Антти Лааксонен

Олимпиадное программирование

Изучение и улучшение алгоритмов на соревнованиях

2-е издание, обновленное и дополненное

Guide to Competitive Programming

Learning and Improving Algorithms Through Contests

Second Edition

Antti Laaksonen

Springer

Олимпиадное программирование

Изучение и улучшение алгоритмов на соревнованиях

2-е издание, обновленное и дополненное

Антти Лааксонен

ДМК Пресс
издательство
Москв , 2020

УДК 004.02: 004.424
ББК 22.18
Л12

Л12 Антти Л ксонен

Олимпи дное прогр ммиров ние. 2-е изд., обновленное и дополненное / пер. с нгл. А. А. Слинкин – М.: ДМК Пресс, 2020. – 328 с.: ил.

ISBN 978-5-97060-878-4

Перед в ми второе, обновленное изд ние книги, котор я уже успел полюбиться чит телям. Автор подробно описыв ет, к к проходят олимпи ды по прогр ммиров нию и к к к ним готовиться, р збир ет б зовые темы, трюки и лгоритмы. В новых р здел х р ссм трив ются темы повышенного уровня: вычисление преобр зов ния Фурье, н хождение потоков миним льной стоимости в гр ф х и использов ние конечных втом тов в з д ч х о строк х.

Спортивное прогр ммиров ние – с мый перспективный интеллекту льный вид спорт ; уже сейч с им увлек ются лучшие умы пл неты, и число уч стников р стет год от год . Рост популярности олимпи дного прогр ммиров ния положительно влияет н другие сферы жизнедеятельности человек . Н выки быстрого решения сложнейших з д ч помогут сегодняшним студент м в будущем эффективно спр вляться с ре льными проблем ми человечеств .

Изд ние будет полезно студент м ф культетов информ ционных технологий и уч стник м олимпи д по прогр ммиров нию.

УДК 004.02: 004.424
ББК 22.18

Original English language edition published by Springer International Publishing AG. Copyright ©Springer International Publishing AG, part of Springer Nature 2020. All rights reserved. This edition has been translated and published under licence from Springer International Publishing AG. Russian-language edition copyright © 2020 by DMK Press. All rights reserved.

Все пр в з щищены. Люб я ч сть этой книги не может быть воспроизведен в к кой бы то ни было форме и к кими бы то ни было средств ми без письменного р зрешения вл дельцев вторских пр в.

М тери л, изложенный в д нной книге, многокр тно проверен. Но, поскольку вероятность технических ошибок все р вно существует, изд тельство не может г р нтиров ть бсолютную точность и пр вильность приводимых сведений. В связи с этим изд тельство не несет ответственности з возможные ошибки, связ нные с использов нием книги.

ISBN 978-3-030-39357-1 (нгл.)
ISBN 978-5-97060-878-4 (рус.)

© Springer Nature Switzerland AG, 2020
© Оформление, перевод н русский язык, изд ние, ДМК Пресс, 2020

Оглавление

От автора

I hope this Russian edition of my book will be useful to future competitive programmers in Russia. I appreciate the competitive programming culture in Russia, especially the international training camps which are known for challenging and high-quality problems. Indeed,competitive programming is a way to learn algorithms together with people from different countries. It does not matter where you come from – everybody is interested in creating efficient algorithms, which is the core of computer science.

Antti Laaksonen

Я надеюсь, что русское издание моей книги будет полезно будущим спортивным программистам России. Я ценю российскую культуру олимпиадного программирования, а в особенности международные учебные кэмпы, которые известны своими сложными и интересными задачами. Так спортивное программирование объединяет людей из разных стран в изучении алгоритмов. Не важно, откуда вы, важна ваша заинтересованность в создании эффективных алгоритмов, которые являются основой Computer Science.

Антти Лааксонен

Вступительное слово Алексея Малеева, основателя Moscow Workshops

Про спортивное программирование в России знают мало. А между тем именно российские команды ежегодно выигрывают самое престижное мировое соревнование по программированию для студентов – International Collegiate Programming Contest (ICPC). История участия студентов из российских вузов на старейшем чемпионате ICPC отсчитывается с далекого 1993 года. В 2000 году студенты Санкт-Петербургского государственного университета показали необычайный прогресс и впервые среди российских команд вырвались в чемпионы мира. С этого года российские команды завоевали 33 золотые медали. Для сравнения: студенты из Китая всего 13 раз удостаивались золота за этот период, европейские участники – 12, США – всего 7. В мире наиболее сильную подготовку демонстрируют российские, китайские команды, студенты из Азии и Польши.

Секрет успеха наших команд во многом объясняется усиленными тренировками. Но важна система, в рамках которой происходит подготовка. И здесь именно России удалось выстроить сильную тренировочную струк-

туру для олимпиадных программистов. Чемпионов готовят ведущие вузы страны, технические и не только: Университет ИТМО, Московский физико-технический институт, Санкт-Петербургский государственный университет, Московский государственный университет им. М.В. Ломоносова, Уральский федеральный университет, Саратовский университет и др.

Объединившись вместе, консорциум из вышеперечисленных вузов в 2012 году запустил первый глобальный проект по подготовке студентов к чемпионату по спортивному программированию на кампусе МФТИ – **Moscow Workshops**. В формате учебно-тренировочных сборов студенты решают контексты и разбирают их с лучшими тренерами. Важная компонента **Moscow Workshops** – это интернациональность. Кэмпы открывают свои двери для молодых участников всего мира, предлагая им не только учиться, но и путешествовать, расширять свой кругозор. На каждом воркшопе участники обогащают свои знания о других странах – посещают экскурсии, знакомятся с местной культурой и людьми.

Уровень подготовки воркшопов очень высокий. Можно сказать, что те, кто прошел обучение в **Moscow Workshops** и успешно прошел отбор начальных этапов ICPC, уже представляют элиту мира программирования. За этими ребятами охотятся крупнейшие IT-компании, они получают лучшие предложения о работе.

В 2016 году по образовательной франшизе мы запустили тренировочный лагерь в Гродно (Западная Белоруссия). С тех пор мы открыли регулярно действующие площадки в Барселоне (Испания) и Коллам (Индия), а в этом году запускаем воркшоп в одном из самых восточных городов России – во Владивостоке. На данный момент в сборах уже приняли участие команды 240 университетов из 60 стран Европы, Азии, Южной и Северной Америки, Африки и Австралии. В 2019 году 11 из 12 команд, завоевавших медали в финале чемпионата ICPC, принимали участие в подготовительных сборах **Moscow Workshops**.

Спортивное программирование – это самый перспективный интеллектуальный вид спорта, который можно назвать шахматами будущего. Уже сейчас им увлекаются лучшие умы планеты, и число участников растет год от года. Рост популярности олимпиадного программирования положительно влияет на другие сферы жизнедеятельности человека. Навыки быстрого решения сложнейших задач помогают сегодняшним студентам в будущем справляться с реальными проблемами человечества креативно и эффективно. В ходе соревнований ребята учатся справляться со сложными моральными и психологическими нагрузками. За это время у них вырабатывается навык управлять рисками, ведь, чтобы выйти в финал ICPC, нужно потратить около 5 лет. Не каждому это суждено. Стрессоустойчивость и нацеленность на результат – это те качества, которые необходимы технологическим предпринимателям для запуска собственных проектов.

Система **Moscow Workshops** также включает онлайн-курсы на платфор-

ме Coursera и онлайн-чемпионаты Opencup.ru, а также сборы для школьников по подготовке к IOI – Moscow Workshops Juniors, что в совокупности дает любому студенту из любого уголка мира, вне зависимости от принадлежности к ведущим мировым университетам, реализоваться в области алгоритмов и программирования.

Студентов, которые горят желанием развиваться в программировании и «кодят», не замечая хода времени, мы ждем на кэмпах **Moscow Workshops** и желаем всегда стремиться к результатам, которые кажутся невозможными.

С уважением, Алексей Малеев,
проректор МФТИ,
основатель Moscow Workshops

Отзыв Дмитрия Гришина, основателя Mail.Ru Group

В мои студенческие годы всегда было много программирования, и я часто участвовал в различных олимпиадах, но только теперь понимаю, насколько мне тогда не хватало подобной книги. Всем, кто стремится покорить олимп международных чемпионатов по программированию, она точно придется по вкусу и поможет на этом нелегком пути достижения цели. Мы в Mail.Ru Group тоже создали несколько чемпионатов по программированию, которые за последние 7 лет стали популярны не только в России, но и далеко за рубежом – уже более 20 стран принимает участие и больше 175 тысяч человек соревновались в Russian Code Cup, VK Cup, Russian AI Cup, ML BootCamp и др. Уверен, что эта книга поможет и тем, кто собирается принять в них участие тоже.

Дмитрий Гришин,
основатель, соучредитель и председатель совета директоров Mail.Ru Group,
основатель венчурного фонда Grishin Robotics

Благодарность от редакции

Редакция издательства «ДМК-Пресс» выражает огромную благодарность Центру развития ИТ-образования при Московском физико-техническом институте и лично его руководителю Алексею Малееву и Mail.Ru Group за помощь в подготовке выпуска этой книги и, конечно, за отличную подготовку наших чемпионов по программированию.

Отзыв Нияза Нигматуллина, двукратного чемпиона мира ACM ICPC 2012 и 2013 годов

Эта книга помогает познакомиться с олимпиадным программированием. Она рассчитана больше на начинающих либо не очень опытных в этом деле читателей, чем на продвинутых. Здесь подробно описано, как проходят олимпиады, что требуется, в чем их цель. Рассказано, как нужно к ним готовиться, какие качества в себе вырабатывать и какие есть способы тренироваться. Подробно разобраны базовые темы, трюки и алгоритмы. Средние и сложные методы преподносятся без четких доказательств. Присутствует много полезных олимпиадных «трюков», которые редко встречаются в научной литературе. Большая часть популярных алгоритмов и методов, используемых в решении задач на олимпиадах, упомянута в этой книге.

К каждой теме автор приложил код на языке C++, что позволяет лучше понять описанное и увидеть способы реализации. Есть отдельная глава, в которой описываются те конструкции и библиотеки C++, которые часто используются на олимпиадах, и только те, которые необходимы: описываются они без подробностей и большой теории, а только на том уровне, чтобы читатель смог ими воспользоваться. Если читатель хочет разобраться в них подробнее, то следует почитать дополнительную информацию по языку из специальных источников. Кроме того, описанные инструменты языка C++ сравниваются на протяжении всей книги на эффективность и удобство использования на олимпиадах.

В книге раскрыт уникальный опыт участников и тренеров олимпиадного программирования.

По правилам этих соревнований в финалах нельзя участвовать более двух раз. За более чем сорокалетнюю историю этих чемпионатов всего шесть человек стали двукратными чемпионами мира, все из Санкт-Петербурга. Четверо – из Университета ИТМО: Геннадий Короткевич, Нияз Нигматуллин, Евгений Капун и Михаил Кевер, и двое – из СПбГУ: Николай Дуров и Андрей Лопатин.

Предисловие ко второму изданию

Во второе издание книги добавлено несколько новых разделов, в которых рассматриваются темы повышенного уровня: вычисление преобразования Фурье, нахождение потоков минимальной стоимости в графах и использование конечных автоматов в задачах о строках.

Я благодарен Олли Матилайнену, который прочитал большую часть нового материала и высказал много полезных замечаний и предложений.

Антти Лааксонен

Хельсинки, Финляндия
Февраль 2020

Предисловие к первому изданию

Эта книга задумана как содержательное введение в современное олимпиадное программирование. Предполагается, что читатель уже знаком с основами программирования, но опыт проектирования алгоритмов или участия в соревнованиях по программированию не обязателен. Поскольку в книге рассматривается широкий круг тем разной степени трудности, она будет полезна как начинающей, так и более искушенной аудитории.

Соревнования по программированию имеют довольно долгую историю. *Международные студенческие олимпиады по программированию* (ICPC) для студентов университетов проводились уже в 1970-х годах, а первая *Международная олимпиада по информатике* для учащихся старших классов состоялась в 1989 году. Ныне оба мероприятия стали регулярными и собирают множество участников со всего мира.

В наши дни олимпиадное программирование популярно как никогда. Важную роль в его распространении сыграл интернет. Существует активное сетевое сообщество увлеченных этим движением программистов, каждую неделю проводятся различные соревнования. Одновременно растет и трудность заданий. Технические приемы, которыми еще несколько лет назад владели лишь лучшие из лучших, стали стандартными инструментами, известными многим.

Своими корнями олимпиадное программирование уходит в научное исследование алгоритмов. Но если ученый приводит доказательство работоспособности своего алгоритма, то олимпиадный программист *реализует* алгоритм и подает его на вход системы оценивания результатов. Эта система прогоняет алгоритм через различные тесты, и если все они проходят, то решение принимается. Это важный элемент олимпиадного программирования, который позволяет *автоматически* получить убедительные аргументы в пользу правильности алгоритма. Вообще, олимпиадное программирование стало отличным способом изучения алгоритмов, т. к. от участника требуется спроектировать алгоритм, который действительно работает, а не ограничиваться наброском идей, может, правильных, а может, и нет.

У олимпиадного программирования есть еще одно достоинство – конкурсные задачи заставляют думать. В формулировках задач не бывает никакого жульничества, в отличие от многих курсов по алгоритмам, где вам предлагают решить красивую задачу, но только последнее предложение звучит, к примеру, так: «*Подсказка*: для решения задачи модифицируйте алгоритм Дейкстры». Ну, а дальше думать особенно нечего, поскольку по-

ход к решению уже известен. В олимпиадном программировании так не бывает. У вас имеется полный комплект инструментов, а уж какими из них воспользоваться, решайте сами.

Решение олимпиадных задач развивает навыки программирования и отладки. Обычно решение засчитывается, только если пройдены все тесты, поэтому программист, стремящийся к успеху, должен реализовывать алгоритм без ошибок. Это умение высоко ценится в программной инженерии, так что неудивительно, что ИТ-компании проявляют интерес к имеющим опыт олимпиадного программирования.

Чтобы стать хорошим олимпиадным программистом, нужно много времени, зато на этом пути можно и многому научиться. Можете быть уверены, что, потратив время на чтение этой книги, решение задач и участие в различных соревнованиях, вы будете лучше понимать, как устроены алгоритмы.

Буду рад вашим отзывам. Вы можете писать мне по адресу ahslaaks@cs.helsinki.fi.

Я благодарен многим людям, приславшим отзывы на черновые редакции этой книги. Они очень помогли сделать книгу лучше. Отдельное спасибо Микко Эрвасти (Mikko Ervasti), Яанне Юннила (Janne Junnila), Яанне Коккала (Janne Kokkala), Туукка Корхонену (Tuukka Korhonen), Патрику Остегярду (Patric Östergård) и Роопе Сальми (Roope Salmi), написавшим подробные рецензии на рукопись. Я также признателен Саймону Рису (Simon Rees) и Уэйну Уилеру (Wayne Wheeler) из издательства Springer за плодотворное сотрудничество в ходе подготовки книги к печати.

Антти Лааксонен

Хельсинки, Финляндия
Октябрь, 2017

Глава 1

Введение

В этой главе рассказывается, что такое олимпиадное программирование, представлен план книги и обсуждаются дополнительные образовательные ресурсы.

В разделе 1.1 описаны элементы олимпиадного программирования, перечислены некоторые популярные соревнования по программированию и даны рекомендации о том, как готовиться к соревнованиям.

В разделе 1.2 обсуждаются цели этой книги и рассматриваемые в ней темы, а также кратко излагается содержание каждой главы.

В разделе 1.3 мы познакомимся со сборником задач CSES. Решение задач параллельно чтению этой книги – отличный способ научиться олимпиадному программированию.

В разделе 1.4 обсуждаются другие книги по олимпиадному программированию и проектированию алгоритмов.

1.1. Что такое олимпиадное программирование?

Олимпиадное программирование состоит из двух частей – проектирования алгоритмов и реализации алгоритмов.

Проектирование алгоритмов. По сути своей, олимпиадное программирование – это придумывание эффективных алгоритмов решения корректно поставленных вычислительных задач. Для проектирования алгоритмов необходимы навыки в решении задач и знание математики. Зачастую решение появляется в результате сочетания хорошо известных методов и новых идей.

Важную роль в олимпиадном программировании играет математика. На самом деле четких границ между проектированием алгоритмов и математикой не существует. Эта книга не требует глубокой математической подготовки. В приложении, которое можно использовать как справочник, описаны некоторые встречающиеся в книге математические понятия и методы, например множества, математическая логика и функции.

Реализация алгоритмов. В олимпиадном программировании решение задачи оценивается путем проверки реализованного алгоритма на ряде тестов. Поэтому придумать алгоритм недостаточно, его еще нуж-

но корректно реализовать, для чего требуется умение программировать. Олимпиадное программирование сильно отличается от традиционной программной инженерии: программы короткие (несколько сотен строк – уже редкость), писать их нужно быстро, а сопровождение после соревнования не требуется.

В настоящее время на соревнованиях по программированию популярнее всего языки C++, Python и Java. Например, среди 3000 лучших участников Google Code Jam 2017 79 % писали на C++, 16 % – на Python и 8 % – на Java. Многие считают C++ самым подходящим выбором для олимпиадного программиста. К его достоинствам можно отнести очень высокую эффективность и тот факт, что в стандартной библиотеке много разнообразных структур данных и алгоритмов.

Все примеры в книге написаны на C++, в них часто используются структуры данных и алгоритмы из стандартной библиотеки. Код отвечает стандарту C++11, который разрешен в большинстве современных соревнований. Если вы еще не знаете C++, самое время начать его изучение.

1.1.1. *Соревнования по программированию*

Международная олимпиада по информатике (**IOI**) – ежегодное соревнование для старшеклассников. От каждой страны допускается команда из четырех человек. Обычно набирается около 300 участников из 80 стран.

IOI проводится в течение двух дней. В каждый день участникам предлагается решить три трудные задачи, на решение отводится пять часов. Задачи разбиты на подзадачи, за каждую из которых начисляются баллы. Хотя участники являются членами команды, соревнуются они самостоятельно.

Участники IOI отбираются на национальных олимпиадах. IOI предшествует множество региональных соревнований, например: Балтийская олимпиада по информатике (BOI), Центрально-Европейская олимпиада по информатике (CEOI) и Азиатско-Тихоокеанская олимпиада по информатике (APIO).

ICPC (Международная студенческая олимпиада по программированию) проводится ежегодно для студентов университетов. В каждой команде три участника; в отличие от IOI, студенты работают вместе, и каждой команде выделяется только один компьютер.

ICPC включает несколько этапов, лучшие команды приглашаются на финальный этап мирового первенства. Хотя в соревновании принимают участие тысячи студентов, количество участников финала ограничено[1], поэтому даже сам выход в финал считается большим достижением.

На соревновании ICPC у команды есть пять часов на решение примерно десяти алгоритмических задач. Решение засчитывается, только если прог-

[1] Точное количество участников финала от года к году меняется. В 2017 году их было 133.

рамма прошла все тесты и показала свою эффективность. В ходе соревнования участники могут видеть результаты соперников, но в начале пятого часа табло замораживается, и результаты последних прогонов не видны.

Онлайновые соревнования. Существует также много онлайновых соревнований, куда допускаются все желающие. В настоящий момент наиболее активен сайт Codeforces, который организует конкурсы почти каждую неделю. Отметим также сайты AtCoder, CodeChef, CS Academy, HackerRank и Topcoder.

Некоторые компании организуют онлайновые соревнования с очными финалами, например: Facebook Hacker Cup, Google Code Jam и Yandex.Algorithm. Разумеется, компании используют такие соревнования для подбора кадров: достойно выступить в соревновании – хороший способ доказать свои таланты в программировании.

1.1.2. *Рекомендации желающим поучаствовать*

Чтобы научиться олимпиадному программированию, нужно напряженно работать. Но способов приобрести практический опыт много, одни лучше, другие хуже.

Решая задачи, нужно иметь в виду, что не так важно *количество* решенных задач, как их *качество*. Возникает соблазн выбирать красивые и легкие задачи, пропуская те, что кажутся трудными и требующими кропотливого труда. Но чтобы отточить свои навыки, нужно отдавать предпочтение именно задачам второго типа.

Важно и то, что большинство олимпиадных задач решается с помощью простого и короткого алгоритма, самое трудное – придумать этот алгоритм. Смысл олимпиадного программирования – не в заучивании сложных и малопонятных алгоритмов, а в том, чтобы научиться решать трудные задачи, обходясь простыми средствами.

Наконец, есть люди, презирающие реализацию алгоритмов, им доставляет удовольствие проектировать алгоритмы, а реализовывать их скучно. Однако умение быстро и правильно реализовать алгоритм – важное преимущество, и этот навык поддается тренировке. Будет очень плохо, если вы потратите большую часть отведенного времени на написание кода и поиск ошибок, а не на обдумывание того, как решить задачу.

1.2. Об этой книге

Программа IOI [17] определяет, какие темы могут предлагаться на Международных олимпиадах по информатике. Именно эта программа стала отправной точкой при отборе материала. Но обсуждаются также более сложные темы, которые исключены из IOI (по состоянию на 2017 год), но могут встречаться в других соревнованиях. К ним относятся, например, максимальные потоки, игра ним и суффиксные массивы.

Многие темы олимпиадного программирования обсуждаются в стандартных учебниках по алгоритмам, но есть и отличия. Например, во многих книгах реализация алгоритмов сортировки и основных структур данных рассматривается с нуля, но такие знания на олимпиаде несущественны, потому что можно пользоваться средствами из стандартной библиотеки. С другой стороны, некоторые темы хорошо известны в олимпиадном сообществе, но редко встречаются в учебниках. Примером может служить дерево отрезков – структура данных, которая используется для решения многих задач без привлечения более хитроумных алгоритмов.

Одна из целей этой книги – *документировать* приемы олимпиадного программирования, которые обычно обсуждаются только на форумах и в блогах. Там, где возможно, даются ссылки на научную литературу по таким методам. Но сделать это можно не всегда, потому что многие методы ныне стали частью *фольклора*, и никто не знает имени первооткрывателя.

Книга организована следующим образом.

- В главе 2 рассматриваются средства языка программирования C++, а затем обсуждаются рекурсивные алгоритмы и поразрядные операции.
- Глава 3 посвящена эффективности: как создавать алгоритмы, способные быстро обрабатывать большие наборы данных.
- В главе 4 обсуждаются алгоритмы сортировки и двоичного поиска с упором на их применение в проектировании алгоритмов.
- В главе 5 дается обзор избранных структур данных в стандартной библиотеке C++: векторов, множеств и отображений.
- Глава 6 представляет собой введение в один из методов проектирования алгоритмов – динамическое программирование. Здесь же приводятся примеры задач, которые можно решить этим методом.
- В главе 7 рассматриваются элементарные алгоритмы на графах, в т. ч. поиск кратчайших путей и минимальные остовные деревья.
- В главе 8 речь пойдет о более сложных вопросах проектирования алгоритмов, в частности об алгоритмах с параллельным просмотром разрядов и амортизационном анализе.
- Тема главы 9 – эффективная обработка запросов по диапазонам массива, таких как вычисление сумм элементов и нахождение минимальных элементов.
- В главе 10 представлены специальные алгоритмы для деревьев, в т. ч. методы обработки запросов к дереву.
- В главе 11 обсуждаются разделы математики, часто встречающиеся в олимпиадном программировании.

- В главе 12 описываются дополнительные алгоритмы на графах, например поиск компонент сильной связности и вычисление максимального потока.
- Глава 13 посвящена геометрическим алгоритмам, в ней описаны методы, позволяющие удобно решать геометрические задачи.
- В главе 14 рассматриваются методы работы со строками, в т. ч. хеширование, Z-алгоритм и суффиксные массивы.
- В главе 15 обсуждаются избранные дополнительные темы, например алгоритмы, основанные на идее квадратного корня, и оптимизация динамического программирования.

1.3. Сборник задач CSES

В *сборнике задач CSES* представлены задачи для практики в олимпиадном программировании. Расположены они в порядке возрастания трудности, а в этой книге обсуждаются все методы, необходимые для их решения. Сборник задач доступен по адресу:

https://cses.fi/problemset/.

Посмотрим, как решить первую задачу из него. Она называется «Странный алгоритм» и формулируется следующим образом.

Рассмотрим алгоритм, принимающий на входе целое положительное число n. Если n четно, то алгоритм делит его на два, а если нечетно, то умножает на три и прибавляет 1. Например, для $n = 3$ получается следующая последовательность:

$$3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

Ваша задача заключается в том, чтобы промоделировать выполнение этого алгоритма для любого заданного n.

Вход

Единственная строка, содержащая целое число n.

Выход

Печатает строку, содержащую все значения, вычисляемые алгоритмом.

Ограничения

- $1 \leqslant n \leqslant 10^6$

Пример

Вход:

3

Выход:

3 10 5 16 8 4 2 1

Эта проблема имеет отношение к знаменитой *гипотезе Коллатца*, которая утверждает, что описанный выше алгоритм завершается для любого *n*. До сих пор она остается недоказанной. Но в этой задаче мы знаем, что начальное значение *n* не превышает миллиона, что существенно упрощает проблему.

Это простая задача моделирования, не требующая глубоких размышлений. Вот как ее можно было бы решить на C++:

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    while (true) {
        cout << n << " ";
        if (n == 1) break;
        if (n%2 == 0) n /= 2;
        else n = n*3+1;
    }
    cout << "\n";
}
```

Сначала программа читает входное число *n*, затем моделирует алгоритм и печатает значение *n* после каждого шага. Легко проверить, что этот код правильно обрабатывает случай *n* = 3, приведенный в формулировке задачи.

Теперь время *отправить* этот код в CSES. Для каждого теста CSES сообщает, пройден он или нет, и показывает вход, ожидаемый и реальный выход.

По результатам тестирования CSES формирует следующий отчет:

test	verdict	time (s)
#1	ACCEPTED	0.06 / 1.00
#2	ACCEPTED	0.06 / 1.00
#3	ACCEPTED	0.07 / 1.00
#4	ACCEPTED	0.06 / 1.00
#5	ACCEPTED	0.06 / 1.00
#6	TIME LIMIT EXCEEDED	_ / 1.00
#7	TIME LIMIT EXCEEDED	_ / 1.00
#8	WRONG ANSWER	0.07 / 1.00
#9	TIME LIMIT EXCEEDED	_ / 1.00
#10	ACCEPTED	0.06 / 1.00

Это означает, что некоторые тесты наш код прошел (ACCEPTED), на некоторых оказался слишком медленным (TIME LIMIT EXCEEDED), а в одном случае дал неверный результат (WRONG ANSWER). Вот уж действительно неожиданно!

Первым не прошел тест для n = 138 367. Локально прогнав программу в этом случае, мы убедимся, что она действительно работает долго. На самом деле она вообще не завершается.

Причина ошибки в том, что в процессе моделирования n может оказаться слишком большим, в том числе больше максимального значения, представимого типом `int`. Чтобы решить проблему, достаточно заменить тип `int` на `long long`. Тогда получится то, что нужно:

test	verdict	time (s)
#1	ACCEPTED	0.05 / 1.00
#2	ACCEPTED	0.06 / 1.00
#3	ACCEPTED	0.07 / 1.00
#4	ACCEPTED	0.06 / 1.00
#5	ACCEPTED	0.06 / 1.00
#6	ACCEPTED	0.05 / 1.00
#7	ACCEPTED	0.06 / 1.00
#8	ACCEPTED	0.05 / 1.00
#9	ACCEPTED	0.07 / 1.00
#10	ACCEPTED	0.06 / 1.00

Как видно из этого примера, даже в очень простые алгоритмы могут вкрасться тонкие ошибки. Олимпиадное программирование учит, как писать алгоритмы, которые действительно работают.

1.4. Другие ресурсы

Помимо этой, уже есть и другие книги по олимпиадному программированию. Первой была книга Skiena, Revilla «Programming Challenges» [32], вышедшая в 2003 году. Позже вышла книга Halim, Halim «Competitive Programming 3» [16]. Обе ориентированы на читателя, не имеющего опыта олимпиадного программирования.

Книга «Looking for a Challenge?» [8] – сборник трудных задач, предлагавшихся на польских олимпиадах, – рассчитана на более подготовленного читателя. Самое интересное в ней – подробный анализ решений.

Разумеется, для подготовки к олимпиадам подойдут и общие книги по алгоритмам. Самая всеобъемлющая из них – книга Cormen, Leiserson,

Rivest, Stein «Introduction to Algorithms»[2] [7], которую также называют просто *CLRS*. Это прекрасный источник для тех, кто хочет узнать обо всех деталях алгоритма и познакомиться со строгим доказательством.

В книге Kleinberg, Tardos «Algorithm Design» [22] основное внимание уделяется технике проектирования алгоритмов и во всех деталях обсуждаются такие темы, как метод «разделяй и властвуй», жадные алгоритмы, динамическое программирование и вычисление максимального потока. Книга Skiena «The Algorithm Design Manual» [31] – практическое руководство, содержащее обширный каталог вычислительных задач и способов их решения.

[2] *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн*. Алгоритмы. Построение и анализ. М.: Вильямс, 2016.

Глава 2

Техника программирования

В этой главе представлены некоторые средства языка C++, полезные в олимпиадном программировании, а также приведены примеры использования рекурсии и поразрядных операций.

В разделе 2.1 рассматриваются избранные вопросы C++, включая ввод и вывод, работу с числами и способы сокращения кода.

Раздел 2.2 посвящен рекурсивным алгоритмам. Сначала мы рассмотрим элегантный способ порождения всех подмножеств и перестановок множества с применением рекурсии. А затем с помощью перебора с возвратом подсчитаем, сколькими способами можно расположить n ферзей на шахматной доске $n \times n$, так чтобы они не били друг друга.

В разделе 2.3 обсуждаются основы поразрядных операций и их применение для представления подмножеств множества.

2.1. Языковые средства

Типичная олимпиадная программа на C++ устроена по такому образцу:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // здесь находится решение
}
```

Строка `#include` в начале программы – специфика компилятора `g++`, она служит для включения всей стандартной библиотеки. Таким образом, не нужно по отдельности включать такие библиотеки, как `iostream`, `vector` и `algorithm`; все они автоматически становятся доступны.

В строке `using` объявляется, что все классы и функции из стандартной библиотеки можно использовать без указания пространства имен. Не будь `using`, нужно было бы писать, к примеру, `std::cout`, а так достаточно просто `cout`.

Для компиляции программы используется команда вида:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

Она порождает двоичный файл `test` из исходного файла `test.cpp`. Флаги означают, что компилятор должен следовать стандарту C++11 (`-std=c++11`), оптимизировать код (`-O2`) и выводить предупреждения о возможных проблемах (`-Wall`).

2.1.1. *Ввод и вывод*

В большинстве олимпиадных задач для ввода и вывода используются стандартные потоки. В C++ стандартный поток ввода называется `cin`, а стандартный поток вывода – `cout`. Можно также использовать C-функции, например `scanf` и `printf`.

Входными данными для программы обычно являются числа и строки, разделенные пробелами и знаками новой строки. Из потока `cin` они читаются следующим образом:

```
int a, b;
string x;
cin >> a >> b >> x;
```

Такой код работает в предположении, что элементы потока разделены хотя бы одним пробелом или знаком новой строки. Например, приведенный выше код может прочитать как входные данные:

```
123 456 monkey
```

так и входные данные:

```
123 456
monkey
```

Поток `cout` используется для вывода:

```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```

Ввод и вывод часто оказываются узкими местами программы. Чтобы повысить эффективность ввода-вывода, можно поместить в начало программы такие строки:

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Отметим, что знак новой строки `"\n"` работает быстрее, чем `endl`, потому что `endl` всегда сопровождается сбросом буфера.

C-функции `scanf` и `printf` – альтернатива стандартным потокам C++. Обычно они работают немного быстрее, но и использовать их сложнее. Вот как можно прочитать из стандартного ввода два целых числа:

```
int a, b;
scanf("%d %d", &a, &b);
```

А вот как их можно напечатать:

```
int a = 123, b = 456;
printf("%d %d\n", a, b);
```

Иногда программа должна прочитать целую входную строку, быть может, содержащую пробелы. Это можно сделать с помощью функции `getline`:

```
string s;
getline(cin, s);
```

Если объем данных заранее неизвестен, то полезен такой цикл:

```
while (cin >> x) {
    // код
}
```

Этот цикл читает из стандартного ввода элементы один за другим, пока входные данные не закончатся.

В некоторых олимпиадных системах для ввода и вывода используются файлы. В таком случае есть простое решение: писать код так, будто работаешь со стандартными потоками, но в начало программы добавить такие строки:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

После этого программа будет читать данные из файла «input.txt», а записывать в файл «output.txt».

2.1.2. *Работа с числами*

Целые числа. Из целых типов в олимпиадном программировании чаще всего используется `int` – 32-разрядный тип[1], принимающий значения из диапазона $-2^{31}...2^{31}-1$ (приблизительно $-2 \cdot 10^9...2 \cdot 10^9$). Если типа `int` недостаточно, то можно взять 64-разрядный тип `long long`. Диапазон его значений $-2^{63}...2^{63}-1$ (приблизительно $-9 \cdot 10^{18}...9 \cdot 10^{18}$).

Ниже определена переменная типа `long long`:

[1] На самом деле стандарт C++ не определяет точные размеры числовых типов, а границы диапазонов зависят от платформы и компилятора. В этом разделе указаны размеры, с которыми вы, вероятнее всего, встретитесь в современных системах.

```
long long x = 123456789123456789LL;
```

Суффикс LL означает, что число имеет тип long long.

Типичная ошибка при использовании типа long long возникает, когда где-то в программе встречается еще и тип int. Например, в следующем коде есть тонкая ошибка:

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Хотя переменная b имеет тип long long, оба сомножителя в выражении a*a имеют тип int, поэтому тип результата тоже int. Из-за этого значение b оказывается неверным. Проблему можно решить, изменив тип a на long long или изменив само выражение на (long long)a*a.

Обычно олимпиадные задачи формулируются так, что типа long long достаточно. Но все же полезно знать, что компилятор g++ поддерживает также 128-разрядный тип __int128_t с диапазоном значений $-2^{127}...2^{127}-1$ (приблизительно $-10^{38}...10^{38}$). Однако этот тип доступен не во всех олимпиадных системах.

Арифметика по модулю. Иногда ответом является очень большое число, но достаточно вывести результат «по модулю m», т. е. остаток от деления на m (например, «7 по модулю 10^9»). Идея в том, что даже когда истинный ответ очень велик, типов int и long long все равно достаточно.

Остаток x от деления на m обозначается $x \bmod m$. Например, $17 \bmod 5 = 2$, поскольку $17 = 3 \cdot 5 + 2$. Важные свойства остатков выражаются следующими формулами:

$$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m;$$
$$(a - b) \bmod m = (a \bmod m - b \bmod m) \bmod m;$$
$$(a \cdot b) \bmod m = (a \bmod m \cdot b \bmod m) \bmod m.$$

Это значит, что можно брать остаток после каждой операции, поэтому числа никогда не станут слишком большими.

Например, следующий код вычисляет $n!$ (n факториал) по модулю m:

```
long long x = 1;
for (int i = 1; i <= n; i++) {
    x = (x*i)%m;
}
cout << x << "\n";
```

Обычно мы хотим, чтобы остаток находился в диапазоне $0...m-1$. Но в C++ и в других языках остаток от деления отрицательного числа равен

нулю или отрицателен. Чтобы не возникали отрицательные остатки, можно поступить следующим образом: сначала вычислить остаток, а если он отрицателен, прибавить m:

```
x = x%m;
if (x < 0) x += m;
```

Но это стоит делать, только если в программе встречается операция вычитания, так что остаток может стать отрицательным.

Числа с плавающей точкой. В большинстве олимпиадных задач целых чисел достаточно, но иногда возникает потребность в числах с плавающей точкой. В C++ наиболее полезен 64-разрядный тип `double`, а в компиляторе `g++` имеется расширение – 80-разрядный тип `long double`. Чаще всего типа `double` достаточно, но `long double` дает более высокую точность.

Требуемая точность ответа обычно указывается в формулировке задачи. Проще всего для вывода ответа использовать функцию `printf` и указать количество десятичных цифр в форматной строке. Например, следующий код печатает значение x с 9 цифрами после запятой:

```
printf("%.9f\n", x);
```

С использованием чисел с плавающей точкой связана одна сложность: некоторые числа невозможно точно представить в таком формате, поэтому неизбежны ошибки округления. Например, в следующем коде получается значение x, немного меньшее 1, тогда как правильное значение равно в точности 1.

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

Числа с плавающей точкой рискованно сравнивать с помощью оператора ==, потому что иногда равные значения оказываются различны из-за ошибок округления. Более правильно считать, что два числа равны, если разность между ними меньше ε, где ε мало. Например, в следующем коде $\varepsilon = 10^{-9}$:

```
if (abs(a-b) < 1e-9) {
    // a и b равны
}
```

Отметим, что хотя числа с плавающей точкой, вообще говоря, не точны, не слишком большие целые числа представляются точно. Так, тип `double` позволяет точно представить все целые числа, по абсолютной величине не большие 2^{53}.

2.1.3. *Сокращение кода*

Имена типов. Ключевое слово typedef позволяет сопоставить типу данных короткое имя. Например, имя long long слишком длинное, поэтому можно определить для него короткий псевдоним ll:

```
typedef long long ll;
```

После этого код

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

можно немного сократить:

```
ll a = 123456789;
ll b = 987654321;
cout << a*b << "\n";
```

Ключевое слово typedef применимо и к более сложным типам. Например, ниже мы сопоставляем вектору целых чисел имя vi, а паре двух целых чисел – тип pi.

```
typedef vector<int> vi;
typedef pair<int,int> pi;
```

Макросы. Еще один способ сократить код – макросы. Макрос говорит, что определенные строки кода следует подменить до компиляции. В C++ макросы определяются с помощью ключевого слова #define.

Например, мы можем определить следующие макросы:

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

После чего код

```
v.push_back(make_pair(y1,x1));
v.push_back(make_pair(y2,x2));
int d = v[i].first+v[i].second;
```

можно сократить до:

```
v.PB(MP(y1,x1));
v.PB(MP(y2,x2));
int d = v[i].F+v[i].S;
```

У макроса могут быть параметры, что позволяет сокращать циклы и другие структуры. Например, можно определить такой макрос:

```
#define REP(i,a,b) for (int i = a; i <= b; i++)
```

После этого код

```
for (int i = 1; i <= n; i++) {
    search(i);
}
```

можно сократить следующим образом:

```
REP(i,1,n) {
    search(i);
}
```

2.2. Рекурсивные алгоритмы

Благодаря *рекурсии* часто удается элегантно реализовать алгоритм. В этом разделе мы обсудим рекурсивные алгоритмы, которые систематически перебирают потенциальные решения задачи. Сначала рассмотрим порождение подмножеств и перестановок множества, а затем обсудим более общую технику перебора с возвратом.

2.2.1. *Порождение подмножеств*

В качестве первого применения рекурсии рассмотрим порождение всех подмножеств множества из n элементов. Например, подмножествами {1, 2, 3} являются 0, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3} и {1, 2, 3}. Следующая рекурсивная функция `search` генерирует подмножества. Функция манипулирует вектором

```
vector<int> subset;
```

который содержит элементы подмножества. Чтобы начать поиск, следует вызвать функцию с параметром 1.

```
void search(int k) {
    if (k == n+1) {
        // обработать подмножество
    } else {
        // включить k в подмножество
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
        // не включать k в подмножество
```

```
        search(k+1);
    }
}
```

При вызове с параметром k функция `search` решает, следует ли включать элемент k в множество или нет, но в обоих случаях вызывает себя с параметром $k + 1$. Если оказывается, что $k = n + 1$, то функция понимает, что все элементы обработаны и подмножество сгенерировано.

На рис. 2.1 показано порождение подмножеств при $n = 3$. При каждом вызове функции выбирается либо верхняя ветвь (k включается в подмножество), либо нижняя (k не включается в подмножество).

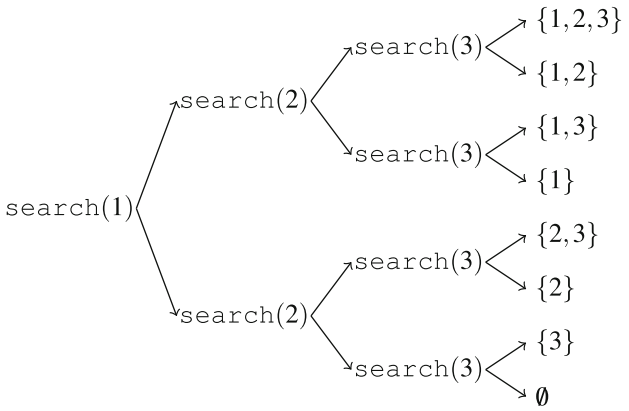


Рис. 2.1. Дерево рекурсии при порождении подмножеств множества {1, 2, 3}

2.2.2. *Порождение перестановок*

Теперь рассмотрим задачу о порождении всех перестановок множества из n элементов. Например, перестановками {1, 2, 3} будут (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) и (3, 2, 1). И снова для решения можно применить рекурсию. Следующая функция манипулирует вектором

```
vector<int> permutation;
```

который содержит перестановку, и массивом

```
bool chosen[n+1];
```

который для каждого элемента показывает, включен он уже в перестановку или нет. Поиск начинается, когда эта функция вызывается без параметров.

```
void search() {
    if (permutation.size() == n) {
        // обработать перестановку
```

```
    } else {
        for (int i = 1; i <= n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}
```

При каждом вызове функция добавляет новый элемент в вектор `permutation` и запоминает в массиве `chosen`, что он был добавлен. Если размер `permutation` оказывается равен размеру множества, то генерируется перестановка.

Отметим, что в стандартной библиотеке C++ имеется функция `next_permutation`, которую также можно использовать для порождения перестановок. Этой функции передается перестановка, а она порождает следующую за ней в лексикографическом порядке. Следующая программа перебирает все перестановки множества {1, 2, ..., *n*}:

```
for (int i = 1; i <= n; i++) {
    permutation.push_back(i);
}
do {
    // обработать перестановку
} while (next_permutation(permutation.begin(), permutation.end()));
```

2.2.3. *Перебор с возвратом*

Алгоритм *перебора с возвратом* начинает работу с пустого решения и шаг за шагом расширяет его. Рекурсивно перебираются все возможные способы построения решения.

В качестве примера рассмотрим задачу о вычислении количества способов расставить *n* ферзей на доске *n*×*n*, так чтобы никакие два не били друг друга. Так, на рис. 2.2 показано два возможных решения для *n* = 4.

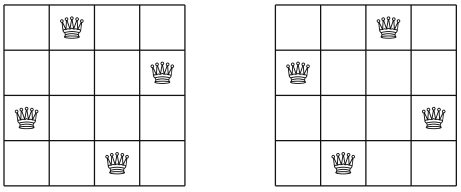


Рис. 2.2. Возможные способы расстановки 4 ферзей на доске 4×4

Задачу можно решить методом перебора с возвратом, рассматривая горизонтали одну за другой. Точнее, на каждой горизонтали можно разместить ровно одного ферзя, так чтобы он не оказался под боем ранее поставленных ферзей. Очередное решение будет найдено, когда на доску поставлены все n ферзей.

Например, на рис. 2.3 показано несколько частичных решений, сгенерированных алгоритмом перебора с возвратом при $n = 4$. Первые три расстановки на нижнем уровне недопустимы, поскольку ферзи бьют друг друга. Но четвертая расстановка допустима, и ее можно дополнить до решения, поставив на доску еще двух ферзей. Сделать это можно единственным способом.

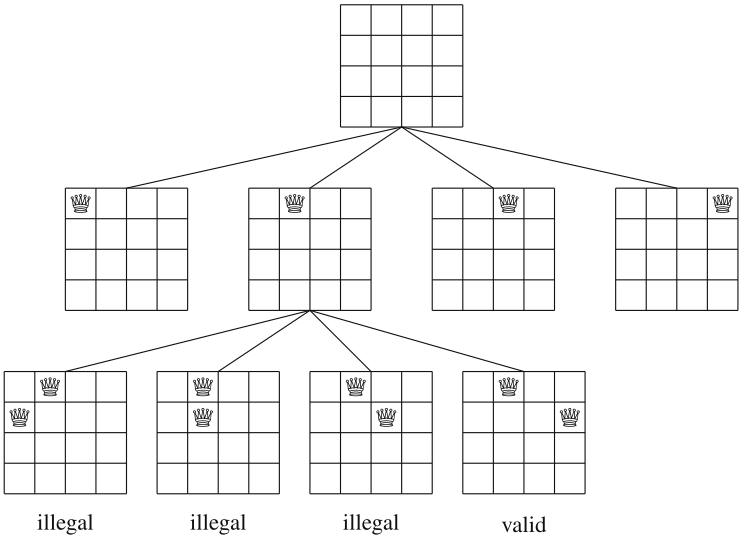


Рис. 2.3. Частичные решения задачи о расстановке ферзей, полученные методом перебора с возвратом

Алгоритм можно реализовать следующим образом:

```
void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (col[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        col[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        col[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```

Поиск начинается вызовом `search(0)`. Размер доски равен n, функция подсчитывает количество решений в переменной `count`. В коде предполагается, что горизонтали и вертикали доски пронумерованы от 0 до $n - 1$. Будучи вызвана с параметром y, функция `search` помещает ферзя на горизонталь y и вызывает себя с параметром $y + 1$. Когда $y = n$, решение найдено, поэтому значение `count` увеличивается на 1.

В массиве `col` запоминаются вертикали, уже занятые ферзями, а в массивах `diag1` и `diag2` запоминаются диагонали. Запрещается ставить ферзя на вертикаль или диагональ, в которой уже находится другой ферзь. На рис. 2.4 показана нумерация вертикалей и диагоналей для доски 4×4.

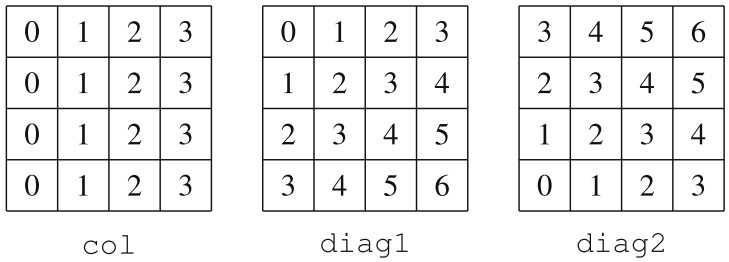


Рис. 2.4. Нумерация массивов при подсчете комбинаций на доске 4×4

Показанный выше алгоритм перебора с возвратом говорит, что существует 92 способа расставить 8 ферзей на доске 8×8. С ростом n поиск быстро замедляется, поскольку число решений возрастает экспоненциально. Так, на современном компьютере требуется около минуты, чтобы вычислить количество расстановок 16 ферзей на доске 16×16 – 14 772 512.

На самом деле до сих пор неизвестен эффективный способ подсчета числа расстановок ферзей для больших n. В настоящее время наибольшее n, для которого результат известен, равно 27: в этом случае существует 234 907 967 154 122 528 расстановок. Это было установлено в 2016 году группой исследователей, использовавших для решения кластер компьютеров [29].

2.3. Побитовые операции

В программировании n-разрядное целое число хранится в виде двоичного числа, содержащего n бит. Например, тип `int` в C++ 32-разрядный, т. е. любое число типа `int` содержит 32 бита. Так, двоичное представление числа 43 типа `int` имеет вид

$$00000000000000000000000000101011.$$

Биты в этом представлении нумеруются справа налево. Преобразование двоичного представления $b_k \ldots b_2 b_1 b_0$ в десятичное число производится по формуле

$$b_k 2^k + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

Например:

$$1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43.$$

Двоичное представление числа может быть *со знаком* и *без знака*. Обычно используется представление со знаком, позволяющее представить положительные и отрицательные числа. n-разрядная переменная со знаком может содержать любое целое число в диапазоне от -2^{n-1} до $2^{n-1} - 1$. Например, тип `int` в C++ знаковый, поэтому переменная типа `int` может содержать любое целое число от -2^{31} до $2^{31} - 1$.

Первый разряд в представлении со знаком содержит знак числа (0 для неотрицательных чисел, 1 – для отрицательных), а остальные $n - 1$ разрядов – абсолютную величину числа. Используется *дополнительный код*, т. е. для получения противоположного числа нужно сначала инвертировать все его биты, а затем прибавить к результату единицу. Например, двоичное представление числа –43 типа `int` имеет вид

11111111111111111111111111010101.

Представление без знака позволяет представить только неотрицательные числа, но верхняя граница диапазона больше. n-разрядная переменная без знака может содержать любое целое число от 0 до $2^n - 1$. Например, в C++ переменная типа `unsigned int` может содержать любое целое число от 0 до $2^{32} - 1$.

Между обоими представлениям существует связь: число со знаком $-x$ равно числу без знака $2^n - x$. К примеру, следующая программа показывает, что число со знаком $x = -43$ равно числу без знака $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Если число больше верхней границы допустимого диапазона, то возникает переполнение. В представлении со знаком число, следующее за $2^{n-1} - 1$, равно -2^{n-1}, а в представлении без знака за $2^n - 1$ следует 0. Рассмотрим следующий код:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Первоначально x принимает значение $2^{31} - 1$. Это наибольшее значение, которое можно сохранить в переменной типа `int`, поэтому следующее за $2^{31} - 1$ значение равно -2^{31}.

2.3.1. *Поразрядные операции*

Операция И. Результатом операции *И* x & y является число, двоичное представление которого содержит единицы в тех позициях, на которых в представлениях x и y находятся единицы. Например, 22 & 26 = 18, поскольку

```
   10110 (22)
&  11010 (26)
=  10010 (18)
```

С помощью операции И можно проверить, является ли число x четным, т. к. x & 1 = 0, если x четно, и x & 1 = 1, если x нечетно. Вообще, x нацело делится на 2^k, если x & $(2^k - 1) = 0$.

Операция ИЛИ. Результатом операции *ИЛИ* x | y является число, двоичное представление которого содержит единицы в тех позициях, на которых хотя бы в одном из представлений x и y находятся единицы. Например, 22 | 26 = 30, поскольку

```
   10110 (22)
|  11010 (26)
=  11110 (30)
```

Операция ИСКЛЮЧАЮЩЕЕ ИЛИ. Результатом операции *ИСКЛЮЧАЮЩЕЕ ИЛИ* x ^ y является число, двоичное представление которого содержит единицы в тех позициях, на которых ровно в одном из представлений x и y находятся единицы. Например, 22 ^ 26 = 12, поскольку

```
   10110 (22)
^  11010 (26)
=  01100 (12)
```

Операция НЕ. Результатом операции *НЕ* ~x является число, в двоичном представлении которого все биты x инвертированы. Справедлива формула ~x = $-x$ – 1, например ~29 = −30. Результат операции *НЕ* на битовом уровне зависит от длины двоичного представления, поскольку инвертируются все биты. Например, в случае 32-разрядных чисел типа `int` имеем:

```
 x =   29  00000000000000000000000000011101
~x = −30  11111111111111111111111111100010
```

Поразрядный сдвиг. Операция поразрядного сдвига влево x << k дописывает в конец числа k нулей, а операция поразрядного сдвига вправо

x >> k удаляет k последних бит. Например, 14 << 2 = 56, поскольку двоичные представления 14 и 56 равны соответственно 1110 и 111000. Аналогично 49 >> 3 = 6, потому что 49 и 6 в двоичном виде равны соответственно 110001 и 110. Отметим, что операция x << k соответствует умножению x на 2^k, а x >> k – делению x на 2^k с последующим округлением с недостатком до целого.

Битовые маски. *Битовой маской* называется число вида 1 << k, содержащее в позиции k единицу, а во всех остальных позициях – нули. Такую маску можно использовать для выделения одиночных битов. В частности, k-й бит числа равен единице тогда и только тогда, когда x & (1 << k) не равно нулю. В следующем фрагменте печатается двоичное представление числа x типа `int`:

```
for (int k = 31; k >= 0; k--) {
   if (x&(1<<k)) cout << "1";
   else cout << "0";
}
```

Аналогичным образом можно модифицировать отдельные биты числа. Выражение x | (1 << k) устанавливает k-й бит x в единицу, выражение x & ~(1 << k) сбрасывает k-й бит x в нуль, а выражение x ^ (1 << k) инвертирует k-й бит x. Далее выражение x & (x − 1) сбрасывает последний единичный бит x в нуль, а выражение x & −x сбрасывает в нуль все единичные биты, кроме последнего. Выражение x | (x − 1) инвертирует все биты после последнего единичного. Наконец, положительное число x является степенью двойки тогда и только тогда, когда x & (x − 1) = 0.

При работе с битовыми масками нужно помнить, что `1<<k` всегда имеет тип `int`. Самый простой способ создать битовую маску типа `long long` – написать `1LL<<k`.

Дополнительные функции. Компилятор g++ предлагает также следующие функции для подсчета битов:

- `__builtin_clz`(x): количество нулей в начале двоичного представления;
- `__builtin_ctz`(x): количество нулей в конце двоичного представления;
- `__builtin_popcount`(x): количество единиц в двоичном представлении;
- `__builtin_parity`(x): четность количества единиц в двоичном представлении.

Эти функции используются следующим образом:

```
int x = 5328; // 00000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
```

```
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

Отметим, что данные функции поддерживают только числа типа `int`, но есть также их варианты для типа `long long`, их имена заканчиваются суффиксом `ll`.

2.3.2. *Представление множеств*

Любое подмножество множества {0, 1, 2, ..., n − 1} можно представить n-разрядным целым числом, в котором единичные биты соответствуют элементам, принадлежащим подмножеству. Такой способ представления эффективен, поскольку для каждого элемента требуется всего один бит памяти, а теоретико-множественные операции можно реализовать с помощью поразрядных операций.

Например, поскольку тип `int` 32-разрядный, числом типа `int` можно представить любое подмножество множества {0, 1, 2, ..., 31}. Двоичное представление множества {1, 3, 4, 8} имеет вид

00000000000000000000000100011010,

что соответствует числу $2^8 + 2^4 + 2^3 + 2^1 = 282$.

В показанном ниже коде объявлена переменная x типа `int`, которая может содержать подмножество множества {0, 1, 2, ..., 31}. Затем в это подмножество добавляются элементы 1, 3, 4, 8 и печатается его размер.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Далее печатаются все элементы, принадлежащие множеству:

```
for (int i = 0; i < 32; i++) {
   if (x&(1<<i)) cout << i << " ";
}
// выводится: 1 3 4 8
```

Операции над множествами. В табл. 2.1 показано, как реализовать теоретико-множественные операции с помощью побитовых. Так, следующий код сначала конструирует множества x = {1, 3, 4, 8} и y = {3, 6, 8, 9}, а затем множество $z = x \cup y$ = {1, 3, 4, 6, 8, 9}:

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

Таблица 2.1. Реализация теоретико-множественных операций с помощью побитовых

Операция	Теоретико-множественная запись	Побитовая запись
Пересечение	$a \cap b$	a & b
Объединение	$a \cup b$	$a \mid b$
Дополнение	$\bar{a}$	$\sim a$
Разность	$a \setminus b$	a & ($\sim b$)

Следующий код перебирает подмножества множества $\{0, 1, \ldots, n - 1\}$:

```
for (int b = 0; b < (1<<n); b++) {
   // обработать подмножество b
}
```

А этот код перебирает только подмножества, содержащие ровно k элементов:

```
for (int b = 0; b < (1<<n); b++) {
   if (__builtin_popcount(b) == k) {
      // обработать подмножество b
}
}
```

Наконец, следующий код перебирает все подмножества множества x:

```
int b = 0;
do {
   // обработать подмножество b
} while (b=(b-x)&x);
```

Битовые множества в C++. В стандартной библиотеке C++ имеется структура `bitset`, соответствующая массиву, все элементы которого равны 0 или 1. Например, следующий код создает битовое множество из 10 элементов:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
```

```
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Функция count возвращает количество единичных битов в битовом множестве:

```
cout << s.count() << "\n"; // 4
```

К битовым множествам также применимы побитовые операции:

```
bitset<10> a, b;
// ...
bitset<10> c = a&b;
bitset<10> d = a|b;
bitset<10> e
```

Глава 3

Эффективность

Эффективность алгоритмов играет центральную роль в олимпиадном программировании. В этой главе мы будем изучать средства, упрощающие проектирование эффективных алгоритмов.

В разделе 3.1 вводится понятие временной сложности, которое позволяет оценить время работы алгоритма, не реализуя его. Временная сложность описывает, как быстро время работы алгоритма увеличивается с ростом размера входных данных.

В разделе 3.2 приведены две задачи, которые можно решить многими способами. В обоих случаях легко спроектировать медленный алгоритм с полным перебором, но, как выясняется, можно создать гораздо более эффективные алгоритмы.

3.1. Временная сложность

Временная сложность алгоритма – это оценка того, сколько времени будет работать алгоритм при заданных входных данных. Зная временную сложность, мы зачастую можем сказать, достаточно ли алгоритм быстрый для решения задачи, даже не реализуя его.

Для описания временной сложности применяется нотация $O(\ldots)$, где многоточием представлена некоторая функция. Обычно буквой n обозначается размер входных данных. Например, если на вход подается массив чисел, то n – это размер массива, а если строка, то n – длина строки.

3.1.1. *Правила вычисления*

Если код включает только линейную последовательность команд, как, например, показанный ниже, то его временная сложность равна $O(1)$.

```
a++;
b++;
c = a+b;
```

Временная сложность цикла оценивает число выполненных итераций. Например, временная сложность следующего кода равна $O(n)$, поскольку

код внутри цикла выполняется n раз. При этом предполагается, что многоточием «...» обозначен код с временной сложностью $O(1)$.

```
for (int i = 1; i <= n; i++) {
    ...
}
```

Временная сложность следующего кода равна $O(n^2)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
```

Вообще, если имеется k вложенных циклов и в каждом цикле перебирается n значений, то временная сложность равна $O(n^k)$.

Временная сложность не сообщает, сколько точно раз выполняется код внутри цикла, она показывает лишь порядок величины, игнорируя постоянные множители. В примерах ниже код внутри цикла выполняется $3n$, $n + 5$ и $\lceil n/2 \rceil$ раз, но временная сложность в каждом случае равна $O(n)$.

```
for (int i = 1; i <= 3*n; i++) {
    ...
}

for (int i = 1; i <= n+5; i++) {
    ...
}

for (int i = 1; i <= n; i += 2) {
...
}
```

С другой стороны, временная сложность следующего кода равна $O(n^2)$, поскольку код внутри цикла выполняется $1 + 2 + \ldots + n = ½(n^2 + n)$ раз.

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        ...
    }
}
```

Если алгоритм состоит из нескольких последовательных частей, то общая временная сложность равна максимуму из временных сложностей отдельных частей, т. е. самая медленная часть является узким местом. Так,

следующий код состоит из трех частей с временной сложностью $O(n)$, $O(n^2)$ и $O(n)$. Поэтому общая временная сложность равна $O(n^2)$.

```
for (int i = 1; i <= n; i++) {
    ...
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
for (int i = 1; i <= n; i++) {
    ...
}
```

Иногда временная сложность зависит от нескольких факторов, поэтому формула включает несколько переменных. Например, временная сложность следующего кода равна $O(nm)$:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        ...
    }
}
```

Временная сложность рекурсивной функции зависит от того, сколько раз она вызывается, и от временной сложности одного вызова. Общая временная сложность равна произведению того и другого. Рассмотрим, к примеру, следующую функцию:

```
void f(int n) {
    if (n == 1) return;
    f(n-1);
}
```

Вызов $f(n)$ приводит к n вызовам функций, и временная сложность каждого вызова равна $O(1)$, поэтому общая временная сложность равна $O(n)$.

В качестве еще одного примера рассмотрим следующую функцию:

```
void g(int n) {
    if (n == 1) return;
    g(n-1);
    g(n-1);
}
```

Что происходит, когда эта функция вызывается с параметром n? Сначала она будет дважды вызвана с параметром $n - 1$, затем четыре раза с параметром $n - 2$, потом восемь раз с параметром $n - 3$ и т. д. Вообще, будет 2^k вызовов с параметром $n - k$, где $k = 0, 1, \ldots, n - 1$. Таким образом, общая временная сложность равна

$$1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1 = O(2^n).$$

3.1.2. *Часто встречающиеся оценки временной сложности*

Ниже перечислены часто встречающиеся оценки временной сложности алгоритмов.

$O(1)$ Время работы алгоритма *с постоянным временем* не зависит от размера входных данных. Типичным примером может служить явная формула, по которой вычисляется ответ.

$O(\log n)$ В *логарифмическом* алгоритме размер входных данных на каждом шаге обычно уменьшается вдвое. Время работы зависит от размера входных данных логарифмически, поскольку $\log_2 n$ – это сколько раз нужно разделить n на 2, чтобы получить 1. Отметим, что основание логарифма во временной сложности не указывается.

$O(\sqrt{n})$ Алгоритм с временной сложностью $O(\sqrt{n})$ медленнее, чем $O(\log n)$, но быстрее, чем $O(n)$. Специальное свойство квадратного корня заключается в том, что $\sqrt{n} = n/\sqrt{n}$, т. е. n элементов можно разбить на $O(\sqrt{n})$ порций по $O(\sqrt{n})$ элементов.

$O(n)$ *Линейный* алгоритм перебирает входные данные постоянное число раз. Зачастую это наилучшая возможная временная сложность, потому что обычно, чтобы получить ответ, необходимо обратиться к каждому элементу хотя бы один раз.

$O(n \log n)$ Такая временная сложность часто означает, что алгоритм сортирует входные данные, поскольку временная сложность эффективных алгоритмов сортировки равна $O(n \log n)$. Есть и другая возможность – в алгоритме используется структура данных, для которой каждая операция занимает время $O(\log n)$.

$O(n^2)$ *Квадратичный* алгоритм нередко содержит два вложенных цикла. Перебрать все пары входных элементов можно за время $O(n^2)$.

