знак -
    rec( idx+1, open+1, nval );                                  // без эффекта, если знак +
  if( open > 0 )  // вариант 2: размещение закрывающей скобки, только это действие возможно
    rec( idx+1, open-1, nval );            // если уже имеется несколько открывающих скобок
  rec( idx+1, open nval ); // вариант 3: обычное состояние, не выполняются никакие действия
}

// Предварительная обработка: создание массива логических значений used, в котором для всех
// элементов изначально устанавливаются значения false, затем выполнение показанного выше
// метода "сверху вниз" динамического программирования с помощью вызова rec( 0, 0, 0 ).
// Решением является количество значений в массиве used, которые имеют значения true.
```

В приведенном выше коде мы видим, что все возможные состояния этой задачи можно представить с помощью трехмерного массива bool visited[idx][open][val]. Цель такой таблицы visited – установка пометки, если определенное состояние было посещено (достигнуто). Так как значения val имеют диапазон от –2900 до 3000 (5901 различное значение), необходимо сместить этот диапазон, чтобы сделать его полностью неотрицательным. В рассмотренном примере для этого использовалось явно определенное постоянное значение +3000. Количество состояний $30 \times 30 \times 6001 \approx 5$ млн со временем обработки $O(1)$ на каждое состояние. Это достаточно быстрое решение.

---

[1] В позиции idx = N (обработано самое последнее число) ситуация остается корректной, если open > 0, так как можно разместить все необходимые закрывающие скобки в конце выражения, например 1 – (2 + 3 – (4 – (5))).

8.3.4. Превышение лимита памяти? Рассмотрим использование сбалансированного бинарного дерева поиска как таблицы запоминания состояний

В разделе 3.5.2 рассматривалась задача динамического программирования: задача об упаковке рюкзака 0–1 (0–1 Knapsack), где состояние определяется как (id, remW). Параметр id имеет диапазон [0..n – 1], а параметр remW имеет диапазон [0..S]. Если автор задачи устанавливает достаточно большое значение $n \times S$, то двумерный массив (для таблицы состояний динамического программирования) с размером $n \times S$ становится чрезмерно большим (что приводит к превышению лимита памяти на олимпиадах по программированию).

К счастью, для задач, подобных задаче об упаковке рюкзака 0–1 с применением метода нисходящего ДП (см. главу 3), нам доподлинно известно, что посещаются не все состояния (тогда как версия «спуска сверху вниз» динамического программирования должна исследовать все состояния). Таким образом, можно сократить время выполнения в уменьшенном пространстве, воспользовавшись сбалансированным бинарным деревом поиска (BST) (C++ STL `map` или Java `TreeMap`) как таблицей запоминания состояний. Это сбалансированное бинарное дерево поиска будет запоминать только те состояния, которые действительно посещаются при методе «спуска сверху вниз» динамического программирования. Следовательно, если имеется только k посещенных состояний, то мы будем использовать лишь пространство $O(k)$ вместо пространства размером $n \times S$. Время выполнения методом «спуска сверху вниз» динамического программирования увеличивается с коэффициентом $O(c \times \log k)$. Но следует отметить, что этот прием редко оказывается полезным из-за слишком большого значения постоянного множителя c в выражении оценки временной сложности.

8.3.5. Превышение лимита памяти/времени? Используйте более эффективное представление состояния

Предложенное «корректное» решение методом динамического программирования (которое дает правильный ответ, но использует больше вычислительных ресурсов) может привести к превышению лимита памяти (MLE) или к превышению лимита времени (TLE), если автор задачи использовал более эффективное представление состояния и установил более жесткие входные ограничения, при которых наше «корректное» решение методом динамического программирования становится неприемлемым. В этом случае нет альтернативы поиску более эффективного представления состояний динамического программирования для уменьшения размера таблицы запоминания состояний DP (и соответствующего ускорения решения, то есть снижения общей временной сложности). Мы подробно рассмотрим применение данного метода на следующем примере.

UVa 1231 – ACORN (ACM ICPC Singapore07, LA 4106)

Краткое описание задачи: дано t дубов (деревьев), высота всех дубов h, высота f, которую белка Jayjay теряет при прыжке с одного дерева на другое, $1 \leqslant t, h \leqslant 2000$, $1 \leqslant f \leqslant 500$, а позиции расположения желудей на каждом дереве: acorn[tree]

[height]; требуется определить максимальное количество желудей, которые Jayjay может собрать за один нисходящий проход по деревьям. Пример: если $t = 3$, $h = 10$, $f = 2$ и acorn[tree][height], как показано на рис. 8.11, то наилучший нисходящий путь позволит собрать 8 желудей (путь обозначен пунктирной линией).

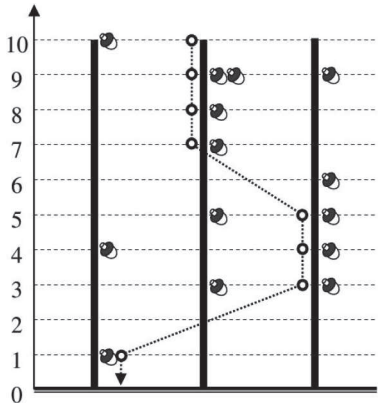


Рис. 8.11. Нисходящий путь

Простейшее решение методом динамического программирования: использовать таблицу `total[tree][height]`, в которой хранится наибольшее возможное количество желудей, собранное белкой Jayjay, когда она находится на конкретном дереве на конкретной высоте. Затем Jayjay рекурсивно пробует либо спуститься на одну единицу высоты (–1) на том же дубе, либо перепрыгнуть с потерей (–*f*) единиц высоты на $t - 1$ другой дуб со своей позиции. В максимальном тестовом варианте требуется 2000×2000 = 4 млн состояний и 4 млн × 2000 = 8 млрд операций. При таком подходе невозможно избежать превышения лимита времени (TLE).

Улучшенное решение методом динамического программирования: в действительности можно пренебречь информацией о том, «на каком дереве белка Jayjay находится в текущий момент», так как вполне достаточно запомнить наилучшее состояние (положение). Причина в том, что, перепрыгивая на любой другой $t - 1$ дуб, Jayjay неизменно уменьшает свою высоту. Создадим таблицу `dp[height]`, в которой хранится наибольшее возможное количество желудей, собранных белкой на конкретной высоте `height`. Исходный код метода «подъема снизу вверх» динамического программирования, в котором требуется обработка всего лишь 2000 состояний, а временная сложность равна 2000 × 2000 = 4 млн (операций), приведен ниже.

```
for( int tree=0; tree < t; tree++ )                                    // инициализация
  dp[h] = max( dp[h], acorn[tree][h] );

for( int height=h-1; height >= 0; height-- )
  for( int tree=0; tree < t; tree++ ) {
    acorn[tree][height] += max( acorn[tree][height+1],                    // от этого дерева
                                                                        // на уровень +1 выше
```

```
            ((height + f <= h) ? dp[height+f] : 0));     // от дерева на высоте height + f
    dp[height] = max( dp[height], acorn[tree][height] );         // обновить и это значение
  }

printf( "%d\n", dp[0] );                                  // решение (ответ) сохранено здесь
```

Файл исходного кода: *ch8_03_UVa1231.cpp/java*

Когда размер структуры данных о состояниях задачи динамического программирования слишком велик и приводит к общей временной сложности DP, которая является неосуществимой, следует подумать о другом, более эффективном (но обычно не очевидном) способе представления возможных состояний. Использование правильного представления состояний – это потенциальное существенное ускорение решения по методу динамического программирования. Напомним, что на олимпиадах по программированию нет неразрешимых задач, поэтому автор задачи обязательно должен подразумевать какой-то трюк.

8.3.6. Превышение лимита памяти/времени? Отбросим один параметр, будем восстанавливать его по другим параметрам

Другим известным приемом сокращения используемого объема памяти для решения задач динамического программирования (следовательно, и для ускорения решения) является отказ от одного важного параметра, который можно восстановить, используя другие параметры. Чтобы продемонстрировать применение этого метода на практике, рассмотрим задачу из финала чемпионата мира ACM ICPC World Finals.

UVa 1099 – Sharing Chocolate (ACM ICPC World Finals Harbin10, LA 4794)

Краткое описание задачи: имеется большая плитка шоколада размером $1 \leqslant w, h \leqslant 100$ для $1 \leqslant n \leqslant 15$ друзей и запрос размера от каждого друга. Можно ли разделить эту плитку шоколада, используя только горизонтальные и вертикальные разрезы (разломы) так, чтобы каждый друг получил одну часть плитки шоколада выбранного им размера?

Например, см. рис. 8.12 (изображение слева). Размер исходной плитки шоколада: ширина $w = 4$ и высота $h = 3$. Если шоколад нужно разделить между четырьмя друзьями, каждый из которых попросил часть плитки размером {6, 3, 2, 1} соответственно, то можно разломить шоколад на четыре части, используя три разреза (разлома), как показано на рис. 8.12 (изображение справа).

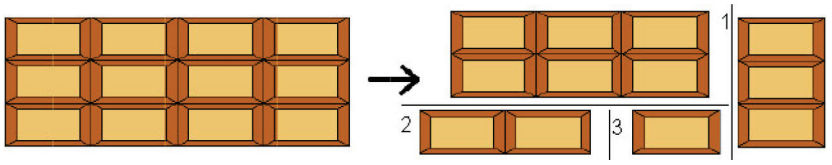


Рис. 8.12 ❖ Графическое представление решения задачи ACM ICPC WF2010 – J – Sharing Chocolate

Те участники олимпиады, которые уже знакомы с методами динамического программирования, должны с легкостью сделать следующие выводы: во-первых, если сумма всех запросов не равна $w \times h$, то задача не имеет решения. В противном случае можно представить различные состояния этой задачи с помощью трех параметров (w, h, bitmask), где w и h – текущие размеры плитки шоколада в рассматриваемый момент времени, bitmask – подмножество друзей, которые уже получили свою порцию шоколада выбранного размера. Но экспресс-анализ показывает, что для этого требуется таблица динамического программирования размером $100 \times 100 \times 2^{15} = 327$ млн элементов. В условиях олимпиады по программированию это слишком много.

Более эффективным представлением состояния является использование только двух параметров: либо (w, bitmask), либо (h, bitmask). Не теряя общей сущности, принимаем набор параметров (w, bitmask). По такой формулировке можно «восстановить» требуемое значение h по формуле sum(bitmask)/w, где sum(bitmask) – сумма размеров частей плитки шоколада, запрошенных и уже полученных друзьями, в битовой маске bitmask (то есть все установленные биты в битовой маске bitmask). Таким образом, мы получаем все требуемые параметры: w, h и bitmask, но при этом размер таблицы динамического программирования сокращается до $100 \times 2^{15} = 3$ млн элементов. Это приемлемый размер.

Простейшие варианты: если битовая маска bitmask содержит только 1 установленный бит, а запрашиваемый размер части плитки шоколада этим другом равен $w \times h$, то это и есть решение. В противном случае решение не найдено.

Обобщенные варианты: если имеется часть плитки шоколада размером $w \times h$, а текущий набор друзей, уже получивших свою долю, определяется битовой маской bitmask = $\text{bitmask}_1 \cup \text{bitmask}_2$, то можно выполнить горизонтальный или вертикальный разрез (разлом) так, чтобы одна часть предназначалась для друзей в битовой маске bitmask_1, а другая часть – для друзей в битовой маске bitmask_2.

В самом худшем варианте временная сложность этой задачи остается огромной, но при правильном отсечении предложенное здесь решение укладывается в ограничения по времени.

Упражнение 8.3.6.1*. Решить задания UVa 10482 – The Candyman Can и UVa 10626 – Buying Coke, используя описанный здесь метод. Определить, какой параметр можно исключить с возможностью его восстановления по другим параметрам для достижения максимальной эффективности.

Кроме нескольких задач динамического программирования в разделе 8.4, в главе 9 предлагается еще ряд задач динамического программирования, которые не упоминаются в текущей главе, поскольку на олимпиадах они встречаются редко. Ниже приведен список таких задач.

1. Раздел 9.3: битоническая задача коммивояжера (еще раз обращаем внимание на метод «исключения одного параметра с возможностью его восстановления по другим параметрам»).
2. Раздел 9.5: задача китайского почтальона (еще один вариант использования динамического программирования с применением битовой мас-

ки для нахождения совершенного паросочетания минимального веса в небольшом произвольном взвешенном графе).
3. Раздел 9.20: задача о порядке умножения матриц (классическая задача динамического программирования).
4. Раздел 9.21: возведение матрицы в степень (можно ускорить преобразования динамического программирования для некоторых редко встречающихся задач DP от $O(n)$ до $O(\log n)$, переписав рекуррентные выражения динамического программирования как операции умножения матриц).
5. Раздел 9.22: задача о независимом множестве максимального веса (в дереве) может быть решена методом динамического программирования.
6. Раздел 9.33: разреженная таблица – используется динамическое программирование.

Задания по программированию, связанные с использованием более усовершенствованного метода динамического программирования

- Динамическое программирование уровня 2 (немного сложнее, чем задания, описанные в главах 3, 4, 5, 6)
  1. **UVa 01172 – The Bridges of… *** (LA 3986, неклассическая задача динамического программирования с минимальным применением поиска паросочетаний, но с ограничениями направления поиска слева направо и типа ОС)
  2. **UVa 01211 – Atomic Car Race *** (LA 3404, Tokyo05, предварительно вычисляемый массив T[L] – время прохождения пути длиной L; динамическое программирование с одним параметром i – контрольный пункт смены покрышек; если $i = n$, то покрышки не меняются)
  3. UVa 10069 – Distinct Subsequences (используется структура данных Java `BigInteger`)
  4. UVa 10081 – Tight Words (используются дублирующиеся элементы)
  5. UVa 10364 – Square (можно воспользоваться методом битовой маски)
  6. *UVa 10419 – Sum-up the Primes* (вывод пути, простые числа)
  7. *UVa 10536 – Game of Euler* (моделируется доска 4×4 и 48 возможных фишек как битовая маска; это простая игра для двух игроков; также см. раздел 5.8)
  8. UVa 10651 – Pebble Solitaire (задача малого размера; разрешима с помощью поиска с возвратами)
  9. *UVa 10690 – Expression Again* (сумма подмножества, динамическое программирование, применяется метод отрицательного смещения, немного простой математики)
  10. UVa 10898 – Combo Deal (похоже на динамическое программирование + битовая маска; хранение состояний как целых чисел)
  11. **UVa 10911 – Forming Quiz Teams *** (подробно рассматривается в этом разделе)
  12. *UVa 11088 – End up with More Teams* (похоже на задание UVa 10911, но здесь речь идет о подборе сочетаний трех игроков в одной команде)

13. UVa 11832 – Account Book (интересная задача динамического программирования; *s*: (id, val); использование смещения для обработки отрицательных чисел; *t*: плюс или минус; вывод решения)
14. UVa 11218 – KTV (остается разрешимой методом полного поиска)
15. *UVa 12324 – Philip J. Fry Problem* (необходимо обязательно сделать замечание: сфера > *n* бесполезна)

- Динамическое программирование уровня 3

1. UVa 00607 – Scheduling Lectures (возвращается пара информационных элементов)
2. *UVa 00702 – The Vindictive Coach* (неявный направленный ациклический граф – нетривиальное решение)
3. *UVa 00812 – Trade on Verweggistan* (объединение жадного алгоритма и динамического программирования)
4. UVa 00882 – The Mailbox… (s: (lo, hi, mailbox_left); попытки перебора всех вариантов)
5. **UVa 01231 – ACORN** * (LA 4106, Singapore07, динамическое программирование с сокращением размерности, решение рассматривалось в этом разделе)
6. **UVa 01238 – Free Parentheses** * (LA 4143, Jakarta08, автор задачи: Феликс Халим (Felix Halim), решение рассматривалось в этом разделе)
7. UVa 01240 – ICPC Team Strategy (LA 4146, Jakarta08)
8. UVa 01244 – Palindromic paths (LA 4336, Amritapuri08, сохранение наилучшего пути (последовательности) между *i*, *j*; таблица динамического программирования содержит строки)
9. *UVa 10029 – Edit Step Ladders* (использовать структуру `map` как таблицу запоминания состояний)
10. *UVa 10032 – Tug of War* (задача динамического программирования об упаковке рюкзака с оптимизацией, чтобы избежать превышения лимита времени)
11. *UVa 10154 – Weights and Measures* (вариант задачи о поиске наибольшей возрастающей подпоследовательности (LIS))
12. UVa 10163 – Storage Keepers (попытка перебора всех возможных безопасных линий *L* и применение динамического программирования; *s*: id, N_left; *t*: нанять/пропустить кандидата id для охраны склада *K*)
13. UVa 10164 – Number Game (немного теории чисел (вычисления по модулю), поиск с возвратами; мемоизация состояния динамического программирования: (sum, taken))
14. UVa 10271 – Chopsticks (важное замечание: 3-й палочкой для еды может быть любая палочка, необходимо без промедления («жадно») выбирать ближайшую смежную палочку; состояние динамического программирования: (pos, k_left), переход (смена): игнорировать эту палочку или взять эту палочку и палочку, непосредственно следующую за ней, затем переместиться в позицию pos+2; отсечь недостижимые состояния, в которых не остается достаточного количества палочек, чтобы собрать из них тройки)

15. UVa 10304 – Optimal Binary... (классическая задача динамического программирования, требует суммирования одномерного диапазона и применения алгоритма Кнута–Яо для ускорения решения до $O(n^2)$)
16. *UVa 10604 – Chemical Reaction* (возможно смешивание любой пары химических веществ до тех пор, пока не останется только два химиката; запоминание оставшихся химикатов с помощью структуры `map`; сортировка остающихся химических веществ помогает увеличить число совпадений с таблицей запоминания)
17. *UVa 10645 – Menu* (s: (days_left, budget_left, prev_dish, prev_dish_count); первые два параметра аналогичны параметрам задачи об упаковке рюкзака; остальные два параметра используются для определения цены конкретного блюда, так как первое, второе и последующие использования этого блюда характеризуются различными значениями (стоимости))
18. UVa 10817 – Headmaster's Headache (*s*: (id, bitmask); пространство: $100 \times 2^{2*8}$)
19. *UVa 11002 – Towards Zero* (простая задача динамического программирования; используется метод отрицательного смещения)
20. *UVa 11084 – Anagram Division* (использование `next_permutation` / алгоритма грубой силы, вероятно, не лучший вариант решения; лучше перейти к формулировке задачи динамического программирования)
21. UVa 11285 – Exchange Rates (ежедневное отслеживание наиболее выгодных обменных курсов канадского и американского долларов (CAD & USD))
22. **UVa 11391 – Blobs in the Board *** (динамическое программирование с применением битовой маски на двумерной сетке)
23. *UVa 12030 – Help the Winners* (*s*: (idx, bitmask, all1, has2); *t*: попытка перебора всех туфель, которые не подходят девушке, выбравшей платье idx)

- Динамическое программирование уровня 4
  1. UVa 00473 – Raucous Rockers (исходные ограничения неочевидны; следовательно, необходимо использовать структуру `vector` переменного размера и компактные состояния)
  2. **UVa 01099 – Sharing Chocolate *** (LA 4794, World Finals Harbin10, подробно рассматривается в этом разделе)
  3. **UVa 01220 – Party at Hali-Bula *** (LA 3794, Tehran06; задача поиска максимального независимого множества (MIS) в дереве; динамическое программирование; кроме того, необходимо проверить, является ли найденное максимальное независимое множество единственным)
  4. UVa 01222 – Bribing FIPA (LA 3797, Tehran06, динамическое программирование на дереве)
  5. **UVa 01252 – Twenty Questions *** (LA 4643, Tokyo09, динамическое программирование, *s*: (bitmask1, bitmask2), где bitmask1 описывает

признаки, о которых мы решаем задать вопрос, а bitmask2 описывает ответы на вопросы о признаках, которые мы задаем)

6. *UVa 10149 – Yahtzee* (динамическое программирование с применением битовой маски; использование карточных правил; утомительно скучная задача)
7. UVa 10482 – The Candyman Can (см. **упражнение 8.3.6.1***)
8. UVa 10626 – Buying Coke (см. **упражнение 8.3.6.1***)
9. *UVa 10722 – Super Lucky Numbers* (необходима структура данных Java `BigInteger`; формулировка состояний динамического программирования непременно должна быть эффективной, чтобы избежать превышения лимита времени (TLE); состояние: (N_digits_left, B, first, previous_digit_is_one), также используется простая комбинаторика для получения ответа)
10. *UVa 11125 – Arrange Some Marbles* (вычисляемые пути в неявном направленном ациклическом графе; динамическое программирование в пространстве с восемью измерениями)
11. *UVa 11133 – Eigensequence* (неявный направленный ациклический граф; нетривиальная задача)
12. *UVa 11432 – Busy Programmer* (неявный направленный ациклический граф; нетривиальная задача)
13. UVa 11472 – Beautiful Numbers (динамическое программирование, состояние с четырьмя параметрами)

- Дополнительно см. некоторые задачи динамического программирования в разделе 8.4 и в главе 9.

8.4. Декомпозиция задачи

Несмотря на то что в задачах на олимпиадах по программированию участники обычно пробуют применить лишь несколько основных структур данных и алгоритмов (многие из этих структур и алгоритмов подробно рассматривались в данной книге), для решения более трудных задач может потребоваться сочетание двух (и более) алгоритмов и/или структур данных. Для решения таких задач сначала необходимо выполнить декомпозицию задачи, то есть разделение ее на компоненты, чтобы можно было решить каждый компонент независимо. Чтобы получить возможность действовать таким способом, необходимо хорошо знать отдельные компоненты (содержимое глав 1–7 и текущей главы до раздела 8.3 включительно).

Несмотря на существование ${}_NC_2$ возможных комбинаций двух из N алгоритмов и/или структур данных, не все эти комбинации имеют смысл. В этом разделе мы собрали и описали некоторые[1] наиболее часто применяемые сочетания двух алгоритмов и/или структур данных на основе нашего практического опыта, полученного при решении приблизительно 1675 заданий UVa в ка-

[1] Этот список не является полным, и вряд ли когда-либо удастся составить полный список всех возможных комбинаций.

честве оценивающих экспертов. Завершается этот раздел описанием редкой комбинации из трех алгоритмов и/или структур данных.

8.4.1. Два компонента: бинарный поиск ответа и прочие

В разделе 3.3.1 рассматривался бинарный поиск ответа в (простой) имитационной задаче, которая не зависела от более изощренных алгоритмов, описанных после раздела 3.3.1. В действительности этот метод можно объединить с некоторыми другими алгоритмами из разделов 3.4–8.3. Ниже перечислены некоторые варианты, с которыми мы встречались, – алгоритмы, дополняющие бинарный поиск ответа:

- жадный алгоритм (см. раздел 3.4), например в заданиях UVa 714, 11526;
- проверка связности графа (см. раздел 4.2), например в заданиях UVa 295, 10876;
- алгоритм поиска кратчайшего пути из одной вершины графа во все остальные (SSSP) (см. раздел 4.4), например в заданиях UVa 10816, IOI 2009 (Mecho);
- алгоритм поиска максимального потока (см. раздел 4.6), например в задании UVa 10983;
- алгоритм поиска максимальных паросочетаний максимальной мощности в двудольном графе (MCBM) (см. раздел 4.7.4), например в заданиях UVa 1221, 10804, 11262;
- операции со структурой данных BigInteger (см. раздел 5.3), например в задании UVa 10606;
- геометрические формулы (см. раздел 7.2), например в заданиях UVa 1280, 10566, 10668, 11646.

В этом разделе рассматриваются два написанных нами примера применения метода бинарного поиска ответа. Сочетание бинарного поиска ответа с другим алгоритмом может быть определено при ответе на следующий вопрос: «Если мы узнали требуемый ответ (методом бинарного поиска) и предполагаем, что этот ответ правильный (истинный – true), то является ли исходная задача разрешимой или неразрешимой (вопрос типа true/false)?».

Бинарный поиск ответа в сочетании с жадным алгоритмом

Краткое описание задачи UVa 714 – Copying Books: имеется $m \leqslant 500$ книг, пронумерованных по порядку 1, 2, ..., m, возможно, с различным количеством страниц ($p_1, p_2, ..., p_m$). Необходимо сделать по одной копии каждой книги. Ваша задача – распределить эти книги между k переписчиками, $k \leqslant m$. Каждая книга может быть передана (присвоена) только одному переписчику, при этом каждый переписчик обязательно должен получить непрерывную последовательность книг (по номерам). Таким образом, существует возрастающая последовательность чисел $0 = b_0 < b_1 < b_2 < ... < b_{k-1} \leqslant b_k = m$, такая, что i-й переписчик ($i > 0$) получает последовательность книг с номерами между $b_{i-1} + 1$ и b_i. Каждый переписчик копирует страницы с одинаковой скоростью. Следовательно, время, необходимое для создания одной копии каждой книги, определяется по переписчику, которому назначена наибольшая часть работы. Теперь вам нуж-

но определить минимальное количество страниц, копируемое переписчиком с максимальным объемом работы.

Существует решение этой задачи методом динамического программирования, но ее также можно решить, вычисляя ответ методом бинарного поиска. Мы рассмотрим это решение на примере с исходными данными $m = 9$, $k = 3$ и $p_1, p_2, \ldots, p_9$, равными 100, 200, 300, 400, 500, 600, 700, 800, 900 соответственно.

Если предположить, что ответ = 1000, то задача «упрощается», то есть: если переписчик с максимальным объемом работы может скопировать только (не более) 1000 страниц, то можно ли решить эту задачу? Ответ – нет. Можно по принципу жадного алгоритма распределить работу от книги 1 до книги m следующим образом: {100, 200, 300, 400} для переписчика 1, {500} для переписчика 2, {600} для переписчика 3. Но при таком подходе три книги {700, 800, 900} остаются нераспределенными. Следовательно, ответ должен быть > 1000.

Если предположить, что ответ = 2000, то в соответствии с жадным алгоритмом можно распределить работу так: {100, 200, 300, 400} для переписчика 1, {600, 700} для переписчика 2, {800, 900} для переписчика 3. Все книги копируются, и даже остаются некоторые резервы времени, то есть для переписчиков 1, 2 и 3 потенциально незадействованными являются книги {500, 700, 300}. Из этого следует, что ответ должен быть ⩽ 2000.

Правильный ответ можно определить методом бинарного поиска в диапазоне [lo..hi], где lo = $\max(p_i)$, $\forall i \in [1..m]$ (количество страниц в самой толстой книге) и hi = $p_1 + p_2 + \ldots + p_m$ (сумма всех страниц во всех книгах). Для самых любознательных читателей приводим оптимальный ответ для тестового варианта рассматриваемого здесь примера: 1700. Временная сложность этого решения равна $O(m \times \log \text{hi})$. Отметим, что дополнительный множитель log hi обычно является незначительным в условиях олимпиады по программированию.

Бинарный поиск ответа в сочетании с геометрическими формулами

Мы рассмотрим задание UVa 11646 – Athletics Track для очередной демонстрации применения метода бинарного поиска ответа. Краткое описание задачи: рассмотрим прямоугольное футбольное поле с окружающей его легкоатлетической дорожкой, как показано на рис. 8.13 (слева), где две дуги с обеих (коротких) сторон поля arc1 и arc2 принадлежат одной окружности с центром в центральной точке футбольного поля. Длина легкоатлетической дорожки ($L1$ + arc1 + $L2$ + arc2) должна быть равна в точности 400 м. Если принять отношение длины L и ширины W футбольного поля равным a:b, то какой должна быть в действительности длина L и ширина W этого футбольного поля при соблюдении описанных выше ограничивающих условий?

Достаточно трудно (но возможно) получить решение с помощью карандаша и бумаги (аналитическое решение), но при использовании компьютера и метода бинарного поиска ответа (в действительности применяется метод бисекции) найти решение гораздо проще.

Начинаем с бинарного поиска значения L. По значению L можно определить $W = b/a \times L$. Предполагаемая длина дуги равна $(400 - 2 \times L)/2$. Теперь можно воспользоваться тригонометрическими формулами для вычисления радиуса r и угла o с помощью треугольника CMX (см. рис. 8.13, справа). CM = $0.5 \times L$ и MX = $0.5 \times W$. Получив r и o, можно вычислить реальную длину дуги arc. Затем

вычисленное значение длины дуги сравнивается с предполагаемой длиной дуги, чтобы решить, необходимо увеличить или уменьшить длину футбольного поля *L*. Соответствующий фрагмент исходного кода показан ниже.

```
lo = 0.0; hi = 400.0;                          // это возможный диапазон значений ответа
while( fabs( lo - hi ) > 1e-9 ) {
  L = (lo + hi) / 2.0;                       // применение метода бисекции для вычисления L
  W = b / a * L;                             // W можно вычислить по L и по отношению a : b
  expected_arc = (400 - 2.0 * L) / 2.0;                        // предполагаемое значение

  CM = 0.5 * L; MX = 0.5 * W;                  // применение тригонометрических формул
  r = sqrt( CM * CM + MX * MX);
  angle = 2.0 * atan( MX / CM ) * 180.0 / PI;              // угол дуги arc = 2x угла o
  this_arc = angle / 360.0 * PI * (2.0 * r);                  // вычисление длины дуги

  if( this_arc > expected_arc ) hi = L; else lo = L;       // уменьшение или увеличение L
}
printf( "Case %d: %.12lf %.12lf\n", caseNo++, L, W );
```

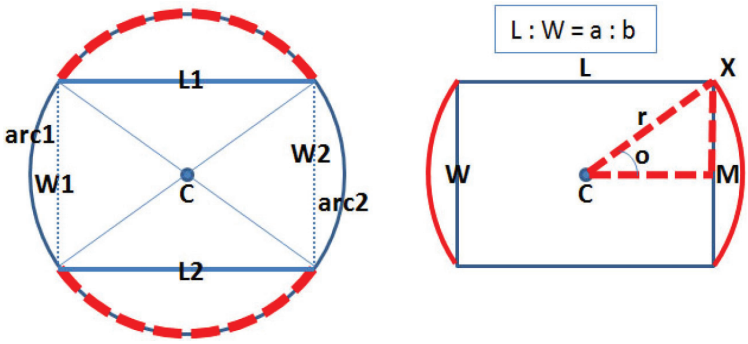


Рис. 8.13 ❖ Athletics Track (задание UVa 11646)

Упражнение 8.4.1.1*. Доказать, что другие стратегии не будут более эффективными, чем стратегия с применением жадного алгоритма, описанная выше при решении задания UVa 714.

Упражнение 8.4.1.2*. Вывести аналитическое решение для задания UVa 11646 вместо используемого в примере метода бинарного поиска ответа.

8.4.2. Два компонента: использование статической задачи RSQ/RMQ

Это сочетание относительно легко определяется. В задаче предполагается использование другого алгоритма для заполнения содержимого статического одномерного массива (который после заполнения никогда не будет изменяться), затем будет выполняться много запросов суммы/минимального и максимального значений (RSQ/RMQ) из диапазона этого статического одномерного

массива. Большинство этих запросов RSQ/RMQ выполняется на завершающем этапе вывода результатов решения задачи. Но иногда такие запросы RSQ/RMQ используются для ускорения работы внутреннего механизма другого алгоритма, используемого для решения той же задачи.

Решение задачи запроса суммы одномерного статического диапазона с применением динамического программирования рассматривалось в разделе 3.5.2. Для решения задачи запроса минимального/максимального значения одномерного статического диапазона использовалась разреженная таблица (это решение методом динамического программирования). Этот подход рассматривается в разделе 9.33. Без такого ускорения с помощью запросов RSQ/RMQ другой алгоритм, необходимый для решения задачи, обычно приводит к превышению лимита времени (TLE).

В качестве простого примера рассмотрим задачу нахождения количества простых чисел в различных диапазонах запроса $[a..b]$ ($2 \leqslant a \leqslant b \leqslant 1\ 000\ 000$). В эту задачу очевидно включена процедура генерации простых чисел (например, алгоритм решета Эратосфена (Sieve), см. раздел 5.5.1). Но поскольку в нашей задаче определен диапазон $2 \leqslant a \leqslant b \leqslant 1\ 000\ 000$, если время ответа на каждый запрос остается равным $O(b - a + 1)$, то это приведет к превышению лимита времени (TLE) при итеративном проходе от a до b, особенно если автор задачи предварительно определил значение $b - a + 1$ приблизительно равным 1 000 000 для (почти) каждого запроса. Необходимо ускорить этап вывода решения до $O(1)$ в каждом запросе, используя для этого запросы суммы одномерного статического диапазона с применением динамического программирования.

8.4.3. Два компонента: предварительная обработка графа и динамическое программирование

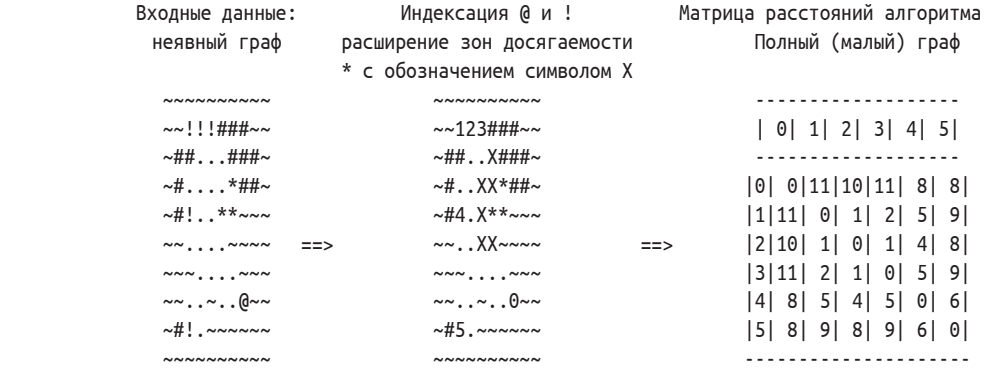
В этом подразделе подробно описана задача, одним из компонентов которой является предварительная обработка графа, так как в задаче явно подразумевается использование графов, а другим компонентом является метод динамического программирования. Мы рассмотрим это сочетание на двух примерах.

Алгоритм поиска кратчайшего пути из одной вершины графа во все остальные (SSSP) / алгоритм поиска кратчайшего расстояния между всеми вершинами (APSP) в сочетании с методом динамического программирования для задачи о коммивояжере

Задание UVa 10937 – Blackbeard the Pirate в этом разделе используется для демонстрации применения сочетания алгоритма поиска кратчайшего пути из одной вершины графа во все остальные (SSSP) / алгоритма поиска кратчайшего расстояния между всеми вершинами (APSP) в сочетании с методом динамического программирования для решения задачи о коммивояжере (DP TSP). Алгоритмы SSSP/APSP обычно применяются для преобразования входных данных (как правило, представленных в форме неявного графа или сетки/решетки) в другой граф (обычно меньшего размера). Затем реализуется метод

динамического программирования для задачи коммивояжера на втором (преобразованном; меньшем) графе.

Заданные входные данные для этой задачи показаны ниже на диаграмме слева. Это «карта» воображаемого острова. Пират Черная Борода высадился на этот остров в месте, обозначенном символом «@». Он спрятал не более 10 сокровищ на этом острове. Сокровища обозначены восклицательным знаком «!». Злобные туземцы обозначены звездочками *. Черная Борода должен держаться на расстоянии не менее одной клетки от злобных туземцев по любому из восьми направлений. Черная Борода должен собрать все свои сокровища и благополучно вернуться на корабль. Ему разрешены переходы только на клетки, обозначенные как «земля» «.», но не на клетки, обозначающие воду «~» или природные препятствия «#».



Понятно, что это задача коммивояжера (TSP) (см. раздел 3.5.3), но прежде чем мы сможем воспользоваться ее решением с применением метода динамического программирования (DP TSP), необходимо сначала преобразовать входные данные в матрицу расстояний.

В этой задаче нас интересуют только позиции символов «@» и «!». Индекс 0 присваивается символу «@», а последующие положительные индексы присваиваются всем символам «!». Увеличим область досягаемости каждого злобного туземца «*», заменив символы земли «.» в окружающих смежных клетках на символы «X». Затем выполняем поиск в ширину в этом невзвешенном неявном графе, начиная с символа «@», всех символов «!», переходя только на клетки, помеченные символом «.» (земля). Это дает нам матрицу расстояний алгоритма поиска кратчайших расстояний между всеми вершинами (APSP), как показано на вышеприведенной диаграмме (справа).

Далее после создания матрицы расстояний алгоритма APSP можно выполнить метод динамического программирования DP TSP, как описано в разделе 3.5.3, чтобы получить ответ (решение). В тестовом варианте, показанном выше, оптимальный путь коммивояжера (пирата Черная Борода): 0-5-4-1-2-3-0 со стоимостью 8 + 6 + 5 + 1 + 1 + 11 = 32.

Стягивание компонентов сильной связности в сочетании с алгоритмом динамического программирования в направленном ациклическом графе

В некоторых современных задачах с использованием направленного графа приходится иметь дело с компонентами сильной связности (Strongly Connected Component – SCC) в направленном (ориентированном) графе (см. раздел 4.2.9). Один из новейших вариантов – задача, в которой требуется выполнение предварительного стягивания всех компонентов сильной связности в заданном направленном графе для образования укрупненных вершин (которые мы называем супервершинами). Исходный направленный граф не обязательно является ациклическим, поэтому невозможно сразу же применить методы динамического программирования к такому графу. Но после стягивания компонентов сильной связности направленного графа получившийся граф с супервершинами представляет собой направленный ациклический граф (DAG) (пример такого графа см. на рис. 4.9). Напомним, что из описания в разделе 4.7.1 нам известно, что направленный ациклический граф очень хорошо подходит для применения методов динамического программирования, так как граф является ациклическим.

Одна из таких задач представлена в задании UVa 11324 – The Largest Clique. Если описать эту задачу кратко, то ее цель – поиск самого длинного пути в направленном ациклическом графе, полученном в результате стягивания компонентов сильной связности. Каждая супервершина имеет вес, представляющий количество исходных вершин, которые были стянуты в одну супервершину.

8.4.4. Два компонента: использование графов

Этот тип объединения задач можно определить следующим образом: одним явным компонентом является алгоритм на графе. Но при этом необходим еще один вспомогательный алгоритм, который обычно представляет некоторую реализацию математического или геометрического правила (для формирования обрабатываемого графа) или даже другой вспомогательный алгоритм на графе. В этом разделе рассматривается такой пример.

В разделе 2.4.1 было отмечено, что в некоторых задачах формируемый граф не обязательно сохранять в какой-либо специализированной для графов структуре данных (неявный граф). Это возможно, если ребра графа выводятся (вычисляются) легко или с помощью некоторых правил. Одна из таких задач – UVa 11730 – Number Transformation.

Хотя описание задачи полностью математическое, в действительности основной задачей является поиск кратчайшего пути из одной вершины во все остальные (SSSP) в невзвешенном графе. Эта задача разрешима с помощью поиска в ширину. Граф генерируется динамически в процессе выполнения поиска в ширину. Источником является число *S*. Затем на каждом этапе поиска в ширину при обработке вершины *u* в очередь помещается непосещенная вер-

шина $u + x$, где x – простой множитель числа u, который не равен 1 и самому числу u. Счетчик уровней поиска в ширину позволяет следить за достижением целевой вершины T с минимальным количеством операций, необходимых для преобразования числа S в число T в соответствии с условиями задачи.

8.4.5. Два компонента: использование математики

В этом сочетании задач одним из компонентов является явная математическая задача, но этим условие не ограничивается. Обычно алгоритмы на графах, описанные в предыдущем подразделе, здесь неприменимы. Поэтому другим компонентом, как правило, является рекурсивный поиск с возвратами или бинарный поиск. Также возможно сочетание двух различных математических алгоритмов в одной задаче. В этом разделе рассматривается такой пример.

В задании UVa 10637 – Coprimes представлена задача разложения числа S ($0 < S \leqslant 100$) на t ($0 < t \leqslant 30$) взаимно простых чисел. Например, для числа $S = 8$ при $t = 3$ можно получить разложения 1 + 1 + 6, 1 + 2 + 5 или 1 + 3 + 4. После ознакомления с описанием задачи создается устойчивое ощущение того, что это чисто математическая задача (из теории чисел). Но для решения потребуется не только алгоритм решета Эратосфена для генерации простых чисел и алгоритм поиска наибольшего общего делителя для проверки, являются ли два числа взаимно простыми, но также подпрограмма поиска с возвратами для генерации всех возможных разложений.

8.4.6. Два компонента: полный поиск и геометрия

Многие задачи (вычислительной) геометрии можно решить простейшим методом грубой силы (хотя некоторые задачи требуют решения на основе принципа «разделяй и властвуй»). Если заданные входные ограничивающие условия позволяют применить решение с использованием полного поиска, то не следует пренебрегать такой возможностью.

Например, задание UVa 11227 – The silver bullet сводится к следующей задаче: задано N ($1 \leqslant N \leqslant 100$) точек на двумерной плоскости, определить максимальное количество точек, которые являются коллинеарными (то есть лежат на одной прямой). Вполне допустимо простое решение с использованием полного поиска $O(N^3)$, так как $N \leqslant 100$ (существует и более эффективное решение). Для каждой пары точек i и j проверяются все остальные $N - 2$ точки – принадлежат ли они линии $i - j$ (коллинеарны линии $i - j$). Это решение легко реализовать с помощью трех вложенных циклов и функции `bool collinear( point p, point q, point r )`, описанной в разделе 7.2.2.

8.4.7. Два компонента: использование эффективной структуры данных

Такая комбинация обычно возникает в некоторых «стандартных» задачах со значительным входным ограничивающим условием, из-за которого приходится использовать более эффективную структуру данных, чтобы избежать превышения лимита времени.

Например, задание UVa 11967 – Hic-Hac-Hoe представляет собой расширение настольной игры Tic-Tac-Toe. Вместо небольшого поля 3×3 в предлагаемом варианте поле не ограничено, «бесконечно». Поэтому невозможно представить игровое поле в виде двумерного массива. К счастью, можно сохранять координаты «крестиков» и «ноликов» в сбалансированном бинарном дереве поиска и обращаться к этому дереву для проверки состояния игры.

8.4.8. Три компонента

В разделах 8.4.1–8.4.7 рассматривались различные примеры задач, включающих два компонента. В этом подразделе описываются два примера более редко встречающихся сочетаний трех различных алгоритмов и/или структур данных.

Простые множители, динамическое программирование, бинарный поиск

Задание UVa 10856 – Recover Factorial можно кратко описать следующим образом: дано N – число простых множителей в X!. Найти минимальное возможное значение X ($N \leqslant 10\,000\,001$). Это достаточно трудная задача. Чтобы решить ее, необходимо разделить ее на несколько компонентов.

Сначала вычисляется количество простых множителей целого числа i, а результат сохраняется в таблице `NumPF[i]` в соответствии со следующим рекуррентным правилом: если i – простое число, то `NumPF[i] = 1` простой множитель, иначе если $i = \text{PF} \times i'$, то `NumPF[i] = 1` + количество простых множителей числа i'. Это количество простых множителей вычисляется $\forall i \in$ `[1..2703665]`. Верхняя граница данного диапазона получена методом проб и ошибок в соответствии с ограничениями, определенными в описании условий задачи.

Далее на втором этапе решения выполняется накопляющее суммирование количеств простых множителей числа N! по формуле `NumPF[i] += NumPF[i-1]`; $\forall i \in [1..N]$. Таким образом, в итоге `NumPF[N]` содержит количество простых множителей числа N!. Это решение методом динамического программирования для задачи запроса суммы одномерного статического диапазона (1D Static RSQ).

Теперь третий этап решения должен стать очевидным: можно выполнить бинарный поиск для определения индекса X, такого, что NumPF[X] = N. Если ответ не найден, то выводится сообщение «Not possible» (Решение невозможно).

Полный поиск, бинарный поиск, жадный алгоритм

В этой версии книги обсуждается задача по программированию, предложенная в финале соревнований на кубок мира ACM ICPC World Finals, которая объединяет три принципа решения задач, которые подробно рассматривались в главе 3, а именно: полный поиск, метод «разделяй и властвуй» (бинарный поиск) и жадный алгоритм.

Финал соревнований на кубок мира ACM ICPC World Finals 2009 – Задача A – A Careful Approach, LA 4445

Краткое описание задачи: рассматривается ситуация с регулированием посадок самолетов. В данном варианте имеется $2 \leqslant n \leqslant 8$ самолетов. Для каждо-

го самолета определен интервал времени («временное окно»), в котором он может безопасно приземлиться. Этот интервал времени задан двумя целыми числами a_i и b_i, представляющими начало и конец закрытого интервала $[a_i..b_i]$, в котором i-й самолет может безопасно приземлиться. Числа a_i и b_i заданы в минутах и соответствуют условию $0 \leqslant a_i \leqslant b_i \leqslant 1440$ (24 часа). В этой задаче можно предположить, что время посадки самолета не имеет значения. Необходимо решить следующие подзадачи:

1) вычислить порядок посадки всех самолетов с учетом соответствующих интервалов (окон) времени. Подсказка: порядок = перестановка = полный поиск?
2) дополнительное условие: операции посадки самолетов должны быть отделены друг от друга максимально возможным интервалом времени, то есть минимальный достижимый интервал между последовательными успешными посадками должен быть наибольшим возможным. Например, если три самолета последовательно садятся в 10:00, в 10:05 и в 10:15, то наименьший интервал равен пяти минутам, то есть времени между посадками первых двух самолетов. Не все интервалы должны быть одинаковы, но наименьший интервал между посадками должен быть максимальным из всех возможных. *Совет*: возможно, это похоже на задачу «покрытия интервала» (см. раздел 3.4.1);
3) вывод ответа (решения) с разделением на минуты и секунды с округлением до ближайшего целого значения секунд.

Графическое представление решения этой задачи показано на рис. 8.14, где линии обозначают допустимые интервалы времени посадки самолетов, а звездочки – оптимальный запланированный момент посадки самолета.

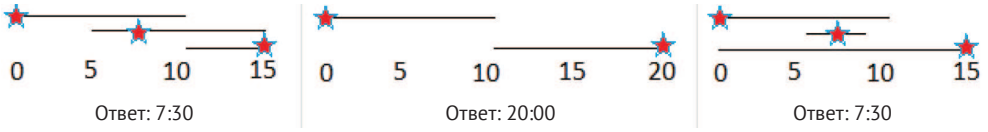


Рис. 8.14 ❖ Графическое представление решения задачи ACM ICPC WF2009 – A – A Careful Approach

Решение: поскольку количество самолетов не превышает 8, оптимальное решение может быть найдено простым перебором всех 8! = 40 320 возможных порядков посадки самолетов. Это компонент решения методом полного поиска, который можно легко реализовать с помощью функции `next_permutation` из модуля `algorithm` стандартной библиотеки C++ STL.

Далее для каждого вычисленного варианта порядка посадки самолетов необходимо знать наибольший возможный интервал (окно) времени посадки. Предположим, что мы считаем, что ответом является конкретный интервал времени L. С помощью жадного алгоритма можно проверить, является ли осуществимым такой интервал L, если приказать первому самолету садиться как можно скорее, а все последующие самолеты будут приземляться через время `max( a[этот_самолет], предыдущее_время_посадки` + L). Этот компонент представляет собой жадный алгоритм.

Размер интервала (окна) времени L, который слишком велик или слишком мал, приведет к тому, что при последней посадке `lastLanding` (см. исходный код ниже) произойдет либо «перелет», либо «недолет» `b[последний_самолет]`, поэтому необходимо уменьшить или увеличить интервал L. Этот компонент решения рассматриваемой задачи представляет собой метод «разделяй и властвуй». Так как нам нужен только ответ, округленный до ближайшего целого числа (секунд), вполне достаточно остановить бинарный поиск при достижении значения погрешности $\varepsilon < 1e - 3$. Для получения более подробной информации рекомендуем внимательно изучить исходный код, приведенный ниже.

```
// World Finals Stockholm 2009, A - A Careful Approach, UVa 1079, LA 4445

#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

int i, n, caseNo = 1, order[8];
double a[8], b[8], L, maxL;

double greedyLanding()       // при определенном порядке посадки и определенном интервале L
                      // попытки посадить эти самолеты и оценить интервал до b[order[n-1]]
{
  double lastLanding = a[order[0]];             // жадный алгоритм, 1-й самолет садится как
                                                                        // можно быстрее
  for( i=1; i < n; i++ ) {                                  // перебор всех других самолетов
    double targetLandingTime = lastLanding + L;
    if( targetLandingTime <= b[order[i]] )
      // возможна посадка: жадно выбираем максимум из a[order[i]] или targetLandingTime
      lastLanding = max( a[order[i]], targetLandingTime );
    else
      return 1;
  }
  // возвращается положительное значение для усиления бинарного поиска с уменьшением L
  // возвращается отрицательное значение для усиления бинарного поиска с увеличением L
  return lastLanding - b[order[n-1]];
}

int main()
{
  while( scanf( "%d", &n ), n ) {                                          // 2 <= n <= 8
    for( i=0; i < n; i++ ) {            // самолет i безопасно садится в интервале [ai, bi]
      scanf( "%lf %lf", &a[i], &b[i] );
      a[i] *= 60; b[i] *= 60;              // изначально время в минутах, перевод в секунды
      order[i] = i;
    }

    maxL = -1.0;                    // переменная для выполнения поиска перестановки порядка
    do {                                                          // посадок самолетов, до 8!
      double lo = 0, hi = 86400;       // минимум 0 секунд, максимум 1 день = 86 400 секунд
      L = -1;                                             // начинаем с невыполнимого решения
      while( fabs( lo - hi ) >= 1e-3 ) {      // бинарный поиск L, погрешность EPS = 1e - 3
        L = (lo + hi) / 2.0;          // нужен ответ, округленный до ближайшего целого числа
        double retVal = greedyLanding();    // округление в меньшую сторону, первый вариант
```

```
        if( retVal <= 1e-2 ) lo = L;                              // необходимо увеличить L
        else                 hi = L;                  // невыполнимо, необходимо уменьшить L
      }
      maxL = max( maxL, L );           // взять максимальное значение по всем перестановкам
    } while( next_permutation( order, order + n ) );           // перебор всех перестановок

    // другой способ округления - использование формата строки printf: %.0lf:%0.2lf
    maxL = (int)(maxL + 0.5);      // округление до ближайшего целого числа, второй вариант
    printf( "Case %d: %d:%0.2d\n", caseNo++, (int)(maxL/60), (int)(maxL%60) );
  }
  return 0;
}
```

Файл исходного кода: *ch8_04_UVa1079.cpp/java*

Упражнение 8.4.8.1. Исходный код, приведенный выше, получил оценку «решение принято» (Accepted), но в нем используется тип данных `double` для переменных `lo`, `hi` и `L`. В действительности в этом нет необходимости, так как все вычисления можно выполнять в целых числах. Кроме того, вместо цикла `while (fabs(lo-hi) >= 1e-3)` можно использовать цикл `for (int i=0; i < 50; i++)`. Перепишите код с учетом этих замечаний.

Задания по программированию, связанные с методикой декомпозиции задачи

- Два компонента – бинарный поиск ответа и другие алгоритмы
  1. UVa 00714 – Copying Books (бинарный поиск ответа + жадный алгоритм)
  2. UVa 01221 – Against Mammoths (LA 3795, Tehran06, бинарный поиск ответа + поиск паросочетания максимальной мощности в двудольном графе; использование алгоритма поиска увеличивающего пути для вычисления паросочетания – см. раздел 4.7.4)
  3. *UVa 01280 – Curvy Little Bottles* (LA 6027, WorldFinals Warsaw12, бинарный поиск ответа и геометрическая формула)
  4. *UVa 10372 – Leaps Tall Buildings…* (бинарный поиск ответа + физика)
  5. UVa 10566 – Crossed Ladders (метод бисекции)
  6. *UVa 10606 – Opening Doors* (решением является наибольший квадрат числа $\leqslant N$, но в этой задаче используется BigInteger; мы применяем (относительно медленный) метод бинарного поиска ответа, чтобы получить $\sqrt{N}$)
  7. UVa 10668 – Expanding Rods (метод бисекции)
  8. UVa 10804 – Gopher Strategy (аналогично заданию UVa 11262)
  9. UVa 10816 – Travel in Desert (бинарный поиск ответа + алгоритм Дейкстры)
  10. **UVa 10983 – Buy one, get… *** (бинарный поиск ответа + поиск максимального потока)
  11. **UVa 11262 – Weird Fence *** (бинарный поиск ответа + поиск паросочетания максимальной мощности в двудольном графе (МСВМ))

12. **UVa 11516 – WiFi *** (бинарный поиск ответа + жадный алгоритм)
13. UVa 11646 – Athletics Track (построение окружности в центре дорожки)
14. *UVa 12428 – Enemy at the Gates* (бинарный поиск ответа + немного теории графов, касающейся задачи о мостах, описанной в главе 4)
15. IOI 2009 – Mecho (бинарный поиск ответа + поиск в ширину)

- Два компонента – использование динамического программирования и запроса суммы или минимального значения из одномерного диапазона
  1. UVa 00967 – Circular (аналогично заданию UVa 897, но здесь этап вывода решения можно ускорить, используя динамическое программирование для вычисления суммы одномерного диапазона)
  2. UVa 10200 – Prime Time (полный поиск, проверка: if isPrime($n^2 + n + 41$) $\forall n \in [a..b]$; важное замечание: эту формулу генерации простых чисел $n^2 + n + 41$ вывел Леонард Эйлер; для $0 \leqslant n \leqslant 40$ она работает, но не обеспечивает достаточной точности для больших значений n; на завершающем этапе используйте динамическое программирование для запроса суммы одномерного диапазона для ускорения решения)
  3. UVa 10533 – Digit Primes (алгоритм решета (Эратосфена); проверка, является ли число простым; динамическое программирование для запроса суммы одномерного диапазона)
  4. UVa 10871 – Primed Subsequence (требуется запрос суммы одномерного диапазона)
  5. **UVa 10891 – Game of Sum *** (двойное применение динамического программирования; первый этап динамического программирования – стандартный запрос суммы одномерного диапазона между двумя индексами i, j; на втором этапе методом динамического программирования вычисляется дерево решения с состояниями (i, j) и перебор всех разделяющихся пунктов; метод минимакса)
  6. **UVa 11105 – Semi-prime H-numbers *** (требуется запрос суммы одномерного диапазона)
  7. **UVa 11408 – Count DePrimes *** (требуется запрос суммы одномерного диапазона)
  8. *UVa 11491 – Erasing and Winning* (жадный алгоритм, оптимизация с применением структуры данных типа разреженная таблица для обработки запроса минимального значения из статического диапазона)
  9. *UVa 12028 – A Gift from…* (генерация массива; сортировка этого массива; подготовка запроса суммы одномерного диапазона; после этого решение становится намного более простым)

- Два компонента – предварительная обработка графа и динамическое программирование
  1. **UVa 00976 – Bridge Building *** (использование некоторого типа заливки для разделения северного и южного берегов; применить этот метод для вычисления стоимости монтажа моста на каждой опоре;

после этой предварительной обработки решение методом динамического программирования должно стать вполне очевидным)

2. UVa 10917 – A Walk Through the Forest (вычисление путей в направленном ациклическом графе; но сначала необходимо сформировать этот направленный ациклический граф, выполняя алгоритм Дейкстры из «дома»)
3. UVa 10937 – Blackbeard the Pirate (поиск в ширину → задача коммивояжера; затем метод динамического программирования или поиск с возвратами; рассматривалась в этом разделе)
4. UVa 10944 – Nuts for nuts… (поиск в ширину → задача коммивояжера; затем применение метода динамического программирования, $n \leqslant 16$)
5. **UVa 11324 – The Largest Clique** * (самые длинные пути в направленном ациклическом графе; но сначала необходимо преобразовать исходный граф в направленный ациклический граф из компонентов сильной связности; топологическая сортировка)
6. **UVa 11405 – Can U Win?** * (поиск в ширину от k и каждого P – максимум 9 элементов; затем применяется динамическое программирование для задачи коммивояжера)
7. *UVa 11693 – Speedy Escape* (вычисление информации о кратчайших путях с использованием алгоритма Флойда–Уоршелла; затем применение динамического программирования)
8. UVa 11813 – Shopping (алгоритм Дейкстры → задача коммивояжера, затем применение динамического программирования, $n \leqslant 10$)

- Два компонента – использование алгоритмов на графах

1. *UVa 00273 – Jack Straw* (пересечение отрезков линий и алгоритм построения транзитивного замыкания Уоршелла)
2. *UVa 00521 – Gossiping* (формирование графа; вершины представляют водителей; построение ребра между двумя водителями, если они могут встретиться; это определяется по математическому правилу (поиск наибольшего общего делителя); если граф является связным, то ответ «да»)
3. *UVa 01039 – Simplified GSM Network* (LA 3270, World Finals Shanghai05, формирование графа с использованием простой геометрии; затем применение алгоритма Флойда–Уоршелла)
4. **UVa 01092 – Tracking Bio-bots** * (LA 4787, World Finals Harbin10, сначала выполняется сжатие графа; затем выполняется полный обход графа от «выхода» с использованием только направлений на юг и на запад; включение-исключение)
5. UVa 01243 – Polynomial-time Red… (LA 4272, Hefei08, алгоритм построения транзитивного замыкания Уоршелла, компоненты сильной связности, транзитивное сокращение в направленном графе)
6. UVa 01263 – Mines (LA 4846, Daejeon10, геометрия, компоненты сильной связности, см. две похожие задачи UVa 11504 и 11770)
7. UVa 10075 – Airlines (функция `gcDistance` – см. раздел 9.11 – с решением задачи о кратчайшем пути между всеми парами вершин (APSP))

8. UVa 10307 – Killing Aliens in Borg Maze (создание графа задачи о поиске кратчайшего пути из одной вершины во все остальные (SSSP) с применением поиска в ширину и минимального остовного дерева (MST))
9. UVa 11267 – The 'Hire-a-Coder'... (проверка двудольности графа, минимальное остовное дерево (MST), возможен отрицательный вес)
10. **UVa 11635 – Hotel Booking *** (алгоритм Дейкстры + поиск в ширину)
11. UVa 11721 – Instant View... (поиск узлов, которые можно привести к компонентам сильной связности с отрицательным циклом)
12. UVa 11730 – Number Transformation (разложение чисел на простые множители, см. 5.5.1)
13. UVa 12070 – Invite Your Friends (LA 3290, Dhaka05, поиск в ширину + алгоритм Дейкстры)
14. *UVa 12101 – Prime Path* (поиск в ширину, использование простых чисел)
15. **UVa 12159 – Gun Fight *** (LA 4407, KualaLumpur08, геометрия, паросочетание максимальной мощности в двудольном графе)

- Два компонента – использование математики
  1. *UVa 01195 – Calling Extraterrestrial...* (LA 2565, Kanazawa02, использование алгоритма решета (Эратосфена) для генерации списка простых чисел, метод грубой силы для каждого простого числа p и использование бинарного поиска для определения соответствующей пары q)
  2. *UVa 10325 – The Lottery* (принцип включения/исключения, метод грубой силы для вычисления подмножества для малых $M \leqslant 15$, наименьшее общее кратное – наибольший общий делитель)
  3. UVa 10427 – Naughty Sleepy... (числа в диапазоне $[10^{(k-1)}..10^k - 1]$, которые содержат k разрядов)
  4. **UVa 10539 – Almost Prime Numbers *** (алгоритм решета (Эратосфена); вычисление «почти простых» чисел: это не простые числа, которые делятся только на одно простое число; можно получить список «почти простых» чисел, перечисляя степени каждого простого числа, например 3 – простое число, а $3^2 = 9$, $3^3 = 27$, $3^4 = 81$ и т. д. – «почти простые» числа; далее можно отсортировать эти «почти простые» числа, затем выполнить бинарный поиск)
  5. **UVa 10637 – Coprimes *** (использование простых чисел и алгоритма вычисления наибольшего общего делителя)
  6. **UVa 10717 – Mint *** (полный поиск + алгоритмы поиска наибольшего общего делителя и наименьшего общего кратного, см. раздел 5.5.2)
  7. *UVa 11282 – Mixing Invitations* (беспорядок (перестановка) и биномиальный коэффициент, использование Java BigInteger)
  8. *UVa 11415 – Count the Factorials* (вычисление количества множителей для каждого целого числа, использование этого метода для поиска количества множителей для каждого числа-факториала и сохранение их в массиве; для каждого запроса – поиск в этом массиве для нахождения первого элемента с искомым значением с применением бинарного поиска)

9. UVa 11428 – Cubes (полный поиск + бинарный поиск)

- Два компонента – полный поиск и геометрия
  1. *UVa 00142 – Mouse Clicks* (метод грубой силы; точка в прямоугольнике; метод `dist`)
  2. UVa 00184 – Laser Lines (метод грубой силы; проверка на коллинеарность `collinear`)
  3. UVa 00201 – Square (вычисление квадратов различных размеров; перебор всех вариантов)
  4. UVa 00270 – Lining Up (градиентная сортировка, полный поиск)
  5. UVa 00356 – Square Pegs And Round… (евклидово расстояние, метод грубой силы)
  6. *UVa 00638 – Finding Rectangles* (метод грубой силы для четырех угловых точек)
  7. *UVa 00688 – Mobile Phone Coverage* (метод грубой силы; разбить область на маленькие прямоугольники и решить, покрывается ли отдельный маленький прямоугольник действием антенны; если покрывается, то добавить площадь этого маленького прямоугольника в ответ)
  8. **UVa 10012 – How Big Is It? *** (перебор всех 8! перестановок, евклидово расстояние `dist`)
  9. UVa 10167 – Birthday Cake (метод грубой силы A и B, проверки `ccw`)
  10. UVa 10301 – Rings and Glue (пересечение окружностей, поиск с возвратами)
  11. UVa 10310 – Dog and Gopher (полный поиск, евклидово расстояние `dist`)
  12. UVa 10823 – Of Circles and Squares (полный поиск; проверяется, находится ли точка внутри окружностей/квадратов)
  13. **UVa 11227 – The silver bullet *** (метод грубой силы; проверка на коллинеарность `collinear`)
  14. UVa 11515 – Cranes (пересечение окружностей, поиск с возвратами)
  15. **UVa 11574 – Colliding Traffic *** (метод грубой силы для всех пар судов; если одна из пар уже столкнулась, то ответ 0.0; иначе выводится квадратное уравнение для определения, когда столкнутся эти два судна, если такая возможность существует; выбор общего минимального времени столкновения; если столкновения не происходит, то выводится сообщение «No collision»)

- Объединение с эффективными структурами данных
  1. *UVa 00843 – Crypt Kicker* (поиск с возвратами; попытка перебора отображения каждой буквы в другую букву алфавита; использование структуры данных Trie (бор, префиксное дерево, луч – см. раздел 6.6) для ускорения, если определенное слово (или часть слова) присутствует в словаре)
  2. *UVa 00922 – Rectangle by the Ocean* (сначала вычислить площадь многоугольника; затем для каждой пары точек определить прямоугольник с помощью этих двух точек; использовать структуру данных `set` для проверки, находится ли третья точка этого прямоугольника

внутри многоугольника; проверка: найденный вариант лучше, чем текущий наилучший)

3. *UVa 10734 – Triangle Partitioning* (в действительности это геометрическая задача с использованием формул треугольника и косинусов, но здесь мы применяем структуру данных, которая допускает неточность при вычислениях с плавающей точкой из-за нормализации стороны треугольника, чтобы убедиться в том, что каждый треугольник обрабатывается только один раз)
4. **UVa 11474 – Dying Tree *** (использовать систему непересекающихся множеств (union-find); сначала объединить все ветви в дереве; далее соединить одно дерево с другим деревом, если между любыми из их ветвей расстояние меньше k (немного геометрии); затем любое дерево, которое может достать до любого доктора, соединить с этим доктором; в конце проверка: соединена ли первая ветвь первого/больного дерева с любым доктором; код может быть достаточно длинным; будьте внимательны)
5. **UVa 11525 – Permutation *** (можно воспользоваться деревом Фенвика (Fenwick tree) и бинарным поиском ответа для нахождения наименьшего индекса i, который имеет RSQ(1, i) = Si)
6. **UVa 11960 – Divisor Game *** (модифицированный алгоритм решета (сита), количество делителей, запрос максимального значения из статического диапазона, задача разрешима с помощью структуры данных типа разреженная таблица)
7. UVa 11966 – Galactic Bounding (использовать систему непересекающихся множеств (union-find) для отслеживания количества непересекающихся множеств/созвездий; если евклидово расстояние $\leqslant D$, то объединить две звезды)
8. *UVa 11967 – Hic-Hac-Hoe* (простой метод грубой силы, но потребуется использование C++ STL `map`, так как невозможно сохранить (бесконечное) поле для этой игры; сохраняется только n координат, и выполняется проверка: существует ли k последовательных (смежных) координат, принадлежащих (любому) одному игроку)
9. *UVa 12318 – Digital Roulette* (метод грубой силы с использованием структуры данных `set`)
10. *UVa 12460 – Careful teacher* (задача поиска в ширину, но необходима структура данных `set` для хранения строк, чтобы ускорить процесс)

- Три компонента

1. **UVa 00295 – Fatman *** (бинарный поиск ответа x для следующего вопроса: если персонаж имеет диаметр x, то может ли он перейти слева направо? Для любой пары препятствий (включая верхнюю и нижнюю стены) создать ребро между препятствиями, если персонаж не может пройти между ними, и проверить: если верхняя и нижняя стены не соединены, то персонаж с диаметром x может пройти; евклидово расстояние)
2. UVa 00811 – The Fortified Forest (LA 5211, World finals Eindhoven99, CH, вычисление периметра многоугольника `perimeter`, итеративная генерация всех подмножеств с использованием битовой маски)

3. **UVa 01040 – The Traveling Judges** * (LA 3271, World Finals Shanghai05, итеративный полный поиск, перебор всех подмножеств из 2^{20} городов, формирование минимального остовного дерева (MST) из этих городов с помощью системы непересекающихся множеств (union-find), сложное форматирование вывода результата)
4. UVa 01079 – A Careful Approach (LA 4445, World Finals Stockholm09, рассматривалась в этой главе)
5. *UVa 01093 – Castles* (LA 4788, World Finals Harbin10, перебор всех возможных корней, динамическое программирование на дереве)
6. UVa 01250 – Robot Challenge (LA 4607, SoutheastUSA09, геометрия, задача о поиске кратчайшего пути из одной вершины во все остальные (SSSP) в направленном ациклическом графе → динамическое программирование, динамическое программирование для вычисления суммы одномерного диапазона)
7. UVa 10856 – Recover Factorial (решение рассматривалось в этом разделе)
8. *UVa 10876 – Factory Robot* (бинарный поиск ответа + геометрия, евклидово расстояние + система непересекающихся множеств (union-find), аналогично заданию UVa 00295)
9. **UVa 11610 – Reverse Prime** * (сначала «переворачиваются» (реверсируются) простые числа, меньшие 10^6; далее дополняются нулями, если необходимо; использование дерева Фенвика и бинарный поиск)

8.5. Решения упражнений, не помеченных звездочкой

Упражнение 8.2.3.1. При использовании языка C++ можно воспользоваться структурой `pair<int, int>` (сокращенная форма: `ii`) для хранения пары элементов информации (целых чисел). Для тройки можно использовать структуру `pair<int, ii>`. Для четверки – `pair<ii, ii>`. При применении языка Java нет возможности воспользоваться стандартными структурами, поэтому необходимо создать класс с реализацией аналогичных объектов (при этом можно использовать структуру данных Java `TreeMap` для правильного хранения этих объектов).

Упражнение 8.2.3.2. Здесь нет другого выбора, кроме использования класса как в языке C++, так и в языке Java. Также возможно применение структуры struct языка C/C++. Затем необходимо реализовать функцию сравнения для этого класса.

Упражнение 8.2.3.3. Поиск в пространстве состояний по существу представляет собой расширение задачи о поиске кратчайшего пути из одной вершины во все остальные (SSSP), которая является задачей минимизации. Задача поиска самого длинного пути (задача максимизации) является NP-трудной, поэтому обычно мы не рассматриваем этот вариант, поскольку сам по себе поиск в пространстве состояний (задача минимизации) уже является достаточно сложной задачей для решения.

Упражнение 8.3.1.1. Решение аналогично решению задания UVa 10911, показанному в разделе 1.2. Но в задаче «поиска паросочетания максимальной мощности» существует вероятность того, что некоторая вершина не попадет в паросочетание. Решение методом динамического программирования с применением битовой маски для небольшого обобщенного графа показано ниже.