$O(n^3)$ *Кубический* алгоритм часто содержит три вложенных цикла. Все тройки входных элементов можно перебрать за время $O(n^3)$.

$O(2^n)$ Такая временная сложность нередко указывает на то, что алгоритм перебирает все подмножества множества входных данных. Например, подмножествами множества {1, 2, 3} являются ∅, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3} и {1, 2, 3}.

$O(n!)$ Такая временная сложность часто означает, что алгоритм перебирает все перестановки входных элементов. Например, перестановками множества {1, 2, 3} являются (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) и (3, 2, 1).

Алгоритм называется *полиномиальным*, если его временная сложность не выше $O(n^k)$, где k – константа. Все приведенные выше оценки временной сложности, кроме $O(2^n)$ и $O(n!)$, полиномиальные. На практике константа k обычно мала, поэтому полиномиальная временная сложность, грубо говоря, означает, что алгоритм может обрабатывать большие входные данные.

Большинство алгоритмов в этой книге полиномиальные. Однако существует много важных задач, для которых полиномиальный алгоритм неизвестен, т. е. никто не знает, как решить их эффективно. Важным классом задач, для которых неизвестен полиномиальный алгоритм, являются *NP-трудные* задачи.

3.1.3. *Оценка эффективности*

Вычислив временную сложность алгоритма, мы можем еще до его реализации проверить, будет ли он достаточно эффективен для решения задачи. Отправной точкой для оценки является тот факт, что современный компьютер может выполнить несколько сотен миллионов простых операций в секунду.

Например, предположим, что для задачи установлено временное ограничение – не более одной секунды – и что размер входных данных равен $n = 10^5$. Если временная сложность алгоритма равна $O(n^2)$, то он должен будет выполнить порядка $(10^5)^2 = 10^{10}$ операций. Это займет, по меньшей мере, несколько десятков секунд, поэтому такой алгоритм слишком медленный для решения задачи. Но если временная сложность равна $O(n \log n)$, то количество операций будет равно всего $10^5 \log 10^5 \approx 1.6 \cdot 10^6$, так что алгоритм гарантированно укладывается в отведенные рамки.

С другой стороны, зная размер входных данных, мы можем *попытаться* угадать временную сложность алгоритма, требуемую для решения задачи. В табл. 3.1 приведены некоторые полезные оценки в предположении, что временное ограничение равно 1 секунде.

Например, если размер входных данных $n = 10^5$, то, вероятно, можно ожидать, что временная сложность алгоритма равна $O(n)$ или $O(n \log n)$. Обладание этой информацией упрощает проектирование алгоритма, поскольку сразу исключает подходы, приводящие к алгоритму с худшей временной сложностью.

При всем при том важно помнить, что временная сложность – всего лишь оценка эффективности, поскольку она скрывает постоянные множители. Например, алгоритм с временной сложностью может выполнять

как $n/2$, так и $5n$ операций, а это существенно влияет на фактическое время работы.

Таблица 3.1. Оценка временной сложности по размеру входных данных

Размер входных данных	Ожидаемая временная сложность
$n \leqslant 10$	$O(n!)$
$n \leqslant 20$	$O(2^n)$
$n \leqslant 500$	$O(n^3)$
$n \leqslant 5000$	$O(n^2)$
$n \leqslant 10^6$	$O(n \log n)$ или $O(n)$
n велико	$O(1)$ или $O(\log n)$

3.1.4. *Формальные определения*

Что *в действительности* означают слова «время работы алгоритма составляет $O(f(n))$»? Что существуют такие константы c и n_0, что алгоритм выполняет *не более* $cf(n)$ операций при любых входных данных размера $n \geqslant n_0$. Таким образом, нотация O дает *верхнюю границу* времени работы алгоритма при достаточно объемных входных данных.

Например, технически правильно будет сказать, что временная сложность следующего алгоритма равна $O(n^2)$.

```
for (int i = 1; i <= n; i++) {
    ...
}
```

Однако оценка $O(n)$ лучше, поэтому давать в этом случае оценку $O(n^2)$ – значит вводить в заблуждение читателя, т. к. любой человек предполагает, что нотация O служит для точной оценки временной сложности.

На практике часто применяются еще два варианта нотации. Буквой Ω обозначается *нижняя граница* времени работы алгоритма. Временная сложность алгоритма равна $\Omega(f(n))$, если существуют константы c и n_0 – такие, что алгоритм выполняет *не менее* $cf(n)$ операций при любых входных данных размера $n \geqslant n_0$. Наконец, буквой Θ обозначается *точная граница*: временная сложность алгоритма равна $\Theta(f(n))$, если она одновременно равна $O(f(n))$ и $\Omega(f(n))$. Так, временная сложность приведенного выше алгоритма одновременно равна $O(n)$ и $\Omega(n)$, поэтому она также равна $\Theta(n)$.

Описанная нотация используется во многих ситуациях, а не только в контексте временной сложности алгоритмов. Например, можно сказать, что массив содержит $O(n)$ элементов или что алгоритм состоит из $O(\log n)$ раундов.

3.2. Примеры проектирования алгоритмов

В этом разделе мы обсудим две задачи проектирования алгоритмов, которые можно решить несколькими способами. Начнем с простых алгоритмов с полным перебором, а затем предложим более эффективные решения, основанные на различных идеях.

3.2.1. *Максимальная сумма подмассивов*

Пусть дан массив n чисел; наша первая задача заключается в том, чтобы вычислить *максимальную сумму подмассивов*, т. е. наибольшую возможную сумму последовательных элементов. Задача приобретает интерес, когда в массиве встречаются элементы с отрицательными значениями. На рис. 3.1 показаны массив и его подмассив с максимальной суммой.

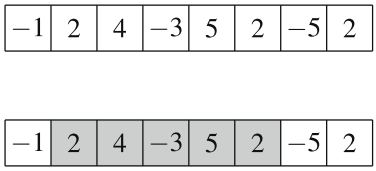


Рис. 3.1. Подмассивом с максимальной суммой является [2,4,–3,5,2]. Его сумма равна 10

Решение со сложностью $O(n^3)$. Задачу можно решить в лоб: перебрать все возможные подмассивы, вычислить сумму элементов в каждом подмассиве и запомнить максимальную сумму. Этот алгоритм реализован в следующем коде:

```
int best = 0;
for (int a = 0; a < n; a++) {
    for (int b = a; b < n; b++) {
        int sum = 0;
        for (int k = a; k <= b; k++) {
            sum += array[k];
        }
        best = max(best,sum);
    }
}
cout << best << "\n";
```

В переменных `a` и `b` хранятся первый и последний индекс подмассива, а в переменную `sum` записывается сумма его элементов. В переменной `best` хранится максимальная сумма, найденная в процессе просмотра. Временная сложность этого алгоритма равна $O(n^3)$, поскольку налицо три вложенных цикла, в которых перебираются входные данные.

Решение со сложностью $O(n^2)$. Алгоритм легко сделать более эффективным, исключив один цикл. Для этого будем вычислять сумму одновременно со сдвигом правого конца подмассива. В результате получается такой код:

```
int best = 0;
for (int a = 0; a < n; a++) {
    int sum = 0;
    for (int b = a; b < n; b++) {
        sum += array[b];
        best = max(best,sum);
    }
}
cout << best << "\n";
```

После подобного изменения временная сложность становится равна $O(n^2)$.

Решение со сложностью $O(n)$. Оказывается, что задачу можно решить и за время $O(n)$, т. е. достаточно и одного цикла. Идея в том, чтобы для каждой позиции массива вычислять максимальную сумму подмассива, заканчивающегося в этой позиции. Тогда для получения окончательного ответа достаточно будет взять максимум из этих сумм.

Рассмотрим подзадачу нахождения подмассива с максимальной суммой, заканчивающегося в позиции k. Есть две возможности:

1. Подмассив состоит из единственного элемента в позиции k.
2. Подмассив состоит из подмассива, заканчивающегося в позиции $k - 1$, за которым следует элемент в позиции k.

Во втором случае, поскольку мы ищем подмассив с максимальной суммой, сумма подмассива, заканчивающегося в позиции $k - 1$, также должна быть максимальной. Таким образом, задачу можно решить эффективно, если вычислять сумму максимального подмассива для каждой позиции последнего элемента слева направо.

Алгоритм реализуется следующей программой:

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
    sum = max(array[k],sum+array[k]);
    best = max(best,sum);
}
cout << best << "\n";
```

В этом алгоритме только один цикл, в котором перебираются входные данные, поэтому его временная сложность равна $O(n)$. Лучшей сложности

добиться нельзя, поскольку любой алгоритм решения этой задачи должен хотя бы один раз проанализировать каждый элемент.

Сравнение эффективности. Насколько приведенные выше алгоритмы эффективны на практике? В табл. 3.2 показано время выполнения этих алгоритмов на современном компьютере для разных значений n. В каждом тесте входные данные генерировались случайным образом, и время чтения входных данных не учитывалось.

Сравнение показывает, что все алгоритмы работают быстро, если размер входных данных мал, но по мере возрастания размера расхождение во времени становится очень заметным. Алгоритм со сложностью $O(n^3)$ замедляется, когда $n = 10^4$, а алгоритм со сложностью $O(n^2)$ – при $n = 10^5$. И лишь алгоритм со сложностью $O(n)$ даже самые большие входные данные обрабатывает мгновенно.

Таблица 3.2. Сравнение времени выполнения различных алгоритмов вычисления максимальной суммы подмассивов

Размер массива n	$O(n^3)$ (с)	$O(n^2)$ (с)	$O(n)$ (с)
10^2	0.0	0.0	0.0
10^3	0.1	0.0	0.0
10^4	> 10.0	0.1	0.0
10^5	> 10.0	5.3	0.0
10^6	> 10.0	> 10.0	0.0
10^7	> 10.0	> 10.0	0.0

3.2.2. *Задача о двух ферзях*

Следующая наша задача будет такой: сколькими способами можно поставить на доску $n \times n$ *двух* ферзей, так чтобы они не били друг друга. На рис. 3.2 показано, что на доске 3×3 это можно сделать 8 способами. Обозначим $q(n)$ количество расстановок на доске $n \times n$. Например, $q(3) = 8$, а в табл. 3.3 приведены значения $q(n)$ для $1 \leqslant n \leqslant 10$.

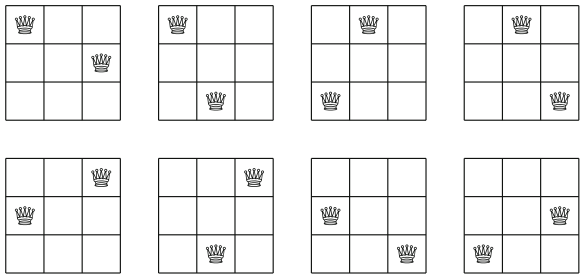


Рис. 3.2. Все возможные способы поставить двух ферзей на доску 3×3, так чтобы они не били друг друга

Таблица 3.3. Значения $q(n)$: число способов поставить двух ферзей на доску $n \times n$, так чтобы они не били друг друга

Размер доски	Число расстановок $q(n)$
1	0
2	0
3	8
4	44
5	140
6	340
7	700
8	1288
9	2184
10	3480

Конечно, задачу можно решить по-простому: перебрать все возможные расстановки двух ферзей и подсчитать те, в которых ферзи не бьют друг друга. Временная сложность такого алгоритма $O(n^4)$, поскольку позицию первого ферзя можно выбрать n^2 способами, а затем поставить второго ферзя $n^2 - 1$ способами.

Так как число расстановок растет очень быстро, алгоритм их подсчета поодиночке будет работать слишком медленно для больших n. Следовательно, для создания эффективного алгоритма нужно придумать способ подсчета расстановок *группами*. Полезно отметить, что очень просто подсчитать, сколько клеток бьет один ферзь (рис. 3.3). Во-первых, он всегда бьет $n - 1$ клеток по горизонтали и $n - 1$ клеток по вертикали. Далее по обеим диагоналям ферзь бьет $d - 1$ клеток, где d – число клеток на диагонали. С учетом всего этого для вычисления числа клеток, в которые можно поставить второго ферзя, нужно время $O(1)$, так что мы получаем алгоритм с временной сложностью $O(n^2)$.

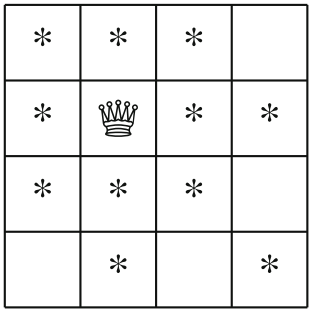


Рис. 3.3. Ферзь бьет все клетки, помеченные знаком *

К задаче можно подойти и по-другому, попытавшись найти рекуррентную формулу для подсчета расстановок, т. е. ответить на вопрос: как, зная $q(n)$, вычислить $q(n + 1)$?

Чтобы получить рекурсивное решение, обратим внимание на последние горизонталь и вертикаль доски $n \times n$ (рис. 3.4). Во-первых, число расстановок, в которых на последней горизонтали и на последней вертикали нет ферзей, равно $q(n - 1)$. Во-вторых, общее число позиций ферзя на последней горизонтали и на последней вертикали равно $2n - 1$. Находящийся в этих позициях ферзь бьет $3(n - 1)$ клеток, поэтому для второго ферзя остается $n^2 - 3(n - 1) - 1$ позиций. Наконец, существует $(n - 1)(n - 2)$ расстановок, в которых оба ферзя находятся на последней горизонтали или на последней вертикали. Поскольку эти расстановки посчитаны дважды, их количество надо вычесть. Объединив все вместе, получаем рекуррентную формулу

$$\begin{aligned} q(n) &= q(n-1) + (2n-1)(n^2 - 3(n-1) - 1) - (n-1)(n-2) \\ &= q(n-1) + 2(n-1)^2(n-2) \end{aligned}$$

и вместе с ней решение сложности $O(n)$.

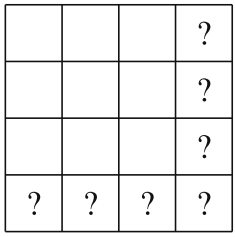


Рис. 3.4. Возможные позиции ферзей на последней горизонтали и вертикали

Однако же существует и замкнутая формула для этой функции:

$$q(n) = \frac{n^4}{2} - \frac{5n^3}{3} + \frac{3n^2}{2} - \frac{n}{3},$$

которую можно доказать по индукции, пользуясь рекуррентной формулой. А раз так, то задачу можно решить за время $O(1)$.

3.3. Оптимизация кода

Временная сложность алгоритма может многое сказать о его эффективности – это правда, но важны и детали реализации. Вот, к примеру, два фрагмента кода, в которых проверяется, есть ли в массиве элемент x:

```
bool ok = false;
for (int i = 0; i < n; i++) {
```

```
    if (a[i] == x) ok = true;
}

bool ok = false;
for (int i = 0; i < n; i++) {
    if (a[i] == x) {ok = true; break;}
}
```

Оба фрагмента работают за время $O(n)$, однако на практике второй может оказаться значительно эффективнее, потому что прекращает работу, как только элемент x найден. Это полезная оптимизация, потому что она действительно повышает производительность кода и к тому же легко реализуется.

Можно ли еще улучшить этот код? Существует классический прием, который можно попробовать: воспользоваться *сторожевым значением*, т. е. добавить в конец массива новый элемент со значением x. Тогда не нужно будет проверять условие $i < n$ в цикле:

```
a[n] = x;
int i;
bool ok = false;
for (i = 0; a[i] != x; i++);
if (i < n) ok = true;
```

Прием изящный, но на практике не особенно полезен: «затык» не в проверке условия $i < n$, поскольку доступ к элементам массива занимает гораздо больше времени. Таким образом, не все оптимизации полезны, некоторые лишь затрудняют понимание кода.

3.3.1. *Результат работы компилятора*

Компилятор C++ преобразует написанный на C++ код в машинный, исполняемый процессором. Важная задача компилятора – *оптимизация* кода. Результирующий код должен быть эквивалентен коду на C++, но при этом работать как можно быстрее. Часто количество возможных оптимизаций велико.

Мы можем получить машинный код (в формате ассемблера), порожденный компилятором g++, задав флаг `-S`:

```
g++ -S test.cpp -o test.out
```

Эта команда создает файл `test.out`, содержащий код на языке ассемблера. Существует также полезный онлайновый инструмент *Compiler Explorer*[1], которым можно воспользоваться для исследования результатов работы различных компиляторов, включая g++.

[1] https://godbolt.org/.

Оптимизации компилятора. Например, рассмотрим следующий код на C++:

```
int collatz(int n) {
    if (n%2 == 0) return n/2;
    else return 3*n+1;
}
```

Ассемблерный код, порожденный g++ (с флагом оптимизации -O2), может выглядеть так:

```
        test    dil, 1
        jne     .L2
        mov     eax, edi
        shr     eax, 31
        add     eax, edi
        sar     eax
        ret
.L2:
        lea     eax, [rdi+1+rdi*2]
        ret
```

Даже в этом крохотном кусочке кода много оптимизаций. Команда `test` проверяет, равен ли 1 самый правый бит n, т. е. является ли число нечетным, и она быстрее, чем операция взятия остатка от деления. Затем команда `sar` выполняет поразрядный сдвиг вправо на один бит, т. е. вычисляет значение $n/2$, причем быстрее, чем операция деления. Наконец, для вычисления значения $3n + 1$ применяется еще один трюк: основная задача команды `lea` – определить *адрес* элемента массива в памяти, но ее можно использовать и для простых вычислений.

Зачастую необязательно оптимизировать программу самостоятельно (например, отдавать предпочтения побитовым операциям вместо деления и взятия остатка), потому что компилятор C++ знает все эти приемы и умеет применять их. Кроме того, компилятор умеет находить и удалять ненужный код. Рассмотрим, например, следующую функцию:

```
void test(int n) {
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s += i;
    }
}
```

Ей соответствует всего одна строка на ассемблере:

```
        ret
```

означающая, что мы просто выходим из функции. Поскольку значение *s* нигде не используется, саму эту переменную и весь цикл можно удалить, и код будет работать за время $O(1)$. Поэтому при измерении времени работы кода важно следить за тем, чтобы его результат как-то использовался (например, можно его напечатать), иначе компилятор может удалить код в процессе оптимизации.

Оптимизации, зависящие от оборудования. Флаг `g++ -march=native` включает оптимизации, зависящие от оборудования. Например, в некоторых процессорах имеются специальные команды, отсутствующие в других процессорах. Слово `native` здесь означает, что компилятор автоматически определяет архитектуру процессора и применяет зависящие от нее оптимизации там, где это возможно.

Например, рассмотрим следующий код, который вычисляет сумму единичных битов с помощью встроенной в `g++` функции `__builtin_popcount`:

```
c = 0;
for (int i = 1; i <= n; i++) {
    c += __builtin_popcount(i);
}
```

Во многих процессорах имеется специальная команда `popcnt`, которая эффективно выполняет операции подсчета битов. Во многих, но не во всех, поэтому `g++` не использует ее автоматически, для этого нужно задать флаг `-march=native`. И в таком случае приведенный выше код будет работать в два-три раза быстрее.

Флаг `-march=native` нечасто задается в олимпиадных системах, но мы можем сами указать архитектуру в своем коде с помощью директивы `#pragma`. Однако в этом контексте значение `native` не поддерживается, требуется задать имя архитектуры. Так, возможно, будет работать следующая директива (в предположении, что программа исполняется процессором с архитектурой Sandy Bridge):

```
#pragma GCC target ("arch=sandybridge")
```

3.3.2. *Особенности процессора*

Исполняя код, процессоры стараются делать это как можно быстрее. Существуют кеши, ускоряющие доступ к памяти, а иногда несколько команд можно выполнять параллельно. Современные процессоры очень сложны, и мало кто понимает, как они в действительности работают.

Кеши. Поскольку доступ к основной памяти сравнительно медленный, в процессорах имеются кеши – участки небольшого объема памяти с более быстрым доступом. Кеши автоматически используются, когда читается или записывается содержимое расположенных рядом ячеек памяти.

В частности, просмотр элементов массива слева направо производится быстро, а в случайном порядке – медленно.

В качестве примера рассмотрим два таких фрагмента:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        s += x[i][j];
    }
}

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        s += x[j][i];
    }
}
```

В обоих случаях вычисляется сумма элементов двумерного массива, но первый вариант работает намного эффективнее, потому что *дружелюбен к кешу*. Элементы массива хранятся в памяти в таком порядке:

$$x[0][0], x[0][1], \ldots, x[0][n-1], x[1][0], x[1][1], \ldots$$

Поэтому лучше, когда во внешнем цикле изменяется первый индекс, а во внутреннем – второй.

Распараллеливание. Современные процессоры могут выполнять несколько команд одновременно, и во многих ситуациях это происходит автоматически. Вообще говоря, две соседние команды могут выполняться параллельно, если они не зависят друг от друга. Рассмотрим такой код:

```
ll f = 1;
for (int i = 1; i <= n; i++) {
    f = (f*i)%M;
}
```

Здесь в цикле вычисляется факториал n по модулю M. Можно попробовать сделать этот код эффективнее, поступив следующим образом (в предположении, что n четно):

```
ll f1 = 1;
ll f2 = 1;
for (int i = 1; i <= n; i += 2) {
    f1 = (f1*i)%M;
    f2 = (f2*(i+1))%M;
}
ll f = f1*f2%M;
```

Идея в том, чтобы использовать две независимые переменные: f_1 будет содержать произведение $1 \cdot 3 \cdot 5 \cdot \ldots \cdot (n-1)$, а f_2 – произведение $2 \cdot 4 \cdot 6 \cdot \ldots \cdot n$. По завершении цикла оба результата объединяются. Этот код обычно работает в два раза быстрее первоначальной версии, потому что процессор может параллельно выполнять команды, изменяющие переменные f_1 и f_2 внутри цикла. Можно даже завести больше переменных (скажем, четыре или восемь) и тем самым еще увеличить скорость работы.

Глава 4

Сортировка и поиск

Многие эффективные алгоритмы основаны на сортировке входных данных, поскольку благодаря сортировке задачу часто можно решить проще. В этой главе обсуждаются теория и практика сортировки как средства проектирования алгоритмов.

В разделе 4.1 рассматриваются три важных алгоритма сортировки: пузырьковая, слиянием и подсчетом. Затем мы научимся использовать алгоритм сортировки, включенный в стандартную библиотеку C++.

В разделе 4.2 показано, как создавать эффективные алгоритмы, используя сортировку в качестве подпрограммы. Например, чтобы быстро определить, являются ли все элементы массива уникальными, можно сначала отсортировать массив, а затем просто проверить пары соседних элементов.

В разделе 4.3 представлен алгоритм двоичного поиска – еще один важный элемент построения эффективных алгоритмов.

4.1. Алгоритмы сортировки

Основная задача сортировки формулируется следующим образом: дан массив, содержащий n элементов, требуется расположить элементы в порядке возрастания. Так, на рис. 4.1 изображен массив до и после сортировки.

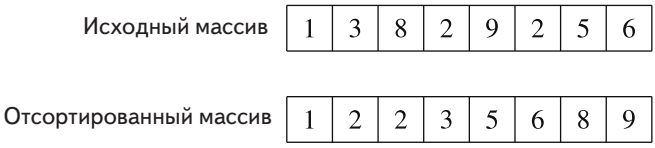


Рис. 4.1. Массив до и после сортировки

В этом разделе мы разберем некоторые общеизвестные алгоритмы сортировки и изучим их свойства. Легко спроектировать алгоритм сортировки с временной сложностью $O(n^2)$, но существуют и более эффективные алгоритмы. Обсудив теорию сортировки, мы перейдем к ее практическому использованию в программах на C++.

4.1.1. *Пузырьковая сортировка*

Пузырьковая сортировка – простой алгоритм с временной сложностью $O(n^2)$. Он состоит из n раундов, на каждом из которых обходятся элементы массива. Если два соседних элемента расположены не в том порядке, то они переставляются. Реализовать алгоритм можно следующим образом:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j],array[j+1]);
        }
    }
}
```

После первого раунда самый большой элемент окажется в правильной позиции. Вообще, после k раундов в правильных позициях будут находиться k самых больших элементов. Следовательно, после n раундов весь массив будет отсортирован.

На рис. 4.2 показан первый раунд пузырьковой сортировки массива.

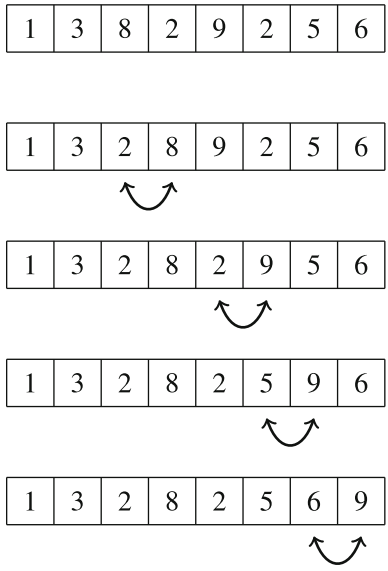


Рис. 4.2. Первый раунд пузырьковой сортировки

Пузырьковая сортировка – пример алгоритма сортировки, в котором переставляются только *соседние* элементы массива. Временная сложность такого алгоритма *обязательно* не лучше $O(n^2)$, потому что в худшем случае потребуется выполнить $O(n^2)$ перестановок.

Инверсии. Для анализа алгоритмов сортировки полезно понятие *инверсии*. Так называется пара индексов массива (a, b) – такая, что $a < b$ и `array`[a] >`array`[b], т. е. элементы массива расположены не в том порядке, в каком нужно. На рис. 4.3 имеется три инверсии: (3, 4), (3, 5) и (6, 7).

0	1	2	3	4	5	6	7
1	2	2	6	3	5	9	8

Рис. 4.3. В этом массиве три инверсии: (3, 4), (3, 5) и (6, 7)

От количества инверсий зависит объем работы, которую нужно проделать для сортировки массива. Массив полностью отсортирован, когда в нем нет инверсий. С другой стороны, если элементы массива расположены в обратном порядке, то число инверсий максимально и равно

$$1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2} = O(n^2),$$

Перестановка пары соседних элементов, расположенных не по порядку, устраняет ровно одну инверсию. Поэтому если алгоритму разрешается переставлять только соседние элементы, то его временная сложность не может быть лучше $O(n^2)$.

4.1.2. *Сортировка слиянием*

Чтобы создать эффективный алгоритм сортировки, требуется переупорядочивать элементы, находящиеся в разных частях массива. Существует несколько таких алгоритмов, работающих за время $O(n \log n)$. Один из них – сортировка слиянием – основан на рекурсии. Этот алгоритм сортирует подмассив `array`[a … b] следующим образом:

1. Если $a = b$, не делать ничего, поскольку подмассив, содержащий всего один элемент, уже отсортирован.
2. Вычислить позицию среднего элемента: $k = \lfloor (a + b)/2 \rfloor$.
3. Рекурсивно отсортировать подмассив `array`[a … k].
4. Рекурсивно отсортировать подмассив `array`[$k + 1$ … b].
5. *Слить* отсортированные подмассивы `array`[a … k] и `array`[$k + 1$ … b] в отсортированный подмассив `array`[a … b].

На рис. 4.4 показано, как сортировка слиянием применяется к массиву из восьми элементов. Сначала массив разбивается на два подмассива из четырех элементов. Затем каждый массив сортируется рекурсивно. И наконец, отсортированные подмассивы сливаются, образуя отсортированный массив из восьми элементов.

Сортировка слиянием – эффективный алгоритм, поскольку на каждом шаге размер подмассива уменьшается вдвое. А для слияния уже отсорти-

рованных подмассивов достаточно линейного времени. Так как уровней рекурсии $O(\log n)$, а обработка на каждом уровне занимает время $O(n)$, то временная сложность алгоритма равна $O(n \log n)$.

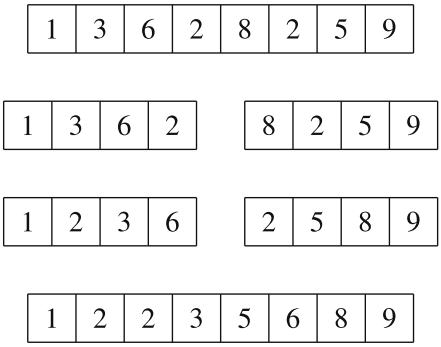


Рис. 4.4. Сортировка массива слиянием

4.1.3. *Нижняя граница временной сложности сортировки*

Можно ли отсортировать массив быстрее, чем за время $O(n \log n)$? Оказывается, что нет, если ограничиться алгоритмами, основанными на сравнении элементов.

Чтобы доказать нижнюю границу временной сложности, будем рассматривать сортировку как процесс, в котором сравнение двух элементов дает дополнительную информацию о массиве. На рис. 4.5 показано дерево, создаваемое в ходе этого процесса.

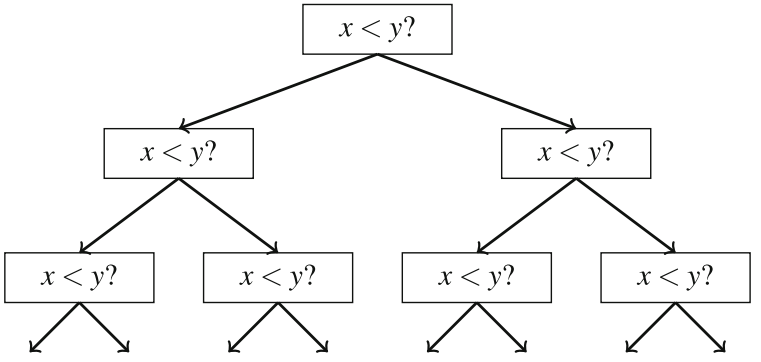


Рис. 4.5. Ход выполнения алгоритма сортировки – сравнение элементов массива

Здесь «$x < y$?» означает, что сравниваются элементы x и y. Если $x < y$, то выбирается левая ветвь дерева, иначе – правая. Результатом процесса является множество всех возможных способов расположить элементы массива, всего их $n!$. Поэтому высота дерева должна быть не меньше

$$\log_2(n!) = \log_2(1) + \log_2(2) + \ldots + \log_2(n).$$

Чтобы получить нижнюю оценку этой суммы, оставим только последние $n/2$ элементов и заменим каждый из них на $\log_2(n/2)$. Это дает оценку

$$\log_2(n!) \geqslant (n/2) \cdot \log_2(n/2),$$

так что высота дерева и число шагов алгоритма в худшем случае равны $\Omega(n \log n)$.

4.1.4. *Сортировка подсчетом*

Нижняя граница $\Omega(n \log n)$ неприменима к алгоритмам, которые не сравнивают элементы массива, а используют какую-то другую информацию. Примером такого алгоритма является *сортировка подсчетом*, при которой массив сортируется за время $O(n)$ в предположении, что все его элементы – целые числа от 0 до c и $c = O(n)$.

Мы будем использовать вспомогательный массив, индексы которого соответствуют элементам исходного массива. Алгоритм обходит исходный массив и вычисляет, сколько раз в нем встречается каждый элемент. На рис. 4.6 показаны исходный массив и соответствующий ему вспомогательный массив. Например, в позиции 3 находится значение 2, поскольку значение 3 встречается в исходном массиве 2 раза.

1	3	6	9	9	3	5	9

0	1	2	3	4	5	6	7	8	9
0	1	0	2	0	1	1	0	0	3

Рис. 4.6. Сортировка массива подсчетом

Построение вспомогательного массива занимает время $O(n)$. После этого создать отсортированный массив можно за время $O(n)$, поскольку из вспомогательного массива можно узнать, сколько раз встречается каждый элемент. Таким образом, полная временная сложность сортировки подсчетом равна $O(n)$.

Сортировка подсчетом – очень эффективный алгоритм, но применим он, только если константа c достаточно мала, так что элементы исходного массива можно использовать в качестве индексов вспомогательного массива.

4.1.5. *Сортировка на практике*

На практике почти никогда не стоит заниматься реализацией доморощенного алгоритма сортировки, поскольку в стандартных библиотеках всех современных языков программирования уже имеются отличные реа-

лизации. Причин выбрать библиотечную функцию много: она наверняка правильна, эффективна, да к тому же ее легко использовать.

В C++ функция `sort` эффективно[1] сортирует содержимое структуры данных. Например, в следующем фрагменте элементы вектора сортируются в порядке возрастания:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

После сортировки вектор будет иметь вид [2, 3, 3, 4, 5, 5, 8]. По умолчанию сортировка производится в порядке возрастания, но можно отсортировать и в порядке убывания:

```
sort(v.rbegin(),v.rend());
```

Обычный массив можно отсортировать следующим образом:

```
int n = 7; // размер массива
int a[] = {4,2,5,3,5,8,3};
sort(a,a+n);
```

А ниже показано, как отсортировать строку `s`:

```
string s = "monkey";
sort(s.begin(), s.end());
```

Под сортировкой строки понимается расположение ее символов в алфавитном порядке. Например, строка «monkey» превращается в «ekmnoy».

Операторы сравнения. Чтобы функция `sort` работала, для типа сортируемых данных должен быть определен *оператор сравнения*. Во время сортировки он будет использоваться, когда требуется определить, какой из двух элементов больше.

В большинстве типов данных C++ имеются встроенные операторы сравнения, так что элементы таких типов будут сортироваться автоматически. Числа сортируются по значениям, строки – в лексикографическом порядке. Пары сортируются сначала по первым элементам, а если они равны, то по вторым:

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
// результат: [(1,2),(1,5),(2,3)]
```

[1] Стандарт C++11 требует, чтобы функция `sort` работала за время $O(n \log n)$, фактическая реализация зависит от компилятора.

Точно так же кортежи сортируются сначала по первым элементам, в случае их совпадения – по вторым и т. д.[2]:

```
vector<tuple<int,int,int>> v;
v.push_back({2,1,4});
v.push_back({1,5,3});
v.push_back({2,1,3});
sort(v.begin(), v.end());
// результат: [(1,5,3),(2,1,3),(2,1,4)]
```

В определенных пользователем классах оператор сравнения автоматически не появляется. Его следует определить в виде функции `operator<`, которая принимает в качестве параметра другой элемент того же типа. Оператор должен возвращать `true`, если объект, от имени которого он вызывается, меньше параметра, и `false` в противном случае.

Например, показанная ниже структура `point` содержит координаты точки `x` и `y`. Оператор сравнения определен так, что точки сортируются сначала по координате `x`, а в случае совпадения – по координате `y`.

```
struct point {
    int x, y;
    bool operator<(const point &p) {
        if (x == p.x) return y < p.y;
        else return x < p.x;
    }
};
```

Функции сравнения. Можно также определить внешнюю *функцию сравнения* и передать ее функции `sort` как функцию обратного вызова. Например, следующая функция сравнивает строки сначала по длине, а при совпадении – в лексикографическом порядке:

```
bool comp(string a, string b) {
    if (a.size() == b.size()) return a < b;
    else return a.size() < b.size();
}
```

Теперь вектор строк можно отсортировать следующим образом:

```
sort(v.begin(), v.end(), comp);
```

[2] Отметим, что в некоторых старых компиляторах для создания кортежа нужно было использовать функцию `make_tuple`, а не просто фигурные скобки (например, `make_tuple(2,1,4)` вместо `{2,1,4}`).

4.2. Решение задач с применением сортировки

Бывает, что задачу легко решить за время $O(n^2)$ с помощью полного перебора, но такой алгоритм будет работать долго, если размер данных велик. Вообще, при проектировании алгоритмов часто требуется найти алгоритм с временной сложностью $O(n)$ или $O(n \log n)$ для задачи, которая тривиально решается за время $O(n^2)$. Один из способов добиться этой цели – применить сортировку.

Допустим, к примеру, что мы хотим проверить, все ли элементы массива уникальны. Можно было бы просто сравнить все пары элементов за время $O(n^2)$:

```
bool ok = true;
for (int i = 0; i < n; i++) {
   for (int j = i+1; j < n; j++) {
      if (array[i] == array[j]) ok = false;
   }
}
```

Однако задачу можно решить за время $O(n \log n)$, если сначала отсортировать массив. Тогда все равные элементы окажутся рядом, поэтому их легко найти за время $O(n)$:

```
bool ok = true;
sort(array, array+n);
for (int i = 0; i < n-1; i++) {
   if (array[i] == array[i+1]) ok = false;
}
```

Есть еще несколько похожих задач, которые тоже решаются за время $O(n \log n)$, например подсчет числа различных элементов, нахождение самого часто встречающегося элемента и нахождение двух элементов с минимальной разностью.

4.2.1. *Алгоритмы заметающей прямой*

В алгоритме *заметающей прямой* задача моделируется как множество событий, обрабатываемых в определенном порядке. Пусть имеется ресторан, и мы знаем время прихода и ухода всех клиентов в течение дня. Задача состоит в том, чтобы найти максимальное число клиентов, находившихся в ресторане одновременно.

На рис. 4.7 показан пример с четырьмя клиентами: *A*, *B*, *C* и *D*. В данном случае максимальное число одновременно присутствующих клиентов равно 3: в промежутке между приходом *A* и уходом *B*.

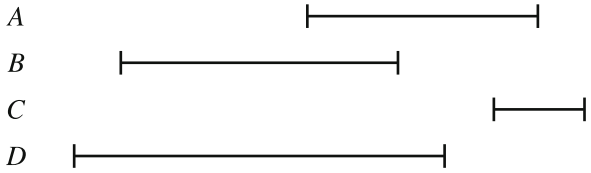


Рис. 4.7. Пример задачи о ресторане

Для решения задачи будем создавать для каждого клиента два события: приход и уход. Затем отсортируем события по времени и обойдем их. Чтобы найти максимальное число клиентов, заведем счетчик и будем увеличивать его значение, когда клиент приходит, и уменьшать, когда уходит. Наибольшее значение, принимаемое счетчиком, и будет искомым ответом.

На рис. 4.8 показаны события в нашем примере. Каждому клиенту сопоставлено два события: «+» обозначает приход, а «–» – уход. Найденный алгоритм работает за время $O(n \log n)$, потому что сортировка событий занимает время $O(n \log n)$, а заметание – время $O(n)$.

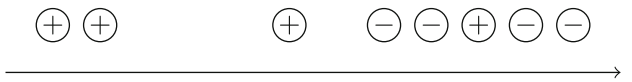


Рис. 4.8. Решение задачи о ресторане с применением алгоритма заметающей прямой

4.2.2. *Составление расписания*

Многие задачи можно решить, отсортировав входные данные, а затем воспользовавшись *жадной* стратегией для построения решения. Жадный алгоритм всегда делает выбор, который кажется наилучшим в данный момент, и впоследствии не пересматривает принятые ранее решения.

Рассмотрим следующую задачу: n событий заданы моментами начала и конца, требуется составить расписание, включающее как можно больше событий. Так, на рис. 4.9 приведен пример, когда оптимальное решение включает два события.

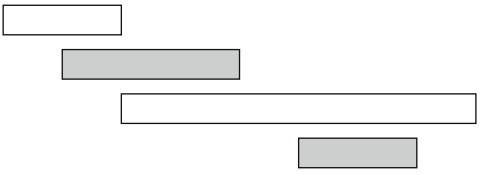


Рис. 4.9. Пример задачи о планировании и оптимальное решение с двумя событиями

В этой задаче сортировать входные данные можно несколькими способами. Первая стратегия – отсортировать события по *продолжительности* и выбирать самые *короткие* события. Но, как видно по рис. 4.10, такая стра-

тегия работает не всегда. Тогда на ум приходит другая идея – отсортировать события по *времени начала* и выбирать в качестве следующего событие, которое *начинается* как можно *раньше*. Но и для этой стратегии можно найти контрпример – см. рис. 4.11.

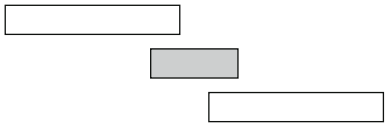


Рис. 4.10. Выбирая короткое событие, мы сможем выбрать только одно событие, хотя можно было бы выбрать оба длинных события

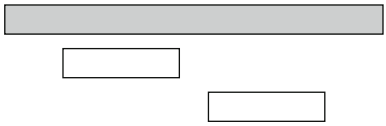


Рис. 4.11. Выбирая первое событие, мы не сможем выбрать другие, хотя можно было бы выбрать два других события

Третья идея – отсортировать события по *времени конца* и выбирать в качестве следующего событие, которое *заканчивается* как можно *раньше*. Оказывается, что этот алгоритм *всегда* дает оптимальное решение. Чтобы доказать это, рассмотрим, что произойдет, если сначала выбрать событие, которое заканчивается позже, чем событие, заканчивающееся первым. Тогда число вариантов выбора следующего события окажется никак не больше, чем если бы мы последовали предложенной стратегии. Следовательно, выбор события, заканчивающегося позже, никогда не даст лучшего решения, и, значит, жадный алгоритм правилен.

4.2.3. *Работы и сроки исполнения*

Наконец, предположим, что дано n работ и для каждой указаны продолжительность и крайний срок. Наша задача – выбрать порядок выполнения работ. Для каждой работы нам начисляется $d - x$ баллов, где d – крайний срок исполнения работы, а x – момент ее фактического завершения.

Какое максимальное число баллов мы можем заработать?

Предположим, что заданы такие работы:

Работа	Продолжительность	Крайний срок
A	4	2
B	3	10
C	2	8
D	4	15

На рис. 4.12 показано оптимальное расписание работ в этом примере. При таком расписании за *C* начисляется 6 баллов, за *B* – 5 баллов, за *A* начисляется 7 баллов, за *D* – 2 балла. Итого – 6 баллов.

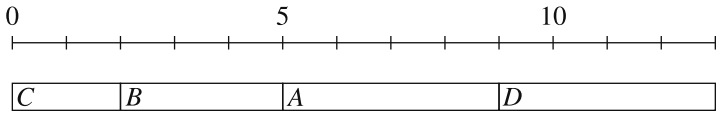


Рис. 4.12. Оптимальное расписание работ

Оказывается, что оптимальное решение задачи вообще не зависит от крайних сроков, а правильная жадная стратегия заключается в том, чтобы выполнять работы в *порядке возрастания продолжительностей*. Действительно, если в некотором решении сначала выполняется длинная работа, а за ней – более короткая, то решение можно улучшить, поменяв эти две работы местами.

На рис. 4.13 показаны две работы *X* и *Y* продолжительностью *a* и *b* соответственно. Изначально выполнение *X* запланировано раньше *Y*. Но, поскольку $a > b$, работы следует поменять местами. Теперь за *X* начисляется на *b* баллов меньше, а за *Y* – на *a* баллов больше, поэтому общее число баллов увеличивается на $a - b > 0$. Таким образом, в оптимальном решении короткие работы всегда предшествуют длинным, поэтому работы следует отсортировать по продолжительности.

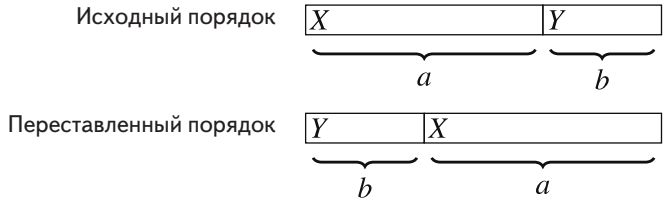


Рис. 4.13. Улучшение решения путем перестановки работ *X* и *Y*

4.3. Двоичный поиск

Двоичный поиск – это алгоритм с временной сложностью $O(\log n)$, который, например, позволяет эффективно проверить, встречается ли в массиве заданное значение. В этом разделе мы сначала рассмотрим реализацию двоичного поиска, а затем посмотрим, как ей воспользоваться для нахождения оптимальных решений задач.

4.3.1. *Реализация поиска*

Пусть дан отсортированный массив с *n* элементами, и мы хотим проверить, встречается ли в нем значение *x*. Мы обсудим два способа реализовать алгоритм двоичного поиска для решения этой задачи.

Первый метод. Самый распространенный способ реализации двоичного поиска напоминает поиск слова в словаре[3]. Определено понятие активного подмассива, который первоначально содержит все элементы массива. Алгоритм состоит из ряда шагов, на каждом из которых диапазон поиска делится пополам. На каждом шаге проверяется средний элемент активного подмассива. Если средний элемент совпадает с искомым значением, то поиск заканчивается. В противном случае поиск рекурсивно продолжается в левой или правой половине подмассива в зависимости от значения среднего элемента. На рис. 4.14 показано, как в массиве ищется элемент со значением 9.

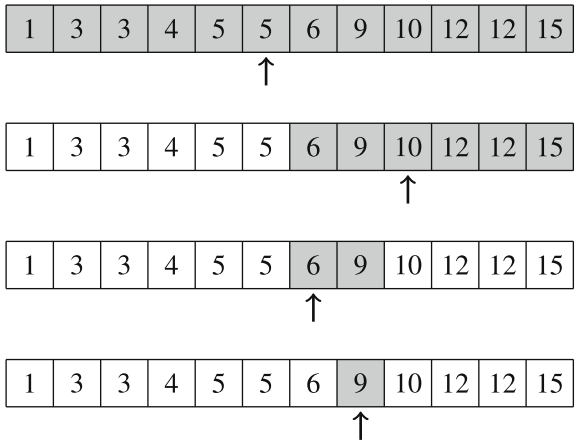


Рис. 4.14. Традиционная реализация двоичного поиска. На каждом шаге проверяется средний элемент активного подмассива, и поиск продолжается в левой или в правой части

Ниже приведена реализация поиска:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x найден в позиции с индексом k
    }
    if (array[k] < x) a = k+1;
    else b = k-1;
}
```

В этой реализации активный подмассив занимает позиции с индексами в диапазоне $a \ldots b$, а начальный диапазон равен $0 \ldots n - 1$. На каждом шаге

[3] Некоторые люди, включая автора книги, все еще пользуются печатными словарями. Другой пример – поиск телефонного номера в печатном телефонном справочнике – выглядит еще более древним.

алгоритм уменьшает размер массива вдвое, так что временная сложность составляет $O(\log n)$.

Второй метод. Другой способ реализации двоичного поиска заключается в том, чтобы просматривать массив слева направо, совершая *прыжки*. Начальная длина прыжка равна $n/2$, и на каждом шаге она уменьшается вдвое: $n/4$, $n/8$, $n/16$ и т. д., пока, наконец, не станет равна 1. На каждом шаге очередной прыжок совершается при условии, что мы не приземлимся за пределами массива или на элементе, значение которого превышает искомое. После всех прыжков либо искомый элемент будет найден, либо он заведомо не встречается в массиве. На рис. 4.15 показано применение этого метода на конкретном примере.

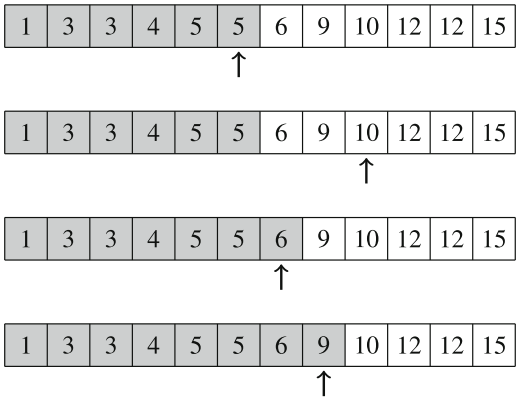


Рис. 4.15. Альтернативная реализация двоичного поиска.
Мы просматриваем массив слева направо, перепрыгивая через элементы

Ниже приведена реализация:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x найден в позиции с индексом k
}
```

Здесь переменная *b* содержит текущую длину прыжка. Временная сложность алгоритма равна $O(\log n)$, потому что код в цикле `while` выполняется не более двух раз для каждой длины прыжка.

4.3.2. *Двоичный поиск по ответу*

Пусть мы решаем некоторую задачу, и существует функция `valid`(x), возвращающая `true`, если x – допустимое решение, и `false` в противном

случае. Кроме того, мы знаем, что valid(x) равно false, когда $x < k$, и равно true, когда $x \geqslant k$. В такой ситуации можно использовать двоичный поиск для эффективного нахождения k.

Идея в том, чтобы с помощью двоичного поиска найти наибольшее значение x, для которого valid(x) равно false. Тогда следующее число $k = x + 1$ – наименьшее значение, для которого valid(k) равно true. Поиск можно реализовать следующим образом:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
   while (!valid(x+b)) x += b;
}
int k = x+1;
```

Начальная длина прыжка z должна быть верхней границей ответа, т. е. любым значением, для которого точно известно, что valid(z) равно true. Алгоритм вызывает функцию valid $O(\log z)$ раз, поэтому время работы зависит от функции valid. Так, если эта функция работает время $O(n)$, то сложность алгоритма равна $O(n \log z)$.

Пример. Рассмотрим следующую задачу: нам нужно выполнить k заданий на n машинах. Каждой машине i сопоставлено целое число p_i – время выполнения одного задания. За какое минимальное время можно обработать все задания?

Пусть $k = 8$, $n = 3$, а времена обработки $p_1 = 2$, $p_2 = 3$, $p_3 = 7$. В этом случае минимальное общее время обработки равно 9, как следует из плана, показанного на рис. 4.16.

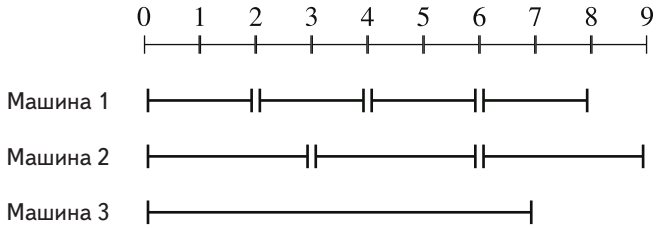


Рис. 4.16. Оптимальное планирование обработки: машина 1 выполняет четыре задания, машина 2 – три задания, машина 3 – одно задание

Назовем valid(x) функцию, которая определяет, можно ли обработать все задания, используя не более x единиц времени. В нашем примере valid(9), очевидно, равно true, что доказывается планом на рис. 4.16. С другой стороны, valid(8) должно быть равно false, потому что минимальное время обработки равно 9.

Вычислить значение valid(x) легко, потому что каждая машина i может выполнить не более $\lfloor x/p_i \rfloor$ заданий за x единиц времени. Следовательно,

если сумма всех $\lfloor x/p_i \rfloor$ равна k или больше, то x – допустимое решение. Далее мы можем применить двоичный поиск, чтобы найти минимальное значение x, для которого `valid`(x) равно `true`.

Насколько эффективен получившийся алгоритм? Вычисление функции `valid` занимает время $O(n)$, поэтому алгоритм работает за время $O(n \log z)$, где z – верхняя граница ответа. Одно из возможных значений z равно kp_1, оно соответствует решению, в котором все задания выполняет первая машина. Очевидно, что это действительно верхняя граница.

Глава 5

Структуры данных

В этой главе мы познакомимся с наиболее важными структурами данных из стандартной библиотеки C++. В олимпиадном программировании чрезвычайно важно знать, какие структуры данных имеются в стандартной библиотеке и как ими пользоваться. Часто это позволяет сэкономить много времени при реализации алгоритма.

В разделе 5.1 сначала описывается структура вектора, который по существу является эффективным динамическим массивом. Затем обсуждаются двусторонние очереди (деки), стеки и очереди.

В разделе 5.2 обсуждаются множества, отображения и очереди с приоритетом. Эти динамические структуры данных часто являются строительными блоками алгоритмов, поскольку поддерживают эффективный поиск и обновление.

В разделе 5.3 приведены некоторые результаты, касающиеся практической эффективности структур данных. Как мы увидим, существуют важные различия в производительности, которые нельзя обнаружить, зная только оценки временной сложности.

5.1. Динамические массивы

В C++ обыкновенные массивы – это структуры фиксированного размера, т. е. после создания изменить размер уже нельзя. Например, в следующем фрагменте создается массив, содержащий n целых значений:

```
int array[n];
```

Динамическим называется массив, размер которого можно изменять в процессе выполнения программы. В стандартной библиотеке C++ имеется несколько динамических массивов, но полезнее всего вектор.

5.1.1. *Векторы*

Вектор – это динамический массив, который позволяет эффективно добавлять элементы в конце и удалять последние элементы. Например, в следующем коде создается пустой вектор, в который затем добавляется три элемента:

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]
```

Доступ к элементам осуществляется так же, как в обычном массиве:

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

Еще один способ создать вектор – перечислить все его элементы:

```
vector<int> v = {2,4,2,5,1};
```

Можно также задать число элементов и их начальное значение:

```
vector<int> a(8); // размер 8, начальное значение 0
vector<int> b(8,2); // размер 8, начальное значение 2
```

Функция `size` возвращает число элементов вектора. В следующем коде мы обходим вектор и печатаем его элементы:

```
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << "\n";
}
```

Обход вектора можно записать и короче:

```
for (auto x : v) {
    cout << x << "\n";
}
```

Функция `back` возвращает последний элемент вектора, а функция `pop_back` удаляет последний элемент:

```
vector<int> v = {2,4,2,5,1};
cout << v.back() << "\n"; // 1
v.pop_back();
cout << v.back() << "\n"; // 5
```

Векторы реализованы так, что функции `push_back` и `pop_back` в среднем имеют сложность $O(1)$. На практике работать с вектором почти так же быстро, как с массивом.

5.1.2. *Итераторы и диапазоны*

Итератором называется переменная, которая указывает на элемент структуры данных. Итератор `begin` указывает на первый элемент струк-

туры, а итератор end – на позицию *за* последним элементом. Например, в случае вектора v, состоящего из восьми элементов, ситуация может выглядеть так:

```
[ 5, 2, 3, 1, 2, 5, 7, 1 ]
  ↑                    ↑
  v.begin()            v.end()
```

Обратите внимание на асимметрию итераторов: begin() указывает на элемент, принадлежащий структуре данных, а end() ведет *за пределы* структуры данных.

Диапазоном называется последовательность соседних элементов структуры данных. Чаще всего диапазон задается с помощью двух итераторов: указывающего на первый элемент и на позицию за последним элементом. В частности, итераторы begin() и end() определяют диапазон, содержащий все элементы структуры данных.

Функции из стандартной библиотеки C++ обычно применяются к диапазонам. Так, в следующем фрагменте сначала вектор сортируется, затем порядок его элементов меняется на противоположный, и, наконец, элементы перемешиваются в случайном порядке.

```
sort(v.begin(),v.end());
reverse(v.begin(),v.end());
random_shuffle(v.begin(),v.end());
```

К элементу, на который указывает итератор, можно обратиться, воспользовавшись оператором *. В следующем коде печатается первый элемент вектора:

```
cout << *v.begin() << "\n";
```

Более полезный пример: функция lower_bound возвращает итератор на первый элемент отсортированного диапазона, значение которого *не меньше x*, а функция upper_bound – итератор на первый элемент, значение которого *не больше x*:

```
vector<int> v = {2,3,3,5,7,8,8,8};
auto a = lower_bound(v.begin(),v.end(),5);
auto b = upper_bound(v.begin(),v.end(),5);
cout << *a << " " << *b << "\n"; // 5 7
```

Отметим, что эти функции правильно работают, только если заданный диапазон отсортирован. В них применяется двоичный поиск, так что для поиска запрошенного элемента требуется логарифмическое время. Если искомый элемент не найден, то функция возвращает итератор на позицию, следующую за последним элементом диапазона.

В стандартной библиотеке C++ много полезных функций, заслуживающих внимания. Например, в следующем фрагменте создается вектор, содержащий уникальные элементы исходного вектора в отсортированном порядке:

```
sort(v.begin(),v.end());
v.erase(unique(v.begin(),v.end()),v.end());
```

5.1.3. *Другие структуры данных*

Двусторонней очередью (деком) называется динамический массив, допускающий эффективные операции с обеих сторон. Как и вектор, двусторонняя очередь предоставляет функции `push_back` и `pop_back`, но вдобавок к ним функции `push_front` и `pop_front`. Используется она следующим образом:

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

Операции двусторонней очереди в среднем имеют сложность $O(1)$. Однако постоянные множители для них больше, чем для вектора, поэтому использовать двусторонние очереди имеет смысл, только когда требуется выполнять какие-то действия на обеих сторонах структуры.

C++ предоставляет также специализированные структуры данных, по умолчанию основанные на двусторонней очереди. Для *стека* определены функции `push` и `pop`, позволяющие вставлять и удалять элементы в конце структуры, а также функция `top`, возвращающая последний элемент без удаления:

```
stack<int> s;
s.push(2); // [2]
s.push(5); // [2,5]
cout << s.top() << "\n"; // 5
s.pop(); // [2]
cout << s.top() << "\n"; // 2
```

В случае *очереди* элементы вставляются в начало, а удаляются из конца. Для доступа к первому и последнему элементам служат функции `front` и `back`.

```
queue<int> q;
q.push(2); // [2]
```

```
q.push(5); // [2,5]
cout << q.front() << "\n"; // 2
q.pop(); // [5]
cout << q.back() << "\n"; // 5
```

5.2. Множества

Множеством называется структура данных, в которой хранится набор элементов. Основные операции над множествами – вставка, поиск и удаление. Множества реализованы так, что все эти операции эффективны, что часто позволяет улучшить время работы алгоритмов, в которых множества используются.

5.2.1. *Множества и мультимножества*

В стандартной библиотеке C++ имеются две структуры, относящиеся к множествам:

- `set` основана на сбалансированном двоичном дереве поиска, его операции работают за время $O(\log n)$;
- `unordered_set` основана на хеш-таблице и работает в среднем[1] $O(1)$.

Обе структуры эффективны, и во многих случаях годится любая. Поскольку используются они одинаково, в примерах мы ограничимся только структурой `set`.

В показанном ниже коде создается множество, содержащее целые числа, и демонстрируются некоторые его операции. Функция `insert` добавляет элемент во множество, функция `count` возвращает количество вхождений элемента во множество, а функция `erase` удаляет элемент из множества.

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

Важным свойством множеств является тот факт, что все их элементы *различны*. Следовательно, функция `count` всегда возвращает 0 (если элемент не принадлежит множеству) или 1 (если принадлежит), а функция

[1] В худшем случае временная сложность операций составляет $O(n)$, но это крайне маловероятно.

insert никогда не добавляет элемент во множество, если он в нем уже присутствует. Это демонстрируется в следующем фрагменте:

```
set<int> s;
s.insert(3);
s.insert(3);
s.insert(3);
cout << s.count(3) << "\n"; // 1
```

Множество в основном можно использовать как вектор, однако доступ к элементам с помощью оператора [] невозможен. В следующем коде печатается количество элементов во множестве, а затем эти элементы перебираются:

```
cout << s.size() << "\n";
for (auto x : s) {
    cout << x << "\n";
}
```

Функция find(*x*) возвращает итератор, указывающий на элемент со значением *x*. Если же множество не содержит *x*, то возвращается итератор end().

```
auto it = s.find(x);
if (it == s.end()) {
    // x не найден
}
```

Упорядоченные множества. Основное различие между двумя структурами множества в C++ – то, что set *упорядочено*, а unordered_set не упорядочено. Поэтому если порядок элементов важен, то следует пользоваться структурой set.

Рассмотрим задачу о нахождении наименьшего и наибольшего значений во множестве. Чтобы сделать это эффективно, необходимо использовать структуру set. Поскольку элементы отсортированы, найти наименьшее и наибольшее значения можно следующим образом:

```
auto first = s.begin();
auto last = s.end(); last--;
cout << *first << " " << *last << "\n";
```

Отметим, что поскольку end() указывает на позицию, следующую за последним элементом, то необходимо уменьшить итератор на единицу.

В структуре set имеются также функции lower_bound(*x*) и upper_bound(*x*), которые возвращают итератор на наименьший элемент множества, зна-

чение которого *не меньше x* или *больше x* соответственно. Если искомого элемента не существует, то обе функции возвращают `end()`.

```
cout << *s.lower_bound(x) << "\n";
cout << *s.upper_bound(x) << "\n";
```

Мультимножества. В отличие от множества, в *мультимножестве* один и тот же элемент может входить несколько раз. В C++ имеются структуры `multiset` и `unordered_multiset`, похожие на `set` и `unordered_set`. В следующем коде в мультимножество три раза добавляется значение 5.

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

Функция `erase` удаляет все копии значения из мультимножества.

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Если требуется удалить только одно значение, то можно поступить так:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

Отметим, что во временной сложности функций `count` и `erase` имеется дополнительный множитель $O(k)$, где k – количество подсчитываемых (удаляемых) элементов. В частности, подсчитывать количество копий значения в мультимножестве с помощью функции `count` *неэффективно*.

5.2.2. *Отображения*

Отображением называется множество, состоящее из пар ключ-значение. Отображение можно также рассматривать как обобщение массива. Если в обыкновенном массиве ключами служат последовательные целые числа 0, 1, ..., $n - 1$, где n – размер массива, то в отображении ключи могут иметь любой тип и необязательно должны быть последовательными.

В стандартной библиотеке C++ есть две структуры отображений, соответствующие структурам множеств: в основе `map` лежит сбалансированное двоичное дерево со временем доступа к элементам $O(\log n)$, а в основе `unordered_map` – техника хеширования со средним временем доступа к элементам $O(1)$.

В следующем фрагменте создается отображение, ключами которого являются строки, а значениями – целые числа:

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

Если в отображении нет запрошенного ключа, то он автоматически добавляется, и ему сопоставляется значение по умолчанию. Например, в следующем коде в отображение добавляется ключ «aybabtu» со значением 0.

```
map<string,int> m;
cout << m["aybabtu"] << "\n"; // 0
```

Функция `count` проверяет, существует ли ключ в отображении:

```
if (m.count("aybabtu")) {
    // ключ существует
}
```

В следующем коде печатаются все имеющиеся в отображении ключи и значения:

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

5.2.3. *Очереди с приоритетом*

Очередь с приоритетом – это мультимножество, которое поддерживает вставку, а также извлечение и удаление минимального или максимального элемента (в зависимости от типа очереди). Вставка и удаление занимают время $O(\log n)$, а извлечение – время $O(1)$.

Очередь с приоритетом обычно основана на структуре пирамиды (heap), представляющей собой двоичное дерево специального вида. Структура `multiset` и так предоставляет все операции, которые определены в очереди с приоритетом, и даже больше, но у очереди с приоритетом есть достоинство – меньшие постоянные множители в оценке временной сложности. Поэтому если требуется только найти минимальный или максимальный элемент, то лучше использовать очередь с приоритетом, а не множество или мультимножество.

По умолчанию элементы очереди с приоритетом в C++ отсортированы в порядке убывания, так что поддерживаются поиск и удаление наибольшего элемента, что и продемонстрировано в следующем коде:

```
priority_queue<int> q;
q.push(3);
```

```
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Для создания очереди с приоритетом, поддерживающей поиск и удаление наименьшего элемента, нужно поступить так:

```
priority_queue<int,vector<int>,greater<int>> q;
```

5.2.4. *Множества, основанные на политиках*

Компилятор g++ предоставляет также несколько структур данных, не входящих в стандартную библиотеку C++. Они называются *структурами, основанными на политиках* (policy-based structure). Для их использования в программу нужно включить такие строки:

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

После этого можно определить структуру данных `indexed_set`, которая похожа на множество, но допускает индексирование как массив. Для значений типа `int` определение выглядит так:

```
typedef tree<int,null_type,less<int>, rb_tree_tag,
             tree_order_statistics_node_update> indexed_set;
```

А создается множество так:

```
indexed_set s;
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
```

Особенность этого множества состоит в том, что доступ можно осуществлять по индексу, который элемент имел бы в отсортированном массиве. Функция `find_by_order` возвращает итератор, указывающий на элемент в заданной позиции:

```
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7
```

А функция `order_of_key` возвращает позицию заданного элемента:

```
cout << s.order_of_key(7) << "\n"; // 2
```

Если элемент отсутствует во множестве, то мы получим позицию, в которой он находился бы, если бы присутствовал:

```
cout << s.order_of_key(6) << "\n"; // 2
cout << s.order_of_key(8) << "\n"; // 3
```

Время работы обеих функций логарифмическое.

5.3. Эксперименты

В этом разделе мы приведем некоторые результаты, касающиеся *практической* эффективности описанных выше структур данных. Хотя временная сложность – отличный инструмент, она не всегда сообщает всю правду об эффективности, поэтому имеет смысл провести эксперименты с настоящими реализациями и наборами данных.

5.3.1. *Сравнение множества и сортировки*

Многие задачи можно решить, применяя как множества, так и сортировку. Важно понимать, что алгоритмы на основе сортировки обычно гораздо быстрее, даже если это не очевидно из одного лишь анализа временной сложности.

В качестве примера рассмотрим задачу о вычислении количества уникальных элементов вектора. Одно из возможных решений – поместить все элементы во множество и вернуть размер этого множества. Поскольку порядок элементов не важен, можно использовать как `set`, так и `unordered_set`. Можно решить задачу и по-другому: сначала отсортировать вектор, а затем обойти его элементы. Подсчитать количество уникальных элементов отсортированного вектора просто.

В табл. 5.1 приведены результаты эксперимента, в котором оба алгоритма тестировались на случайных векторах чисел типа `int`. Оказалось, что алгоритм на основе `unordered_set` примерно в два раза быстрее алгоритма на основе `set`, а алгоритм на основе сортировки быстрее алгоритма на основе `set` более чем в 10 раз. Отметим, что временная сложность обоих алгоритмов равна $O(n \log n)$, и тем не менее алгоритм на основе сортировки работает гораздо быстрее. Причина в том, что сортировка – простая операция, тогда как сбалансированное двоичное дерево поиска, применяемое в реализации `set`, – сложная структура данных.

Таблица 5.1. Результаты эксперимента по вычислению количества уникальных элементов вектора. Первые два алгоритма вставляют элементы во множество, а последний сортирует вектор, а затем просматривает соседние элементы

Размер входных данных	`set` (с)	`unordered_set` (с)	Сортировка (с)
10^6	0.65	0.34	0.11
$2 \cdot 10^6$	1.50	0.76	0.18
$4 \cdot 10^6$	3.38	1.63	0.33
$8 \cdot 10^6$	7.57	3.45	0.68
$16 \cdot 10^6$	17.35	7.18	1.38

5.3.2. *Сравнение отображения и массива*

Отображения – удобные структуры данных, по сравнению с массивами, поскольку позволяют использовать индексы любого типа, но и постоянные множители велики. В следующем эксперименте мы создали вектор, содержащий n случайных целых чисел от 1 до 10^6, а затем искали самое часто встречающееся значение путем подсчета числа вхождений каждого элемента. Сначала мы использовали отображения, но поскольку число 10^6 достаточно мало, то можно использовать и массивы.

Результаты эксперимента сведены в табл. 5.2. Хотя `unordered_map` примерно в три раза быстрее `map`, массив все равно почти в 100 раз быстрее. Таким образом, по возможности следует пользоваться массивами, а не отображениями. Особо отметим, что хотя временная сложность операций `unordered_map` равна $O(1)$, скрытые постоянные множители, характерные для этой структуры данных, довольно велики.

Таблица 5.2. Результаты эксперимента по вычислению самого часто встречающегося элемента вектора. В первых двух алгоритмах используются отображения, а в последнем – обыкновенный массив

Размер входных данных	`map` (с)	`unordered_map` (с)	Массив (с)
10^6	0.55	0.23	0.01
$2 \cdot 10^6$	1.14	0.39	0.02
$4 \cdot 10^6$	2.34	0.73	0.03
$8 \cdot 10^6$	4.68	1.46	0.06
$16 \cdot 10^6$	9.57	2.83	0.11

5.3.3. *Сравнение очереди с приоритетом и мультимножества*

Верно ли, что очереди с приоритетом действительно быстрее мультимножеств? Чтобы выяснить это, мы провели еще один эксперимент. Мы

создали два вектора, содержащие *n* случайных чисел типа `int`. Сначала мы добавили все элементы первого вектора в структуру данных, а затем обошли второй вектор и на каждом шаге удаляли наименьший элемент из структуры данных и добавляли в нее новый элемент.

Результаты эксперимента представлены в табл. 5.3. Оказалось, что с помощью очереди с приоритетом эта задача решается примерно в пять раз быстрее, чем с помощью мультимножества.

Таблица 5.3. Результаты эксперимента по добавлению и удалению элементов в мультимножество и в очередь с приоритетом

Размер входных данных	`multiset` (с)	`priority_queue` (с)
10^6	1.17	0.19
$2 \cdot 10^6$	2.77	0.41
$4 \cdot 10^6$	6.10	1.05
$8 \cdot 10^6$	13.96	2.52
$16 \cdot 10^6$	30.93	5.95

Глава 6

Динамическое программирование

Динамическое программирование – это техника проектирования алгоритмов, которую можно использовать для нахождения оптимальных решений задач и подсчета числа таких решений. Эта глава является введением в динамическое программирование, а его применения не раз встретятся далее на страницах этой книги.

В разделе 6.1 обсуждаются основные элементы динамического программирования в контексте задачи о размене монет. Дан набор монет разных номиналов, и требуется составить заданную денежную сумму, используя минимальное число монет. Существует простой жадный алгоритм решения этой задачи, но, как мы увидим, он не всегда дает оптимальное решение. А с помощью динамического программирования мы сможем построить эффективный алгоритм, который гарантированно находит оптимальное решение.

В разделе 6.2 рассмотрены избранные задачи, демонстрирующие некоторые возможности динамического программирования: нахождение наибольшей возрастающей подпоследовательности в массиве, нахождение оптимального пути на двумерной сетке и порождение всех возможных сумм весов в задаче о рюкзаке.

6.1. Основные понятия

В этом разделе мы рассмотрим основные понятия динамического программирования в контексте задачи о размене монет. Сначала мы представим жадный алгоритм, который не всегда находит оптимальное решение, а затем покажем, как можно эффективно решить задачу, применив динамическое программирование.

6.1.1. *Когда жадный алгоритм не работает*

Пусть имеется множество номиналов монет `coins` = $\{c_1, c_2, \ldots, c_k\}$ и денежная сумма n. Задача заключается в том, чтобы разменять сумму n, ис-

пользуя как можно меньше монет. Количество монет одного номинала не ограничено. Например, если coins = {1, 2, 5} и n = 12, то оптимальное решение 5 + 5 + 2 = 12, так что достаточно трех монет.

Существует естественный жадный алгоритм решения задачи: всегда выбирать монету максимально возможного номинала, так чтобы общая сумма не превысила n. Например, если n = 12, то сначала выбираем две монеты номинала 5, а затем одну монету номинала 2. Стратегия кажется разумной, но всегда ли она оптимальна?

Оказывается, что не всегда. Например, если coins = {1, 3, 4} и n = 6, то оптимальное решение состоит из двух монет (3 + 3 = 6), тогда как жадный алгоритм дает решение с тремя монетами (4 + 1 + 1 = 6). Этот простой контрпример показывает, что жадный алгоритм не корректен[1].

Тогда как же решить задачу? Можно было бы, конечно, попытаться отыскать другой жадный алгоритм, только вот никаких очевидных стратегий не просматривается. Альтернатива – применить алгоритм полного перебора всех возможных способов размена. Такой алгоритм точно даст правильные результаты, но при больших входных данных будет работать очень медленно.

Однако, воспользовавшись динамическим программированием, мы можем создать алгоритм, который близок к полному перебору, но при этом эффективен. Следовательно, можно применять его к обработке больших входных данных, сохраняя уверенность в правильности результата. Ко всему прочему, ту же технику можно применять к решению многих других задач.

6.1.2. *Нахождение оптимального решения*

Чтобы воспользоваться динамическим программированием, мы должны сформулировать задачу рекурсивно, так чтобы ее решение можно было получить, зная решения меньших подзадач. В задаче о размене монет естественная рекурсивная постановка заключается в вычислении значений следующей функции solve(x): каково минимальное число монет, сумма номиналов которых равна x? Очевидно, значение функции зависит от номиналов монет. Например, ниже приведены значения функции для небольших x в случае, когда coins = {1, 3, 4}:

```
solve(0) = 0
solve(1) = 1
solve(2) = 2
solve(3) = 1
solve(4) = 1
solve(5) = 2
```

[1] Интересно знать, когда жадный алгоритм все-таки работает. В работе Pearson [28] описан эффективный алгоритм ответа на этот вопрос.

$$\begin{aligned}\texttt{solve}(6) &= 2\\ \texttt{solve}(7) &= 2\\ \texttt{solve}(8) &= 2\\ \texttt{solve}(9) &= 3\\ \texttt{solve}(10) &= 3\end{aligned}$$

Например, $\texttt{solve}(10) = 3$, потому что для размена суммы 10 нужно, по крайней мере, 3 монеты. Оптимальное решение $3 + 3 + 4 = 10$.

Важное свойство функции `solve` заключается в том, что ее можно вычислить, зная значения для меньших аргументов. Идея в том, чтобы рассмотреть *первую* монету, выбранную для размена. Так, в примере выше первой монетой может быть 1, 3 или 4. Если первой выбрать монету 1, то останется решить задачу о размене суммы 9 с помощью минимального количества монет: это задача того же вида, что и исходная, только меньше по размеру. Разумеется, то же рассуждение применимо к монетам 3 и 4. Поэтому можно выписать следующую рекурсивную формулу вычисления минимального количества монет:

$$\begin{aligned}\texttt{solve}(x) = \min(&\texttt{solve}(x-1)+1,\\ &\texttt{solve}(x-3)+1,\\ &\texttt{solve}(x-4)+1).\end{aligned}$$

Базой рекурсии является равенство $\texttt{solve}(0) = 0$, поскольку для размена нулевой суммы монеты не нужны. А дальше можно написать, например:

$$\texttt{solve}(10) = \texttt{solve}(7) + 1 = \texttt{solve}(4) + 2 = \texttt{solve}(0) + 3 = 3.$$

Теперь мы готовы представить общую рекурсивную функцию, которая вычисляет минимальное количество монет, необходимых для размена суммы x:

$$\texttt{solve}(x) = \begin{cases}\infty & x < 0\\ 0 & x = 0\\ \min\limits_{c \in \texttt{coins}} \texttt{solve}(x-c)+1 & x > 0\end{cases}$$

Прежде всего если $x < 0$, то значение бесконечно, потому что разменять отрицательную сумму невозможно. Далее, если $x = 0$, то значение равно 0, потому для размена нулевой суммы монеты не нужны. Наконец, если $x > 0$, то переменная c пробегает все возможные варианты выбора первой монеты.

Отыскав рекурсивную функцию, решающую задачу, мы уже можем написать реализацию решения на C++ (константа `INF` обозначает бесконечность):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Однако эта функция неэффективна, потому что сумму можно разменять многими способами, и функция проверяет все. По счастью, это несложно исправить и сделать функцию эффективной.

Запоминание. Ключевая идея динамического программирования – *запоминание*, т. е. мы сохраняем каждое значение функции в массиве сразу после его вычисления. И если впоследствии это значение понадобится снова, то мы можем достать его из массива, не делая рекурсивных вызовов. Для этого создадим два массива:

```
bool ready[N];
int value[N];
```

где `ready`[x] – признак, показывающий, было ли вычислено значение `solve`(x), а `value`[x] – само значение, если оно было вычислено. Константа N выбирается так, чтобы все необходимые значения уместились в массив.

Теперь функцию можно реализовать эффективно:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    ready[x] = true;
    value[x] = best;
    return best;
}
```

Функция сначала обрабатывает рассмотренные выше случаи $x < 0$ и $x = 0$. Затем она проверяет (глядя на `ready`[x]), сохранено ли значение `solve`(x) в элементе `value`[x], и если да, то сразу возвращает его. В противном случае значение `solve`(x) вычисляется рекурсивно и сохраняется в `value`[x].

Эффективность этой функции объясняется тем, что значение для каждого параметра x рекурсивно вычисляется только один раз. А после того как значение solve(x) сохранено в value[x], его можно легко получить, когда функция снова будет вызвана с параметром x. Временная сложность алгоритма равна $O(nk)$, где n – подлежащая размену сумма, а k – количество номиналов монет.

Итеративная реализация. Отметим, что массив value можно также заполнить *итеративно*, воспользовавшись следующим циклом:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

На самом деле участники олимпиад предпочитают именно такую реализацию, поскольку она короче и постоянные множители в ней меньше. Далее мы тоже будем использовать итеративные реализации в примерах. Тем не менее рассуждать о динамическом программировании часто проще в терминах рекурсивных функций.

Построение решения. Иногда нас просят не только найти значение в оптимальном решении, но и привести пример построения самого решения. Чтобы сделать это для задачи о размене монет, объявим новый массив, в котором для каждой размениваемой суммы будем хранить первую монету в оптимальном решении:

```
int first[N];
```

Теперь модифицируем исходный алгоритм следующим образом:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first[x] = c;
        }
    }
}
```

Показанный ниже код печатает монеты, составляющие оптимальное решение для размена суммы n:

```
while (n > 0) {
   cout << first[n] << "\n";
   n -= first[n];
}
```

6.1.3. *Подсчет решений*

Рассмотрим теперь другой вариант задачи о размене монет: требуется найти, сколькими способами можно разменять сумму x монетами заданных номиналов. Например, если coins = {1, 3, 4} и $x = 5$, то всего есть 6 способов:

- 1 + 1 + 1 + 1 + 1;
- 1 + 1 + 3;
- 1 + 3 + 1;
- 3 + 1 + 1;
- 1 + 4;
- 4 + 1.

И эту задачу можно решить рекурсивно. Обозначим solve(x) число способов разменять сумму x. Например, если coins = {1, 3, 4}, то solve(5) = 6, и рекурсивная формула имеет вид:

$$\begin{aligned}\text{solve}(x) = {} & \text{solve}(x-1)+ \\ & \text{solve}(x-3)+ \\ & \text{solve}(x-4).\end{aligned}$$

А в общем виде рекурсивная функция выглядит так:

$$\text{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{coins}} \text{solve}(x-c) & x > 0 \end{cases}$$

Если $x < 0$, то значение равно нулю, поскольку решений нет. Если $x = 0$, то значение равно 1, поскольку нулевую сумму можно разменять только одним способом. В остальных случаях вычисляем сумму всех значений вида solve($x - c$), где c – элемент coins.

Следующий код заполняет массив count, в котором count[x] равно значению solve(x) для $0 \leqslant x \leqslant n$:

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
   for (auto c : coins) {
      if (x-c >= 0) {
         count[x] += count[x-c];
      }
```

```
    }
}
```

Часто число решений настолько велико, что вычислять его точно необязательно, а достаточно дать ответ по модулю m, например $m = 10^9 + 7$. Это можно сделать, изменив код, так чтобы все вычисления производились по модулю m. В предыдущем фрагменте достаточно добавить строку

```
count[x] %= m;
```

после строки

```
count[x] += count[x-c];
```

6.2. Другие примеры

Обсудив основные идеи динамического программирования, мы можем рассмотреть ряд задач, которые эффективно решаются этим методом. Как мы увидим, динамическое программирование – универсальная техника, имеющая много применений в проектировании алгоритмов.

6.2.1. *Наибольшая возрастающая подпоследовательность*

Наибольшей возрастающей подпоследовательностью в массиве из n элементов называется самая длинная последовательность элементов массива, простирающаяся слева направо и такая, что каждый следующий элемент больше предыдущего. На рис. 6.1 показана наибольшая возрастающая подпоследовательность в массиве из восьми элементов.

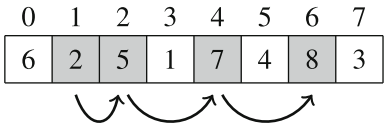


Рис. 6.1. Наибольшая возрастающая подпоследовательность в этом массиве [2, 5, 7, 8]

Для эффективного поиска наибольшей возрастающей подпоследовательности мы воспользуемся динамическим программированием. Обозначим `length`(k) длину наибольшей возрастающей подпоследовательности, оканчивающейся в позиции k. Если мы сумеем вычислить все значения `length`(k) для $0 \leqslant k \leqslant n - 1$, то найдем и длину наибольшей возрастающей подпоследовательности. Значения этой функции для нашего массива приведены ниже:

`length`(0) = 1
`length`(1) = 1

length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2

Например, length(6) = 4, поскольку наибольшая возрастающая подпоследовательность, оканчивающаяся в позиции 6, состоит из 4 элементов.

Чтобы вычислить значение length(k), мы должны найти позицию $i < k$, для которой array[i] < array[k] и length(i) максимально. Тогда length(k) = length(i) + 1, поскольку это оптимальный способ добавить array[k] в подпоследовательность. Но если такой позиции i не существует, то length(k) = 1, т. е. подпоследовательность состоит только из элемента array[k].

Поскольку значение функции всегда можно вычислить, зная ее значения при меньших аргументах, мы можем воспользоваться динамическим программированием. В следующем коде значения функции запоминаются в массиве length.

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k],length[i]+1);
        }
    }
}
```

Понятно, что получившийся алгоритм работает за время $O(n^2)$[2].

6.2.2. *Пути на сетке*

Следующая наша задача – поиск пути из левого верхнего в правый нижний угол сетки $n \times n$ при условии, что разрешено двигаться только вниз и вправо. В каждой клетке находится целое число, и путь должен быть таким, чтобы сумма значений в лежащих на нем клетках была максимальной.

На рис. 6.2 показан оптимальный путь на сетке 5×5. Сумма значений вдоль пути равна 67, и это наибольшая сумма на путях из левого верхнего в правый нижний угол.

[2] С помощью динамического программирования эту задачу можно решить эффективнее – за время $O(n \log n)$. Сможете ли вы найти такое решение?

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Рис. 6.2. Оптимальный путь из левого верхнего в правый нижний угол

Пронумеруем строки и столбцы сетки числами от 1 до n, и пусть `value`[y][x] равно значению в клетке (y, x). Обозначим `sum`(y, x) максимальную сумму на пути из левого верхнего угла в клетку `square`(y, x). Тогда `sum`(n, n) – максимальная сумма на путях из левого верхнего в правый нижний угол. Так, в нашем примере сетки `sum`(5, 5) = 67. Справедлива формула

$$\texttt{sum}(y, x) = \max(\texttt{sum}(y, x - 1),\ \texttt{sum}(y - 1, x)) + \texttt{value}[y][x],$$

основанная на наблюдении, что путь, заканчивающийся в клетке (y, x), может приходить в нее либо из клетки (y, x − 1), либо из клетки (y − 1, x) (рис. 6.3). Поэтому мы выбираем направление, доставляющее максимум сумме. Положим `sum`(y, x) = 0, если y = 0 или x = 0, чтобы рекуррентная формула была справедлива также для клеток, примыкающих к левому и верхнему краю.

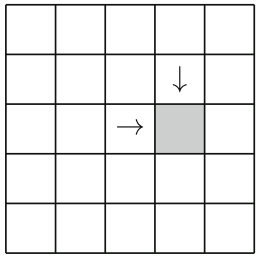


Рис. 6.3. Два возможных способа дойти до клетки

Поскольку у функции два параметра, массив в методе динамического программирования тоже должен быть двумерным, например:

```
int sum[N][N];
```

а суммы вычисляются следующим образом:

```
for (int y = 1; y <= n; y++) {
   for (int x = 1; x <= n; x++) {
      sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
```

```
    }
}
```

Временная сложность этого алгоритм равна $O(n^2)$.

6.2.3. *Задачи о рюкзаке*

Под задачами о *рюкзаке* (или о ранце) понимаются задачи, в которых дано множество предметов и требуется найти подмножества, обладающие некоторыми свойствами. Часто такие задачи можно решить методом динамического программирования.

В этом разделе нас будет интересовать такая задача: пусть дан список весов $[w_1, w_2, \ldots, w_n]$, требуется найти все суммы, которые можно получить сложением весов. На рис. 6.4 показаны возможные суммы для весов [1, 3, 3, 5]. В этом случае возможны все суммы от 0 до 12, за исключением 2 и 10. Например, сумма 7 возможна, потому что образована весами [1, 3, 3].

0	1	2	3	4	5	6	7	8	9	10	11	12
✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓

Рис. 6.4. Образование сумм с использованием весов [1, 3, 3, 5]

Для решения задачи рассмотрим подзадачи, в которых для построения сумм используются только первые k весов. Положим `possible`(x, k) = `true`, если сумму x можно образовать из первых k весов, и `possible`(x, k) = `false` в противном случае. Значения функции можно вычислить рекурсивно по формуле

$$\texttt{possible}(x, k) = \texttt{possible}(x - w_k, k - 1) \text{ или } \texttt{possible}(x, k - 1),$$

основанной на том факте, что вес w_k либо входит в сумму, либо нет. Если вес w_k включается, то остается образовать сумму $x - w_k$, используя только первые $k - 1$ весов, а если не включается, то требуется образовать сумму x, используя первые $k - 1$ весов. Базой рекурсии являются случаи

$$\texttt{possible}(x, k) = \begin{cases} \texttt{true} & x = 0 \\ \texttt{false} & x \neq 0 \end{cases},$$

поскольку если веса вообще не используются, то можно образовать только сумму 0. Наконец, значение `possible`(x, n) сообщает, можно ли образовать сумму x, используя *все* веса.

На рис. 6.5 показаны все значения функции для весов [1, 3, 3, 5] (символом ✓ обозначены значения `true`). Например, глядя на строку $k = 2$, мы понимаем, что суммы [0, 1, 3, 4] можно образовать из весов [1, 3].

	0	1	2	3	4	5	6	7	8	9	10	11	12
$k = 0$	✓												
$k = 1$	✓	✓											
$k = 2$	✓	✓		✓	✓								
$k = 3$	✓	✓		✓	✓		✓	✓					
$k = 4$	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓

Рис. 6.5. Решение задачи о рюкзаке для весов [1, 3, 3, 5] методом динамического программирования

Обозначим m полную сумму весов. Показанной выше рекурсивной функции соответствует следующее решение методом динамического программирования:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
   for (int x = 0; x <= m; x++) {
      if (x-w[k] >= 0) {
         possible[x][k] |= possible[x-w[k]][k-1];
      }
      possible[x][k] |= possible[x][k-1];
   }
}
```

Но есть и более компактный способ реализовать вычисление, применяя всего лишь одномерный массив possible[x], показывающий, можно ли выбрать подмножество весов, дающих в сумме x. Хитрость в том, чтобы для каждого нового веса обновлять массив справа налево:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
   for (int x = m-w[k]; x >= 0; x--) {
      possible[x+w[k]] |= possible[x];
   }
}
```

Отметим, что общую идею динамического программирования, представленную в этом разделе, можно применить и к другим задачам о рюкзаке, например в случае, когда даны веса и ценности предметов и требуется найти подмножество с максимальной ценностью, соблюдая при этом ограничение на суммарный вес.

6.2.4. *От перестановок к подмножествам*

С помощью динамического программирования часто можно заменить итерирование по перестановкам итерированием по подмножествам. Выгода здесь в том, что количество подмножеств 2^n значительно меньше количества перестановок $n!$. Например, при $n = 20$ $n! \approx 2.4 \cdot 10^{18}$, а $2^n \approx 10^6$. Следовательно, для некоторых значений n все подмножества можно обойти эффективно, а все перестановки – нельзя.

В качестве примера рассмотрим следующую задачу: имеется лифт с максимальной грузоподъемностью x и n человек, желающих подняться с первого на последний этаж. Пассажиры пронумерованы от 0 до $n - 1$, вес i-го пассажира равен `weight`[i]. За какое минимальное количество поездок удастся перевезти всех на верхний этаж?

Пусть, например, $x = 12$, $n = 5$ и веса таковы:

- `weight`[0] = 2
- `weight`[1] = 3
- `weight`[2] = 4
- `weight`[3] = 5
- `weight`[4] = 9

В этом случае минимальное число поездок равно 2. Одно из оптимальных решений выглядит так: сначала перевезти пассажиров 0, 2 и 3 (суммарный вес 11), а затем пассажиров 1 и 4 (суммарный вес 12).

Задачу легко решить за время $O(n!n)$, проверив все возможные перестановки n человек. Но, применив динамическое программирование, мы сможем найти более эффективный алгоритм с временной сложностью $O(2^n n)$. Идея в том, чтобы для каждого подмножества пассажиров вычислить два значения: минимальное число необходимых поездок и минимальный вес пассажиров в последней группе.

Обозначим `rides`(S) минимальное число поездок для подмножества S, а `last`(S) – минимальный вес последней группы в решении с минимальным числом поездок. Так, в примере выше

$$\texttt{rides}(\{3, 4\}) = 2 \quad \text{и} \quad \texttt{last}(\{3, 4\}) = 5,$$

поскольку оптимальный способ поднять пассажиров 3 и 4 на последний этаж – везти их по отдельности, включив пассажира 4 в первую группу, тогда будет минимизирован вес второй группы. Понятно, что наша конечная цель – вычислить значение `rides`($\{0 \ldots n - 1\}$).

Мы можем вычислять значения функций рекурсивно, а затем применить динамическое программирование. Чтобы вычислить значения для подмножества S, мы перебираем всех пассажиров, принадлежащих S, и производим оптимальный выбор последнего пассажира p, который входит в лифт. Каждый такой выбор порождает подзадачу с меньшим под-

множеством пассажиров. Если last$(S \setminus p)$ + weight[p] ⩽ x, то мы можем включить p в последнюю группу. В противном случае придется выполнить еще одну поездку специально для p.

Вычисление по методу динамического программирования удобно реализовать с помощью поразрядных операций. Сначала объявим массив

```
pair<int,int> best[1<<N];
```

в котором для каждого подмножества S хранится пара (rides(S), last(S)). Для пустого подмножества поездки не нужны:

```
best[0] = {0,0};
```

Заполнить массив можно следующим образом:

```
for (int s = 1; s < (1<<n); s++) {
    // начальное значение: необходимо n+1 поездок
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
            auto option = best[s^(1<<p)];
            if (option.second+weight[p] <= x) {
                // добавить p в существующую группу пассажиров
                option.second += weight[p];
            } else {
                // предусмотреть для p отдельную поездку
                option.first++;
                option.second = weight[p];
            }
            best[s] = min(best[s], option);
        }
    }
}
```

Отметим, что этот цикл обладает следующим свойством: для любых двух подмножеств S_1 и S_2 – таких, что $S_1 \subset S_2$, S_1, – обрабатывается раньше S_2. Следовательно, используемые в динамическом программировании значения вычисляются в правильном порядке.

6.2.5. *Подсчет количества замощений*

Иногда состояния в решении методом динамического программирования оказываются сложнее, чем фиксированные комбинации значений. В качестве примера рассмотрим задачу о нахождении количества различных способов замостить сетку размера $n \times m$ плитками размера 1×2 и 2×1.

Например, существует 781 способ замостить сетку 4×7, один из них показан на рис. 6.6.

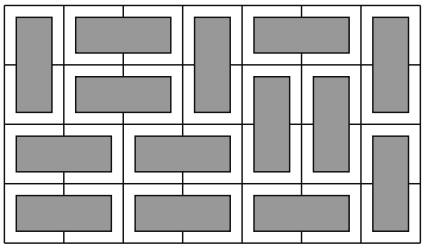


Рис. 6.6. Один из способов заполнить сетку 4×7 плитками 1×2 и 2×1

Задачу можно решить методом динамического программирования, рассматривая сетку ряд за рядом. Каждый ряд в решении можно представить строкой, содержащей m символов из множества {⊓,⊔,⊏,⊐}. Так, решение, показанное на рис. 6.6, состоит из четырех рядов, которым соответствуют такие строки:

- ⊓⊏⊐⊓⊏⊐⊓
- ⊔⊏⊐⊔⊓⊓⊔
- ⊏⊐⊏⊐⊔⊔⊓
- ⊏⊐⊏⊐⊏⊐⊔

Пронумеруем ряды сетки числами от 1 до n. Обозначим count(k, x) число таких решений для рядов 1...k, что ряду k соответствует строка x. Здесь можно воспользоваться динамическим программированием, потому что состояние ряда ограничено только состоянием предыдущего ряда.

Решение допустимо, если ряд 1 не содержит символа ⊔, ряд n не содержит символа ⊓ и все соседние ряды совместимы. Например, ряды ⊔⊏⊐⊔⊓⊓⊔ и ⊏⊐⊏⊐⊔⊔⊓ совместимы, а ряды ⊓⊏⊐⊓⊏⊐⊓ и ⊏⊐⊏⊐⊏⊐⊔ – нет.

Поскольку ряд состоит из m символов и каждый символ можно выбрать четырьмя способами, общее число различных рядов не превышает 4^m. Мы можем перебрать $O(4^m)$ возможных состояний каждого ряда, и для каждого ряда существует $O(4^m)$ возможных состояний предыдущего ряда, поэтому временная сложность решения равна $O(n4^{2m})$. На практике разумно повернуть сетку, так чтобы более короткая сторона имела длину m, поскольку множитель 4^{2m} доминирует в оценке сложности.

Эффективность решения можно повысить, представив ряды в более компактной форме. Оказывается, что достаточно знать, какие колонки предыдущей строки содержат верхний квадратик вертикальной плитки. Поэтому для представления ряда достаточно символов ⊓ и □, где □ – комбинация символов ⊔, ⊏ и ⊐. При таком представлении существует всего 2^m различных строк, так что временная сложность равна $O(n2^{2m})$.

Напоследок отметим, что существует явная формула для количества замощений:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot \left(\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1} \right).$$

Эта формула очень эффективна, поскольку вычисляет количество замощений за время $O(nm)$, но, так как ответ выражается как произведение вещественных чисел, возникает проблема: как точно представлять промежуточные результаты.

Глава 7

Алгоритмы на графах

Многие программистские задачи можно решить, если рассмотреть ситуацию как граф и воспользоваться подходящим алгоритмом. В этой главе мы познакомимся с основами теории графов и некоторыми важными алгоритмами на графах.

В разделе 7.1 обсуждается терминология теории графов и структуры данных, используемая для представления графов.

В разделе 7.2 описаны два фундаментальных алгоритма обхода графа. Поиск в глубину – это простой способ посетить все вершины, достижимые из начальной, а при поиске в ширину вершины посещаются в порядке возрастания расстояния от начальной вершины.

В разделе 7.3 представлены алгоритмы нахождения кратчайших путей во взвешенных графах. Простой алгоритм Беллмана–Форда находит кратчайшие пути от начальной вершины ко всем остальным. Алгоритм Дейкстры более эффективен, но требуется, чтобы веса всех ребер были неотрицательны. Алгоритм Флойда–Уоршелла находит кратчайшие пути между всеми парами вершин.

В разделе 7.4 изучаются специальные свойства ориентированных ациклических графов. Мы узнаем, как выполнить топологическую сортировку и как эффективно обрабатывать такие графы с помощью динамического программирования.

Раздел 7.5 посвящен графам, в которых у каждой вершины имеется единственная вершина-преемник. Мы обсудим эффективный способ нахождения вершин-преемников и алгоритм Флойда нахождения циклов в графе.

В разделе 7.6 изложены алгоритмы Краскала и Прима для построения минимальных остовных деревьев. Алгоритм Краскала основан на эффективной структуре данных – системе непересекающихся множеств, которая имеет и другие применения в проектировании алгоритмов.

7.1. Основы теории графов

В этом разделе мы введем терминологию, которая используется при обсуждении графов и их свойств. А затем займемся структурами данных для представления графов при программировании алгоритмов.

7.1.1. Терминология

Граф состоит из *вершин*, соединенных *ребрами*. Мы будем обозначать буквой n количество вершин графа, а буквой m – количество ребер. Ребра нумеруются числами $1, 2, \ldots, n$. На рис. 7.1 изображен граф с 5 вершинами и 7 ребрами.

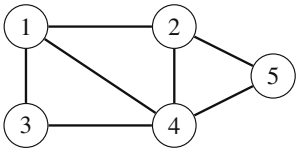


Рис. 7.1. Граф с 5 вершинами и 7 ребрами

Путь ведет из одной вершины в другую и проходит по ребрам графа. *Длиной* пути называется количество ребер в нем. На рис. 7.2 показан путь $1 \to 3 \to 4 \to 5$ длины 3 из вершины 1 в вершину 5. *Циклом* называется путь, в котором последняя вершина совпадает с первой. На рис. 7.3 изображен цикл $1 \to 3 \to 4 \to 1$.

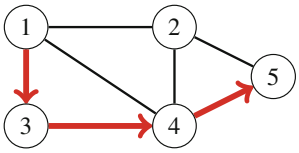


Рис. 7.2. Путь из вершины 1 в вершину 5

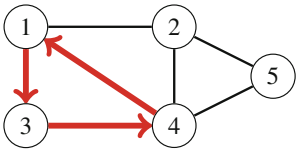


Рис. 7.3. Цикл с тремя вершинами

Граф называется *связным*, если между любыми двумя вершинами существует путь. Левый граф на рис. 7.4 связный, а правый – нет, потому что из вершины 4 нельзя попасть ни в какую другую.

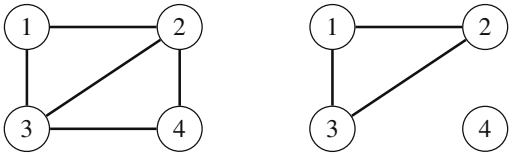


Рис. 7.4. Левый граф связный, правый – нет

Связные части графа называются его *компонентами связности*. Граф на рис. 7.5 состоит из трех компонент связности: {1, 2, 3}, {4, 5, 6, 7} и {8}.

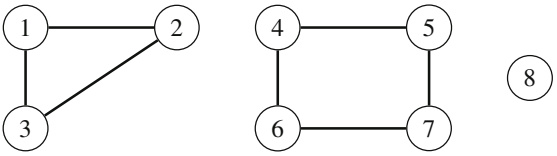


Рис. 7.5. Граф с тремя компонентами связности

Деревом называется связный граф без циклов. На рис. 7.6 приведен пример дерева.

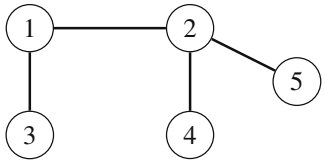


Рис. 7.6. Дерево

В *ориентированном* графе по каждому ребру можно проходить только в одном направлении. На рис. 7.7 показан пример ориентированного графа. В нем имеется путь 3 → 1 → 2 → 5 из вершины 3 в вершину 5, но не существует пути из вершины 5 в вершину 3.

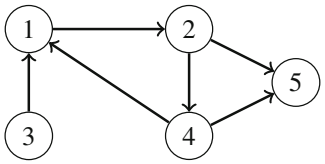


Рис. 7.7. Ориентированный граф

Во *взвешенном* графе каждому ребру сопоставлен *вес*. Часто веса интерпретируются как длины ребер, а длиной пути считается сумма весов составляющих его ребер. На рис. 7.8 изображен взвешенный граф, длина пути 1 → 3 → 4 → 5 равна 1 + 7 + 3 = 11. Это *кратчайший* путь из вершины 1 в вершину 5.

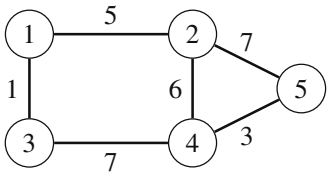


Рис. 7.8. Взвешенный граф

Две вершины называются *соседними*, или *смежными*, если они соединены ребром. *Степенью* вершины называется число соседних с ней вершин. На рис. 7.9 показаны степени всех вершин графа. Так, степень вершины 2 равна 3, потому что ее соседями являются вершины 1, 4 и 5.

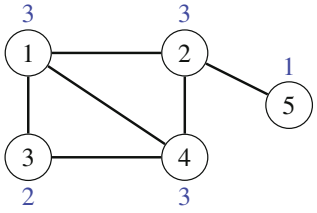


Рис. 7.9. Степени вершин

Сумма степеней всех вершин графа равна $2m$, где m – число ребер, поскольку каждое ребро увеличивает степени ровно двух вершин на единицу. Таким образом, сумма степеней вершин всегда четна. Граф называется *регулярным*, если степени всех его вершин одинаковы (равны некоторой константе d). Граф называется *полным*, если степень каждой его вершины равна $n - 1$, т. е. в графе присутствуют все возможные ребра.

В ориентированном графе *полустепенью захода* вершины называется число ребер, оканчивающихся в этой вершине, а *полустепенью исхода* – число ребер, начинающихся в вершине. На рис. 7.10 показаны полустепени захода и исхода для каждой вершины графа. Например, для вершины 2 полустепень захода равна 2, а полустепень исхода – 1.

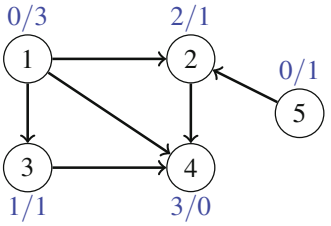


Рис. 7.10. Полустепени захода и исхода

Граф называется *двудольным*, если его вершины можно раскрасить в два цвета, так что цвета любых двух соседних вершин различны. Можно доказать, что граф является двудольным тогда и только тогда, когда в нем нет цикла с нечетным числом вершин. На рис. 7.11 изображены двудольный граф и его раскраска.

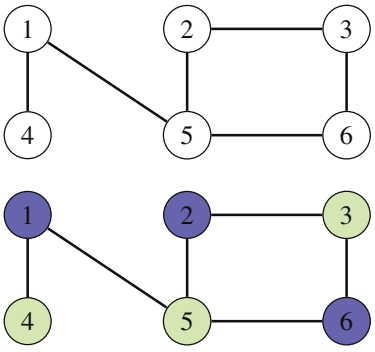


Рис. 7.11. Двудольный граф и его раскраска

7.1.2. *Представление графа*

Есть несколько способов представить граф в алгоритмах. Выбор структуры данных зависит от размера графа и способа его обработки в алгоритме. Ниже рассмотрены три популярных представления.

Списки смежности. В этом случае каждой вершине *x* сопоставляется *список смежности*, включающий вершины, соединенные с *x* ребром. Списки смежности – самый популярный способ представления графов, они позволяют эффективно реализовать большинство алгоритмов.

Списки смежности удобно хранить в массиве векторов, который объявлен следующим образом:

```
vector<int> adj[N];
```

Константа *N* выбрана так, чтобы в массиве поместились все списки смежности. Например, граф на рис. 7.12 (*a*) можно сохранить так:

```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

Неориентированные графы можно хранить аналогично, только каждое ребро нужно учитывать в двух списках смежности (для обоих направлений).

Для взвешенных графов структуру следует дополнить:

```
vector<pair<int,int>> adj[N];
```

В этом случае список смежности вершины *a* содержит пару (*b*,*w*), если существует ребро с весом *w*, направленное от *a* к *b*. Граф на рис. 7.12 (*b*) можно сохранить следующим образом:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

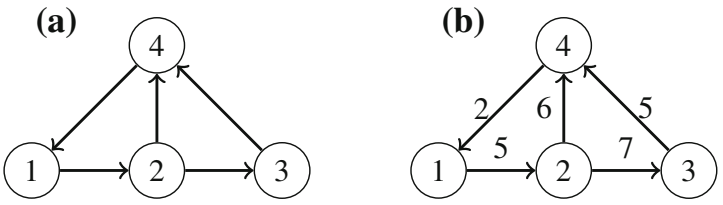


Рис. 7.12. Примеры графов

Списки смежности позволяют эффективно находить вершины, в которые можно перейти из данной по одному ребру. Следующий цикл обходит все вершины, в которые можно попасть из вершины *s*:

```
for (auto u : adj[s]) {
    // обработать вершину u
}
```

Матрица смежности показывает, какие ребра есть в графе. С ее помощью можно эффективно проверить, существует ли ребро между двумя вершинами. Матрицу можно хранить в виде массива

```
int adj[N][N];
```

где элемент adj[*a*][*b*] равен 1, если существует ребро, ведущее из вершины *a* в вершину *b*, а в противном случае равен 0. Так, матрица смежности для графа на рис. 7.12 (*a*) имеет вид:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Если граф взвешенный, то представление в виде матрицы смежности можно обобщить: если две вершины соединены ребром, то в матрице хранится вес этого ребра. Так, граф на рис. 7.12 (*b*) представляется следующей матрицей:

$$\begin{bmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 7 & 6 \\ 0 & 0 & 0 & 5 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

Недостаток матрицы смежности заключается в том, что она содержит n^2 элементов, большая часть которых обычно равна 0. Поэтому такое представление не годится для больших графов.

Список ребер содержит все ребра графа в некотором порядке. Это представление удобно, если алгоритм обрабатывает все ребра и не требуется находить ребра, начинающиеся в заданной вершине.

Список ребер можно хранить в векторе

```
vector<pair<int,int>> edges;
```

где наличие пары (*a*, *b*) означает, что существует ребро из вершины *a* в вершину *b*. Граф на рис. 7.12 (*a*) можно представить следующим образом:

```
edges.push_back({1,2});
edges.push_back({2,3});
```

```
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

Для взвешенных графов структуру можно обобщить:

```
vector<tuple<int,int,int>> edges;
```

Каждый элемент этого списка имеет вид (*a*,*b*,*w*), это означает, что существует ребро с весом *w*, ведущее из вершины *a* в вершину *b*. Например, граф на рис. 7.12 (*b*) можно представить следующим образом[1]:

```
edges.push_back({1,2,5});
edges.push_back({2,3,7});
edges.push_back({2,4,6});
edges.push_back({3,4,5});
edges.push_back({4,1,2});
```

7.2. Обход графа

В этом разделе обсуждается два фундаментальных алгоритма на графах: поиск в глубину и поиск в ширину. В обоих случаях задается начальная вершина и ставится задача посетить все достижимые из нее вершины. Различие заключается в порядке посещения вершин.

7.2.1. *Поиск в глубину*

Поиск в глубину (depth-first search – DFS) – прямолинейный способ обхода графа. Алгоритм начинает работу в начальной вершине и перебирает все вершины, достижимые из нее по ребрам графа.

Поиск в глубину всегда следует по одному пути, пока на нем еще имеются вершины. После этого он возвращается назад и начинает исследовать другие части графа. Алгоритм запоминает посещенные вершины, так что каждая обрабатывается только один раз.

На рис. 7.13 показан порядок обработки вершин при поиске в глубину. Поиск может начинаться с любой вершины: в данном случае мы начали с вершины 1. Сначала исследуется путь $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$, затем алгоритм возвращается к вершине 1 и посещает оставшуюся вершину 4.

Реализация. Поиск в глубину удобно реализовать рекурсивно. Показанная ниже функция `dfs` начинает поиск с заданной вершины. Предполагается, что граф представлен списками смежности, хранящимися в массиве

```
vector<int> adj[N];
```

[1] В некоторых старых компиляторах вместо фигурных скобок следует использовать функцию `make_tuple` (например, `make_tuple(1,2,5)`, а не `{1,2,5}`).

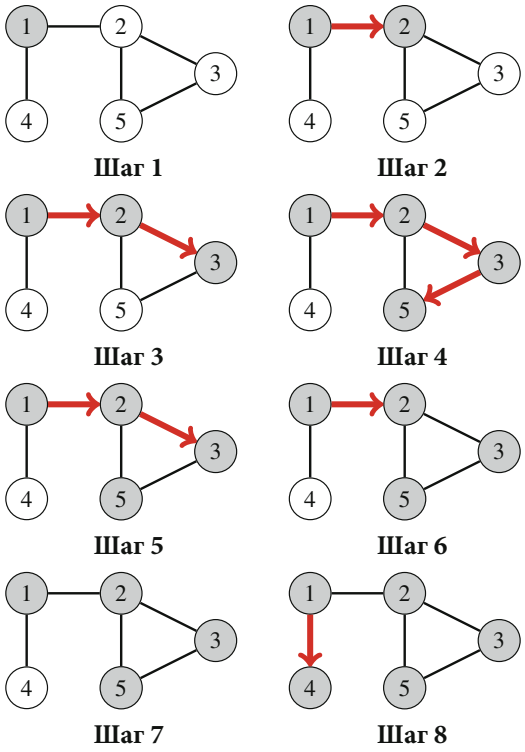


Рис. 7.13. Поиск в глубину

Кроме того, используется массив

```
bool visited[N];
```

в котором запоминаются посещенные вершины. В начальный момент все элементы этого массива равны `false`, но когда алгоритм заходит в вершину *s*, в элемент `visited`[*s*] записывается `true`. Функцию можно реализовать следующим образом:

```
void dfs(int s) {
    if (visited[s]) return;
        visited[s] = true;
        // обработать вершину s
        for (auto u: adj[s]) {
            dfs(u);
    }
}
```

Временная сложность поиска в глубину равна $O(n + m)$, где n – число вершин, а m – число ребер, поскольку алгоритм обрабатывает каждую вершину и каждое ребро ровно один раз.

7.2.2. *Поиск в ширину*

При *поиске в ширину* (breadth-first search – BFS) вершины графа посещаются в порядке возрастания расстояния от начальной вершины. Следовательно, используя поиск в ширину, мы сможем вычислить расстояния от начальной вершины до всех остальных. Однако реализовать поиск в ширину труднее, чем поиск в глубину.

В процессе поиска в ширину мы обходим вершины уровень за уровнем. Сначала посещаются вершины, отстоящие от начальной на расстояние 1, затем – на расстояние 2 и т. д. Процесс продолжается, пока не останется непосещенных вершин.

На рис. 7.14 показано, как происходит обход графа при поиске в ширину. Предположим, что поиск начинается с вершины 1. Сначала мы посещаем вершины 2 и 4, удаленные на расстояние 1, затем – вершины 3 и 5, удаленные на расстояние 2, и, наконец, вершину 6, удаленную на расстояние 3.

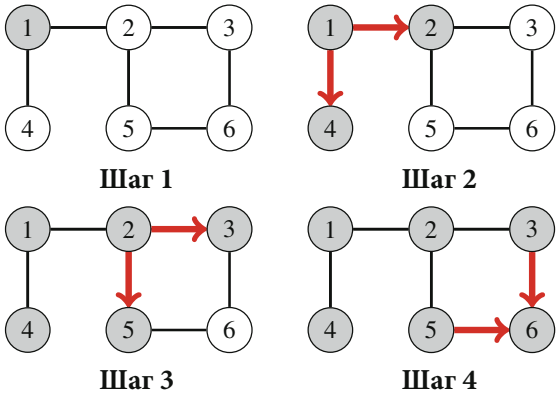


Рис. 7.14. Поиск в ширину

Реализация. Поиск в ширину труднее реализовать, чем поиск в глубину, потому что алгоритм посещает вершины, находящиеся в разных частях графа. Типичная реализация основана на очереди вершин. На каждом шаге обрабатывается следующий узел из очереди.

В приведенном ниже коде предполагается, что граф представлен списками смежности и что определены следующие структуры данных:

```
queue<int> q;
bool visited[N];
int distance[N];
```

Очередь `q` содержит вершины, подлежащие обработке в порядке возрастания расстояния. Новые вершины добавляются в конец очереди, а обрабатывается вершина, находящаяся в начале очереди. В массиве `visited` хранится информация о том, какие вершины уже посещались, а в массиве

distance – расстояния от начальной вершины до всех остальных вершин графа.

Поиск, начинающийся в вершине *x*, реализуется следующим образом:

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
    int s = q.front(); q.pop();
    // обработать вершину s
    for (auto u : adj[s]) {
        if (visited[u]) continue;
            visited[u] = true;
            distance[u] = distance[s]+1;
            q.push(u);
    }
}
```

Временная сложность поиска в ширину, как и поиска в глубину, равна $O(n+m)$, где n – число вершин, а m – число ребер.

7.2.3. *Применения*

С помощью алгоритмов обхода мы можем проверить многие свойства графа. Обычно применимы как поиск в глубину, так и поиск в ширину, но на практике поиск в глубину предпочтительнее, потому что он проще реализуется. В описываемых ниже применениях предполагается, что граф неориентированный.

Проверка связности. Граф называется связным, если между любыми двумя вершинами существует путь. Следовательно, для проверки связности мы можем начать с произвольной вершины и выяснить, все ли вершины достижимы из нее.

На рис. 7.15 видно, что поиск в глубину, начатый из вершины 1, не посещает все вершины, поэтому можно заключить, что граф не связный. Можно также найти все компоненты графа: для этого нужно перебрать все вершины и начинать новый поиск в глубину, если текущая вершина не принадлежит ни одной из уже найденных компонент связности.

Обнаружение циклов. Граф содержит цикл, если в процессе обхода мы встречаем такую вершину, что одна из соседних с ней (кроме той, что предшествует ей на текущем пути) уже посещалась. На рис. 7.16 поиск в глубину из вершины 1 обнаруживает, что в графе имеется цикл. При переходе из вершины 2 в вершину 5 мы видим, что соседняя с 5 вершина 3 уже посещалась. Следовательно, граф содержит цикл, проходящий через вершину 3, например $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

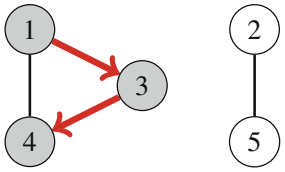


Рис. 7.15. Проверка связности графа

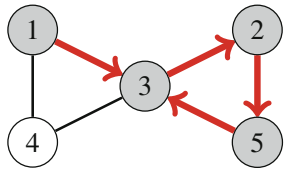


Рис. 7.16. Нахождение цикла в графе

Есть и другой способ узнать, содержит ли граф цикл: просто посчитать количество вершин и ребер в каждой компоненте. Если компонента содержит c вершин и ни одного цикла, то в ней должно быть ровно $c - 1$ ребер (т. е. она должна быть деревом). Если число ребер равно c или больше, то компонента обязательно содержит цикл.

Проверка на двудольность. Граф называется двудольным, если его вершины можно раскрасить двумя цветами, так что никакие две соседние вершины не будут окрашены одним цветом. Удивительно, как просто можно проверить граф на двудольность с помощью алгоритмов обхода.

Идея в том, чтобы выбрать два цвета X и Y, окрасить начальную вершину цветом X, всех ее соседей – цветом Y, всех их соседей – цветом X и т. д. Если в какой-то момент мы обнаруживаем, что две соседние вершины окрашены одним цветом, значит, граф не является двудольным. В противном случае граф двудольный, и мы нашли доказывающую это раскраску.

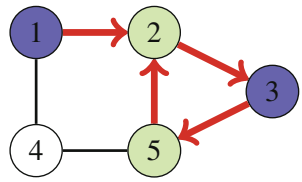


Рис. 7.17. Конфликт при проверке на двудольность

Поиск в глубину из вершины 1 показывает, что граф на рис. 7.17 не является двудольным, т. к. вершины 2 и 5 окрашены одним цветом, хотя и являются соседними.

Этот алгоритм корректен, потому что при наличии всего двух цветов цвет начальной вершины однозначно определяет цвета всех остальных вершин, принадлежащих той же компоненте связности.

Отметим, что в общем случае трудно определить, можно ли раскрасить вершины графа в k цветов, так чтобы никакие две соседние вершины не были окрашены одним цветом. Для $k = 3$ эта задача является NP-трудной.

7.3. Кратчайшие пути

Нахождение кратчайшего пути между двумя вершинами графа – важная задача, имеющая много практических применений. Очевидный пример – нахождение кратчайшего маршрута между двумя городами, если известна карта дорог и длины всех участков.

В невзвешенном графе длина пути равна количеству ребер в нем, и для поиска кратчайшего пути достаточно применить поиск в ширину. Но

в этом разделе нас будут интересовать взвешенные графы, для которых нужны более сложные алгоритмы.

7.3.1. *Алгоритм Беллмана–Форда*

Алгоритм Беллмана–Форда находит кратчайшие пути из начальной вершины во все вершины графа. Алгоритм применим к любым графам, не содержащим цикла с отрицательной длиной. Если граф содержит такой цикл, то алгоритм обнаружит это.

Алгоритм запоминает расстояния от начальной вершины до всех вершин графа. В начальный момент расстояние до начальной вершины равно 0, а до всех остальных бесконечно. Затем алгоритм уменьшает расстояния, отыскивая ребра, которые укорачивают пути, и останавливается, когда ни одно расстояние нельзя уменьшить.

На рис. 7.18 показано, как алгоритм Беллмана–Форда обрабатывает граф. Сначала алгоритм уменьшает расстояния, используя ребра 1 → 2, 1 → 3 и 1 → 4, затем – используя ребра 2 → 5 и 3 → 4, и, наконец, используя ребро 4 → 5. После этого не осталось ребер, с помощью которых можно уменьшить расстояния, и, значит, найденные расстояния окончательны.

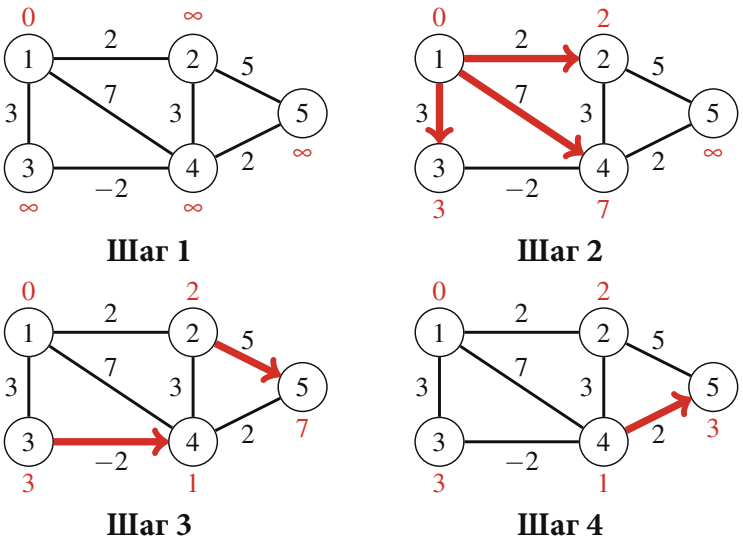


Рис. 7.18. Алгоритм Беллмана–Форда

Реализация. Приведенная ниже реализация алгоритма Беллмана–Форда определяет кратчайшие расстояния от вершины *x* до всех вершин графа. Предполагается, что граф представлен списком ребер, содержащим кортежи вида (*a*,*b*,*w*); каждый такой кортеж означает, что существует ребро веса *w*, соединяющее вершины *a* и *b*.

Алгоритм состоит из $n - 1$ раундов. На каждом раунде алгоритм перебирает все ребра графа и пытается уменьшить расстояния. Строится массив distance, в котором хранятся расстояния от вершины x до каждой вершины графа. Константа INF обозначает бесконечное расстояние.

```
for (int i = 1; i <= n; i++) {
    distance[i] = INF;
}
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

Временная сложность этого алгоритма равна $O(nm)$, поскольку на каждом из $n - 1$ раундов перебираются все m ребер. Если в графе нет отрицательных циклов, то после $n - 1$ раундов расстояния уже не могут измениться, потому что любой кратчайший путь содержит не более $n - 1$ ребер.

На практике алгоритм можно оптимизировать несколькими способами. Во-первых, окончательные расстояния обычно становятся известны еще до выполнения всех $n - 1$ раундов, так что можно просто остановить алгоритм, если в очередном раунде ни одно расстояние не удалось уменьшить. Существует также *усовершенствованный алгоритм SPFA* (Shortest Path Faster Algorithm – ускоренный алгоритм поиска кратчайшего пути [10]), который поддерживает очередь вершин, потенциально способных уменьшить расстояния. Обрабатывать нужно лишь вершины из этой очереди, что часто повышает эффективность поиска.

Отрицательные циклы. Алгоритм Беллмана–Форда можно использовать также для проверки наличия циклов с отрицательной длиной. В этом случае длину любого пути, содержащего такой цикл, можно уменьшать бесконечно много раз, поэтому понятие кратчайшего пути теряет смысл. Например, граф на рис. 7.19 содержит отрицательный цикл $2 \to 3 \to 4 \to 2$ длины -4.

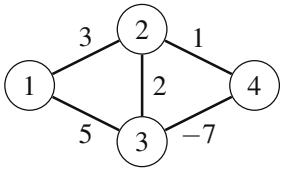


Рис. 7.19. Граф с отрицательным циклом

Для обнаружения отрицательного цикла с помощью алгоритма Беллмана–Форда нужно выполнить n раундов алгоритма. Если в последнем раунде хотя бы одно расстояние уменьшилось, то граф содержит отрицательный цикл. Отметим, что этот алгоритм обнаруживает отрицательный цикл вне зависимости от выбора начальной вершины.

7.3.2. *Алгоритм Дейкстры*

Алгоритм Дейкстры, как и алгоритм Беллмана–Форда, находит кратчайшие пути из начальной вершины во все вершины графа. Но он более эффективен и может применяться для обработки графов большого размера. Однако требуется, чтобы в графе не было ребер с отрицательным весом.

Как и алгоритм Беллмана–Форда, алгоритм Дейкстры хранит расстояния до вершин и уменьшает их в процессе поиска. На каждом шаге алгоритм Дейкстры выбирает из еще не обработанных вершин ту, до которой расстояние минимально. Затем перебираются все ребра, начинающиеся в этой вершине, и с их помощью уменьшаются расстояния. Алгоритм Дейкстры эффективен, потому что каждое ребро обрабатывается только один раз – так как в графе нет ребер с отрицательным весом.

На рис. 7.20 показано, как алгоритм Дейкстры обрабатывает граф. Как и в алгоритме Беллмана–Форда, начальные расстояния до всех вершин, кроме стартовой, равны бесконечности. Алгоритм обрабатывает вершины в порядке 1, 5, 4, 2, 3 и каждый раз уменьшает расстояния с помощью ребер, начинающихся в текущей вершине. Отметим, что после обработки любой вершины расстояние до нее больше не изменяется.

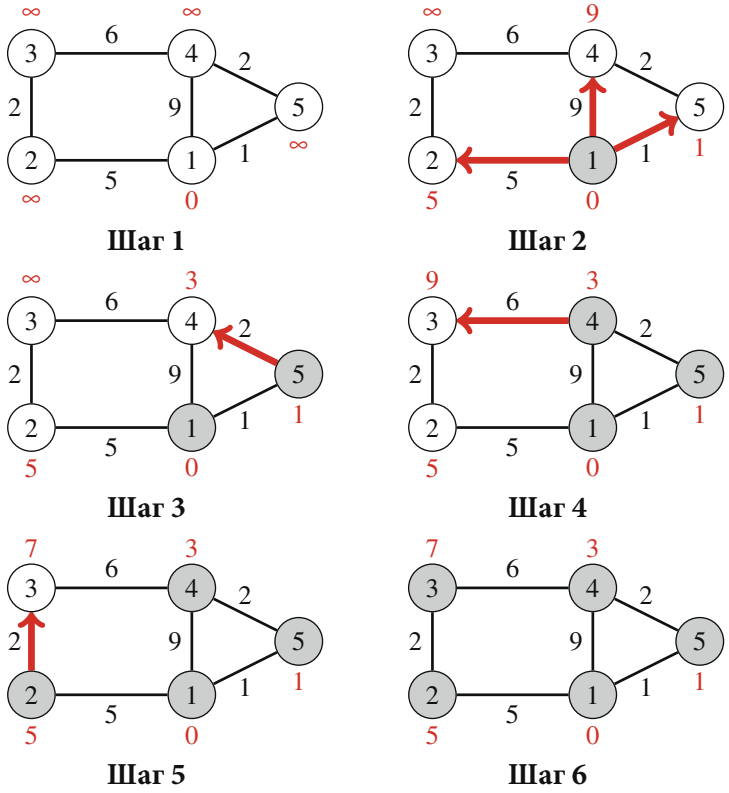


Рис. 7.20. Алгоритм Дейкстры

Реализация. Для эффективной реализации алгоритма Дейкстры необходимо эффективно находить еще не обработанную вершину, отстоящую на минимальное расстояние. Для этого подойдет очередь с приоритетом, в которой оставшиеся вершины хранятся, упорядоченные по возрастанию расстояния. Эта структура данных позволяет найти следующую вершину за логарифмическое время.

В типичной реализации алгоритма Дейкстры из учебника используется очередь с приоритетом, предоставляющая операцию изменения находящегося в очереди значения. Это дает возможность хранить в очереди единственный экземпляр каждой вершины и обновлять расстояние до него по мере необходимости. Но очередь с приоритетом из стандартной библиотеки не предоставляет такой операции, поэтому в олимпиадном программировании обычно используется несколько иная реализация. Идея в том, чтобы добавлять в очередь новый экземпляр вершины всякий раз, как изменяется расстояние до нее.

Наша реализация алгоритма Дейкстры вычисляет минимальные расстояния от вершины x до всех остальных вершин графа. Граф представлен в виде списков смежности, так что `adj`[a] содержит пару (b, w), если существует ребро веса w, соединяющее вершины a и b. Очередь с приоритетом

```
priority_queue<pair<int,int>> q;
```

содержит пары вида ($-d$, x), означающие, что текущее расстояние до вершины x равно d. Массив `distance` содержит расстояния до всех вершин, а массив `processed` позволяет узнать, была ли вершина обработана.

Отметим, что в очереди с приоритетом хранятся расстояния до вершин *со знаком минус*. Дело в том, что по умолчанию реализация очереди в стандартной библиотеке C++ находит максимальный элемент, а нам нужен минимальный. Изменив знак расстояния, мы сможем воспользоваться имеющейся реализацией очереди без каких-либо изменений[2]. Отметим также, что в очереди может находиться несколько экземпляров вершины, но обработан будет только тот экземпляр, в котором расстояние минимально.

Код приведен ниже:

```
for (int i = 1; i <= n; i++) {
    distance[i] = INF;
}
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
```

[2] Конечно, можно было бы объявить очередь, как описано в разделе 5.2.3, и пользоваться положительными расстояниями, но реализация стала бы длиннее.