```
int MCM( int bitmask )
{
  if( bitmask == (1 << N) - 1 )                                  // все вершины учтены
    return 0;                                          // больше нет возможных паросочетаний
  if( memo[bitmask] != -1 )
    return memo[bitmask];

  int p1, p2;
  for( p1=0; p1 < N; p1++ )                                      // поиск свободной вершины p1
    if( !(bitmask & (1 << p1)) )
      break;

  // Это главное (ключевое) различие:
  // У нас есть возможность НЕ брать вершину p1 в паросочетание
  int ans = MCM( bitmask | (1 << p1) );

  // Предположим, что рассматриваемый небольшой граф хранится в матрице смежности AdjMat
  for( p2=0; p2 < N; p2++ )                                  // поиск свободных соседей вершины p1
    if( AdjMat[p1][p2] && p2 != p1 && !(bitmask & (1 << p2)) )
      ans = max( ans, 1 + MCM( bitmask | (1 << p1) | (1 << p2) ) );

  return memo[bitmask] = ans;
}
```

Упражнение 8.4.8.1. Решение см. в разделе 3.3.1.

8.6. Замечания к главе

Мы существенно улучшили содержимое главы 8, как и обещали в замечаниях к этой главе в предыдущем (втором) издании. В этом (третьем) издании глава 8 содержит приблизительно в два раза больше страниц и упражнений по двум причинам. Во-первых, мы решили несколько более трудных задач в интервале между вторым и третьим изданиями. Во-вторых, мы переместили некоторые более трудные задачи, которые ранее рассматривались в предыдущих главах (во втором издании), в главу 8 – особенно из главы 7 (в раздел 8.4.6).

В третьем издании глава 8 уже не является самой последней главой. Мы добавили главу 9, в которой рассматриваются темы, редко встречающиеся на олимпиадах по программированию, которые могут заинтересовать любознательных читателей.

Таблица 8.1

Статистические характеристики	Первое издание	Второе издание	Третье издание
Количество страниц	–	15	33 (+120 %)
Описанные задания	–	3	5 + 8* = 13 (+333 %)
Задания по программированию	–	83	177 (+113 %)

Количество упражнений по программированию в каждом разделе показано в табл. 8.2.

Таблица 8.2

Раздел	Название	Количество	% в главе	% в книге
8.2	Более эффективные методы поиска	36	20	2
8.3	Более эффективные методы динамического программирования	51	29	3
8.4	Декомпозиция задачи	90	51	5

Рис. 8.15 ❖ Авторы книги впервые присутствуют вместе на финальных соревнованиях на кубок мира ACM ICPC World Finals

Глава 9

Малораспространенные темы

Знание – сокровище, которое повсюду следует за тем, кто им обладает.

– Китайская пословица

Общий обзор и мотивация

В этой главе мы рассмотрим малораспространенные «экзотические» темы информатики (computer science), которые могут встречаться (хотя обычно не появляются) на обычной олимпиаде по программированию. Эти задачи, структуры данных и алгоритмы в основном имеют одноразовое применение, в отличие от более обобщенных тем, которые рассматривались в главах 1–8. Изучение тем этой главы можно воспринимать как «невыгодное, нерентабельное» занятие, поскольку при существенных затратах на изучение конкретной темы велика вероятность того, что она вообще не появится на олимпиаде по программированию. Тем не менее мы уверены в том, что такие малораспространенные темы будут весьма привлекательны для тех, кто действительно желает расширить свои знания в области информатики.

Эту главу можно пропустить без какого-либо ущерба для подготовки к международным студенческим олимпиадам по программированию (ICPC) и подобным соревнованиям, поскольку вероятность появления любой из тем этой главы в любом случае остается крайне низкой[1]. Но даже если такие малораспространенные темы появляются, участники, обладающие предварительными знаниями по этим темам, получат преимущество над прочими участниками, не имеющими подобных знаний. Возможно, некоторые хорошо подготовленные участники смогут вывести решение из основных концепций непосредственно во время олимпиады, даже если они встречаются с таким типом задач впервые, но обычно они находят решение гораздо медленнее, чем участники, предварительно ознакомившиеся с задачей и особенно с ее решением.

---

[1] Некоторые из этих тем, также с весьма низкой вероятностью, используются как вопросы на интервью с кандидатами, претендующими на работу в IT-компаниях.

На международных олимпиадах по информатике для школьников (IOI) малораспространенные темы не включаются в программу соревнований [20]. Таким образом, участники международных олимпиад по информатике для школьников могут отложить изучение материала данной главы до начала обучения в высшем учебном заведении (университете).

В этой главе стиль обсуждения каждой темы остается предельно лаконичным, то есть почти все описания занимают одну-две страницы. В большинстве обсуждений не содержится примеров исходного кода, поскольку читатели, тщательно изучившие материал глав 1–8, не должны испытывать каких-либо затруднений при преобразовании приведенных здесь алгоритмов в работающий код. В этой главе имеется лишь несколько помеченных звездочкой упражнений (без советов/решений).

Рассматриваемые здесь малораспространенные темы перечислены в алфавитном порядке в списке содержания в начале книги. Но если вы не обнаружили название, используемое в этой главе, то рекомендуем воспользоваться предметным указателем в конце книги, чтобы проверить, не используются ли другие названия рассматриваемых здесь тем.

9.1. Задача 2-SAT

Описание задачи

Имеется конъюнкция (логическое умножение) дизъюнкций (логических сложений) (AND of ORs), где каждая дизъюнкция (операция OR) содержит два аргумента, которые могут быть переменными или отрицанием переменных. Дизъюнкции таких пар называются дизъюнктами (clauses), а формула известна под названием 2-КНФ (конъюнктивная нормальная форма) (2-CNF – conjunctive normal form). Задача 2-SAT – найти правильные присваиваемые этим переменным логические значения (то есть истина (true) или ложь (false)), при которых формула 2-КНФ является истинной, то есть в каждой скобке должен содержаться по крайней мере один член, вычисление которого дает результат true (истина).

Пример 1: $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ – решение существует, так как можно присвоить x_1 = true и x_2 = false (другой вариант присваивания: x_1 = false и x_2 = true).

Пример 2: $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ – это выражение решения не имеет. Можно перебрать все восемь возможных комбинаций логических значений x_1, x_2 и x_3 и убедиться в том, что ни одна из этих комбинаций не дает в результате истинности формулы 2-КНФ.

Решения

Полный перебор

Участники, обладающие лишь поверхностными знаниями о задаче выполнимости (Satisfiability), могут подумать, что эта задача является NP-полной, следовательно, попытаются применить решение в форме полного перебора. Если формула 2-КНФ содержит n переменных и m скобок, то попытка перебо-

ра всех возможных 2^n вариантов присваивания и проверка каждого варианта присваивания при сложности $O(m)$ имеет обобщенную временну́ю сложность $O(2^n \times m)$. Вероятнее всего, это приведет к превышению лимита времени (TLE – Time Limit Exceeded).

Задача 2-SAT представляет собой особый частный случай задачи выполнимости, для которой предлагается полиномиальное решение, один из вариантов которого рассматривается ниже.

Приведение к графу импликаций и поиск компонент сильной связности

Сначала необходимо определить, что скобка в формуле 2-КНФ действительно удовлетворяет условию $(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$. Таким образом, взяв за основу верную формулу 2-КНФ, можно построить соответствующий импликационный граф (implication graph). В этом импликационном графе каждая переменная имеет две вершины: сама переменная и отрицание/инверсия этой переменной[1]. Ребро соединяет одну вершину с другой, если эти переменные связаны друг с другом импликацией в соответствующей формуле 2-КНФ. Для двух приведенных выше примеров формул 2-КНФ импликационные графы показаны на рис. 9.1.

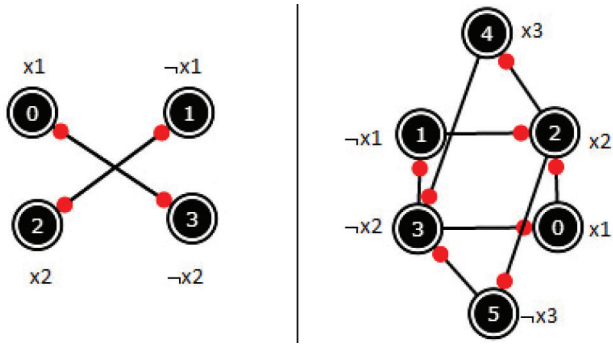


Рис. 9.1 ❖ Импликационные графы для примера 1 (слева) и для примера 2 (справа)

Как можно видеть на рис. 9.1, формула 2-КНФ с n переменными (исключая отрицания) и m клаузулами имеет $V = \theta(2n) = \theta(n)$ вершин и $E = O(2m) = O(m)$ ребер в импликационном графе.

Теперь формула 2-КНФ является выполнимой (имеет решение) тогда и только тогда, если «не существует переменной, принадлежащей той же компоненте сильной связности, что и ее отрицание».

На рис. 9.1 слева видно, что существуют две компоненты сильной связности: {0,3} и {1,2}. Поскольку здесь нет переменных, принадлежащих тому же компо-

[1] Прием программирования: переменной присваивается индекс i, а ее отрицанию – следующий индекс $i + 1$. Таким образом, можно находить переменную по ее отрицанию, и наоборот, используя битовую операцию $i \oplus 1$, где $\oplus$ – оператор «исключающее ИЛИ» (XOR).

ненту сильной связности, что и соответствующие им отрицания, мы делаем вывод о том, что формула 2-КНФ из примера 1 является выполнимой.

На рис. 9.1 справа можно видеть, что все шесть вершин принадлежат одному и тому же компоненту сильной связности. Следовательно, вершина 0 (представляющая x_1) и вершина 1 (представляющая[1] $\neg x_1$), вершина 2 (x_2) и вершина 3 ($\neg x_2$), вершина 4 (x_3) и вершина 5 ($\neg x_3$) находятся в одном и том же компоненте сильной связности. Таким образом, мы приходим к выводу о том, что формула 2-КНФ из примера 2 не является выполнимой.

Для поиска компонента сильной связности в ориентированном графе (орграфе) можно воспользоваться алгоритмом Тарьяна (Tarjan's SCC), рассматриваемым в разделе 4.2.9, или алгоритмом Косараджу (Kosaraju's SCC; иногда встречается написание Косарайю), рассматриваемым в разделе 9.17.

Упражнение 9.1.1*. Для поиска действительно верного присваивания логических значений необходимо выполнить немного больше работы, нежели простая проверка отсутствия переменных, принадлежащих тому же компоненту сильной связности, что и их отрицания. Какие дополнительные действия требуются для действительного нахождения верных вариантов присваивания логических значений в выполнимой формуле 2-КНФ?

Задание по программированию, связанное с задачей @-SAT

1. **UVa 10319 – Manhattan *** (самая трудная часть решения задач, включающих 2-SAT, состоит в определении того факта, что это действительно задача типа 2-SAT, и в последующем построении графа импликаций. Для рассматриваемой здесь задачи каждая улица (street) и авеню (avenue) определяются как переменные, для которых истина (true) означает, что они могут использоваться только в одном направлении, а ложь (false) означает, что они могут использоваться лишь в другом направлении. Простой путь будет представлен в одной из следующих форм: (street a ∧ avenue b) ∨ (avenue c ∧ street d). Такую форму можно преобразовать в формулу 2-КНФ: $(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$. Необходимо создать граф импликаций и проверить, является ли он выполнимым, используя критерий принадлежности к компоненту сильной связности, как описано выше. Следует отметить, что существует особый случай, когда скобка содержит только один литерал, то есть простой путь использует лишь одну улицу (street) или лишь одну авеню (avenue))

---

[1] Следует отметить, что при использовании приема программирования, описанного выше, мы можем с легкостью проверить, что вершина 0 и вершина 1 представляют переменную и ее отрицание, выполнив операцию if $1 = 0 \oplus 1$.

9.2. Задача о картинной галерее

Описание задачи

Задача о картинной галерее (Art gallery problem), или музейная задача, – это хорошо известное семейство задач видимости (visibility), или просматриваемости, в вычислительной геометрии. В этом разделе рассматривается несколько ее вариантов. Общепринятые термины и объекты, используемые в рассматриваемых здесь вариантах: простой (не обязательно выпуклый) многоугольник P для описания помещения картинной галереи (музея); множество точек S для описания охранников, при этом каждый охранник представлен точкой в многоугольнике P; правило, по которому точка $A \in S$ защищает другую точку $B \in P$ тогда и только тогда, когда $A \in S$, $B \in P$ и отрезок AB расположен внутри многоугольника P. Вопрос: все ли точки внутри многоугольника P защищены множеством точек S. Многие варианты задачи о картинной галерее классифицированы как P-полные задачи. В этой книге мы сосредоточимся на задачах, имеющих полиномиальные решения.

1. Вариант 1: определение верхней границы наименьшего количества точек S.
2. Вариант 2: определение существования критической точки C в многоугольнике P и существования другой точки $D \in P$ – таких, что если охранник находится в позиции C, то не может защищать точку D.
3. Вариант 3: определение возможности охраны многоугольника P только одним охранником.
4. Вариант 4: определение наименьшего количества точек в S, если охранники могут располагаться только в вершинах многоугольника P и лишь вершины должны быть охраняемыми.

Отметим, что существует намного больше вариантов этой задачи, и по крайней мере одна книга[1] была написана на эту тему [49].

Решения

1. Решение для варианта 1 – теоретическая работа – доказательство теоремы Вацлава Хватала (Václav Chvátal) о картинной галерее. Хватал доказал, что $[n/3]$ охранников всегда достаточно и иногда необходимо для того, чтобы защитить простой многоугольник с n вершинами (здесь доказательство не приводится).
2. Решение для варианта 2 предполагает проверку факта невыпуклости многоугольника P (следовательно, и наличия критической точки). Здесь можно воспользоваться противоположной версией функции `isConvex`, приведенной в разделе 7.3.4.
3. Решение для варианта 3 может стать затруднительным для того, кто ранее не был знаком с подобным решением. Можно воспользоваться функ-

[1] PDF-версия этой книги размещена по адресу http://cs.smith.edu/~orourke/books/ArtGalleryTheorems/art.html.

цией `cutPolygon`, описанной в разделе 7.3.6. Отсечение многоугольника *P* выполняется с помощью всех линий, образованных ребрами *P*, в направлении против часовой стрелки, и всегда оставляется левая сторона. Если в конце этой операции останется непустой многоугольник, то один охранник может быть помещен в этот непустой многоугольник с возможностью охраны всего многоугольника *P* полностью.

4. Решение для варианта 4 предполагает вычисление минимального вершинного покрытия (minimum vertex cover) «графа видимости» многоугольника *P*. Вообще говоря, это еще одна *P*-полная задача.

Задания по программированию, связанные с задачей о картинной галерее

1. **UVa 00588 – Video Surveillance *** (см. приведенное выше решение варианта 3)
2. **UVa 10078 – Art Gallery *** (см. приведенное выше решение варианта 2)
3. **UVa 10243 – Fire; Fire; Fire *** (вариант 4: эта задача может быть приведена к задаче минимального вершинного покрытия в дереве (Tree); для этого варианта существует полиномиальное DP-решение, которое рассматривается в разделе 4.7.1)
4. LA 2512 – Art Gallery (см. приведенное выше решение для варианта 3 и вычисление площади многоугольника)
5. LA 3617 – How I Mathematician… (вариант 3)

9.3. Битоническая задача коммивояжера

Описание задачи

Битоническая задача коммивояжера (Bitonic Traveling Salesman Problem (TSP)) может быть описана следующим образом: задан список координат вершин размера *n* в двумерном евклидовом пространстве, предварительно отсортированный по координатам *x* (и при равенстве координат *x* – по координатам *y*). Выполняется поиск маршрута, который начинается с крайней левой вершины, затем следует строго слева направо до того момента, пока не достигнет крайней правой вершины, после чего маршрут прокладывается строго справа налево обратно к начальной вершине. Такое поведение маршрута называется битоническим (bitonic).

Полученный в итоге маршрут может не являться самым коротким возможным путем при стандартном определении задачи о коммивояжере (см. раздел 3.5.2). На рис. 9.2 показано сравнение этих двух вариантов задачи о коммивояжере. Маршрут коммивояжера 0-3-5-6-4-1-2-0 не является битоническим, хотя сначала выдерживает направление слева направо (0-3-5-6), затем возвращается справа налево (6-4-1). В конце маршрута имеются участки, на которых он опять «разворачивается» в направлении слева направо (1-2), затем возвращается к направлению справа налево (2-0). Другой маршрут 0-2-3-5-6-

4-1-0 является правильным решением битонической задачи коммивояжера, поскольку его можно разделить на два фрагмента пути: 0-2-3-5-6, который следует строго в направлении слева направо, и 6-4-1-0, сохраняющий направление справа налево.

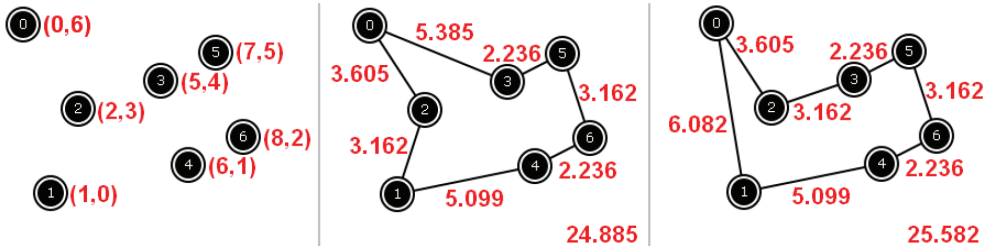


Рис. 9.2 ❖ Сравнение стандартной задачи коммивояжера с битонической задачей коммивояжера

Решения

Несмотря на то что маршрут битонической задачи коммивояжера по набору из n вершин обычно длиннее, чем маршрут стандартной задачи коммивояжера, это битоническое ограничение позволяет вычислить «достаточно хороший маршрут» за время $O(n^2)$ с использованием динамического программирования (dynamic programming), как будет показано ниже, по сравнению со временем $O(2^n \times n^2)$ для нахождения маршрута стандартной задачи коммивояжера (см. раздел 3.5.2).

Основным положением, необходимым для вывода решения с использованием динамического программирования, является тот факт, что мы можем (и должны) разделить искомый маршрут на два фрагмента пути: слева направо (ЛП) и справа налево (ПЛ). Оба фрагмента пути включают вершину 0 (крайнюю левую вершину) и вершину $n - 1$ (крайнюю правую вершину). Фрагмент пути ЛП начинается с вершины 0 и заканчивается в вершине $n - 1$. Фрагмент пути ПЛ начинается с вершины $n - 1$ и заканчивается в вершине 0.

Напомним, что все вершины были отсортированы по координатам x (и при равенстве координат x – по координатам y). Поэтому можно рассматривать вершины поочередно, одну за другой. Оба фрагмента пути ЛП и ПЛ начинаются с вершины 0. Пусть v – следующая рассматриваемая вершина. Для каждой вершины $v \in [1...n - 2]$ определяется, необходимо ли добавить вершину v как следующую точку фрагмента пути ЛП (для расширения фрагмента пути ЛП вправо) или как предыдущую точку обратного фрагмента пути ПЛ (в текущий момент фрагмент пути ПЛ начинается с вершины v и возвращается в вершину 0). Для этого необходимо отслеживать еще два параметра: $p1$ и $p2$. Пусть $p1/p2$ представляют конечную/начальную вершину в фрагментах пути ЛП/ПЛ соответственно.

Самый простой случай – когда вершина $v = n - 1$, то есть нужно просто соединить фрагменты пути ЛП и ПЛ с вершиной $n - 1$.

С учетом всех приведенных выше соображений можно написать простое решение с использованием динамического программирования, подобное приведенному ниже.

```
double dp1( int v, int p1, int p2 ) {           // вызывается с параметрами dp1( 1, 0, 0 )
  if( v == n-1 )
    return d[p1][v] + d[v][p2];
  if( memo3d[v][p1][p2] > -0.5 )
    return memo3d[v][p1][p2];
  return memo3d[v][p1][p2] = min(
    d[p1][v] + dp1( v+1, v, p2 ),       // расширяется ЛП: p1->v, ПЛ остается в вершине p2
    d[v][p2] + dp1( v+1, p1, v ));      // ЛП остается в вершине p1, ПЛ расширяется: p2<-v
}
```

Но сложность по времени решения `dp1` с тремя параметрами (`v`, `p1`, `p2`) составляет $O(n^3)$. Это неэффективное решение, так как параметр v может быть отброшен и восстановлен из выражения 1 + max($p1$, $p2$) (см. описание этой методики оптимизации динамического программирования посредством отбрасывания одного параметра и восстановления по другим параметрам в разделе 8.3.6). Улучшенное решение с использованием динамического программирования, приведенное ниже, имеет сложность $O(n^2)$.

```
double dp2( int p1, int p2 ) {                  // вызывается с параметрами dp2( 0, 0 )
  int v = 1 + max( p1, p2 );        // одна эта строка ускоряет поиск маршрута битонической
                                                                    // задачи коммивояжера
  if( v == n-1 )
    return d[p1][v] + d[v][p2];
  if( memo2d[p1][p2] > -0.5 )
    return memo2d[p1][p2];
  return memo2d[p1][p2] = min(
    d[p1][v] + dp2( v, p2 ),            // расширяется ЛП: p1->v, ПЛ остается в вершине p2
    d[v][p2] + dp2( p1, v ));           // ЛП остается в вершине p1, ПЛ расширяется: p2<-v
}
```

Задания по программированию, связанные с битонической задачей коммивояжера

1. **UVa 01096 – The Islands** * (LA 4791, World Finals Harbin10, вариант битонической задачи коммивояжера; графический вывод реального найденного маршрута (пути))
2. **UVa 01347 – Tour** * (LA 3305, Southeastern Europe 2005, это собственно основная версия битонической задачи коммивояжера, поэтому можно начать именно с нее)

9.4. Разбиение скобок на пары

Описание задачи

Программистам хорошо известны разнообразные формы скобок – (), {}, [] и т. д., – поскольку они достаточно часто используются в исходном коде, особенно в операторах if и в циклах. Скобки могут быть вложенными: (()), {{}}, [[]] и т. д. Правильно оформленный и отформатированный код непременно должен содержать правильную скобочную последовательность. Задача о соответствии скобок обычно включает вопрос о правильности (сбалансированности) уровней вложенности в заданном наборе скобок. Например, (()), ({}), (){} [] – правильное соответствие (сбалансированность) скобок, а в случаях ((), (},)(наблюдается некорректность в наборах скобок.

Решения

Скобки поочередно считываются слева направо. Каждый раз, когда встречается закрывающая скобка, необходимо найти соответствующую ей самую последнюю открывающую скобку (того же типа). Затем эта пара удаляется из рассмотрения, и процесс продолжается. Для этого требуется структура данных типа «последним пришел – первым вышел» (LIFO), то есть стек (stack) (см. раздел 2.2).

Процесс начинается с пустым стеком. Когда встречается открывающая скобка, она помещается в стек. При обнаружении закрывающей скобки проверяется совпадение ее типа с типом скобки, находящейся на вершине стека. Скобка на вершине стека должна соответствовать текущей закрывающей скобке. Если обнаружено соответствие, то скобка на вершине извлекается из стека и удаляется из дальнейшего рассмотрения. Только в том случае, когда обработана самая последняя скобка (то есть найдено соответствие скобок) и обнаружено, что стек снова пуст, можно с уверенностью сказать, что скобки образуют правильную скобочную последовательность.

Поскольку каждая из n скобок проверяется только один раз, а все операции со стеком имеют сложность $O(1)$, то сложность алгоритма в целом равна $O(n)$.

Варианты

Количество правильных скобочных последовательностей с n парами скобок вычисляется по формуле Каталана (Catalan) (см. раздел 5.4.3). Оптимальный метод умножения матриц (то есть задача о порядке перемножения матриц (Matrix Chain Multiplication)) также подразумевает определение сбалансированности скобок. Этот вариант можно решить с использованием динамического программирования (см. раздел 9.20).

Задания по программированию, связанные с задачей о соответствии скобок:

1. **UVa 00551 – Nesting a Bunch of... *** (классическая задача о соответствии скобок с использованием стека (метод `stack`))
2. **UVa 00673 – Parentheses Balance *** (классическая задача, аналогичная задаче UVa 551)
3. **UVa 11111 – Generalized Matrioshkas *** (задача о соответствии скобок с дополнением некоторыми особенностями)

9.5. Задача китайского почтальона

Описание задачи

Задача китайского почтальона (Chinese Postman Problem)[1], или маршрут почтальона, или задача инспекции дорог (Route Inspection Problem), – это задача поиска (длины) кратчайшего маршрута/циклического обхода с посещением каждого ребра (связного) неориентированного взвешенного графа. Если в графе есть эйлеров цикл (замкнутый маршрут, который проходит любое ребро ровно один раз; см. раздел 4.7.3), то сумма весов ребер вдоль этого циклического маршрута, покрывающего все ребра в эйлеровом графе, является оптимальным решением для данной задачи. Но это самый простой случай. Если граф не является эйлеровым, например показанный на рис. 9.3 слева, то решение задачи китайского почтальона найти труднее.

Решения

Для решения данной задачи важно четко понять, что неэйлеров граф G обязательно должен иметь четное число вершин нечетной степени (в соответствии с леммой о рукопожатиях, доказанной самим Эйлером). Назовем подмножество вершин графа G, которые имеют нечетные степени, именем T. Далее создадим полный граф K_n, где n – размер подмножества T. Набор T формирует вершины графа K_n. Ребро (i, j) в графе K_n имеет вес (весовой коэффициент), представляющий вес кратчайшего пути от i до j, например на рис. 9.3 (средний граф) ребро 2-5 в графе K_4 имеет вес 2 + 1 = 3, вычисленный из пути 2-4-5, а ребро 3-4 в графе K4 имеет вес 3 + 1 = 4, вычисленный из пути 3-5-4.

Теперь если удвоить ребра, выбранные по совершенному паросочетанию минимального веса в этом полном графе K_n, то будет выполнено преобразование не эйлерова графа G в другой граф G', который является эйлеровым. Причина в том, что при удвоении таких ребер мы в действительности добавляем ребро между парой вершин с нечетной степенью (следовательно, превращаем

[1] Такое название задача получила потому, что впервые ее исследовал китайский математик Квон Мей-Ко в 1960 году (в 1962 году исследование было переведено на английский язык).

их в вершины с четной степенью). Совершенное паросочетание минимального веса гарантирует, что это преобразование выполнено способом с минимальной стоимостью. Решением с совершенным паросочетанием минимального веса для графа K_4, показанного на рис. 9.3 (в середине), является выбор ребра 2-4 (с весом 2) и ребра 3-5 (с весом 3).

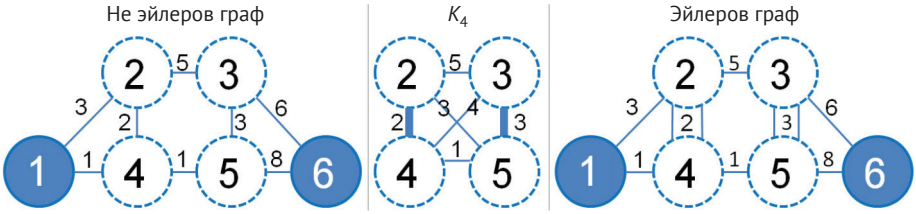


Рис. 9.3 ❖ Пример задачи китайского почтальона

После удвоения ребра 2-4 и ребра 3-5 мы возвращаемся к простому случаю задачи китайского почтальона. На рис. 9.3 (справа) показан эйлеров граф. В таком эйлеровом графе искомый маршрут прост. Один из таких маршрутов: 1->2->4->5->3->6->5->3->2->4->1 с суммарным весом 34 (сумма весов всех ребер в преобразованном эйлеровом графе G', которая представляет собой сумму весов всех ребер в графе G плюс стоимость совершенного паросочетания минимального веса в графе K_n).

Таким образом, самая трудная часть решения задачи китайского почтальона состоит в нахождении совершенного паросочетания минимального веса для графа K_n, который не является двудольным графом. Если значение n мало, то задачу можно решить с помощью динамического программирования с применением битовых масок, описанного в разделе 8.3.1.

Задание по программированию, связанное с задачей китайского почтальона

1. **UVa 10296 – Jogging Trails *** (см. описание выше)

9.6. Задача о паре ближайших точек

Описание задачи

Задан набор S из n точек на двумерной плоскости. Необходимо найти две точки с наименьшим евклидовым расстоянием между ними.

Решения

Полный перебор

Простейшее решение: вычисление расстояний между всеми парами точек и вывод минимального расстояния. Но для этого потребуется время $O(n^2)$.

Разделяй и властвуй

Можно воспользоваться следующей стратегией «разделяй и властвуй», чтобы сократить время выполнения до $O(n \times \log n)$. Для этого необходимо выполнить следующие три шага.

1. Разделение: точки в наборе S сортируются по координатам x (при равенстве координат x выполняется дополнительная сортировка по координате y). Затем набор S делится на два набора точек S_1 и S_2 с помощью вертикальной линии $x = d$ так, чтобы $|S_1| = |S_2|$ или $|S_1| = |S_2| + 1$, то есть число точек в каждом наборе должно быть сбалансировано.
2. Управление («властвование»): если в наборе S имеется только одна точка, то возвращается ∞. Если в наборе S содержатся только две точки, то возвращается евклидово расстояние между ними.
3. Объединение: пусть d_1 и d_2 – наименьшие расстояния в наборах S_1 и S_2 соответственно. Пусть d_3 – наименьшее расстояние между всеми парами точек (p_1, p_2), где p_1 – точка в наборе S_1, а p_2 – точка в наборе S_2. Тогда наименьшее расстояние равно $\min(d_1, d_2, d_3)$, то есть ответ может находиться в меньшем подмножестве точек S_1, или в подмножестве S_2, или одна точка в наборе S_1, а вторая точка в наборе S_2, с пересечением линии $x = d$.

Шаг объединения (третий шаг) при простейшем способе выполнения оставляет неизменной сложность $O(n^2)$. Но его можно оптимизировать. Пусть $d' = \min(d_1, d_2)$. Для каждой точки слева от разделяющей линии $x = d$ самая близкая точка, лежащая справа от разделяющей линии, может располагаться только внутри прямоугольника с шириной d' и высотой $2 \times d'$. Можно доказать (здесь доказательство не приводится), что внутри этого прямоугольника может находиться не более шести таких точек. Это означает, что на шаге объединения потребуется всего лишь время выполнения $O(6n)$, а общая временна́я сложность решения по методу «разделяй и властвуй» будет равна $T(n) = 2 \times T(n/2) + O(n)$, то есть $O(n \times \log n)$.

Упражнение 9.6.1*. Существует и более простое решение, нежели описанный выше метод «разделяй и властвуй». Используется алгоритм сканирующей прямой (sweep line algorithm). Точки в наборе S сканируются слева направо. Предположим, что текущий наилучший найденный ответ равен d, и теперь мы исследуем точку i. Предполагаемая новая точка, наиболее близкая к i, если такая существует, обязательно должна иметь координату y в пределах d единиц от точки i. Проверяются все такие точки-кандидаты, и соответственно обновляется значение d (которое будет постепенно уменьшаться). Выполнить реализацию этого решения и проанализировать его временну́ю сложность.

Задания по программированию, связанные с задачей о паре ближайших точек

1. **UVa 10245 – The Closest Pair Problem *** (классическая задача, описанная выше)
2. **UVa 11378 – Bey Battle *** (это тоже задача о паре ближайших точек)

9.7. Алгоритм Диница

В разделе 4.6 рассматривался алгоритм Форда–Фалкерсона (Ford-Fulkerson) с потенциально непредсказуемой сложностью $O(|f^*|E)$ и более предпочтительный алгоритм Эдмондса–Карпа (Edmonds-Karp) со сложностью $O(VE^2)$ (поиск увеличивающего пути с использованием метода поиска в ширину (BFS)) для решения задачи о максимальном потоке (Max Flow problem). По данным, собранным в 2013 году, большинство (если не все) задач о максимальном потоке, рассматриваемых в этой книге, разрешимы с помощью алгоритма Эдмондса–Карпа.

Существует несколько других алгоритмов нахождения максимального потока, которые теоретически улучшают производительность по сравнению с алгоритмом Эдмондса–Карпа. Одним из них является алгоритм Диница (Dinic's algorithm) со сложностью $O(V^2E)$. Поскольку в типичном потоковом графе обычно $V < E$ и $E << V^2$, даже самый наихудший случай временно́й сложности алгоритма Диница теоретически лучше, чем для алгоритма Эдмондса–Карпа. Несмотря на то что авторы этой книги никогда не встречали случай, при котором применение алгоритма Эдмондса–Карпа приводило к превышению лимита времени (TLE), а применение алгоритма Диница принималось как верное решение (AC) на одном и том же потоковом графе, тем не менее, возможно, использование алгоритма Диница более предпочтительно на олимпиадах по программированию, хотя бы из соображений надежности.

Алгоритм Диница использует ту же основную идею, что и алгоритм Эдмондса–Карпа, то есть выполняет итеративный поиск увеличивающего пути. Но алгоритм Диница применяет концепцию «блокирующих потоков» для поиска увеличивающих путей. Понимание этой концепции является самым главным условием для расширения более простого алгоритма Эдмондса–Карпа до алгоритма Диница.

Определим `dist[v]` как длину наикратчайшего пути от исходной вершины s до вершины v в остаточной сети. Тогда назовем слоистой сетью такую сеть L, что ребро (u, v) остаточной сети включается в слоистую сеть L, если `dist[v] = dist[u] + 1`. В таком случае «блокирующий поток» – это $s - t$ поток f, такой, что после передачи через поток f от s до t слоистая сеть L уже не содержит увеличивающего пути $s - t$.

Было доказано (см. [11]), что расстояние (по количеству ребер) в каждом блокирующем потоке возрастает как минимум на единицу при каждой итерации. В этом алгоритме может существовать не более $V - 1$ блокирующих потоков, поскольку не может быть больше максимума $V - 1$ ребер вдоль «самого длинного» простого пути от s до t. Слоистую сеть можно сформировать при помощи метода поиска в ширину (BFS) за время $O(E)$, а блокирующий поток в каждой слоистой сети может быть найден за время $O(VE)$. Таким образом, наихудшим вариантом временно́й сложности для алгоритма Диница является $O(V \times (E + VE)) = O(V^2E)$.

Реализацию алгоритма Диница можно упростить по аналогии с реализацией алгоритма Эдмондса–Карпа, показанной в разделе 4.6. По алгоритму Эдмондса–Карпа выполняется поиск в ширину, который уже генерирует вспомогательную слоистую сеть L, но он используется только лишь для поиска одного увеличивающего пути при вызове функции `augment(t, INF)`. В алгоритме Дини-

ца необходимо использовать информацию, сгенерированную методом поиска в ширину, несколько иным способом. Главное изменение: вместо поиска блокирующего потока методом поиска в глубину (DFS) в слоистой сети L можно имитировать этот процесс, выполняя процедуру `augment` из каждой вершины v, которая напрямую соединена с вершиной-стоком t в графе уровня, например ребро (v,t) существует в графе уровня L, и мы вызываем процедуру `augment(v, INF)`. Это позволит найти многие (но не всегда все) требуемые увеличивающие пути, которые формируют блокирующий поток в слоистой сети L.

Упражнение 9.7.1*. Реализовать вариант алгоритма Диница, взяв за основу код алгоритма Эдмондса–Карпа, приведенный в разделе 4.6, с использованием главного изменения, предложенного выше. Также использовать модифицированный список смежности из **упражнения 4.6.3.3***. После этого с учетом внесенных изменений заново решить различные упражнения по программированию, предложенные в разделе 4.6. Наблюдаются ли какие-либо улучшения во время выполнения?

Упражнение 9.7.2*. Используя структуру данных под названием динамические деревья (dynamic trees), можно сократить временну́ю сложность поиска блокирующего потока с $O(VE)$ до $O(E \times \log V)$, следовательно, общая наихудшая временна́я сложность алгоритма Диница становится равной $O(VE \log V)$. Исследовать и реализовать этот вариант алгоритма Диница.

Упражнение 9.7.3*. Что происходит, если алгоритм Диница используется в потоковом графе, моделирующем задачу паросочетания максимальной мощности в двудольном графе (МСВМ), рассматриваемую в разделе 4.7.4? (Совет: см. раздел 9.12.)

9.8. Формулы или теоремы

На предыдущих олимпиадах по программированию встречались задачи с редко используемыми формулами или теоремами. Знание методов решения таких задач дает не вполне заслуженное преимущество над другими участниками, если одна из подобных редких формул или теорем используется на олимпиаде по программированию, в которой вы участвуете.

1. Формула (теорема) Кэли (Cayley's Formula) о числе деревьев: существует n^{n-2} остовных деревьев (spanning trees) в полном графе с n пронумерованными вершинами. Пример: UVa 10843 – Anne's game.
2. Беспорядок, или перестановка без неподвижных точек (derangement): перестановка элементов множества таким способом, чтобы ни один из элементов не появлялся на своей первоначальной позиции. Количество перестановок без неподвижных точек der(n) может быть вычислено следующим образом: der(n) = (n – 1) × (der(n – 1) + der(n – 2)), где der(0) = 1, а der(1) = 0. Простая задача о перестановках без неподвижных точек: UVa 12024 – Hats (см. раздел 5.6).

3. Теорема Эрдёша–Галлаи (Erdős–Gallai theorem) определяет необходимое и достаточное условие для конечной последовательности натуральных чисел, которую можно сопоставить последовательности степеней вершин некоторого простого графа. Последовательность неотрицательных целых чисел $d_1 \geqslant d_2 \geqslant \ldots \geqslant d_n$ может быть последовательностью степеней простого графа с n вершинами тогда и только тогда, когда сумма $\sum_{i=1}^{n} d_i$ четна, а неравенство $\sum_{i=1}^{k} d_i \leqslant k \times (k-1) + \sum_{i=k+1}^{n} \min(d_i, k)$ справедливо для $1 \leqslant k \leqslant n$. Пример: UVa 10720 – Graph Construction.
4. Формула Эйлера для планарного графа[1]: $V - E + F = 2$, где F – число граней[2] этого планарного графа. Пример: UVa 10178 – Count the Faces.
5. Задача Мозера о разделении круга (Moser's Circle): определение числа частей, на которые можно разделить круг, если n точек на его окружности соединены хордами, среди которых нет трех прямых, пересекающихся в одной точке. Решение: $g(n) = C_n^4 + C_n^2 + 1$. Пример: UVa 10213 – How Many Pieces of Land?
6. Теорема (формула) Пика (Pick's Theorem)[3]: пусть i – количество целочисленных точек внутри многоугольника, A – площадь этого многоугольника, b – количество целочисленных точек на границе того же многоугольника, тогда $A = i + b/2 - 1$. Пример: UVa 10088 – Trees on My Island.
7. Количество остовных деревьев в полном двудольном графе $K_{n,m}$ равно $m^{n-1} \times n^{m-1}$. Пример: UVa 11719 – Gridlands Airport.

Задания по программированию, связанные с редко используемыми формулами и теоремами

1. UVa 10088 – Trees on My Island (теорема (формула) Пика)
2. UVa 10178 – Count the Faces (формула Эйлера, поиск нескольких объединений)
3. **UVa 10213 – How Many Pieses… *** (задача Мозера о разделении круга; формулу трудно вывести; $g(n) = C_n^4 + C_n^2 + 1$)
4. **UVa 10720 – Graph Construction *** (теорема Эрдёша–Галлаи)
5. UVa 10843 – Anna's game (формула (теорема) Кэли для подсчета количества остовных деревьев в графе с n вершинами, равного n^{n-2}; использовать Java BigInteger)
6. UVa 11414 – Dreams (аналогично заданию UVa 10720; теорема Эрдёша–Галлаи)
7. **UVa 11719 – Gridlands Airports *** (подсчет количества остовных деревьев в полном двудольном графе; использовать Java BigInteger)

---

[1] Граф, который может быть изображен в двумерном евклидовом пространстве (на плоскости) без каких-либо пересечений ребер во внутренней точке какого-то из ребер.

[2] При изображении планарного графа без каких-либо его ребер любой замкнутый контур, ограничивающий область, в которой нет других ребер, пересекающих этот контур, образует грань (face).

[3] Вывел и доказал Георг Александр Пик (Georg Alexander Pick) в 1899 году.

9.9. Алгоритм последовательного исключения переменных, или метод Гаусса

Описание задачи

Линейное уравнение определяется как уравнение, в котором порядок неизвестных (переменных) является линейным (то есть представляет собой совокупность постоянных величин или произведение постоянных величин и первой степени неизвестной переменной). Например, уравнение $X + Y = 2$ является линейным, а уравнение $X^2 = 4$ нелинейное.

Система линейных уравнений определяется как набор из n неизвестных (переменных) в (как правило) n линейных уравнениях, например $X + Y = 2$ и $2X + 5Y = 6$. Решение этой системы: $X = 1⅓$, $Y = ⅔$. Следует отметить отличие от линейного диофантова уравнения (см. раздел 5.5.9), так как решение для системы линейных уравнений может быть нецелым.

В редких случаях в задании олимпиады по программированию может встретиться подобная система линейных уравнений. Знание решения, особенно его реализации, может оказаться полезным.

Решения

Для решения системы линейных уравнений можно воспользоваться одним из хорошо известных методов, таким как алгоритм последовательного исключения переменных, предложенный Гауссом (Gaussian Elimination algorithm). Этот алгоритм достаточно часто описывается в инженерно-технической литературе в разделе «Численные методы». Некоторые книги по информатике также уделяют внимание описанию этого алгоритма, например [8]. Здесь мы рассмотрим реализацию этого относительно простого алгоритма со сложностью $O(n^3)$ с использованием функции на языке C++.