```
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b]) {
            distance[b] = distance[a]+w;
            q.push({-distance[b],b});
        }
    }
}
```

Временная сложность этой реализации равна $O(n + m \log m)$, потому что алгоритм перебирает все вершины графа и для каждого ребра добавляет в очередь с приоритетом не более одного расстояния.

Отрицательные ребра. Эффективность алгоритма Дейкстры основана на том, что в графе нет отрицательных ребер. Если же такие ребра присутствуют, то алгоритм может давать неправильные результаты. Рассмотрим граф на рис. 7.21. Из вершины 1 в вершину 4 ведет кратчайший путь $1 \rightarrow 3 \rightarrow 4$, его длина равна 1. Однако алгоритм Дейкстры неправильно находит путь $1 \rightarrow 2 \rightarrow 4$, поскольку жадно следует по ребрам с минимальным весом.

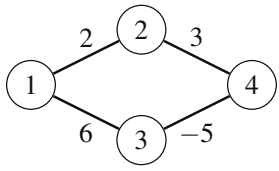


Рис. 7.21. Пример графа, на котором алгоритм Дейкстры дает неверный результат

7.3.3. *Алгоритм Флойда–Уоршелла*

Алгоритм Флойда–Уоршелла предлагает другой подход к задаче поиска кратчайших путей. В отличие от других алгоритмов, он находит кратчайшие пути между всеми парами вершин за один проход.

Алгоритм манипулирует матрицей, содержащей расстояния между парами вершин. В начальный момент матрица инициализируется на основе матрицы смежности. Затем алгоритм выполняет несколько раундов, на каждом из которых выбирает новую вершину, которая в дальнейшем может быть промежуточной вершиной в путях, и уменьшает расстояния, используя эту вершину.

Продемонстрируем работу алгоритма Флойда–Уоршелла на примере графа на рис. 7.22. В этом случае начальная матрица выглядит так:

$$\begin{bmatrix} 0 & 5 & \infty & 9 & 1 \\ 5 & 0 & 2 & \infty & \infty \\ \infty & 2 & 0 & 7 & \infty \\ 9 & \infty & 7 & 0 & 2 \\ 1 & \infty & \infty & 2 & 0 \end{bmatrix}$$

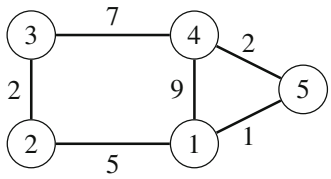


Рис. 7.22. Входные данные для алгоритма Флойда–Уоршелла

В первом раунде новой промежуточной вершиной становится вершина 1. Между вершинами 2 и 4 обнаруживается новый путь длины 14, поскольку их соединяет вершина 1. Обнаруживается также новый путь длины 6 между вершинами 2 и 5.

$$\begin{bmatrix} 0 & 5 & \infty & 9 & 1 \\ 5 & 0 & 2 & \mathbf{14} & \mathbf{6} \\ \infty & 2 & 0 & 7 & \infty \\ 9 & \mathbf{14} & 7 & 0 & 2 \\ 1 & \mathbf{6} & \infty & 2 & 0 \end{bmatrix}$$

Во втором раунде новой промежуточной вершиной становится вершина 2. В результате создаются новые пути между вершинами 1 и 3 и вершинами 3 и 5.

$$\begin{bmatrix} 0 & 5 & \mathbf{7} & 9 & 1 \\ 5 & 0 & 2 & 14 & 6 \\ \mathbf{7} & 2 & 0 & 7 & \mathbf{8} \\ 9 & 14 & 7 & 0 & 2 \\ 1 & 6 & \mathbf{8} & 2 & 0 \end{bmatrix}$$

Алгоритм продолжает работу до тех пор, пока все вершины не окажутся промежуточными. И в этот момент матрица будет содержать минимальные расстояния между всеми парами вершин:

$$\begin{bmatrix} 0 & 5 & 7 & 3 & 1 \\ 5 & 0 & 2 & 8 & 6 \\ 7 & 2 & 0 & 7 & 8 \\ 3 & 8 & 7 & 0 & 2 \\ 1 & 6 & 8 & 2 & 0 \end{bmatrix}$$

Например, мы видим, что кратчайшее расстояние между вершинами 2 и 4 равно 8. Ему соответствует путь, показанный на рис. 7.23.

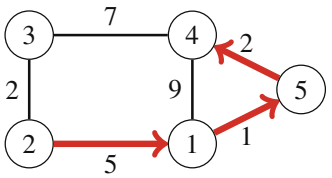


Рис. 7.23. Кратчайший путь между вершинами 2 и 4

Реализация. Реализовать алгоритм Флойда–Уоршелла очень просто. Приведенная ниже реализация строит матрицу расстояний, в которой элемент `dist`[*a*][*b*] равен кратчайшему расстоянию между вершинами *a* и *b*. Вначале алгоритм инициализирует `dist` на основе матрицы смежности графа `adj`:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) dist[i][j] = 0;
        else if (adj[i][j]) dist[i][j] = adj[i][j];
        else dist[i][j] = INF;
    }
}
```

После этого кратчайшие расстояния можно найти следующим образом:

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            dist[i][j] = min(dist[i][j],dist[i][k]+dist[k][j]);
        }
    }
}
```

Временная сложность алгоритма равна $O(n^3)$, так как он содержит три вложенных цикла, в которых перебираются вершины графа.

Поскольку реализация алгоритма Флойда–Уоршелла так проста, его можно порекомендовать даже тогда, когда требуется найти *лишь один* кратчайший путь в графе. Однако его можно использовать только для небольших графов, когда кубическая временная сложность приемлема.

7.4. Ориентированные ациклические графы

Важный класс графов составляют *ориентированные ациклические графы* (directed acyclic graph – DAG). Такие графы не содержат циклов, и если это предположение допустимо, то многие задачи решаются проще. В частности, можно выполнить топологическую сортировку графа, а затем применить динамическое программирование.

7.4.1. *Топологическая сортировка*

Топологической сортировкой называется упорядочение вершин графа – такое, что если существует путь из вершины *a* в вершину *b*, то вершина *a* будет записана раньше *b*. На рис. 7.24 показана одна из возможных топологических сортировок – [4, 1, 5, 2, 3, 6].

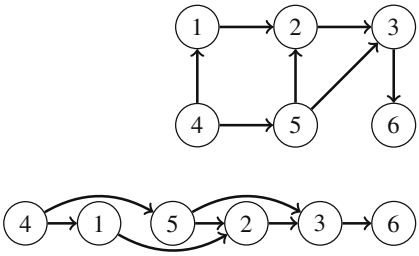


Рис. 7.24. Граф и его топологическая сортировка

Ориентированный граф допускает топологическую сортировку тогда и только тогда, когда он является ациклическим. Если граф содержит цикл, то построить топологическую сортировку невозможно, потому что ни одна вершина цикла не может предшествовать никакой другой в смысле топологического порядка. Оказывается, что поиск в глубину позволяет как проверить наличие циклов в ориентированном графе, так и построить топологическую сортировку, если циклов нет.

Идея заключается в том, чтобы перебирать вершины графа и начинать поиск в глубину в текущей вершине, если она еще не была обработана. В процессе поисков вершина может находиться в одном из трех состояний:

- состояние 0: вершина еще не обработана (белая);
- состояние 1: вершина обрабатывается (светло-серая);
- состояние 2: вершина обработана (темно-серая).

Вначале все вершины находятся в состоянии 0. Когда поиск в первый раз доходит до некоторой вершины, та переходит в состояние 1. После того как все исходящие из вершины ребра будут обработаны, вершина перейдет в состояние 2.

Если граф содержит цикл, мы обнаружим это в процессе поиска, потому что рано или поздно наткнемся на вершину в состоянии 1. В таком случае построить топологическую сортировку невозможно. Если же граф не содержит циклов, то мы сумеем построить топологическую сортировку, добавляя в список вершины в момент их перехода в состояние 2.

Теперь все готово к построению топологической сортировки графа из примера. Первый поиск (рис. 7.25) переходит от вершины 1 к вершине 6 и добавляет в список вершины 6, 3, 2 и 1. Затем второй поиск (рис. 7.26) переходит от вершины 4 к вершине 5 и добавляет в список вершины 5 и 4.

Окончательный инвертированный список [4, 5, 1, 2, 3, 6] определяет топологическую сортировку (рис. 7.27). Отметим, что топологическая сортировка не единственна, для одного и того же графа может быть несколько топологических сортировок.

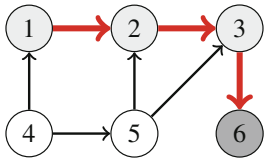


Рис. 7.25. При первом поиске в список добавляются вершины 6, 3, 2 и 1

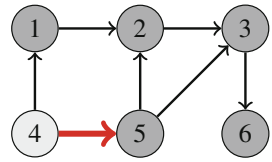


Рис. 7.26. При втором поиске в список добавляются вершины 5 и 4

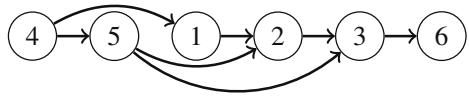


Рис. 7.27. Окончательная топологическая сортировка

На рис. 7.28 показан граф, не имеющий топологической сортировки. В процессе поиска мы встречаем вершину 2 в состоянии 1, а это индикатор того, что граф содержит цикл. И действительно, налицо цикл $2 \to 3 \to 5 \to 2$.

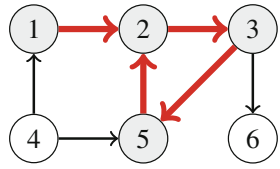


Рис. 7.28. У этого графа нет топологической сортировки, поскольку он содержит цикл

7.4.2. *Динамическое программирование*

Динамическое программирование позволяет эффективно отвечать на многие вопросы о путях в ориентированных ациклических графах, например:

- какой самый короткий или самый длинный путь из вершины *a* в вершину *b*?
- сколько существует различных путей?
- чему равно минимальное (максимальное) количество ребер в пути?
- какие вершины встречаются в каждом возможном пути?

Отметим, что для графов общего вида многие из сформулированных выше задач решить трудно или они некорректно поставлены.

Для примера решим задачу о вычислении количества путей из вершины *a* в вершину *b*. Обозначим `paths`(*x*) количества путей из вершины *a*

в вершину x. Базой рекурсии является равенство paths(a) = 1. А paths(x) для других значений аргумента вычисляется по рекуррентной формуле

$$\text{paths}(x) = \text{paths}(s_1) + \text{paths}(s_2) + \ldots + \text{paths}(s_k),$$

где $s_1, s_2, \ldots, s_k$ – вершины, для которых существует ребро, ведущее в x. Поскольку граф ациклический, значения paths можно вычислять в порядке топологической сортировки.

На рис. 7.29 показаны значения paths для нашего примера, в котором мы хотим вычислить количество путей из вершины 1 в вершину 6. Имеем

$$\text{paths}(6) = \text{paths}(2) + \text{paths}(3),$$

поскольку в вершине 6 заканчиваются ребра 2 → 6 и 3 → 6. Так как paths(2) = 2 и paths(3) = 2, мы заключаем, что paths(6) = 4. Вот эти пути:

- 1 → 2 → 3 → 6;
- 1 → 2 → 6;
- 1 → 4 → 5 → 2 → 3 → 6;
- 1 → 4 → 5 → 2 → 6.

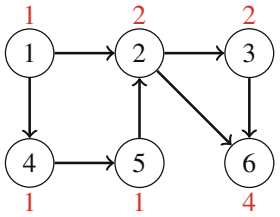


Рис. 7.29. Вычисление количества путей из вершины 1 в вершину 6

Обработка кратчайших путей. Динамическим программированием можно также воспользоваться для ответов на вопросы, касающиеся *кратчайших* путей в графах общего вида (необязательно ациклических). Точнее, если мы знаем минимальные расстояния от начальной вершины до других вершин (например, в результате применения алгоритма Дейкстры), то легко можем создать ориентированный ациклический *граф кратчайших путей*, который для каждой вершины показывает возможные способы ее достижения из начальной вершины по кратчайшему пути. Так, на рис. 7.30 показаны граф и соответствующий ему граф кратчайших путей.

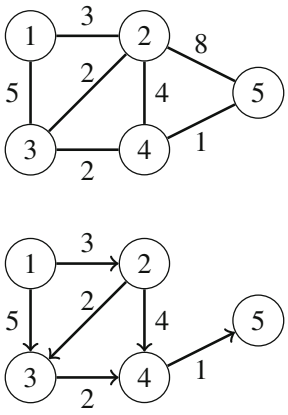


Рис. 7.30. Граф и его граф кратчайших путей

Еще раз о размене монет. На самом деле любую задачу динамического программирования можно представить в виде ориентированного ациклического графа, в котором каждая вершина соответствует состоянию динамического программирования, а ребра показывают, как состояния зависят друг от друга.

Рассмотрим, к примеру, задачу о размене суммы n монетами номиналов $\{c_1, c_2, \ldots, c_k\}$ (раздел 6.1.1). В этом случае мы можем построить граф, в котором каждая вершина соответствует денежной сумме, а ребра показывают, как можно выбирать монеты. Так, на рис. 7.31 показан граф для монет с номиналами {1, 3, 4} и $n = 6$. При таком представлении кратчайший путь из вершины 0 в вершину n соответствует решению с наименьшим числом монет, а общее число путей из вершины 0 в вершину n равно общему числу решений.

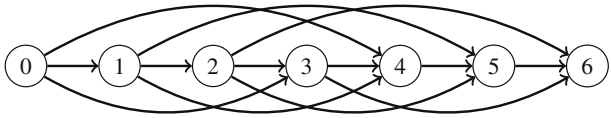


Рис. 7.31. Задача о размене монет как ориентированный ациклический граф

7.5. Графы преемников

Еще один специальный класс ориентированных графов – *графы преемников* (successor graph), в которых полустепень исхода каждой вершины равна 1, т. е. у каждой вершины всего один *преемник*. Граф преемников состоит из одной или нескольких компонент связности. Каждая компонента содержит один цикл, в который могут вести несколько путей.

Графы преемников иногда называют *функциональными графами*, потому что любому графу преемников соответствует функция succ(x), определяющая ребра графа. В качестве параметра x выступает вершина графа, а значением функции является преемник этой вершины. Например, следующая функция

x	1	2	3	4	5	6	7	8	9
succ(x)	3	5	7	6	2	2	1	6	3

определяет граф, изображенный на рис. 7.32.

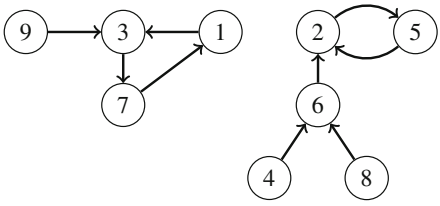


Рис. 7.32. Граф преемников

7.5.1. *Нахождение преемников*

Поскольку у каждой вершины графа преемников имеется ровно один преемник, мы можем определить функцию succ(x, k), которая возвращает вершину, в которую мы попадем, если выйдем из вершины x и сделаем k шагов вперед. В графе на рис. 7.32 succ(4, 6) = 2, так как, выйдя из вершины 4, мы через 6 шагов попадем в вершину 2 (рис. 7.33).

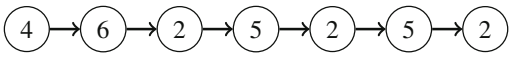


Рис. 7.33. Проход по графу преемников

Вычислить succ(x, k) можно «в лоб» – начать с вершины x и сделать k шагов вперед; это займет время $O(k)$. Однако после некоторой предварительной обработки любое значение succ(x, k) можно будет вычислить за время $O(\log k)$.

Обозначим u максимальное количество шагов, которое нас в принципе может интересовать. Идея заключается в том, чтобы заранее вычислить все значения succ(x, k), где k – степень двойки, не превосходящая u. Это можно сделать эффективно, потому что имеет место следующее рекуррентное соотношение:

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x, k) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases},$$

поскольку любой путь длины k, начинающийся в вершине x, можно разбить на два пути длины $k/2$. Вычисление всех значений succ(x, k), где k – степень двойки, не превосходящая u, занимает время $O(n \log u)$, так как для каждой вершины вычисляется $O(\log u)$ значений. В нашем примере первые значения таковы:

x	1	2	3	4	5	6	7	8	9
succ(x, 1)	3	5	7	6	2	2	1	6	3
succ(x, 2)	7	2	1	2	5	5	3	2	7
succ(x, 4)	3	2	7	2	5	5	1	2	3
succ(x, 8)	7	2	1	2	5	5	3	2	7
...									

После этого предварительного вычисления любое значение succ(x, k) можно вычислить, представив k в виде суммы степеней двойки. В таком представлении будет $O(\log k)$ слагаемых, поэтому вычисление succ(x, k) занимает время $O(\log k)$. Например, для вычисления succ(x, 11) мы воспользуемся формулой

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

В случае нашего графа

$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

7.5.2. *Обнаружение циклов*

Рассмотрим граф преемников, который содержит только путь, заканчивающийся циклом. Мы можем задать следующие вопросы: если выйти из начальной вершины, то какова будет первая встреченная вершина, принадлежащая циклу, и сколько всего вершин содержит цикл? Так, на рис. 7.34 мы начинаем проход с вершины 1, первой встреченной вершиной, принадлежащей циклу, будет 4, а всего в цикле три вершины (4, 5, 6).

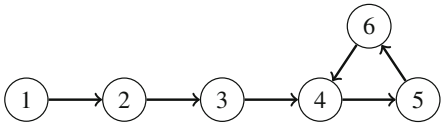


Рис. 7.34. Цикл в графе преемников

Существует простой способ обнаружить цикл: идти по графу и запоминать все посещенные вершины. Как только некоторая вершина встретится во второй раз, мы можем сказать, что она и является первой вершиной цикла. Этот метод требует времени $O(n)$ и потребляет память объемом $O(n)$. Но имеются и более эффективные алгоритмы обнаружения циклов. Их временная сложность по-прежнему равна $O(n)$, но зато они потребляют всего $O(1)$ памяти, что может быть существенно при больших n.

Одним из них является *алгоритм Флойда*, который проходит по графу и использует два указателя a и b. Вначале оба указывают на начальную вершину x. На каждой итерации a сдвигается вперед на один шаг, а b – на два шага. Процесс продолжается, пока оба указателя не встретятся:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

В этот момент указатель a прошел k шагов, а указатель b – $2k$ шагов, поэтому длина цикла делит k. Следовательно, первую вершину, принадлежащую циклу, можно найти, переставив указатель a на x и продвигая указатели вперед по одному шагу, пока они снова не встретятся.

```
a = x;
while (a != b) {
    a = succ(a);
```

```
    b = succ(b);
}
first = a;
```

Теперь длину цикла можно найти следующим образом:

```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

7.6. Минимальные остовные деревья

Остовное дерево содержит все вершины графа и некоторое подмножество его ребер, такое, что любые две вершины соединены путем, проходящим по этим ребрам. Как и любое дерево, остовное дерево является связным ациклическим графом. *Весом* остовного дерева называется суммарный вес его ребер. На рис. 7.35 показаны граф и одно из его остовных деревьев. Вес этого дерева равен 3 + 5 + 9 + 3 + 2 = 22.

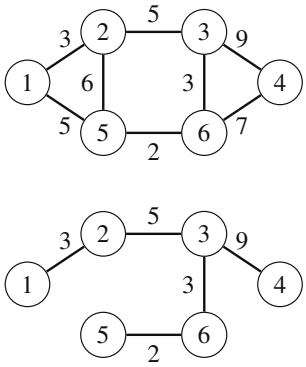


Рис. 7.35. Граф и его остовное дерево

Минимальным остовным деревом называется остовное дерево минимального веса. На рис. 7.36 показано минимальное остовное дерево вышеупомянутого графа с весом 20. Аналогично *максимальным остовным деревом* называется остовное дерево максимального веса. На рис. 7.37 показано максимальное остовное дерево того же графа с весом 32. Отметим, что у графа может быть несколько минимальных и максимальных остовных деревьев.

Для построения минимального и максимального остовных деревьев есть несколько жадных алгоритмов. В этом разделе обсуждаются два ал-

горитма, которые обрабатывают ребра графа, упорядоченные по весу. Мы будем искать минимальное остовное дерево, но для поиска максимального остовного дерева применимы те же алгоритмы – только ребра нужно обрабатывать в противоположном порядке.

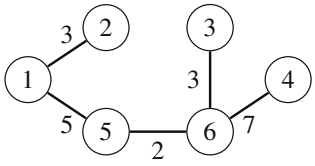


Рис. 7.36. Минимальное остовное дерево с весом 20

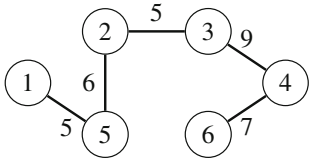


Рис. 7.37. Максимальное остовное дерево с весом 32

7.6.1. *Алгоритм Краскала*

Алгоритм Краскала строит минимальное остовное дерево, жадно добавляя в граф ребра. Начальное остовное дерево содержит только вершины графа и ни одного ребра. Затем алгоритм перебирает ребра, отсортированные в порядке возрастания весов, и всякий раз добавляет ребро в граф, если при этом не образуется цикл.

Алгоритм манипулирует компонентами связности графа. Вначале каждая вершина графа принадлежит отдельной компоненте. При добавлении в граф нового ребра две компоненты объединяются. В конце все вершины будут принадлежать одной компоненте, она и является минимальным остовным деревом.

Для примера построим минимальное остовное дерево графа на рис. 7.35. Первым делом отсортируем ребра в порядке возрастания весов:

Ребро	Вес
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

Затем проходим по списку и добавляем ребро в граф, если оно соединяет две разные компоненты. Шаги алгоритма показаны на рис. 7.38. В начальный момент каждая вершина принадлежит своей компоненте. Затем в граф добавляются первые ребра из списка (5–6, 1–2, 3–6 и 1–5). Следую-

щее ребро, 2–3, не добавляется, потому что это привело бы к образованию цикла. Так же обстоит дело для ребра 2–5. Ребро 4–6 добавляется, после чего минимальное остовное дерево готово.

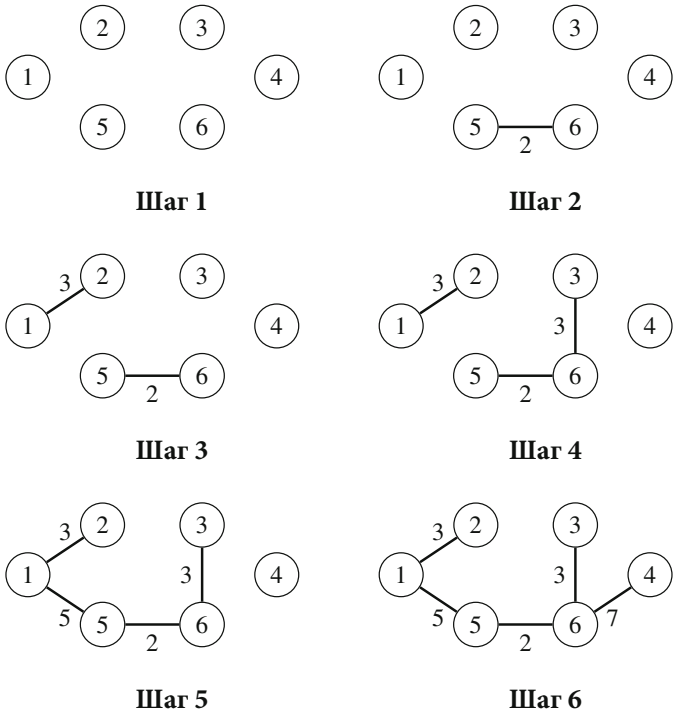


Рис. 7.38. Алгоритм Краскала

Почему это работает? А действительно – *почему* алгоритм Краскала работает? Почему жадная стратегия гарантирует нахождение именно минимального остовного дерева? Посмотрим, что произойдет, если ребро с минимальным весом *не* включено в остовное дерево – в нашем случае это означает, что не включено ребро 5–6. Мы не знаем точную структуру такого остовного дерева, но какие-то ребра оно должно содержать наверняка. Предположим, что дерево выглядит, как показано на рис. 7.39.

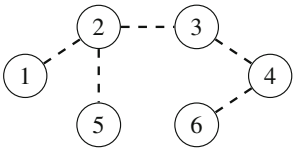


Рис. 7.39. Гипотетическое минимальное остовное дерево

Однако остовное дерево на рис. 7.39 не может быть минимальным, потому что никто не мешает нам удалить из него какое-то ребро и заменить его ребром с минимальным весом 5–6. Получится остовное дерево с *меньшим* весом, показанное на рис. 7.40.

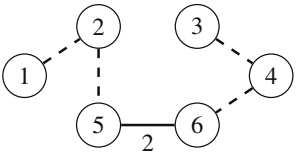


Рис. 7.40. Включение ребра 5–6 уменьшает вес остовного дерева

Поэтому включать ребро с минимальным весом необходимо, если мы хотим получить минимальное остовное дерево. Применяя аналогичное рассуждение, можно показать, что необходимо включать также ребро со следующим по величине весом и т. д. Поэтому алгоритм Краскала всегда дает минимальное остовное дерево.

Реализация. При реализации алгоритма Краскала удобно использовать представление графа в виде списка ребер. На первом этапе мы сортируем этот список за время $O(m \log m)$, а на втором этапе строим минимальное остовное дерево:

```
for (...) {
   if (!same(a,b)) unite(a,b);
}
```

Цикл перебирает находящиеся в списке ребра и обрабатывает каждое ребро (a, b), соединяющее вершины a и b. Нам необходимы две функции: `same` определяет, принадлежат ли вершины a и b одной связной компоненте, а `unite` объединяет компоненты, содержащие a и b.

Проблема в том, как эффективно реализовать функции `same` и `unite`. Можно, например, реализовать `same` в виде обхода графа и проверить, существует ли путь из a в b. Но временная сложность такой функции была бы равна $O(n + m)$, и алгоритм работал бы слишком медленно, т. к. `same` вызывается для каждого ребра графа.

Мы решим задачу с помощью системы непересекающихся множеств – структуры данных, позволяющей реализовать обе функции за время $O(\log n)$. Таким образом, временная сложность алгоритма Краскала после сортировки списка ребер будет равна $O(m \log n)$.

7.6.2. *Система непересекающихся множеств*

Система непересекающихся множеств (union-find structure) состоит из коллекции множеств. Эти множества попарно не пересекаются, т. е. каждый элемент принадлежит только одному множеству. Поддерживаются две операции с временной сложностью $O(\log n)$: `unite` объединяет два множества, а `find` находит представителя множества, содержащего заданный элемент.

В системе непересекающихся множеств в каждом множестве выбирается по одному представителю, и существует путь от любого элемента множества к его представителю. Рассмотрим, к примеру, множества {1, 4, 7}, {5} и {2, 3, 6, 8}. На рис. 7.41 показано, как можно их представить.

В данном случае представителями являются элементы 4, 5 и 2. Чтобы найти представителя любого элемента, нужно проследовать по пути, который начинается в этом элементе. Например, элемент 2 является представителем элемента 6, потому что существует путь $6 \rightarrow 3 \rightarrow 2$. Два элемента

принадлежат одному и тому же множеству тогда и только тогда, когда у них одинаковые представители.

Чтобы объединить два множества, представитель одного соединяется с представителем другого. На рис. 7.42 показан один из способов соединить множества {1, 4, 7} и {2, 3, 6, 8}. Начиная с этого момента представителем всего множества является элемент 2, а старый представитель 4 указывает на него.

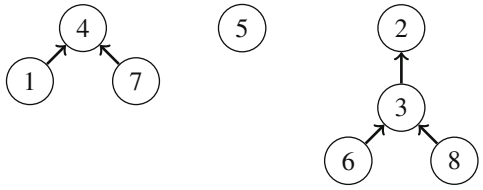


Рис. 7.41. Система трех непересекающихся множеств

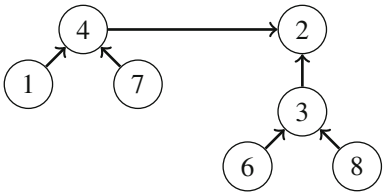


Рис. 7.42. Объединение двух множеств в одно

Эффективность системы непересекающихся множеств зависит от того, как множества объединяются. Оказывается, что можно следовать такой простой стратегии: всегда соединять представителя *меньшего* множества с представителем *большего* (если размеры множеств одинаковы, то выбор произволен). При такой стратегии длина любого пути будет иметь порядок $O(\log n)$, так что мы сможем эффективно найти представителя любого элемента, проследовав по соответствующему пути.

Реализация. Систему непересекающихся множеств удобно реализовать с помощью массивов. Для каждого элемента в массиве `link` хранится следующий элемент пути или сам этот элемент, если он является представителем, а в массиве `size` для каждого представителя хранится размер соответствующего множества.

Вначале каждое множество состоит из одного элемента:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

Функция `find` возвращает представителя элемента x. Его можно найти, проследовав по пути, начинающемуся в x.

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

Функция `same` проверяет, принадлежат ли элементы a и b одному множеству. Это легко сделать, воспользовавшись функцией `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

Функция unite объединяет множества, содержащие элементы *a* и *b* (они должны принадлежать разным множествам). Сначала функция находит представителей множеств, а затем соединяет представителя меньшего множества с представителем большего.

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

Временная сложность функции find равна $O(\log n)$ в предположении, что длина каждого пути равна $O(\log n)$. В этом случае функции same и unite также работают за время $O(\log n)$. Функция unite гарантирует, что длина каждого пути будет иметь порядок $O(\log n)$, поскольку соединяет меньшее множество с большим.

Сжатие путей. Функцию find можно реализовать и по-другому:

```
int find(int x) {
    if (x == link[x]) return x;
    return link[x] = find(link[x]);
}
```

В этой функции используется *сжатие путей*: после выполнения операции каждый элемент, находящийся на пути, указывает непосредственно на своего представителя. Можно показать, что при такой реализации find амортизированное время выполнения операций системы непересекающихся множеств имеет порядок $O(\alpha(n))$, где $\alpha(n)$ – обратная функция Аккермана, которая растет очень медленно (почти постоянна). Однако сжатие путей нельзя использовать в некоторых приложениях системы непересекающихся множеств, например в задаче о динамической связности (раздел 15.5.4).

7.6.3. *Алгоритм Прима*

Алгоритм Прима дает другой способ построения минимального остовного дерева. Сначала он добавляет в дерево произвольную вершину, а при добавлении каждой следующей вершины выбирает ребро с минимальным весом. После того как все вершины добавлены, минимальное остовное дерево построено.

Алгоритм Прима напоминает алгоритм Дейкстры. Разница в том, что алгоритм Дейкстры всегда выбирает вершину с минимальным расстоянием от начальной вершины, а алгоритм Прима выбирает для добавления в дерево вершину, принадлежащую ребру с минимальным весом.

На рис. 7.43 показано, как алгоритм Прима строит минимальное остовное дерево для рассмотренного выше графа, начиная с вершины 1. Как и алгоритм Дейкстры, алгоритм Прима можно эффективно реализовать с помощью очереди с приоритетом. Очередь должна содержать все вершины, которые можно соединить с уже построенной частью дерева одним ребром, причем эти вершины должны храниться в порядке возрастания весов соответствующих ребер.

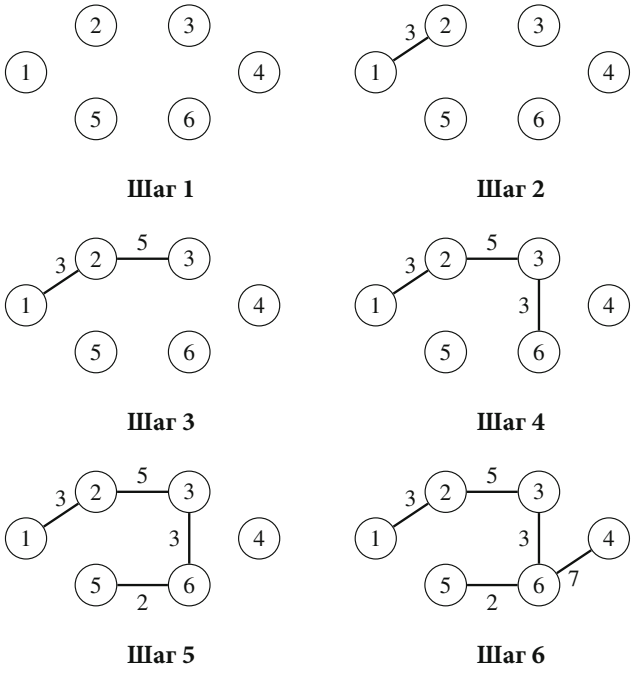


Рис. 7.43. Алгоритм Прима

Временная сложность алгоритма Прима равна $O(n + m \log m)$, т. е. такая же, как у алгоритма Дейкстры. На практике алгоритмы Прима и Краскала одинаково эффективны, так что выбор между ними – дело вкуса. Тем не менее на олимпиадах чаще выбирают алгоритм Краскала.

Глава 8

Избранные вопросы проектирования алгоритмов

В разделе 8.1 рассматриваются алгоритмы с параллельным просмотром разрядов, в которых для эффективной обработки данных используются поразрядные операции. В типичном случае мы можем заменить цикл for поразрядными операциями, что существенно уменьшает время работы алгоритма.

В разделе 8.2 изложена техника амортизационного анализа, применяемая для оценки времени выполнения последовательности операций в алгоритме. С помощью этой техники мы проанализируем алгоритмы нахождения ближайшего меньшего элемента и минимума в скользящем окне.

В разделе 8.3 обсуждаются троичный поиск и другие методы эффективного вычисления минимумов некоторых функций.

8.1. Алгоритмы с параллельным просмотром разрядов

Алгоритмы с параллельным просмотром разрядов основаны на том факте, что отдельными разрядами числа можно манипулировать параллельно, применяя поразрядные операции. Поэтому один из способов проектирования алгоритмов – представить шаги алгоритма таким образом, чтобы их можно было эффективно реализовать с помощью поразрядных операций.

8.1.1. *Расстояние Хэмминга*

Расстоянием Хэмминга `hamming`(a, b) между строками a и b одинаковой длины называется количество позиций, в которых эти строки различаются. Например:

$$\texttt{hamming}(01101, 11001) = 2.$$

Рассмотрим следующую задачу: дано n битовых строк длины k, вычислить минимальное расстояние Хэмминга между двумя строками. Например, для строк [00111, 01101, 11110] ответом будет 2, потому что

- hamming(00111, 01101) = 2;
- hamming(00111, 11110) = 3;
- hamming(01101, 11110) = 3.

Задачу можно решить «в лоб», вычислив расстояние Хэмминга между каждыми двумя строками. Временная сложность такого алгоритма равна $O(n^2k)$. Для вычисления расстояния между строками a и b служит следующая функция:

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```

Но поскольку строки состоят из бит, решение можно оптимизировать, если хранить строки в виде целых чисел и вычислять расстояние между ними с помощью поразрядных операций. В частности, если $k \leqslant 32$, то строки можно хранить как значения типа int и для вычисления расстояния использовать такую функцию:

```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

Здесь операция *ИСКЛЮЧАЮЩЕЕ ИЛИ* строит строку, в которой единицы находятся в тех позициях, где a и b различаются. Затем число единичных разрядов вычисляется функцией __builtin_popcount.

В табл. 8.1 приведены результаты сравнения исходного алгоритма и алгоритма с параллельным просмотром разрядов с точки зрения времени выполнения на современном компьютере. Алгоритм с параллельным просмотром разрядов оказался примерно в 20 раз быстрее.

Таблица 8.1. Время работы алгоритмов, вычисляющих минимальное расстояние Хэмминга для n битовых строк длины $k = 30$

Размер n	Исходный алгоритм (с)	Алгоритм с параллельным просмотром разрядов (с)
5000	0.84	0.06
10 000	3.24	0.18
15 000	7.23	0.37
20 000	12.79	0.63
25 000	19.99	0.97

8.1.2. *Подсчет подсеток*

Рассмотрим далее такую задачу: дана сетка $n \times n$ с черными (1) и белыми (0) клетками, вычислить количество подсеток, у которых все угловые клетки черные. На рис. 8.1 показаны две такие подсетки.

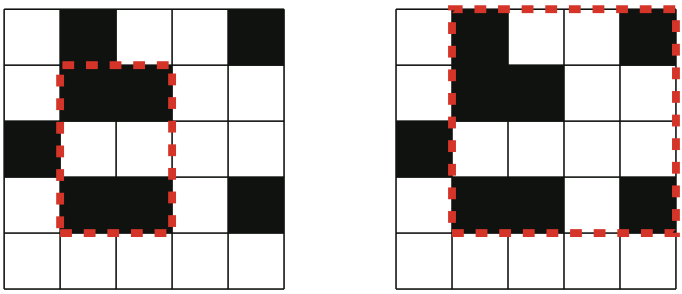


Рис. 8.1. Эта сетка содержит две подсетки с черными углами

Для решения задачи существует алгоритм с временной сложностью $O(n^3)$: перебрать все $O(n^2)$ пар строк и для каждой пары (a, b) вычислить – за время $O(n)$ – количество столбцов – таких, что на их пересечении со строками a и b находится черная клетка. В следующем коде предполагается, что `color`[*y*][*x*] – цвет клетки на пересечении строки *y* и столбца *x*:

```
int count = 0;
for (int i = 0; i < n; i++) {
   if (color[a][i] == 1 && color[b][i] == 1) {
      count++;
   }
}
```

Зная, что существует `count` столбцов, в которых обе интересующие нас клетки черные, мы можем вычислить количество искомых подсеток, в которых *a* – первая строка, а *b* – вторая, по формуле `count(count – 1)/2`.

Чтобы построить алгоритм с параллельным просмотром разрядов, представим каждую строку *k* в виде битового множества длины *n* `row`[*k*], в котором единицы соответствуют черным клеткам. Тогда количество столбцов, на пересечении которых со строками *a* и *b* находятся черные клетки, можно вычислить с помощью операции *И* и подсчета единичных разрядов. Благодаря битовым множествам это делается просто:

```
int count = (row[a]&row[b]).count();
```

В табл. 8.2 приведены результаты сравнения исходного алгоритма и алгоритма с параллельным просмотром разрядов для сеток разного размера. В некоторых случаях алгоритм с параллельным просмотром разрядов оказался в 30 раз быстрее.

Таблица 8.2. Время работы алгоритмов подсчета подсеток

Размер сетки n	Исходный алгоритм (с)	Алгоритм с параллельным просмотром разрядов (с)
1000	0.65	0.05
1500	2.17	0.14
2000	5.51	0.30
2500	12.67	0.52
3000	26.36	0.87

8.1.3. *Достижимость в графах*

Для заданного ориентированного ациклического графа с n вершинами рассмотрим задачу о вычислении для каждой вершины x значения функции reach(x): числа вершин, достижимых из x. На рис. 8.2 показаны граф и значения reach для него.

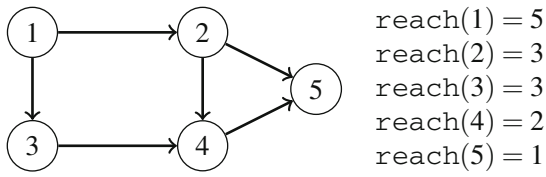


Рис. 8.2. Граф и значения reach для него.
Например, reach(2) = 3, потому что из вершины 2 достижимы вершины 2, 4 и 5

Задачу можно решить методом динамического программирования за время $O(n^2)$, построив для каждой вершины список достижимых из нее вершин. Затем мы представим каждый список в виде битового множества размера n. Это позволит эффективно вычислить объединение двух списков с помощью операции *ИЛИ*. В предположении, что reach – массив битовых множеств, а граф представлен списками смежности в массиве adj, вычисление для вершины x можно выполнить следующим образом:

```
reach[x][x] = 1;
for (auto u : adj[x]) {
    reach[x] |= reach[u];
}
```

В табл. 8.3 приведено время работы этого алгоритма с параллельным просмотром разрядов. В каждом случае граф состоял из n вершин и $2n$ случайных ребер $a \to b$, где $a < b$. Отметим, что при больших n алгоритм потребляет очень много памяти. Во многих соревнованиях объем памяти не превышает 512 Мб.

Таблица 8.3. Время работы алгоритмов подсчета достижимых вершин

Размер графа n	Время работы (с)	Потребление памяти (МБ)
$2 \cdot 10^4$	0.06	50
$4 \cdot 10^4$	0.17	200
$6 \cdot 10^4$	0.32	450
$8 \cdot 10^4$	0.51	800
10^5	0.78	1250

8.2. Амортизационный анализ

Часто временная сложность алгоритма непосредственно вытекает из его структуры, но иногда прямолинейный анализ не дает истинной картины эффективности. Для анализа последовательности операций с различной временной сложностью можно применить *амортизационный анализ*. Идея в том, чтобы оценить суммарное время выполнения всех операций, а не каждой в отдельности.

8.2.1. *Метод двух указателей*

В этом методе используются два указателя, пробегающих массив. Каждый указатель сдвигается только в одном направлении, что гарантирует эффективность алгоритма. В качестве первого примера этой техники рассмотрим следующую задачу: даны массив n положительных целых чисел и число x, требуется найти подмассив, сумма которого в точности равна x, или сообщить, что такого подмассива не существует.

С помощью метода двух указателей задачу можно решить за время $O(n)$. Указатели будут отмечать первый и последний элементы подмассива. На каждом шаге левый указатель сдвигается на одну позицию вправо, а правый сдвигается вправо на столько позиций, чтобы сумма получившегося подмассива не превосходила x. Если сумма оказывается в точности равна x, решение найдено.

На рис. 8.3 показано, как обрабатывается массив, когда сумма $x = 8$. Вначале подмассив содержит значения 1, 3 и 2, сумма которых равна 6. Затем левый указатель сдвигается на одну позицию вправо, а правый остается на месте, потому что иначе сумма превысила бы x. Наконец, левый указатель сдвигается еще на одну позицию вправо, а правый сдвигается на две позиции. Сумма этого подмассива равна 2 + 5 + 1 = 8, так что искомый подмассив найден.

Время работы алгоритма зависит от того, на сколько позиций сдвинется правый указатель. Не существует полезной верхней границы величины сдвига на *одном* шаге, но мы знаем, что на протяжении работы алгоритма

этот указатель сдвинется *суммарно на* $O(n)$ позиций, потому что сдвиг возможен только вправо. Поскольку левый и правый указатели сдвигаются на $O(n)$ позиций, то алгоритм работает за время $O(n)$.

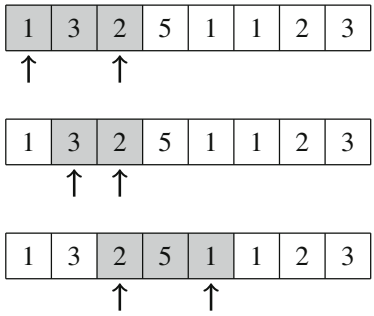


Рис. 8.3. Нахождение подмассива с суммой 8 методом двух указателей

Задача о сумме двух элементов. Методом двух указателей можно решить также *задачу о сумме двух элементов* (2SUM): даны массив n чисел и число x, требуется найти в массиве два элемента, сумма которых равна x, или сообщить, что таких элементов не существует.

Для решения задачи сначала отсортируем массив в порядке возрастания, а затем пробежимся по нему, сдвигая два указателя. Левый указатель первоначально указывает на первый элемент и сдвигается на одну позицию вправо на каждом шаге. Правый указатель первоначально указывает на последний элемент и на каждом шаге сдвигается влево, пока сумма левого и правого элементов остается не больше x. Если сумма в точности равна x, то решение найдено.

На рис. 8.4 показано, как обрабатывается массив при $x = 12$. Вначале сумма элементов равна $1 + 10 = 11$, это меньше x. Затем левый указатель сдвигается на одну позицию вправо, а правый – на три позиции влево, так что получается сумма $4 + 7 = 11$. После этого левый указатель сдвигается еще на одну позицию вправо. Правый указатель остается на месте, что дает нам решение $5 + 7 = 12$.

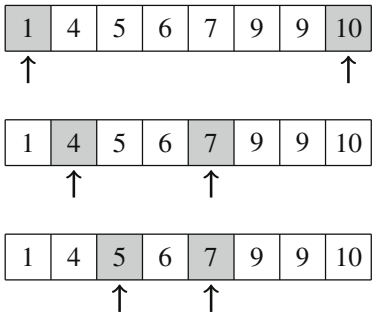


Рис. 8.4. Решение задачи о сумме двух элементов методом двух указателей

Временная сложность этого алгоритма равна $O(n \log n)$, потому что для предварительной сортировки массива нужно время $O(n \log n)$, а затем оба указателя сдвигаются на $O(n)$ позиций.

Отметим, что задачу можно решить и по-другому, тоже за время $O(n \log n)$, воспользовавшись двоичным поиском. Для этого мы сначала сортируем массив, а затем обходим его и для каждого элемента выполняем двоичный поиск второго элемента, такого, что сумма обоих равна x. На самом деле многие задачи, которые можно решить методом двух указателей, решаются также с помощью сортировки или структур на основе множеств, иногда при этом добавляется логарифмический множитель.

Интересна также более общая задача о сумме k элементов (kSUM). В этом случае требуется найти k элементов, сумма которых равна x. Оказывается, что задачу 3SUM можно решить за время $O(n^2)$, обобщив описанный выше алгоритм 2SUM. Сможете ли вы найти решение? Долгое время считалось, что $O(n^2)$ – лучшая оценка временной сложности для задачи 3SUM. Но в 2014 г. Grønlund и Pettie [14] показали, что это не так.

8.2.2. *Ближайшие меньшие элементы*

Амортизационный анализ часто используется для оценки числа операций, выполненных над структурой данных. Распределение операций может быть неравномерным, т. е. большинство операций имеет место на определенном этапе алгоритма, но их общее число ограничено.

Предположим, к примеру, что мы хотим для каждого элемента массива найти *ближайший меньший элемент*, т. е. первый элемент, предшествующий данному и меньший его по величине. Если такого элемента не существует, то алгоритм должен сообщить об этом. Мы предложим эффективное решение этой задачи с помощью стека.

Будем обходить массив слева направо, поддерживая при этом стек элементов массива. На каждом шаге мы удаляем из стека элементы до тех пор, пока элемент на вершине не окажется меньше текущего элемента или стек не станет пустым. После этого мы сообщаем, что элемент на вершине стека – ближайший меньший текущего или, если стек пуст, что такого элемента не существует. И наконец, помещаем текущий элемент в стек.

На рис. 8.5 показано, как этот алгоритм обрабатывает массив. Вначале в стек помещается элемент 1. Поскольку это первый элемент массива, очевидно, что у него нет ближайшего меньшего. Далее в стек помещаются элементы 3 и 4. Ближайшим меньшим элементом для 4 является 3, а ближайшим меньшим для 3 – 1. Следующий элемент 2 меньше элементов 3 и 4, находящихся на вершине стека, поэтому оба они удаляются. И ближайшим меньшим элементом для 2 оказывается 1. Затем элемент 2 помещается в стек. Работа алгоритма продолжается, пока не будет обработан весь массив.

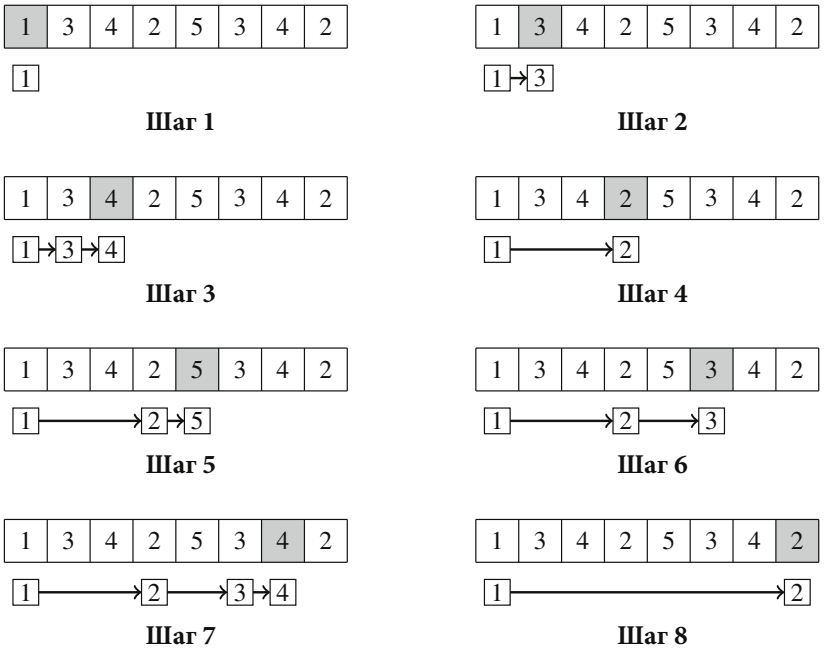


Рис. 8.5. Нахождение ближайших меньших элементов за линейное время с использованием стека

Эффективность алгоритма зависит от общего числа операций со стеком. Если текущий элемент больше элемента на вершине стека, то он сразу добавляется в стек, это эффективно. Но иногда стек содержит несколько бóльших элементов, и на их удаление приходится тратить время. Тем не менее каждый элемент помещается в стек *ровно один раз* и удаляется из стека *не более одного раза*. Поэтому с каждым элементом связано $O(1)$ операций со стеком, так что временная сложность алгоритма равна $O(n)$.

8.2.3. *Минимум в скользящем окне*

Скользящим окном называется подмассив постоянного размера, который движется вдоль массива слева направо. В каждой позиции окна мы вычисляем какую-то информацию о попавших внутрь элементах. Далее мы рассмотрим задачу о запоминании *минимума в скользящем окне* – для каждого скользящего окна требуется найти наименьшее значение.

Для вычисления минимумов в скользящих окнах можно воспользоваться той же идеей, что при нахождении ближайших меньших элементов. Но на этот раз нам понадобится очередь, в которой каждый элемент больше предыдущего, а первый элемент всегда соответствует минимальному элементу внутри окна. После каждого сдвига окна мы будем удалять из конца очереди элемент до тех пор, пока очередной элемент не окажется меньше вновь появившегося в окне элемента или очередь не опустеет. Первый

элемент мы также удалим, если он больше не находится внутри окна. И наконец, мы помещаем новый элемент в очередь.

На рис. 8.6 показано, как алгоритм обрабатывает массив, когда размер скользящего окна равен 4. В первой позиции окна наименьшее значение равно 1. Затем окно сдвигается на одну позицию вправо. Новый элемент 3 меньше элементов 4 и 5, находящихся в очереди, поэтому эти элементы удаляются из очереди, а элемент 3 помещается в нее. Наименьшим по-прежнему остается элемент 1. Далее окно снова сдвигается вправо, и наименьший элемент 1 «выпадает» из окна. Поэтому он удаляется из очереди, и наименьшим становится элемент 3. В очередь также помещается новый элемент 4. Следующий новый элемент 1 меньше всех элементов в очереди, поэтому все они удаляются, и в очереди остается только элемент 1. Наконец, окно достигает последней позиции. Элемент 2 помещается в очередь, но наименьшим элементом внутри окна по-прежнему является 1.

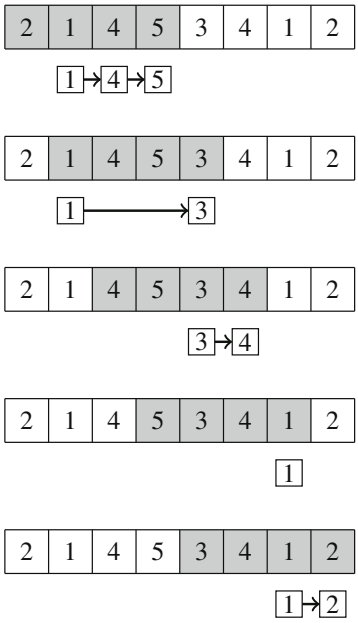


Рис. 8.6. Нахождение минимумов в скользящем окне за линейное время

Поскольку каждый элемент помещается в очередь ровно один раз, а удаляется не более одного раза, то алгоритм работает за время $O(n)$.

8.3. Нахождение минимальных значений

Рассмотрим функцию $f(x)$, которая сначала убывает, достигает минимума, после чего возрастает. Пример такой функции приведен на рис. 8.7, ее минимум отмечен стрелочкой. Если мы знаем, что функция ведет себя таким образом, то можем эффективно найти минимальное значение.

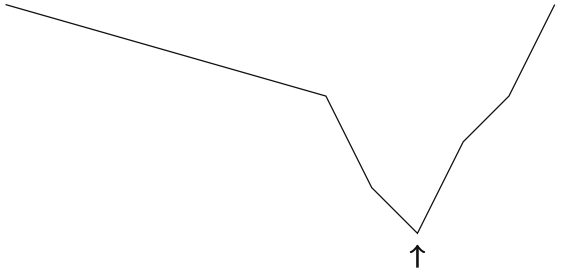


Рис. 8.7. Функция и ее минимальное значение

8.3.1. *Тернарный поиск*

Тернарный поиск – это эффективный способ нахождения минимумов функций, которые сначала монотонно убывают, а затем монотонно возрастают. Пусть известно, что значение x, доставляющее минимум $f(x)$, находится в диапазоне $[x_L, x_R]$. Идея заключается в том, чтобы разбить диапазон на три равные части $[x_L, a]$, $[a, b]$ и $[b, x_R]$, положив

$$a = \frac{2x_L + x_R}{3} \quad \text{и} \quad b = \frac{x_L + 2x_R}{3}.$$

Тогда если $f(a) < f(b)$, то можно заключить, что минимум принадлежит отрезку $[x_L, b]$, в противном случае он должен находиться внутри отрезка $[a, x_R]$. Затем поиск рекурсивно продолжается, пока размер отрезка не станет достаточно мал.

На рис. 8.8 показан первый шаг тернарного поиска. Поскольку $f(a) > f(b)$, новым диапазоном становится $[a, x_R]$.

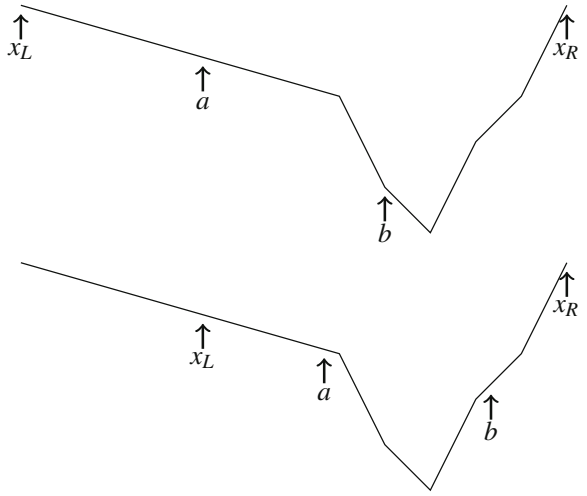


Рис. 8.8. Нахождение минимума методом тернарного поиска

На практике часто рассматриваются функции, параметрами которых являются целые числа, а поиск завершается, когда будет найден диапазон, содержащий единственный элемент. Поскольку размер нового диапазона всегда составляет 2/3 от размера предыдущего, временная сложность алгоритма равна $O(\log n)$, где n – число элементов в исходном диапазоне.

Отметим, что при работе с целыми параметрами можно использовать *бинарный поиск* вместо тернарного, поскольку достаточно найти первое значение x, для которого $f(x) \leqslant f(x + 1)$.

8.3.2. *Выпуклые функции*

Функция называется *выпуклой*, если отрезок, соединяющий любые две точки на ее графике, целиком лежит выше или ниже соответствующей дуги графика. На рис. 8.9 показан график выпуклой функции $f(x) = x^2$. Действительно, отрезок, соединяющий точки a и b, лежит выше дуги графика.

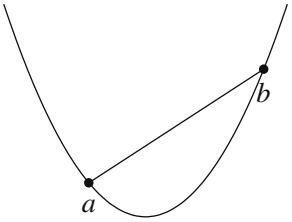


Рис. 8.9. Пример выпуклой функции: $f(x) = x^2$

Если известно, что минимальное значение выпуклой функции находится в диапазоне $[x_L, x_R]$, то для его нахождения можно применить тернарный поиск. Отметим, однако, что выпуклая функция может достигать минимума в нескольких точках. Например, функция $f(x) = 0$ выпуклая, и ее минимум равен 0.

Выпуклые функции обладают несколькими полезными свойствами: если $f(x)$ и $g(x)$ – выпуклые функции, то $f(x) + g(x)$ и $\max(f(x), g(x))$ тоже выпуклы. Следовательно, если имеется n выпуклых функций f_1, f_2, ..., f_n, то сразу можно сказать, что функция $f_1 + f_2 + ... + f_n$ выпуклая, и применить для нахождения ее минимума тернарный поиск.

8.3.3. *Минимизация сумм*

Пусть дано n чисел a_1, a_2, ..., a_n и требуется найти значение x, доставляющее минимум сумме

$$|a_1 - x| + |a_2 - x| + ... + |a_n - x|.$$

Например, для чисел [1, 2, 9, 2, 6] оптимальное решение $x = 2$, при этом получается сумма

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

Поскольку каждая функция $|a_k - x|$ выпукла, их сумма также выпукла, поэтому для нахождения оптимального значения x можно применить тернарный поиск. Но есть и более простое решение. Оказывается, что оптимальное значение x всегда равно *медиане* чисел, т. е. среднему элементу, получающемуся после их сортировки. Так как список [1, 2, 9, 2, 6] после сортировки переходит в [1, 2, 2, 6, 9], то медиана равна 2.

Медиана всегда дает оптимальное решение, потому что если x меньше медианы, то сумма уменьшится при увеличении x, а если x больше медианы, то сумма уменьшится при уменьшении x. Если n четно и существует две медианы, то обе они, а также все значения между ними дают оптимальные решения.

Далее рассмотрим задачу о минимизации функции

$$(a_1 - x)^2 + (a_2 - x)^2 + \ldots + (a_n - x)^2.$$

Для чисел [1, 2, 9, 2, 6] оптимальное решение $x = 4$, при этом получается сумма

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46.$$

И снова функция выпуклая, поэтому задачу можно было бы решить методом тернарного поиска, но есть и более простое решение: минимум достигается, когда x – *среднее арифметическое* чисел. В нашем примере среднее арифметическое равно $(1 + 2 + 9 + 2 + 6)/5 = 4$. Это можно доказать, представив сумму в виде

$$nx^2 - 2x(a_1 + a_2 + \ldots + a_n) + (a_1^2 + a_2^2 + \ldots + a_n^2).$$

Последнее слагаемое не зависит от x, так что его можно игнорировать. А первые два слагаемых образуют функцию $nx^2 - 2xs$, где $s = a_1 + a_2 + \ldots + a_n$. Это парабола, направленная рогами вверх, ее корни равны $x = 0$ и $x = 2s/n$, а минимальное значение достигается в точке $x = s/n$, расположенной посередине между корнями, а это и есть среднее арифметическое чисел $a_1, a_2, \ldots, a_n$.

Глава 9

Запросы по диапазону

В этой главе мы обсудим структуры данных для эффективной обработки запросов к массивам по диапазону. Типичные примеры таких запросов: вычисление суммы по диапазону и нахождение минимума по диапазону.

В разделе 9.1 рассматривается простая ситуация, когда массив не модифицируется между запросами. В таком случае достаточно подвергнуть массив предварительной обработке, чтобы можно было эффективно найти ответ на любой возможный запрос. Сначала мы научимся обрабатывать запросы о сумме с помощью массива префиксных сумм, а затем обсудим алгоритм разреженной таблицы для обработки запросов о минимуме.

В разделе 9.2 представлены две древовидные структуры, позволяющие эффективно обрабатывать запросы и изменять значения элементов массива. Двоичное индексное дерево поддерживает запросы о сумме и может рассматриваться как динамический вариант массива префиксных сумм. Дерево отрезков – более универсальная структура, поддерживающая запросы о сумме, запросы о минимуме и некоторые другие. Все операции над обеими структурами имеют логарифмическую сложность.

9.1. Запросы к статическим массивам

В этом разделе мы рассмотрим ситуацию, когда массив *статический*, т. е. не изменяется между запросами. Тогда достаточно подвергнуть его предварительной обработке с целью эффективного получения ответов на запросы по диапазону.

Сначала обсудим простой способ обработки запросов о сумме с помощью массива префиксных сумм, который допускает обобщение на многомерный случай. Затем изучим несколько более сложный алгоритм разреженной таблицы для обработки запросов о минимуме. Запросы о максимуме обрабатываются аналогично.

9.1.1. *Запросы о сумме*

Обозначим $\mathtt{sum}_q(a, b)$ («запрос о сумме по диапазону») сумму элементов массива в диапазоне $[a, b]$. Любой запрос о сумме можно эффективно обработать, если сначала построить *массив префиксных сумм*. Каждый эле-

мент этого массива равен сумме нескольких первых элементов исходного массива, т. е. значение k-го элемента равно $\mathrm{sum}_q(0, k)$. На рис. 9.1 показаны массив и соответствующий ему массив префиксных сумм.

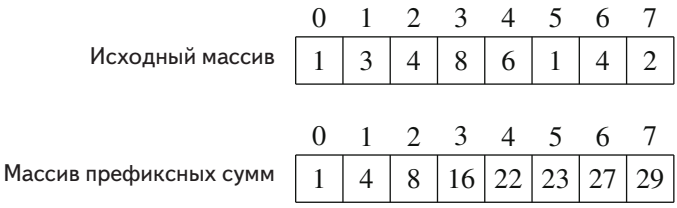


Рис. 9.1. Массив и соответствующий ему массив префиксных сумм

Массив префиксных сумм можно построить за время $O(n)$. Поскольку он содержит значения $\mathrm{sum}_q(0, k)$ для всех k, то любое значение вида $\mathrm{sum}_q(a, b)$ можно вычислить за время $O(1)$ по формуле

$$\mathrm{sum}_q(a, b) = \mathrm{sum}_q(0, b) - \mathrm{sum}_q(0, a - 1).$$

Положив $\mathrm{sum}_q(0, -1) = 0$, мы распространим эту формулу и на случай $a = 0$.

На рис. 9.2 показано, как вычисляется сумма по диапазону [3, 6] с помощью массива префиксных сумм. Легко вычислить, что $\mathrm{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. Пользуясь же массивом префиксных сумм, мы можем обойтись всего двумя значениями:

$$\mathrm{sum}_q(3, 6) = \mathrm{sum}_q(0, 6) - \mathrm{sum}_q(0, 2) = 27 - 8 = 19.$$

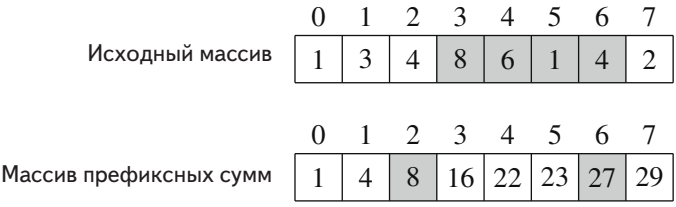


Рис. 9.2. Вычисление запроса о сумме по диапазону с помощью массива префиксных сумм

Многомерный случай. Эта идея обобщается и на многомерный случай. На рис. 9.3 показан двумерный массив префиксных сумм, который можно использовать для вычисления суммы по любому прямоугольному подмассиву за время $O(1)$. Каждая сумма в этом массиве соответствует подмассиву, начинающемуся в левом верхнем углу массива. Сумму по серому подмассиву можно вычислить по формуле

$$S(A) - S(B) - S(C) + S(D),$$

где $S(X)$ – сумма элементов прямоугольного подмассива, занимающего левый верхний угол исходного массива вплоть до позиции X.

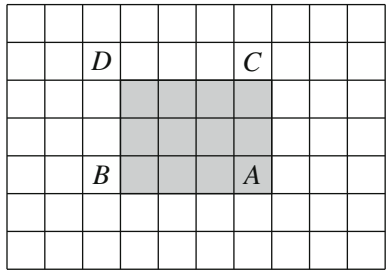


Рис. 9.3. Вычисление суммы по диапазону в двумерном случае

9.1.2. *Запросы о минимуме*

Обозначим $\texttt{min}_q(a, b)$ («запрос о минимуме по диапазону») минимальный элемент массива в диапазоне $[a, b]$. Мы рассмотрим метод, позволяющий обработать запрос о минимуме по диапазону за время $O(1)$ после предварительной обработки, занимающей время $O(n \log n)$. Этот метод, предложенный в работе Bender, Farach-Colton [3], часто называют *алгоритмом разреженной таблицы*.

Идея заключается в том, чтобы заранее вычислить $\texttt{min}_q(a, b)$ для случаев, когда $b - a + 1$ (длина диапазона) является степенью двойки. На рис. 9.4 показаны предвычисленные значения для массива из 8 элементов.

Рис. 9.4. Предварительная обработка для запросов о минимуме

Количество предвычисляемых значений имеет порядок $O(n \log n)$, поскольку существует $O(\log n)$ длин диапазонов, являющихся степенью двойки. Эти значения можно эффективно вычислить по рекуррентной формуле

$$\texttt{min}_q(a, b) = \min(\texttt{min}_q(a, a + w - 1),\ \texttt{min}_q(a + w, b)),$$

где $b - a + 1$ – степень двойки, а $w = (b - a + 1)/2$. Вычисление всех значений занимает время $O(n \log n)$.

После этого любое значение $\texttt{min}_q(a, b)$ можно вычислить за время $O(1)$, взяв минимум двух предвычисленных значений. Пусть k – наибольшая степень двойки, не превосходящая $b - a + 1$. Значение $\texttt{min}_q(a, b)$ вычисляется по формуле

$$\texttt{min}_q(a, b) = \min(\texttt{min}_q(a, a + k - 1),\ \texttt{min}_q(b - k + 1, b)).$$

В этой формуле диапазон $[a, b]$ представлен как объединение двух диапазонов длины k: $[a, a + k - 1]$ и $[b - k + 1, b]$.

Для примера рассмотрим диапазон [1, 6] на рис. 9.5. Его длина равна 6, а максимальная степень двойки, не превосходящая 6, – это 4. Таким образом, диапазон [1, 6] является объединением диапазонов [1, 4] и [3, 6]. Поскольку $\texttt{min}_q(1, 4) = 3$ и $\texttt{min}_q(3, 6) = 1$, можно заключить, что $\texttt{min}_q(1, 6) = 1$.

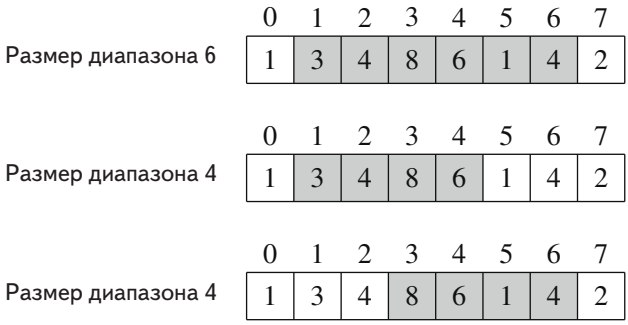


Рис. 9.5. Вычисление минимума по диапазону с помощью двух перекрывающихся диапазонов

Отметим, что с помощью довольно сложных методов можно обработать запрос о минимуме по диапазону за время $O(1)$ после предобработки, занимающей лишь время $O(n)$ (см., например, Fischer, Heun [12]), но эти методы выходят за рамки книги.

9.2. Древовидные структуры

В этом разделе мы представим две древовидные структуры, позволяющие обрабатывать запросы по диапазону и изменять значения элементов массива за логарифмическое время. Сначала обсудим двоичные индексные деревья, поддерживающие запросы о сумме, а затем – деревья отрезков, поддерживающие также запросы других видов.

9.2.1. *Двоичные индексные деревья*

Двоичное индексное дерево (или *дерево Фенвика*) [11] можно рассматривать как динамический вариант массива префиксных сумм. Оно предоставляет две операции с временной сложностью $O(\log n)$: обработка запроса о сумме по диапазону и изменение значения. Хотя в названии этой

структуры данных фигурирует слово «дерево», обычно она представляется в виде массива. При обсуждении двоичных индексных деревьев мы будем предполагать, что все массивы индексируются, начиная с 1, потому что это упрощает реализацию структуры.

Обозначим $p(k)$ наибольшую степень двойки, делящую k. Двоичное индексное дерево хранится в массиве `tree` таким образом, что

$$\texttt{tree}[k] = \texttt{sum}_q(k - p(k) + 1, k),$$

т. е. элемент в позиции k содержит сумму по заканчивающемуся в этой позиции диапазону длины $p(k)$. Например, поскольку $p(6) = 2$, `tree`[6] содержит значение $\texttt{sum}_q(5, 6)$. На рис. 9.6 показаны массив и соответствующее ему двоичное индексное дерево. На рис. 9.7 показано соответствие между значениями двоичного индексного дерева и диапазонами исходного массива.

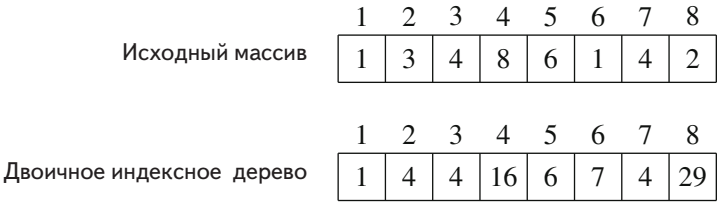


Рис. 9.6. Массив и его двоичное индексное дерево

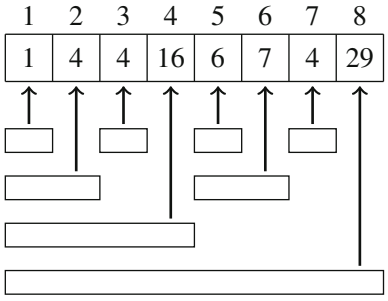


Рис. 9.7. Соответствие между двоичным индексным деревом и диапазонами

Зная двоичное индексное дерево, мы можем вычислить любое значение вида $\texttt{sum}_q(1, k)$ за время $O(\log n)$, поскольку диапазон $[1, k]$ всегда можно разбить на $O(\log n)$ поддиапазонов, суммы по которым хранятся в дереве. Например, чтобы вычислить $\texttt{sum}_q(1, 7)$, мы разобьем диапазон [1, 7] на три поддиапазона: [1, 4], [5, 6] и [7, 7] (рис. 9.8). Поскольку суммы по этим поддиапазонам уже имеются в дереве, то мы можем вычислить сумму по всему диапазону по формуле:

$$\texttt{sum}_q(1, 7) = \texttt{sum}_q(1, 4) + \texttt{sum}_q(5, 6) + \texttt{sum}_q(7, 7) = 16 + 7 + 4 = 27.$$

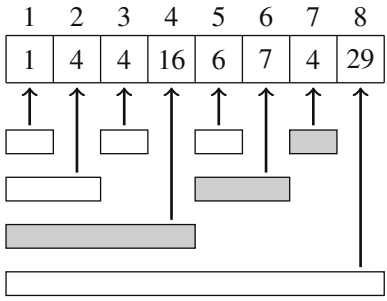


Рис. 9.8. Обработка запроса о сумме по диапазону с помощью двоичного индексного дерева

А чтобы вычислить значение $\mathrm{sum}_q(a, b)$, где $a > 1$, мы применим тот же прием, что в случае массивов префиксных сумм:

$$\mathrm{sum}_q(a, b) = \mathrm{sum}_q(1, b) - \mathrm{sum}_q(1, a - 1).$$

И $\mathrm{sum}_q(1, b)$, и $\mathrm{sum}_q(1, a - 1)$ можно вычислить за время $O(\log n)$, поэтому полная временная сложность равна $O(\log n)$.

После изменения любого элемента массива необходимо обновить двоичное индексное дерево. Например, если изменяется элемент 3, то следует обновить суммы по диапазонам [3, 3], [1, 4] и [1, 8] (рис. 9.9). Поскольку каждый элемент массива принадлежит $O(\log n)$ таким диапазонам, то достаточно обновить $O(\log n)$ элементов дерева.

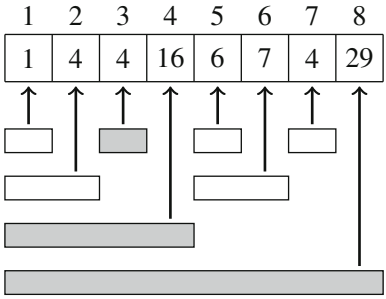


Рис. 9.9. Обновление значения в двоичном индексном дереве

Реализация. Операции двоичного индексного дерева эффективно реализуются с помощью поразрядных операций. Нам понадобится следующий факт: значение $p(k)$ можно вычислить по формуле:

$$p(k) = k \;\&\; -k,$$

которая выделяет самый младший единичный бит k.

Следующая функция вычисляет $\mathrm{sum}_q(1, k)$:

```
int sum(int k) {
   int s = 0;
```

```
    while (k >= 1) {
        s += tree[k];
        k -= k&-k;
    }
    return s;
}
```

А эта функция увеличивает значение k-го элемента массива на x (x может иметь любой знак):

```
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k&-k;
    }
}
```

Временная сложность обеих функций равна $O(\log n)$, потому что они обращаются к $O(\log n)$ элементам двоичного индексного дерева, а переход к следующей позиции занимает время $O(1)$.

9.2.2. *Деревья отрезков*

Дерево отрезков – это структура данных, предоставляющая две операции с временной сложностью $O(\log n)$: обработка запроса по диапазону и изменение элемента массива. Деревья отрезков поддерживают запросы о сумме, о минимуме и многие другие. Своими корнями деревья отрезков уходят в геометрические алгоритмы (см., например, Bentley, Wood [4]), а элегантная восходящая реализация, представленная в этом разделе, взята из книги Stańczyk [34].

Дерево отрезков – это двоичное дерево, в котором вершины нижнего уровня соответствуют элементам массива, а остальные несут информацию, необходимую для выполнения запросов по диапазону. При обсуждении деревьев отрезков мы будем предполагать, что размер массива – степень двойки, и использовать индексирование, начиная с нуля, потому что для такого массива строить дерево отрезков удобнее. Если размер массива не является степенью двойки, то всегда можно добавить в конец элементы.

Сначала обсудим деревья отрезков, поддерживающие запросы о сумме. На рис. 9.10 показаны массив и соответствующее ему дерево отрезков. Каждая внутренняя вершина дерева соответствует некоторому диапазону массива, размер которого равен степени двойки. Для дерева отрезков, поддерживающего запросы о сумме, значение в каждой внутренней вершине равно сумме элементов соответствующего диапазона, и его можно вычислить как сумму значений в левой и правой дочерних вершинах.

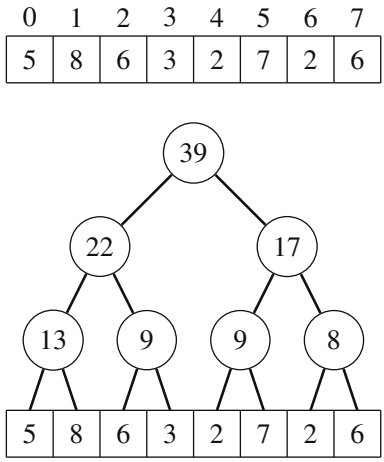


Рис. 9.10. Массив и соответствующее ему дерево отрезков для запросов о суммах

Оказывается, что любой диапазон [a, b] можно разбить на $O(\log n)$ поддиапазонов, таких, что суммы их элементов уже хранятся в дереве отрезков. На рис. 9.11 показан диапазон [2, 7] в исходном массиве и в дереве отрезков. В данном случае диапазону соответствуют две вершины и $\mathtt{sum}_q(2, 7) = 9 + 17 = 26$. Если при вычислении этой суммы используются вершины, расположенные как можно выше, то понадобится не более двух вершин на каждом уровне дерева. Поэтому общее число участвующих в вычислении вершин имеет порядок $O(\log n)$.

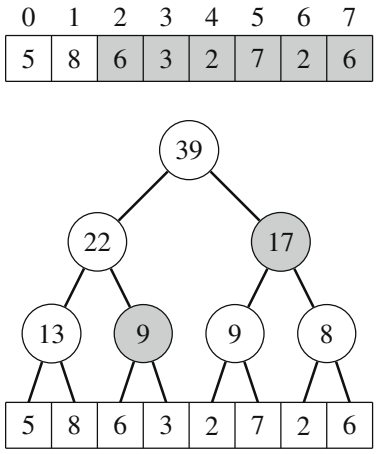


Рис. 9.11. Обработка запроса о сумме по диапазону с помощью дерева отрезков

После изменения некоторого элемента массива необходимо обновить все вершины, значения в которых зависят от измененного элемента. Для

этого следует пройти по пути от измененного элемента к корню дерева, обновляя все встретившиеся вершины. На рис. 9.12 показано, какие вершины обновляются после изменения пятого элемента. Путь от листа дерева к корню содержит $O(\log n)$ вершин, поэтому при любом изменении обновляется $O(\log n)$ вершин.

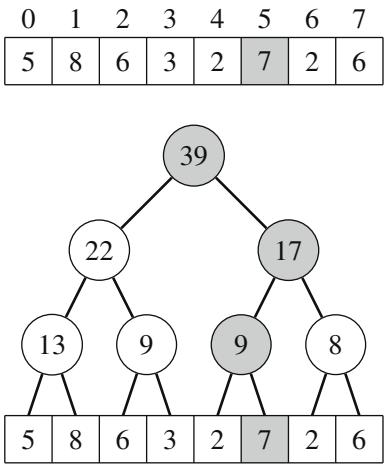


Рис. 9.12. Как изменение элемента массива отражается на дереве отрезков

Реализация. Хранить дерево отрезков удобно в массиве из $2n$ элементов, где n – размер исходного массива. Вершины хранятся сверху вниз: `tree`[1] – корень, `tree`[2] и `tree`[3] – его дочерние вершины и т. д. Наконец, элементы `tree`[n] по `tree`[$2n - 1$] соответствуют нижнему уровню дерева, на котором находятся элементы исходного массива. Отметим, что элемент `tree`[0] не используется.

На рис. 9.13 показано, как хранится дерево, изображенное на рис. 9.11. Отметим, что родителем элемента `tree`[k] является `tree`[$\lfloor k/2 \rfloor$], его левым потомком – `tree`[$2k$], а правым потомком – `tree`[$2k + 1$]. Кроме того, любая вершина (кроме корня) занимает четную позицию, если это левый потомок своего родителя, и нечетную – если правый.

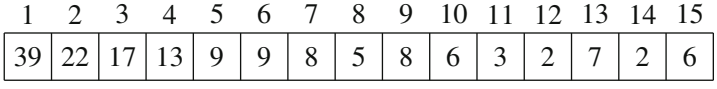


Рис. 9.13. Хранение дерева отрезков в виде массива

Следующая функция вычисляет значение $\mathrm{sum}_q(a, b)$:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
```

```
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Функция работает с диапазоном в массиве, представляющем дерево отрезков. Первоначально это диапазон $[a + n, b + n]$. На каждой итерации цикла диапазон поднимается на один уровень дерева выше, и значения в вершинах, не принадлежащих расположенному выше диапазону, прибавляются к сумме.

Следующая функция увеличивает k-й элемент массива на x:

```
void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k]+tree[2*k+1];
    }
}
```

Сначала обновляется значение на нижнем уровне, а затем – значения всех внутренних вершин дерева на пути до корня.

Обе функции работают за время $O(\log n)$, поскольку дерево отрезков для массива с n элементами содержит $O(\log n)$ уровней, а та и другая функции на каждой итерации поднимаются на один уровень выше.

Другие запросы. Деревья отрезков способны поддержать и другие запросы по диапазону, при условии что диапазон можно разбить на две половины, вычислить ответ для каждой части и эффективно объединить оба ответа. Примером может служить нахождение минимума и максимума, наибольшего общего делителя, а также поразрядные операции И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ.

Так, дерево отрезков на рис. 9.14 поддерживает запросы о минимуме. В нем каждая вершина содержит наименьшее значение в соответствующем диапазоне массива. В корне дерева находится минимум по всему массиву. Операции можно реализовать так же, как и выше, но вместо сумм вычисляются минимумы. Структура дерева отрезков также позволяет использовать для нахождения элементов массива метод двоичного поиска. Например, если дерево поддерживает запросы о минимуме, то можно найти позицию элемента с наименьшим значением за время $O(\log n)$. На рис. 9.15 показано, как элемент с наименьшим значением 1 ищется путем спуска по дереву, начиная с корня.

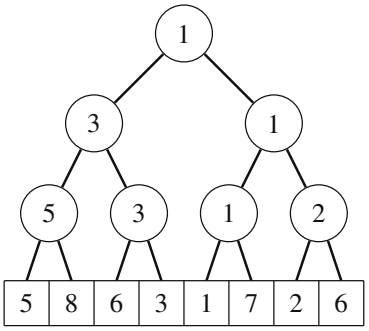


Рис. 9.14. Дерево отрезков для обработки запросов о минимуме по диапазону

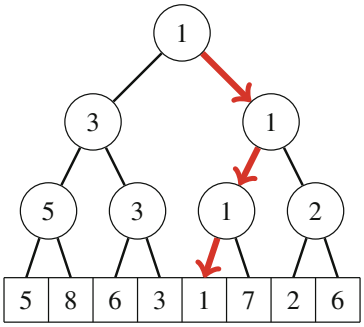


Рис. 9.15. Использование двоичного поиска для нахождения минимального элемента

9.2.3. *Дополнительные приемы*

Сжатие координат. В структурах данных, основанных на массивах, элементы индексируются последовательными целыми числами – и это можно считать ограничением. Трудности возникают, когда нужны большие индексы. Например, если требуется использовать индекс 10^9, то массив должен содержать 10^9 элементов, для чего нужно слишком много памяти.

Но если все индексы, которые могут понадобиться алгоритму, известны заранее, то это ограничение можно обойти, воспользовавшись *сжатием координат*. Идея заключается в том, чтобы заменить исходные числа последовательными целыми числами 0, 1, 2 и т. д. Для этого мы определим функцию сжатия координат c. Она сопоставляет каждому исходному индексу i сжатый индекс $c(i)$ таким образом, что если a и b – два числа и $a < b$, то $c(a) < c(b)$. Сжатые индексы удобно использовать для выполнения запросов по диапазону.

На рис. 9.16 приведен пример сжатия координат. Здесь реально используются только индексы 2, 5 и 7, а все остальные элементы массива равны нулю. В результате сжатия остаются индексы $c(2) = 0$, $c(5) = 1$, $c(7) = 2$, что позволяет создать сжатый массив всего из трех элементов.

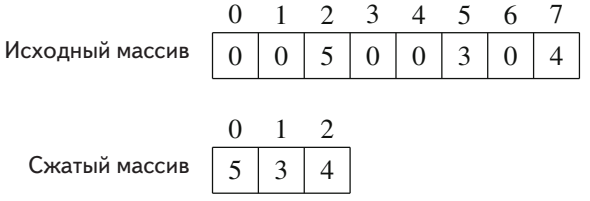


Рис. 9.16. Сжатие массива с помощью сжатия индексов

После сжатия координат мы можем, к примеру, построить дерево отрезков для сжатого массива и выполнять запросы. Нужно будет внести

только одну модификацию – сжимать координаты перед выполнением запроса: диапазону $[a, b]$ исходного массива соответствует диапазон $[c(a), c(b)]$ сжатого.

Обновление диапазона. До сих пор мы рассматривали структуры данных, которые поддерживают запросы по диапазону и изменение одного значения. Теперь обратимся к противоположной ситуации: требуется обновить диапазон и извлечь одно значение. Рассмотрим операцию увеличения всех элементов в диапазоне $[a, b]$ на x.

Оказывается, что представленные в этой главе структуры данных можно использовать и для такой цели. Для этого построим *разностный массив*, в котором будут храниться разности между соседними элементами исходного массива. Тогда исходный массив является массивом префиксных сумм для разностного. На рис. 9.17 показаны массив и соответствующий ему разностный массив. Так, значению 2 в шестой позиции исходного массива соответствует сумма 3 − 2 + 4 − 3 = 2 в разностном массиве.

	0	1	2	3	4	5	6	7
Исходный массив	3	3	1	1	1	5	2	2

	0	1	2	3	4	5	6	7
Разностный массив	3	0	−2	0	0	4	−3	0

Рис. 9.17. Массив и соответствующий ему разностный массив

У разностного массива имеется полезное свойство: чтобы изменить диапазон исходного массива, достаточно изменить всего два элемента разностного. Точнее, чтобы увеличить все элементы в диапазоне $[a, b]$ на x, достаточно увеличить элемент разностного массива в позиции a на x и уменьшить элемент в позиции $b + 1$ на x. Например, чтобы увеличить все элементы исходного массива с первого по четвертый на 3, мы прибавим 3 к первому элементу разностного массива и вычтем 3 из пятого элемента (рис. 9.18).

	0	1	2	3	4	5	6	7
Исходный массив	3	6	4	4	4	5	2	2

	0	1	2	3	4	5	6	7
Разностный массив	3	3	−2	0	0	1	−3	0

Рис. 9.18. Обновление диапазона массива с помощью разностного массива

Таким образом, мы изменяем только отдельные значения в разностном массиве и выполняем для него запросы о сумме, следовательно, можно

воспользоваться двоичным индексным деревом или деревом отрезков. Более трудная задача – придумать структуру данных, которая поддерживала бы *одновременно* запросы по диапазону и обновления диапазона. В разделе 15.2.1 мы увидим, что это можно сделать с помощью ленивого дерева отрезков.

Глава 10

Алгоритмы на деревьях

У деревьев имеются специальные свойства, благодаря которым алгоритмы, специализированные для деревьев, работают эффективнее, чем для графов общего вида. В этой главе приведена подборка таких алгоритмов.

В разделе 10.1 вводятся общие понятия и алгоритмы, относящиеся к деревьям. Центральная задача – найти диаметр дерева, т. е. максимальное расстояние между двумя его вершинами. Мы изучим два алгоритма ее решения с линейным временем.

Раздел 10.2 посвящен обработке запросов к деревьям. Мы узнаем, как с помощью обхода дерева обрабатывать различные запросы, касающиеся поддеревьев и путей. А затем обсудим методы нахождения наименьшего общего предка и пакетный алгоритм, основанный на объединении структур данных.

В разделе 10.3 представлено два метода обработки деревьев: центроидная декомпозиция и heavy-light декомпозиция.

10.1. Базовые понятия

Дерево – это связный ациклический граф, содержащий n вершин и $n - 1$ ребер. Удаление любого ребра разбивает дерево на две компоненты связности, а добавление любого ребра создает цикл. Между любыми двумя вершинами дерева существует единственный путь. *Листьями* дерева называются вершины, имеющие только одного соседа.

Рассмотрим дерево, изображенное на рис. 10.1. В нем 8 вершин и 7 ребер, листьями являются вершины 3, 5, 7 и 8.

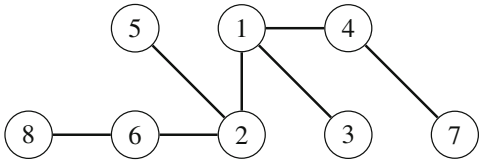


Рис. 10.1. Дерево, имеющее 8 вершин и 7 ребер

В *корневом* дереве одна из вершин считается *корнем* дерева, а все остальные располагаются под ней. Вершины, расположенные непосредственно

под некоторой вершиной, называются ее *дочерними вершинами*, а вершина, расположенная непосредственно над некоторой вершиной, – ее *родителем*. У каждой вершины, кроме корня, имеется ровно один родитель. У корня родителя нет. Корневое дерево имеет рекурсивную структуру: каждая вершина играет роль корня *поддерева*, содержащего эту вершину и все вершины, принадлежащие поддеревьям ее дочерних вершин.

На рис. 10.2 показано корневое дерево с корнем в вершине 1. Дочерними для вершины 2 являются вершины 5 и 6, а родителем вершины 2 – вершина 1. Поддерево вершины 2 состоит из вершин 2, 5, 6 и 8.

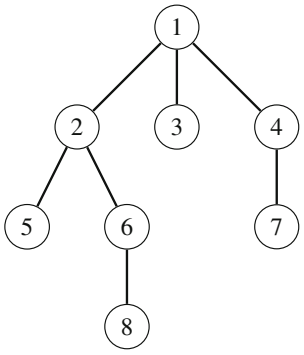


Рис. 10.2. Корневое дерево, в котором вершина 1 является корнем

10.1.1. *Обход дерева*

Для обхода вершин дерева применимы общие алгоритмы обхода графов. Однако обход дерева реализовать проще, чем в общем случае, поскольку в нем нет циклов и невозможно попасть в вершину с нескольких направлений.

Чаще всего для обхода дерева применяют поиск в глубину, начиная с произвольной вершины. Можно использовать следующую рекурсивную функцию:

```
void dfs(int s, int e) {
    // обработать вершину s
    for (auto u : adj[s]) {
        if (u != e) dfs(u, s);
    }
}
```

Функции передаются два параметра: текущая вершина *s* и предыдущая вершина *e*. Параметр *e* гарантирует, что далее будут просматриваться только еще не посещенные вершины.

При таком вызове поиск начинается с вершины *x*:

```
dfs(x, 0);
```

При первом вызове функции $e = 0$, поскольку предыдущей вершины еще нет, и, следовательно, разрешено двигаться по дереву в любом направлении.

Динамическое программирование можно использовать для вычисления некоторой информации в процессе обхода дерева. Показанный

ниже код для каждой вершины *s* вычисляет count[*s*] – количество вершин в ее поддереве. Поддерево содержит саму вершину и все вершины, принадлежащие поддеревьям ее дочерних вершин, поэтому количество вершин можно вычислить рекурсивно:

```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

Обход двоичного дерева. В двоичном дереве каждая вершина имеет левое и правое поддеревья (возможно, пустые). Существует три распространенных способа обхода двоичных деревьев:

- *прямой порядок*: сначала обрабатывается корневая вершина, после чего обходится левое, а затем правое поддерево;
- *внутренний порядок*: сначала обходится левое поддерево, затем обрабатывается корневая вершина, потом обходится правое поддерево;
- *обратный порядок*: сначала обходится левое поддерево, затем правое поддерево, потом обрабатывается корневая вершина.

Для дерева на рис. 10.3 прямым порядком будет [1, 2, 4, 5, 6, 3, 7], внутренним – [4, 2, 6, 5, 1, 3, 7], а обратным – [4, 6, 5, 2, 7, 3, 1].

Если известны последовательности посещения вершин в прямом и внутреннем порядке, то можно реконструировать его структуру. Например, единственное дерево с прямым порядком вершин [1, 2, 4, 5, 6, 3, 7] и внутренним порядком [4, 2, 6, 5, 1, 3, 7] показано на рис. 10.3. Обратный и внутренний порядок также однозначно определяют структуру дерева. Но деревьев с одними и теми же прямым и обратным порядками вершин может быть несколько.

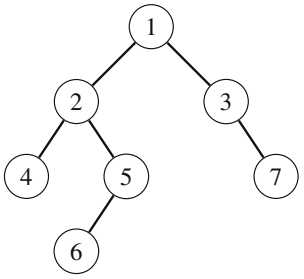


Рис. 10.3. Двоичное дерево

10.1.2. *Вычисление диаметра*

Диаметром дерева называется максимальная длина пути между двумя его вершинами. На рис. 10.4 изображено дерево диаметра 4, которому соответствует путь длины 4 между вершинами 6 и 7. Отметим, что в этом дереве есть и другой путь длины 4 между вершинами 5 и 7.

Ниже мы обсудим два алгоритма вычисления диаметра дерева с временной сложностью $O(n)$. Первый основан на динамическом программировании, во втором используется поиск в глубину.

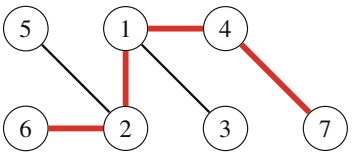


Рис. 10.4. Дерево диаметра 4

Первый алгоритм. Общий подход к задачам о деревьях заключается в том, чтобы сначала произвольным образом выбрать корень, а затем решить задачу отдельно для каждого поддерева. На этой идее основан первый алгоритм вычисления диаметра.

Сделаем важное наблюдение: на каждом пути в корневом дереве имеется *высшая точка*. Таким образом, для каждой вершины x мы можем вычислить длину самого длинного пути с высшей точкой x. Один из таких путей соответствует диаметру дерева. На рис. 10.5 высшей точкой пути, соответствующей диаметру, является вершина 1.

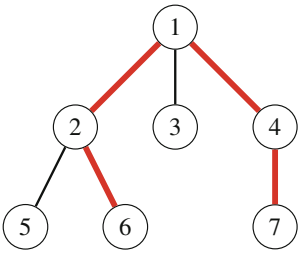


Рис. 10.5. Вершина 1 – высшая точка на пути, реализующем диаметр

Для каждой вершины x мы вычислим два значения:

- `toLeaf`(x): максимальная длина пути от x до листа;
- `maxLength`(x): максимальная длина пути с высшей точкой x.

На рис. 10.5 `toLeaf`(1) = 2, поскольку существует путь $1 \to 2 \to 6$, и `maxLength`(1) = 4, поскольку существует путь $6 \to 2 \to 1 \to 4 \to 7$. В данном случае `maxLength`(1) равно диаметру.

Чтобы вычислить эти значения для всех вершин за время $O(n)$, можно воспользоваться динамическим программированием. Сначала для вычисления `toLeaf`(x) мы переберем дочерние вершины x, выберем из них вершину c с максимальным значением `toLeaf`(c) и прибавим к этому значению 1. Затем для вычисления `maxLength`(x) мы выберем две различные дочерние вершины a и b – такие, что сумма `toLeaf`(a) + `toLeaf`(b) максимальна, и прибавим к этой сумме 2. (Случаи, когда у x меньше двух дочерних вершин, особые, но они легко разбираются.)

Второй алгоритм. Еще один эффективный способ вычислить диаметр дерева основан на поиске в глубину. Сначала выберем в дереве произвольную вершину a и найдем вершину b, самую далекую от a. Затем найдем вершину c, самую далекую от b. Диаметр дерева равен расстоянию между b и c.

На рис. 10.6 показан один из способов выбора вершин a, b и c при вычислении диаметра дерева.

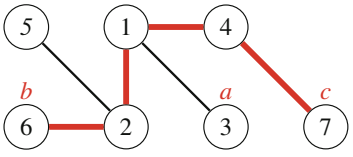


Рис. 10.6. Вершины a, b и c в процессе вычисления диаметра

Это элегантный метод, но почему он дает правильный результат? Полезно нарисовать дерево так, чтобы путь, реализующий диаметр, был расположен горизонтально, а остальные вершины «свисали» с него (рис. 10.7). Буквой *x* обозначено место, в котором путь из вершины *a* соединяется с путем, реализующим диаметр. Самая далекая от *a* вершина – это *b*, *c* или еще какая-то вершина, отстоящая от *x* на такое же или большее расстояние. Таким образом, эту вершину всегда можно выбрать в качестве конечной точки пути, реализующего диаметр.

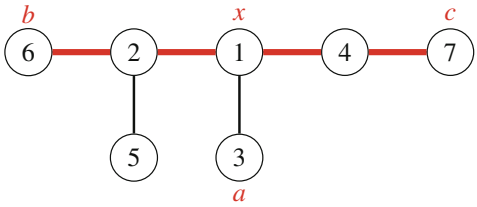
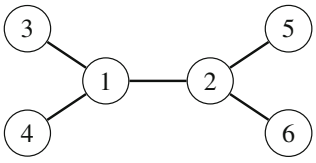


Рис. 10.7. Почему этот алгоритм работает

10.1.3. *Все максимальные пути*

Наша следующая задача – для каждой вершины дерева *x* вычислить значение `maxLength`(*x*): максимальную длину пути, начинающегося в *x*. На рис. 10.8 показаны дерево и значения `maxLength` для него. Это можно рассматривать как обобщение задачи о диаметре дерева, поскольку наибольшая из этих длин равна диаметру. Эту задачу также можно решить за время $O(n)$.



$$\texttt{maxLength}(1) = 2$$
$$\texttt{maxLength}(2) = 2$$
$$\texttt{maxLength}(3) = 3$$
$$\texttt{maxLength}(4) = 3$$
$$\texttt{maxLength}(5) = 3$$
$$\texttt{maxLength}(6) = 3$$

Рис. 10.8. Вычисление длин максимальных путей

И снова выберем какую-то вершину в качестве корня. Первая часть задачи – для каждой вершины *x* вычислить максимальную длину пути, проходящего *вниз* через дочернюю вершину *x*. Так, самый длинный путь из вершины 1 проходит через ее дочернюю вершину 2 (рис. 10.9). Эту часть легко решить за время $O(n)$, воспользовавшись динамическим программированием, как и выше.

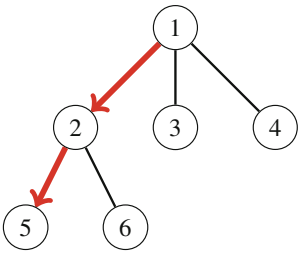


Рис. 10.9. Самый длинный путь, начинающийся в вершине 1

Вторая часть задачи – для каждой вершины *x* вычислить путь, проходящий *вверх*, через ее ро-

дителя p. Например, самый длинный путь из вершины 3 проходит через ее родителя 1 (рис. 10.10). На первый взгляд, достаточно сначала перейти в p, а затем выбрать самый длинный путь (вверх или вниз) из p. Однако это *не всегда* правильно, потому что такой путь может проходить через x (рис. 10.11). И тем не менее вторую часть задачи можно решить за время $O(n)$, если хранить длины *двух* максимальных путей для каждой вершины x:

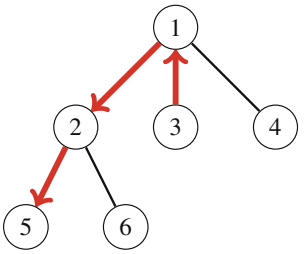


Рис. 10.10. Самый длинный путь, начинающийся в вершине 3 и проходящий через ее родителя

- $\texttt{maxLength}_1(x)$: длина максимального пути из x в листовую вершину;
- $\texttt{maxLength}_2(x)$: длина максимального пути из x в листовую вершину, следующего в другом направлении, нежели первый путь.

Так, на рис. 10.11 $\texttt{maxLength}_1(1) = 2$ для пути $1 \to 2 \to 5$ и $\texttt{maxLength}_2(1) = 1$ для пути $1 \to 3$.

Наконец, чтобы определить длину максимального пути из вершины x, проходящего вверх через ее родителя p, рассмотрим два случая: если путь, соответствующий $\texttt{maxLength}_1(p)$, проходит через x, то длина максимального пути равна $\texttt{maxLength}_2(p) + 1$, в противном случае длина максимального пути равна $\texttt{maxLength}_1(p) + 1$.

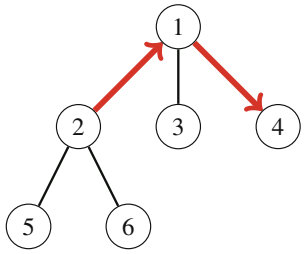


Рис. 10.11. В этом случае следует выбрать второй по длине путь из родителя

10.2. Запросы к деревьям

В этом разделе мы будем рассматривать *запросы* к корневым деревьям. Обычно они относятся к поддеревьям и путям в дереве и могут быть обработаны за постоянное или логарифмическое время.

10.2.1. *Нахождение предков*

k-м *предком* вершины x корневого дерева называется вершина, в которую мы попадем, поднявшись на k уровней вверх от x. Обозначим $\texttt{ancestor}(x, k)$ k-го предка вершины x (или 0, если такого предка не существует). На рис. 10.12 $\texttt{ancestor}(2, 1) = 1$, а $\texttt{ancestor}(8, 2) = 4$.

Вычислить $\texttt{ancestor}(x, k)$ проще всего, последовательно выполнив k шагов по дереву. Но временная сложность этого метода равна $O(k)$, что может оказаться медленно, поскольку в дереве с n вершинами может встретиться путь длины n.

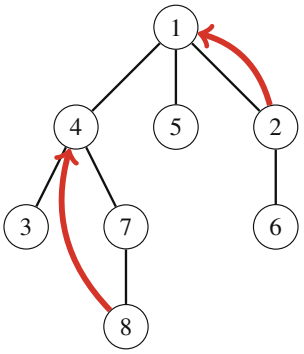


Рис. 10.12. Нахождение предков вершин

К счастью, значение `ancestor`(x, k) для любой вершины x можно вычислить за время $O(\log k)$ после предобработки. Как и в разделе 7.5.1, идея заключается в том, чтобы предварительно вычислить все значения `ancestor`(x, k) для случаев, когда k – степень двойки. Для дерева, изображенного на рис. 10.12, получаем:

x	1 2 3 4 5 6 7 8
`ancestor`(x, 1)	0 1 4 1 1 2 4 7
`ancestor`(x, 2)	0 0 1 0 0 1 1 4
`ancestor`(x, 4)	0 0 0 0 0 0 0 0
...	

Поскольку у любой вершины заведомо меньше n предков, достаточно вычислить $O(\log n)$ значений для каждой вершины, так что предобработка займет время $O(n \log n)$. После этого любое значение `ancestor`(x, k) можно будет вычислить за время $O(\log k)$, представив k в виде суммы степеней двойки.

10.2.2. *Поддеревья и пути*

Массив обхода дерева содержит вершины корневого дерева в порядке их посещения в процессе поиска в глубину из корня. На рис. 10.13 показаны дерево и массив обхода для него.

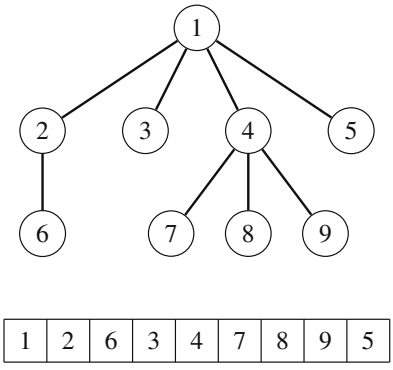
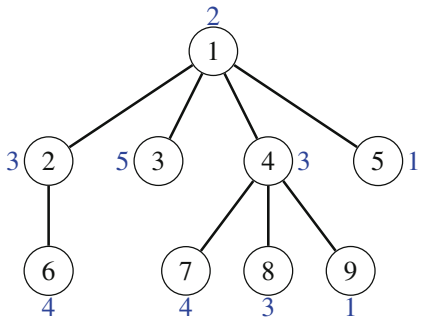


Рис. 10.13. Дерево и массив его обхода

У массива обхода дерева есть важное свойство: каждому поддереву соответствует его подмассив, первым элементом которого является корень этого поддерева. На рис. 10.14 показан подмассив, соответствующий поддереву вершины 4.

Рис. 10.14. Поддерево вершины 4 в массиве обхода дерева

Запросы о поддеревьях. Предположим, что каждой вершине дерева сопоставлено некоторое значение, и наша задача – обработать запросы двух типов: изменить значение в вершине и вычислить сумму значений в поддереве вершины. Для решения этой задачи мы построим массив обхода дерева, содержащий три значения для каждой вершины: ее идентификатор, размер поддерева и значение в вершине. На рис. 10.15 показаны дерево и соответствующий ему массив.



ИД вершины	1	2	6	3	4	7	8	9	5
Размер поддерева	9	2	1	1	4	1	1	1	1
Значение в вершине	2	3	4	5	3	4	3	1	1

Рис. 10.15. Массив обхода дерева для вычисления суммы по поддереву

Пользуясь этим массивом, мы можем вычислить сумму значений в любом поддереве. Для этого нужно сначала определить размер поддерева, а затем просуммировать значения в соответствующих вершинах. На рис. 10.16 показано, к каким значениям мы обращаемся при вычислении суммы значений в поддереве вершины 4. Из последней строки массива видно, что сумма значений равна 3 + 4 + 3 + 1 = 11.

ИД вершины	1	2	6	3	**4**	7	8	9	5
Размер поддерева	9	2	1	1	**4**	1	1	1	1
Значение в вершине	2	3	4	5	**3**	**4**	**3**	**1**	1

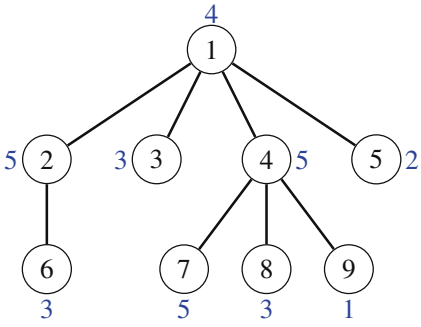
Рис. 10.16. Вычисление суммы значений в поддереве вершины 4

Для эффективного ответа на такие запросы достаточно сохранить последнюю строку массива в двоичном индексном дереве или в дереве отрезков. После этого мы сможем изменять значение и вычислять сумму значений за время $O(\log n)$.

Запросы о путях. С помощью массива обхода дерева мы можем эффективно вычислять суммы значений на путях из корня в любую вершину

дерева. Рассмотрим, например, обработку запросов двух типов: изменение значения в вершине и вычисление суммы значений на пути из корня в вершину.

Для решения этой задачи мы построим массив обхода дерева, в котором для каждой вершины будут храниться ее идентификатор, размер поддерева и сумма значений на пути от корня к этой вершине (рис. 10.17). Если значение в некоторой вершине увеличивается на x, то суммы для всех вершин, принадлежащих ее поддереву, тоже увеличиваются на x. На рис. 10.18 показано, как будет выглядеть массив после увеличения значения в вершине 4 на 1.



ИД вершины	1	2	6	3	4	7	8	9	5
Размер поддерева	9	2	1	1	4	1	1	1	1
Сумма по пути	4	9	12	7	9	14	12	10	6

Рис. 10.17. Массив обхода дерева для вычисления сумм по путям

ИД вершины	1	2	6	3	4	7	8	9	5
Размер поддерева	9	2	1	1	4	1	1	1	1
Сумма по пути	4	9	12	7	10	15	13	11	6

Рис. 10.18. Увеличение значения в вершине 4 на 1

Чтобы поддержать обе операции, мы должны уметь увеличивать все значения в диапазоне и получать одно значение. Это можно сделать за время $O(\log n)$ с помощью двоичного индексного дерева или дерева отрезков (см. раздел 9.2.3).

10.2.3. *Наименьшие общие предки*

Наименьшим общим предком двух вершин корневого дерева называется самая низко расположенная вершина, поддерево которой содержит обе эти вершины. Так, на рис. 10.19 наименьшим общим предком вершин 5 и 8 является вершина 2.

Эффективный поиск наименьшего общего предка двух вершин – часто встречающаяся задача. Мы обсудим два способа ее решения.

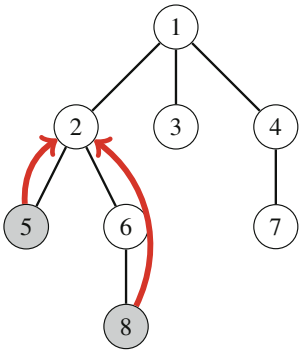


Рис. 10.19. Наименьшим общим предком вершин 5 и 8 является вершина 2

Первый метод. Поскольку мы умеем эффективно находить k-й предок любой вершины дерева, то можем воспользоваться этим для разбиения задачи на две части. Нам понадобятся два указателя, первоначально направленных на вершины, для которых мы ищем наименьший общий предок.

Сначала проверим, что вершины, на которые направлены указатели, расположены на одном уровне дерева. Если это не так, сдвинем один из указателей вверх. После этого определим, какое минимальное число сдвигов обоих указателей вверх необходимо, чтобы они оказались направлены на одну и ту же вершину. Эта вершина и будет наименьшим общим предком. Поскольку обе части алгоритма можно выполнить за время $O(\log n)$, если воспользоваться предварительно вычисленной информацией, то наименьший общий предок любых двух вершин можно найти за время $O(\log n)$.

На рис. 10.20 показано, как найти наименьший общий предок вершин 5 и 8 в рассматриваемом примере. Сначала мы сдвигаем второй указатель на один уровень вверх, так что вершина 6, на которую он указывает, находится на том же уровне, что и вершина 5. Затем после сдвига обоих указателей на один шаг вверх они указывают на вершину 2, которая и является наименьшим общим предком.

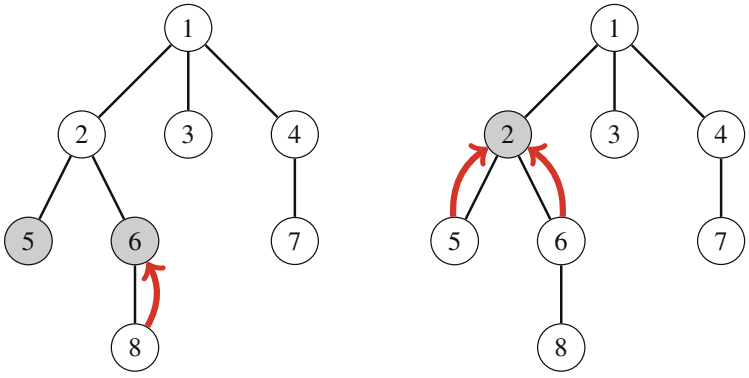


Рис. 10.20. Два шага для нахождения наименьшего общего предка вершин 5 и 8

Второй метод. Другой способ решения этой задачи был предложен в работе Bender, Farach-Colton [3]. Он основан на расширенном массиве обхода дерева, который всегда называют *эйлеровым обходом дерева*. Для постро-

ения массива мы обходим вершины дерева, применяя поиск в глубину, и добавляем вершину в массив при *каждом* ее прохождении (а не только при первом посещении). Поэтому вершина, имеющая k дочерних вершин, встретится в массиве $k + 1$ раз, а всего в нем будет $2n - 1$ элементов. В каждом элементе массива хранятся два значения: идентификатор вершины и ее глубина в дереве. На рис. 10.21 показан массив, соответствующий дереву из рассматриваемого примера.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ИД вершины	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
Глубина	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Рис. 10.21. Эйлеров обход дерева
для обработки запросов о наименьшем общем предке

Теперь можно найти наименьший общий предок вершин a и b, отыскав вершину с *минимальной* глубиной, находящуюся между вершинами a и b в массиве. На рис. 10.22 показано, как ищется наименьший общий предок вершин 5 и 8. Вершина с минимальной глубиной, находящаяся между ними, – это вершина 2 глубины 2, поэтому она и является наименьшим общим предком.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ИД вершины	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
Глубина	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1
				↑											

Рис. 10.22. Нахождение наименьшего общего предка вершин 5 и 8

Отметим, что поскольку одна вершина может встречаться в массиве несколько раз, возможно несколько способов выбрать промежуток между вершинами a и b. Но при любом выборе наименьший общий предок определяется правильно.

При использовании этой техники для нахождения наименьшего общего предка двух вершин достаточно обработать запрос о минимуме по диапазону. Стандартный способ обработки таких запросов с помощью дерева отрезков имеет временную сложность $O(\log n)$. Но, поскольку массив статический, мы можем отвечать на них и за время $O(1)$ после предобработки, занимающей время $O(n \log n)$.

Вычисление расстояний. Наконец, рассмотрим задачу о вычислении расстояния между вершинами a и b (т. е. длины пути между ними). Оказывается, что ее можно свести к задаче о нахождении наименьшего об-

щего предка. Сначала произвольным образом выберем корень дерева. Тогда расстояние между вершинами a и b можно будет вычислить по формуле

$$\texttt{depth}(a) + \texttt{depth}(b) - 2 \cdot \texttt{depth}(c),$$

где c – наименьший общий предок a и b.

Чтобы вычислить расстояние между вершинами 5 и 8 на рис. 10.23, мы сначала находим их наименьший общий предок – вершину 2. Затем, поскольку глубины вершин depth(5) = 3, depth(8) = 4 и depth(2) = 2, то расстояние между вершинами 5 и 8 равно 3 + 4 – 2 · 2 = 3.

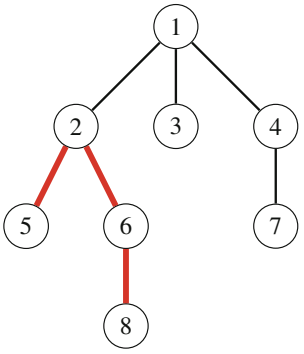


Рис. 10.23. Вычисление расстояния между вершинами 5 и 8

10.2.4. *Объединение структур данных*

До сих пор мы рассматривали оперативные алгоритмы обработки запросов к дереву. Они позволяют обрабатывать запросы один за другим, так что ответ на предыдущий запрос дается до получения следующего. Но существует много задач, когда свойство оперативности необязательно и можно воспользоваться *пакетными* алгоритмами. Такому алгоритму передается весь набор запросов, и он вправе обрабатывать их в любом порядке. Зачастую спроектировать пакетный алгоритм проще, чем оперативный.

Один из способов проектирования пакетного алгоритма состоит в том, чтобы выполнить обход дерева в глубину и создать в вершинах некоторые структуры данных. В каждой вершине s создается структура данных d[s], основанная на структурах данных в дочерних вершинах s. Эта структура данных позволяет обрабатывать все запросы, относящиеся к s.

В качестве примера рассмотрим следующую задачу: дано корневое дерево, с каждой вершиной которого ассоциировано некоторое значение. Требуется отвечать на запросы о том, сколько вершин со значением x имеется в поддереве вершины s. На рис. 10.24 поддерево вершины 4 содержит две вершины со значением 3.

Для ответа на такие запросы можно воспользоваться структурой отображения (значения на число вершин с таким значением). На рис. 10.25 показаны отображения для вершины 4 и ее дочерних вершин. Создав такую структуру данных для каждой вершины, мы легко сможем обработать все запросы, поскольку на запрос, относящийся к определенной вершине, можно

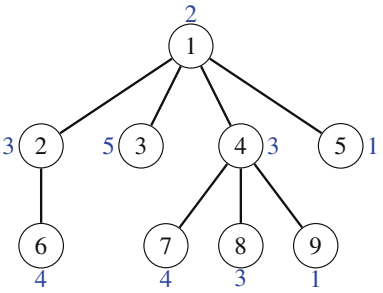


Рис. 10.24. Поддерево вершины 4 содержит две вершины со значением 3

отвечать сразу после создания структуры данных для нее. Но создавать все структуры с нуля было бы слишком медленно. Вместо этого мы создадим в каждой вершине s начальную структуру данных d[s], содержащую только значение, ассоциированное с s. После этого мы пробежимся по дочерним вершинам s и *объединим* d[s] и все структуры d[u], где u – дочерняя вершина s. Например, в дереве из нашего примера отображение для вершины 4 создано путем объединения отображений на рис. 10.26, где первое отображение – начальная структура данных для вершины 4, а остальные три соответствуют вершинам 7, 8 и 9.

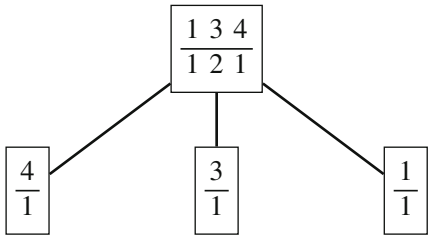


Рис. 10.25. Обработка запросов с помощью отображений

Рис. 10.26. Объединение отображений в вершине

Объединение в вершине s выполняется следующим образом: перебираем все дочерние вершины s и в каждой вершине u объединяем d[s] с d[u]. Мы всегда копируем содержимое d[u] в d[s]. Но предварительно содержимое d[s] и d[u] *обменивается*, если d[s] меньше d[u]. В результате каждое значение копируется всего $O(\log n)$ раз в процессе обхода дерева, что гарантирует эффективность алгоритма.

Для эффективного обмена структур данных a и b можно использовать такой код:

```
swap(a,b);
```

Гарантируется, что он работает за постоянное время, если a и b – структуры данных из стандартной библиотеки C++.

10.3. Более сложные приемы

В этом разделе мы рассмотрим еще два метода работы с деревьями. Центроидная декомпозиция заключается в разбиении дерева на меньшие поддеревья и их рекурсивной обработке. При разновесной декомпозиции (heavy-light decomposition) дерево представляется в виде множества специальных путей, что позволяет эффективно обрабатывать запросы о путях.

10.3.1. *Центроидная декомпозиция*

Центроидом дерева с n вершинами называется такая вершина, удаление которой разбивает дерево на поддеревья, каждое из которых содержит не более $\lfloor n/2 \rfloor$ вершин. В каждом дереве существует центроид; чтобы его найти, нужно выбрать произвольный корень и всегда переходить в поддерево с максимальным числом вершин до тех пор, пока текущая вершина не окажется центроидом.

В методе центроидной декомпозиции мы сначала находим центроид дерева и обрабатываем все проходящие через него пути. После этого центроид удаляется, а оставшиеся поддеревья обрабатываются рекурсивно. Поскольку удаление центроида оставляет поддеревья, размер которых не превосходит половины размера исходного дерева, временная сложность такого алгоритма равна $O(n \log n)$, при условии что время обработки каждого поддерева линейно.

На рис. 10.27 показан первый шаг алгоритма центроидной декомпозиции. В этом дереве вершина 5 является единственным центроидом, поэтому сначала обрабатываются пути, проходящие через вершину 5. Затем эта вершина удаляется из дерева, и рекурсивно обрабатываются три поддерева: {1, 2}, {3, 4} и {6, 7, 8}.

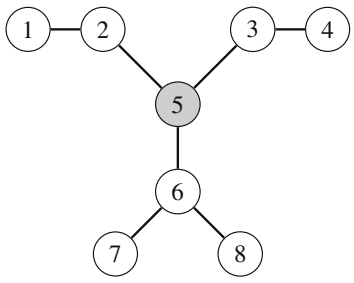


Рис. 10.27. Центроидная декомпозиция

С помощью центроидной декомпозиции мы, например, можем эффективно вычислить количество путей длины x в дереве. Для этого мы сначала находим центроид и вычисляем количество путей, проходящих через него. Это можно сделать за линейное время. Затем удаляем центроид и рекурсивно обрабатываем меньшие деревья. Временная сложность алгоритма равна $O(n \log n)$.

10.3.2. *Heavy-light декомпозиция*

При *heavy-light декомпозиции* (heavy-light decomposition)[1] множество вершин дерева разбивается на множество *тяжелых* (heavy) путей. Тяжелые пути создаются таким образом, что на пути между любыми двумя вершинами находится $O(\log n)$ подпутей, являющихся тяжелыми путями. Благодаря этой технике мы можем манипулировать вершинами на путях между вершинами дерева почти как элементами массива, добавляя лишь коэффициент порядка $O(\log n)$.

Для построения тяжелых путей мы сначала произвольным образом выбираем корень дерева. Первый тяжелый путь начинается в корне и всегда

[1] В работе Sleator, Tarjan [33] эта идея введена в контексте динамических деревьев с операциями link и cut.

идет в вершину, для которой размер поддерева максимален. Затем рекурсивно обрабатываются остальные поддеревья. На рис. 10.28 показано четыре тяжелых пути: 1–2–6–8, 3, 4–7 и 5 (два из них состоят всего из одной вершины).

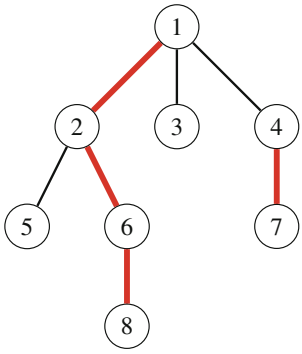


Рис. 10.28. Разновесная декомпозиция

Теперь рассмотрим какой-нибудь путь между двумя вершинами дерева. Поскольку при создании тяжелых путей всегда выбиралось поддерево максимального размера, то мы заведомо сможем разбить путь на $O(\log n)$ подпутей, каждый из которых является подпутем единственного тяжелого пути. На рис. 10.28 путь между вершинами 7 и 8 можно разбить на два тяжелых подпути: 7–4, затем 1–2–6–8.

Достоинство heavy-light декомпозиции состоит в том, что каждый тяжелый путь можно рассматривать как массив вершин. Например, с каждым тяжелым путем можно сопоставить дерево отрезков и поддержать сложные запросы о путях, например вычисление максимального значения вершины на пути или увеличение значения, ассоциированного с каждой вершиной на пути. Такие запросы можно обработать за время $O(\log^2 n)$[2], поскольку каждый путь состоит из $O(\log n)$ тяжелых путей, а каждый тяжелый путь можно обработать за время $O(\log n)$.

Хотя с помощью heavy-light декомпозиции можно решить много задач, следует помнить, что часто существует другое решение, реализовать которое проще. В частности, вместо heavy-light декомпозиции нередко можно использовать методы, описанные в разделе 10.2.2.

[2] $\log^k n$ – то же самое, что $(\log n)^k$.

Глава 11

Математика

В этой главе речь пойдет о математических методах, регулярно встречающихся в олимпиадном программировании. Мы обсудим теоретические результаты и научимся применять их в алгоритмах.

В разделе 11.1 обсуждаются вопросы теории чисел. Мы узнаем об алгоритмах нахождения простых множителей числа, об арифметике по модулю и об эффективных способах решения уравнений в целых числах.

В разделе 11.2 изучаются различные подходы к решению комбинаторных задач: как эффективно подсчитать допустимые комбинации объектов. Среди прочего будут рассмотрены биномиальные коэффициенты, числа Каталана и формула включений-исключений.

В разделе 11.3 показано, как использовать матрицы при программировании алгоритмов. Например, мы узнаем, как ускорить работу алгоритма динамического программирования за счет эффективного возведения матрицы в степень.

В разделе 11.4 сначала обсуждаются простые методы вычисления вероятностей событий и понятие марковской цепи. После этого рассматриваются примеры рандомизированных алгоритмов.

Раздел 11.5 посвящен теории игр. Сначала мы изучим оптимальную стратегию простой игры в палочки, основанную на теории игры ним, а затем обобщим эту стратегию на широкий круг других игр.

11.1. Теория чисел

Теория чисел – это область математики, в которой изучаются целые числа. В этом разделе мы обсудим избранные вопросы и алгоритмы теории чисел, в т. ч. нахождение простых чисел, разложение на простые множители и решение уравнений в целых числах.

11.1.1. *Простые числа и разложение на простые множители*

Целое число a называется *множителем*, или *делителем* целого числа b, если a делит b. Если a – множитель b, то мы пишем $a \mid b$, иначе $a \nmid b$. Например, множителями числа 24 являются 1, 2, 3, 4, 6, 8, 12 и 24.