```
#define MAX_N 100     // определите это значение в соответствии с собственными требованиями

struct AugmentedMatrix { double mat[MAX_N][MAX_N+1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination( int N, AugmentedMatrix Aug )                    // O(N^3)
{
  // ввод: N, расширенная матрица Aug, вывод: вектор-столбец X - решение
  int i, j, k, l; double t; ColumnVector X;

  for( j = 0; j < N-1; j++ ) {            // первый этап - прямой ход исключения переменных
    l = j;
    for( i = j+1; i < N; i++ )        // определение строки, содержащей наибольшее значение
                                                                        // в текущем столбце
      if( fabs( Aug.mat[i][j] ) > fabs( Aug.mat[l][j] ) )
        l = i;                                                  // запомнить эту строку l
    // перестановка строк: строку l в самую верхнюю позицию
    // причина: минимизация ошибок (погрешностей) при операциях с плавающей точкой
    for( k = j; k <= N; k++ )
      // t - временная переменная типа double
      t = Aug.mat[j][k], Aug.mat[j][k] = Aug.mat[l][k], Aug.mat[l][k] = t;
```

```
    for( i = j+1; i < N; i++ )                    // реальное выполнение этапа прямого хода
                                                                  // исключения переменных
      for( k = N; k >= j; k-- )
        Aug.mat[i][k] -= Aug.mat[j][k] * Aug.mat[i][j] / Aug.mat[j][j];
  }
  for( j = N-1; j >= 0; j-- ) {   // этап обратного хода - подстановка вычисленных значений
    for( t = 0.0, k = j+1; k < N; k++ ) t += Aug.mat[j][k] * X.vec[k];
    X.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j];      // в векторе X формируется решение
  }
  return X;
}
```

Файл исходного кода: *GaussianElimination.cpp/java*

Пример выполнения алгоритма

В этом подразделе демонстрируется пошаговое выполнение алгоритма Гаусса для последовательного исключения переменных в системе линейных уравнений на следующем примере. Предположим, что имеется система линейных уравнений:

```
 X = 9 - Y - 2Z
2X + 4Y = 1 + 3Z
 3X - 5Z = -6Y
```

В первую очередь необходимо преобразовать эту систему уравнений в основную форму, то есть изменить порядок записи неизвестных переменных, разместив одинаковые переменные во всех уравнениях слева направо. После преобразования получаем:

```
1X + 1Y + 2Z = 9
2X + 4Y - 3Z = 1
3X + 6Y - 5Z = 0
```

Затем перепишем эту систему линейных уравнений в форме умножения матриц: $A \times x = b$. Этот прием также используется в разделе 9.21. Получаем:

$$\begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 9 \\ 1 \\ 0 \end{bmatrix}.$$

Далее будем работать с матрицей A (размер $N \times N$) и с вектором-столбцом b (размер $N \times 1$). Объединим их в расширенную матрицу (augmented matrix) с размером $N \times (N+1)$ (последний столбец с тремя стрелками – это комментарий для более понятного объяснения):

$$\left[\begin{array}{ccc|c} 1 & 1 & 2 & 9 \\ 2 & 4 & -3 & 1 \\ 3 & 6 & -5 & 0 \end{array}\right] \begin{array}{l} \rightarrow 1X + 1Y + 2Z = 9 \\ \rightarrow 2X + 4Y - 3Z = 1 \\ \rightarrow 3X + 6Y - 5Z = 0 \end{array}.$$

Затем передаем эту расширенную матрицу в функцию `GaussianElimination`, представленную в предыдущем разделе. Первый этап – прямой ход исключения переменных. Выбирается наибольшее абсолютное значение в столбце $j = 0$ из строки $i = 0$ при последовательном сравнении значений по строкам, затем строка с найденным наибольшим значением меняется местами со строкой $i = 0$. Этот (дополнительный) шаг необходим только для минимизации погрешности (ошибки) при выполнении операций с плавающей точкой. В рассматриваемом здесь примере после перестановки строк 0 и 2 получаем:

$$\left[\begin{array}{ccc|c|l} \underline{3} & 6 & -5 & 0 & \rightarrow \underline{3X + 6Y - 5Z = 0} \\ 2 & 4 & -3 & 1 & \rightarrow 2X + 4Y - 3Z = 1 \\ \underline{1} & 1 & 2 & 9 & \rightarrow \underline{1X + 1Y + 2Z = 9} \end{array}\right].$$

Основной операцией алгоритма Гаусса на этом этапе прямого хода исключения переменных является исключение переменной X (первой переменной) из всех последующих строк $i + 1$. В рассматриваемом примере переменная X исключается из строк 1 и 2. Обратите особое внимание на комментарий «реальное выполнение этапа прямого хода исключения переменных» в коде функции `GaussianElimination` из предыдущего раздела. Теперь получаем:

$$\left[\begin{array}{ccc|c|l} 3 & 6 & -5 & 0 & \rightarrow 3X + 6Y - 5Z = 0 \\ 0 & 0 & 0.33 & 1 & \rightarrow \underline{0X + 0Y + 0.33Z = 1} \\ 0 & -1 & 3.67 & 9 & \rightarrow \underline{0X - 1Y + 3.67Z = 9} \end{array}\right].$$

Далее исключаем следующую переменную (теперь это переменная Y). Последовательным сравнением выбираем наибольшее абсолютное значение в столбце $j = 1$ из строки row = 1, затем меняем местами эту строку со строкой $i = 1$. В рассматриваемом примере после перестановки строк 1 и 2 получаем следующую расширенную матрицу, в которой оказывается, что переменная Y уже исключена из строки 2:

$$\left[\begin{array}{l|ccc|c|l} \text{строка } 0 & 3 & 6 & -5 & 0 & \rightarrow 3X + 6Y - 5Z = 0 \\ \text{строка } 1 & 0 & \underline{-1} & 3.67 & 9 & \rightarrow \underline{0X - 1Y + 3.67Z = 9} \\ \text{строка } 2 & 0 & \underline{0} & 0.33 & 1 & \rightarrow \underline{0X + 0Y + 0.33Z = 1} \end{array}\right].$$

После того как получена нижняя треугольная часть расширенной матрицы, содержащая только нулевые значения, можно начать второй этап: обратный ход с подстановкой вычисленных значений переменных. Обратите внимание на несколько последних строк в исходном коде функции `GaussianElimination`: после исключения переменных X и Y осталась только переменная Z в строке 2. Теперь можно с уверенностью сказать, что $Z = 1/0.33 = 3$.

$$\left[\text{строка } 2 \left| 0 \quad 0 \quad 0.33 \right| 1 \right| \rightarrow 0X + 0Y + 0.33Z = 1 \rightarrow Z = 1/0.33 = 3\Big].$$

После получения значения $Z = 3$ можно начать обработку строки 1: $Y = (9 - 3.67 * 3) / -1 = 2$.

$$\left[\text{строка } 1 \,\middle|\, 0 \quad -1 \quad 3.67 \,\middle|\, 9 \,\middle|\, \rightarrow 0X - 1Y + 3.67Z = 9 \rightarrow Y = (9 - 3.67 * 3)/-1 = 2\right].$$

Наконец, после получения значений $Z = 3$ и $Y = 2$ можно перейти к вычислению строки 0: $X = (0 - 6 * 2 + 5 * 3) / 3 = 1$. Система уравнений решена.

$$\left[\text{строка } 0 \,\middle|\, 3 \quad 6 \quad -5 \,\middle|\, 0 \,\middle|\, \rightarrow 3X + 6Y - 5Z = 0 \rightarrow X = (0 - 6 * 3 + 5 * 3)/3 = 1\right].$$

Таким образом, решением рассматриваемой здесь системы линейных уравнений являются значения $X = 1$, $Y = 2$ и $Z = 3$.

Задания по программированию, связанные с алгоритмом Гаусса (последовательное исключение переменных)

1. **UVa 11319 – Stupid Sequence? *** (решить систему первых семи линейных уравнений; затем использовать все 1500 уравнений для выполнения проверок «осмысленной последовательности» (smart sequence))

9.10. Паросочетание в графах

Описание задачи

Паросочетание в графах: выбор такого подмножества ребер M в графе $G(V, E)$, что любые два ребра из этого подмножества не имеют общей вершины. Нас главным образом интересует паросочетание максимальной мощности (maximum cardinality matching), то есть необходимо знать максимальное количество ребер, которое можно определить как паросочетание в графе G. Еще одна широко известная задача – определение совершенного паросочетания (perfect matching), при котором имеется паросочетание максимальной мощности, в котором участвуют все вершины графа (ни одна вершина не пропущена). Отметим, что если V нечетно, то совершенное паросочетание не существует. Задачу нахождения совершенного паросочетания можно решить простым нахождением паросочетания максимальной мощности и последующей проверкой участия в нем всех вершин графа.

В задачах о паросочетаниях в графах, предлагаемых на олимпиадах по программированию, существуют два важных атрибута, которые могут (существенно) изменить уровень сложности: является ли исходный граф двудольным (bipartite) (если не является, то задача усложняется) и является ли исходный граф невзвешенным (если граф взвешенный, то задача усложняется). Эти две характеристики создают четыре варианта[1], описанных ниже и графически изображенных на рис. 9.4.

[1] Существуют и другие варианты задач о паросочетании в графах, помимо этих четырех, например задача о стабильных браках (о стабильном соответствии – Stable Marriage problem). Но в этом разделе все внимание сосредоточено на этих четырех вариантах.

1. Паросочетание максимальной мощности в невзвешенном двудольном графе (Unweighted MCBM). Это самый простой и наиболее часто встречающийся вариант.
2. Паросочетание максимальной мощности во взвешенном двудольном графе (Weighted MCBM). Эта задача похожа на предыдущую, но здесь ребра в графе *G* имеют веса. Обычно требуется найти паросочетание максимальной мощности в двудольном графе (MCBM) с минимальным общим весом.
3. Паросочетание максимальной мощности в невзвешенном графе (Unweighted MCM). В этом варианте граф не обязательно является двудольным.
4. Паросочетание максимальной мощности во взвешенном графе (Weighted MCM). Это самый трудный вариант.

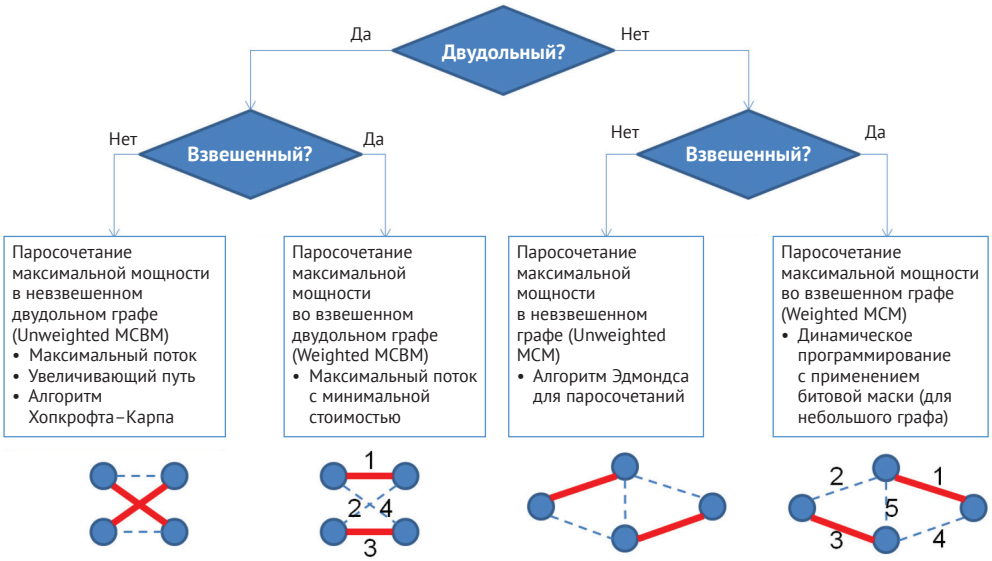


Рис. 9.4 ❖ Четыре самых часто встречающихся варианта задачи о паросочетаниях в графах на олимпиадах по программированию

Решения

Решения задачи нахождения паросочетания максимальной мощности в невзвешенном двудольном графе (Unweighted MCBM)

Это самый простой вариант задачи, и некоторые решения уже рассматривались ранее в разделах 4.6 («Поток в сети») и в 4.7.4 («Двудольный граф»). В следующем списке кратко описаны три возможных решения для задач о нахождении паросочетания максимальной мощности в невзвешенном двудольном графе (Unweighted MCBM):

1) приведение задачи о нахождении паросочетания максимальной мощности в невзвешенном двудольном графе (Unweighted MCBM) к задаче

нахождения максимального потока (Max Flow). Более подробно см. разделы 4.6 и 4.7.4. Временна́я сложность зависит от выбранного алгоритма нахождения максимального потока;

2) алгоритм определения увеличивающего пути со сложностью $O(V^2 + VE)$ для задачи Unweighted MCBM. Более подробно см. раздел 4.7.4. Этот способ решения достаточно эффективен для различных олимпиадных заданий, связанных с задачей Unweighted MCBM;
3) алгоритм Хопкрофта–Карпа со сложностью $O(\sqrt{V}E)$ для задачи Unweighted MCBM. Более подробно см. раздел 9.12.

Решения задачи нахождения паросочетания максимальной мощности во взвешенном двудольном графе (Weighted MCBM)

Когда ребра в двудольном графе имеют веса, не все возможные паросочетания максимальной мощности в двудольном графе являются оптимальными. Необходимо выбрать одно (не обязательно уникальное) паросочетание максимальной мощности, которое имеет общий минимальный суммарный вес. Одним из возможных решений этой задачи[1] является приведение задачи Weighted MCBM к задаче нахождения максимального потока с минимальной стоимостью (MCMF) (см. раздел 9.23).

Например, на рис. 9.5 изображен один из тестовых вариантов задания UVa 10746 – Crime Wave – The Sequel. Это задача нахождения паросочетания максимальной мощности в двудольном графе для полного двудольного графа $K_{n,m}$, но каждое ребро этого графа имеет связанную с ним стоимость. Мы суммируем

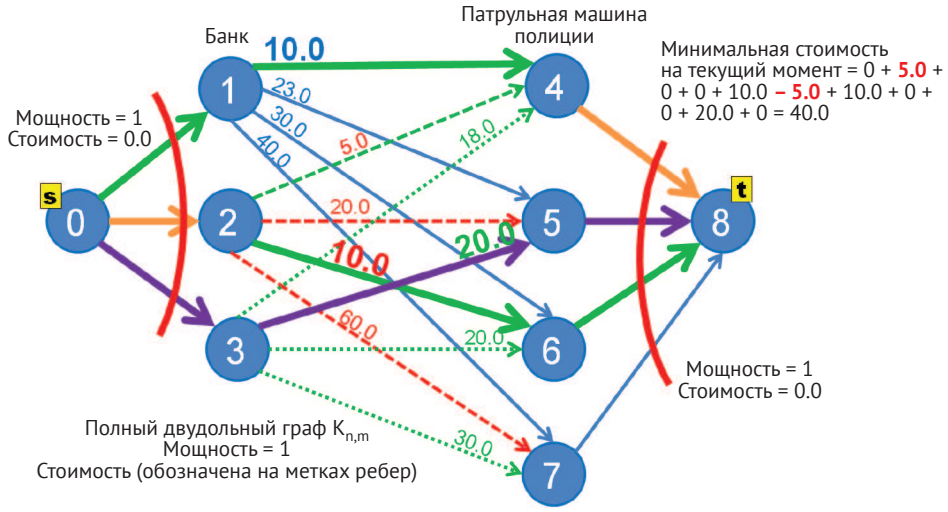


Рис. 9.5 ❖ Пример тестового варианта при решении задания UVa 10746: три паросочетания с минимальной стоимостью = 40

[1] Другим возможным решением, если требуется получить совершенное паросочетание в двудольном графе с минимальной стоимостью, является венгерский алгоритм, он же алгоритм Куна–Мункерса.

ребра, исходящие из источника *s* к вершинам из набора слева с мощностью 1 и стоимостью 0. Мы также суммируем ребра, исходящие из вершин набора справа и направленные в сток *t* тоже с мощностью 1 и стоимостью 0. Ориентированные ребра от левого набора вершин к правому набору имеют мощность 1 и стоимость, определяемую условиями данной конкретной задачи. После завершения формирования этого взвешенного потокового графа можно применить к нему алгоритм нахождения максимального потока с минимальной стоимостью (MCMF), как описано в разделе 9.23, для получения требуемого решения: поток 1 = 0 → 2 → 4 → 8 со стоимостью 5, поток 2 = 0 → 1 → 4 → 2 (отмена потока 2-4) → 6 → 8 со стоимостью 15 и поток 3 = 0 → 3 → 5 → 8 со стоимостью 20. Минимальная общая стоимость равна 5 + 15 + 20 = 40.

Решения задачи нахождения паросочетания максимальной мощности в невзвешенном графе (Unweighted MCM)

Задача нахождения паросочетаний в двудольных графах решается просто, но для графов вообще найти решение трудно. В прошлом исследователи в области информатики считали, что этот вариант представлял собой еще одну *NP*-полную задачу до тех пор, пока Джек Эдмондс (Jack Edmonds) не опубликовал полиномиальный алгоритм ее решения в статье «Paths, trees, and flowers» [13].

Главная проблема состоит в том, что в обобщенном графе могут встречаться увеличивающие циклы нечетной длины. Эдмондс называет такие циклы blossom (цветок). Основной принцип алгоритма Эдмондса заключается в повторяющемся сжатии подобных цветков (поэтому его часто называют алгоритмом сжатия цветков) таким способом, чтобы поиск увеличивающих путей возвращался к простому варианту, как в двудольном графе. Затем алгоритм Эдмондса реорганизовывает паросочетания, когда эти обнаруженные цветки восстанавливаются (поднимаются).

Реализация алгоритма Эдмондса для поиска паросочетаний нетривиальна. Поэтому, чтобы сделать этот вариант задачи о паросочетаниях более управляемым, многие авторы задач ограничивают размер невзвешенных обобщенных графов достаточно малым значением, например число вершин $V \leqslant 18$, так что сложность $O(V \times 2^V)$ позволяет решить эту задачу методом динамического программирования с применением алгоритма битовой маски (см. **упражнение 8.3.1.1**).

Решение задачи нахождения паросочетания максимальной мощности во взвешенном графе (Weighted MCM)

Вероятно, это самый трудно решаемый вариант задачи. Задан обобщенный граф, ребра которого имеют соответствующие веса. В обычном контексте олимпиады по программированию наиболее вероятным решением является метод динамического программирования с применением битовой маски (см. раздел 8.3.1), поскольку авторы задач обычно предлагают в условиях задачи только обобщенные графы небольшого размера.

Визуальное представление паросочетаний в графе

Чтобы помочь читателям лучше понять все перечисленные выше варианты задач о паросочетаниях в графе и их решения, авторы книги создали следую-

щее средство для визуального представления паросочетаний в графах: http://www.comp.nus.edu.sg/~stevenha/visualization/matching.html. С помощью этого инструмента визуализации вы можете нарисовать собственный граф, и система предложит правильно выбранный алгоритм (или несколько алгоритмов) нахождения паросочетаний в графе на основе двух характеристик: является ли исходный граф двудольным и/или взвешенным. Отметим, что визуальное представление алгоритма Эдмондса для поиска паросочетаний в нашем инструментальном средстве, вероятно, является одним из первых в мире.

Упражнение 9.10.1*. Реализовать алгоритм Куна–Манкреса (Kuhn-Munkres) (см. статью-источник [39, 45]).

Упражнение 9.10.2*. Реализовать алгоритм Эдмондса для нахождения паросочетаний в графе (см. статью-источник [13]).

Задания по программированию, связанные с задачей о нахождении паросочетаний в графах

- См. некоторые задачи о назначениях (паросочетания в двудольных графах с учетом мощности) в разделе 4.6
- См. некоторые задачи о нахождении паросочетаний максимальной мощности в невзвешенном двудольном графе (Unweighted MCBM) и их варианты в разделе 4.7.4
- См. некоторые задачи о нахождении паросочетаний максимальной мощности во взвешенном двудольном графе (Weighted MCBM) в разделе 9.23
- Паросочетание максимальной мощности в невзвешенном графе (Unweighted MCM):
  - UVa 11439 – Maximizing the ICPC * (бинарный поиск по ответу для получения минимального веса; этот вес используется для восстановления графа; также использовать алгоритм Эдмондса для проверки возможности получения совершенного паросочетания в обобщенном графе)
- См. задачи о нахождении паросочетаний максимальной мощности в (не)взвешенном графе ((Un)weighted MCM) в разделе 8.3 (динамическое программирование)

9.11. Кратчайшее расстояние на сфере (ортодромия)

Описание задачи

Сфера – это абсолютно круглый геометрический объект в трехмерном пространстве.

Ортодромия (Great-Circle Distance) – кратчайшее расстояние между двумя точками A и B на сфере (в общем случае – на любой поверхности вращения),

то есть кратчайшее расстояние вдоль пути на поверхности сферы. Этот путь представляет собой дугу большой окружности (Great Circle) рассматриваемой сферы, которая проходит через две точки A и B. Большую окружность можно мысленно представить как результат разрезания плоскостью сферы на две равные полусферы (см. рис. 9.6, слева и в середине).

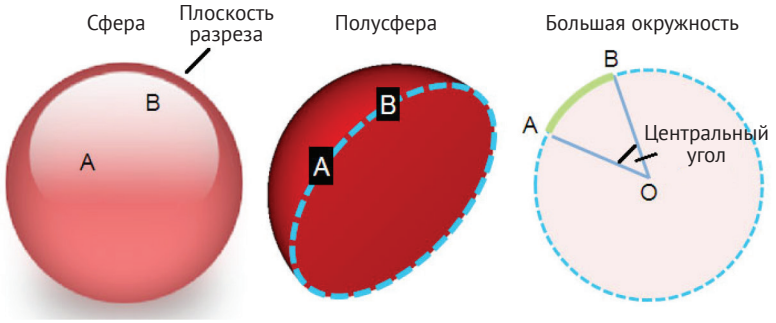


Рис. 9.6 ❖ Слева: сфера, в середине: полусфера и большая окружность, справа: ортодромия gcDistance (дуга A-B)

Решения

Для нахождения кратчайшего расстояния по поверхности сферы между точками A и B необходимо вычислить центральный угол AOB (см. рис. 9.6 – справа), где O – центр большой окружности (кроме того, это центр сферы). Зная радиус сферы / большой окружности, можно определить длину дуги A-B, которая и является искомым кратчайшим расстоянием, то есть ортодромией.

Несмотря на то что в настоящее время подобные задачи встречаются достаточно редко, все же некоторые олимпиадные задания содержат такие понятия, как «Земля», «авиалинии» и т. п., для которых используется ортодромия как мера расстояний. Обычно две точки на поверхности сферы заданы как земные координаты, то есть пара (широта, долгота). Приведенный ниже библиотечный код поможет вычислить кратчайшее расстояние по большой окружности (ортодромию) между двумя заданными точками на сфере и радиус этой сферы. Здесь не учитывается геодезическое отклонение, поскольку для олимпиадного программирования оно не имеет особого значения.

```
double gcDistance( double pLat, double pLong,
                   double qLat, double qLong, double radius ) {
  pLat *= PI / 180; pLong *= PI / 180;                 // преобразование градусов в радианы
  qLat *= PI / 180; qLong *= PI / 180;
  return radius * acos( cos(pLat) * cos(pLOng) * cos(qLat) * cos(qLong) +
                        cos(pLat) * sin(pLong) * cos(qLat) * sin(qLong) +
                        sin(pLat) * sin(qLat) );
}
```

Файл исходного кода: *UVa11817.cpp/java*

Задания по программированию, связанные с задачей о вычислении ортодромии

1. UVa 00535 – Globetrotter (функция gcDistance)
2. UVa 10316 – Airline Hub (функция gcDistance)
3. UVa 10897 – Travelling Distance (функция gcDistance)
4. UVa 11817 – Tunnelling The Earth (функция gcDistance; евклидово расстояние в трехмерном пространстве)

9.12. Алгоритм Хопкрофта–Карпа

Алгоритм Хопкрофта–Карпа (Hopcroft-Karp) [28] – это еще один алгоритм решения задачи о нахождении паросочетаний максимальной мощности в невзвешенном двудольном графе (Unweighted MCBM) на основе решения задачи о максимальном потоке (Max Flow) (требующей большего объема кода) и алгоритма нахождения увеличивающего пути (Augmenting Path) (являющегося более предпочтительным методом), как было описано в разделе 4.7.4.

По мнению авторов, главной причиной использования алгоритма Хопкрофта–Карпа с бóльшим объемом кода вместо более простого и лаконичного (по объему кода) алгоритма нахождения увеличивающего пути для решения задачи Unweighted MCBM является скорость во время выполнения. Алгоритм Хопкрофта–Карпа имеет сложность $O(\sqrt{V}E)$, то есть выполняется (намного) быстрее, чем алгоритм нахождения увеличивающего пути со сложностью $O(VE)$ для двудольных графов среднего размера ($V \approx 500$) (и плотности).

Крайним случаем является полный двудольный граф $K_{n,m}$ с числом вершин $V = n + m$ и числом ребер $E = n \times m$. В таком двудольном графе алгоритм нахождения увеличивающего пути имеет наихудшую временну́ю сложность $O((n + m) \times n \times m)$. Если $m = n$, то получаем решение $O(n^3)$, которое является удовлетворительным только для $n \leqslant 200$.

Главная проблема алгоритма нахождения увеличивающего пути со сложностью $O(VE)$ заключается в том, что он может в первую очередь начать исследование более длинных увеличивающих путей (так как в сущности это «модифицированный поиск в глубину»). Это неэффективный метод. Исследуя в первую очередь самые короткие увеличивающие пути, Хопкрофт и Карп доказали, что их алгоритм будет выполнен всего лишь за $O(\sqrt{V})$ итераций [28]. На каждой итерации алгоритм Хопкрофта–Карпа выполняет поиск в ширину (BFS) $O(E)$ из всех свободных вершин левого набора и находит увеличивающие пути с растущими длинами (начиная с длины 1: свободное ребро, длина 3: свободное ребро, ребро, входящее в паросочетание, и снова свободное ребро, длина 5, длина 7 и т. д.). Затем алгоритм обращается к другой процедуре поиска в глубину (DFS) $O(E)$ для наращивания этих увеличивающих путей (алгоритм Хопкрофта–Карпа способен наращивать более одного паросочетания за одну итерацию). Таким образом, алгоритм Хопкрофта–Карпа имеет временну́ю сложность $O(\sqrt{V}E)$.

Для крайнего случая полного двудольного графа $K_{n,m}$, описанного выше, в наихудшем случае алгоритм Хопкрофта–Карпа имеет сложность $O(\sqrt{(n+m)} \times n \times m)$. Если $n = m$, то получаем решение $O(n^{5/2})$, которое можно считать удовлетворительным для $n \leqslant 600$. Таким образом, если автор задачи «достаточно коварен», для того чтобы задать $n \approx 500$ и относительно плотный двудольный граф для задачи Unweighted MCBM, то применение алгоритма Хопкрофта–Карпа более эффективно и безопасно по сравнению со стандартным алгоритмом нахождения увеличивающего пути (тем не менее см. **упражнение 4.7.4.3***, в котором применяется прием, обеспечивающий выполнение алгоритма нахождения увеличивающего пути «достаточно быстро», даже если в исходных данных задан относительно плотный и крупный двудольный граф).

Упражнение 9.12.1*. Реализовать алгоритм Хопкрофта–Карпа, начав с применения алгоритма нахождения увеличивающего пути, описанного в разделе 4.7.4, с использованием идеи, описанной в этом разделе.

Упражнение 9.12.2*. Исследовать похожие свойства и различия в алгоритмах Хопкрофта–Карпа и Диница (см. раздел 9.7).

9.13. Вершинно- и реберно-непересекающиеся пути

Описание задачи

Два пути, которые начинаются из вершины-источника s и заканчиваются в вершине-стоке t, называются независимыми (вершинно-непересекающимися – vertex-disjoint), если они не используют совместно какие-либо вершины, кроме s и t.

Два пути, которые начинаются из вершины-источника s и заканчиваются в вершине-стоке t, называются реберно-непересекающимися (edge-disjoint), если они не используют совместно какое-либо ребро (но могут иметь общие вершины, помимо s и t).

В заданном графе G необходимо найти максимальное число вершинно- и реберно-непересекающимися путей от источника s до стока t.

Решения

Задача поиска (максимального количества) независимых путей от источника s до стока t может быть сведена к задаче поиска (максимального) потока (Network (Max) Flow). В графе G формируется поток $N = (V, E)$ с пропускными способностями вершин, где N – точная копия графа G, за исключением того, что пропускная способность каждой вершины $v \in V$ равна 1 (то есть каждую вершину можно использовать только один раз; методику работы с пропускными способностями вершин см. в разделе 4.6), а пропускная способность каждого ребра $e \in E$ также равна 1 (то есть каждое ребро тоже можно использовать только один раз). Затем выполняется алгоритм Эдмондса–Карпа обычным образом.

Рис. 9.7 ❖ Сравнение максимального числа независимых путей и максимального числа реберно непересекающихся путей

Задача поиска (максимального количества) реберно-непересекающихся путей от источника s до стока t похожа на задачу поиска (максимального количества) независимых путей. Единственное различие заключается в том, что здесь не нужны пропускные способности вершин, то есть подразумевается, что два реберно-непересекающихся пути могут использовать одну и ту же вершину. На рис. 9.7 показано сравнение максимального числа независимых путей и максимального числа реберно-непересекающихся путей от источника $s = 0$ до стока $t = 1$.

Задания по программированию, связанные с задачей поиска независимых и реберно-непересекающихся путей

1. **UVa 00563 – Crimewave *** (проверка равенства максимального числа независимых путей в потоковом графе – с учетом мощности отдельных ребер и отдельных вершин – количеству банков b; анализ нижней границы полученного решения, чтобы определить, является ли стандартное решение поиска максимального потока достаточным (удовлетворяющим условию) даже для наибольшего тестового варианта)
2. **UVa 01242 – Necklace *** (LA 4271, Hefei08, для заданного по условию ожерелья необходимо найти два возможных реберно-непересекающихся потока s-t)

9.14. Количество инверсий

Описание задачи

Задача о количестве инверсий определяется следующим образом: задан список чисел, определить (вычислить) минимальное количество перестановок при пузырьковой сортировке (то есть перестановок в парах смежных чисел в процессе сортировки пузырьком), которое необходимо, чтобы привести список в отсортированный (обычно по возрастанию) порядок.

Например, если список имеет вид {3, 2, 1, 4}, то потребуется три перестановки при пузырьковой сортировке, чтобы упорядочить этот список по воз-

растанию: перестановка (3, 2), чтобы получить {2, 3, 1, 4}, перестановка (3, 1), чтобы получить {2, 1, 3, 4}, и завершающая перестановка (2, 1) для получения конечного результата {1, 2, 3, 4}.

Решения

Решение $O(n^2)$

Самое очевидное решение – подсчет количества перестановок, требуемых непосредственно во время выполнения алгоритма пузырьковой сортировки $O(n^2)$.

Решение $O(n \times \log n)$

Более эффективное решение «разделяй и властвуй» $O(n \times \log n)$ для этой задачи инверсии индекса представляет собой модификацию алгоритма сортировки слиянием (merge sort). Во время выполнения процесса слияния в этом алгоритме сортировки если начальный элемент правого отсортированного подмассива меньше, чем начальный элемент левого отсортированного подмассива, то мы говорим, что «имеет место инверсия». Добавляется счетчик инверсий индекса по размеру текущего левого подмассива. После завершения сортировки слиянием выводится значение этого счетчика. Поскольку добавляются только шаги $O(1)$ при сортировке слиянием, это решение имеет ту же временну́ю сложность, что и сортировка слиянием, то есть $O(n \times \log n)$.

В рассмотренном выше примере предлагался исходный список {3, 2, 1, 4}. При сортировке слиянием этот список разделяется на два подмассива {3, 2} и {1, 4}. Для левого подмассива потребуется одна перестановка (инверсия), так как необходимо поменять местами 3 и 2, чтобы получить {2, 3}. Правый подмассив инверсий не содержит, поскольку он уже отсортирован. Далее выполняется слияние подмассивов, то есть {2, 3} объединяется с {1, 4}. В первую очередь берется число 1 из начала правого подмассива. Есть еще две инверсии, так как в левом подмассиве содержатся два элемента {2, 3}, которые необходимо поменять местами с 1. После этого инверсий больше нет. Следовательно, в этом примере есть всего три инверсии.

Задания по программированию, связанные с задачей инверсии индекса

1. UVa 00299 – Train Swapping (возможно решение методом пузырьковой сортировки $O(n^2)$)
2. **UVa 00612 – DNA Sorting *** (требуется использование `stable_sort` $O(n^2)$)
3. **UVa 10327 – Flip Sort *** (возможно решение методом пузырьковой сортировки $O(n^2)$)
4. UVa 10810 – Ultra Quicksort (требуется применение метода сортировки слиянием $O(n \times \log n)$)
5. UVa 11495 – Bubbles and Buckets (требуется применение метода сортировки слиянием $O(n \times \log n)$)
6. **UVa 11858 – Frosh Week *** (требуется применение метода сортировки слиянием $O(n \times \log n)$; используется 64-битовое целое число)

9.15. Задача Иосифа Флавия

Описание задачи

Задача Иосифа Флавия – это классическая историческая задача, в условиях которой определено, что изначально n людей, пронумерованных от 1 до n, становятся в круг. Каждый k-й человек удаляется из круга (в первоначальной версии в круг становились воины Иосифа Флавия, и они убивали каждого k-го). Процесс счета и удаления повторяется до тех пор, пока не останется только один человек, который считается «выжившим» (исторические источники сообщают, что этим выжившим оказался сам Иосиф Флавий).

Решения

Полный перебор для случаев с небольшим числом участников

Частные варианты задачи Иосифа Флавия с небольшим n разрешимы методом полного поиска (см. раздел 3.2), то есть простой имитацией процесса с помощью циклического массива (или замкнутого связного списка). Варианты задачи Иосифа Флавия с большим количеством участников требуют более эффективных методов решения.

Особый случай при k = 2

Существует элегантный метод определения позиции последнего выжившего человека для $k = 2$ с использованием двоичного представления числа n. Если бинарное представление числа n записать в виде $n = 1b_1b_2b_3...b_n$, то решением будет двоичное число $b_1b_2b_3...b_n1$, то есть мы перемещаем самый старший значимый бит числа n в конец, чтобы сделать его самым младшим значимым битом. При таком подходе задача Иосифа Флавия для $k = 2$ может быть решена с применением $O(1)$ операций.

Обобщенный случай

Пусть $F(n, k)$ обозначает позицию выжившего человека для круга с размером n и правилом исключения k при нумерации участников 0, 1, ..., $n - 1$ (потом будет прибавлено значение 1, чтобы получить окончательный ответ для соответствия формату исходной задачи, описанной выше). После исключения («уничтожения») k-го человека размер круга сокращается на единицу до размера $n - 1$, а позиция выжившего теперь определяется как $F(n - 1, k)$. Это отношение можно записать как равенство $F(n, k) = (F(n - 1, k) + k) \% n$. В простейшем случае при $n = 1$ имеем $F(1, k) = 0$. Такая рекуррентная последовательность имеет сложность по времени $O(n)$.

Другие варианты

Задача Иосифа Флавия известна и в нескольких других вариантах, описывать которые в данной книге не имеет смысла.

Задания по программированию, связанные с задачей Иосифа Флавия

1. UVa 00130 – Roman Roulette (исходная задача Иосифа Флавия)
2. UVa 00133 – The Dole Queue (перебор, аналогично заданию UVa 130)
3. UVa 00151 – Power Crisis (исходная задача Иосифа Флавия)
4. UVa 00305 – Joseph (ответ может быть предпросчитан)
5. UVa 00402 – M*A*S*H (модифицированная задача Иосифа Флавия, моделирование)
6. UVa 00440 – Eeny Meeny Moo (перебор, аналогично заданию UVa 151)
7. UVa 10015 – Joseph's Cousin (модифицированная задача Иосифа Флавия, динамическое значение k, вариант задания UVa 305)
8. **UVa 10771 – Barbarian tribes *** (перебор, заданный входной размер невелик)
9. **UVa 10774 – Repeated Josephus *** (вариант задачи Иосифа Флавия с повторением при $k = 2$)
10. **UVa 11351 – Last Man Standing *** (используется обобщенный вариант рекуррентности задачи Иосифа Флавия)

9.16. Ход коня

Описание задачи

В шахматах конь может ходить только по замысловатой траектории, напоминающей латинскую букву L (или русскую букву Г). Формально конь может перейти с клетки (r_1, c_1) на другую клетку (r_2, c_2) на шахматной доске размером $n \times n$ тогда и только тогда, когда $(r_1 - r_2)^2 + (c_1 - c_2)^2 = 5$. Общий запрос: определить длину наикратчайшего пути при перемещении коня с начальной клетки на целевую клетку. На одной и той же шахматной доске может быть много таких заданий.

Решения

Одна операция поиска в ширину для каждого запроса

Если размер шахматной доски невелик, то можно запустить выполнение одной операции поиска в ширину (BFS) для каждого задания. Для каждого запроса выполняется поиск в ширину из начальной клетки. Каждая клетка может быть связана не более чем с восемью соседними клетками (некоторые клетки, расположенные вдоль края доски, связаны с меньшим числом соседних клеток). Поиск в ширину останавливается, как только найдена целевая клетка. При решении этой задачи поиска кратчайшего пути можно использовать метод поиска в ширину, так как граф невзвешенный (см. раздел 4.4.2). Поскольку на шахматной доске имеется $O(n^2)$ клеток, общая временна́я сложность равна $O(n^2 + 8n^2) = O(n^2)$ на одно задание (запрос) или $O(Qn^2)$ при Q заданиях (запросах).

Один предварительный запуск поиска в ширину и обработка особых случаев

Приведенное выше решение не является самым эффективным способом решения задачи о ходе коня. Если заданная шахматная доска имеет достаточно большой размер и предложено несколько заданий (запросов), например $n = 1000$ и $Q = 16$ в задании UVa 11643 – Knight Tour, то описанный выше подход приведет к превышению лимита времени (TLE).

Более эффективному решению способствует полное понимание того, что если шахматная доска достаточно велика и мы выбираем две произвольные клетки (r_a, c_a) и (r_b, c_b) в середине доски с кратчайшим маршрутом коня между ними, состоящим из d ходов, то смещение позиций клеток на постоянную величину не изменяет итоговое решение (ответ), то есть кратчайший маршрут коня из клетки $(r_a + k, c_a + k)$ в клетку $(r_b + k, c_b + k)$ также содержит d ходов при постоянном значении k.

Таким образом, можно выполнить только одну операцию поиска в ширину (BFS) из произвольно выбранной исходной клетки, а затем вносить некоторые поправки для получения окончательного ответа (решения). Но существует несколько особых (без преувеличения) случаев с угловыми клетками, требующих специальной обработки. Обнаружение этих особых случаев может стать существенной проблемой и привести ко множеству неверных ответов, если участник олимпиады незнаком с ними. Чтобы сделать данный раздел более интересным, мы оставим этот важнейший последний этап без описания, предлагая его в качестве упражнения повышенной сложности, отмеченного звездочкой. После того как вы получите ответ (решение) этой задачи, попробуйте решить задание UVa 11643.

Упражнение 9.16.1*. Найти описанные выше особые случаи и исследовать (решить) их. Советы:

1) рассмотреть отдельно случаи $3 \leqslant n \leqslant 4$ и $n \geqslant 5$;
2) особое внимание уделите угловым клеткам и клеткам вдоль границ доски;
3) что происходит, если начальная клетка и целевая клетка расположены слишком близко друг к другу?

Задания по программированию, связанные с задачей о ходе коня

1. **UVa 00439 – Knight Moves *** (одной операции поиска в ширину на один запрос достаточно)
2. **UVa 11643 – Knight Tour *** (расстояние между любыми двумя интересующими нас позициями может быть получено с использованием предварительно вычисленной таблицы поиска в ширину (плюс обработка особых угловых случаев); по существу, это классическая задача коммивояжера (TSP), см. раздел 3.5.2)

9.17. Алгоритм Косараджу

Поиск компонент сильной связности (Strongly Connected Components – SCC) в ориентированном графе – это классическая задача теории графов, рассматриваемая в разделе 4.2.9. Мы уже разбирали усовершенствованный поиск в глубину (DFS) , называемый алгоритмом Тарьяна, который может решить эту задачу за время $O(V+E)$.

В этом разделе представлен еще один алгоритм на основе поиска в глубину, который можно использовать для поиска компонент сильной связности, – алгоритм Косараджу (Kosaraju). Основная идея этого алгоритма заключается в том, что поиск в глубину выполняется дважды. Первый поиск в глубину выполняется в исходном ориентированном графе, при этом вершины записываются в порядке убывания времени выхода, как при выполнении топологической сортировки[1] в разделе 4.2.5. Второй поиск в глубину выполняется на транспонированном ориентированном графе в порядке убывания времени выхода, определенного в процессе первого поиска в глубину. Этих двух проходов поиска в глубину вполне достаточно, чтобы найти компоненты сильной связности в ориентированном графе. Реализация этого алгоритма на языке C++ показана ниже. Рекомендуем читателю более подробно изучить работу алгоритма Косараджу по другим источникам, например [7].