Целое число $n > 1$ называется *простым*, если его единственными положительными множителями являются 1 и n. Например, числа 7, 19 и 41 простые, а 35 – не простое (*составное*), потому что $5 \cdot 7 = 35$. Для каждого целого числа $n > 1$ существует единственное *разложение на простые множители*:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

где $p_1, p_2, \ldots, p_k$ – различные простые числа, а $\alpha_1, \alpha_2, \ldots, \alpha_k$ – положительные целые числа. Например, число 84 разлагается на простые множители следующим образом:

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

Обозначим $\tau(n)$ количество простых множителей целого числа n. Например, $\tau(12) = 6$, поскольку множителями 12 являются 1, 2, 3, 4, 6 и 12. Вычислить $\tau(n)$ можно по формуле

$$\tau(n) = \prod_{i=1}^{k} (\alpha_i + 1),$$

потому что для каждого простого p_i существует $\alpha_i + 1$ способов выбрать, сколько раз оно входит в множитель. Например, $12 = 2^2 \cdot 3$, поэтому $\tau(12) = 3 \cdot 2 = 6$.

Обозначим $\sigma(n)$ сумму всех множителей числа n. Например, $\sigma(12) = 28$, т. к. $1 + 2 + 3 + 4 + 6 + 12 = 28$. Значение $\sigma(n)$ вычисляется по формуле

$$\sigma(n) = \prod_{i=1}^{k} (1 + p_i + \cdots + p_i^{\alpha_i}) = \prod_{i=1}^{k} \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

следующей из формулы суммы геометрической прогрессии. Например, $\sigma(12) = (2^3 - 1)/(2 - 1) \cdot (3^2 - 1)/(3 - 1) = 28$.

Основные алгоритмы. Если целое число n не простое, то его можно представить в виде произведения $a \cdot b$, где $a \leqslant \sqrt{n}$ или $b \leqslant \sqrt{n}$, поэтому среди его множителей обязательно имеется число от 2 до $\lfloor\sqrt{n}\rfloor$. Следовательно, проверить простоту числа и найти его разложение на простые множители можно за время $O(\sqrt{n})$.

Показанная ниже функция `prime` проверяет, является ли целое число n простым. Она пытается поделить n на все целые числа от 2 до $\lfloor\sqrt{n}\rfloor$; если ни одно из них не является делителем, то n простое.

```
bool prime(int n) {
   if (n < 2) return false;
   for (int x = 2; x*x <= n; x++) {
      if (n%x == 0) return false;
   }
```

```
    return true;
}
```

Следующая функция строит вектор, содержащий разложение n на простые множители. Функция делит число n на его простые множители и добавляет их в вектор. Процесс заканчивается, когда у n не останется множителей между 2 и $\lfloor\sqrt{n}\rfloor$. Если при этом $n > 1$, то оно простое и добавляется в качестве последнего множителя.

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

Отметим, что каждый простой множитель встречается в этом векторе столько раз, какова его степень в разложении n. Например, $12 = 2^2 \cdot 3$, поэтому функция вернет результат [2, 2, 3].

Свойства простых чисел. Легко доказать, что простых чисел бесконечно много. Если бы это было не так, то мы могли бы построить множество $P = \{p_1, p_2, \ldots, p_n\}$, содержащее все простые числа. В него вошли бы $p_1 = 2$, $p_2 = 3$, $p_3 = 5$ и т. д. Но тогда рассмотрим число

$$p_1 p_2 \ldots p_n + 1.$$

Оно больше каждого из элементов P и не делится ни на один из них. Следовательно, его наименьший простой множитель больше каждого из чисел $p_1, p_2, \ldots, p_n$ в противоречии с тем, что p_n – наибольшее простое число. Таким образом, множество простых чисел должно быть бесконечным.

Обозначим $\pi(n)$ количество простых чисел, не превосходящих n. Например, $\pi(10) = 4$, поскольку существует четыре простых числа, не превосходящих 10: 2, 3, 5 и 7. Можно доказать, что

$$\pi(n) \approx \frac{n}{\ln n},$$

т. е. простые числа встречаются довольно часто. Так, приближенное значение $\pi(10^6)$ равно $10^6 / \ln 10^6 \approx 72\,382$, а точное значение равно 78 498.

11.1.2. *Решето Эратосфена*

Решето Эратосфена – это алгоритм предварительной обработки, который строит массив `sieve`, позволяющий для каждого целого числа от 2 до n эффективно определить, является ли оно простым. Если x простое, то `sieve`[x] = 0, иначе `sieve`[x] = 1. На рис. 11.1 показано содержимое массива `sieve` для $n = 20$.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1

Рис. 11.1. Результат работы решета Эратосфена для $n = 20$

Для построения массива алгоритм перебирает числа $2 \cdot n$. Обнаружив новое простое число x, алгоритм помечает, что числа $2x$, $3x$, $4x$ и т. д. не простые. Ниже показана возможная реализация алгоритма в предположении, что вначале все элементы `sieve` равны нулю:

```
for (int x = 2; x <= n; x++) {
   if (sieve[x]) continue;
      for (int u = 2*x; u <= n; u += x) {
         sieve[u] = 1;
   }
}
```

Внутренний цикл алгоритма выполняется $\lfloor n/x \rfloor$ раз для каждого значения x. Следовательно, время работы может быть оценено сверху частичной суммой гармонического ряда:

$$\sum_{x=2}^{n} \lfloor n/x \rfloor = \lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \lfloor n/4 \rfloor + \cdots = O(n \log n).$$

На самом деле алгоритм более эффективен, поскольку внутренний цикл выполняется, только если число x простое. Можно показать, что временная сложность алгоритма равна всего лишь $O(n \log \log n)$, что весьма близко к $O(n)$. На практике решето Эратосфена очень эффективно; в табл. 11.1 показано реальное время выполнения для различных n.

Таблица 11.1. Время работы решета Эратосфена

Верхняя граница n	Время работы (с)
10^6	0.01
$2 \cdot 10^6$	0.03
$4 \cdot 10^6$	0.07
$8 \cdot 10^6$	0.14

Верхняя граница *n*	Время работы (с)
$16 \cdot 10^6$	0.28
$32 \cdot 10^6$	0.57
$64 \cdot 10^6$	1.16
$128 \cdot 10^6$	2.35

Существует несколько обобщений решета Эратосфена. Например, для каждого числа *k* можно вычислить его наименьший простой множитель (рис. 11.2). А после этого мы с помощью решета можем эффективно разложить на множители любое число от 2 до *n*. (Отметим, что количество простых множителей числа *n* имеет порядок $O(\log n)$.)

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	3	2	5	2	7	2	3	2	11	2	13	2	3	2	17	2	19	2

Рис. 11.2. Обобщенное решето Эратосфена для $n = 20$

11.1.3. *Алгоритм Евклида*

Наибольшим общим делителем (НОД) целых чисел *a* и *b*, gcd(*a*, *b*), называется наибольшее целое число, которое делит одновременно *a* и *b*. Например, gcd(30, 12) = 6. С наибольшим общим делителем тесно связано понятие *наименьшего общего кратного* (НОК) – наименьшего целого числа, которое делится одновременно на *a* и на *b*; оно обозначается lcm(*a*, *b*). Формулу

$$\text{lcm}(a,b) = \frac{ab}{\text{gcd}(a,b)}$$

можно использовать для вычисления наименьшего общего кратного. Например, lcm(30, 12) = 360/gcd(30, 12) = 60.

Один из способов нахождения gcd(*a*, *b*) – разложить *a* и *b* на простые множители, а затем для каждого простого числа взять наибольшую степень, в которой оно входит в оба разложения. Так, чтобы вычислить gcd(30, 12), мы можем построить разложения $30 = 2 \cdot 3 \cdot 5$ и $12 = 2^2 \cdot 3$, а затем сделать вывод, что $\text{gcd}(30, 12) = 2 \cdot 3 = 6$. Однако для больших *a* и *b* этот метод неэффективен.

Алгоритм Евклида дает эффективный способ вычисления gcd(*a*, *b*). Он основан на формуле

$$\text{gcd}(a, b) = \begin{cases} a & b = 0 \\ \text{gcd}(b, a \bmod b) & b \neq 0 \end{cases}.$$

Например:

$$\gcd(30, 12) = \gcd(12, 6) = \gcd(6, 0) = 6.$$

Этот алгоритм можно реализовать следующим образом:

```
int gcd(int a, int b) {
   if (b == 0) return a;
   return gcd(b, a%b);
}
```

Почему данный алгоритм работает? Чтобы разобраться в этом, обратимся к рис. 11.3, где $x = \gcd(a, b)$. Поскольку число x делит a и b, оно должно делить и $a \bmod b$, откуда следует справедливость рекуррентной формулы.

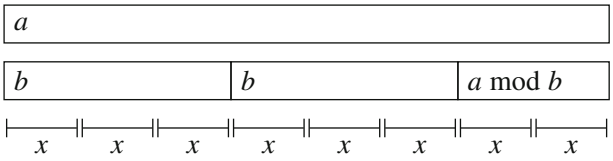


Рис. 11.3. Почему работает алгоритм Евклида

Можно доказать, что алгоритм Евклида работает за время $O(\log n)$, где $n = \min(a, b)$.

Расширенный алгоритм Евклида. Алгоритм Евклида можно модифицировать так, чтобы он давал числа x и y – такие, что

$$ax + by = \gcd(a, b).$$

Например, при $a = 30$ и $b = 12$ имеем

$$30 \cdot 1 + 12 \cdot (-2) = 6.$$

Эту задачу также можно решить, воспользовавшись тождеством $\gcd(a, b) = \gcd(b, a \bmod b)$. Предположим, что задача уже решена для $\gcd(b, a \bmod b)$, и мы знаем x' и y' – такие, что

$$bx' + (a \bmod b)y' = \gcd(a, b).$$

Тогда поскольку $a \bmod b = a - \lfloor a/b \rfloor \cdot b$, то

$$bx' + (a - \lfloor a/b \rfloor \cdot b)y' = \gcd(a, b),$$

или

$$ay' + b(x' - \lfloor a/b \rfloor \cdot y') = \gcd(a, b).$$

Таким образом, мы можем взять $x = y'$ и $y = x' - \lfloor a/b \rfloor \cdot y'$. На этой идее основана показанная ниже функция, которая возвращает кортеж $(x, y, \gcd(a, b))$, удовлетворяющий уравнению.

```
tuple<int,int,int> gcd(int a, int b) {
    if (b == 0) {
        return {1,0,a};
    } else {
        int x,y,g;
        tie(x,y,g) = gcd(b,a%b);
        return {y,x-(a/b)*y,g};
    }
}
```

Эту функцию можно использовать следующим образом:

```
int x,y,g;
tie(x,y,g) = gcd(30,12);
cout << x << " " << y << " " << g << "\n"; // 1 -2 6
```

11.1.4. *Возведение в степень по модулю*

Часто бывает нужно эффективно вычислить значение $x^n \bmod m$. Это можно сделать за время $O(\log n)$, воспользовавшись следующим рекуррентным соотношением:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ четно} \\ x^{n-1} \cdot x & x \text{ нечетно} \end{cases}.$$

Например, для вычисления x^{100} мы сначала вычисляем x^{50}, а затем пользуемся формулой $x^{100} = x^{50} \cdot x^{50}$. Далее для вычисления x^{50} мы сначала вычисляем x^{25} и т. д. Поскольку при четном n показатель степень уменьшается вдвое, это вычисление занимает время $O(\log n)$.

Алгоритм реализуется следующей функцией:

```
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x,n/2,m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

11.1.5. *Теорема Эйлера*

Два целых числа a и b называются *взаимно простыми*, если $\gcd(a, b) = 1$. *Функция Эйлера* $\varphi(n)$ определяет количество целых чисел от 1 до n, взаимно простых с n. Например, $\varphi(10) = 4$, потому что числа 1, 3, 7 и 9 взаимно просты с 10.

Для любого n значение $\varphi(n)$ можно вычислить, зная разложение n на простые множители, по формуле

$$\varphi(n) = \prod_{i=1}^{k} p_i^{\alpha_i - 1}(p_i - 1).$$

Например, поскольку $10 = 2 \cdot 5$, то $\varphi(10) = 20 \cdot (2 - 1) \cdot 50 \cdot (5 - 1) = 4$.

Теорема Эйлера утверждает, что

$$x^{\varphi(m)} \bmod m = 1$$

для всех положительных взаимно простых чисел x и m. Так, согласно теореме Эйлера $7^4 \bmod 10 = 1$, поскольку 7 и 10 – взаимно простые числа и $\varphi(10) = 4$.

Если m простое, то $\varphi(m) = m - 1$, и эта формула принимает вид

$$x^{m-1} \bmod m = 1.$$

В таком виде она известна как *малая теорема Ферма*. Из нее следует, что

$$x^n \bmod m = x^{n \bmod (m-1)} \bmod m,$$

и этот факт можно использовать для вычисления x^n при очень больших n.

Вычисление обратной величины по модулю. *Обратной величиной x относительно умножения* по модулю m называется такое число $\text{inv}_m(x)$, что

$$x \cdot \text{inv}_m(x) \bmod m = 1.$$

Например, $\text{inv}_{17}(6) = 3$, т. к. $6 \cdot 3 \bmod 17 = 1$.

Обращение по модулю позволяет делить числа по модулю m, поскольку деление x эквивалентно умножению на $\text{inv}_m(x)$. Например, зная, что $\text{inv}_{17}(6) = 3$, мы можем вычислить, чему равно 36/6 mod 17, найдя значение $36 \cdot 3 \bmod 17$.

Обращение по модулю возможно тогда и только тогда, когда x и m взаимно просты. В этом случае справедлива формула

$$\text{inv}_m(x) = x^{\varphi(m)-1},$$

основанная на теореме Эйлера. В частности, если число m простое, то $\varphi(m) = m - 1$, и эта формула принимает вид

$$\text{inv}_m(x) = x^{m-2}.$$

Например:

$$\text{inv}_{17}(6) \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

Данная формула позволяет эффективно вычислять обратные величины по модулю, применяя алгоритм возведения в степень по модулю (см. раздел 11.1.4).

11.1.6. *Решение уравнений в целых числах*

Диофантовы уравнения. *Диофантовым уравнением* называется уравнение вида

$$ax + by = c,$$

где a, b и c – постоянные, а найти требуется x и y. Все числа в этом уравнении должны быть целыми. Например, одним из решений уравнения

$$5x + 2y = 11$$

является $x = 3$, $y = -2$.

Для эффективного решения диофантова уравнения можно воспользоваться расширенным алгоритмом Евклида (раздел 11.1.3), который находит целые числа x и y – такие, что удовлетворяется уравнение

$$ax + by = \gcd(a, b).$$

Диофантово уравнение разрешимо тогда и только тогда, когда c делится на $\gcd(a, b)$.

Например, найдем целые x и y, удовлетворяющие уравнению

$$39x + 15y = 12.$$

Это уравнение разрешимо, потому что $\gcd(39, 15) = 3$ и $3 \mid 12$. Расширенный алгоритм Евклида дает

$$39 \cdot 2 + 15 \cdot (-5) = 3,$$

а после умножения на 4 это равенство принимает вид

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

так что решением уравнения является $x = 8$, $y = -20$.

Решение диофантова уравнения не единственно, потому что, зная одно решение, можно получить еще бесконечно много. Если пара (x, y) является решением, то решениями будут и все пары вида

$$\left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)} \right),$$

где k – любое целое число.

Китайская теорема об остатках. Эта теорема касается решения системы уравнений вида

$$\begin{aligned} x &= a_1 \bmod m_1 \\ x &= a_2 \bmod m_2 \\ &\ldots \\ x &= a_n \bmod m_n \end{aligned}$$

где числа $m_1, m_2, \ldots, m_n$ попарно взаимно простые.

Можно доказать, что решением этой системы уравнений является

$$x = a_1 X_1 \operatorname{inv}_{m_1}(X_1) + a_2 X_2 \operatorname{inv}_{m_2}(X_2) + \ldots + a_n X_n \operatorname{inv}_{m_n}(X_n),$$

где

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

В этом решении для каждого $k = 1, 2, \ldots, n$

$$a_k X_k \operatorname{inv}_{m_k}(X_k) \bmod m_k = a_k,$$

потому что

$$X_k \operatorname{inv}_{m_k}(X_k) \bmod m_k = 1.$$

Поскольку все остальные члены суммы делятся на m_k, они не влияют на остаток и $x \bmod m_k = a_k$.

Например, решением системы

$$x = 3 \bmod 5$$
$$x = 4 \bmod 7$$
$$x = 2 \bmod 3$$

является число

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Найдя одно решение x, мы можем получить еще бесконечно много решений вида

$$x + m_1 m_2 \ldots m_n.$$

11.2. Комбинаторика

Комбинаторика изучает методы подсчета комбинаций объектов. Обычно наша цель состоит в том, чтобы эффективно подсчитать число комбинаций, не генерируя каждую в отдельности. В этом разделе мы обсудим избранные комбинаторные методы, применимые к широкому кругу задач.

11.2.1. *Биномиальные коэффициенты*

Биномиальный коэффициент $\binom{n}{k}$ – это число способов, которыми можно выбрать k элементов из множества, содержащего n элементов. Например, $\binom{5}{3} = 10$, поскольку у множества {1, 2, 3, 4, 5} есть 10 подмножеств по 3 элемента:

{1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 4}, {1, 3, 5},
{1, 4, 5}, {2, 3, 4}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5}.

Биномиальные коэффициенты можно вычислить, пользуясь рекуррентной формулой

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

с начальными значениями

$$\binom{n}{0} = \binom{n}{n} = 1.$$

Чтобы понять, почему эта формула верна, рассмотрим произвольный элемент множества x. Если мы включим x в подмножество, то останется выбрать $k - 1$ элементов из $n - 1$. Если же мы не станем включать x в подмножество, то нужно будет выбрать k элементов из $n - 1$.

Биномиальные коэффициенты можно также вычислить по формуле:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

основанной на следующем рассуждении. Существует $n!$ перестановок n элементов. Из каждой перестановки мы включаем в подмножество первые k элементов. Поскольку порядок элементов внутри и вне подмножества не играет роли, результат делится на $k!$ и на $(n - k)!$.

Для биномиальных коэффициентов имеет место тождество

$$\binom{n}{k} = \binom{n}{n-k},$$

поскольку мы разбиваем множество n элементов на два подмножества: первое содержит k элементов, а второе – $n - k$ элементов.

Сумма биномиальных коэффициентов равна

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n.$$

Название «биномиальные коэффициенты» связано с формулой возведения бинома $(a + b)$ в степень n:

$$(a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \cdots + \binom{n}{n-1} a^1 b^{n-1} + \binom{n}{n} a^0 b^n.$$

Биномиальные коэффициенты входят также в треугольник Паскаля (рис. 11.4), в котором каждое значение равно сумме двух расположенных над ним.

```
                1
            1       1
        1       2       1
    1       3       3       1
1       4       6       4       1
...     ...     ...     ...     ...
```

Рис. 11.4. Первые пять строк треугольника Паскаля

Мультиномиальные коэффициенты. *Мультиномиальный коэффициент*

$$\binom{n}{k_1, k_2, \ldots, k_m} = \frac{n!}{k_1!k_2!\cdots k_m!}$$

показывает, сколькими способами множество n элементов можно разбить на подмножества размера k_1, k_2, ..., k_m, где $k_1 + k_2 + \ldots + k_m = n$. Мультиномиальные коэффициенты можно считать обобщением биномиальных коэффициентов; при $m = 2$ эта формула превращается в формулу для биномиальных коэффициентов.

Ящики и шары. «Ящики и шары» – полезная модель, описывающая размещение k шаров в n ящиках. Рассмотрим три случая.

Случай 1. В каждом ящике не более одного шара. Например, при $n = 5$ и $k = 2$ существует 10 комбинаций (рис. 11.5). В этом случае число комбинаций в точности равно биномиальному коэффициенту $\binom{n}{k}$.

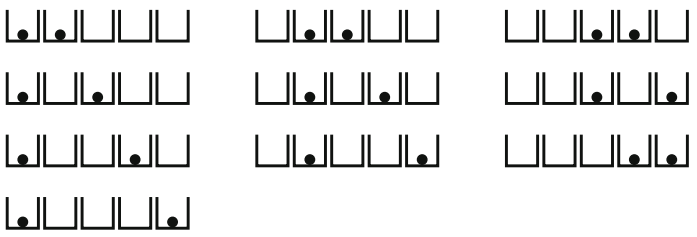


Рис. 11.5. Случай 1: в каждом ящике не более одного шара

Случай 2. В ящике может быть несколько шаров. Так, при $n = 5$ и $k = 2$ существует 15 комбинаций (рис. 11.6). В этом случае процесс размещения шаров в ящиках можно представить в виде строки, состоящей из символов «о» и «→». Предположим, что мы начинаем с левого ящика. Символ «о» означает, что мы кладем шар в текущий ящик, а символ «→» – что переходим к следующему ящику. Каждое решение является строкой длины $k + n - 1$, состоящей из k символов «о» и $n - 1$ символов «→». Так, решению в правом верхнем углу на рис. 11.6 соответствует строка «→ → о → о →». Отсюда вытекает, что число комбинаций равно $\binom{k+n-1}{k}$.

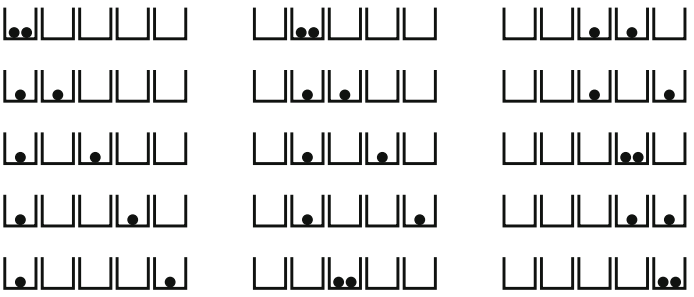


Рис. 11.6. Случай 2: ящик может содержать несколько шаров

Случай 3. В каждом ящике не более одного шара, и, кроме того, никакие два соседних ящика не могут содержать шары одновременно. При $n = 5$ и $k = 2$ существует 6 комбинаций (рис. 11.7). В этом случае можно предположить, что k шаров уже размещены по ящикам и между каждыми двумя соседними ящиками находится один пустой. Наша задача – выбрать позиции остальных пустых ящиков. Всего таких ящиков $n - 2k + 1$, а позиций для них $k + 1$. Следовательно, по формуле, выведенной в случае 2, число решений равно $\binom{n-k+1}{n-2k+1}$.

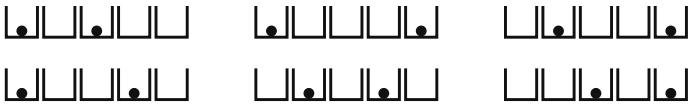


Рис. 11.7. Случай 3: каждый ящик содержит не более одного шара, и никакие два соседних ящика не содержат шары одновременно

11.2.2. *Числа Каталана*

Число Каталана C_n определяет, сколько существует способов правильно расставить скобки в выражении, содержащем n левых и n правых скобок. Например, $C_3 = 5$, т. е. существует пять способов расставить три левые и три правые скобки:

- `()()()`
- `(())()`
- `()(())`
- `((()))`
- `(()())`

Правильная расстановка скобок определяется следующими правилами:

- пустая расстановка правильна;
- если расстановка A правильна, то расстановка (A) также правильна;
- если расстановки A и B правильны, то расстановка AB также правильна.

По-другому охарактеризовать правильные расстановки скобок можно, сказав, что если взять любой префикс такой расстановки, то левых скобок в нем будет не меньше, чем правых, а в расстановке в целом левых и правых скобок поровну.

Числа Каталана можно вычислить по формуле:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1},$$

которая получается, если рассмотреть способы разбиения расстановки скобок на две части, обе из которых являются правильными расстановками и первая содержит минимально возможное число скобок, но при этом не пуста. Для каждого i первая часть содержит $i + 1$ пар скобок, и число правильных расстановок равно произведению следующих величин:

- C_i – количество способов построить расстановку из скобок, входящих в первую часть без учета самых внешних скобок;
- C_{n-i-1} – количество способов построить расстановку из скобок, входящих во вторую часть.

Базой рекурсии является случай $C_0 = 1$, поскольку вообще без скобок можно построить только пустую расстановку.

Числа Каталана можно также вычислить по формуле:

$$C_n = \frac{1}{n+1}\binom{2n}{n},$$

которую можно объяснить так. Всего существует $\binom{2n}{n}$ способов расставить n левых и n правых скобок (необязательно правильно). Посчитаем, сколько из этих расстановок *неправильны*.

Если расстановка скобок неправильна, то она должна содержать префикс, в котором правых скобок больше, чем левых. Идея в том, чтобы выбрать самый короткий из таких префиксов и поменять в нем каждую скобку на противоположную. Например, расстановка `())()(` имеет префикс `())` и после обращения скобок принимает вид `)(((()(`. В получающейся расстановке будет $n + 1$ левых и $n - 1$ правых скобок. На самом деле существует единственный способ получить произвольную расстановку $n + 1$ левых и $n - 1$ правых скобок описанным выше способом. Число таких расстановок равно $\binom{2n}{n+1}$, именно столько существует неправильных расстановок скобок. Следовательно, число правильных расстановок равно

$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}.$$

Подсчет деревьев. С помощью чисел Каталана можно также подсчитать количество некоторых древовидных структур. Прежде всего C_n равно числу двоичных деревьев с n вершинами в предположении, что левая и правая дочерняя вершины различаются. Например, существует 5 двоичных деревьев с 3 вершинами, поскольку $C_3 = 5$ (рис. 11.8). Далее, C_n равно числу корневых деревьев общего вида с $n + 1$ вершинами. На рис. 11.9 показано 5 корневых деревьев с 4 вершинами.

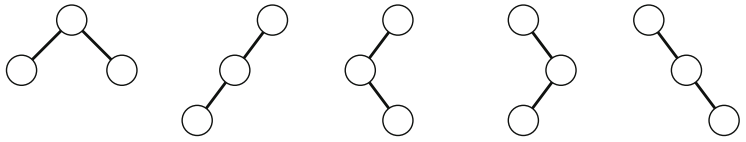


Рис. 11.8. Существует 5 двоичных деревьев с 3 вершинами

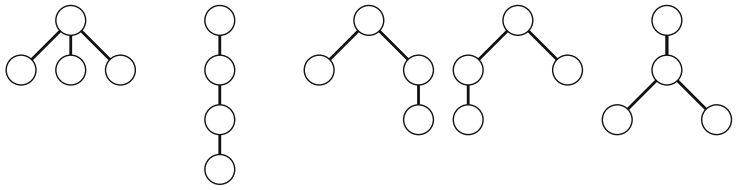


Рис. 11.9. Существует 5 корневых деревьев с 4 вершинами

11.2.3. *Включение-исключение*

Формула *включения-исключения* позволяет подсчитать размер объединения множеств, если известны размеры пересечений, и наоборот. Простой пример ее применения дает формула

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

где A и B – множества, а $|X|$ обозначает размер X. Эта формула иллюстрируется на рис. 11.10. В данном случае мы хотим вычислить размер объединения $A \cup B$, которому соответствует область, принадлежащая хотя бы одному кругу. Площадь $A \cup B$ можно вычислить, сложив площади A и B и вычтя из результата площадь пересечения $A \cap B$.

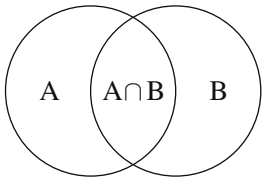


Рис. 11.10. Принцип включения-исключения для двух множеств

Та же идея применима и в случае, когда множеств больше двух. В случае трех множеств формула включений-исключений имеет вид:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|,$$

она иллюстрируется на рис. 11.11.

В общем случае размер объединения $X_1 \cup X_2 \cup \ldots \cup X_n$ можно вычислить, перебрав все возможные пересечения некоторых множеств X_1, X_2, ..., X_n. Если пересечение содержит нечетное число множеств, то его размер учитывается со знаком плюс, а если четное – то со знаком минус.

Аналогичные формулы существуют для вычисления размера пересечения по известным размерам объединений, например:

$$|A \cap B| = |A| + |B| - |A \cup B|$$

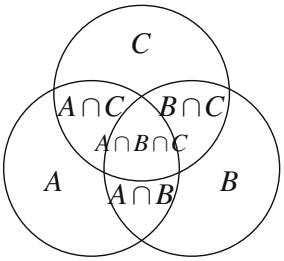


Рис. 11.11. Принцип включения-исключения для трех множеств

и

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Подсчет беспорядков. В качестве примера подсчитаем количество *беспорядков* множества {1, 2, ..., *n*}, т. е. перестановок, не оставляющих ни одного элемента на своем месте. При $n = 3$ существует три беспорядка: (2, 3, 1) и (3, 1, 2).

Один из подходов к решению задачи основан на формуле включений-исключений. Обозначим X_k множество перестановок, в которых элемент k находится в позиции k. При $n = 3$ эти множества таковы:

$$X_1 = \{(1, 2, 3), (1, 3, 2)\},$$
$$X_2 = \{(1, 2, 3), (3, 2, 1)\},$$
$$X_3 = \{(1, 2, 3), (2, 1, 3)\}.$$

Число беспорядков равно

$$n! - |X_1 \cup X_2 \cup \ldots \cup X_n|,$$

поэтому достаточно вычислить $|X_1 \cup X_2 \cup \ldots \cup X_n|$. Благодаря формуле включений-исключений эта задача сводится к вычислению размеров пересечений. Кроме того, пересечение c различных множеств X_k содержит $(n - c)!$ элементов, потому что состоит из всех перестановок, в которых c элементов находятся на своих местах. Следовательно, размеры пересечений легко вычисляются. Например, при $n = 3$

$$\begin{aligned} |X1 \cup X2 \cup X3| &= |X1| + |X2| + |X3| \\ &\quad - |X1 \cap X2| - |X1 \cap X3| - |X2 \cap X3| \\ &\quad + |X1 \cap X2 \cap X3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

так что число беспорядков равно $3! - 4 = 2$.

Но эту задачу можно решить и *без* формулы включений-исключений. Обозначим $f(n)$ количество беспорядков множества {1, 2, ..., *n*}. Можно воспользоваться следующей рекуррентной формулой:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2. \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases}$$

Для доказательства этой формулы рассмотрим, где может оказаться элемент 1 в беспорядке. Существует $n - 1$ способов выбрать элемент x, который займет место элемента 1. В каждом случае есть два варианта.

Вариант 1: элемент x переставляется с элементом 1. Тогда остается построить беспорядок из $n - 2$ элементов.

Вариант 2: элемент x переставляется с элементом, отличным от 1. Тогда нужно построить беспорядок из $n - 1$ элементов, потому что мы не можем переставить x на место элемента 1, а все остальные элементы должны быть переставлены.

11.2.4. *Лемма Бёрнсайда*

Лемму Бёрнсайда можно использовать для подсчета различных комбинаций таким образом, что симметричные комбинации учитываются только один раз. Лемма утверждает, что число таких комбинаций равно

$$\frac{1}{n}\sum_{k=1}^{n} c(k),$$

где n – число способов изменить положение комбинации, а $c(k)$ – число комбинаций, оставшихся неизменными после применения k-го способа.

В качестве примера вычислим количество ожерелий с n жемчужинами m возможных цветов. Два ожерелья считаются симметричными, если они переходят друг в друга в результате поворота. На рис. 11.12 показаны четыре симметричных ожерелья, учитываемых при подсчете как одна комбинация.

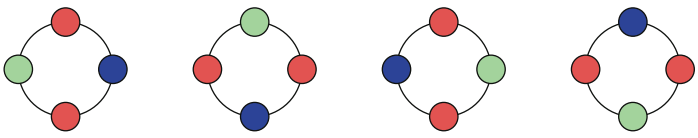


Рис. 11.12. Четыре симметричных ожерелья

Существует n способов изменить положение ожерелья, поскольку его можно повернуть на $k = 0, 1, \ldots, n - 1$ шагов по часовой стрелке. Например, при $k = 0$ ни одно из m^n ожерелий не меняется, а при $k = 1$ не меняются только m ожерелий, в которых все жемчужины одного цвета. В общем случае не меняются $m^{\gcd(k,n)}$ ожерелий, поскольку участки соседних жемчужин длины $\gcd(k, n)$ становятся на место друг друга. Поэтому, согласно лемме Бёрнсайда, количество различных ожерелий равно

$$\frac{1}{n}\sum_{k=0}^{n-1} m^{\gcd(k,n)}.$$

Так, число различных ожерелий с 4 жемчужинами 3 цветов равно

$$\frac{3^4+3+3^2+3}{4}=24.$$

11.2.5. *Теорема Кэли*

Теорема Кэли утверждает, что существует всего n^{n-2} различных помеченных деревьев с n вершинами. Вершины помечаются числами 1, 2, ..., n, а два дерева считаются различными, если они отличаются структурой или пометкой. Например, при $n = 4$ существует $4^{4-2} = 16$ помеченных деревьев, все они показаны на рис. 11.13.

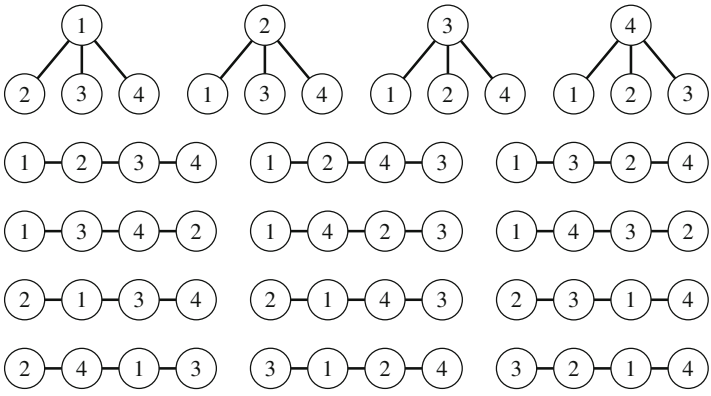


Рис. 11.13. Существует 16 различных помеченных деревьев с 4 вершинами

Для доказательства теоремы Кэли можно воспользоваться кодами Прюфера. *Кодом Прюфера* называется последовательность $n - 2$ чисел, описывающих помеченное дерево. Код строится путем применения следующего процесса, который удаляет из дерева $n - 2$ листьев. На каждом шаге удаляется лист с наименьшей меткой, а метка его единственного соседа добавляется к коду. Дереву, изображенному на рис. 11.14, соответствует код Прюфера [4, 4, 2], поскольку мы удаляем листья 1, 3, 4.

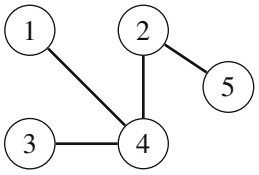


Рис. 11.14. Этому дереву соответствует код Прюфера [4, 4, 2]

Код Прюфера можно построить для любого дерева, и, что важнее, по коду Прюфера можно реконструировать исходное дерево. Следовательно, количество помеченных деревьев с n вершинами равно n^{n-2} – количеству кодов Прюфера длины n.

11.3. Матрицы

Математическому понятию *матрицы* в программировании соответствует двумерный массив. Например:

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

– матрица размера 3×4, т. е. она состоит из 3 строк и 4 столбцов. Элемент матрицы на пересечении i-й строки и j-го столбца обозначается $[i, j]$. Так, для показанной выше матрицы $A[2, 3] = 8$ и $A[3, 1] = 9$.

Частным случаем матрицы является *вектор* – одномерная матрица размера n×1. Например:

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

– вектор из трех элементов.

Транспонированной матрицей A^T называется матрица, в которой строки и столбцы матрицы A переставлены местами, т. е. $A^T[i, j] = A[j, i]$:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}.$$

Матрица называется *квадратной*, если число строк равно числу столбцов. Ниже приведен пример квадратной матрицы:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}.$$

11.3.1. *Операции над матрицами*

Сумма $A + B$ матриц A и B определена, если обе матрицы одного размера. Результатом является матрица, каждый элемент которой равен сумме соответственных элементов A и B. Например:

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

Результатом умножения матрицы A на скалярное значение x является матрица, каждый элемент которой равен произведению соответственного элемента A на x. Например:

$$2\cdot\begin{bmatrix}6 & 1 & 4\\ 3 & 9 & 2\end{bmatrix}=\begin{bmatrix}2\cdot6 & 2\cdot1 & 2\cdot4\\ 2\cdot3 & 2\cdot9 & 2\cdot2\end{bmatrix}=\begin{bmatrix}12 & 2 & 8\\ 6 & 18 & 4\end{bmatrix}.$$

Произведение AB матриц A и B определено, если A имеет размер $a\times n$, а B имеет размер $n\times b$, т. е. ширина A равна высоте B. Результатом является матрица размера $a\times b$, элементы которой вычисляются по формуле:

$$AB[i,j]=\sum_{k=1}^{n}(A[i,k]\cdot B[k,j]).$$

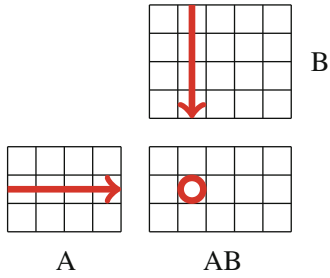


Рис. 11.15. Иллюстрация умножения матриц

Иначе говоря, каждый элемент AB есть сумма произведений элементов A и B, расположенных, как показано на рис. 11.15. Например:

$$\begin{bmatrix}1 & 4\\ 3 & 9\\ 8 & 6\end{bmatrix}\cdot\begin{bmatrix}1 & 6\\ 2 & 9\end{bmatrix}=\begin{bmatrix}1\cdot1+4\cdot2 & 1\cdot6+4\cdot9\\ 3\cdot1+9\cdot2 & 3\cdot6+9\cdot9\\ 8\cdot1+6\cdot2 & 8\cdot6+6\cdot9\end{bmatrix}=\begin{bmatrix}9 & 42\\ 21 & 99\\ 20 & 102\end{bmatrix}.$$

Приведенную выше формулу можно использовать непосредственно для вычисления произведения C двух матриц A и B размера $n\times n$ за время $O(n^3)$:[1]