```
void Kosaraju( int u, int pass )    // pass = 1 (исходный), 2 (обратный, транспонированный)
{
  dfs_num[u] = 1;
  vii neighbor;              // используются различные списки смежных вершин в двух проходах
  if( pass == 1) neighbor = AdjList[u]; else neighbor = AdjListT[u];
  for( int j=0; j < (int)neighbor.size(); j++ ) {
    ii v = neighbor[j];
    if( dfs_num[v.first] == DFS_WHITE )
      Kosaraju( v.first, pass);
  }
  S.push_back( u );             // как при поиске топологической сортировки в разделе 4.2.5
}

// in int main()
  S.clear();         // первый проход для записи обхода в обратном порядке в исходном графе
  dfs_num.assign( N, DFS_WHITE);
  for( i=0; i < N; i++ )
    if( dfs_num[i] == DFS_WHITE )
      Kosaraju( i, 1 );
  numSCC = 0;                           // второй проход: обнаружение SCC-областей на основе
                                                                  // результатов 1-го прохода
  dfs_num.assign( N, DFS_WHITE );
  for( i = N-1; i >= 0; i-- )
    if( dfs_num[S[i]] == DFS_WHITE ) {
      numSCC++;
```

[1] Но это может оказаться и некорректной топологической сортировкой, так как исходный ориентированный граф может содержать циклы.

```
      Kosaraju( S[i], 2 );   // AdjListT -> транспозиция (обратный порядок) исходного графа
    }
printf( "There are %d SCCs\n", numSCC );
```

Файл исходного кода: *UVa11838.cpp/java*

Для алгоритма Косараджу требуется подпрограмма транспозиции (обращения) графа (или явное создание двух структур данных графа), которая кратко описана в разделе 2.4.1. Необходимо выполнение двух проходов по структуре данных графа. Алгоритм Тарьяна, представленный в разделе 4.2.9, не требует подпрограммы транспозиции графа и выполняется всего лишь за один проход. Тем не менее оба этих алгоритма поиска компонент сильной связности одинаково хороши и могут использоваться для решения многих (если не всех) задач класса SCC, приведенных в данной книге.

9.18. Наименьший общий предок

Описание задачи

Дано корневое дерево T с n вершинами, в котором наименьший общий предок (Lowest Common Ancestor – LCA) между двумя вершинами u и v, или LCA(u, v) определен как наименьшая вершина в дереве T, для которой обе вершины u и v являются потомками. Допускается ситуация, в которой вершина является потомком самой себя, то есть возможны случаи LCA(u, v) = u или LCA(u, v) = v.

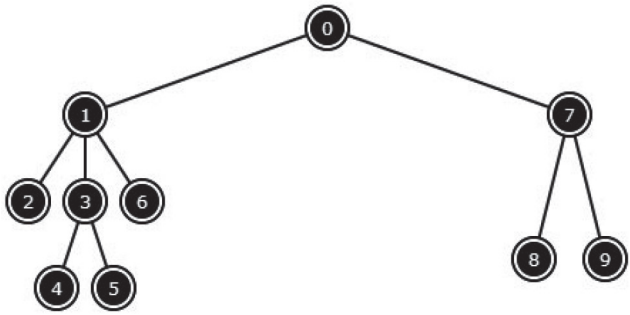


Рис. 9.8 ❖ Пример однокорневого дерева T с n = 10 вершинами

Например, на рис. 9.8 можно определить, что LCA(4, 5) = 3, LCA(4, 6) = 1, LCA(4, 1) = 1, LCA(8, 9) = 7, LCA(4, 8) = 0 и LCA(0, 0) = 0.

Решения

Полный перебор

Простейшее прямолинейное решение реализуется в два этапа: из первой вершины u выполняется подъем в корень дерева T с фиксацией (записью) всех

пройденных на этом пути вершин (сложность может быть равной $O(n)$, если дерево не сбалансировано). Из второй вершины v также выполняется подъем в корень дерева T, но в этом случае мы останавливаемся, если встречается общая вершина, уже зафиксированная на первом этапе (сложность второго этапа также может быть равна $O(n)$, если наименьшим общим предком u и v является корень дерева и дерево не сбалансировано). Эта общая вершина и есть наименьший общий предок LCA. Требуется время $O(n)$ на каждый запрос (u, v), и процесс может стать очень медленным, если запросов слишком много.

Например, если необходимо вычислить LCA(4, 6) в дереве на рис. 9.8 с применением описанного выше полного поиска, то сначала будет выполнен подъем (обход) по пути $4 \to 3 \to 1 \to 0$ с записью этих четырех вершин. Затем выполняется подъем (обход) по пути $6 \to 1$ и остановка. Найден наименьший общий предок в вершине 1.

Сведение к задаче поиска минимума на отрезке

Можно привести задачу поиска наименьшего общего предка (LCA) к задаче поиска минимума на отрезке (Range Minimum Query – RMQ) (см. раздел 2.4.3). Если структура дерева T не изменяется при выполнении всех Q запросов, то можно воспользоваться структурой данных «разреженная таблица» со временем предподсчета $O(n \times \log n)$ и временем ответа на вопрос RMQ $O(1)$. Подробности о структуре данных «разреженная таблица» (Sparse Table) приведены в разделе 9.33. В текущем разделе рассматривается процесс сведения задачи LCA к задаче поиска минимума на отрезке.

Задачу поиска наименьшего общего предка (LCA) можно привести к задаче поиска минимума на отрезке (RMQ) с линейным временем выполнения. Основная идея состоит в следующем наблюдении: наименьший общий предок LCA(u, v) – это самая высокая вершина в дереве, посещенная между посещениями вершин u и v во время прохода при поиске в глубину. Поэтому необходимо всего лишь выполнить поиск в глубину в дереве и записать информацию о глубине и времени посещения каждого узла. Следует отметить, что мы посещаем всего $2*n - 1$ вершин при поиске в глубину, поскольку внутренние вершины будут посещены несколько раз. Во время этого поиска в глубину нужно создать три массива: `E[0..2*n-2]` (для записи последовательности посещенных вершин), `L[0..2*n-2]` (для записи глубины каждой посещенной вершины) и `H[0..N-1]` (где в `H[i]` записывается индекс первой встретившейся вершины (узла) `i` в последовательности `E`).

Главная часть реализации этой задачи показана ниже:

```
int L[2*MAX_N], E[2*MAX_N], H[MAX_N], idx;

void dfs( int cur, int depth )
{
  H[cur] = idx;
  E[idx] = cur;
  L[idx++] = depth;
  for( int i=0; i < children[cur].size(); i++ ) {
    dfs( children[cur][i], depth+1 );
    E[idx] = cur;                                    // возврат в текущий узел (вершину)
```

```
    L[idx++] = depth;
  }
}

void buildRMQ()
{
  idx = 0;
  memset( H, -1, sizeof H );
  dfs( 0, 0 );              // предполагается, что корень находится в вершине с индексом 0
}
```

Файл исходного кода: *LCA.cpp/java*

Например, если функция `dfs( 0, 0 )` вызвана для дерева, изображенного на рис. 9.8, то получим[1]:

Таблица 9.1. Приведение задачи LCA к задаче RMQ

Индекс (Index)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
H	0	1	2	4	5	7	10	13	14	16	−1	−1	−1	−1	−1	−1	−1	−1	−1
E	0	1	2	1	3	4	3	5	3	*(1)*	6	1	0	7	8	7	9	7	0
L	0	1	2	1	2	3	2	3	2	1	2	1	0	1	2	1	2	1	0

Поскольку мы должны работать с этими тремя массивами, можно решить задачу поиска наименьшего общего предка (LCA) с использованием задачи поиска минимума на отрезке (RMQ). Предположим, что `H[u] < H[v]`, в противном случае поменяем местами *u* и *v*. Отметим, что рассматриваемая задача сводится к поиску вершины с наименьшей глубиной в массиве `E[H[u]..H[v]]`. Таким образом, решение определяется как LCA(u,v) = `E[RMQ(H[u], H[v])]`, где `RMQ( i, j )` выполняется в массиве `L`. Если воспользоваться структурой данных Sparse Table, рассматриваемой в разделе 9.33, то этот массив `L` необходимо обработать на этапе создания.

Например, если нужно вычислить LCA(4,6) в дереве на рис. 9.8, то вычисляется `H[4] = 5` и `H[6] = 10` и выполняется поиск вершины с наименьшей глубиной в массиве `E[5..10]`. Вызов функции `RMQ( 5, 10 )` для массива `L` (см. подчеркнутые элементы в строке `L` табл. 9.1) возвращает индекс 9. Значение `E[9] = 1` (см. значение, выделенное курсивом, в строке `E` табл. 9.1), следовательно, значение 1 определяется как ответ (решение) на запрос LCA(4,6).

Задания по программированию, связанные с задачей поиска наименьшего общего предка

1. UVa 10938 – Flea circus (описанная выше задача поиска наименьшего общего предка)
2. **UVa 12238 – Ants Colony** (очень похоже на задание UVa 10938)

[1] В разделе 4.2.1 массив `H` имеет имя `dfs_num`.

9.19. Создание магических квадратов (нечетной размерности)

Описание задачи

Магический квадрат – это двумерный массив размера $n \times n$, который содержит целые числа от $[1..n^2]$ с «магическим» свойством: сумма целых чисел в каждой строке, столбце и диагонали одинакова. Например, для $n = 5$ можно составить приведенный ниже магический квадрат, в котором сумма целых чисел в строках, столбцах и диагоналях равна 65.

$$\begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}.$$

Задача: сформировать магический квадрат с заданной размерностью n, при условии что n нечетно.

Решения

Не зная специальных алгоритмов, можно воспользоваться возвратной рекурсией, помещая поочередно каждое целое число $\in [1..n^2]$ на требуемое место. При больших значениях n такое решение полного поиска работает слишком медленно.

К счастью, существует превосходная «стратегия конструирования» магических квадратов с нечетной размерностью под названием «сиамский метод» (метод Симона де ла Лубер). Мы начинаем с пустого двумерного квадратного массива. Сначала целое число 1 записывается в середину первой строки. Затем мы перемещаемся на северо-восток, при необходимости циклически возвращаясь на противоположную границу квадрата в соответствующую (по вертикали) ячейку. Если новая посещенная ячейка пуста, мы добавляем в нее очередное целое число. Если же ячейка уже занята, то переходим на одну строку ниже, заполняем ее очередным числом и продолжаем движение на северо-восток. Выполнение этого сиамского метода показано на рис. 9.9. Мы полагаем, что самостоятельный вывод этой стратегии, вероятно, будет не столь очевидным (хотя восстановить стратегию возможно, если достаточно долго изучать несколько уже построенных этим способом магических квадратов нечетной размерности).

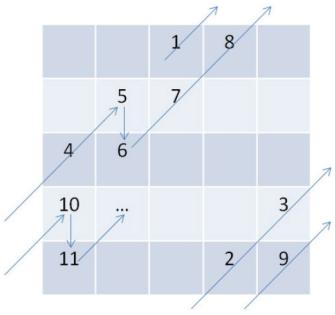


Рис. 9.9 ❖ Стратегия создания магического квадрата с нечетной размерностью n

Существуют и другие особые случаи формирования магических квадратов с различными раз-

мерностями. Возможно, нет необходимости в изучении всех таких случаев, поскольку, вероятнее всего, они не встретятся на олимпиадах по программированию. Тем не менее мы можем предположить, что некоторые участники олимпиад, знакомые с этими стратегиями составления магических квадратов, получат преимущество, если будет предложено такое задание.

Задания по программированию, связанные с задачей создания магических квадратов

1. **UVa 01266 – Magic Square *** (реализация разобранного выше алгоритма составления магического квадрата)

9.20. Задача о порядке умножения матриц

Описание задачи

Задано n матриц: $A_1, A_2, \ldots, A_n$, каждая матрица A_i имеет размер $P_{i-1} \times P_i$. Найти полную расстановку скобок в произведении $A_1 \times A_2 \times \ldots \times A_n$, при которой количество скалярных операций умножения минимально. Произведение матриц считается имеющим полную расстановку скобок в следующих двух случаях:

1) единственная матрица;
2) произведение двух произведений матриц с полной расстановкой скобок, заключенное во внешние скобки.

Например, задан массив для трех матриц $P = \{10, 100, 5, 50\}$ (который определяет, что матрица $A1$ имеет размер 10×100, матрица $A2$ имеет размер 100×5, матрица $A3$ имеет размер 5×50). Можно определить полную расстановку скобок при умножении этих трех матриц двумя следующими способами:

1) $(A1 \times (A2 \times A3)) = 100 \times 5 \times 50 + 10 \times 100 \times 50 = 75\,000$ скалярных произведений;
2) $((A1 \times A2) \times A3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ скалярных произведений.

Из приведенного выше примера становится ясно, что стоимость (накладные расходы) умножения этих трех матриц – в численном выражении количества скалярных произведений – зависит от выбора полной расстановки скобок в выражении произведения матриц. Но полная исчерпывающая проверка перебором всех возможных вариантов полной расстановки скобок выполняется слишком медленно, поскольку существует огромное количество таких возможных вариантов (для заинтересованного читателя: существует $\mathrm{Cat}(n-1)$ полных расстановок скобок для n матриц – см. раздел 5.4.3).

Операция умножения матриц

Мы можем умножить две матрицы: a с размером $p \times q$ и b с размером $q \times r$, если количество столбцов матрицы a равно количеству строк матрицы b (согласование внутренней размерности матриц). Результатом этого умножения является

матрица c с размером $p \times r$. Стоимость (накладные расходы) такой корректной операции умножения матриц равна $O(p \times q \times r)$ произведений, и эта операция может быть реализована с помощью приведенного ниже кода на языке C++.

```
#define MAX_N 10           // при необходимости можно увеличить или уменьшить это значение
struct Matrix { int mat[MAX_N][MAX_N]; };

Matrix matMul( Matrix a, Matrix b, int p, int q, int r )                    // O(p*q*r)
{
  Matrix c; int i, j, k;
  for( i=0; i < p; i++ )
    for( j=0; j < r; j++ )
      for( c.mat[i][j] = k = 0; k < q; k++ )
        c.mat[i][j] += a.mat[i][k] + b.mat[k][j];
  return c;
}
```

Например, если у нас имеется 2×3 матрица a и 3×1 матрица b, показанные ниже, то потребуется 2×3×1 = 6 скалярных произведений.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \times \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \end{bmatrix} = \begin{bmatrix} c_{1,1} = a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} + a_{1,3} \times b_{3,1} \\ c_{2,1} = a_{2,1} \times b_{1,1} + a_{2,2} \times b_{2,1} + a_{2,3} \times b_{3,1} \end{bmatrix}.$$

Если обе матрицы квадратные с размером $n \times n$, то операция умножения таких матриц выполняется за время $O(n^3)$ (см. раздел 9.21, в котором рассматривается алгоритм, очень похожий на рассматриваемый здесь).

Решения

Задача о порядке умножения матриц обычно представляет собой один из классических примеров, иллюстрирующих методику динамического программирования (dynamic programming – DP). Поскольку динамическое программирование подробно рассматривалось в разделе 3.5, здесь мы напомним только основные принципы. Отметим, что для решения этой задачи мы в действительности не будем перемножать матрицы, как было показано в предыдущем подразделе. Необходимо только найти оптимальный вариант полной расстановки скобок для n матриц.

Пусть cost(i, j), где $i < j$ обозначает количество скалярных произведений, необходимых для умножения матриц $A_i \times A_{i+1} \times \ldots \times A_j$. У нас имеются следующие рекуррентные отношения полного поиска:

1) cost(i, j) = 0, если $i = j$;
2) cost(i, j) = min(cost(i, k) + cost(k + 1, j) + $P_{i-1} \times P_k \times P_j$), $\forall k \in [i \ldots j - 1]$.

Оптимальная стоимость записана в cost(1, n). Существует $O(n^2)$ различных пар подзадачи (i, j). Поэтому необходима таблица динамического программирования размера $O(n^2)$. Каждая подзадача требует до $O(n)$ вычислений. Следовательно, сложность по времени такого решения методом динамического программирования для задачи о порядке умножения матриц равна $O(n^3)$.

Задания по программированию, связанные с задачей о порядке умножения матриц

1. **UVa 00348 – Optimal Array Mult ... *** (как описано выше, результатом также является оптимальное решение; отметим, что оптимальная последовательность умножения матриц не является единственной, например допустим, что все матрицы являются квадратными)

9.21. Возведение матрицы в степень

Некоторые определения и примеры использования

В этом разделе рассматривается особый вид матрицы[1]: квадратная матрица (square matrix)[2]. Точнее говоря, мы рассматриваем особую операцию с квадратной матрицей: степени квадратной матрицы. С математической точки зрения $M^0 = \mathrm{I}$ и $M^p = \prod_{i=1}^{p} M$. I – это единичная матрица (identity matrix)[3], а p – заданная степень квадратной матрицы M. Если можно выполнить эту операцию за время $O(n^3 \log p)$, что является главной темой текущего подраздела, то мы сможем решить некоторые более интересные задачи, предлагаемые на олимпиадах по программированию, например:

- вычисление одного отдельного[4] числа Фибоначчи fib(p) за время $O(\log p)$ вместо $O(p)$. Если предположить, что $p = 2^{30}$, то решение $O(p)$ приведет к превышению лимита времени (TLE), но решение $\log_2(p)$ требует всего лишь 30 шагов. Такое решение достижимо при использовании следующего равенства:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^p = \begin{bmatrix} \mathrm{fib}(p+1) & \underline{\mathbf{fib(p)}} \\ \underline{\mathbf{fib(p)}} & \mathrm{fib}(p-1) \end{bmatrix}.$$

  Например, для вычисления fib(11) мы просто умножаем приведенную выше матрицу Фибоначчи 11 раз, то есть возводим ее в степень 11. Ответ находится на побочной диагонали полученной матрицы.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} = \begin{bmatrix} 144 & \underline{\mathbf{89}} \\ \underline{\mathbf{89}} & 55 \end{bmatrix} = \begin{bmatrix} \mathrm{fib}(12) & \underline{\mathbf{fib(11)}} \\ \underline{\mathbf{fib(11)}} & \mathrm{fib}(10) \end{bmatrix};$$

---

[1] Матрица – это прямоугольный двумерный массив чисел. Матрица размером $m \times n$ содержит m строк и n столбцов. Элементы матрицы обычно записываются как имя матрицы с двумя подстрочными индексами.

[2] Квадратная матрица – это матрица с одинаковым количеством строк и столбцов, то есть ее размер $n \times n$.

[3] Единичная матрица содержит единицы на главной диагонали, а все остальные ее элементы равны нулю.

[4] Если необходимо вычислить fib(n) для всех $n \in [0..n]$, то следует использовать решение с применением динамического программирования с временно́й сложностью $O(n)$.

- вычисление количества путей длины L в графе, хранящемся в виде матрицы смежности вершин, которая представляет собой квадратную матрицу, за время $O(n^3 \log L)$. Пример: рассмотрим небольшой граф с размером $n = 4$, сохраненный в матрице смежности вершин M, приведенной ниже. Различные пути от вершины 0 до вершины 1 с разными длинами показаны в элементе $M[0][1]$ после возведения матрицы M в степень L.

```
Граф:     0->1 с длиной 1: 0->1 (только 1 путь)
          0->1 с длиной 2: невозможен
0--1      0->1 с длиной 3: 0->1->2->1 (и 0->1->0->1)
   |      0->1 с длиной 4: невозможен
   2--3   0->1 с длиной 5: 0->1->2->3->2->1 (и 4 других пути)
```

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}; \quad M^2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}; \quad M^3 = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 2 & 0 & 3 & 0 \\ 0 & 3 & 0 & 2 \\ 1 & 0 & 2 & 0 \end{bmatrix}; \quad M^5 = \begin{bmatrix} 0 & 5 & 0 & 3 \\ 5 & 0 & 8 & 0 \\ 0 & 8 & 0 & 5 \\ 3 & 0 & 5 & 0 \end{bmatrix};$$

- ускорение решения некоторых задач динамического программирования, как показано ниже в текущем разделе.

Основной принцип возведения в степень

Обсудим следующую ситуацию: предположим, что встроенные библиотечные функции, такие как pow(base, p) или другие подобные функции, которые могут возводить число base в заданную целую степень p, не существуют. Тогда если мы выполняем возведение в степень «по определению», как показано ниже, то получим весьма неэффективное решение $O(p)$, особенно если значение p велико[1].

```
int normalExp( int base, int p )        // для упрощения используется только тип данных int
{
  int ans = 1;        // также предполагается, что число ans не превышает значения 2^31 - 1
  for( int i=0; i < p; i++ ) ans *= base;                              // это решение O(p)
  return ans;
}
```

Существует более эффективное решение, использующее принцип «разделяй и властвуй». Для выражения A^p можно записать следующие частные варианты:

- $A^0 = 1$ (простейший случай);
- $A^1 = A$ (еще один простейший случай, но см. **упражнение 9.21.1**);
- $A^p = A^{p-1} \times A$, если p нечетно;
- $A^p = (A^{p/2})^2$, если p четно.

Поскольку при таком подходе появляется возможность деления значения p на два, возведение в степень выполняется за время $O(\log p)$.

---

[1] Если в заданиях олимпиады по программированию вам встретится исходное значение «гигантского» размера, например 10^9 (миллиард – 1B), то автор задания обычно подразумевает поиск логарифмического решения. Отметим, что $\log_2(1B) \approx \log_2(2^{30})$, то есть требуются все те же 30 шагов.

Например, при возведении в степень по определению: $2^9 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \approx O(p)$ операций умножения.

Но если использовать принцип «разделяй и властвуй», то: $2^9 = 2^8 \times 2 = (2^4)^2 \times 2 = ((2^2)^2)^2 \times 2 \approx O(\log p)$ операций умножения.

Простая рекурсивная реализация такой операции возведения в степень с применением принципа «разделяй и властвуй» (без учета тех случаев, когда результат выходит за границы диапазона 32-битового целого числа) показана ниже.

```
int fastExp( int base, int p )                                  // это решение O(log p)
{
  if( p == 0 ) return 1;
  else if( p == 1) return base;              // см. Упражнение в конце текущего подраздела
  else {
    int res = fastExp( base, p/2 ); res *= res;
    if( p % 2 == 1 ) res *= base;
    return res;
  }
}
```

Упражнение 9.21.1*. Действительно ли необходима обработка второго простейшего варианта: `if( p == 1 ) return base;`?

Упражнение 9.21.2*. Возведение числа в заданную (целую) степень может быстро привести к переполнению. Заслуживает внимания вариант вычисления $\text{base}^p \pmod m$. Перепишите функцию `fastExp( base, p )` как функцию `modPow( base, p, m )` (также см. разделы 5.3.2 и 5.5.8).

Упражнение 9.21.3*. Перепишите рекурсивную реализацию с применением принципа «разделяй и властвуй» в итеративной форме. Совет: продолжайте изучение текущего раздела.

Возведение квадратной матрицы в степень

Можно воспользоваться все той же эффективной методикой возведения в степень $O(\log p)$, описанной выше, для выполнения операции возведения в степень квадратной матрицы с временной сложностью $O(n^3 \log p)$, поскольку каждая операция умножения матриц[1] имеет временну́ю сложность $O(n^3)$. Итеративная реализация (для сравнения с рекурсивной реализацией, приведенной выше) показана ниже.

```
#define MAX_N 2                                                 // Размер матрицы Фибоначчи;
                                  // при необходимости можно увеличить/уменьшить это значение
```

---

[1] Существует более быстрый, но более сложный алгоритм умножения матриц: алгоритм Штрассена (Strassen) $O(n^{2.8074})$. Обычно этот алгоритм не используется на олимпиадах по программированию. Умножение двух матриц Фибоначчи в разделе 9.21 требует только лишь $2^3 = 8$ операций умножения, так как $n = 2$. Это можно оценить как $O(1)$. Таким образом, можно вычислить $\text{fib}(p)$ за время $O(\log p)$.

```
struct Matrix { int mat[MAX_N][MAX_N]; };                    // возвращается двумерный массив

Matrix matMul( Matrix a, Matrix b )                                  // это решение O(n^3)
{
  Matrix ans; int i, j, k;
  for( i=0; i < MAX_N; i++ )
    for( j=0; j < MAX_N; j++ )
      for( ans.mat[i][j] = k = 0; k < MAX_N; k++ )          // если необходимо, используйте
        ans.mat[i][j] += a.mat[i][k] * b.mat[k][j];       // арифметическую операцию взятия
                                                                          // по модулю
  return ans;
}