```
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= n; j++) {
      for (int k = 1; k <= n; k++) {
         C[i][j] += A[i][k]*B[k][j];
      }
   }
}
```

Умножение матриц ассоциативно, т. е. $A(BC) = (AB)C$, но не коммутативно, т. е. в общем случае $AB \neq BA$.

Единичной матрицей называется квадратная матрица, в которой элементы на диагонали равны 1, а все остальные элементы равны 0. Ниже показана единичная матрица 3×3:

[1] Хотя прямолинейного алгоритма сложности $O(n^3)$ для олимпиадного программирования достаточно, существуют *теоретически* более эффективные алгоритмы. Первый такой алгоритм с временной сложностью $O(n^{2.81})$ открыл в 1969 году Штрассен [35], теперь он так и называется – *алгоритмом Штрассена*. Лучший из известных на сегодняшний день алгоритмов был предложен в работе Le Gall [13] в 2014 году, его временная сложность $O(n^{2.37})$.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

При умножении на единичную матрицу исходная матрица не изменяется, например:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \text{ и } \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Степень A^k определена, если матрица A квадратная:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ раз}}.$$

Например:

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

По определению, A^0 – единичная матрица:

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Матрицу A^k можно эффективно вычислить за время $O(n^3 \log k)$ с помощью алгоритма из раздела 11.1.4. Например:

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

11.3.2. *Линейные рекуррентные соотношения*

Линейным рекуррентным соотношением называется функция $f(n)$, начальные значения которой равны $f(0)$, $f(1)$, ..., $f(k-1)$, а следующие вычисляются рекурсивно по формуле

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \ldots c_k f(n-k),$$

где $c_1, c_2, \ldots, c_k$ – постоянные коэффициенты.

Для вычисления любого значения $f(n)$ за время $O(kn)$ можно воспользоваться динамическим программированием, последовательно вычисляя $f(0)$, $f(1)$, ..., $f(n)$. Однако, как мы вскоре увидим, значение $f(n)$ можно вычислить и за время $O(k^3 \log n)$, применяя операции над матрицами. Это существенное улучшение в случае, когда k мало, а n велико.

Числа Фибоначчи. Простым примером линейного рекуррентного соотношения является функция, определяющая последовательность чисел Фибоначчи:

$$f(0) = 0,$$
$$f(1) = 1,$$
$$f(n) = f(n-1) + f(n-2).$$

В данном случае $k = 2$, $c_1 = c_2 = 1$.

Для эффективного вычисления чисел Фибоначчи представим эти формулы в виде квадратной матрицы X размера 2×2, для которой справедливо следующее соотношение:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} == \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}.$$

Тогда значения $f(i)$ и $f(i+1)$ подаются на «вход» X, и X вычисляет по ним $f(i+1)$ и $f(i+2)$. Оказывается, что матрица X равна

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

Например:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Таким образом, $f(n)$ можно вычислить по формуле

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Значение X^n вычисляется за время $O(\log n)$, поэтому $f(n)$ тоже можно вычислить за время $O(\log n)$.

Общий случай. Рассмотрим теперь произвольное линейное рекуррентное соотношение $f(n)$. Наша цель – построить матрицу X, для которой

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Такая матрица имеет вид

$$X = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & \cdots & c_1 \end{bmatrix}.$$

В первых $k - 1$ строках один элемент равен 1, а все остальные равны нулю. Эти строки заменяют $f(i)$ на $f(i + 1)$, $f(i + 1)$ – на $f(i + 2)$ и так далее. А в последней строке находятся коэффициенты рекуррентного соотношения для вычисления нового значения $f(i + k)$.

Теперь $f(n)$ можно вычислить за время $O(k^3 \log n)$ по формуле

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

11.3.3. *Графы и матрицы*

У степеней матриц смежности графов есть ряд интересных свойств. Если M – матрица смежности невзвешенного графа, то M^n для каждой пары вершин (a, b) содержит количество путей, которые начинаются в a, заканчиваются в b и состоят ровно из n ребер. Вершина может встречаться в пути несколько раз.

Рассмотрим граф на рис. 11.16(*a*). Вот его матрица смежности:

$$M = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

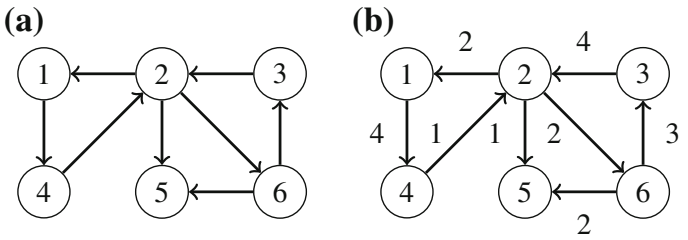


Рис. 11.16. Примеры графов, соответствующих операциям над матрицами

Тогда матрица

$$M^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

позволяет определить число путей, содержащих ровно 4 ребра. Например, $M^4[2, 5] = 2$, потому что существует два пути с 4 ребрами из вершины 2 в вершину 5: $2 \to 1 \to 4 \to 2 \to 5$ и $2 \to 6 \to 3 \to 2 \to 5$.

Аналогичная идея в применении к взвешенному графу позволяет для каждой пары вершин (a, b) вычислить длину кратчайшего пути из a в b, содержащего ровно n ребер. Для этого мы определим умножение матриц по-другому, чтобы вычислять не число путей, а их минимальные веса.

В качестве примера рассмотрим граф на рис. 11.16 (b). Построим матрицу смежности, в которой ∞ означает, что ребра не существует, а любое другое значение интерпретируется как вес соответствующего ребра. Матрица графа имеет вид

$$M = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

При определении матричного умножения мы вместо формулы

$$AB[i,j] = \sum_{k=1}^{n} (A[i,k] \cdot B[k,j]),$$

будем использовать формулу

$$AB[i,j] = \min_{k=1}^{n} (A[i,k] + B[k,j]),$$

т. е. будем вычислять минимумы вместо сумм и суммы вместо произведений. При такой модификации операция возведения матрицы в степень дает веса кратчайших путей. Например:

$$M^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

и, значит, кратчайший путь с 4 ребрами из вершины 2 в вершину 5 имеет вес 8. Это путь 2 → 1 → 4 → 2 → 5.

11.3.4. *Метод Гаусса*

Метод Гаусса – это способ решения системы линейных уравнений. Идея заключается в том, чтобы представить систему в виде матрицы и применить к ней последовательность простых операций над строками, которые сохраняют существенную информацию о системе и вместе с тем позволяют найти значения всех переменных.

Пусть дана система n линейных уравнений с n переменными:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n &= b_2 \\ \ldots & \\ a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{n,n}x_n &= b_n \end{aligned}$$

Представим ее в виде матрицы:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{bmatrix}.$$

Для решения системы мы хотим преобразовать эту матрицу к виду

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & c_n \end{bmatrix},$$

откуда сразу находим решение $x_1 = c_1$, $x_2 = c_2$, ..., $x_n = c_n$. В процессе преобразования мы будем использовать операции трех типов:

1. Перестановка двух строк.
2. Умножение всех элементов одной строки на неотрицательное число.
3. Сложение одной строки, умноженной на скаляр, с другой строкой.

После выполнения любой из этих операций множество решений системы не изменяется. Мы можем последовательно обработать все столбцы матрицы таким образом, что временная сложность алгоритма составит $O(n^3)$.

Рассмотрим следующую систему уравнений:

$$\begin{aligned} 2x_1 + 4x_2 + x_3 &= 16 \\ x_1 + 2x_2 + 5x_3 &= 17 \\ 3x_1 + x_2 + x_3 &= 8 \end{aligned}$$

Для нее матрица имеет вид

$$\begin{bmatrix} 2 & 4 & 1 & 16 \\ 1 & 2 & 5 & 17 \\ 3 & 1 & 1 & 8 \end{bmatrix}.$$

Будем обрабатывать эту матрицу по столбцам. Наша цель на каждом шаге – сделать так, чтобы в нужной позиции столбца оказалась единица, а во всех остальных – нули. При обработке первого столбца сначала умножим первую строку на $\frac{1}{2}$:

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 1 & 2 & 5 & 17 \\ 3 & 1 & 1 & 8 \end{bmatrix}.$$

Затем прибавляем первую строку, умноженную на –1, ко второй и первую строку, умноженную на –3, к третьей:

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 0 & 0 & \frac{9}{2} & 9 \\ 0 & -5 & -\frac{1}{2} & -16 \end{bmatrix}.$$

После этого переходим к обработке второго столбца. Поскольку второй элемент второго столбца равен нулю, мы сначала переставляем вторую и третью строки:

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 0 & -5 & -\frac{1}{2} & -16 \\ 0 & 0 & \frac{9}{2} & 9 \end{bmatrix}.$$

Теперь умножаем вторую строку на $-\frac{1}{5}$ и прибавляем ее произведение на скаляр –2 к первой:

$$\begin{bmatrix} 1 & 0 & \frac{3}{10} & \frac{8}{5} \\ 0 & 1 & \frac{1}{10} & \frac{16}{5} \\ 0 & 0 & \frac{9}{2} & 9 \end{bmatrix}.$$

Осталось обработать третий столбец. Сначала умножаем третью строку на $\frac{2}{9}$, а затем прибавляем ее произведение на скаляр $-\frac{3}{10}$ ко второй строке и произведение на скаляр $-\frac{1}{10}$ к первой строке:

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \end{bmatrix}.$$

Теперь из последнего столбца матрицы следует, что решение исходной системы уравнений $x_1 = 1$, $x_2 = 3$, $x_3 = 2$.

Отметим, что метод Гаусса работает правильно, только если система уравнений имеет *единственное* решение. Например, у системы

$$\begin{aligned} x_1 + x_2 &= 2 \\ 2x_1 + 2x_2 &= 4 \end{aligned}$$

бесконечно много решений, потому что оба уравнения содержат одну и ту же информацию. С другой стороны, система

$$x_1 + x_2 = 5$$
$$x_1 + x_2 = 7$$

неразрешима, т. к. уравнения противоречат друг другу. Если у системы нет единственного решения, то алгоритм это обнаружит, потому что на каком-то этапе мы не сможем успешно обработать столбец.

11.4. Вероятность

Вероятностью называется вещественное число от 0 до 1, показывающее, насколько вероятно событие. Если событие произойдет наверняка, то его вероятность равна 1, а вероятность невозможного события равна 0. Вероятность события обозначается $P(\ldots)$, где многоточие описывает событие. Например, бросание игральной кости может завершиться одним из шести исходов: 1, 2, …, 6, и P(«выпало четное число»)=1/2.

Чтобы вычислить вероятность события, мы можем либо воспользоваться комбинаторикой, либо смоделировать процесс, порождающий события. Рассмотрим эксперимент: на стол выкладываются три верхние карты из перетасованной колоды[2]. Какова вероятность, что все три карты будут иметь одинаковое достоинство (например, ♠8, ♣8 и ♦8)?

Можно вычислить вероятность по формуле:

$$\frac{\textit{число благоприятных исходов}}{\textit{общее число исходов}}.$$

В нашем примере благоприятными считаются исходы, при которых все три карты имеют одинаковое достоинство. Таких исходов $13\binom{4}{3}$, потому что существует 13 способов выбрать достоинство карты и $\binom{4}{3}$ способов выбрать три масти из четырех. Всего же исходов $\binom{52}{3}$, поскольку мы выбираем 3 карты из 52. Следовательно, вероятность события равна

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

Другой способ вычисления вероятности – смоделировать процесс, порождающий события. В нашем случае мы вытаскиваем три карты, поэтому процесс состоит из трех шагов. Требуется, чтобы каждый шаг был успешным.

Вытаскивание первой карты всегда успешно, поскольку нас устраивает любая карта. Второй шаг будет успешным с вероятностью 3/51, потому что осталась 51 карта и 3 из них имеют такое же достоинство, как первая. Точ-

[2] В колоде 52 карты. У каждой карты есть масть (пики ♠, бубны ♦, трефы ♣, черви ♥) и достоинство (целое число от 1 до 13).

но так же третий шаг будет успешным с вероятностью 2/50. Следовательно, вероятность успешного завершения всего процесса равна

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

11.4.1. *Операции с событиями*

События удобно представлять с помощью множеств. Например, множество возможных исходов бросания кости {1, 2, 3, 4, 5, 6}, а любое его подмножество является событием. Событию «выпало четное число» соответствует множество {2, 4, 6}.

Каждому исходу x сопоставляется вероятность $p(x)$, а вероятность $P(X)$ события X вычисляется по формуле:

$$P(X) = \sum_{x \in X} p(x).$$

В случае бросания кости $p(x) = 1/6$ для любого исхода x, поэтому вероятность события «выпало четное число» равна

$$p(2) + p(4) + p(6) = 1/2.$$

Поскольку события представлены множествами, мы можем манипулировать ими, применяя обычные теоретико-множественные операции.

- *Дополнение* $\overline{A}$ означает «A не произошло». Так, при бросании кости дополнение события $A = \{2, 4, 6\}$ равно $\overline{A} = \{1, 3, 5\}$.
- *Объединение* $A \cup B$ означает «произошло A или B». Объединением событий $A = \{2, 5\}$ и $B = \{4, 5, 6\}$ является $A \cup B = \{2, 4, 5, 6\}$.
- *Пересечение* $A \cap B$ означает «произошло A и B». Пересечением событий $A = \{2, 5\}$ и $B = \{4, 5, 6\}$ является $A \cap B = \{5\}$.

Дополнение. Вероятность события $\overline{A}$ вычисляется по формуле

$$P(\overline{A}) = 1 - P(A).$$

Иногда дополнения позволяют легко решить задачу, противоположную исходной. Например, вероятность выбросить хотя бы одну шестерку при бросании кости 10 раз равна

$$1 - (5/6)^{10}.$$

Здесь 5/6 – вероятность выбросить любое число, кроме шести, а $(5/6)^{10}$ – вероятность ни разу из десяти не выбросить шестерку. Дополнительная вероятность и есть ответ на исходный вопрос.

Объединение. Вероятность события $A \cup B$ вычисляется по формуле

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

Например, рассмотрим события A = «выпало четное число» и B = «выпало число меньше 4». В этом случае $A \cup B$ означает «выпавшее число четно или меньше 4», а его вероятность равна

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

Если события A и B *дизъюнктны*, т. е. $A \cap B$ пусто, то вероятность события $A \cup B$ равна просто

$$P(A \cup B) = P(A) + P(B).$$

Пересечение. Вероятность события $A \cap B$ вычисляется по формуле

$$P(A \cap B) = P(A)P(B|A),$$

где $P(B|A)$ – *условная вероятность* того, что B произойдет, если известно, что A произошло. Так, для событий из предыдущего примера $P(B|A) = 1/3$, потому что мы знаем, что исход принадлежит множеству, и лишь один из исходов в этом множестве меньше 4. Следовательно,

$$P(A \cap B) = P(A)P(B|A) = 1/2 \cdot 1/3 = 1/6.$$

События A и B называются *независимыми*, если

$$P(A|B) = P(A) \text{ и } P(B|A) = P(B),$$

т. е. тот факт, что B произошло, не изменяет вероятность A, и наоборот. В этом случае вероятность пересечения равна

$$P(A \cap B) = P(A)P(B).$$

11.4.2. *Случайные величины*

Случайной величиной называется значение, порождаемое случайным процессом. Например, в случае бросания двух костей можно определить случайную величину

X = «сумма исходов».

Если имели место исходы [4, 6] (т. е. на первой кости выпала четверка, а на второй шестерка), то значение X равно 10.

Будем обозначать $P(X = x)$ вероятность того, что случайная величина X равна x. Так, при бросании двух костей $P(X = 10) = 3/36$, поскольку общее число исходов равно 36 и существует три способа получить в сумме 10: [4, 6], [5, 5], [6, 4].

Математическое ожидание. *Математическое ожидание* $E[X]$ описывает среднее значение случайной величины X. Его можно записать в виде суммы

$$\sum_x P(X = x)x,$$

где x пробегает все возможные значения X.

Так, при бросании кости математическое ожидание исхода равно

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

Полезным свойством математического ожидания является *линейность*. Это означает, что математическое ожидание суммы $E[X_1 + X_2 + \ldots + X_n]$ равно сумме математических ожиданий $E[X_1] + E[X_2] + \ldots + E[X_n]$. Это верно даже в случае, когда случайные величины зависимы. Например, при бросании двух костей математическое ожидание выпавших значений равно

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Теперь рассмотрим задачу о случайном размещении n шаров по n ящикам. Пусть требуется найти математическое ожидание количества пустых ящиков. Каждый шар с одинаковой вероятностью может быть помещен в любой ящик.

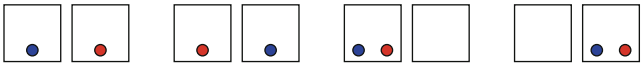


Рис. 11.17. Возможные способы разместить два шара в трех ящиках

На рис. 11.17 показаны вероятности при $n = 2$. В этом случае математическое ожидание количества пустых ящиков равно

$$\frac{0+0+1+1}{4} = \frac{1}{2}.$$

А в общем случае вероятность того, что один ящик окажется пустым, равна

$$\left(\frac{n-1}{n}\right)^n,$$

потому что в нем не должно оказаться ни одного шара. В силу свойства линейности математическое ожидание количества пустых ящиков равно

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

Распределения. *Распределение* случайной величины X описывает вероятности принимаемых X значений, $P(X = x)$. Например, распределение суммы двух брошенных костей таково:

x	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

Случайная величина X с *равномерным распределением* принимает любое из n возможных значений a, $a + 1$, …, b с вероятностью $1/n$. Так, при бросании кости $a = 1$, $b = 6$ и $P(X = x) = 1/6$ для каждого значения x.

Математическое ожидание равномерно распределенной величины X равно

$$E[X] = \frac{a+b}{2}.$$

Если производится n испытаний и вероятность успеха в каждом испытании равна p, то случайная величина X, равная количеству успешных испытаний, имеет *биномиальное распределение*:

$$P(X = x) = p^x(1 - p)^{n-x} \binom{n}{x},$$

где p^x и $(1 - p)^{n-x}$ соответствуют успешным и неудачным испытаниям, а $\binom{n}{x}$ определяет, сколькими способами можно выбрать порядок испытаний.

Например, если бросить кость десять раз, то вероятность ровно три раза выбросить шестерку равна $(1/6)^3(5/6)^7\binom{10}{3}$.

Математическое ожидание биномиально распределенной случайной величины X равно

$$E[X] = pn.$$

Пусть вероятность успеха в каждом испытании равна p и испытания продолжаются до первого успеха. Тогда случайная величина X, равная числу проведенных испытаний, имеет *геометрическое распределение*:

$$P(X = x) = (1 - p)^{x-1} p,$$

где $(1 - p)^{x-1}$ соответствует неудачным испытаниям, а p – первому успешному испытанию.

Например, если бросать кость до первого выпадения шестерки, то вероятность, что число бросаний окажется равным в точности 4, равна $(5/6)^3 1/6$.

Математическое ожидание геометрически распределенной случайной величины X равно

$$E[X] = \frac{1}{p}.$$

11.4.3. *Марковские цепи*

Марковской цепью называется случайный процесс, состоящий из состояний и переходов между ними. Для каждого состояния известны вероятности перехода в другие состояния. Марковскую цепь можно представить в виде графа, вершины которого соответствуют состояниям, а ребра описывают переходы.

В качестве примера рассмотрим следующую задачу. Мы находимся на первом этаже n-этажного здания. На каждом шаге мы случайным образом принимаем решение – подняться или спуститься на один этаж – с двумя исключениями: с первого этажа мы всегда поднимаемся, а с последнего всегда спускаемся. Какова вероятность после k шагов оказаться на этаже m?

В этой задаче каждый этаж соответствует состоянию марковской цепи. На рис. 11.18 показана цепь при $n = 5$.

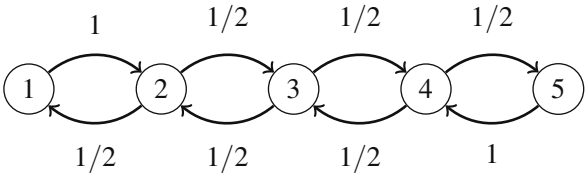


Рис. 11.18. Марковская цепь для пятиэтажного здания

Распределением вероятности марковской цепи является вектор $[p_1, p_2, \ldots, p_n]$, в котором p_k – вероятность нахождения в состоянии k. При этом имеет место равенство $p_1 + p_2 + \ldots + p_n = 1$.

В данном случае начальное распределение равно [1, 0, 0, 0, 0], поскольку мы находимся на первом этаже. Следующее распределение – [0, 1, 0, 0, 0], поскольку с первого этажа можно попасть только на второй. А затем мы можем либо подняться, либо спуститься на этаж, так что следующее распределение равно [1/2, 0, 1/2, 0, 0]. И так далее.

Для моделирования блуждания, описываемого марковской цепью, эффективно будет применить динамическое программирование. Идея заключается в том, чтобы запоминать распределение вероятности и на каждом шаге перебирать все возможные ходы. С помощью этого метода мы можем смоделировать блуждание с m шагами за время $O(n^2m)$.

Переходы марковской цепи можно также представить в виде матрицы, которая обновляет распределение вероятности. В рассматриваемом случае матрица имеет вид:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

При умножении распределения вероятности на эту матрицу мы получаем новое распределение после выполнения одного шага. Так, переход от распределения [1, 0, 0, 0, 0] к распределению [0, 1, 0, 0, 0] производится следующим образом:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Применяя эффективный алгоритм возведения матрицы в степень, мы сможем вычислить распределение после m шагов за время $O(n^3 \log m)$.

11.4.4. *Рандомизированные алгоритмы*

Иногда при решении задачи можно применить случайный выбор, даже если она не связана с вероятностью. Такие алгоритмы называются *рандомизированными*. Существует два основных типа рандомизированных алгоритмов:

- *алгоритм типа Монте-Карло* иногда может давать неверный ответ. Чтобы такой алгоритм был полезен, вероятность неверного ответа должна быть мала;
- *алгоритм типа Лас-Вегас* всегда дает правильный ответ, но время его работы является случайной величиной. Наша цель – спроектировать алгоритм, который с высокой вероятностью будет работать эффективно.

Мы рассмотрим три задачи, которые можно решить с помощью рандомизированных алгоритмов.

Порядковые статистики. *k-й порядковой статистикой* массива является элемент, который после сортировки в порядке возрастания оказывается на k-м месте. Любую порядковую статистику легко вычислить за время $O(n \log n)$, предварительно отсортировав массив, но так ли необходимо сортировать весь массив, чтобы найти единственный элемент?

Оказывается, что порядковую статистику можно найти с помощью алгоритма типа Лас-Вегас со средним временем работы $O(n)$. Алгоритм выбирает из массива случайный элемент x и перемещает все элементы, меньшие x, влево от него, а все остальные – вправо. Для массива n элементов это занимает время $O(n)$.

Предположим, что в левой части оказалось a элементов, а в правой – b элементов. Если $a = k$, то элемент x является k-й порядковой статистикой. В противном случае, если $a > k$, мы рекурсивно находим k-ю порядковую статистику для левой части, а если $a < k$, то рекурсивно находим r-ю порядковую статистику для правой части, где $r = k - a - 1$. Поиск продолжается, пока не будет найден нужный элемент.

Если элемент x всегда выбирается случайно, то размер массива на каждом шаге уменьшается примерно вдвое, поэтому временная сложность нахождения k-й порядковой статистики приблизительно равна

$$n + n/2 + n/4 + n/8 + \ldots = O(n).$$

Отметим, что в худшем случае временная сложность алгоритма равна $O(n^2)$, потому что может случиться так, что x всегда оказывается одним из наименьших или наибольших элементов массива, и тогда потребуется $O(n)$ шагов. Однако вероятность этого настолько мала, что таким развитием событий можно пренебречь.

Проверка произведения матриц. Пусть даны матрицы A, B, C размера $n \times n$. Наша следующая задача заключается в том, чтобы *проверить* справедливость равенства $AB = C$. Конечно, ее можно решить за время $O(n^3)$, просто вычислив произведение AB, но хочется надеяться, что проверить ответ можно и быстрее.

Оказывается, что задачу можно решить с помощью алгоритма типа Монте-Карло с временной сложностью $O(n^2)$. Идея проста: случайным образом выбрать вектор X, содержащий n элементов, и вычислить матрицы ABX и CX. Если $ABX = CX$, то мы считаем, что $AB = C$, иначе – что $AB \neq C$.

Временная сложность этого алгоритма равна $O(n^2)$, т. к. именно столько времени необходимо для вычисления ABX и CX. Для эффективного вычисления ABX представим произведение в виде $A(BX)$, тогда потребуется только два умножения матриц размера $n \times n$ и $n \times 1$.

Недостаток алгоритма заключается в том, что существует небольшая вероятность признать равенство $AB = C$, хотя на самом деле это не так. Например:

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

но

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

Впрочем, на практике вероятность ошибки мала, и ее можно еще уменьшить, если проверять результат на нескольких случайных векторах, а не на одном.

Раскраска графа. Пусть дан граф с n вершинами и m ребрами. Последняя наша задача – раскрасить вершины графа двумя цветами, так чтобы, по крайней мере, для $m/2$ ребер их концы были разного цвета. На рис. 11.19 показан пример правильной раскраски гра-

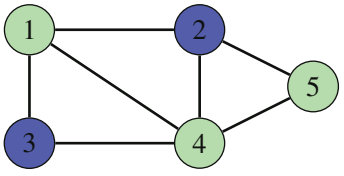


Рис. 11.19. Правильная раскраска графа

фа. В данном случае в графе семь ребер, и концы пяти из них раскрашены в разные цвета.

Задачу можно решить с помощью алгоритма типа Лас-Вегас, который генерирует случайные раскраски до тех пор, пока не будет найдена правильная. При случайном раскрашивании цвет каждой вершины выбирается независимо с вероятностью 1/2. Поэтому математическое ожидание числа ребер с концами разного цвета равно $m/2$. Поскольку можно ожидать, что случайная раскраска правильна, то на практике искомая раскраска будет найдена быстро.

11.5. Теория игр

В этом разделе мы будем рассматривать игры с двумя игроками, которые ходят попеременно. Предполагается, что множеств допустимых ходов для обоих игроков одинаково и что в игре нет элементов случайности. Наша цель – найти стратегию, обеспечивающую выигрыш вне зависимости от ходов противника, если такая стратегия существует.

Оказывается, что для таких игр существует общая стратегия, а для анализа игры можно воспользоваться *теорией игры ним*. Сначала проанализируем простые игры, в которых игроки берут палочки из кучки, а затем обобщим найденную стратегию на другие игры.

11.5.1. *Состояния игры*

Рассмотрим игру, в начале которой имеется кучка из n палочек. Два игрока ходят по очереди, и на каждом ходе игрок может взять из кучки 1, 2 или 3 палочки. Игрок, который забирает последнюю палочку, выигрывает.

Например, при $n = 10$ игра может протекать следующим образом:

- игрок *A* берет 2 палочки (остается 8 палочек);
- игрок *B* берет 3 палочки (остается 5 палочек);
- игрок *A* берет 1 палочку (остается 4 палочки);
- игрок *B* берет 2 палочки (остается 2 палочки);
- игрок *A* берет 2 палочки и выигрывает.

Состояния игры 0, 1, 2, …, n соответствуют количеству оставшихся палочек.

Выигрышным называется состояние, в котором игрок выигрывает, если применяет оптимальную стратегию, а *проигрышным* – состояние, в котором игрок проигрывает, если противник применяет оптимальную стратегию. Оказывается, что любое состояние игры является либо выигрышным, либо проигрышным.

В рассмотренной выше игре состояние 0, очевидно, проигрышное, потому что игрок не может сделать ни одного хода. Состояния 1, 2, 3 выиг-

рышные, потому что игрок может забрать 1, 2 или 3 палочки и выиграть. Состояние 4, напротив, проигрышное, потому что любой ход приводит к состоянию, в котором выигрывает противник.

Вообще, если существует ход, который ведет из текущего состояния в проигрышное, то это выигрышное состояние, в противном случае – проигрышное. Пользуясь этим наблюдением, мы можем классифицировать все состояния игры, начав с проигрышных состояний, в которых нет ни одного возможного хода. На рис. 11.20 показана классификация состояний 0…15 (*W* обозначает выигрышное состояние, *L* – проигрышное).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	*W*	*W*	*W*	*L*	*W*	*W*	*W*	*L*	*W*	*W*	*W*	*L*	*W*	*W*	*W*

Рис. 11.20. Классификация состояний 0–15 в игре в палочки

Эту игру легко проанализировать: состояние *k* проигрышное, если *k* делится на 4, и выигрышное в противном случае. Оптимальная стратегия – всегда брать столько палочек, чтобы количество оставшихся в кучке делилось на 4. В конечном итоге палочек не останется, и противник проиграет. Разумеется, для применения этой стратегии необходимо, чтобы при своем ходе количество палочек *не было* кратно 4, иначе мы ничего не сможем сделать, и противник выиграет, если будет играть оптимально.

Рассмотрим другую игру с палочками: в каждом состоянии *k* разрешено брать из кучки любое число палочек *x*, которое меньше *k* и делит *k*. Например, в состоянии 8 можно взять 1, 2 или 4 палочки, а в состоянии 7 – только одну палочку. На рис. 11.21 состояния 1…9 показаны в виде *графа состояний*, вершинами которого являются состояния, а ребрами – переходы между ними.

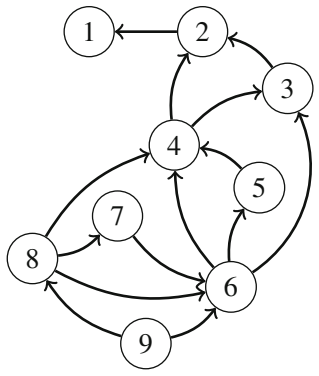


Рис. 11.21. Граф состояний в игре в делимость

В этой игре конечным состоянием всегда является 1 – проигрышное, потому что в нем нет допустимых ходов. На рис. 11.22 показана классификация состояний 1…9. Оказывается, что в этой игре все четные состояния выигрышные, а все нечетные – проигрышные.

1	2	3	4	5	6	7	8	9
L	*W*	*L*	*W*	*L*	*W*	*L*	*W*	*L*

Рис. 11.22. Классификация состояний 0…9 в игре в делимость

11.5.2. *Игра ним*

Ним – простая игра, имеющая большое значение в теории игр, поскольку ту же стратегию можно применять и в других играх. Сначала мы рассмотрим ним, а затем обобщим стратегию на другие игры.

В игре ним есть n кучек, и в каждой из них сколько-то палочек. Игроки ходят попеременно, каждый игрок выбирает кучку, в которой еще остались палочки, и берет из нее любое число палочек. Выигрывает тот, кто забирает последнюю палочку.

Состояния в ним имеют вид $[x_1, x_2, \ldots, x_n]$, где x_i – количество палочек в i-й кучке. Так, [10, 12, 5] – состояние, в котором есть три кучки, содержащие 10, 12 и 5 палочек. Состояние [0, 0, …, 0] проигрышное, т. к. в нем нет возможных ходов; оно всегда является конечным.

Анализ. Оказывается, что состояние в ним легко классифицировать, вычислив *ним-сумму* $s = x_1 \oplus x_2 \oplus \ldots \oplus x_n$, где знаком $\oplus$ обозначена операция *ИСКЛЮЧАЮЩЕЕ ИЛИ*. Состояния, в которых ним-сумма равна 0, проигрышные, остальные – выигрышные. Например, ним-сумма состояния [10, 12, 5] равна $10 \oplus 12 \oplus 5 = 3$, так что это состояние выигрышное.

Но как ним-сумма связана с игрой ним? Это можно объяснить, глядя на то, как меняется ним-сумма при изменении состояния игры.

Проигрышные состояния. Конечное состояние [0, 0, …, 0] проигрышное, и его ним-сумма, как и следовало ожидать, равна 0. В других проигрышных состояниях любой ход приводит к выигрышному состоянию, потому что при изменении одного значения x_i ним-сумма также изменяется, поэтому после хода ним-сумма становится отлична от 0.

Выигрышные состояния. Мы можем перейти в проигрышное состояние, если существует кучка i, для которой $x_i \oplus s < x_i$. В таком случае можно забрать из кучки i столько палочек, чтобы в ней осталось $x_i \oplus s$ палочек, а это приведет к проигрышному состоянию. Такая кучка существует всегда, для нее x_i содержит единичный бит в позиции самого левого единичного бита s.

Пример. Рассмотрим состояние [10, 12, 5]. Это выигрышное состояние, поскольку его ним-сумма равна 3. Следовательно, должен существовать ход, приводящий в проигрышное состояние. Найдем такой ход.

Ним-сумма состояния вычисляется следующим образом:

10	1010
12	1100
5	0101
3	0011

В данном случае кучка с 10 палочками – единственная, для которой число палочек содержит единицу в позиции самого левого единичного бита ним-суммы:

10	10<u>1</u>0
12	1100
5	0101
3	00<u>1</u>1

Новый размер кучки должен быть равен 10 ⊕ 3 = 9, так что следует забрать всего одну палочку. После этого игра переходит в проигрышное состояние [9, 12, 5]:

9	1001
12	1100
5	0101
3	0000

Игра мизер. В этом варианте игры ним цель противоположная – игрок, забравший последнюю палочку, проигрывает. Оказывается, что в игре мизер оптимальная стратегия почти такая же, как в стандартной игре ним.

Идея в том, чтобы сначала играть в мизер как в стандартную игру, а в конце игры изменить стратегию. Смена стратегии происходит в момент, когда после следующего хода в каждой кучке останется не более одной палочки. В стандартной игре мы сделали бы ход, после которого осталось бы четное число кучек с одной палочкой. А в игре мизер мы пойдем так, чтобы число кучек с одной палочкой стало нечетным.

Эта идея работает, потому что в игре обязательно имеется состояние, в котором стратегию можно сменить, и это состояние выигрышное, т. к. содержит ровно одну кучку с числом палочек больше 1, поэтому его ним-сумма не равна 0.

11.5.3. *Теорема Шпрага–Гранди*

Теорема Шпрага–Гранди обобщает стратегию игры ним на все игры, удовлетворяющие следующим требованиям:

- два игрока ходят попеременно;
- возможные ходы в каждом состоянии игры не зависят от того, чья очередь ходить;
- игра заканчивается, когда игрок не может сделать следующий ход;
- игра гарантированно заканчивается за конечное время;
- игроки обладают полной информацией о состояниях и допустимых ходах, и в игре нет элемента случайности.

Числа Гранди. Идея в том, чтобы для каждого состояния игры вычислять *число Гранди*, являющееся аналогом числа палочек в кучке ним. Зная числа Гранди всех состояний, мы сможем играть так же, как в ним.

Число Гранди состояния игры вычисляется по формуле

$$\text{mex}(\{g_1, g_2, \ldots, g_n\}),$$

где $g_1, g_2, \ldots, g_n$ – числа Гранди состояний, в которые мы можем перейти из данного, а функция mex возвращает наименьшее неотрицательное число, не принадлежащее множеству. Например, mex({0, 1, 3}) = 2. Если в состоянии нет возможных ходов, то его число Гранди равно 0, потому что mex(∅) = 0.

На рис. 11.23 показан граф состояний, в котором каждому состоянию сопоставлено его число Гранди. Число Гранди проигрышного состояния равно 0, а выигрышного состояния – положительно.

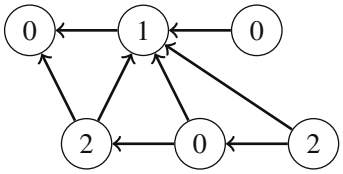


Рис. 11.23. Числа Гранди состояний игры

Рассмотрим состояние, для которого число Гранди равно x. Можно считать, что это соответствует кучке с x палочками в игре ним. В частности, если $x > 0$, то мы можем перейти в состояния с числами Гранди 0, 1, …, $x - 1$, что можно уподобить взятию палочек из кучки. Но есть одно отличие: может оказаться возможен переход в состояние с числом Гранди, большим x, что соответствует «добавлению» палочек в кучку. Однако противник всегда может сделать в точности противоположный ход, поэтому на стратегию это не влияет.

В качестве примера рассмотрим игру, в которой игроки двигают фигурку в лабиринте. Каждая клетка лабиринта либо доступна, либо нет. Получив ход, игрок должен переместить фигурку на любое число шагов влево или вверх. Выигрывает тот, кто сделает последний ход. На рис. 11.24 показана начальная конфигурация, @ обозначает фигурку, а * – квадратики, в которые она может попасть. Состояниями игры являются все доступные клетки. На рис. 11.25 показаны числа Гранди состояний этой игры.

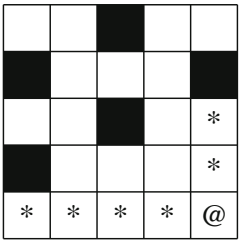


Рис. 11.24. Возможные ходы при первом ходе

Согласно теореме Шпрага–Гранди, каждое состояние игры в лабиринт соответствует кучке в ним. Например, число Гранди правой нижней клетки равно 2, т. е. это выигрышное состояние. Мы можем перейти в проигрышное состояние и тем самым выиграть игру, передвинув фигурку на четыре клетки влево или на две вверх.

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

Рис. 11.25. Числа Гранди состояний игры

Подыгры. Предположим, что игра состоит из подыгр, и, дождавшись своей очереди, игрок сначала выбирает подыгру, а затем ход в ней. Игра заканчи-

вается, когда невозможно сделать ход ни в одной подыгре. В таком случае число Гранди игры равно ним-сумме чисел Гранди подыгр. Тогда в игру можно играть, как в ним, вычисляя числа Гранди всех подыгр, а затем их ним-сумму.

В качестве примера рассмотрим игру, состоящую из трех лабиринтов. На каждом ходе игрок выбирает лабиринт, а затем двигает в нем фигурку. На рис. 11.26 показана начальная конфигурация игры, а на рис. 11.27 – соответствующие числа Гранди. В этой конфигурации ним-сумма чисел Гранди равна $2 \oplus 3 \oplus 3 = 2$, так что выигрывает первый игрок. Один из оптимальных ходов – переместить фигурку на две клетки вверх в первом лабиринте, что дает ним-сумму $0 \oplus 3 \oplus 3 = 0$.

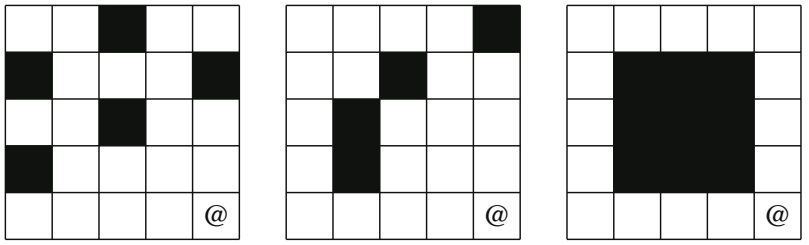


Рис. 11.26. Игра, состоящая из трех подыгр

0	1	■	0	1
■	0	1	2	■
0	2	■	1	0
■	3	0	4	1
0	4	1	3	2

0	1	2	3	■
1	0	■	0	1
2	■	0	1	2
3	■	1	2	0
4	0	2	5	3

0	1	2	3	4
1	■	■	■	0
2	■	■	■	1
3	■	■	■	2
4	0	1	2	3

Рис. 11.27. Числа Гранди для подыгр

Игра Гранди. Иногда один ход разбивает игру на независимые подыгры. В таком случае число Гранди состояния игры равно

$$\text{mex}(\{g_1, g_2, \ldots, g_n\}),$$

где n – число возможных ходов, а

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \ldots \oplus a_{k,m},$$

т. е. ход k разбивает игру на m подыгр с числами Гранди $a_{k,1}, a_{k,2}, \ldots a_{k,m}$.

Примером такой игры является *игра Гранди*. Первоначально имеется одна кучка с n палочками. На каждом ходе игрок выбирает кучку и разбивает ее на две непустые кучки разного размера. Игрок, сделавший последний ход, выигрывает.

Обозначим $g(n)$ число Гранди кучки размера n. Его можно вычислить, перебрав все способы разбиения кучки на две. Так, при $n = 8$ есть три возможности: 1 + 7, 2 + 6 и 3 + 5, поэтому

$$g(8) = \text{mex}(\{g(1) \oplus g(7), g(2) \oplus g(6), g(3) \oplus g(5)\}).$$

В этой игре значение $g(n)$ зависит от $g(1)$, ..., $g(n - 1)$. Начальные условия – $g(1) = g(2) = 0$, потому что кучки с одной и двумя палочками нельзя разбить на меньшие. Вот несколько первых чисел Гранди:

$$g(1) = 0$$
$$g(2) = 0$$
$$g(3) = 1$$
$$g(4) = 0$$
$$g(5) = 2$$
$$g(6) = 1$$
$$g(7) = 0$$
$$g(8) = 2$$

Для $n = 8$ число Гранди равно 2, поэтому игру можно выиграть. К выигрышу ведет ход, заключающийся в создании кучек 1 + 7, потому что $g(1) \oplus g(7) = 0$.

11.6. Преобразование Фурье

Пусть даны два полинома $f(x)$ и $g(x)$. В этом разделе наша задача заключается в том, чтобы эффективно вычислить произведение $f(x)g(x)$. Например, если $f(x) = 2x + 3$ и $g(x) = 5x + 1$, то искомый результат $f(x)g(x) = 10x^2 + 17x + 3$. Простой способ вычислить произведение – перебрать все пары членов $f(x)$ и $g(x)$ и просуммировать их произведения, как показано ниже:

$$f(x)g(x) = 2x \cdot 5x + 2x \cdot 1 + 3 \cdot 5x + 3 \cdot 1 = 10x^2 + 17x + 3.$$

Однако этот метод *медленный*: требуется время $O(n^2)$, где n – степень полиномов. По счастью, произведение можно вычислить за время $O(n \log n)$, воспользовавшись быстрым преобразованием Фурье (БПФ). Идея этого алгоритма заключается в том, чтобы преобразовать полиномы в специальное представление значениями в точках, позволяющее упростить вычисление.

11.6.1. *Работа с полиномами*

Рассмотрим полином степени $n – 1$:

$$f(x) = c_0 + c_1 x + \ldots + c_{n-1} x^{n-1}.$$

Существует два стандартных способа его представления.

- *Представление коэффициентами*: создается список

$$[c_0, c_1, \ldots, c_{n-1}],$$

  содержащий коэффициенты полиномов.

- *Представление значениями в точках*: создается список

$$[(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_{n-1}, f(x_{n-1}))],$$

  содержащий значения полинома в n различных точках. Это представление основано на том, что полином степени $n - 1$ однозначно определен значениями в n различных точках.

Например, рассмотрим полином $f(x) = x^3 + 2x + 5$, для которого представление коэффициентами имеет вид [5, 2, 0, 3]. Чтобы создать представление значениями в точках, выберем произвольно n разных точек и вычислим в них полином. Одно из возможных представлений имеет вид [(0, 5), (1, 8), (2, 17), (3, 38)], т. е. $f(0) = 5$, $f(1) = 8$, $f(2) = 17$ и $f(3) = 38$.

У обоих способов есть достоинства. Зная представление коэффициентами, легко вычислить значение полинома в любой точке. Но для вычисления произведения полиномов $f(x)$ и $g(x)$ представление значениями в точках удобнее: если мы знаем, что $f(x_i) = a_i$ и $g(x_i) = b_i$ в некоторой точке x_i, то легко вычислить $f(x_i)g(x_i) = a_i b_i$. Например, если известно, что $f(1) = 5$ и $g(1) = 6$, то сразу можно сказать, что $f(1)g(1) = 30$.

Тем не менее во всех случаях, кроме вычисления произведений, мы обычно предпочитаем использовать представление коэффициентами. Поэтому для вычисления произведения полиномов $f(x)$ и $g(x)$, заданных своими коэффициентами, можно поступить следующим образом:

1) создать представления $f(x)$ и $g(x)$ значениями в точках;
2) вычислить произведение $f(x)g(x)$ в этом представлении;
3) создать представление $f(x)g(x)$ коэффициентами.

Заметим, что если степени $f(x)$ и $g(x)$ равны $n - 1$, то степень $f(x)g(x)$ равна $2n - 2$. Следовательно, на шаге 1 нужно вычислить $2n - 1$ значений, чтобы на шаге 3 можно было однозначно найти правильный полином.

Для шага 2 нужно время $O(n)$, потому что требуется всего лишь вычислить произведение во всех точках. Шаги 1 и 3 труднее, но ниже мы покажем, как выполнить их за время $O(n \log n)$, применив алгоритм БПФ. Идея в том, чтобы представить полином значениями во вполне определенных точках комплексной плоскости, благодаря чему можно легко переходить от одного представления к другому.

11.6.2. *Алгоритм БПФ*

Пусть дан вектор $a = [c_0, c_1, \ldots, c_{n-1}]$, представляющий полином

$$f(x) = c_0 + c_1x + \ldots + c_{n-1}x^{n-1}.$$

Тогда его *преобразованием Фурье a* называется вектор

$$t = [f(\omega_n^0), f(\omega_n^1), \ldots, f(\omega_n^{n-1})],$$

где

$$\omega_n = e^{2\pi i/n} = \cos(2\pi/n) + \sin(2\pi/n)i.$$

Вектор t соответствует представлению полинома $f(x)$ значениями в точках $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$. Значение ω_n – комплексное число, называемое *главным корнем из единицы*, это число удовлетворяет равенству $\omega_n^n = 1$. На рис. 11.28 показано положение чисел ω_4 и ω_8, а также их степеней на комплексной плоскости.

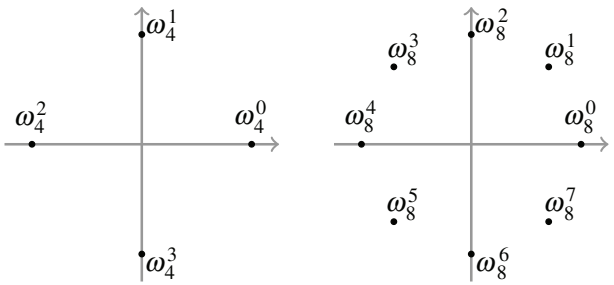


Рис. 11.28. Степени ω_4 и ω_8 на комплексной плоскости

Алгоритм *быстрого преобразования Фурье* (*БПФ*) вычисляет преобразование Фурье за время $O(n \log n)$. Для эффективного вычисления используются свойства чисел ω_n. Начиная с этого момента будем предполагать, что n (длина входного вектора a) – степень двойки. Если это не так, можно добавить нули в конец вектора перед началом работы алгоритма.

Идея алгоритма БПФ заключается в том, чтобы разбить вектор $a = [c_0, c_1, \ldots, c_{n-1}]$ на два вектора $a_{ЧЕТ} = [c_0, c_2, \ldots, c_{n-2}]$ и $a_{НЕЧЕТ} = [c_1, c_3, \ldots, c_{n-1}]$. Оба вектора состоят из $n/2$ элементов и представляют полиномы $c_0 + c_2x + c_4x^2 + \ldots + c_{n-2}x^{n/2-1}$ и $c_1 + c_3x + c_5x^2 + \ldots + c_{n-1}x^{n/2-1}$ соответственно. Затем алгоритм рекурсивно вычисляет преобразования Фурье $a_{ЧЕТ}$ и $a_{НЕЧЕТ}$, получая векторы $t_{ЧЕТ}$ и $t_{НЕЧЕТ}$. Наконец, алгоритм вычисляет преобразование Фурье вектора a по формуле

$$t[k] = t_{ЧЕТ}[k \bmod (n/2)] + t_{НЕЧЕТ}[k \bmod (n/2)]\ \omega_n^k.$$

Эта формула правильна, потому что $\omega_{n/2}^k = \omega_n^{2k}$ и $\omega_n^k = \omega_n^{k \bmod n}$ (см. рис. 11.28). Поскольку алгоритм разбивает входной вектор длины n на два вектора длины $n/2$ и обрабатывает их рекурсивно, время его работы составляет $O(n \log n)$.

Алгоритм БПФ можно также использовать для вычисления *обратного* преобразования Фурье, т. е. для преобразования представления значениями в точках в представление коэффициентами. Удивительно, что если вычислить преобразование Фурье вектора

$$t = [f(\omega_n^0), f(\omega_n^1), \ldots, f(\omega_n^{n-1})],$$

заменив ω_n на $1/\omega_n$ и поделив все выходные значения на n, на выходе получится представление исходного вектора a коэффициентами.

Реализация. Хорошо реализовать алгоритм БПФ не так-то просто. В частности, не стоит создавать новые векторы и обрабатывать их рекурсивно, потому что при такой реализации значения постоянных множителей будут велики. Часто алгоритм рассматривается как *черный ящик*, который умеет эффективно вычислять преобразования Фурье, предпочитая не вдаваться в детали. Показанная ниже реализация основана на псевдокоде, приведенном в CLRS [7]; если вам интересно, что именно делает этот код, обратитесь к книге.

Сначала определим тип комплексного числа `cd`, вещественная и мнимая части которого имеют тип `double`, а также переменную `pi`, равную π.

```
typedef complex<double> cd;
double pi = acos(-1);
```

Функция `fft` выполняет алгоритм БПФ. Ей передается вектор `a`, содержащий коэффициенты полинома, и дополнительный параметр `d`. Если `d` равен 1 (по умолчанию), то функция вычисляет прямое преобразование Фурье, а если –1, то обратное. Как уже было сказано, функция предполагает, что n – степень двойки.

Сначала функция строит вектор `r`, равный *поразрядной обратной перестановке* `a`, соответствующей порядку, в котором производится доступ к значениям на нижнем уровне рекурсии. Этот прием позволяет вычислить преобразование без создания дополнительных векторов и рекурсивных вызовов. После этого функция вычисляет преобразования Фурье векторов длины 2, 4, 8, …, n. Наконец, если требовалось вычислить обратное преобразование Фурье, то все выходные значения делятся на n.

```
vector<cd> fft(vector<cd> a, int d = 1) {
    int n = a.size();
    vector<cd> r(n);
    for (int k = 0; k < n; k++) {
        int b = 0;
        for (int z = 1; z < n; z *= 2) {
            b *= 2;
            if (k&z) b++;
        }
```

```
        r[b] = a[k];
    }
    for (int m = 2; m <= n; m *= 2) {
        cd wm = exp(cd{0,d*2*pi/m});
        for (int k = 0; k < n; k += m) {
            cd w = 1;
            for (int j = 0; j < m/2; j++) {
                cd u = r[k+j];
                cd t = w*r[k+j+m/2];
                r[k+j] = u+t;
                r[k+j+m/2] = u-t;
                w = w*wm;
            }
        }
    }
    if (d == -1) {
        for (int i = 0; i < n; i++) r[i] /= n;
    }
    return r;
}
```

Ниже показано, как с помощью функции `fft` вычислить произведение $f(x) = 2x + 3$ и $g(x) = 5x + 1$. Сначала преобразовываем оба полинома в представление значениями в точках, затем вычисляем произведение и, наконец, выполняем обратное преобразование в представление коэффициентами. Получается $10x^2 + 17x + 3$, как и положено.