Matrix matPow( Matrix base, int p )                              // это решение O(n^3 log p)
{
  Matrix ans; int i, j;
  for( i=0; i < MAX_N; i++ )
    for( j=0; j < MAX_N; j++ )
      ans.mat[i][j] = (i == j);                               // подготовка единичной матрицы
  while( p ) { // итеративная версия возведения в степень по принципу "разделяй и властвуй"
    if( p&1 ) ans = matMul( ans, base );          // если p нечетно (последний бит равен 1)
    base = matMul( base, base );                       // возведение в квадрат матрицы base
    p >>= 1;                                                          // p делится на 2
  }
  return ans;
}
```

Файл исходного кода: *UVa10229.cpp/java*

Ускорение решения с использованием динамического программирования с возведением в степень матрицы

В этом разделе мы рассмотрим, как получить (вывести) требуемые квадратные матрицы для двух задач динамического программирования, и продемонстрируем, как возведение этих двух квадратных матриц в заданные степени может ускорить вычисление решения исходных задач динамического программирования.

Начнем с матрицы Фибоначчи 2×2. Известно, что fib(0) = 0, fib(1) = 1, а для $n \geqslant 2$ определено, что *fib(n)* = fib(n – 1) + fib(n – 2). Можно вычислить fib(n) за время $O(n)$, используя динамическое программирование с вычислением fib(n) при последовательном переборе значений из интервала [2..n]. Но эти преобразования метода динамического программирования можно сделать более быстрыми, если переписать рекуррентную формулу чисел Фибоначчи в матричной форме, как показано ниже.

Сначала запишем две известные версии рекуррентной формулы чисел Фибоначчи, соответствующие двум типам рекуррентности:

fib(n + 1) + fib(n) = fib(n + 2);
fib(n) + fib(n – 1) = fib(n + 1).

Потом перепишем эти рекуррентные формулы в матричной форме:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} \text{fib}(n+1) \\ \text{fib}(n) \end{bmatrix} = \begin{bmatrix} \text{fib}(n+2) \\ \text{fib}(n+1) \end{bmatrix}.$$

Теперь мы получили выражения $a \times \text{fib}(n + 1) + b \times \text{fib}(n) = \text{fib}(n + 2)$ и $c \times \text{fib}(n + 1) + d \times \text{fib}(n) = \text{fib}(n + 1)$. Отметим, что при записи рекуррентных формул динамического программирования в приведенном выше виде мы получаем квадратную матрицу 2×2. Соответствующие значения для a, b, c, d обязательно должны быть равны 1, 1, 1, 0, а это и есть матрица Фибоначчи, показанная выше. Одна операция умножения матриц продвигает вычисление числа Фибоначчи методом динамического программирования на один шаг вперед. Если умножить эту матрицу Фибоначчи 2×2 саму на себя p раз, то процесс вычисления методом динамического программирования продвинется на p шагов вперед. Тогда мы получим:

$$\underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \cdots \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{p} \times \begin{bmatrix} \text{fib}(n+1) \\ \text{fib}(n) \end{bmatrix} = \begin{bmatrix} \text{fib}(n+1+p) \\ \text{fib}(n+p) \end{bmatrix}.$$

Например, если задано $n = 0$ и $p = 11$ и используется возведение матрицы в степень $O(\log p)$ вместо прямого перемножения матриц p раз, то выполняются следующие вычисления:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{11} \times \begin{bmatrix} \text{fib}(1) \\ \text{fib}(0) \end{bmatrix} = \begin{bmatrix} 144 & 89 \\ 89 & 55 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 144 \\ \underline{\mathbf{89}} \end{bmatrix} = \begin{bmatrix} \text{fib}(12) \\ \mathbf{fib(11)} \end{bmatrix}.$$

Матрицу Фибоначчи можно также записать, как было показано выше, то есть:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{p} = \begin{bmatrix} \text{fib}(p+1) & \text{fib}(p) \\ \text{fib}(p) & \text{fib}(p-1) \end{bmatrix}.$$

Рассмотрим еще один пример получения требуемой квадратной матрицы для другой задачи динамического программирования: UVa 10655 – Contemplation, Algebra. Задача описывается очень просто: даны значения $p = a + b$, $q = a \times b$ и n, найти значение выражения $a^n + b^n$.

Сначала изменим целевое выражение так, чтобы можно было воспользоваться заданными в условии равенствами $p = a + b$ и $q = a \times b$:

$$a^n + b^n = (a + b) \times (a^{n-1} + b^{n-1}) - (a \times b) \times (a^{n-2} + b^{n-2}).$$

Далее определим $X_n = a^n + b^n$, чтобы получить $X_n = p \times X_{n-1} - q \times X_{n-2}$. Затем запишем эту рекуррентную формулу дважды в следующей форме:

$$p \times X_{n+1} - q \times X_n = X_{n+2};$$
$$p \times X_n - q \times X_{n-1} = X_{n+1}.$$

Затем перепишем эти рекуррентные выражения в матричной форме:

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix} = \begin{bmatrix} X_{n+2} \\ X_{n+1} \end{bmatrix}.$$

Если возвести эту квадратную матрицу 2×2 в степень n (за время $O(\log n)$), а затем умножить полученную квадратную матрицу на $X_1 = a^1 + b^1 = a + b = p$ и $X_0 = a^0 + b^0 = 1 + 1 = 2$, то получим X_{n+1} и X_n. Требуемый ответ: X_n. Этот вариант быстрее, чем стандартное вычисление методом динамического программирования $O(n)$ для той же рекуррентной формулы.

$$\begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} X_1 \\ X_0 \end{bmatrix} = \begin{bmatrix} X_{n+1} \\ X_n \end{bmatrix}.$$

Задания по программированию, связанные с задачей о возведении матрицы в степень

1. UVa 10229 – Modular Fibonacci (задача, рассмотренная в этом разделе + операция взятия по модулю)
2. **UVa 10518 – How Many Calls? *** (вывод шаблона получаемых ответов для малого значения n; ответ равен $2 \times \text{fib}(n) - 1$; затем используется решение задания UVa 10229)
3. **UVa 10655 – Contemplation, Algebra *** (рассматривается в этом разделе)
4. UVa 10870 – Recurrences (сначала формируется требуемая матрица; затем матрица возводится в степень)
5. **UVa 11486 – Finding Paths in Grid *** (моделируется как матрица смежности вершин; матрица смежности вершин возводится в степень N за время $O(\log N)$ для получения количества путей)
6. **UVa 12470 – Tribonacci** (очень похоже на задание UVa 10229; матрица 3×3 = [0 1 0; 0 0 1; 1 1 1]; ответ содержится в элементе матрицы matrix[1][1] после возведения исходной матрицы в степень n и взятия по модулю 1 000 000 009)

9.22. Задача о независимом множестве максимального веса

Описание задачи

Задан граф G со взвешенными вершинами, найти независимое множество вершин максимального веса (Max Weighted Independent Set – MWIS) этого графа G. Независимое множество (Independent Set – IS)[1] – это множество вершин в гра-

[1] Дополнением независимого множества вершин является вершинное покрытие (vertex cover).

фе, таких, что любые две вершины из этого множества не являются смежными. Наша задача – выбрать независимое множество вершин графа G с максимальным суммарным весом (вершин). Это трудная задача в общем случае. Но если заданный граф G является деревом или двудольным графом, то для этих типов существуют эффективные решения.

Решения

Для дерева

Если граф G является деревом[1], то можно найти решение задачи о независимом множестве максимального веса (MWIS) для графа G с использованием динамического программирования[2]. Пусть $C(v,\text{taken})$ является задачей MWIS для корневого поддерева в v, если она принята как часть общей задачи MWIS. Тогда имеем следующие рекуррентные формулы:

1. Если v – это вершина-лист:
   a) $C(v, \text{true}) = w(v)$
      % Если лист v взят, то вес этого поддерева равен весу этого листа v;
   b) $C(v, \text{false}) = 0$
      % Если лист v не взят, то вес этого поддерева равен 0.
2. Если v – это внутренняя вершина:
   a) $C(v, \text{true}) = w(u) + \sum_{\text{ch} \in \text{children}(v)} C(\text{ch}, \text{false})$
      % Если лист v взят, то мы добавляем вес этого листа v, но все потомки v исключаются (не принимаются);
   b) $C(v, \textit{false}) = \sum_{\text{ch} \in \text{children}(v)} \max(C(\text{ch}, \text{true}), C(\text{ch}, \text{false}))$
      % Если лист v не взят, то потомки v могут быть взяты или не взяты.
      % Возвращается большее значение.

Ответ (решение): $\max(C(\text{root},1), C(\text{root},0))$ – взятый или невзятый корень. Это решение с применением динамического программирования требует только $O(V)$ пространства (памяти) и времени $O(V)$.

Для двудольного графа

Если G является двудольным графом, то мы должны свести задачу о независимом множестве вершин максимального веса (MWIS)[3] к задаче нахождения

---

[1] Для большинства задач, связанных с деревьями, в первую очередь необходимо определить «корень дерева», если оно пока еще не является корневым деревом (то есть деревом с корнем). Если дерево не содержит вершину, выделенную как корень, то в качестве корня выбирается произвольная вершина. После этого могут возникать дополнительные подзадачи, относящиеся к поддеревьям, как в нашем случае задача MWIS для дерева.

[2] Некоторые задачи оптимизации для дерева могут быть решены с применением методов динамического программирования. Решение обычно предполагает передачу информации как в направлении к родителю, так и в направлении от родителя, а также получение информации от потомка и направление ее к потомку в корневом дереве.

[3] Задача о невзвешенном максимальном независимом множестве (Max Independent Set – MIS) в двудольном графе может быть сведена к задаче нахождения паросочетания максимальной мощности в двудольном графе (Max Cardinality Bipartite Matching – MCBM) – см. раздел 4.7.4.

максимального потока (Max Flow) . Мы назначаем исходную стоимость вершины (вес взятия этой вершины) как мощность пути от источника до этой вершины для левого множества вершин двудольного графа и мощность пути от этой вершины до стока для правого множества вершин двудольного графа. Затем мы присваиваем «бесконечную» мощность любому ребру между левым и правым множествами вершин. Независимое множество вершин максимального веса (MWIS) для этого двудольного графа – это вес стоимостей всех вершин минус максимальное значение потока этого потокового графа.

9.23. Максимальный поток минимальной стоимости

Описание задачи

Задача о потоке минимальной стоимости (Min Cost Flow) подразумевает поиск самого дешевого возможного пути передачи определенного объема (количества) (обычно максимального) потока через потоковую сеть (потоковый граф). В этой задаче каждое ребро имеет два атрибута: мощность (пропускная способность) потока через данное конкретное ребро и цена передачи одной единицы измерения потока через это ребро. Авторы некоторых задач предпочитают упростить условия, устанавливая мощность (пропускную способность) ребер равной постоянному целому числу и изменяя только цену ребер.

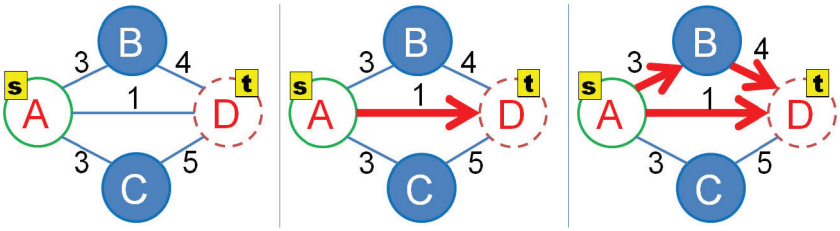


Рис. 9.10 ❖ Пример задачи максимального потока минимальной стоимости (MCMF) (UVa 10594 [47])

На рис. 9.10 слева показан измененный вариант задания UVa 10594. Здесь каждое ребро имеет одинаковую пропускную способность 10 единиц, а цена указана на метках ребер. Необходимо передать 20 единиц потока от A до D (отметим, что максимальный поток этого потокового графа равен 30 единицам). Этому условию удовлетворяет передача 10 единиц потока $A \to D$ со стоимостью $1 \times 10 = 10$ (рис. 9.10, в середине) плюс другие 10 единиц потока $A \to B \to D$ со стоимостью $(3 + 4) \times 10 = 70$ (рис. 9.10, справа). Общая стоимость равна $10 + 70 = 80$, и это минимальная стоимость. Отметим, что при выборе иного варианта передачи 20 единиц потока: через $A \to D$ (10 единиц) и $A \to C \to D$ – общая стоимость была бы равна $1 \times 10 + (3 + 5) \times 10 = 10 + 80 = 90$. Это больше оптимальной стоимости, равной 80.

Решения

Задачу максимального потока минимальной стоимости (Min Cost Max Flow – MCMF) можно решить, заменив поиск в ширину $O(E)$ (для поиска кратчайшего по количеству переходов по узлам увеличивающего пути) в алгоритме Эдмондса–Карпа на алгоритм Беллмана–Форда $O(VE)$ (для поиска кратчайшего/самого дешевого – в единицах измерения стоимости пути – увеличивающего пути). Нам необходим алгоритм поиска кратчайшего пути, который способен обрабатывать ребра отрицательного веса, поскольку ребра отрицательного веса могут появляться при отмене передачи конкретной величины потока, то есть при проходе по ребру в обратном направлении (так как мы должны вычесть цену этого фрагмента увеличивающего пути, поскольку отмена потока означает, что мы не будем использовать это ребро). Пример см. на рис. 9.5.

Необходимость использования алгоритма поиска кратчайшего пути, подобного алгоритму Форда–Беллмана, замедляет реализацию решения MCMF приблизительно до $O(V^2E^2)$, но авторы большинства задач MCMF обычно смягчают ограничения исходного графа, чтобы сбалансировать сложность задачи.

Задания по программированию, связанные с задачей максимального потока минимальной стоимости

1. UVa 10594 – Data Flow (основная задача максимального потока минимальной стоимости)
2. **UVa 10746 – Crime Wave – The Sequel *** (поиск паросочетаний в двудольном графе с минимальным весом)
3. UVa 10806 – Dijkstra, Dijkstra (передача двух потоков по несмежным ребрам с минимальной стоимостью)
4. **UVa 10888 – Warehouse *** (поиск в ширину / поиск кратчайшего пути из одной вершины во все остальные (SSSP); поиск паросочетаний в двудольном графе с минимальным весом)
5. **UVa 11301 – Great Wall of China *** (моделирование, пропускная способность вершин, задача максимального потока минимальной стоимости (MCMF))

9.24. Минимальное покрытие путями в ориентированном ациклическом графе

Описание задачи

Задача минимального покрытия путями (Minimal Path Cover – MPC) в ориентированном ациклическом графе (Directed Acyclic Graph – DAC) рассматривается как задача поиска минимального количества путей, покрывающих каждую вершину ориентированного ациклического графа $G = (V, E)$. Говорят, что путь $v_0, v_1, \ldots, v_k$ покрывает все вершины, включенные в этот путь.

Задача, которая послужила поводом для обсуждения этой темы, – UVa 1201 – Taxi Cab Scheme: предположим, что вершины на рис. 9.11A обозначают пассажиров. Тогда ребро между двумя вершинами u–v существует в том случае, если одно такси может обслужить пассажира u после этого пассажира v. Вопрос: какое минимальное количество такси потребуется для обслуживания всех пассажиров?

Ответ: два такси. На рис. 9.11D можно видеть одно из возможных оптимальных решений. Одно такси (пунктирная линия) обслуживает пассажира 1, пассажира 2, затем пассажира 4. Другое такси (штриховая линия) обслуживает пассажира 3 и пассажира 5. Все пассажиры получили обслуживание при использовании всего лишь двух такси. Отметим, что существует еще одно оптимальное решение: 1 → 3 → 5 и 2 → 4.

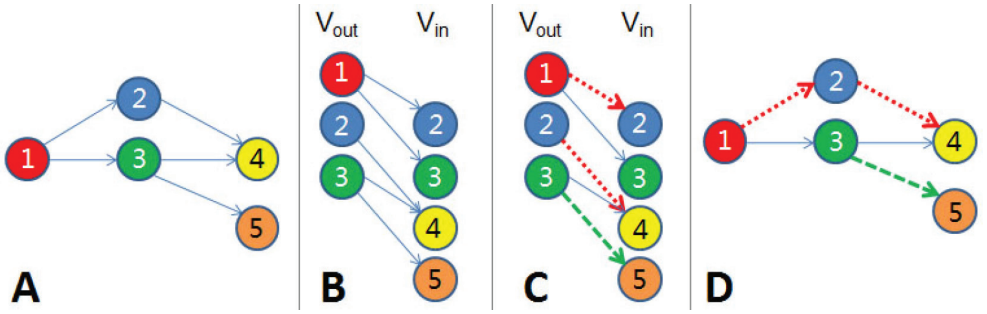


Рис. 9.11 ❖ Минимальное покрытие путями
в ориентированном ациклическом графе (из задания UVa 1201 [47])

Решения

Эта задача имеет полиномиальное решение: создается двудольный граф $G' = (V_{out} \cup V_{in}, E')$ из графа G, где $V_{out} = \{v \in V : v$ имеет положительную исходящую степень$\}$, $V_{in} = \{v \in V : v$ имеет положительную степень входа$\}$ и $E' = \{(u, v) \in (V_{out}, V_{in}) : (u, v) \in E\}$. Граф G' является двудольным графом. Поиск паросочетаний в двудольном графе G' заставляет нас выбрать по меньшей мере одно исходящее ребро из каждой вершины $u \in V_{out}$ (и точно так же, как минимум, одно входящее ребро для каждой вершины $v \in V_{in}$). Ориентированный ациклический граф G изначально содержит n вершин, которые могут быть покрыты n путями длины 0 (самими этими вершинами). Одно паросочетание между вершиной a и вершиной b, применяющее ребро (a, b), означает, что мы можем использовать на один путь меньше, так как ребро $(a, b) \in E'$ может покрыть обе вершины $a \in V_{out}$ и $b \in V_{in}$. Следовательно, если паросочетание максимальной мощности в двудольном графе (МСВМ) G' имеет размер m, то необходимо всего лишь $n - m$ путей для покрытия каждой вершины в исходном графе G.

Паросочетание максимальной мощности в двудольном графе G', необходимое для решения задачи минимального покрытия путями в исходном графе G, может быть найдено с помощью нескольких полиномиальных алгоритмов, например алгоритмом нахождения максимального потока, использованием

алгоритма поиска увеличивающих путей или алгоритма Хопкрофта–Карпа (см. раздел 9.10). Поскольку решение задачи нахождения паросочетания максимальной мощности в двудольном графе выполняется за полиномиальное время, решение задачи минимального покрытия путями в ориентированном ациклическом графе также выполняется за полиномиальное время. Следует отметить, что задача минимального покрытия путями в общем случае является *NP*-трудной задачей.

Задания по программированию, связанные с задачей минимального покрытия путями в обобщенном графе

1. **UVa 01184 – Air Raid** * (LA 2696, Dhaka02, задача минимального покрытия путями в ориентированном ациклическом графе ≈ задаче нахождения паросочетания максимальной мощности в двудольном графе)
2. **UVa 01201 – Taxi Cab Scheme** * (LA 3126, NWEurope04, задача минимального покрытия путями в ориентированном ациклическом графе)

9.25. Блинная сортировка

Описание задачи

Блинная сортировка (pancake sorting)[1] – классическая задача информатики, но она редко применяется на практике. Задачу можно описать следующим образом: имеется стопка из N блинов. Самый нижний блин в стопке имеет индекс 0, а самый верхний блин в стопке имеет индекс $N - 1$. Размер блина определяется его диаметром (целое число ∈ [1..MAX_D]). Все блины в стопке имеют различные диаметры. Например, стопка A из $N = 5$ блинов с диаметрами {3, 8, 7, 6, 10} может быть представлена в более наглядном виде:

```
4 (верхний блин)  10
3                  6
2                  7
1                  8
0 (нижний блин)    3
----------------------
индекс             A
```

Задача состоит в сортировке стопки блинов в убывающем порядке, то есть самый большой блин должен находиться внизу, а самый маленький блин должен переместиться на вершину стопки. Но чтобы сделать задачу более приближенной к реальной жизни, вводится условие, по которому сортировка стопки блинов может выполняться только с помощью последовательности «перево-

[1] Билл Гейтс (Bill Gates, основатель корпорации Microsoft, бывший генеральный (исполнительный) директор этой корпорации, а ныне председатель совета директоров) до настоящего времени опубликовал только одну научную исследовательскую работу, которая была посвящена как раз блинной сортировке [22].

рачиваний» нескольких блинов, а операция переворачивания обозначена как функция flip(*i*). Операция переворачивания flip(*i*) заключается во вставке шпателя (лопаточки) между двумя блинами в стопке (захват блинов от индекса *i* до индекса *N* – 1) и переворачивании блинов на лопаточке (то есть изменяется ее порядок на обратный – часть стопки [*i* .. *N* – 1]).

Например, стопку *A* можно преобразовать в стопку *B* при помощи операции flip(0), то есть вставить шпатель под блин с индексом 0 (при этом захвачены блины до индекса 4 включительно), затем перевернуть все захваченные блины. Стопку *B* можно превратить в стопку *C* операцией flip(3). Операция flip(1) позволяет перейти от стопки *C* к стопке *D*. И так далее. Наша цель – отсортировать стопку блинов в убывающем порядке (сверху вниз), то есть требуется в конечном итоге получить стопку *E*.

```
    4 (верхний блин)  10 <-- 3 <-- 8 <-- 6       3
    3                 6      8 <-- 3     7   …   6
    2                 7      7     7     3       7
    1                 8      6     6 <-- 8       8
    0 (нижний блин)   3 <--  10    10    10      10
-------------------------------------------------------
    индекс            A      B     C     D   …   E
```

Чтобы сделать задачу более трудной, предлагается вычислить минимальное число операций flip(*i*), то есть переворотов, необходимых для того, чтобы стопка из *N* блинов была отсортирована в убывающем порядке.

Вам предлагается целое число *T* в первой строке, затем *T* тестовых вариантов (случаев), по одному в каждой строке. Каждый тестовый вариант начинается с целого числа *N*, за которым следует *N* целых чисел, описывающих начальное содержимое стопки блинов. Необходимо вычислить и вывести одно целое число – минимальное количество операций переворота flip(*i*) для полной сортировки стопки.

Ограничения: $1 \leqslant T \leqslant 100$, $1 \leqslant N \leqslant 10$, $N \leqslant \text{MAX\_D} \leqslant 1\,000\,000$.

Примеры тестовых вариантов

Пример входных данных

```
7
4   4 3 2 1
8   8 7 6 5 4 1 2 3
5   5 1 2 4 3
5   555555 111111 222222 444444 333333
8   1000000 999999 999998 999997 999996 999995 999994 999993
5   3 8 7 6 10
10  8 1 9 2 0 5 7 3 6 4
```

Пример выходных данных (результатов)

```
0
1
2
2
```

```
0
4
11
```

Объяснение

- Первая стопка уже отсортирована в убывающем порядке.
- Вторую стопку можно отсортировать одним вызовом функции flip(5).
- Третью (и четвертую) исходную стопку можно отсортировать в убывающем порядке с помощью вызова функции flip(3), затем flip(1): два переворота.
- Несмотря на то что пятая исходная стопка содержит большие целые числа, она уже отсортирована в убывающем порядке, поэтому операции переворотов не требуются: ноль переворотов.
- Шестая исходная стопка в действительности представляет собой пример, рассмотренный в разделе «Описание задачи». Эту стопку можно отсортировать за минимальное количество операций, равное 4:
  - решение 1: flip(0), flip(1), flip(2), flip(1): четыре переворота;
  - решение 2: flip(1), flip(2), flip(1), flip(0): также четыре переворота.
- Седьмая стопка с $N = 10$ предлагается читателю для тестирования скорости во время выполнения вашего решения.

Решения

Необходимо сразу же отметить тот факт, что диаметры блинов в действительности не имеют значения. Нужно лишь написать простой код для сортировки этих (возможно, огромных) диаметров блинов в диапазоне [1..1 млн] и заново присвоить им метки в диапазоне $[0..N-1]$. Таким образом, можно описать любую стопку блинов просто как перестановку N целых чисел.

Если требуется только получение отсортированной стопки блинов, то можно воспользоваться неоптимальным жадным (Greedy) алгоритмом $O(2 \times N - 3)$: самый большой блин переворачивается на вершину стопки, затем выполняется переворот самого большого блина в самый низ стопки. Второй по размеру блин переворачивается на вершину, затем следует переворот, в результате которого верхний блин оказывается вторым снизу, и т. д. Продолжая выполнение этих нехитрых операций, можно отсортировать стопку блинов за $O(2 \times N - 3)$ шагов вне зависимости от начального состояния.

Но если требуется получить минимальное количество переворотов (операций flip), то необходимо смоделировать эту задачу как задачу поиска кратчайших путей в невзвешенном графе (State Space graph) (см. раздел 8.2.3). Вершина этого графа – это перестановка из N блинов (целых чисел). Вершина соединяется невзвешенными ребрами с $O(N - 1)$ другими вершинами посредством различных операций переворота (минус одна операция, поскольку переворот самого верхнего блина ничего не меняет). Затем используется поиск в ширину (BFS) от начальной перестановки (вершины начального состояния) для нахождения кратчайшего пути к целевой перестановке (где перестановка соответствует отсортированной в обратном порядке стопке блинов). Граф содержит $O(N!)$ вершин. Следовательно, поиск в ширину $O(V + E)$ выполняется за время $O(N \times N!)$

для каждого тестового варианта или за время $O(T \times N \times N!)$ для всех тестовых вариантов. Отметим, что написание кода такого поиска в ширину уже само по себе является весьма сложной задачей (см. разделы 4.4.2 и 8.2.3). Но и это решение остается слишком медленным для самого большого тестового варианта.

Простая оптимизация заключается в выполнении поиска в ширину, начиная с целевой перестановки (то есть со стопки, отсортированной в убывающем порядке) до всех прочих перестановок только по одному разу для всех возможных N из диапазона [1..10]. Это решение имеет временну́ю сложность, приблизительно равную $O(10 \times N \times N! + T)$, что намного быстрее предыдущего решения. Но даже оно остается слишком медленным для обычных условий олимпиады по программированию.

Улучшенным решением является более интеллектуальный метод поиска, который называется «встретимся в середине» (meet in the middle) (двунаправленный поиск в ширину) для уменьшения пространства поиска до приемлемого управляемого уровня (см. раздел 8.2.4). Сначала выполняется определенный предварительный анализ (можно также обратиться к материалу «Pancake Number» http://oeis.org/A058986) для определения того факта, что для наибольшего тестового варианта при $N = 10$ необходимо не более чем 11 переворотов для сортировки любой исходной стопки в целевое состояние. Таким образом, мы предварительно планируем (вычисляем) процесс поиска в ширину от целевой перестановки ко всем прочим перестановкам для всех $N \in [1..10]$, но этот процесс останавливается сразу при достижении глубины $\lfloor 11/2 \rfloor = 5$. Затем для каждого тестового варианта выполняется поиск в ширину, начиная с исходной перестановки, опять же с ограничением максимальной глубины поиска 5. Если обнаруживается общая вершина с предварительно выполненным поиском в ширину от целевой перестановки, то мы знаем, что это расстояние от исходной перестановки до данной вершины плюс расстояние от целевой перестановки до этой же вершины. Если общая вершина не обнаружена, то мы знаем, что решением является максимальное количество переворотов: 11. В самом большом тестовом варианте с $N = 10$ для всех тестовых вариантов это решение имеет временну́ю сложность, приблизительно равную $O((10 + T) \times 10^5)$, что в данном случае является вполне приемлемым результатом.

Задания по программированию, связанные с задачей блинной сортировки

1. **UVa 00120 – Stacks of Flapjacks *** (блинная сортировка, версия с использованием жадного алгоритма)
2. Задача блинной сортировки, описанная в этом разделе

9.26. Ро-алгоритм Полларда для разложения на множители целых чисел

В разделе 5.5.4 рассматривался оптимизированный алгоритм перебора делителей (пробного деления – trial division), который может применяться для поиска простых множителей чисел до $\approx 9 \times 10^{13}$ (см. **упражнение 5.5.4.1**) в услови-

ях олимпиады по программированию (то есть решение задачи за «несколько секунд», а не за несколько минут/часов/дней). Но что, если на олимпиаде по программированию предлагается разложить на простые множители 64-битовое беззнаковое целое число (то есть число $\approx 1 \times 10^{19}$)?

Для ускорения процедуры разложения на множители (факторизации) целого числа можно воспользоваться ро-алгоритмом (ρ-алгоритмом) Полларда (Pollard's rho algorithm) [52, 3]. Основная идея этого алгоритма состоит в том, что два целых числа x и y сравнимы по модулю p (p – это один из множителей целого числа n, которое необходимо разложить на множители) с вероятностью 0.5 после того, как было случайно выбрано «несколько ($1.177\sqrt{p}$) целых чисел».

Подробности теоретического обоснования этого алгоритма, вероятно, не столь важны для олимпиадного программирования. В этом разделе просто представлена работающая реализация алгоритма на языке C++, которая может использоваться для обработки составного целого числа, соответствующего 64-битовым беззнаковым целым числам, определяемым условиями олимпиадного программирования. Но ро-алгоритм Полларда не способен определить случай, когда n является большим простым числом, из-за особенностей работы этого алгоритма. Для обработки такого особого случая необходимо реализовать быстрый (вероятностный) алгоритм проверки простого числа, например алгоритм Миллера–Рабина (Miller–Rabin) (см. **упражнение 5.3.2.4***).

```
#define abs_val(a) (((a)>0?(a):-(a))
typedef long long ll;

ll mulmod( ll a, ll b, ll c ) // возвращает (a*b)%c и минимизирует вероятность переполнения
{
  ll x = 0, y = a % c;
  while( b > 0 ) {
    if( b % 2 == 1) x = (x + y) % c;
    y = (y * 2) % c;
    b /= 2;
  }
  return x%c;
}

ll gcd( ll a, ll b ) { return !b ? a : gcd( b, a % b ); }           // стандартный алгоритм
                                                                       // поиска НОД (gcd)

ll pollard_rho( ll n )
{
  int i = 0, k = 2;
  ll x = 3, y = 3;                // случайный "посев" (seed) = 3, возможны другие значения
  while( 1 ) {
    i++;
    x = (mulmod( x, x, n ) + n - 1) % n;                                  // функция генерации
    ll d = gcd( abs_val( y - x ), n );                                      // ключевой момент
    if( d != 1 && d != n ) return d;                 // найден один нетривиальный множитель
    if( i == k ) y = x, k *= 2;
  }
}

int main()
{
```

```
  ll n = 20635128449815740 47LL;                    // предполагается, что n не является
                                                    // большим простым числом
  ll ans = pollard_rho( n );                // разложение n на два нетривиальных множителя
  if( ans > n / ans ) ans = n / ans;                // сделать ans меньшим множителем
  printf( "%11d %11d", ans, n/ans );          // ожидаемый вывод: 1112041493 1855607779
}   // return 0;
```

Файл исходного кода: *Pollardsrho.cpp/java*

Можно также реализовать ро-алгоритм Полларда на языке Java и воспользоваться функцией `isProbablePrime` из класса `BigInteger`. При таком способе реализации можно принять значение n больше, чем $2^{64} - 1$, например 17 798 655 664 295 576 020 099, то есть $\approx 2^{74}$, и разложить его на множители 143 054 969 437 × 124 418 296 927. Но время выполнения ро-алгоритма Полларда увеличивается при увеличении значения n. Тот факт, что разложение на множители целых чисел является весьма трудной задачей, продолжает оставаться главной концепцией современной криптографии.

Хорошей идеей является тестирование полной реализации ро-алгоритма Полларда (то есть включение в реализацию алгоритма быстрой вероятностной проверки чисел на простоту и прочих мелких деталей) для решения двух приведенных ниже заданий по программированию.

Задания по программированию, связанные с ро-алгоритмом Полларда

1. **UVa 11476 – Factoring Large(t)…*** (см. задачу, описанную выше в этом разделе)
2. POJ 1811 – Prime Test (проверка числа на простоту), см. http://poj.org/problem?id=1811

9.27. Постфиксный калькулятор и преобразование выражений

Алгебраические выражения

Существуют три типа алгебраических выражений: инфиксные (infix) (естественный для людей способ записи алгебраических выражений), префиксные (prefix)[1] (польская запись) и постфиксные (postfix) (обратная польская запись). В инфиксных/префиксных/постфиксных выражениях оператор размещается между/перед/после двух операндов соответственно. В табл. 9.2 показаны примеры трех выражений в инфиксной, префиксной и постфиксной форме и их значения.

[1] Одним из языков программирования, использующим префиксные выражения, является Scheme (диалект языка Lisp).

Таблица 9.2. Примеры инфиксного, префиксного и постфиксного выражений

Инфиксное выражение	Префиксное выражение	Постфиксное выражение	Значение
2 + 6 * 3	+ 2 * 6 3	2 6 3 * +	20
(2 + 6) * 3	* + 2 6 3	2 6 + 3 *	24
4 * (1 + 2 * (9 / 3) - 5	* 4 - + 1 * 2 / 9 3 5	4 1 2 9 3 / * + 5 - *	8

Калькулятор с постфиксной записью выражений

Постфиксные выражения более эффективны с точки зрения вычислений, чем инфиксные выражения. Во-первых, не нужны (сложные вложенные) скобки, так как правила приоритетов операций уже встроены в постфиксные выражения. Во-вторых, промежуточные результаты можно вычислять сразу же после ввода оператора. Этих двух функциональных возможностей лишены инфиксные выражения.

Постфиксные выражения можно вычислять за время $O(n)$ с использованием алгоритма постфиксного калькулятора. Мы начинаем вычисления с пустым стеком. Считывается выражение слева направо, по одному токену за один шаг. Если встречается операнд, то мы добавляем его в стек. Если встречается оператор, снимаются два самых верхних элемента стека, выполняется требуемая операция, затем результат добавляется в стек. Наконец, когда все входные элементы считаны, возвращается самый верхний элемент в стеке (только один) как окончательный результат.

Поскольку каждый из n входных элементов обрабатывается только один раз, а все стековые операции имеют сложность $O(1)$, то этот алгоритм постфиксного калькулятора выполняется за время $O(n)$.

Пример постфиксных вычислений подробно показан в табл. 9.3.

Таблица 9.3. Пример постфиксных вычислений

Постфиксные вычисления	Стек (от дна к вершине)	Примечания
4 1 2 9 3 / * + 5 – *	4 1 2 9 3	Первые пять элементов – операнды
4 1 2 9 3 / * + 5 – *	4 1 2 3	Извлекаются 3 и 9, вычисляется 9/3, возвращается в стек результат 3
4 1 2 9 3 / * + 5 – *	4 1 6	Извлекаются 3 и 2, вычисляется 2 * 3, возвращается в стек результат 6
4 1 2 9 3 / * + 5 – *	4 7	Извлекаются 6 и 1, вычисляется 1 + 6, возвращается в стек результат 7
4 1 2 9 3 / * + 5 – *	4 7 5	Это операнд
4 1 2 9 3 / * + 5 – *	4 7 5	Извлекаются 5 и 7, вычисляется 7 – 5, возвращается в стек результат 2
4 1 2 9 3 / * + 5 – *	4 2	Извлекаются 2 и 4, вычисляется 4 * 2, возвращается в стек результат 8
4 1 2 9 3 / * + 5 – *	8	Из стека извлекается 8 как окончательный результат

Упражнение 9.27.1*. Что произойдет, если заменить постфиксные выражения на префиксные? Как вычислить префиксное выражение за время $O(n)$?

Преобразование инфиксных выражений в постфиксные

Зная о том, что постфиксные выражения более эффективны с точки зрения вычислений, чем инфиксные выражения, многие компиляторы выполняют преобразование инфиксных выражений в исходном коде (в большинстве языков программирования используются инфиксные выражения) в постфиксные выражения. Для использования более эффективно работающего постфиксного калькулятора, принцип работы которого был показан в предыдущем разделе, необходима возможность преобразования инфиксных выражений в постфиксные наиболее рациональным способом. Одним из возможных вариантов является алгоритм сортировочной станции (Shunting yard), разработанный Эдсгером Дейкстрой (Edsger Dijkstra) (он же автор алгоритма Дейкстры – см. раздел 4.4.3).

Алгоритм сортировочной станции обладает свойствами, схожими с алгоритмом проверки соответствия скобок (см. раздел 9.4) и алгоритмом работы постфиксного калькулятора из предыдущего раздела. Алгоритм сортировочной станции также использует стек, который изначально пуст. Выражение считывается слева направо, по одному элементу (литералу, токену) за один шаг. Если встречается операнд, то он сразу же выводится (передается в поток вывода); если встречается открывающая скобка, то она помещается в стек. При обнаружении закрывающей скобки выводятся элементы с вершины стека до тех пор, пока из стека не будет извлечена закрывающая скобка (но открывающая скобка не выводится). Если встретился оператор, то вывод продолжается, а затем извлекается самый верхний элемент стека, если он имеет больший или равный приоритет в сравнении с текущим оператором, или пока не встретится открывающая скобка – тогда текущий обрабатываемый оператор помещается в стек. В завершение процедуры мы продолжаем вывод, затем извлекаем элементы с вершины стека до тех пор, пока стек не станет пустым.

Поскольку каждый из n элементов обрабатывается только один раз, а все стековые операции имеют сложность по времени $O(1)$, алгоритм сортировочной станции выполняется за время $O(n)$.

Пример выполнения алгоритма сортировочной станции показан в табл. 9.4.

Таблица 9.4. Пример выполнения алгоритма сортировочной станции

Инфиксное выражение	Стек	Постфиксное выражение	Примечания
4 * (1 + 2 * (9 / 3) – 5)		4	Немедленный вывод
4 * (1 + 2 * (9 / 3) – 5)	*	4	Запись в стек
4 * (1 + 2 * (9 / 3) – 5)	* (	4	Запись в стек
4 * (1 + 2 * (9 / 3) – 5)	* (	4 1	Немедленный вывод
4 * (1 + 2 * (9 / 3) – 5)	* (+	4 1	Запись в стек
4 * (1 + 2 * (9 / 3) – 5)	* (+	4 1 2	Немедленный вывод
4 * (1 + 2 * (9 / 3) – 5)	* (+ *	4 1 2	Запись в стек
4 * (1 + 2 * (9 / 3) – 5)	* (+ * (	4 1 2	Запись в стек
4 * (1 + 2 * (9 / 3) – 5)	* (+ * (	4 1 2 9	Немедленный вывод
4 * (1 + 2 * (9 / 3) – 5)	* (+ * (/	4 1 2 9	Запись в стек
4 * (1 + 2 * (9 / 3) – 5)	* (+ * (/	4 1 2 9 3	Немедленный вывод

Таблица 9.4 *(окончание)*

Инфиксное выражение	Стек	Постфиксное выражение	Примечания
4 * (1 + 2 * (9 / 3) – 5)	* (+ *	4 1 2 9 3 /	Вывод только «/»
4 * (1 + 2 * (9 / 3) – 5)	* (–	4 1 2 9 3 / * +	Вывод «*», затем «+»
4 * (1 + 2 * (9 / 3) – 5)	* (–	4 1 2 9 3 / * + 5	Немедленный вывод
4 * (1 + 2 * (9 / 3) – 5)	*	4 1 2 9 3 / * + 5 –	Вывод только «–»
4 * (1 + 2 * (9 / 3) – 5)		4 1 2 9 3 / * + 5 – *	Стек пуст

Задания по программированию, связанные с обработкой постфиксных выражений

1. **UVa 00727 – Equation** * (классическая задача преобразования инфиксных выражений в постфиксные)

9.28. Римские цифры

Описание задачи

Римская система нумерации и записи чисел применялась в Древнем Риме. В действительности это десятичная система счисления, но в ней используются определенные буквы латинского алфавита вместо цифр [0..9] (подробности описаны ниже). Это не позиционная система записи, и в ней нет символа, обозначающего ноль.

В римской системе записи чисел задействованы 7 букв с соответствующими десятичными значениями: I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000. В римской системе также имеются следующие устойчивые пары букв: IV = 4, IX = 9, XL = 40, XC = 90, CD = 400, CM = 900.

В задачах по программированию, связанных с римской системой записи чисел, обычно предполагается преобразование из арабской системы записи чисел (десятичная система счисления и записи чисел, которую мы используем повседневно) в римскую систему записи чисел и наоборот. Такие задачи появляются весьма редко на олимпиадах по программированию, и решение для предложенного преобразования может быть получено сразу же после прочтения условий задачи.

Решения

В этом разделе мы предлагаем одну из библиотек преобразования, которая использовалась для решения нескольких программных задач, связанных с обработкой римских цифр. Несмотря на то что вы сами могли бы написать этот код преобразования без особого труда, по крайней мере вам не придется заниматься отладкой[1], если вы уже используете эту библиотеку.

[1] Если в задаче использован другой стандарт римских цифр, то может потребоваться внесение некоторых изменений в приведенный здесь исходный код.

```
void AtoR( int A )
{
  map<int, string> cvt;
  cvt[1000] = "M"; cvt[900] = "CM"; cvt[500] = "D"; cvt[400] = "CD";
  cvt[100]  = "C"; cvt[90]   = "XC"; cvt[50] = "L"; cvt[40] = "XL";
  cvt[10]  = "X"; cvt[9]   = "IX"; cvt[5]   "V"; cvt[4]   "IV";
  cvt[1]   = "I";
  // обработка выполняется в направлении от больших значений к меньшим значениям
  for( map<int, string>::reverse_iterator i = cvt.rbegin();  i != cvt.rend(); i++ ) {
    while( A >= i->first ) {
      printf( "%s", ((string)i->second).c_str() );
      A -= i->first;
    }
  printf( "\n" );
}

void RtoA( char R[] )
{
  map<char, int> RtoA;
  RtoA['I'] = 1; RtoA['V'] = 5; RtoA['X'] = 10; RtoA['L'] = 50; RtoA['C'] = 100;
  RtoA['D'] = 500; RtoA['M'] = 1000;

  int value = 0;
  for( int i=0; R[i]; i++ )
    if( R[i+1] && RtoA[R[i]] < RtoA[R[i+1]] ) {     // сначала проверяется следующий символ
      value += RtoA[R[i+1]] - RtoA[R[i]];                                  // по определению
      i++;                                                              // пропустить этот символ
    }
    else value += RtoA[R[i]];
  printf("%d\n", value );
}
```

Файл исходного кода: *UVa11616.cpp/java*

Задания по программированию, связанные с обработкой римских цифр

1. **UVa 00344 – Roman Digit it is *** (подсчитать, сколько символов, обозначающих римские цифры, используется для формирования записи всех чисел от 1 до *N*)
2. UVa 00759 – The Return of the... (римские цифры + проверка валидности)
3. **UVa 11616 – Roman Numerals *** (задача преобразования чисел, записанных в римской системе)
4. **UVa 12397 – Roman Numerals *** (преобразование, каждая римская цифра имеет значение)

9.29. *k*-я порядковая статистика

Описание задачи

Задача *k*-й порядковой статистики в массиве – это задача поиска *k*-го наименьшего[1] элемента в массиве из *n* элементов. Минимальный (наименьший) элемент является первым элементом порядковой статистики, максимальный (наибольший) элемент – *n*-й элемент порядковой статистики, а медиане (срединному элементу) соответствует $n/2$ порядковая статистика (если *n* четно, то существуют два срединных элемента).

Задача выбора наименьшего элемента в массиве используется как мотивационный пример в начале главы 3. В этом разделе мы более подробно рассмотрим саму эту задачу, ее варианты и различные решения.

Решения

Особые случаи: k = 1 и k = n

Поиск минимального ($k = 1$) или максимального ($k = n$) элемента в произвольном массиве может быть выполнен за $\Omega(n - 1)$ сравнений: самый первый элемент определяется как временный ответ, затем этот временный ответ сравнивается с другими $n - 1$ элементами поочередно и сохраняется наименьшее (или наибольшее, в зависимости от требований) значение. В конечном итоге мы получаем окончательное решение. $\Omega(n - 1)$ сравнений – это нижняя граница, то есть улучшить данный процесс невозможно. Хотя эта задача решается просто для случаев $k = 1$ и $k = n$, поиск других порядковых статистик – общая форма задачи выбора наименьшего элемента – более труден.

Алгоритм O(n²), статические данные

Простейший алгоритм поиска *k*-го наименьшего элемента: нахождение наименьшего элемента, «отбрасывание» его (например, установкой для него «фиктивного большого значения») и повторение этого процесса *k* раз. Когда значение *k* близко к 1 (или к *n*), этот алгоритм $O(kn)$ можно продолжать считать работающим за время $O(n)$, то есть трактовать *k* как «малую константу». Но наихудшим случаем является вариант, при котором требуется найти средний ($k = n/2$) элемент (медиану), тогда временна́я сложность алгоритма становится равной $O(n/2 \times n) = O(n^2)$.

Алгоритм O(n × log n), статические данные

Более эффективный алгоритм – сортировка (то есть предварительная обработка) массива за время $O(n \times \log n)$. После сортировки массива можно найти *k*-й наименьший элемент за время $O(1)$, просто возвращая содержимое по индексу $k - 1$ (при индексации, начинающейся с 0). Основной частью этого алгоритма является этап сортировки. Предполагая использование эффективного

---

[1] Следует отметить, что поиск *k*-го наибольшего элемента равнозначен поиску $(n - k + 1)$-го наименьшего элемента.

алгоритма сортировки $O(n \times \log n)$, можно считать, что общая временна́я сложность алгоритма в целом будет равна $O(n \times \log n)$.

Алгоритм с предполагаемой сложностью O(n), статические данные

Еще более эффективный алгоритм для решения задачи выбора наименьшего элемента предполагает применение принципа «разделяй и властвуй». При этом основная идея заключается в использовании алгоритма разбиения множества чисел $O(n)$ (вероятностная версия, с внесением элемента случайности) с применением алгоритма быстрой сортировки Quick Sort как подпрограммы.

Вероятностная версия алгоритма разбиения `RandomizedPartition( A, l, r )` – это алгоритм разбиения заданного диапазона чисел `[l..r]` массива `A` относительно (случайно выбранной) центральной позиции. Центральная позиция $A[p]$ – это один из элементов массива A, где $p \in [l..r]$. После разбиения все элементы $\leqslant A[p]$ размещаются до центральной позиции, а все элементы $> A[p]$ размещаются после центральной позиции. Возвращается итоговый индекс q центральной позиции. Вероятностная версия алгоритма разбиения множества чисел может быть выполнена за время $O(n)$.

После вычисления `q = RandomizedPartition( A, 0, n-1 )` все элементы $\leqslant A[p]$ будут размещены до центральной позиции, следовательно, $A[q]$ теперь находится в правильной порядковой статистике, которая равна $q + 1$. Далее существуют три возможных варианта:

1) $q + 1 = k$, тогда $A[q]$ – требуемый ответ. Это значение возвращается, и процесс останавливается;
2) $q + 1 > k$, тогда требуемый ответ находится в левой части разбиения, то есть в диапазоне $A[0..q - 1]$;
3) $q + 1 < k$, тогда требуемый ответ находится в правой части разбиения, то есть в диапазоне $A[q + 1..n - 1]$.

Этот процесс можно повторять рекурсивно с постоянно уменьшающимся диапазоном поиска до тех пор, пока не будет найден требуемый ответ. Ниже приведен фрагмент кода C++, который реализует этот алгоритм.

```
int RandomizedSelect( int A[], int l, int r, int k )
{
  if( l == r) return A[l];
  int q = RandomizedPartition( A, l, r );
  if( q+1 == k )     return A[q];
  else if( q+1 > k ) return RandomizedSelect( A, l, q-1, k );
  else               return RandomizedSelect( A, q+1, r, k );
}
```

Данный алгоритм `RandomizedSelect` выполняется предположительно за время $O(n)$, при этом чрезвычайно мала вероятность наихудшего случая $O(n^2)$, поскольку на каждом шаге используется случайно выбранный центральный пункт. Полный анализ предполагает оценку вероятности и ожидаемых значений. Заинтересованным читателям рекомендуется ознакомиться с другими источниками, где описан полный анализ алгоритма, например [7].

Упрощенный (нестрогий) анализ состоит в предположении о том, что алгоритм `RandomizedSelect` на каждом шаге делит массив на две части и n является степенью двойки. Следовательно, функция `RandomizedPartition` выполняется за

время $O(n)$ на первом этапе, за время $O(n/2)$ на втором этапе, за время $O(n/4)$ на третьем этапе, наконец, за время $O(1)$ на этапе с номером $1 + \log_2 n$. Стоимость алгоритма `RandomizedSelect` определяется в основном стоимостью алгоритма `RandomizedPartition`, поскольку все прочие шаги `RandomizedSelect` имеют временную сложность $O(1)$. Таким образом, общая стоимость равна $O(n + n/2 + n/4 + \ldots + n/n) = O(n \times (1/1 + 1/2 + 1/4 + \ldots + 1/n)) \leqslant O(2n) = O(n)$.

Решение с использованием стандартной библиотеки для алгоритма с предполагаемой сложностью O(n), статические данные

В стандартной библиотеке C++ STL имеется функция `nth_element`, объявленная в заголовочном файле `<algorithm>`. Функция `nth_element` содержит реализацию алгоритма с предполагаемой сложностью $O(n)$, описанную в предыдущем подразделе. По данным на 24 мая 2013 года неизвестно, существует ли эквивалент этой функции на языке Java.

Предварительная обработка O(n × log n), алгоритм O(log n), динамические данные

Все рассмотренные выше решения предполагали, что заданный массив является статическим, то есть неизменяемым при каждом запросе k-го минимального элемента. Но если содержимое массива часто изменяется, то есть добавляется новый элемент, удаляется существующий элемент или изменяется значение какого-либо существующего элемента, то все приведенные выше решения становятся неэффективными.

Если исходные данные являются динамическими, то необходимо использовать сбалансированное двоичное дерево поиска (Binary Search Tree) (см. раздел 2.3). Сначала все n элементов вставляются в сбалансированное двоичное дерево поиска за время $O(n \times \log n)$. Мы также добавляем информацию о размере каждого поддерева, корень которого расположен в каждой вершине (то есть увеличиваем дерево). При таком подходе можно найти k-й наименьший элемент за время $O(\log n)$, сравнивая k с q – размером левого поддерева данного корня:

1) если $q + 1 = k$, то текущий корень является требуемым ответом. Возвращается это значение, и процесс останавливается;
2) если $q + 1 > k$, то требуемый ответ находится в левом поддереве относительно текущего корня;
3) если $q + 1 < k$, то требуемый ответ находится в правом поддереве относительно текущего корня, и теперь выполняется поиск $(k - q - 1)$-го наименьшего элемента в этом правом поддереве. Такая корректировка значения k необходима для обеспечения правильности процесса.

Этот процесс, который похож на алгоритм с ожидаемой сложностью $O(n)$ для статической задачи поиска, можно рекурсивно повторять до тех пор, пока не будет найден требуемый ответ. Поскольку проверка размера поддерева может быть выполнена за время $O(1)$, если мы правильно выполнили увеличение сбалансированного двоичного дерева поиска, в наихудшем случае весь алгоритм в целом выполняется за время $O(\log n)$, от корня до самого глубокого листа этого дерева.

Но поскольку необходимо улучшение сбалансированного двоичного дерева поиска, то в этом алгоритме невозможно использование встроенных методов C++ STL `<map>`/`<set>` (или методов Java `TreeMap`/`TreeSet`), так как отсутствует возможность такого улучшения в библиотечном коде. Следовательно, потребуется написание собственной программы сбалансированного двоичного дерева поиска (например, АВЛ-дерева – см. рис. 9.12, красно-черного дерева и т. д. – любой вариант потребует времени для написания кода), то есть такая задача поиска наименьшего элемента с динамическими данными может стать достаточно трудной для решения.

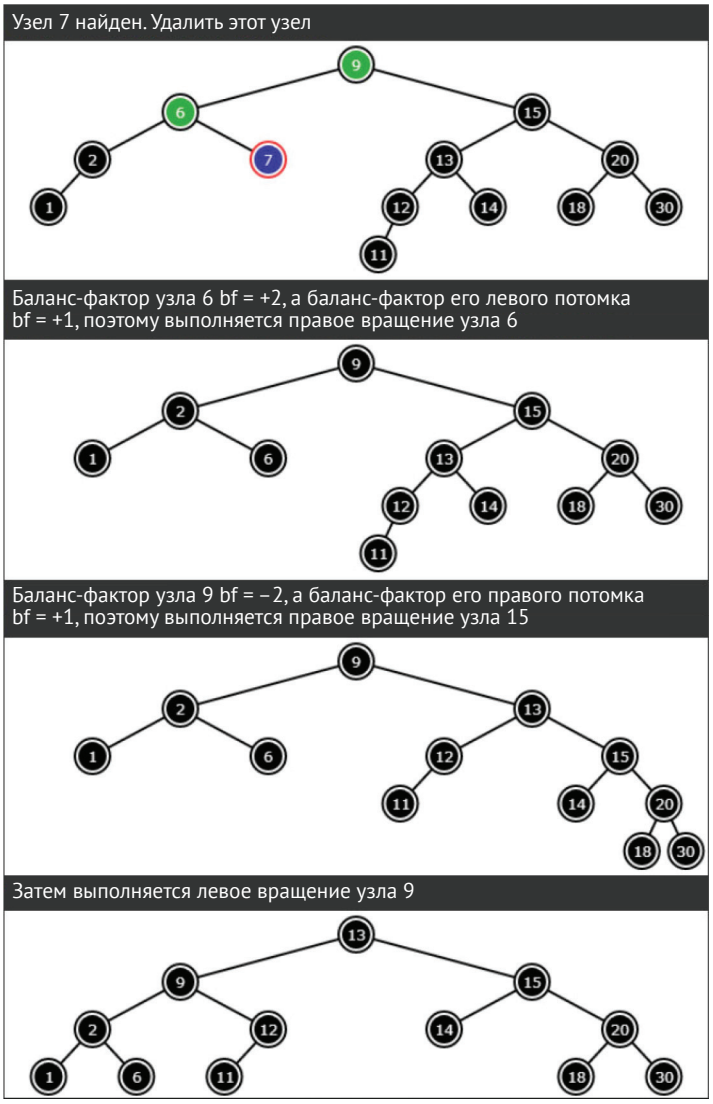


Рис. 9.12 ❖ Пример балансировки АВЛ-дерева при удалении вершины (удаление вершины 7)

Инструментальное средство визуализации:
http://www.comp.nus.edu.sg/~stevenha/visualisation/bst.html

9.30. Алгоритм ускоренного поиска кратчайшего пути

Алгоритм ускоренного поиска кратчайшего пути (Shortest Path Faster Algorithm – SPFA) – это алгоритм, использующий очередь (queue) для исключения избыточных операций в алгоритме Форда–Беллмана (Ford–Bellman). Этот алгоритм был опубликован на китайском языке Дуань Фаньдином (Duan Fanding) в 1994 году. По данным 2013 года этот алгоритм широко распространен среди китайских программистов, но не так хорошо известен в других регионах мира.

Алгоритм ускоренного поиска кратчайшего пути (SPFA) требует использования следующих структур данных:

1) графа, хранящего список смежных вершин: `AdjList` (см. раздел 2.4.1);
2) структуры `vi dist` для сохранения расстояния от источника до каждой вершины (здесь `vi` – сокращенное наименование (псевдоним) для структуры `vector<int>`);
3) очереди `queue<int>` для хранения обрабатываемых вершин;
4) структуры `vi in_queue` для пометки факта присутствия или отсутствия вершины в очереди.

Первые три структуры данных те же самые, что и в алгоритмах Дейкстры или Форда–Беллмана, рассмотренных в разделе 4.4. Четвертая структура данных используется только в алгоритме SPFA. Алгоритм ускоренного поиска кратчайшего пути можно записать в виде следующего фрагмента кода.

```
// внутренняя функция int main()
  // изначально только для S определено расстояние =0 и факт присутствия в очереди
  vi dist( n, INF ); dist[S];
  queue<int> q; q.push( S );
  vi in_queue( n, 0 ); in_queue[S] = 1;

  while( !q.empty() ) {
    int u = q.front(); q.pop(); in_queue[u] = 0;
    for( j=0; j < (int)AdjList[u].size(); j++ ) {            // все соседи (смежные узлы) u
      int v = AdjList[u][j].first, weight_u_v = AdjList[u][j].second;
      if( dist[u] + weight_u_v < dist[v] ) {                 // если можно ослабить условие
        dist[v] = dist[u] + weight_u_v;                                  // ослабить его
        if( !in_queue[v] ) {                                          // добавить в очередь
         q.push( v );                // только если эта вершина уже не находится в очереди
         in_queue[v] = 1;
        }
      }
    }
  }
```

Файл исходного кода: *UVa10986.cpp/java*

Этот алгоритм выполняется за время $O(kE)$, где k – это количество зависимостей в исследуемом графе. Максимальное значение k может быть равно V (тогда получаем равенство временно́й сложности для алгоритма Форда–Беллмана). Но мы протестировали данный алгоритм для большинства задач о поиске кратчайшего пути из одной вершины графа во все остальные (SSSP), включенных в задачи UVa, приведенные в этой книге. Алгоритм ускоренного поиска кратчайшего пути (который использует очередь) работает так же быстро, как алгоритм Дейкстры (который использует очередь с приоритетами).

Алгоритм ускоренного поиска кратчайшего пути (SPFA) может обрабатывать ребра с отрицательным весом. Если граф не содержит отрицательный цикл, SPFA успешно работает с ним. Если в графе имеется отрицательный цикл (или несколько отрицательных циклов), то SPFA также способен определить, должна ли обязательно существовать некоторая вершина (среди вершин отрицательного цикла), которая включена в очередь более $V - 1$ раз. Можно изменить приведенный выше фрагмент кода для записи количества вхождений в очередь каждой вершины. Если обнаруживается, что какая-либо вершина включена в очередь более $V - 1$ раз, то можно сделать вывод о том, что этот граф содержит отрицательный цикл (или несколько отрицательных циклов).

9.31. Метод скользящего окна

Описание задачи

Существует несколько вариантов задач с применением метода скользящего окна. Но во всех задачах основным является один и тот же принцип: сдвиг (или «скольжение» – slide) подмассива (называемого окном (window), которое может иметь статическую или динамически изменяемую длину) по линейному закону слева направо по исходному массиву из n элементов, для того чтобы вычислить некоторую величину. Ниже описаны некоторые варианты таких задач.

1. Поиск наименьшего размера подмассива (наименьшей длины окна), такого, что сумма элементов этого подмассива больше или равна заданной постоянной величине S, за время $O(n)$. Примеры:
   - для массива A_1 = {5, 1, 3, [5, 10], 7, 4, 9, 2, 8} и S = 15 ответ: 2, подмассив выделен подчеркиванием;
   - для массива A_2 = {1, 2, [3, 4, 5]} и S = 11 ответ: 3, подмассив выделен подчеркиванием.
2. Поиск наименьшего размера подмассива (наименьшей длины окна), такого, что в элементах внутреннего подмассива содержатся все целые числа из диапазона `[1..K]`. Примеры:
   - для массива A = {1, [2, 3, 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4], 5, 3, 1, 10, 3, 3} и K = 4 ответ: 13, подмассив выделен подчеркиванием;
   - для того же массива A = {[1, 2, 3], 7, 1, 12, 9, 11, 9, 6, 3, 7, 5, 4, 5, 3, 1, 10, 3, 3} и K = 3 ответ: 3, подмассив выделен подчеркиванием.
3. Поиск максимальной суммы элементов подмассива с (статическим) размером K. Примеры:

- для массива A_1 = {10, [50, 30, 20], 5, 1} и K = 3 ответ: 100 при суммировании элементов подчеркнутого подмассива;
- для массива $A2$ = {49, 70, 48, [61, 60], 60} и K = 2 ответ: 121 при суммировании элементов подчеркнутого подмассива.

4. Найти минимальное значение элемента каждого возможного подмассива с (статическим) размером K. Пример:
   - для массива A = {0, 5, 5, 3, 10, 0, 4}, n = 7 и K = 3 существует $n - K + 1$ = 7 – 3 + 1 = 5 возможных подмассивов с размером K = 3, а именно: {0, 5, 5}, {5, 5, 3}, {5, 3, 10}, {3, 10, 0} и {10, 0, 4}. Минимальные значения в каждом подмассиве: 0, 3, 3, 0, 0 соответственно.

Решения

Мы не будем рассматривать простейшие (примитивные) решения для описанных выше вариантов задач скользящего окна и сразу перейдем к решениям $O(n)$ для экономии места и времени. Рассматриваемые ниже четыре решения выполняются за время $O(n)$ при обычном «скольжении» окна по исходному массиву из n элементов, но с применением некоторых хитроумных приемов.

Для варианта 1 применим окно, которое постепенно увеличивает свой размер (добавляя текущий элемент в конец окна, то есть справа) с прибавлением значения этого текущего элемента к текущей сумме, но постепенно уменьшает свой размер (удаляя начальный элемент окна, то есть слева), как только текущая сумма становится $\geqslant S$. Наименьшая длина окна сохраняется на протяжении всего процесса и после его завершения выводится как ответ.

Для варианта 2 применяется окно, которое постепенно увеличивает свой размер, если диапазон `[1..K]` пока еще не покрыт элементами текущего окна, или постепенно уменьшает свой размер в противном случае. Наименьшая длина окна сохраняется на протяжении всего процесса и после его завершения выводится как ответ. Проверка покрытия или непокрытия диапазона `[1..K]` может быть упрощена с помощью применения одного из методов анализа частоты встречаемости. Если все целые числа $\in$ `[1..K]` имеют ненулевую частоту встречаемости, то мы можем утверждать, что диапазон `[1..K]` покрыт полностью. Рост размера окна увеличивает частоту встречаемости конкретного (включаемого) целого числа, что может привести к полному покрытию диапазона `[1..K]` (отсутствие «дыр»), в то время как сокращение размера окна уменьшает частоту встречаемости удаленного целого числа, и если частота встречаемости этого целого числа снижается до 0, то ранее считавшийся покрытым диапазон `[1..K]` теперь уже не является полностью покрытым (появилась «дыра»).

Для варианта 3 первые K целых чисел вставляются в окно, вычисляется их сумма, которая объявляется текущей максимальной суммой. Затем окно перемещается вправо с добавлением одного элемента с правой стороны и удалением одного элемента с левой стороны – так сохраняется размер окна K. К сумме прибавляется значение добавленного элемента и вычитается значение удаленного элемента. Новая полученная сумма сравнивается с текущей максимальной суммой и, если она больше, становится новой максимальной суммой. Этот процесс сдвига окна повторяется $n - K$ раз, и найденная максимальная сумма выводится как ответ.

Для варианта 4 решение является достаточно сложным, особенно если n велико. Для получения решения $O(n)$ необходимо воспользоваться структурой данных deque (double-ended queue – двухсторонняя очередь) для моделирования окна. Дело в том, что структура данных deque поддерживает эффективные $O(1)$ операции вставки и удаления элементов в начале и конце очереди (см. описание структуры данных deque в разделе 2.2). В этом варианте мы полагаем, что окно (то есть дек deque) сортируется в возрастающем порядке, то есть самый первый элемент дека deque имеет минимальное значение. Но это изменяет порядок элементов в исходном массиве. Для отслеживания присутствия или отсутствия в данный момент элемента в текущем окне также необходимо запоминать индекс каждого элемента. Подробности этого процесса более наглядно продемонстрированы в приведенном ниже исходном коде C++. Сортируемое окно может уменьшаться с обеих сторон (в начале и в конце), а также может увеличиваться с конца, следовательно, здесь необходимо применение структуры данных deque[1].

```
void SlidingWindow( int A[], int n, int K )
{
  // ii --- или пара pair<int, int> --- представляет пару (A[i], i)
  deque<ii> window;        // объект window должен быть отсортирован в возрастающем порядке
  for( int i=0; i < n; i++ ) {                                        // это алгоритм O(n)
    while( !window.empty() && window.back().first >= A[i] )
      window.pop_back(); // это позволяет всегда сохранять отсортированное состояние window

    window.push_back( ii( A[i], i ));

    // второе поле используется для проверки: является ли этот элемент частью текущего окна
    while( window.front().second <= i - K)                             // ленивое удаление
      window.pop_front();

    if( i + 1 >= K )                                      // от первого окна длины K и далее
      printf( "%d\n", window.front().first );                        // ответ для этого окна
  }
}
```

Задания по программированию

1. **UVa 01121 – Subsequence** * (скользящее окно, вариант 1)
2. **UVa 11536 – Smallest Sub-Array** * (скользящее окно, вариант 2)
3. IOI 2011 – Hottest (практическая задача; скользящее окно, вариант 3)
4. IOI 2011 – Ricehub (скользящее окно++)
5. IOI 2012 – Tourist Plan (практическая задача; еще один вариант задачи скользящего окна; наилучшее решение (ответ): начало из города 0, завершение в городе $i \in [0..N-1]$ – сумма благоприятностей самых лучших $K-i$ городов $\in [0..i]$; использовать структуру данных priority_queue (очередь с приоритетами); выводится наибольшая сумма)

[1] Следует отметить, что для ранее рассмотренных вариантов 1–3 нет никакой необходимости в использовании структуры данных deque.

9.32. Алгоритм сортировки с линейным временем работы

Описание задачи

Задан (неотсортированный) массив из n элементов. Можно ли выполнить его сортировку за время $O(n)$?

Теоретическое ограничение

В общем случае нижней границей сложности обобщенного – основанного на сравнении элементов – алгоритма сортировки является $\Omega(n \times \log n)$ (см. доказательство с использованием модели дерева решений в других источниках, например [7]). Тем не менее если n элементов массива обладают некоторым особым свойством, то возможно получение более быстрого алгоритма сортировки с линейным временем работы $O(n)$, если не применять метод сравнения элементов. Ниже рассматриваются два примера таких алгоритмов.

Решения

Сортировка подсчетом

Если массив A содержит n целых чисел в малом диапазоне `[L..R]` (например, «возраст человека» в диапазоне `[1..99]` лет, как в задании UVa 11462 – Age Sort), то можно воспользоваться алгоритмом сортировки подсчетом (counting sort). Для подробного описания этого алгоритма, приведенного ниже, используем массив `A = {2, 5, 2, 2, 3, 3}`. Алгоритм сортировки подсчетом выполняется следующим образом:

1) подготавливается «массив частот» `f` с размером `k = R-L+1` и инициализируется нулями.
   Для массива `A`, используемого в примере, `L = 2`, `R = 5` и `k = 4`;
2) выполняется один проход по массиву `A` и обновляется частота встречаемости каждого обнаруженного целого числа, то есть для каждого `i` ∈ `[0..n-1]` вычисляется `f[A[i]-L]++`.
   В рассматриваемом примере: `f[0] = 3`, `f[1] = 2`, `f[2] = 0`, `f[3] = 1`;
3) после определения частоты встречаемости каждого целого числа из заданного малого диапазона вычисляются префиксные суммы для каждого `i`, то есть `f[i] = [f-1] + f[i]` ∀`i` ∈ `[1..k-1]`. Теперь `f[i]` содержит количество элементов, меньших или равных `i`.
   В рассматриваемом примере: `f[0] = 3`, `f[1] = 5`, `f[2] = 5`, `f[3] = 6`;
4) далее выполняется обратный проход от $i = n - 1$ до $i = 0$.
   Элемент `A[i]` помещается по индексу `f[A[i]-L]-1`, так как это правильная локация для `A[i]`.
   Единица вычитается из выражения `f[A[i]-L]`, чтобы следующая копия элемента `A[i]`, если такая существует, помещалась непосредственно перед текущим элементом `A[i]`.

В рассматриваемом примере сначала помещается элемент `A[5] = 3` по индексу `f[A[5]-2]-1 = f[1]-1 = 5-1 = 4` и значение `f[1]` уменьшается на единицу, то есть становится равным 4.
Далее элемент `A[4] = 3` – то же значение, что и у элемента `A[5]=3`, – теперь помещен по индексу `f[A[4]-2]-1 = f[1]-1 = 4-1 = 3` и после вычитания единицы `f[1]` становится равным 3.
Затем элемент `A[3]` помещается по индексу `f[A[3]-2]-1 = 2`, и значение `f[0]` уменьшается на единицу и становится равным 2.
После этого повторяются еще три аналогичных шага до тех пор, пока не будет получен отсортированный массив: `{2, 2, 2, 3, 3, 5}`.

Алгоритм сортировки подсчетом имеет временну́ю сложность $O(n + k)$. Если $k = O(n)$, то этот алгоритм теоретически выполняется за линейное время без операции сравнения целых чисел. Но в условиях олимпиадного программирования k обычно не бывает слишком большим, чтобы избежать превышения лимита памяти (Memory Limit Exceeded). Например, при использовании алгоритма сортировки подсчетом может возникнуть проблема при сортировке массива `A` с $n = 3$ `{1, 1000000000, 2}`, поскольку здесь k слишком велико.

Поразрядная (цифровая) сортировка

Если массив A содержит n неотрицательных целых чисел в относительно широком диапазоне $[L..R]$, но эти числа состоят из относительно небольшого количества цифр, то можно воспользоваться алгоритмом поразрядной (цифровой) сортировки (Radix sort).

Идея алгоритма поразрядной сортировки проста. Сначала все целые числа приводятся к виду, при котором они состоят из одинакового количества d цифр, где d – наибольшее количество цифр в самом большом числе массива A: если необходимо, добавляются незначащие нули слева. Затем алгоритм поразрядной сортировки сортирует эти числа, рассматривая поочередно по одной цифре, начиная с наименьшей значимой цифры (разряда) по направлению к наибольшей значимой цифре (разряду). Также используется другой алгоритм устойчивой сортировки как подпрограмма для сортировки цифр (разрядов), например такой, как алгоритм сортировки подсчетом $O(n + k)$, рассмотренный в предыдущем подразделе. Пример:

Ввод d = 4	Дополнение нулями	Сортировка по 4-му разряду	Сортировка по 3-му разряду	Сортировка по 2-му разряду	Сортировка по 1-му разряду
323	0323	032(2)	00(1)3	0(0)13	(0)013
1257	1257	032(3)	03(2)2	1(2)57	(0)322
13	0013	001(3)	03(2)3	0(3)22	(0)323
322	0322	125(7)	12(5)7	0(3)23	(1)257

Для массива из n d-разрядных целых чисел будет выполнено $O(d)$ проходов алгоритма сортировки подсчетом, каждый из которых имеет временну́ю сложность $O(n + k)$. Таким образом, временна́я сложность алгоритма поразрядной сортировки равна $O(d \times (n + k))$. Если алгоритм поразрядной сортировки используется для обработки n 32-битовых чисел со знаком ($d \approx 10$ разрядов (цифр)) и $k = 10$, при таких условиях алгоритм поразрядной сортировки выполняется за время $O(10 \times (n + 10))$. Его все еще можно рассматривать как алгоритм

с линейным временем выполнения, но в оценке сложности слишком велик постоянный множитель.

С учетом трудностей написания сложной подпрограммы поразрядной сортировки по сравнению с вызовом стандартного метода C++ STL `sort` $O(n \times \log n)$ (или метода `Collections.sort` языка Java) рассмотренный здесь алгоритм поразрядной сортировки редко используется на олимпиадах по программированию. В этой книге мы применяли подобное сочетание алгоритмов поразрядной сортировки и сортировки подсчетом только в реализации задачи о суффиксном массиве (Suffix Array) (см. раздел 6.6.4).

Упражнение 9.32.1*. Что необходимо сделать, если мы хотим использовать алгоритм поразрядной сортировки, но массив A содержит отрицательные числа (как минимум одно, но может быть и несколько)?

Задания по программированию, связанные с применением алгоритма поразрядной сортировки

1. **UVa 11462 – Age Sort *** (стандартная задача с применением алгоритма поразрядной сортировки)

9.33. Структура данных «разреженная таблица»

В разделе 2.4.3 рассматривалась структура данных дерево отрезков (Segment tree), и мы выяснили, что эту структуру данных можно использовать для решения задачи о запросе минимального значения из диапазона (RMQ) – задачи нахождения индекса, которому соответствует минимальный элемент в диапазоне `[i..j]` заданного массива A. Требуется время $O(n)$ для предварительной обработки, чтобы сформировать дерево сегментов, а когда дерево отрезков готово к работе, каждый запрос минимального значения из диапазона выполняется за время $O(\log n)$. Применяя дерево отрезков, мы можем работать с динамической версией задачи о запросе минимального значения из диапазона, то есть при любом обновлении обрабатываемого массива обычно требуется время $O(\log n)$ для обновления соответствующей структуры дерева отрезков.

Но в некоторых задачах, включающих задачу о запросе минимального значения из диапазона, массив A после первого запроса никогда не изменяется. Этот вариант называют статической задачей о запросе минимального значения из диапазона. Несмотря на то, что дерево отрезков очевидно можно использовать для решения статической задачи о запросе минимального значения из диапазона, для этой статической версии существует альтернативное решение с использованием динамического программирования (DP) со временем предварительной обработки $O(n \times \log n)$ и временем $O(1)$ на каждый запрос RMQ. Одним из примеров такого подхода является решение задачи о наименьшем общем предке (LCA), описанное в разделе 9.18.

Основная идея решения с применением динамического программирования заключается в разделении массива A на подмассивы с длиной 2^j для каждого неотрицательного целого числа j, такого, что $2^j \leqslant n$. Мы сохраняем массив `SpT` с размером $n \times \log n$, где в элементе `SpT[i][j]` хранится индекс минимального значения из подмассива, начинающегося с индекса i и имеющего длину 2^j. Этот массив `SpT` будет разреженным, поскольку не все его элементы (ячейки) содержат значения (отсюда и название – разреженная таблица). Аббревиатура `SpT` используется для того, чтобы отличать эту структуру данных от дерева сегментов (ST).

Для создания массива `SpT` используется методика, похожая на используемую во многих алгоритмах, работающих по принципу «разделяй и властвуй», таких как сортировка слиянием. Нам известно, что в массиве с длиной 1 единственный элемент является наименьшим элементом. Это простейший случай. Для поиска индекса наименьшего элемента в массиве с размером 2^j можно сравнивать значения по индексам наименьших элементов в двух различных подмассивах с размером 2^{j-1} и запоминать индекс наименьшего из двух сравниваемых элементов. Для формирования такого массива SpT требуется время $O(n \times \log n)$. Предлагаем тщательно изучить конструктор класса `RMQ`, приведенный ниже в исходном коде, реализующем этот вариант массива `SpT`.

Легко понять, как следует обрабатывать запрос, если длина диапазона представлена степенью числа 2. Поскольку это именно та информация, которая хранится в массиве `SpT`, нужно просто возвращать соответствующий элемент этого массива. Но для вычисления результата запроса с произвольным начальным и конечным индексом необходимо передать извлеченный элемент в два подмассива меньшего размера из рассматриваемого диапазона и определить минимальное значение из двух сравниваемых. Отметим, что два сравниваемых подмассива могут перекрываться, потому что самое главное здесь – необходимость полного покрытия всего диапазона этими двумя подмассивами, а все, что остается за пределами диапазона, не имеет значения. Это всегда возможно, даже если длина подмассивов должна быть степенью числа 2. Сначала определяется длина диапазона запроса, которая равна `j-i+1`. Затем вычисляется $\log_2$ этой длины и результат округляется вниз, то есть `k` = $\lfloor$`log2(j-i+1)`$\rfloor$. Таким образом, $2^k \leqslant$ `(j-i+1)`. На рис. 9.13 показано, как могут выглядеть два подмассива, полностью перекрывающие рассматриваемый диапазон. Поскольку здесь могут возникать дополнительные подзадачи, связанные с возможным перекрытием подмассивов, этот этап решения классифицируется как динамическое программирование (DP).

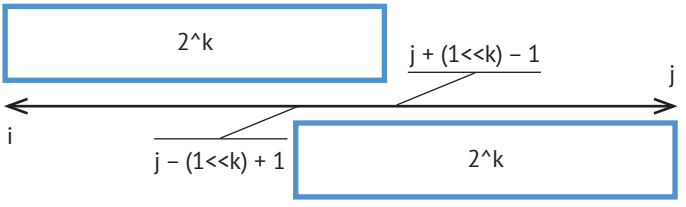


Рис. 9.13 ❖ Наглядное представление задачи о запросе минимального значения из диапазона RMQ(i, j)

Пример реализации разреженной таблицы для решения статической задачи о запросе минимального значения из диапазона (RMQ) показан ниже. Вы можете сравнить эту версию с версией реализации дерева сегментов из раздела 2.4.3.

```
#define MAX_N 1000                                  // измените это значение, если необходимо
#define LOG_TWO_N                    // 2^10 > 1000, измените это значение, если необходимо

class RMQ                // Range Minimum Query - запрос минимального значения из диапазона
{
private:
  int _A[MAX_N], SpT[MAX_N][LOG_TWO_N];
public:
  RMQ( int n, int A[] ) {    // конструктор, а также подпрограмма предварительной обработки
    for( int i=0; i < n; i++ ) {
      _A[i] = A[i];
      SpT[i][0] = i;             // RMQ в подмассиве начинается с индекса i + длина 2^0 = 1
    }
    // следующие два вложенных цикла имеют суммарную временную сложность O(n log n)
    for( int j=1; (1<<j) <= n; j++ )                    // для каждого j обязательно условие
                                                                    // 2^j <= n, O(log n)
      for( int i=0; i+(1<<j)-1 < n; i++ )                 // для каждого допустимого i, O(n)
        if( _A[SpT[i][j-1]] < _A[SpT[i+(1<<(j-1))][j-1]] )          // запрос мин. значения
                                                                    // из диапазона (RMQ)
          SpT[i][j] = SpT[i][j-1];                      // начало с индекса i, длина 2^(j-1)
        else                                  // начало с индекса i + 2^(j-1), длина 2^(j-1)
          SpT[i][j] = SpT[i+(1<<(j-1))][j-1];
  }

  int query( int i, int j ) {                       // этот запрос выполняется за время O(1)
    int k = (int)floor( log( (double)j-i+1 ) / log( 2.0 ) );              // 2^k <= (j-i+1)
    if( _A[SpT[i][k]] <= _A[SpT[j-(1<<k)+1][k]] )  return SpT[i][k];
    else                                           return SpT[j-(1<<k)+1][k];
  }
};
```

Файл исходного кода: *SparseTable.cpp/java*

Для тестового варианта, приведенного в разделе 2.4.3, с исходными данными $n = 7$ и $A = \{18, 17, 13, 19, 15, 11, 20\}$ содержимое разреженной таблицы SpT приведено в табл. 9.5.

Таблица 9.5. Содержимое разреженной таблицы

Индекс	**0**	**1**	**2**
0	0	1	2
1	1	2	2
2	2	2	5
3	3	4	5
4	4	5	пусто
5	5	5	пусто
6	6	пусто	пусто

В первом столбце значение $j = 0$ определяет запрос минимального значения из диапазона (RMQ) в подмассиве, начинающемся с индекса i и имеющем длину $2^0 = 1$. Получаем `SpT[i][j] = i`.

Во втором столбце значение $j = 1$ определяет запрос минимального значения из диапазона (RMQ) в подмассиве, начинающемся с индекса i и имеющем длину $2^1 = 2$. Отметим, что последняя строка пуста.

Во втором столбце значение $j = 2$ определяет запрос минимального значения из диапазона (RMQ) в подмассиве, начинающемся с индекса i и имеющем длину $2^2 = 4$. Отметим, что здесь пустыми являются три последние строки.

9.34. Задача о ханойских башнях

Описание задачи

Классическое описание задачи: имеются три стержня и n дисков, причем все диски различного размера (диаметра). В начальной позиции все диски надеты в порядке возрастания диаметров сверху вниз на один из стержней (стержень A). Задача – переместить все n дисков на другой стержень (стержень C). Нельзя помещать диск поверх диска меньшего диаметра. За один ход разрешено перемещать только один диск с вершины стопки дисков на одном стержне на другой стержень.

Решения

Существует простое решение возвратной рекурсией для классической задачи о ханойских башнях. Задачу перемещения n дисков со стержня A на стержень C с использованием промежуточного стержня B можно разделить на следующие три подзадачи:

1) перемещение $n - 1$ дисков со стержня A на стержень B с использованием промежуточного стержня C. После выполнения этого рекурсивного шага на стержне A остается только n-й диск (самый большой);
2) перемещение n-го диска со стержня A на стержень C;
3) перемещение $n - 1$ дисков со стержня B на стержень C с использованием промежуточного стержня A. Эти $n - 1$ дисков будут размещены поверх n-го диска, который теперь является самым нижним на стержне C.

Следует отметить, что описанные выше шаги 1 и 3 являются рекурсивными. В самом простом (базовом) случае при $n = 1$ мы просто перемещаем единственный диск с исходного на целевой стержень без использования промежуточного стержня. Пример кода реализации на C++ приведен ниже.

```
#include <cstdio>
using namespace std;

void solve( int count, char source, char destination, char intermediate )
{
  if( count == 1 )
    printf( "Move top disc from pole %c to pole %c\n", source, destination );
```

```
  else {
    solve( count-1, source, intermediate, destination );
    solve( 1, source, destination, intermediate );
    solve( count-1, intermediate, destination, source );
  }
}

int main()
{
  solve( 3, 'A', 'B', 'C' );        // попробуйте задать большее значение первого параметра
}   // return 0;
```

Минимальное количество ходов, требуемое для решения классической задачи о ханойских башнях для n дисков с использованием приведенного здесь рекурсивного решения перебором с возвратами, равно $2^n - 1$.

Задания по программированию, связанные с задачей о ханойских башнях

1. **UVa 10017 – The Never Ending... *** (классическая задача)

9.35. Замечания к главе

По состоянию на 24 мая 2013 года глава 9 содержит 34 малораспространенные темы. 10 из них являются редко применяемыми алгоритмами (выделены полужирным шрифтом). Другие 24 – редко встречающиеся задачи.

Алгоритм Диница
Алгоритм Косараджу
Алгоритм последовательного исключения переменных Гаусса
Алгоритм сортировки с линейным временем работы
Алгоритм ускоренного поиска кратчайшего пути
Алгоритм Хопкрофта–Карпа
Битоническая задача коммивояжера
Блинная сортировка
Возведение матрицы в степень
Выбор наименьшего элемента в массиве
Задача 2-SAT
Задача Иосифа Флавия
Задача китайского почтальона
Задача о картинной галерее
Задача о независимом множестве максимального веса
Задача о паре ближайших точек
Задача о порядке умножения матриц
Задача о ханойских башнях
Задача о ходе коня
Инверсия индекса массива
Максимальный поток минимальной стоимости

Метод скользящего окна
Минимальное покрытие путями в направленном ациклическом графе
Наименьший общий предок
Вершинно и реберно не пересекающиеся пути
Кратчайшее расстояние на сфере
Паросочетание в графах
Постфиксный калькулятор и преобразование выражений
Римские цифры
Ро-алгоритм Полларда для разложения на множители целых чисел
Создание магических квадратов (нечетной размерности)
Соответствие скобок
Структура данных «разреженная таблица»
Формулы или теоремы

Но даже после включения столь многих тем в третье издание этой книги мы более чем уверены в том, что существует множество других тем информатики, которые мы пока еще не рассматривали.

Эту главу и третье издание книги мы завершаем списком интересных тем, которые в конечном итоге не были включены в третье издание из-за «превышения лимита времени на написание» этой книги, ограниченного датой 24 мая 2013 года.

Существует множество других не столь известных и потому непривычных структур данных, которые редко используются на олимпиадах по программированию: фибоначчиева куча, разнообразные методики хеширования (хеш-таблицы), heavy-light decomposition корневого дерева, дерево интервалов, kD-дерево, связный список (мы сознательно избегали рассмотрения этой структуры данных в книге), сжатое префиксное дерево, range-tree дерево, список с пропусками, декартово дерево и т. д.

Тема сетевого потока (Network Flow) гораздо шире и глубже, нежели описание, приведенное в разделе 4.6 и в некоторых подразделах текущей главы. Кроме того, можно дополнить материалы по другим темам, таким как задача об исключении бейсбольных команд (Baseball Elimination), задача о циркуляции в сетевом потоке, дерево Гомори–Ху (Gomory-Hu), алгоритм проталкивания предпотока, алгоритм Штер–Вагнера (Stoer-Wagner) для решения задачи о наименьшем разрезе в неориентированных взвешенных графах и малоизвестный алгоритм Суурбалле (John W. Suurballe, 1974).

Можно было бы добавить более подробные описания еще нескольких алгоритмов в разделе 9.10, а именно: алгоритма Эдмондса (Jack Edmonds) (сжатия цветков) [13], алгоритма Гейла–Шейпли (Gale-Shapely) для решения задачи о марьяже (прочном браке) и алгоритма Куна–Манкреса (Kuhn-Munkres) (венгерский) [39, 45].

Также можно было бы добавить многие другие математические задачи и алгоритмы, например китайскую теорему об остатках, обратную величину по модулю, функцию Мебиуса, некоторые особенно трудные и малоизвестные задачи теории чисел, разнообразные численные методы и т. д.

В разделах 6.4 и 6.6 рассматривались решения с использованием алгоритма Кнута–Морриса–Пратта (КМР) и суффиксного дерева/массива для задачи поиска подстроки (совпадений в строках). Задача поиска подстроки – это глубоко

проработанная и исследованная тема, поэтому для ее решения разработаны и другие алгоритмы: Ахо–Корасика (Aho-Corasick), Боера–Мура (Boyer-Moore) и Рабина–Карпа (Rabin-Karp).

В разделе 8.2 рассматривалось несколько более усовершенствованных методов поиска. Некоторые задачи на олимпиадах по программированию являются *NP*-сложными (или *NP*-полными) задачами, но имеющими малый объем входных данных. Решением таких задач обычно является креативный полный перебор. В этой книге мы рассматривали некоторые *NP*-трудные/*NP*-полные задачи, но и эту тему можно дополнить, например задачей о раскраске графа, задачей о максимальной клике в неориентированном графе, задачей о путешествующем покупателе (Traveling Purchaser – частный случай задачи о коммивояжере) и т. д.

В заключение перечислим множество других интересных тем, которые, возможно, будут включены в последующие издания этой книги: преобразование Барроуза–Уиллера (Burrows-Wheeler transform), алгоритм Чу–Лью/Эдмондса (Chu-Liu/Edmonds), код Хаффмана (Huffman coding), алгоритм Карпа для решения задачи о циркуляции потока минимальной стоимости, методы линейного программирования, окружности Мальфатти (Malfatti circles), задача о покрытии минимальным количеством кругов, минимальное остовное дерево (с минимальным диаметром), минимальное остовное дерево с ограничением степени каждой вершины, другие библиотеки вычислительной геометрии, которые не рассматривались в главе 7, дерево оптимального бинарного поиска для иллюстрации ускорения метода динамического программирования Кнута–Яо (Knuth-Yao) [2], алгоритм вращающихся кронциркулей (Rotating Calipers), задача поиска кратчайшей общей объединяющей строки, задача Штейнера (Steiner Tree) о минимальном дереве, задача тернарного (троичного) поиска, задача-головоломка тримино (Triomino) и т. д.

Таблица 9.6

Статистические характеристики	Первое издание	Второе издание	Третье издание
Количество страниц	–	–	58
Описанные задания	–	–	15*
Задания по программированию	–	–	80

Приложение A

uHunt

uHunt (http://uhunt.felix-halim.net) – это инструмент для самостоятельного образования, размещенный на сайте архива задач университета Вальядолида (UVa) (UVa online-judge, UVa OJ [47]), созданный одним из авторов этой книги (Феликсом Халимом). Его цель – сделать решение задач по программированию в UVa OJ увлекательным занятием. Ниже мы перечислим его основные функции, помогающие в достижении этой цели.

1. Результаты проверки решений и статистическая информация по недавно отправленным решениям предоставляются практически в реальном времени, так что пользователи могут быстро исправить или улучшить свои решения (см. рис. A.1). Пользователи могут сразу увидеть рейтинг своих решений и сравнить их с другими решениями с точки зрения производительности. Значительный разрыв в значениях производительности между решением, представленным пользователем, и лучшими результатами означает, что пользователь все еще не обладает достаточными знаниями определенных алгоритмов, структур данных или специальных приемов, чтобы добиться более высокой производительности своего решения. uHunt также имеет функцию «статистического сравнения». Если у вас есть соперник (или лучший пользователь UVa, которым вы восхищаетесь), вы можете сравнить свой список решенных задач с его списком и затем попытаться решить задачи, которые может решить ваш соперник.

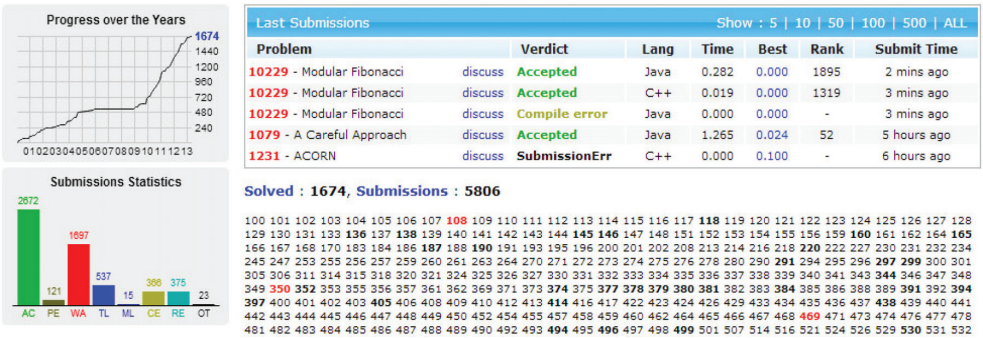


Рис. A.1 ❖ Статистические данные Стивена по состоянию на 24 мая 2013 г.

2. Веб-API для других разработчиков, позволяющий создать свой собственный инструмент. API uHunt использовался для создания полноценной системы управления олимпиадными соревнованиями, утилиты командной строки для отправки решений и получения обратной связи через консоль и мобильного приложения для просмотра статистических данных.
3. Средство взаимопомощи пользователей. Виджет чата в правом верхнем углу страницы используется для обмена идеями и позволяет пользователям помогать друг другу в решении задач. Это создает благоприятную среду для обучения, где пользователь всегда может обратиться за помощью.

Volume : ALL View : [unsolved | solved | both] Show : [25 | 50 | 100]

No	Number	Problem Title		nos	anos	%anos	dacu	best
1	705	Slash Maze	discuss	4662	1898	40%	1078	0.000
2	10254	The Priest Mathematician	discuss	3810	1670	43%	843	0.004
3	10202	Pairsumonious Numbers	discuss	3012	1187	39%	836	0.000
4	134	Loglan-A Logical Langu...	discuss	3304	892	26%	736	0.000
5	132	Bumpy Objects	discuss	3083	1241	40%	728	0.000
6	254	Towers of Hanoi	discuss	5884	1081	18%	723	0.008
7	704	Colour Hash	discuss	3593	1538	42%	699	0.008
8	302	John's trip	discuss	7031	1418	20%	648	0.006
9	10776	Determine The Combin...	discuss	1878	838	44%	618	0.000

Volumes

v1	72%
v2	42%
v3	61%
v4	75%
v5	58%
v6	58%
v7	44%
v8	40%
v9	40%
v10	15%
v11	18%
v12	52%

Рис. А.2 ❖ Поиск следующих по уровню сложности, но все еще простых задач с использованием параметра «dacu»

4. Выбор следующих задач для решения, упорядоченный по возрастающей сложности (аппроксимируется количеством принятых решений для различных задач). Это полезно для пользователей, которые хотят решать задачи, сложность которых соответствует их текущему уровню подготовки. Это пригодится начинающим: если пользователь – все еще новичок и ему нужно укрепить уверенность в своих силах, то ему следует решать задачи с постепенно возрастающим уровнем сложности. Такой способ подготовки гораздо лучше, нежели многочисленные неудачные попытки решать сложные задачи, для которых не удается получить оценку «зачтено» (AC), не зная, что не так в решении. 149 008 пользователей UVa фактически предоставляют статистические данные по каждой задаче, и эту информацию можно использовать для приблизительной оценки сложности задач. Более легкие задачи будут иметь большее число отправленных решений и большее количество положительных оценок (AC). Однако, поскольку пользователь UVa по-прежнему может отправлять решения для тех задач, для которых он(а) уже получил(а) положительную оценку (AC), то одно лишь количество задач, получивших оценку AC, не является точной метрикой, позволяющей определить, является ли данная задача легкой. Предположим, например, что есть серьезная задача, которую пытается решить один хороший программист, который отправляет 50 вариантов решения, получающих оценку AC лишь для того, чтобы улучшить производительность своего решения. Эта задача не легче, чем другая (более простая) задача, которую успешно решат 49 различных пользователей и получат оценку AC. Чтобы решить эту проблему,

стандартным критерием сортировки в uHunt является параметр «dacu» (*distinct* accepted users), что означает «уникальные пользователи, отправившие правильное решение и получившие положительную оценку». Сложная задача в приведенном выше примере имеет dacu = 1, тогда как более простая задача имеет dacu = 49 (см. рис. A.3).

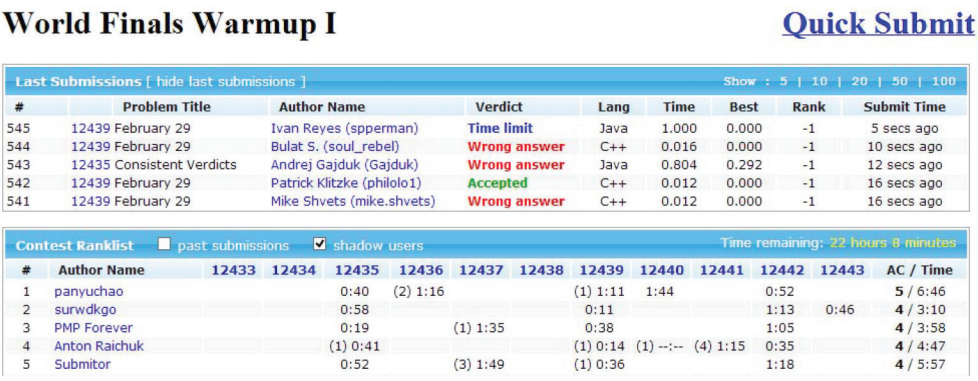


Рис. A.3 ❖ Мы можем вернуться к прошедшим соревнованиям с помощью «виртуального соревнования»

5. Средства для создания виртуальных соревнований. Несколько пользователей могут захотеть провести закрытый турнир среди них по некоторой выборке задач, установив определенную продолжительность соревнований. Такие «закрытые состязания» полезны для команды, а также подходят для индивидуальных тренировок. В некоторых соревнованиях есть «призраки» (то есть участники прошлых соревнований), так что пользователи могут сравнить свои навыки с навыками реальных участников в прошлом.
6. В этой книге собрано воедино ≈ 1675 задач по программированию, относящихся к различным категориям (см. рис. A.4). Пользователи могут отслеживать, какие из задач, упомянутых в данной книге, они выполнили, и видеть ход своей работы. Эти задачи по программированию можно использовать даже без книги. Теперь пользователь может настроить свою программу обучения для решения *задач, относящихся к определенным категориям*. Без такой (ручной) категоризации трудно реализовать подобный режим обучения. Мы также помечаем звездочками (*) задачи, которые рассматриваем как обязательные * (до трех задач в одной категории).

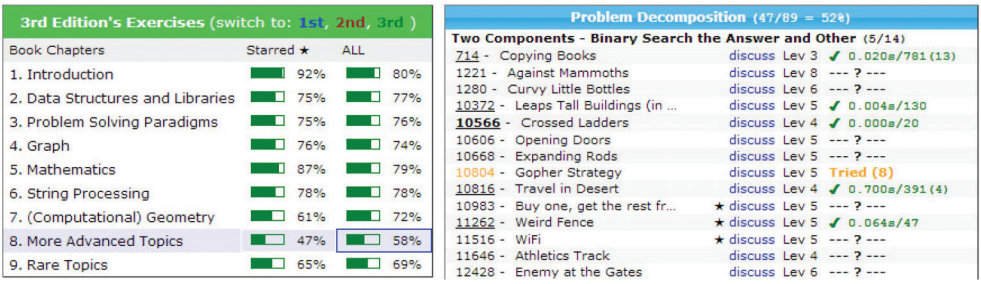


Рис. А.4 ❖ Задачи по программированию, опубликованные в этой книге, содержатся в uHunt

Создание такого веб-инструмента, как uHunt, – весьма непростая задача. На вход поступает более 11 796 315 запросов на проверку решенной задачи от ≈ 149 008 пользователей (примерно одно событие отправки решения на проверку каждые несколько секунд). Статистика и рейтинги должны обновляться часто, и такое обновление должно выполняться быстро. Для решения этой проблемы Феликс использует множество продвинутых структур данных (некоторые из них выходят за рамки этой книги), например метод «Database cracking» [29], дерево Фенвика, сжатие данных и т. д.

Рис. А.5 ❖ Успехи Стивена и Феликса в решении задач из архива университета Вальядолида (UVa) (с 2000 г. по настоящее время)

Мы сами широко используем этот инструмент на разных этапах нашей жизни, как это видно по рис. А.5. На наших графиках прогресса можно увидеть две основные вехи: интенсивное обучение Феликса, которое в итоге привело к победе на олимпиаде ACM ICPC в Гаосюне (2006) Феликса и его команду ICPC (см. рис. А.6), и интенсивное решение задач Стивеном в последние четыре года (конец 2009 – настоящее время), что помогло в подготовке этой книги.

Рис. A.6 ❖ Андриан, Феликс и Андоко
стали победителями ACM ICPC в Гаосюне в 2006 г.

Приложение B

Благодарности

Задачи, обсуждаемые в этой книге, в основном взяты с сайта архива задач университета Вальядолида UVa [47], из архива задач ACM ICPC Live Archive [33] и материалов прошедших соревнований IOI (в основном 2009–2012 гг.). Мы связывались со следующими авторами, чтобы получить их разрешения (список приводится в алфавитном порядке; их принадлежность к соответствующим организациям указывается на момент 2013 г.):

1) Brian C. Dean (Брайан К. Дин, Университет Клемсона, Америка);
2) Colin Tan Keng Yan (Колин Тан Кенг Ян, Национальный университет Сингапура, Сингапур);
3) Дерек Кисман (Университет Ватерлоо, Канада);
4) Gordon V. Cormack (Гордон В. Кормак, Университет Ватерлоо, Канада);
5) Howard Cheng (Говард Ченг, Университет Летбридж, Канада);
6) Jane Alam Jan (Джейн Алам Ян, Google);
7) Jim Knisely (Джим Найзли, Университет Боба Джонса, Америка);
8) Jittat Fakcharoenphol (Джиттат Факчароенфол, Университет Касецарт, Таиланд);
9) Manzurur Rahman Khan (Манзурур Рахман Хан, Google);
10) Melvin Zhang Zhiyong (Мелвин Чжан Чжиюн) (Национальный университет Сингапура, Сингапур);
11) Michal (Misof) Forišek (Михал (Мисоф) Форишек, Университет Комениуса, Словакия);
12) Mohammad Mahmudur Rahman (Мохаммад Махмудур Рахман, Университет Южной Австралии, Австралия);
13) Norman Hugh Anderson (Норман Хью Андерсон, Национальный университет Сингапура, Сингапур);
14) Ondřej Lhoták (Ондрей Лотак, Университет Ватерлоо, Канада);
15) Petr Mitrichev (Петр Митричев, Google);
16) Piotr Rudnicki (Петр Рудницкий, Университет Альберты, Канада);
17) Rob Kolstad (Роб Колстад, компьютерная олимпиада США);
18) Rujia Liu (Руцзя Лю, Университет Цинхуа, Китай);
19) Shahriar Manzoor (Шахриар Мансур, Юго-восточный университет, Бангладеш);
20) Sohel Hafiz (Сохель Хафиз, Техасский университет в Сан-Антонио, Америка);
21) Soo Yuen Jien (Су Юн Цзень, Национальный университет Сингапура, Сингапур);

22) Tan Sun Teck (Тан Сун Тек, Национальный университет Сингапура, Сингапур);
23) TopCoder, Inc (за задачу PrimePairs в разделе 4.7.4).

Ниже мы приводим подборку фотографий некоторых из этих авторов, с которыми нам удалось встретиться лично.

Однако в этой книге перечислены и рассмотрены тысячи (≈ 1675) задач, со множеством авторов которых нам пока не удалось связаться. Если вы являетесь авторами этих задач или знаете человека, задачи которого используются в данной книге, мы просим вас сообщить нам об этом. Мы публикуем обновленную версию этого раздела книги на нашем веб-сайте: https://sites.google.com/site/stevenhalim/home/credits.

Список используемой литературы

[1] Ahmed Shamsul Arefin. *Art of Programming Contest* (*Ахмед Шамсул Арефин.* Конкурс «Искусство программирования») (*со старого сайта Стивена*). Gyankosh Prokashoni (доступен онлайн), 2006.

[2] Wolfgang W. Bein, Mordecai J. Golin, Lawrence L. Larmore, and Yan Zhang. The Knuth-Yao Quadrangle-Inequality Speedup is a Consequence of Total-Monotonicity. (*Вольфганг В. Бейн, Мордехай Дж. Голин, Лоуренс Л. Лармор и Ян Чжан.* Ускорение алгоритма Кнута–Яо для неравенства четырехугольника как следствие полной монотонности). *ACM Transactions on Algorithms*, 6 (1): 17, 2009.

[3] Richard Peirce Brent. An Improved Monte Carlo Factorization Algorithm (*Ричард Пирс Брент.* Усовершенствованный алгоритм факторизации Монте-Карло). *BIT Numerical Mathematics*, 20 (2): 176–184, 1980.

[4] Brilliant: https://brilliant.org/.

[5] Frank Carrano. *Data Abstraction and Problem Solving with C++: Walls and Mirrors* (*Фрэнк Каррано.* Абстракция данных и решение задач с помощью C ++: стены и зеркала). Addison Wesley, 5-е изд., 2006.

[6] Yoeng-jin Chu and Tseng-hong Liu. On the Shortest Arborescence of a Directed Graph (*Ён-чжин Чу и Цен-хон Лю.* О наикратчайшем ориентированном дереве ориентированного графа). *Science Sinica*, 14: 1396–1400, 1965.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm* (*Томас Х. Кормен, Чарльз Лейзерсон, Рональд Л. Ривест и Клифф Стейн.* Введение в алгоритм). MIT Press, 2-е изд., 2001.

[8] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms* (*Санджой Дасгупта, Христос Пападимитриу и Умеш Вазирани.* Алгоритмы). McGraw Hill, 2008.

[9] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational Geometry: *Algorithms and Applications* (*Марк де Берг, Марк ван Кревельд, Марк Овермарс и Отфрид Чонг Шварцкопф.* Вычислительная геометрия: алгоритмы и приложения). Springer, 2-е изд., 2000.

[10] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs (*Эдсгер Вайб Дейкстра.* Заметка о двух задачах, связанных с графами). *Numerische Mathematik*, 1: 269–271, 1959.

[11] *Ефим Диниц.* Алгоритм решения задачи о максимальном потоке в сети со степенной оценкой. Докл. Академии наук СССР, 11: 1277–1280, 1970.

[12] Adam Drozdek. *Data structures and algorithms in Java* (*Адам Дроздек.* Структуры данных и алгоритмы на Java). Cengage Learning, 3-е изд., 2008.

[13] Jack Edmonds. Paths, trees, and flowers (*Джек Эдмондс.* Пути, деревья и цветы). *Canadian Journal on Maths*, 17: 449–467, 1965.

[14] Jack Edmonds and Richard Manning Karp. Theoretical improvements in algorithmic efficiency for network flow problems. (*Джек Эдмондс и Ричард Мэннинг Карп*. Теоретическое обоснование улучшения алгоритмической эффективности для задач сетевого потока). *Journal of the ACM*, 19 (2): 248–264, 1972.

[15] Susanna S. Epp. *Discrete Mathematics with Applications* (*Сюзанна С. Эпп*. Дискретная математика и ее приложения). Брукс-Коул, 4-е изд., 2010.

[16] Fabian Ernst, Jeroen Moelands, and Seppo Pieterse. Teamwork in Prog Contests: 3 * 1 = 4 (*Фабиан Эрнст, Йерун Меландс и Сеппо Питерс*. Работа в команде на соревнованиях по программированию: 3 * 1 = 4) // http://xrds.acm.org/article.cfm?aid=332139.

[17] Проект «Эйлер»: http://projecteuler.net/.

[18] Peter M. Fenwick. A new data structure for cumulative frequency tables (*Питер М. Фенвик*. Новая структура данных для кумулятивных таблиц частот). *Software: Practice and Experience*, 24 (3): 327–336, 1994.

[19] Robert W. Floyd. Algorithm 97: Shortest Path (*Роберт В. Флойд*. Алгоритм 97: Кратчайший путь). *Communications of the ACM*, 5 (6): 345, 1962.

[20] Michal Forišek. IOI Syllabus (*Михал Форишек*. Программа IOI) // http://people.ksp.sk/~misof/ioi-syllabus/ioi-syllabus-2009.pdf.

[21] Michal Forišek. The difficulty of programming contests increases (*Михал Форишек*. Возрастающая сложность соревнований по программированию. Доклад на конференции *International Conference on Informatics in Secondary Schools* (Международной конференции по информатике в средних школах), 2010.

[22] William Henry, Gates and Christos Papadimitriou. Bounds for Sorting by Prefix Reversal (*Уильям Генри, Гейтс и Христос Пападимитриу*. Границы для сортировки изменением префикса). Discrete Mathematics, 27: 47–57, 1979.

[23] Felix Halim, Roland Hock Chuan Yap, and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs (*Феликс Халим, Роланд Хок Чуан Яп и Юнчжэн Ву*. Основанный на MapReduce алгоритм нахождения максимального потока для больших графов сетей «тесного мира») // *ICDCS*, 2011.

[24] Steven Halim and Felix Halim. Competitive Programming in National University of Singapore (*Стивен Халим и Феликс Халим*. Олимпиадное программирование в Национальном университете Сингапура) // *A new learning paradigm: competition supported by technology*. Ediciones Sello Editorial S. L., 2010, 2010.

[25] Steven Halim, Roland Hock Chuan Yap, and Felix Halim. Engineering SLS for the Low Autocorrelation Binary Sequence Problem (*Стивен Халим, Роланд Хок Чуан Яп и Феликс Халим*. Реализация пошагового линейного поиска для задачи двоичной последовательности с низкой автокорреляцией) // *Constraint Programming*, стр. 640–645, 2008.

[26] Steven Halim, Roland Hock Chuan Yap, and Hoong Chuin Lau. An Integrated White+Black Box Approach for Designing & Tuning SLS (*Стивен Халим, Роланд Хок Чуан Яп и Хунг Чуин Лау*. Интегрированный подход «белый + черный ящик» для проектирования и настройки пошагового линейного поиска) // *Constraint Programming*, стр. 332–347, 2007.

[27] Steven Halim, Koh Zi Chun, Loh Victor Bo Huai, and Felix Halim. Learning Algorithms with Unified and Interactive Visualization (*Стивен Халим, Ко Зи Чун, Лох Виктор Бо Хуай и Феликс Халим*. Алгоритмы обучения с унифицированной и интерактивной визуализацией). Olympiad in Informatics, 6: 53–68, 2012.

[28] John Edward Hopcroft and Richard Manning Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs (*Джон Эдвард Хопкрофт и Ричард Мэннинг Карп*. Алгоритм $n^{5/2}$ для максимальных совпадений в двудольных графах) // SIAM Journal on Computing, 2 (4): 225–231, 1973.

[29] Stratos Idreos. Database Cracking: Towards Auto-tuning Database Kernels (*Стратос Идреос*. Взлом базы данных: на пути к автоматической настройке ядер баз данных). Докторская диссертация, CWI and University of Amsterdam, 2010.

[30] Учебники по алгоритмам TopCoder Inc.: http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static.

[31] TopCoder Inc. PrimePairs. Copyright 2009 TopCoder, Inc. Все права защищены: http://www.topcoder.com/stat?c=problem_statement&pm=10187&rd=13742.

[32] TopCoder Inc. Single Round Match (SRM): http://www.topcoder.com/tc.

[33] Competitive Learning Institute. ACM ICPC Live Archive (Институт олимпиадного программирования. Архив ACM ICPC): http://livearchive.onlinejudge.org/.

[34] IOI. Международная олимпиада по информатике // http://ioinformatics.org.

[35] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time (*Джузеппе Ф. Итальяно, Луиджи Лаура и Федерико Сантарони*. Поиск сильных мостов и сильных точек сочленения за линейное время). *Combinatorial Optimization and Applications*, 6508: 157–169, 2010.

[36] Arthur B. Kahn. Topological sorting of large networks (*Артур Б. Кан*. Топологическая сортировка крупных сетей). *Communications of the ACM*, 5 (11): 558–562, 1962.

[37] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted Longest-Common-Prefix Array (*Юха Кярккайнен, Джованни Манзини и Саймон Дж. Пуглиси*. Массив с самым длинным общим префиксом) // *CPM, LNCS 5577*, стр. 181–192, 2009.

[38] Jon Kleinberg and Eva Tardos. *Algorithm Design*. (*Джон Кляйнберг и Ева Тардос*. Разработка алгоритмов). Addison Wesley, 2006.

[39] Harold W. Kuhn. The Hungarian Method for the assignment problem (*Гарольд В. Кун*. Венгерский метод решения задач о назначениях). *Naval Research Logistics Quarterly*, 2: 83–97, 1955.

[40] Anany Levitin. *Introduction to The Design & Analysis of Algorithms* (*Ананий Левитин*. Введение в разработку и анализ алгоритмов). Addison Wesley, 2002.

[41] Rujia Liu. Algorithm Contests for Beginners (In Chinese) (*Руджа Лю*. Конкурс алгоритмов для начинающих (на китайском языке)). Tsinghua University Press, 2009.

[42] Rujia Liu and Liang Huang. The Art of Algorithms and Programming Contests (In Chinese) (*Руджа Лю и Лян Хуан*. Конкурс «Искусство алгоритмов и программирования» (на китайском языке)). Tsinghua University Press, 2003.

[43] Udi Manbers and Gene Myers. Suffix arrays: a new method for on-line string searches (*Уди Манберс и Джин Майерс*. Суффиксные массивы: новый метод поиска строк в сети). *SIAM Journal of Computing*, 22 (5): 935–948, 1993.
[44] Gary Lee Miller. Riemann's Hypothesis and Tests for Primality (*Гари Ли Миллер*. Гипотеза Римана и проверка чисел на простоту). *Journal of Computer and System Sciences*, 13 (3): 300–317, 1976.
[45] James Munkres. Algorithms for the Assignment and Transportation Problems (*Джеймс Мункрес*. Алгоритмы для задач назначения и транспортных задач // *Journal of the Society for Industrial and Applied Mathematics*, 5 (1): 32–38, 1957.
[46] Институт математики и информатики Литвы. Олимпиады по информатике: http://www.mii.lt/olympiads_in_informatics/.
[47] Университет Вальядолида. Онлайн-арбитр: http://uva.onlinejudge.org.
[48] США. Олимпиада по информатике. Программа обучения USACO Gateway: http://train.usaco.org/usacogate.
[49] Joseph O'Rourke. *Art Gallery Theorems and Algorithms* (*Джозеф О'Рурк*. Теоремы и алгоритмы в задаче о картинной галерее). Oxford University Press, 1987.
[50] Joseph O'Rourke. *Computational Geometry in C* (*Джозеф О'Рурк*. Вычислительная геометрия на языке C). Cambridge University Press, 2-е изд., 1998.
[51] David Pearson. A polynomial-time algorithm for the change-making problem (*Дэвид Пирсон*. Алгоритм полиномиального времени для задачи размена монет). *Operations Research Letters*, 33 (3): 231–234, 2004.
[52] John M. Pollard. A Monte Carlo Method for Factorization (*Джон М. Поллард*. Метод Монте-Карло в факторизации) // *BIT Numerical Mathematics*, 15 (3): 331–334, 1975.
[53] George Pólya. How to Solve It (*Джордж Поля*. Как решать задачи). Princeton University Press, 2-е изд., 1957.
[54] Janet Prichard and Frank Carrano. *Data Abstraction and Problem Solving with Java: Walls and Mirrors* (*Джанет Причард и Фрэнк Каррано*. Абстракция данных и решение задач с использованием Java: стены и зеркала). Addison Wesley, 3-е изд., 2010.
[55] Michael Oser Rabin. Probabilistic algorithm for testing primality (*Майкл Осер Рабин*. Вероятностный алгоритм проверки простоты) // Journal of Number Theory, 12 (1): 128–138, 1980.
[56] Kenneth H. Rosen. *Elementary Number Theory and its Applications* (*Кеннет Х. Розен*. Элементарная теория чисел и ее приложения). Addison Wesley Longman, 4-е изд., 2000.
[57] Kenneth H. Rosen. *Discrete Mathematics and its Applications* (*Кеннет Х. Розен*. Дискретная математика и ее приложения). McGraw-Hill, 7-е изд., 2012.
[58] Robert Sedgewick. *Algorithms in C++, Part 1-5* (*Роберт Седжвик*. Алгоритмы на C ++, части 1–5). Addison Wesley, 3-е изд., 2002.
[59] Steven S. Skiena. *The Algorithm Design Manual* (*Стивен С. Шиена*. Руководство по разработке алгоритмов). Springer, 2008.
[60] Steven S. Skiena and Miguel A. Revilla. *Programming Challenges* (*Стивен С. Шиена и Мигель А. Ревилла*. Проблемы программирования). Springer, 2003.
[61] SPOJ. Онлайн-арбитр «Сфера»: http://www.spoj.pl/.

[62] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction* (*Вин-Кин Сунг*. Алгоритмы в биоинформатике: практическое введение). CRC Press (Taylor & Francis Group), 1-е изд., 2010.
[63] Robert Endre Tarjan. Depth-first search and linear graph algorithms (*Роберт Андре Тарьян*. Алгоритмы поиска в глубину и линейные графы) // *SIAM Journal of Computing*, 1 (2): 146–160, 1972.
[64] Jeffrey Trevers and Stanley Milgram. An Experimental Study of the Small World Problem (*Джеффри Треверс и Стэнли Милгрэм*. Экспериментальное исследование проблемы «тесного мира») // *Sociometry*, 32 (4): 425–443, 1969.
[65] Esko Ukkonen. On-line construction of suffix trees (*Эско Укконен*. Онлайн-построение суффиксных деревьев). *Algorithmica*, 14 (3): 249–260, 1995.
[66] Бэйлорский университет. Международная студенческая олимпиада по программированию ACM // http://icpc.baylor.edu/icpc.
[67] Tom Verhoeff. 20 Years of IOI Competition Tasks (*Том Верхофф*. 20 лет соревнований IOI) // *Olympiads in Informatics*, 3: 149–166, 2009.
[68] Adrian Vladu and Cosmin Negru¸seri. Suffix arrays – a programming contest approach (*Адриан Владу и Космин Негруссери*. Суффиксные массивы – подход к решению задач на олимпиадах по программированию) // *GInfo*, 2005.
[69] Henry S. Warren. *Hacker's Delight*. (*Генри С. Уоррен*. Восторг хакера). Pearson, 1-е изд., 2003.
[70] Stephen Warshall. A theorem on Boolean matrices (*Стивен Уоршелл*. Теорема о булевых матрицах) // *Journal of the ACM*, 9 (1): 11–12, 1962.
[71] Википедия. Свободная энциклопедия // http://en.wikipedia.org.

Предметный указатель

Б

В

Г

Н

О

П

P

C

Т

У

Ф

Х