```
int n = 4;
vector<cd> f = {3,2,0,0};
vector<cd> g = {1,5,0,0};
auto tf = fft(f);
auto tg = fft(g);
vector<cd> tp(n);
for (int i = 0; i < n; i++) tp[i] = tf[i]*tg[i];
auto p = fft(tp,-1); // [3,17,10,0]
```

Хотя алгоритм БПФ оперирует комплексными числами, входные и выходные значения часто являются целыми. После вычисления произведения мы можем написать `(int)(p[i].real()+0.5)`, чтобы получить вещественную часть комплексного числа `p[i]` и преобразовать ее в целое.

11.6.3. *Вычисление свертки*

В общем случае мы можем использовать алгоритм БПФ для вычисления *свертки* двух массивов за время $O(n \log n)$. Если даны массивы a и b, то

сверткой $c = a * b$ называется массив, элементы которого вычисляются по формуле

$$c(t) = \sum_{i+j=k} a[i]b[i].$$

Если a и b – векторы коэффициентов двух полиномов, то свертка представляет коэффициенты произведения этих полиномов, но можно вычислять и свертки без всякой связи с полиномами. Приведем несколько примеров.

Комбинации. Имеются яблоки и бананы, у каждого плода есть вес, равный целому числу от 1 до n. Для каждого веса $w \leqslant 2n$ мы хотим узнать, сколькими способами можно выбрать одно яблоко и один банан, совокупный вес которых равен w.

Для решения задачи можно создать массивы a и b, так что $a[i]$ равно числу яблок веса i, а $b[i]$ – число бананов веса i. Тогда свертка этих массивов дает искомый результат.

Обработка сигнала. Можно считать, что массив a – *сигнал*, а массив b – *маска*, модифицирующая этот сигнал. Маска сдвигается вдоль сигнала слева направо, и в каждой позиции вычисляется сумма произведений. Результат можно вычислить как свертку, если сначала обратить маску, т. е. поменять порядок элементов на противоположный.

Пусть, например, $a = [5, 1, 3, 4, 2, 1, 2]$ и $b = [1, 3, 2]$. Сначала создадим обращенную маску $b' = [2, 3, 1]$, а затем вычислим свертку

$$c = a * b' = [10, 17, 14, 18, 19, 12, 9, 7, 2].$$

На рис. 11.29 показана интерпретация значений $c[1]$ и $c[5]$.

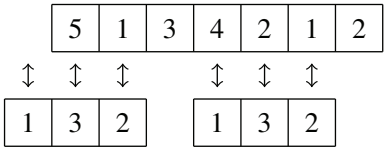


Рис. 11.29. Обработка сигнала:
$c[1] = 5 \cdot 3 + 1 \cdot 2 = 17$ и $c[5] = 4 \cdot 1 + 2 \cdot 3 + 1 \cdot 2$

Разности. Дана битовая строка s длины n, требуется для каждого $k = 1, 2, \ldots, n - 1$ найти, сколькими способами можно выбрать две позиции i и j такие, что $s[i] = s[j] = 1$ и $j - i = k$.

Для решения этой задачи вычислим свертку $c = s * s'$, где s' – обращение s. Тогда элемент $c[n + k - 1]$ дает ответ для k (можно также считать, что s одновременно является сигналом и маской).

Глава 12

Дополнительные алгоритмы на графах

В этой главе обсуждаются некоторые дополнительные алгоритмы на графах.

В разделе 12.1 описан алгоритм нахождения компонент сильной связности графа. Затем мы покажем, как с помощью этого алгоритма эффективно решить задачу 2-выполнимости.

Раздел 12.2 посвящен эйлеровым и гамильтоновым путям. Эйлеров путь проходит по всем ребрам графа в точности один раз, а гамильтонов путь заходит в каждую вершину графа ровно один раз. Хотя на первый взгляд эти задачи очень схожи, их вычислительная сложность отличается кардинально.

В разделе 12.3 мы сначала покажем, как найти максимальный поток между источником и стоком в графе, а затем увидим, как можно свести несколько других задач на графах к задаче о максимальном потоке.

В разделе 12.4 рассматриваются свойства поиска в глубину и задачи, относящиеся к двусвязным графам.

12.1. Сильная связность

Ориентированный граф называется *сильно связным*, если существует путь из любой вершины в любую другую вершину. Так, левый граф на рис. 12.1 сильно связный, а правый – нет. Правый граф не является сильно связным, например потому, что не существует пути из вершины 2 в вершину 1.

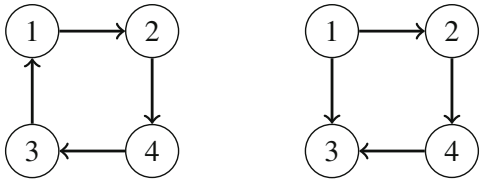


Рис. 12.1. Граф слева является сильно связным, а граф справа – нет

Любой ориентированный граф можно разбить на компоненты сильной связности. Каждая такая компонента состоит из максимального множест-

ва вершин – такого, что существует путь из любой вершины, принадлежащей множеству, в любую другую. Эти компоненты образуют ациклический *граф компонент*, представляющий глубинную структуру исходного графа. На рис. 12.2 показаны граф, его компоненты сильной связности и соответствующий граф компонент. Компонентами являются $A = \{1, 2\}$, $B = \{3, 6, 7\}$, $C = \{4\}$ и $D = \{5\}$.

Поскольку граф компонент ациклический, обработать его проще, чем исходный. Благодаря отсутствию циклов мы можем построить для него топологическую сортировку и обработать ее, применив динамическое программирование.

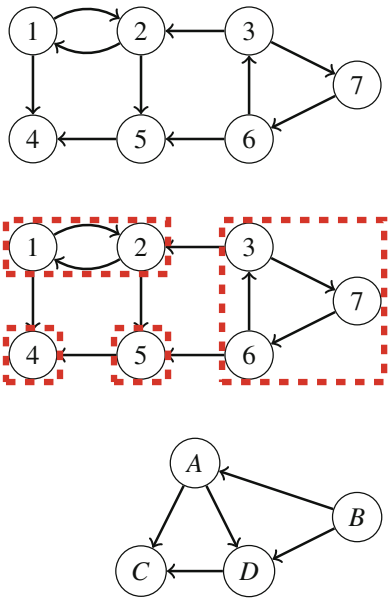


Рис. 12.2. Граф, его компоненты сильной связности и граф компонент

12.1.1. *Алгоритм Косарайю*

Алгоритм Косарайю дает эффективный способ нахождения компонент сильной связности графа. Он включает два поиска в глубину: первый строит список вершин, отражающий структуру графа, а второй формирует компоненты сильной связности.

На первом этапе алгоритма Косарайю строится список вершин в порядке их посещения в процессе поиска в глубину. Алгоритм перебирает все вершины и начинает поиск в глубину из каждой еще не обработанной. По завершении обработки вершина добавляется в список.

На рис. 12.3 показан порядок обработки вершин графа из нашего примера. Нотация *x/y* означает, что обработка вершины началась в момент *x* и закончилась в момент *y*. В результате получается список [4, 5, 2, 1, 6, 7, 3].

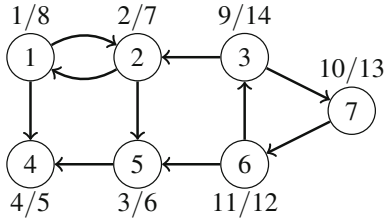


Рис. 12.3. Порядок обработки вершин

На втором этапе алгоритма Косарайю формируются компоненты сильной связности. Сначала все ребра графа инвертируются. Тем самым гарантируется, что в процессе второго поиска будут найдены правильные компоненты сильной связности. На рис. 12.4 показан наш граф с инвертированными ребрами.

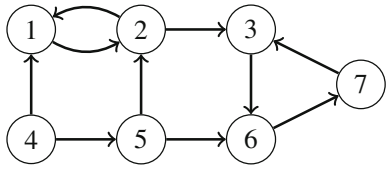


Рис. 12.4. Граф с инвертированными ребрами

После этого алгоритм пробегает по списку вершин, созданному на первом этапе, в *обратном* порядке. Если вершина еще не принадлежит никакой компоненте, то создается новая компонента, для чего алгоритм начинает поиск в глубину, который добавляет все найденные новые вершины в новую компоненту. Отметим, что поскольку все ребра инвертированы, компоненты не «просачиваются» в другие части графа.

На рис. 12.5 показано, как алгоритм обрабатывает наш граф. Вершины обрабатываются в порядке [3, 7, 6, 1, 2, 5, 4]. Сначала по вершине 3 генерируется компонента {3, 6, 7}. Затем вершины 7 и 6 пропускаются, поскольку уже принадлежат некоторой компоненте. После этого по вершине 1 генерируется компонента {1, 2}, а вершина 2 пропускается. Наконец, по вершинам 5 и 4 генерируются компоненты {5} и {4}.

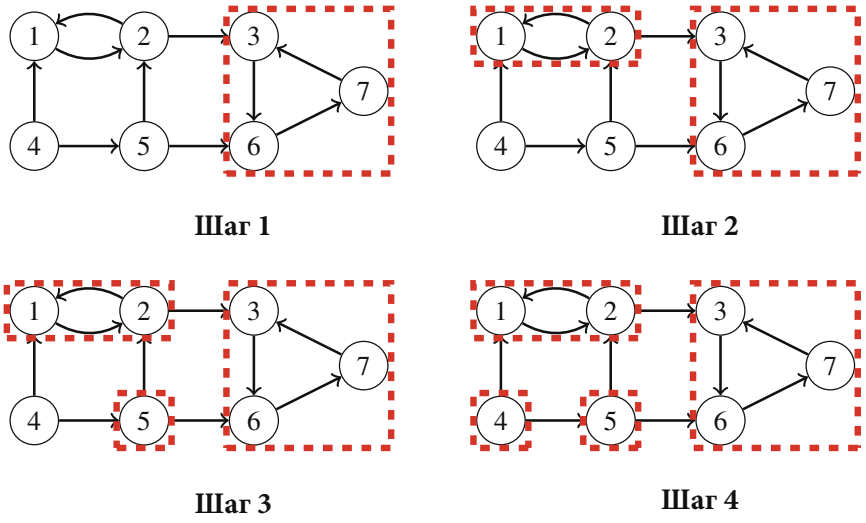


Рис. 12.5. Построение компонент сильной связности

Временная сложность алгоритма равна $O(n + m)$, поскольку выполняется два поиска в глубину.

12.1.2. *Задача 2-выполнимости*

В задаче *2-выполнимости*, или *2SAT*, дана логическая формула

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \ldots \wedge (a_m \vee b_m),$$

где a_i и b_i – либо логическая переменная (x_1, x_2, …, x_n), либо отрицание логической переменной($\neg x_1$, $\neg x_2$, …, $\neg x_n$). Символами «∧» и «∨» обозначаются логические операторы *И* и *ИЛИ*. Задача состоит в том, чтобы присвоить каждой переменной значение, так чтобы формула была истинной, или доказать, что это невозможно.

Например, формула

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

истинна, если переменным присвоены следующие значения:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}.$$

Однако формула

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

ложна при любых значениях переменных. Дело в том, что мы не можем выбрать значение x_1, не приводящее к противоречию. Если x_1 равно `false`, то x_2 и $\neg x_2$ должны быть равны `true`, что невозможно, а если x_1 равно `true`, то x_3 и $\neg x_3$ должны быть равны `true`, что также невозможно.

Задачу 2SAT можно представить *графом импликаций*, вершины которого соответствуют переменным x_i и отрицаниям $\neg x_i$, а ребра определяют связи между этими переменными. Каждая пара $(a_i \vee b_i)$ порождает два ребра: $\neg a_i \to b_i$ и $\neg b_i \to a_i$. Это означает, что если a_i не выполняется, то должно выполняться b_i, и наоборот. На рис. 12.6 показан граф импликаций для формулы L_1, а на рис. 12.7 – для формулы L_2.

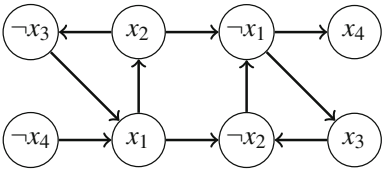


Рис. 12.6. Граф импликаций для L_1

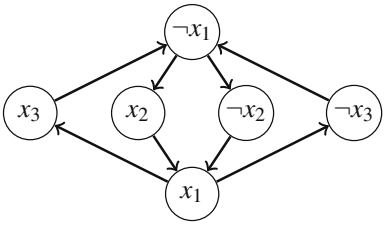


Рис. 12.7. Граф импликаций для L_2

Из структуры графа импликаций понятно, можно ли присвоить переменным значения, так чтобы формула была истинной. Это возможно тогда и только тогда, когда не существует вершин x_i и $\neg x_i$ – таких, что обе они принадлежат одной и той же компоненте сильной связности. Если такие вершины существуют, то граф содержит как путь из x_i в $\neg x_i$, так и путь из $\neg x_i$ в x_i , поэтому x_i и $\neg x_i$ должны быть равны `true` одновременно, что невозможно. В графе импликаций для L_1 не существует вершин x_i и $\neg x_i$, принадлежащих одной и той же компоненте сильной связности, поэтому задача имеет решение. Напротив, в графе импликаций для L_2 все вершины принадлежат одной компоненте сильной связности, так что решений нет.

Если решение существует, то значения переменных можно найти, обойдя вершины графа компонент в порядке обратной топологической сортировки. На каждом шаге обрабатывается компонента, которая не содержит ребер, ведущих в необработанную компоненту. Если переменным, попав-

шим в компоненту, еще не были присвоены значения, то их значения задаются в соответствии со значениями в компоненте, а ранее присвоенные значения не изменяются. Процесс продолжается до тех пор, пока каждой переменной не будет присвоено значение.

На рис. 12.8 показан граф компонентов для L_1. Компонентами являются $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ и $D = \{x_4\}$. При построении решения сначала обрабатывается компонента D, и x_4 получает значение `true`. После этого обрабатывается компонента C, в результате чего x_1 и x_2 получают значения `false`, а x_3 – значение `true`. Теперь всем переменным присвоены значения, поэтому обработка оставшихся компонент A и B ничего не изменит.

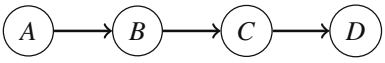


Рис. 12.8. Граф компонент для L_1

Этот метод работает в силу одной особенности структуры графа импликаций: если существует путь из вершины x_i в вершину x_j и из вершины x_j в вершину $\neg x_j$, то x_i никогда не получит значения `true`. Причина в том, что существует также путь из вершины $\neg x_j$ в вершину $\neg x_i$, и как x_i, так и x_j получают значение `false`.

Более трудна задача о *3-выполнимости* (*3SAT*), в которой каждая часть формулы имеет вид $(a_i \vee b_i \vee c_i)$. Она является NP-трудной, т. е. для нее неизвестен эффективный алгоритм решения.

12.2. Полные пути

В этом разделе мы рассмотрим два специальных типа путей в графах: эйлеров путь, проходящий по всем ребрам графа ровно один раз, и гамильтонов путь, заходящий в каждую вершину ровно один раз. На первый взгляд, эти пути очень похожи, но вычислительная сложность их нахождения разительно отличается.

12.2.1. *Эйлеровы пути*

Эйлеровым путем называется путь, который проходит по каждому ребру графа ровно один раз. Если такой путь начинается и заканчивается в одной и той же вершине, то он называется *эйлеровым циклом*. На рис. 12.9 показан эйлеров путь из вершины 2 в вершину 5, а на рис. 12.10 – эйлеров цикл, начинающийся в вершине 1.

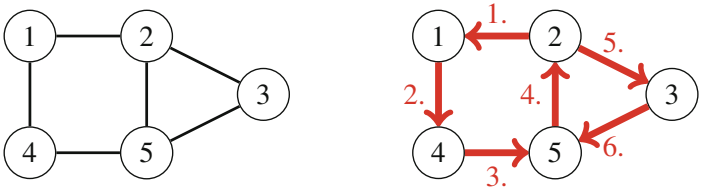


Рис. 12.9. Граф и его эйлеров путь

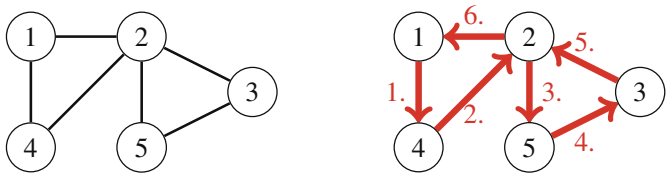


Рис. 12.10. Граф и его эйлеров цикл

Существование эйлеровых путей и циклов зависит от степеней вершин. В неориентированном графе эйлеров путь существует тогда и только тогда, когда все ребра принадлежат одной и той же компоненте связности и

- степень каждой вершины четна *или*
- существует ровно две вершины нечетной степени, а степени всех остальных четны.

В первом случае каждый эйлеров путь является эйлеровым циклом. Во втором случае конечными точками эйлерова пути являются вершины нечетной степени, и этот путь не является циклом. На рис. 12.9 вершины 1, 3 и 4 имеют степень 2, а вершины 2 и 5 – степень 3. Имеется ровно две вершины нечетной степени, поэтому между вершинами 2 и 5 существует эйлеров путь, не являющийся циклом. На рис. 12.10 степени всех вершин четны, поэтому в графе существует эйлеров цикл.

Чтобы понять, существуют ли эйлеровы пути в ориентированном графе, будем рассматривать полустепени захода и исхода. Ориентированный граф содержит эйлеров путь тогда и только тогда, когда все ребра принадлежат одной и той же компоненте сильной связности и

- полустепени захода и исхода каждой вершины равны *или*
- существует одна вершина, для которой полустепень захода на единицу больше полустепени исхода, еще одна вершина, для которой полустепень исхода на единицу больше полустепени захода, а во всех остальных вершинах полустепени захода и исхода равны.

В первом случае каждый эйлеров путь является эйлеровым циклом, а во втором случае в графе существует эйлеров путь, начинающийся в вершине, где полустепень исхода больше, и заканчивающийся в вершине, где полустепень захода больше. На рис. 12.11 у вершин 1, 3 и 4 полустепени захода и исхода равны 1, у вершины 2 полустепень захода равна 1, а полустепень исхода равна 2, и у вершины 5 полустепень захода равна 2, а полустепень исхода равна 1. Следовательно, граф содержит эйлеров путь из вершины 2 в вершину 5.

Построение. Эффективный способ построения эйлерова цикла графа дает *алгоритм Хирхольцера.* Он состоит из нескольких раундов, на каждом из которых в цикл добавляются новые ребра. Конечно, предполагается, что граф содержит эйлеров цикл, иначе алгоритм Хирхольцера не сможет его найти.

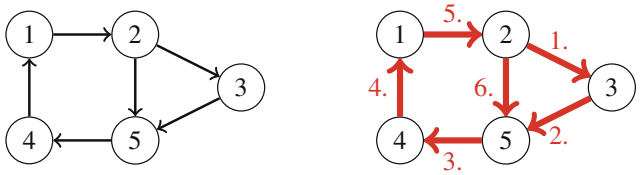


Рис. 12.11. Ориентированный граф и его эйлеров путь

Алгоритм начинает работу с пустого цикла, содержащего единственную вершину, а затем расширяет этот цикл, добавляя в него подциклы на каждом шаге. Процесс продолжается, пока в цикл не будут включены все ребра. Для расширения цикла мы находим в цикле вершину *x*, для которой существует исходящее ребро, не включенное в цикл. Затем строится новый путь из вершины *x*, содержащий только ребра, еще не включенные в цикл. Рано или поздно этот путь вернется в вершину *x*, с которой началось создание подцикла.

Если граф не содержит эйлерова цикла, но содержит эйлеров путь, то алгоритм Хирхольцера можно использовать для нахождения этого пути. Для этого нужно включить в граф дополнительное ребро и исключить его, после того как будет построен цикл. Например, в случае неориентированного графа дополнительное ребро проводится между двумя вершинами нечетной степени.

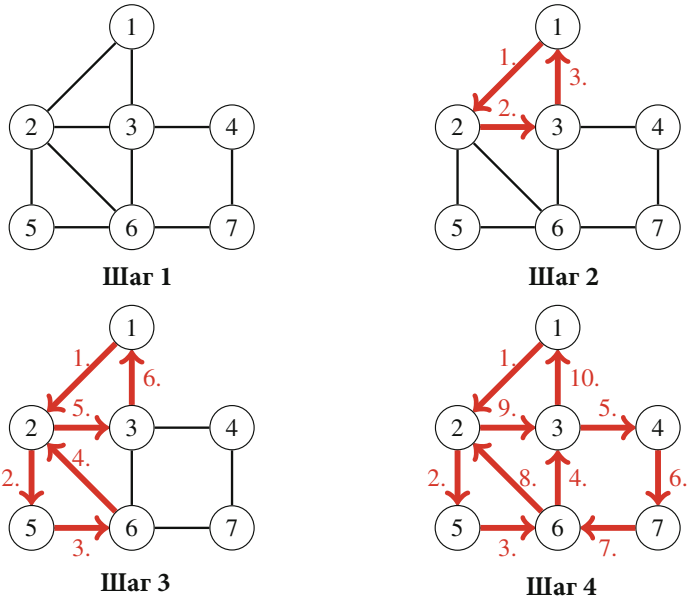


Рис. 12.12. Алгоритм Хирхольцера

На рис. 12.12 показано, как алгоритм Хирхольцера строит эйлеров путь в неориентированном графе. Сначала алгоритм добавляет подпуть 1 → 2 → 3 → 1, затем подпуть 2 → 5 → 6 → 2 и, наконец, подпуть 6 → 3 → 4 → 7 → 6.

В этот момент в цикл добавлены все ребра, так что мы успешно построили эйлеров цикл.

12.2.2. *Гамильтоновы пути*

Гамильтоновым путем называется путь, который заходит в каждую вершину графа ровно один раз. Если такой путь начинается и заканчивается в одной и той же вершине, то он называется *гамильтоновым циклом*. На рис. 12.13 показаны граф и его гамильтонов цикл.

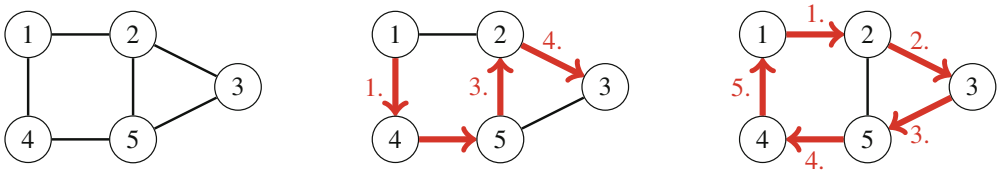


Рис. 12.13. Граф, гамильтонов путь и гамильтонов цикл

Задачи, относящиеся к гамильтоновым путям, являются NP-трудными: в общем случае неизвестно, как эффективно проверить, существует ли в графе гамильтонов путь или цикл. Конечно, в некоторых частных случаях гамильтонов путь заведомо существует. Например, если граф полный, т. е. между любой парой вершин имеется ребро, то гамильтонов путь есть наверняка.

Простой метод нахождения гамильтонова пути – алгоритм перебора с возвратом, который перебирает все возможные способы построения пути. Временная сложность такого алгоритма не меньше $O(n!)$, потому что выбрать порядок n вершин можно $n!$ разными способами. Но, воспользовавшись динамическим программированием, можно создать более эффективный алгоритм с временной сложностью $O(2^n n^2)$, который для каждого подмножества вершин S и каждой вершины $x \in S$ определяет, существует ли путь, заходящий в каждую вершину S ровно один раз и заканчивающийся в вершине x.

12.2.3. *Применения*

Последовательности де Брёйна. *Последовательностью де Брёйна* называется строка над фиксированным алфавитом из k символов, которая содержит в качестве подстроки любую строку длины n ровно один раз. Длина такой строки равна $k^n + n - 1$. При $n = 3$ и $k = 2$ примером последовательности де Брёйна может служить строка

0001011100.

Ее подстроками являются все комбинации трех бит: 000, 001, 010, 011, 100, 101, 110 и 111.

Последовательность де Брёйна соответствует эйлерову пути в графе, вершины которого содержат строки из $n - 1$ символов, а каждое ребро добавляет в строку один символ. Граф на рис. 12.14 соответствует случаю $n = 3$, $k = 2$. Чтобы создать последовательность де Брёйна, мы начинаем с произвольной вершины и, следуя эйлерову пути, проходим по каждому ребру ровно один раз. Сложив символы в начальной вершине с символами на ребрах, мы получим строку из $k^n + n - 1$ символов, которая является последовательностью де Брёйна.

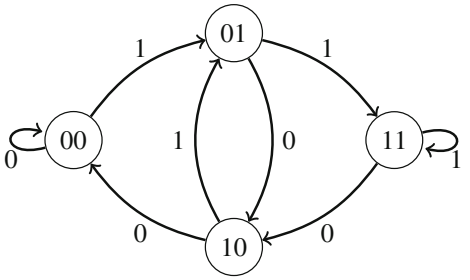


Рис. 12.14. Построение последовательности де Брёйна по эйлерову пути

Маршруты шахматного коня. *Маршрутом шахматного коня* называется такая последовательность ходов коня на доске $n \times n$ в соответствии с шахматными правилами, при которой каждое поле посещается ровно один раз. Маршрут называется *замкнутым*, если конь возвращается в исходное поле, и *незамкнутым* в противном случае. На рис. 12.15 показан незамкнутый маршрут шахматного коня.

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Рис. 12.15. Незамкнутый маршрут шахматного коня на доске 5×5

Маршрут шахматного коня соответствует гамильтонову пути в графе, вершины которого представляют поля доски, а две вершины соединены ребром, если конь может перейти из одного поля в другое за один ход. Для построения маршрута естественно применить перебор с возвратом. Поскольку количество возможных ходов велико, эффективность поиска можно повысить с помощью *эвристик*, т. е. попыток предложить такие ходы, при которых маршрут будет найден быстрее.

1				*a*
		2		
b				*e*
	c		*d*	

Рис. 12.16. Применение правила Варнсдорфа для построения маршрута коня

Правило Варнсдорфа – простая и эффективная эвристика для нахождения маршрута коня. Оно позволяет построить маршрут даже на большой доске. Идея в том, чтобы всегда ставить коня на поле, с которого можно пойти на *минимальное* число еще не пройденных полей. На рис. 12.16 конь может пойти

на любое из пяти полей (поля *a*...*e*). В этом случае правило Варнсдорфа требует, чтобы конь пошел на поле *a*, поскольку после этого будет возможен единственный ход. При любом другом выборе конь попал бы на поле, с которого можно пойти на три поля.

12.3. Максимальные потоки

В задаче о *максимальном потоке* дан ориентированный взвешенный граф, содержащий две специальные вершины: *источник*, в который не входит ни одно ребро, и *сток*, из которого не исходит ни одно ребро. Задача заключается в том, чтобы отправить максимальный поток из источника в сток. У каждого ребра имеется пропускная способность, и во всех промежуточных вершинах входящий и исходящий потоки должны быть равны.

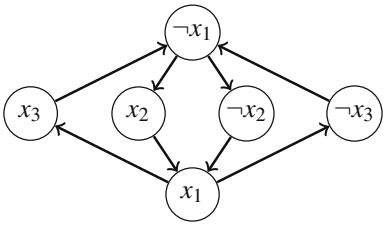


Рис. 12.17. Граф с источником 1 и стоком 6

В графе на рис. 12.17 вершина 1 является источником, а вершина 6 – стоком. Максимальный поток в этом графе равен 7, как показано на рис. 12.18. Нотация *v*/*k* означает, что через ребро с пропускной способностью *k* единиц проходит поток величиной *v* единиц. Величина потока равна 7, потому что источник отправляет 3 + 4 единиц потока, а сток получает 5 + 2 единиц. Легко видеть, что этот поток максимален, потому что суммарная пропускная способность ребер, входящих в сток, равна 7.

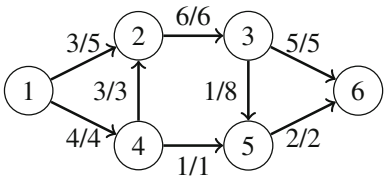


Рис. 12.18. Максимальный поток в графе равен 7

Оказывается, что задача о максимальном потоке связана с другой задачей на графах – задачей о *минимальном разрезе*, в которой наша цель – удалить из графа такое множество ребер, чтобы не осталось ни одного пути из источника в сток и при этом суммарная пропускная способность удаленных ребер была минимальна.

В графе на рис. 12.17 величина минимального разреза равна 7, поскольку достаточно удалить ребра 2 → 3 и 4 → 5, как показано на рис. 12.19. После удаления этих ребер путей из источника в сток не останется. Величина разреза равна 6 + 1 = 7, и этот разрез минимален, потому что не существует разреза с пропускной способностью меньше 7.

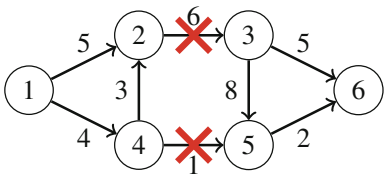


Рис. 12.19. Минимальный разрез в графе равен 7

То, что максимальный поток в нашем примере равен минимальному разрезу, – не случайность. На самом деле они равны

всегда, так что эти понятия – две стороны одной медали. Далее мы обсудим алгоритм Форда–Фалкерсона для нахождения максимального потока и минимального разреза. А заодно поймем, *почему* они равны.

12.3.1. *Алгоритм Форда–Фалкерсона*

Алгоритм Форда–Фалкерсона находит максимальный поток в графе. Вначале поток пуст, а на каждом шаге ищется путь от источника к стоку, на котором поток больше. Если алгоритм не может увеличить поток, значит, найденный к этому моменту поток максимален.

В алгоритме используется специальное представление графа, в котором для каждого исходного ребра добавлено *обратное* ребро между теми же вершинами, направленное противоположно. Вес ребра показывает, какой еще поток можно было бы пропустить через него. В начале работы алгоритма вес каждого исходного ребра равен его пропускной способности, а вес каждого обратного ребра равен нулю. На рис. 12.20 показано новое представление графа из нашего примера.

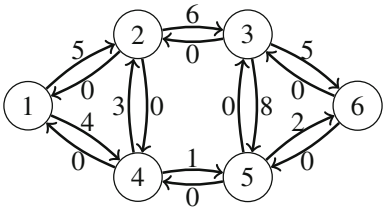


Рис. 12.20. Представление графа в алгоритме Форда–Фалкерсона

Алгоритм Форда–Фалкерсона состоит из нескольких раундов. На каждом раунде ищется путь от источника к стоку, для которого веса всех ребер положительны. Если таких путей несколько, можно выбрать произвольный. После выбора пути поток увеличивается на x единиц, где x – наименьший вес принадлежащего пути ребра. Одновременно вес каждого ребра на пути уменьшается на x, а вес каждого обратного ребра увеличивается на x.

Идея заключается в том, что увеличение потока уменьшает величину потока, который можно пропустить по ребрам в будущем. С другой стороны, обратные ребра позволяют впоследствии отменить поток, если окажется, что было бы выгоднее пустить поток по-другому. Алгоритм увеличивает поток, пока существуют пути от источника к стоку по ребрам с положительными весами. Если таких путей не осталось, алгоритм завершается – найденный поток максимален.

На рис. 12.21 показано, как алгоритм Форда–Фалкерсона находит максимальный поток в нашем графе. В данном случае понадобилось четыре раунда. На первом раунде выбирается путь $1 \to 2 \to 3 \to 5 \to 6$. Минимальный вес ребра на этом пути равен 2, поэтому поток увеличивается на 2 единицы. Затем алгоритм выбирает еще три пути, которые увеличивают поток на 3, 1 и 1 единицу. После этого путей с ребрами положительного веса не осталось, поэтому максимальный поток равен $2 + 3 + 1 + 1 = 7$.

Нахождение путей. Алгоритм Форда–Фалкерсона не говорит, как следует выбирать пути, увеличивающие поток. В любом случае[1], алгоритм рано или поздно закончится и правильно вычислит максимальный поток. Однако от способа выбора путей зависит его эффективность. Проще всего воспользоваться поиском в глубину. Обычно этот подход работает хорошо, но в худшем случае может оказаться, что каждый путь увеличивает поток только на одну единицу, и алгоритм будет работать медленно. По счастью, такой ситуации можно избежать, прибегнув к одному из описанных ниже методов.

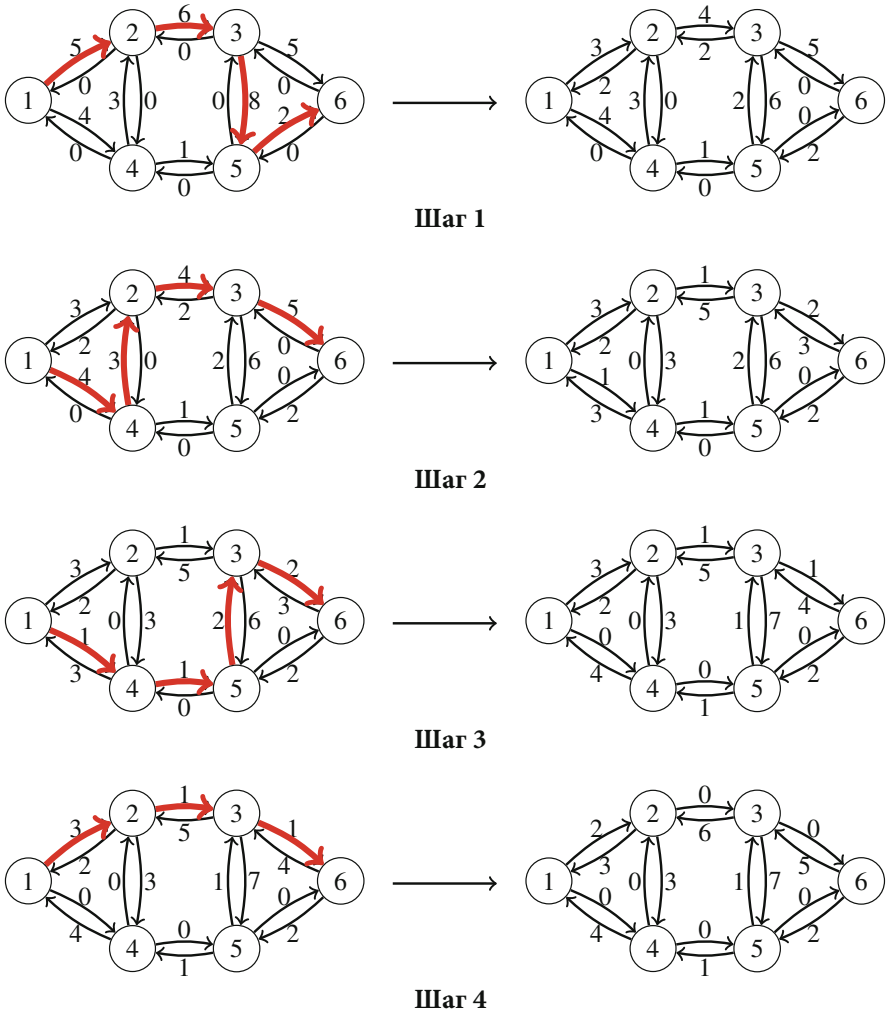


Рис. 12.21. Алгоритм Форда–Фалкерсона

[1] Утверждение верно только в случае, когда все пропускные способности целочисленные или рациональные – если это не так, то существуют примеры сетей и выбора путей в них, когда алгоритм не закончится никогда, а величина потока не будет сходиться к максимальной. – *Прим. ред.*

Алгоритм Эдмондса–Карпа выбирает путь с наименьшим числом ребер. Это можно сделать, применив для нахождения путей поиск в ширину вместо поиска в глубину. Можно доказать, что при этом поток возрастает быстро, а временная сложность равна $O(m^2n)$.

Алгоритм масштабирования пропускной способности[2] применяет поиск в глубину для нахождения путей, для которых вес каждого ребра не меньше некоторого целочисленного порога. Первоначально порог равен большому числу, например сумме весов всех ребер графа. Если путь найти не удается, величина порога делится на 2. Алгоритм завершается, когда величина порога станет равна 0. Временная сложность равна $O(m^2 \log c)$, где c – начальное значение порога.

На практике алгоритм масштабирования пропускной способности реализовать проще, потому что для нахождения путей применяется поиск в глубину. Оба алгоритма достаточно эффективны для решения задач, обычно предлагаемых на олимпиадах.

Минимальные разрезы. Оказывается, что одновременно с вычислением максимального потока алгоритм Форда–Фалкерсона находит и минимальный разрез. Рассмотрим граф, порожденный алгоритмом, и обозначим A множество вершин, достижимых из источника по ребрам положительного веса. Тогда минимальный разрез состоит из ребер исходного графа, которые начинаются в некоторой вершине, принадлежащей A, заканчиваются в некоторой вершине, не принадлежащей A, и таких, что их пропускная способность использована в максимальном потоке целиком. В графе на рис. 12.22 множество A состоит из вершин 1, 2 и 4, а в минимальный разрез входят ребра $2 \to 3$ и $4 \to 5$ с суммарным весом $6 + 1 = 7$.

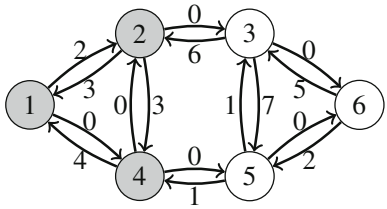


Рис. 12.22. Вершины 1, 2 и 3 принадлежат множеству A

Почему поток, вычисленный этим алгоритмом, максимален, а разрез минимален? Потому что в графе не может быть потока, величина которого больше величины любого разреза. Следовательно, если величины потока и разреза равны, то поток максимален, а разрез минимален.

Чтобы понять, почему это так, рассмотрим любой разрез графа – такой, что источник принадлежит A, сток принадлежит B и существуют ребра между этими множествами (рис. 12.23). Величина разре-

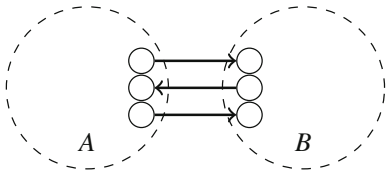


Рис. 12.23. Пропуск потока из A в B

[2] Этот элегантный алгоритм не слишком хорошо известен, его детальное описание можно найти в книге Ahuja, Magnanti, Orlin [1].

за равна сумме весов ребер, идущих из A в B. Это верхняя граница потока в графе, потому что поток должен течь из A в B. Следовательно, величина максимального потока меньше либо равна величине любого разреза графа. С другой стороны, алгоритм Форда–Фалкерсона находит поток, величина которого *в точности* равна величине некоторого разреза. Поэтому такой поток должен быть максимален, а разрез минимален.

12.3.2. *Непересекающиеся пути*

Многие задачи на графах можно решить, сведя их к задаче о максимальном потоке. В качестве первого примера рассмотрим следующую задачу: дан ориентированный граф с источником и стоком, и требуется найти максимальное число непересекающихся путей от источника к стоку.

Реберно не пересекающиеся пути. Сначала решим задачу о нахождении максимального числа *реберно не пересекающихся путей*, когда каждое ребро может встречаться не более чем в одном пути. В графе на рис. 12.24 максимальное число реберно не пересекающихся путей равно 2 ($1 \to 2 \to 4 \to 3 \to 6$ и $1 \to 4 \to 5 \to 6$).

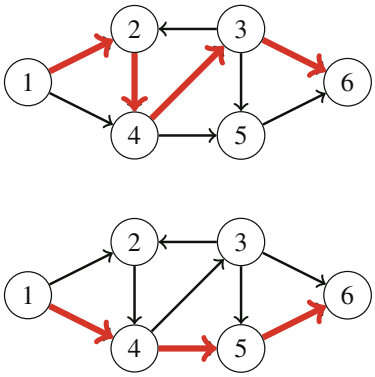


Рис. 12.24. Два реберно не пересекающихся пути из вершины 1 в вершину 6

Оказывается, что максимальное число реберно не пересекающихся путей всегда равно максимальному потоку в графе в случае, когда пропускная способность каждого ребра равна 1. После того как максимальный поток построен, реберно не пересекающиеся пути можно найти жадно, следуя по путям от источника к стоку.

Вершинно не пересекающиеся пути. Далее рассмотрим задачу о нахождении максимального числа *вершинно не пересекающихся путей* от источника к стоку. В этом случае каждая вершина, кроме источника и стока, должна встречаться не более чем в одном пути, вследствие чего максимальное число таких путей может оказаться меньше. Действительно, в

нашем примере графа максимальное число вершинно не пересекающихся путей равно 1 (рис. 12.25).

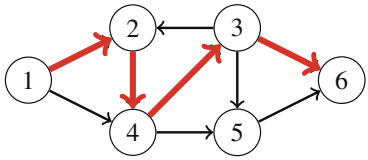


Рис. 12.25. Вершинно не пересекающийся путь из вершины 1 в вершину 6

Эту задачу тоже можно свести к задаче о максимальном потоке. Поскольку каждая вершина встречается самое большее в одном пути, мы должны ограничить поток, протекающий через вершины. Стандартный способ добиться этого – расщепить каждую вершину на две, так чтобы в первую входили те же ребра, что в исходную, а из второй исходили те же ребра, что из исходной. Кроме того, мы добавляем новое ребро, идущее из первой вершины во вторую. На рис. 12.26 показаны получившийся граф и максимальный поток в нем.

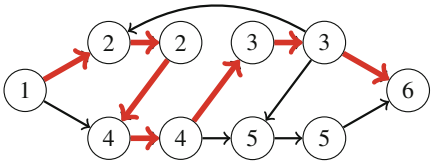


Рис. 12.26. Построение, ограничивающее поток через вершины

12.3.3. *Максимальные паросочетания*

Максимальным паросочетанием в графе называется множество максимального размера, состоящее из пар вершин, соединенных ребрами, и такое, что каждая вершина графа принадлежит не более чем одной паре. Для решения задачи о максимальном паросочетании в графе общего вида нужны хитроумные алгоритмы, но она становится гораздо проще, если предположить, что граф двудольный. В таком случае задачу можно свести к задаче о максимальном потоке.

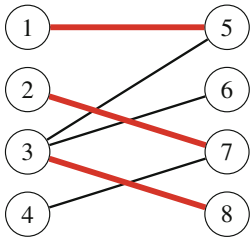


Рис. 12.27. Максимальное паросочетание

Вершины двудольного графа можно разбить на две группы, так что любое ребро соединяет какую-то вершину из левой группы с какой-то вершиной из правой. На рис. 12.27 показано максимальное паросочетание в двудольном графе с левой группой {1, 2, 3, 4} и правой группой {5, 6, 7, 8}.

Чтобы свести задачу к задаче о максимальном потоке, добавим в граф еще две вершины: источник и сток. Тогда величина максимального потока в получившемся графе будет равна размеру максимального паросочетания в исходном графе. На рис. 12.28 показаны это сведение и максимальный поток для нашего примера графа.

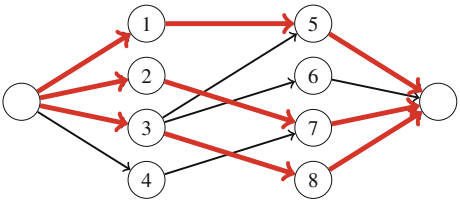


Рис. 12.28. Максимальное паросочетание как максимальный поток

Теорема Холла. Эту теорему можно использовать, чтобы узнать, существует ли в двудольном графе паросочетание, содержащее все левые или все правые вершины. Если количество левых и правых вершин одинаково, то теорема Холла дает ответ на вопрос, можно ли построить *совершенное паросочетание*, содержащее все вершины графа.

Пусть требуется найти паросочетание, содержащее все левые вершины. Обозначим X произвольное множество левых вершин, а $f(X)$ – множество их соседей. Согласно теореме Холла, паросочетание, содержащее все левые вершины, существует тогда и только тогда, когда для любого множества X имеет место неравенство $|X| \leqslant |f(X)|$.

Продемонстрируем теорему Холла на нашем примере. Сначала возьмем $X = \{1, 3\}$ и соответственно $f(X) = \{5, 6, 8\}$ (рис. 12.29). Для них условие теоремы Холла выполняется, потому что $|X| = 2$ и $|f(X)| = 3$. Теперь пусть $X = \{2, 4\}$ и $f(X) = \{7\}$ (рис. 12.30). Для них $|X| = 2$ и $|f(X)| = 1$, так что условие теоремы Холла не выполняется. Значит, для этого графа совершенного паросочетания не существует. Это и неудивительно, поскольку мы уже знаем, что размер максимального паросочетания для него равен 3, а не 4.

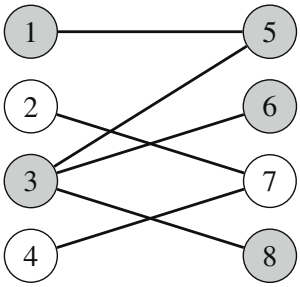


Рис. 12.29. $X = \{1, 3\}$ и $f(X) = \{5, 6, 8\}$

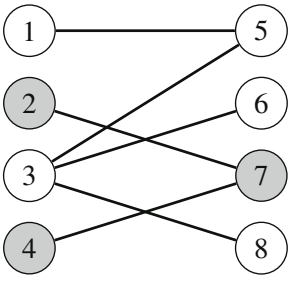


Рис. 12.30. $X = \{2, 4\}$ и $f(X) = \{7\}$

Если условие теоремы Холла не выполняется, то множество X объясняет, *почему* такое паросочетание невозможно. Поскольку X содержит больше вершин, чем $f(X)$, пары существуют не для всех вершин из X. Так, на рис. 12.30 обе вершины 2 и 4 следовало бы соединить с вершиной 7, что недопустимо.

Теорема Кёнига. *Минимальным вершинным покрытием* графа называется минимальное множество вершин – такое, что для любого ребра графа

хотя бы одна его вершина входит в это множество. В общем случае задача о нахождении минимального вершинного покрытия является NP-трудной. Но если граф двудольный, то по теореме Кёнига размер минимального вершинного покрытия равен размеру максимального паросочетания. А раз так, то мы можем вычислить этот размер с помощью алгоритма нахождения максимального потока.

Поскольку размер максимального паросочетания в нашем примере графа равен 3, то по теореме Кёнига размер минимального вершинного покрытия тоже равен 3. На рис. 12.31 показан пример такого покрытия.

Вершины, *не* принадлежащие минимальному вершинному покрытию, образуют *максимальное независимое множество*. Это максимальное множество вершин – такое, что никакие две принадлежащие ему вершины не соединены ребром. Задача о нахождении максимального независимого множества в графе общего вида также является NP-трудной, но для двудольного графа ее можно эффективно решить, пользуясь теоремой Кёнига. На рис. 12.32 показано максимальное независимое множество для нашего графа.

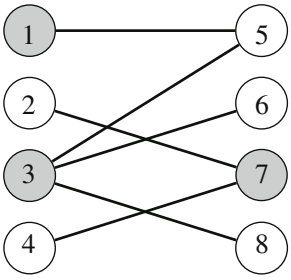


Рис. 12.31. Минимальное вершинное покрытие

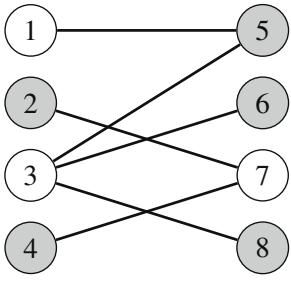


Рис. 12.32. Максимальное независимое множество

12.3.4. *Покрытие путями*

Покрытием путями называется такое множество путей, что любая вершина графа принадлежит, по крайней мере, одному пути. Оказывается, что для ориентированного ациклического графа задачу о нахождении минимального покрытия путями можно свести к задаче о нахождении максимального потока в другом графе.

Покрытия вершинно не пересекающимися путями. В случае покрытия *вершинно не пересекающимися* путями каждая вершина принадлежит ровно одному пути. Для примера рассмотрим граф на рис. 12.33. Его минимальное покрытие вершинно не пересекающимися путями состоит из трех путей (рис. 12.34).

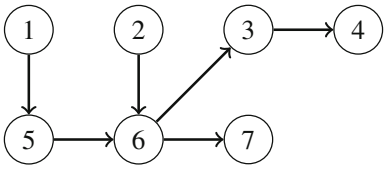


Рис. 12.33. Пример графа для построения покрытия путями

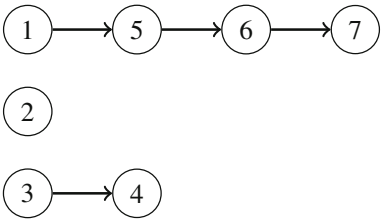


Рис. 12.34. Минимальное покрытие вершинно не пересекающимися путями

Чтобы найти минимальное покрытие вершинно не пересекающимися путями, построим *граф паросочетаний*, в котором каждая вершина исходного графа представлена двумя вершинами: левой и правой. Между вершиной слева и вершиной справа проведено ребро, если такое ребро существует в исходном графе. Кроме того, граф паросочетаний содержит источник и сток, причем источник соединен ребрами со всеми левыми вершинами, а все правые вершины соединены ребрами со стоком. Каждому ребру, принадлежащему максимальному паросочетанию в графе паросочетаний, соответствует ребро в минимальном покрытии вершинно не пересекающимися путями исходного графа. Следовательно, размер минимального покрытия вершинно не пересекающимися путями равен $n - c$, где n – количество вершин исходного графа, а c – размер максимального паросочетания.

На рис. 12.35 показан граф паросочетаний для графа на рис. 12.33. Размер максимального паросочетания в нем равен 4, поэтому минимальное покрытие вершинно не пересекающимися путями состоит из 7 – 4 = 3 путей.

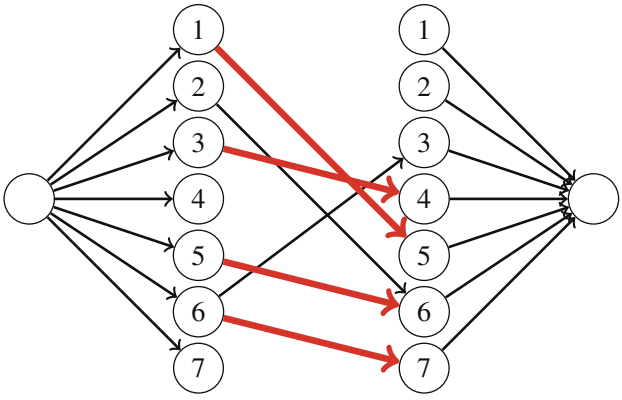


Рис. 12.35. Граф паросочетаний для нахождения минимального покрытия вершинно не пересекающимися путями

Покрытие общими путями. Так называется покрытие путями, в котором допускаются вершины, принадлежащие сразу нескольким путям. Ми-

нимальное покрытие общими путями может быть меньше минимального покрытия вершинно не пересекающимися путями, потому что одну вершину можно использовать в путях несколько раз. Снова рассмотрим граф на рис. 12.33. Для него минимальное покрытие общими путями состоит из двух путей (рис. 12.36).

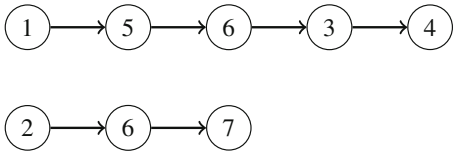


Рис. 12.36. Минимальное покрытие общими путями

Минимальное покрытие общими путями можно найти почти так же, как минимальное покрытие вершинно не пересекающимися путями. Достаточно добавить несколько ребер в граф паросочетаний, так чтобы ребро $a \rightarrow b$ существовало всегда, когда существует путь из a в b в исходном графе (возможно, проходящий через несколько вершин). На рис. 12.37 показан такой граф паросочетаний для нашего примера.

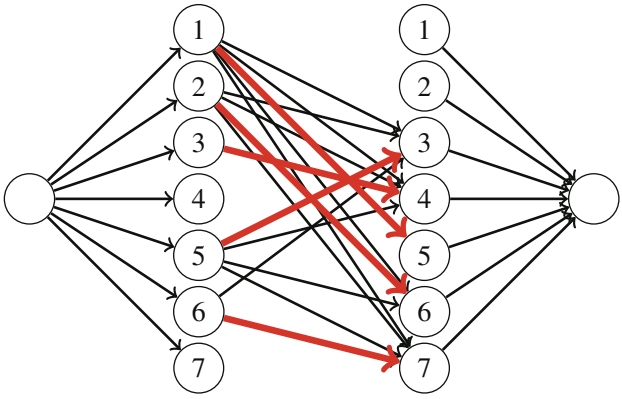


Рис. 12.37. Граф паросочетаний для нахождения минимального покрытия общими путями

Теорема Дилуорса. *Антицепью* называется множество вершин графа – такое, что между любыми двумя принадлежащими ему вершинами не существует пути, проходящего по ребрам графа. Теорема Дилуорса утверждает, что в ориентированном ациклическом графе размер минимального покрытия общими путями равен размеру максимальной антицепи. В графе на рис. 12.38 вершины 3 и 7 образуют антицепь из двух вершин. Это максимальная антицепь, потому что минимальное покрытие этого графа общими путями состоит из двух путей (рис. 12.36).

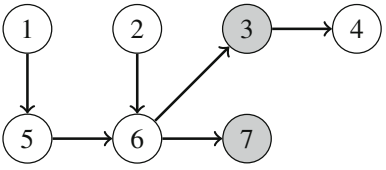


Рис. 12.38. Вершины 3 и 7 образуют максимальную антицепь

12.4. Деревья поиска в глубину

В процессе обработки связного графа поиск в глубину попутно создает корневое ориентированное *остовное дерево*, которое называется *деревом поиска в глубину*. Ребра графа можно классифицировать по их роли в поиске. В неориентированном графе могут встречаться ребра двух типов: *ребра дерева*, принадлежащие дереву поиска в глубину, и *обратные ребра*, ведущие в уже посещенные вершины. Отметим, что обратное ребро всегда ведет в предок вершины.

На рис. 12.39 показаны граф и его дерево поиска в глубину. Сплошными линиями изображены ребра дерева, а штриховыми – обратные ребра.

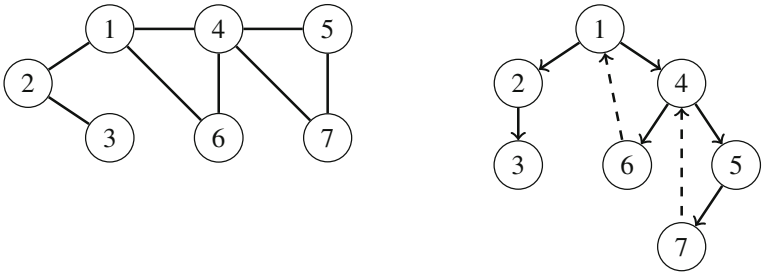


Рис. 12.39. Граф и его дерево поиска в глубину

В этом разделе мы обсудим некоторые применения деревьев поиска в глубину при обработке графов.

12.4.1. *Двусвязность*

Связный граф называется *двусвязным*, если он остается связным после удаления любой вершины (и инцидентных ей ребер). Левый граф на рис. 12.40 является двусвязным, а правый – нет, поскольку после удаления вершины 3 граф распадается на две компоненты: {1, 4} и {2, 5}.

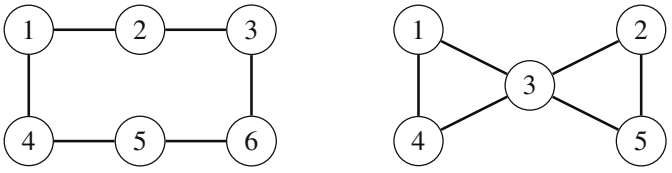


Рис. 12.40. Граф слева двусвязный, а граф справа – нет

Вершина называется *точкой сочленения*, если в результате ее удаления граф перестает быть связным. Следовательно, в двусвязном графе нет точек сочленения. Аналогично ребро называется *мостом*, если его удаление нарушает связность графа. На рис. 12.41 вершины 4, 5 и 7 являются точками сочленения, а ребра 4–5 и 7–8 – мостами.

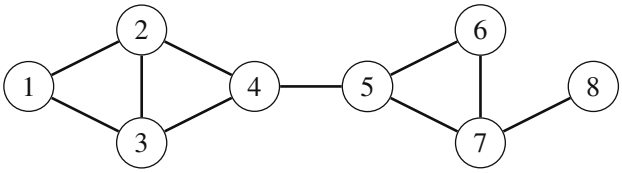


Рис. 12.41. Граф с тремя точками сочленения и двумя мостами

Для эффективного нахождения всех точек сочленения и мостов можно воспользоваться поиском в глубину. Чтобы найти мосты, мы начинаем поиск из произвольной вершины и строим дерево поиска в глубину. На рис. 12.42 показано дерево поиска в глубину для нашего графа.

Ребро $a \rightarrow b$ является мостом тогда и только тогда, когда это ребро дерева и не существует обратных ребер из поддерева b в a или в любой предок a. Так, на рис. 12.42 ребро $5 \rightarrow 4$ является мостом, потому что не существует обратных ребер из вершин $\{1, 2, 3, 4\}$ в вершину 5. Однако ребро $6 \rightarrow 7$ мостом не является, потому что существует обратное ребро $7 \rightarrow 5$, а вершина 5 является предком вершины 6.

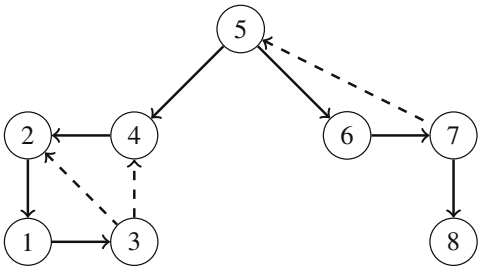


Рис. 12.42. Нахождение мостов и точек сочленения с помощью поиска в глубину

Найти точки сочленения несколько труднее, но дерево поиска в глубину поможет и в этом случае. Во-первых, если вершина x – корень дерева, то она является шарниром тогда и только тогда, когда имеет две или более дочерние вершины. Если же x – не корень, то она является точкой сочленения тогда и только тогда, когда имеет дочернюю вершину, поддерево которой не содержит обратного ребра в вершину-предок x.

В графе на рис. 12.42 вершина 5 является точкой сочленения, поскольку она корневая и имеет две дочерние вершины, а вершина 7 – точка сочленения, потому что дерево ее дочерней вершины 8 не содержит обратного ребра в вершину-предок 7. Однако вершина 2 не является точкой сочленения, потому что существует обратное ребро $3 \rightarrow 4$, а вершина 8 – не точка сочленения, потому что у нее вообще нет дочерних вершин.

12.4.2. *Эйлеровы подграфы*

Эйлеров подграф графа содержит все его вершины и часть ребер – такую, что степень каждой вершины четна. На рис. 12.43 показаны граф и его эйлеров подграф.

Рассмотрим задачу о вычислении количества эйлеровых подграфов связного графа. Оказывается, что существует простая формула: количество эйлеровых подграфов равно 2^k, где k – число обратных ребер в дереве поиска графа в глубину. Отметим, что $k = m - (n - 1)$, где n – число вершин, а m – число ребер.

Дерево поиска в глубину помогает понять, что эта формула справедлива. Рассмотрим произвольное фиксированное подмножество обратных ребер в дереве поиска в глубину. Чтобы создать эйлеров граф, содержащий эти ребра, мы должны выбрать подмножество ребер дерева, так чтобы степень каждой вершины была четна. Для этого мы будем обрабатывать дерево снизу вверх и включать ребро дерева в подграф тогда и только тогда, когда оно ведет в вершину, степень которой с учетом этого ребра станет четной. Тогда, поскольку сумма степеней вершин всегда четна, степень корневой вершины тоже будет четна.

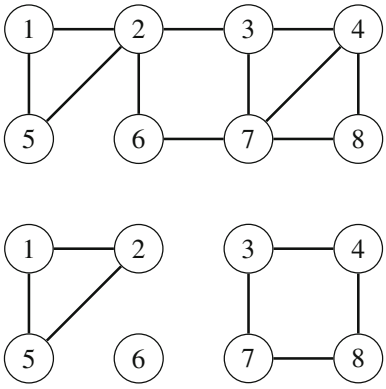


Рис. 12.43. Граф и его эйлеров подграф

12.5. Потоки минимальной стоимости

В задаче о *потоке минимальной стоимости* задан ориентированный граф с источником и стоком. С каждым ребром ассоциировано две величины: *пропускная способность*, т. е. максимальный поток, который можно пропустить по этому ребру, и *стоимость* – цена за единицу потока через это ребро. Задача заключается в том, чтобы пропустить k единиц потока из источника в сток, минимизировав полную стоимость потока.

Задача о потоке минимальной стоимости похожа на задачу о максимальном потоке (раздел 12.3), но имеет два отличия. Во-первых, мы хотим

пропустить ровно k единиц потока, даже если есть возможность пропустить больше. Во-вторых, у ребер есть стоимости, и требуется найти решение, минимизирующее полную стоимость потока.

Например, на рис. 12.44 показан граф, в котором вершина 1 является источником, а вершина 4 – стоком. Запись a; b означает, что пропускная способность ребра равна a, а его стоимость равна b. В частности, мы можем пропустить не более 5 единиц потока из вершины 2 в вершину 3, при этом цена за единицу потока будет равна 3. На рис. 12.45 показан оптимальный способ пропустить $k = 4$ единицы потока из источника в сток. Стоимость этого решения равна 29, и вычисляется она следующим образом:

- пропустить 2 единицы потока из вершины 1 в вершину 2 (стоимость $2 \cdot 1 = 2$);
- пропустить 1 единицу потока из вершины 2 в вершину 3 (стоимость $1 \cdot 3 = 3$);
- пропустить 1 единицу потока из вершины 2 в вершину 4 (стоимость $1 \cdot 8 = 8$);
- пропустить 2 единицы потока из вершины 1 в вершину 3 (стоимость $2 \cdot 5 = 10$);
- пропустить 3 единицы потока из вершины 3 в вершину 4 (стоимость $3 \cdot 2 = 6$).

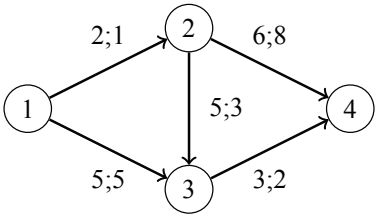


Рис. 12.44. Задача о потоке минимальной стоимости

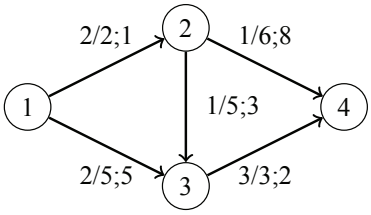


Рис. 12.45. Оптимальный способ пропустить 4 единицы потока

Заметим, что задача о потоке минимальной стоимости поставлена в очень общем виде, и у нее есть ряд частных случаев. Если требуется определить максимальное значение k, не обращая внимания на стоимости, то получаем задачу о максимальном потоке. А если пропускная способность каждого ребра бесконечна (или, по крайней мере, не меньше k), то задача сводится к нахождению пути минимальной стоимости из источника в сток.

12.5.1. *Алгоритм путей минимальной стоимости*

В предположении, что входной граф не содержит циклов, имеющих отрицательную стоимость, то задачу о потоке минимальной стоимости можно решить, воспользовавшись модифицированным вариантом алгоритма Форда–Фалкерсона (раздел 12.3). Как и в задаче о максимальном потоке,

мы строим пути, генерирующие поток из источника в сток. Оказывается, что если всегда выбирать путь с минимальной полной стоимостью, то получающийся поток будет оптимальным решением задачи о потоке минимальной стоимости [9].

Чтобы воспользоваться алгоритмом Форда–Фалкерсона, мы сначала для каждого ребра добавим ребро, идущее в противоположном направлении и имеющее пропускную способность 0 и стоимость $-c$, где c – стоимость оригинального ребра[3]. В процессе работы алгоритма стоимости ребер не изменяются. Затем мы выполняем алгоритм Форда–Фалкерсона и всегда выбираем путь минимальной стоимости из источника в сток. Мы увеличиваем поток и обновляем пропускные способности, как в задаче о максимальном потоке, со следующим исключением: если текущий поток равен f и путь привел бы к его увеличению на x, где $f + x > k$, то мы увеличиваем поток только на $k - f$ и сразу же завершаем работу.

Хотя в графе нет циклов отрицательной стоимости, в нем могут быть ребра отрицательной стоимости. Поэтому мы строим пути минимальной стоимости, применяя алгоритм Беллмана–Форда, поддерживающий отрицательные стоимости ребер. Результирующий алгоритм работает за время $O(nmk)$, потому что каждый путь увеличивает поток по крайней мере на единицы, а для нахождения пути с помощью алгоритма Беллмана–Форда требуется время $O(nm)$.

На рис. 12.46 показано, как работает алгоритм для нашего графа в предположении, что требуемый поток $k = 4$. Сначала строится путь $1 \to 2 \to 3 \to 4$, стоимость которого равна $1 + 3 + 2 = 6$. Этот путь увеличивает сам поток на 2, а его стоимость – на $2 \cdot 6 = 12$. Затем алгоритм строит путь $1 \to 3 \to 4$, который увеличивает поток 1, а его стоимость – на 7. Наконец, алгоритм строит путь $1 \to 3 \to 2 \to 4$, который увеличивает поток 1, а его стоимость – на 10. Заметим, что последний путь мог бы увеличить поток на 2, но увеличивает только на 1, потому что требуемый поток равен 4. Полная стоимость решения равна $12 + 7 + 10 = 29$, как мы и ожидали.

Почему этот алгоритм работает? Он основан на факте, который мы здесь доказывать не будем: если в графе (с добавленными противоположными ребрами) имеется поток величины f и не существует цикла с отрицательной стоимостью, в котором каждое ребро имеет положительную пропускную способность, то этот поток является потоком величины f минимальной стоимости.

Мы знаем, что в исходном графе нет циклов с отрицательной стоимостью, а поскольку мы всегда строим путь минимальной стоимости из источника в сток, гарантируется, что цикл с отрицательной стоимостью никогда и не возникнет. Поэтому, коль скоро мы можем построить поток

[3] Если существует как ребро из a в b, так и ребро из b в a, то необходимо добавить противоположное ребро для каждого из них. Таким образом, мы не можем комбинировать ребра, как в задаче о максимальном потоке, потому что ребра имеют стоимость, которую нужно принимать во внимание.

величины k, не создавая циклов с отрицательной стоимостью, результирующий поток обязан быть потоком минимальной стоимости величины k.

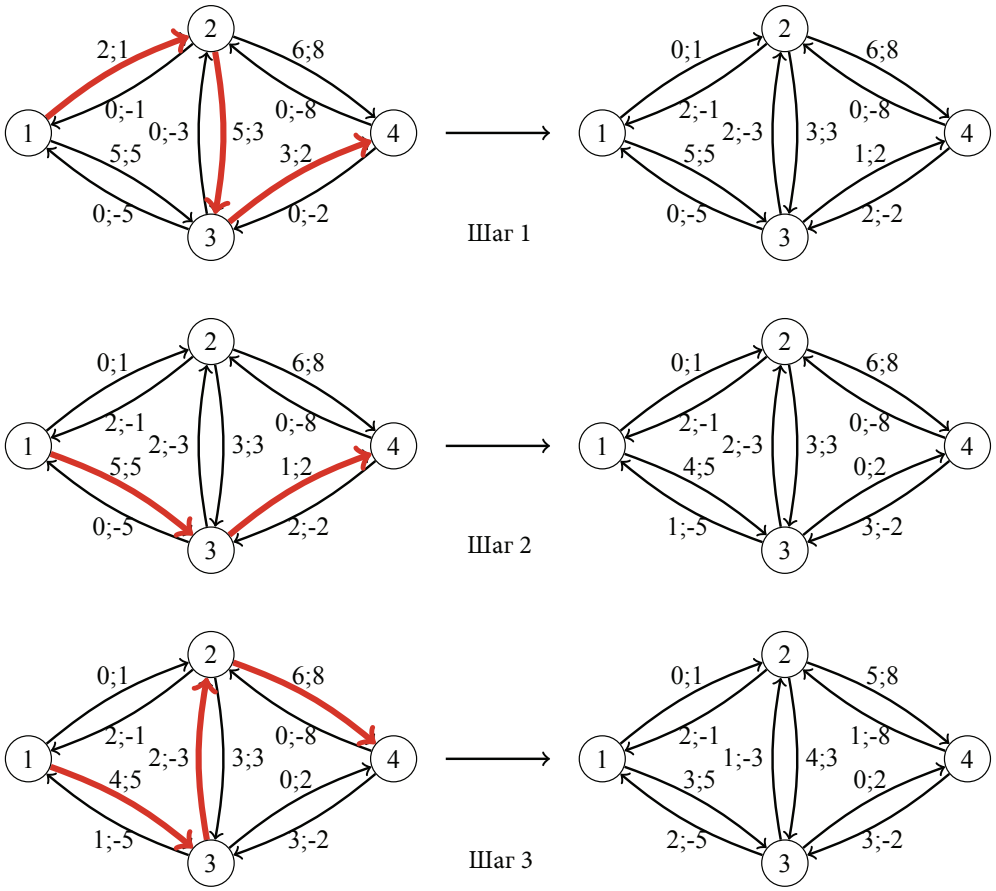


Рис. 12.46. Вычисление потока минимальной стоимости (k = 4) с помощью алгоритма путей минимальной стоимости

12.5.2. *Паросочетания минимального веса*

Одно из приложений потоков минимальной стоимости – решение задачи о *паросочетаниях минимального веса в двудольном графе*. Пусть дан взвешенный двудольный граф, требуется найти паросочетание величины k с минимальным полным весом. Эта задача является обобщением задачи о максимальном паросочетании в двудольном графе, и решить ее можно аналогично, воспользовавшись алгоритмом потока минимальной стоимости.

Например, предположим, что в компании имеется n работников и n задач, что каждому работнику можно назначить ровно одну задачу и что для каждого работника известна стоимость выполнения им каждой зада-

чи. Какова минимальная полная стоимость при оптимальном назначении работ? Например, для следующих входных данных

Работник	Задача 1	Задача 2	Задача 3
Анна	150	400	200
Джон	400	350	200
Мария	500	100	250

оптимальное решение таково: назначить задачу 1 Анне, задачу 2 – Марии и задачу 3 – Джону. Полная стоимость этого решения равна 150 + 100 + 200 = 450.

На рис. 12.47 показано, как эту задачу можно представить в виде задачи о потоке минимальной стоимости. Создаем граф с $2n + 2$ вершинами: источник, сток и по одной вершине для каждого работника и каждой задачи. Пропускная способность каждого ребра равна 1, стоимость каждого ребра, исходящего из источника или входящего в сток, равна 0, а стоимость ребра, соединяющего работника с задачей, равна стоимости назначения этой задачи данному работнику. Тогда поток минимальной стоимости величины n в этом графе соответствует оптимальному решению.

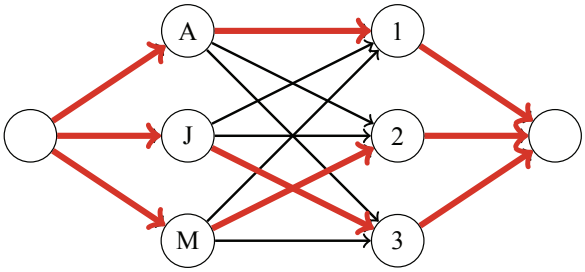


Рис. 12.47. Нахождение оптимального назначения путем представления паросочетания минимального веса в виде потока минимальной стоимости

12.5.3. *Улучшение алгоритма*

Если бы мы знали, что в графе, который используется в алгоритме путей минимальной стоимости, нет ребер с отрицательной стоимостью и положительной пропускной способностью, то могли бы улучшить алгоритм, воспользовавшись алгоритмом Дейкстры вместо алгоритма Беллмана–Форда. Оказывается, что это можно сделать, *модифицировав* граф, так что в нем не будет ребер с отрицательной стоимостью и положительной пропускной способностью, но в то же время каждому пути минимальной стоимости в новом графе будет соответствовать путь минимальной стоимости в исходном графе.

Мы применим следующий прием, который используется также в алгоритме Джонсона [18]. Предположим, что каждой вершине x сопоставлено

значение $p[x]$, которое может быть любым числом. Тогда можно модифицировать граф, так что стоимость ребра из вершины a в вершину b станет равна $c(a, b) + p[a] - p[b]$, где $c(a, b)$ – его первоначальная стоимость. При такой модификации ни один путь минимальной стоимости не изменяется: если стоимость пути из вершины x в вершину y в исходном графе равна k, то стоимость того же пути в новом графе равна $k + p[x] - p[y]$, где $p[x] - p[y]$ постоянно для всех путей из x в y. Это так, потому что значения p в промежуточных вершинах пути взаимно уничтожаются.

Идея заключается в том, чтобы выбрать значения p таким образом, чтобы после модификации не осталось ребер отрицательной стоимости. Для этого достаточно положить $p[x]$ равным минимальной стоимости пути из источника в вершину x. После этого для любого ребра из a в b имеем

$$p[b] \leqslant p[a] + c(a, b),$$

а это означает, что

$$c(a, b) + p[a] - p[b] \geqslant 0,$$

т. е. новая стоимость ребра неотрицательна.

Теперь можно реализовать алгоритм путей минимальной стоимости следующим образом. Сначала выполняем один раз алгоритм Беллмана–Форда, начиная из источника, и строим пути минимальной стоимости во все вершины, которых можно достичь по ребрам с положительной пропускной способностью. Затем модифицируем стоимости ребер, выбирая значения p так, чтобы все ребра с положительной пропускной способностью получили неотрицательную стоимость. После этого запускаем основной алгоритм, который генерирует поток и использует алгоритм Дейкстры для нахождения путем минимальной стоимости. Мы всегда строим пути минимальной стоимости во все вершины, которых можно достичь по ребрам с положительной пропускной способностью, а потом обновляем стоимости ребер в соответствии со значениями p. Затем используем первоначальные стоимости ребер при вычислении стоимости нового пути. Итоговый алгоритм работает за время $O(nm + k(m \log n))$, поскольку мы один раз выполнили алгоритм Беллмана–Форда и не более k раз алгоритм Дейкстры.

На рис. 12.48 показано, как улучшенный алгоритм находит поток минимальной стоимости в нашем графе. Мы уже модифицировали начальные стоимости ребер с помощью алгоритма Беллмана–Форда, а теперь три раза выполнили алгоритм Дейкстры для построения путей минимальной стоимости. Стоимость каждого ребра с положительной пропускной способностью неотрицательна, поэтому алгоритм Дейкстры работает правильно. Заметим, что каждый путь соответствует пути на рис. 12.46; различаются только стоимости ребер, поэтому мы должны использовать исходные стоимости при вычислении стоимости результирующего потока.

В первый раз мы используем алгоритм Беллмана–Форда, потому что в исходном графе могут существовать ребра с положительной пропускной способностью и отрицательной стоимостью. Но после этого мы уверены, что таких ребер нет, поэтому можем использовать алгоритм Дейкстры. Отметим, что пропускные способности некоторых ребер изменяются, когда поток увеличивается после построения пути, но это никогда не приводит к появлению ребер отрицательной стоимости, потому что все такие ребра принадлежат пути минимальной стоимости из источника в сток: если путь ведет из вершины a в вершину b, то мы знаем, что $p[b] = p[a] + c(a, b)$, а это означает, что как новая стоимость пути из a в b, так и новая стоимость пути из b в a будут равны 0.

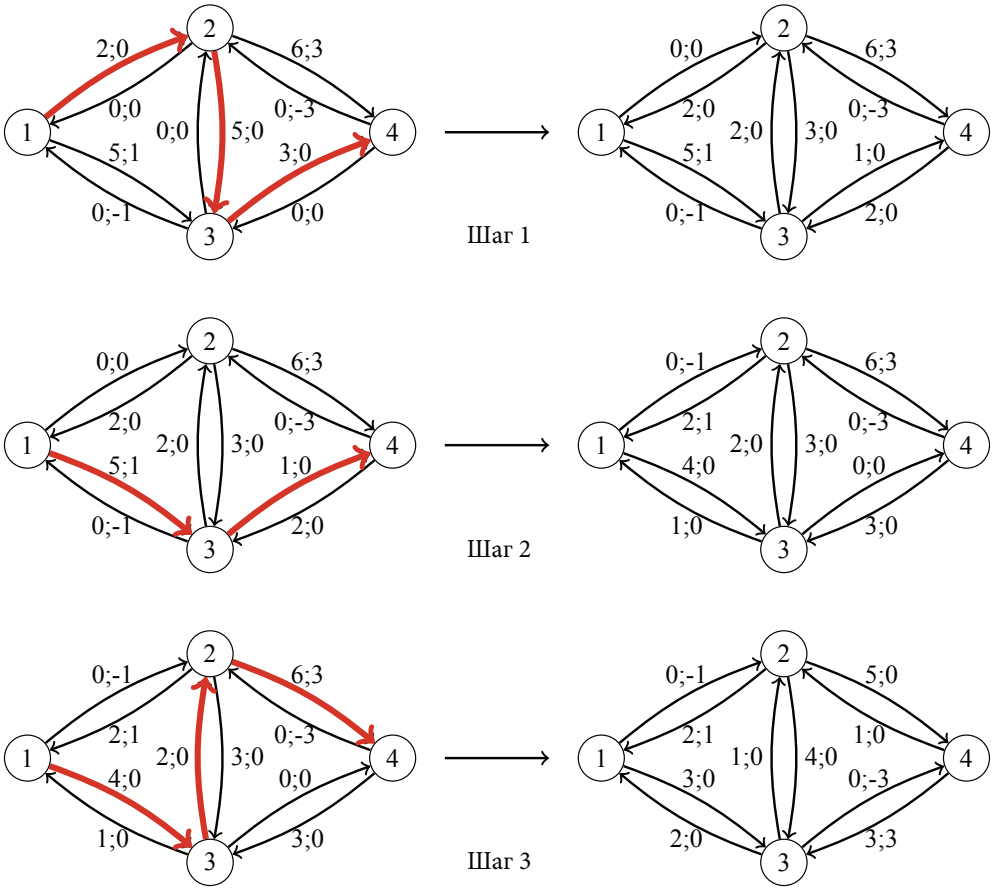


Рис. 12.48. Нахождение потока минимальной стоимости ($k = 4$) с помощью улучшенного алгоритма

При практической реализации улучшенного алгоритма путей минимальной стоимости необязательно модифицировать стоимости ребер. Можно просто прибавлять и вычитать значения p в момент построения путей, а затем обновлять значения p после каждого раунда.

Глава 13

Геометрия

В этой главе обсуждаются алгоритмы, относящиеся к геометрии. Наша цель – найти способы, позволяющие *удобно* решать геометрические задачи, избегая особых случаев и корявых реализаций.

В разделе 13.1 мы познакомимся с классом комплексных чисел из стандартной библиотеки C++, в котором имеются полезные средства для решения геометрических задач. Затем мы узнаем о применении векторного произведения для решения таких задач, как проверка пересечения двух отрезков и вычисление расстояния от точки до прямой. Наконец, мы обсудим различные способы вычисления площади многоугольников и специальные свойства манхэттенского расстояния.

Раздел 13.2 посвящен алгоритмам на основе заметающей прямой, которые играют важную роль в вычислительной геометрии. Мы увидим, как такие алгоритмы используются для подсчета точек пересечения, нахождения ближайших точек и построения выпуклой оболочки.

13.1. Технические средства в геометрии

При решении геометрических задач возникает общая проблема: найти подход, позволяющий свести количество особых случаев к минимуму, и придумать удобный способ реализации решения. В этом разделе мы рассмотрим ряд инструментов, упрощающих решение геометрических задач.

13.1.1. *Комплексные числа*

Комплексное число имеет вид $x + yi$, где $i = \sqrt{-1}$ – так называемая *мнимая единица*. Геометрически комплексное число интерпретируется как точка на двумерной плоскости (x, y) или как вектор, соединяющий начало координат с точкой (x, y). На рис. 13.1 изображено комплексное число $4 + 2i$.

В стандартной библиотеке C++ имеется класс `complex` для представления комплексных чисел, он полезен при решении геометрических задач. С помощью этого класса мы можем представлять точки и векторы как комплексные числа и использовать методы класса для операций над этими объектами. Для этого сначала определим тип координат `C`. В зависимости от ситуации это может быть тип `long long` или `long double`. Вообще го-

воря, лучше по возможности работать с целочисленными координатами, поскольку вычисления в этом случае точны.

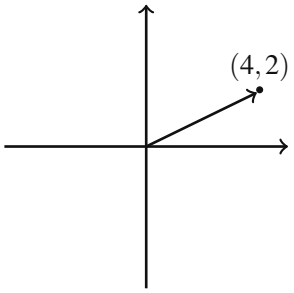


Рис. 13.1. Комплексное число 4 + 2*i*, интерпретируемое как точка и как вектор

Вот возможные определения типа координат:

```
typedef long long C;

typedef long double C;
```

Затем определим тип комплексного числа `P`, представляющий точку или вектор:

```
typedef complex<C> P;
```

Наконец, определим макросы для ссылки на координаты *x* и *y*:

```
#define X real()
#define Y imag()
```

Например, в следующем фрагменте создается точка $p = (4, 2)$ и печатаются ее координаты *x* и *y*:

```
P p = {4,2};
cout << p.X << "" << p.Y << "\n"; // 4 2
```

А ниже создаются векторы $v = (3, 1)$ и $u = (2, 2)$, после чего вычисляется их сумма $s = v + u$.

```
P v = {3,1};
P u = {2,2};
P s = v+u;
cout << s.X << "" << s.Y << "\n"; // 5 3
```

Функции. В классе `complex` имеются также функции, полезные для решения геометрических задач. Описанные ниже функции следует использовать, только когда тип координат `long double` (или другой тип с плавающей точкой).

Функция `abs(v)` вычисляет длину $|v|$ вектора $v = (x, y)$ по формуле $\sqrt{x^2+y^2}$. Ее можно также использовать для вычисления расстояния между точками (x_1, y_1) и (x_2, y_2), поскольку это расстояние равно длине вектора $(x_2 - x_1, y_2 - y_1)$. В следующем коде вычисляется расстояние между точками (4, 2) и (3, –1).

```
P a = {4,2};
P b = {3,-1};
cout << abs(b-a) << "\n"; // 3.16228
```

Функция `arg(v)` вычисляет *полярный угол* между вектором $v = (x, y)$ и положительным направлением оси x, т. е. для вектора, направленного вправо, угол равен нулю и возрастает против часовой стрелки. Функция возвращает угол в радианах (r радиан равно $180r/\pi$ градусов).

Функция `polar(s, a)` конструирует вектор длины s, наклоненный к оси x под углом a, выраженным в радианах. Для поворота вектора на угол a его нужно умножить на вектор длины 1 с полярным углом a.

В следующем коде вычисляется полярный угол вектора (4, 2), затем вектор поворачивается на 1/2 радиана, и снова вычисляется полярный угол:

```
P v = {4,2};
cout << arg(v) << "\n"; // 0.463648
v *= polar(1.0,0.5);
cout << arg(v) << "\n"; // 0.963648
```

13.1.2. *Точки и прямые*

Векторное произведение $a \times b$ векторов $a = (x_1, y_1)$ и $b = (x_2, y_2)$ определяется как $x_1y_2 - x_2y_1$. Оно дает направление вектора b относительно вектора a. На рис. 13.2 показаны три возможных случая:

- $a \times b > 0$: b повернут относительно a влево;
- $a \times b = 0$: направления a и b совпадают или противоположны;
- $a \times b < 0$: b повернут относительно a вправо.

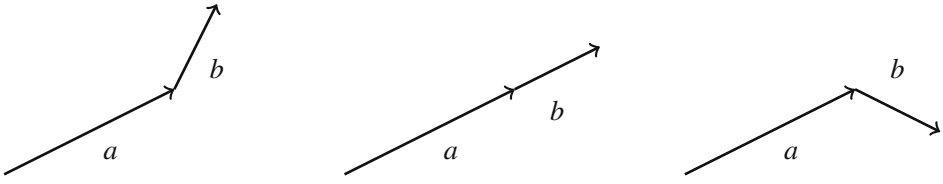


Рис. 13.2. Интерпретация векторного произведения

Например, векторное произведение векторов $a = (4, 2)$ и $b = (1, 2)$ равно $4 \cdot 2 - 2 \cdot 1 = 6$, что соответствует первому случаю на рис. 13.2. Ниже показано, как вычисляется векторное произведение:

```
P a = {4,2};
P b = {1,2};
C p = (conj(a)*b).Y; // 6
```

Этот код работает, потому что функция `conj` меняет знак координаты y вектора, а координата y произведения векторов $(x_1, -y_1)$ и (x_2, y_2) равна $x_1y_2 - x_2y_1$.

Рассмотрим некоторые применения векторного произведения.

Проверка положения точки относительно прямой. Векторное произведение позволяет узнать, как расположена точка относительно прямой: слева или справа. Предположим, что прямая проходит через точки s_1 и s_2, что мы смотрим из s_1 в s_2 и что дана точка p. Так, на рис. 13.3 точка p расположена слева от прямой.

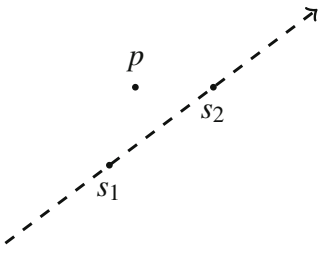


Рис. 13.3. Проверка положения точки относительно прямой

Положение точки p можно узнать, вычислив векторное произведение $(p - s_1) \times (p - s_2)$. Если оно положительно, то p расположена слева от прямой, если отрицательно, то справа, а если равно нулю, то точки s_1, s_2 и p лежат на одной прямой.

Пересечение отрезков. Предположим теперь, что требуется узнать, пересекаются ли два отрезка ab и cd. Если отрезки пересекаются, то возможны три случая.

Случай 1. Отрезки лежат на одной прямой и частично перекрываются. В этом случае количество точек пересечения бесконечно. Так, на рис. 13.4 все точки между c и b являются точками пересечения. Чтобы распознать этот случай, можно воспользоваться векторным произведением и проверить, лежат ли все четыре точки на одной прямой. Если да, то отсортируем их и проверим, перекрываются ли отрезки.

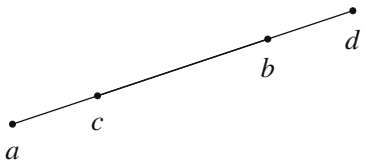


Рис. 13.4. Случай 1: отрезки лежат на одной прямой и перекрываются

Случай 2. Отрезки имеют общий конец, который является единственной точкой пересечения. На рис. 13.5 точкой пересечения является $b = c$. Этот случай легко проверить, т. к. всего существует четыре возможные точки пересечения: $a = c$, $a = d$, $b = c$ и $b = d$.

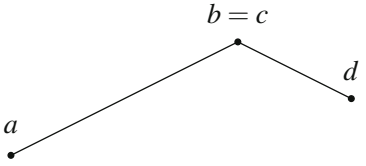


Рис. 13.5. Случай 2: отрезки имеют общий конец

Случай 3. Существует одна точка пересечения, не являющаяся общим концом отрезков. На рис. 13.6 p – точка пересечения. В этом случае отрезки пересекаются тогда и только тогда, когда точки c и d лежат по разные стороны от прямой, проходящей через a и b, а точки a и b – по разные стороны от прямой, проходящей через c и d. Это можно проверить, вычислив векторные произведения.

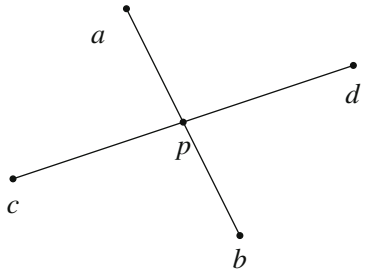


Рис. 13.6. Случай 3: отрезки пересекаются в точке, не являющейся концом

Расстояние от точки до прямой. Еще одно свойство векторного произведения состоит в том, что площадь треугольника можно вычислить по формуле

$$\frac{|(a-c)\times(b-c)|}{2},$$

где a, b и c – вершины треугольника. Зная это, мы можем вывести формулу для вычисления кратчайшего расстояния между точкой и прямой. На рис. 13.7 буквой d обозначено кратчайшее расстояние между точкой p и прямой, проходящей через точки s_1 и s_2.

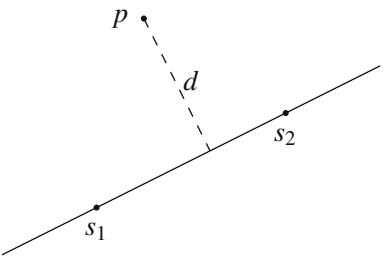


Рис. 13.7. Вычисление расстояния от точки p до прямой

Площадь треугольника с вершинами s_1, s_2 и p можно вычислить двумя способами: $\frac{1}{2}|s_2 - s_1|d$ (стандартная школьная формула) и $\frac{1}{2}((s_1 - p)\times(s_2 - p))$ (формула с применением векторного произведения). Следовательно, кратчайшее расстояние равно

$$d = \frac{(s_1 - p)\times(s_2 - p)}{|s_2 - s_1|}.$$

Точка внутри многоугольника. Наконец, обсудим, как узнать, находится ли точка внутри или вне многоугольника. На рис. 13.8 точка a находится внутри, а точка b – вне многоугольника.

Для решения этой задачи удобно провести *луч* из точки в произвольном направлении и посчитать, сколько раз он пересекает границу многоугольника. Если это число нечетное, то точка находится внутри, а если четное, то вне.

Так, на рис. 13.9 лучи, проведенные из точки a, пересекают границу многоугольника 1 и 3 раза, поэтому a находится внутри. Аналогично лучи, проведенные из точки b, пересекают границу 0 и 2 раза, поэтому b находится вне многоугольника.

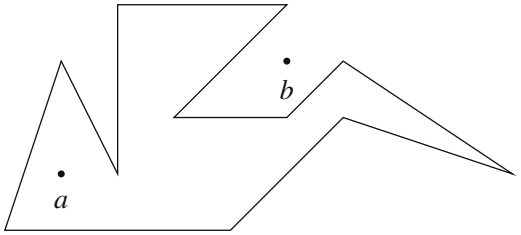


Рис. 13.8. Точка *a* находится внутри многоугольника, а точка *b* вне его

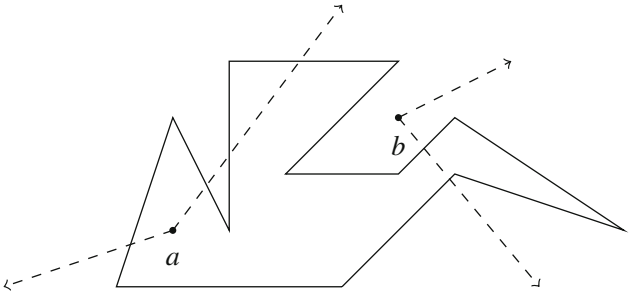


Рис. 13.9. Проведение лучей из точек *a* и *b*

13.1.3. *Площадь многоугольника*

Общая формула вычисления площади многоугольника иногда называется *формулой шнурования*:

$$\frac{1}{2}\left|\sum_{i=1}^{n-1}(p_i \times p_{i+1})\right| = \frac{1}{2}\left|\sum_{i=1}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)\right|.$$

Здесь вершины многоугольника $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ упорядочены таким образом, что p_i и p_{i+1} – соседние вершины, а первая вершина совпадает с последней, т. е. $p_1 = p_n$.

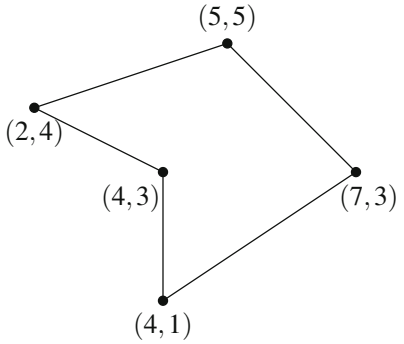


Рис. 13.10. Многоугольник площади 17/2

Например, площадь многоугольника на рис. 13.10 равна

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

Для вывода этой формулы многоугольник покрывается трапециями, одна сторона которых совпадает со стороной многоугольника, а другая лежит на горизонтальной оси $y = 0$. На рис. 13.11 показана одна такая трапеция.

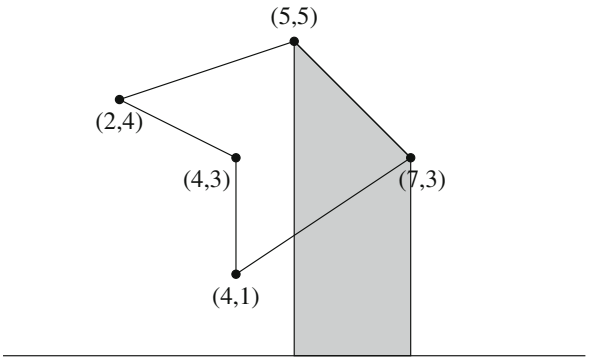


Рис. 13.11. Вычисление площади многоугольника методом трапеций

Площадь каждой трапеции равна

$$(x_{i+1} - x_i)\frac{y_i + y_{i+1}}{2},$$

где (x_i, y_i) и (x_{i+1}, y_{i+1}) – координаты вершин p_i и p_{i+1} многоугольника. Если $x_{i+1} > x_i$, то площадь положительна, а если $x_{i+1} < x_i$, то отрицательная. Просуммировав площади всех трапеций, получим формулу площади многоугольника:

$$\left|\sum_{i=1}^{n-1}(x_{i+1} - x_i)\frac{y_i + y_{i+1}}{2}\right| = \frac{1}{2}\left|\sum_{i=1}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)\right|.$$

Мы берем абсолютную величину, потому что сумма может оказаться положительной или отрицательной в зависимости от направления обхода границы многоугольника.

Теорема Пика дает еще один способ вычисления площади многоугольника, все вершины которого имеют целочисленные координаты. Она утверждает, что площадь равна

$$a + b/2 - 1,$$

где a – количество целых точек внутри многоугольника, а b – количество целых точек на его границе. Например, площадь многоугольника на рис. 13.12 равна

$$6 + 7/2 - 1 = 17/2.$$

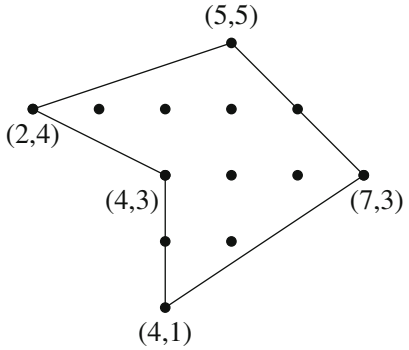


Рис. 13.12. Вычисление площади многоугольника с помощью теоремы Пика

13.1.4. *Метрики*

Метрика определяет расстояние между двумя точками. Обычное *евклидово расстояние* между точками (x_1, y_1) и (x_2, y_2) равно

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Манхэттенское расстояние между точками (x_1, y_1) и (x_2, y_2) вычисляется по формуле

$$|x_1 - x_2| + |y_1 - y_2|.$$

На рис. 13.13 евклидово расстояние между точками равно

$$\sqrt{(5 - 2)^2 + (2 - 1)^2} = \sqrt{10},$$

а манхэттенское расстояние между ними же равно

$$|5 - 2| + |2 - 1| = 4.$$

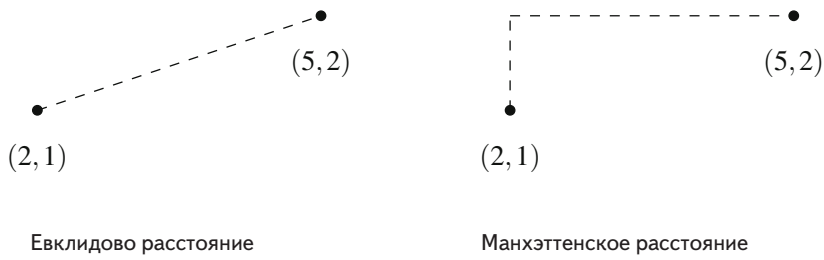


Рис. 13.13. Две метрики

На рис. 13.14 показано, как выглядят единичные круги – множества точек, удаленных от центра на расстояние, не большее 1, – при использовании евклидовой и манхэттенской метрик.

Рис. 13.14. Единичные круги

Некоторые задачи проще решить, если использовать манхэттенское, а не евклидово расстояние. Например, пусть дано множество точек на плоскости и требуется найти пару точек, для которых манхэттенское расстояние максимально. Так, на рис. 13.15 манхэттенское расстояние между точками B и C максимально и равно 5.

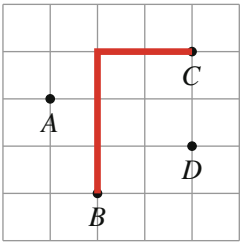


Рис. 13.15. Манхэттенское расстояние между точками B и C максимально

При работе с манхэттенским расстоянием бывает полезно применить преобразование координат $(x, y) \to (x + y, y - x)$. При этом происходят поворот на 45° и масштабирование. На рис. 13.16 показан результат этого преобразования в нашем примере.

Рассмотрим теперь две точки $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$, которые после преобразования перешли в $p_1' = (x_1', y_1')$ и $p_2' = (x_2', y_2')$. Манхэттенское расстояние между p_1 и p_2 можно выразить двумя способами:

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x_1' - x_2'|, |y_1' - y_2'|).$$

Например, если $p_1 = (1, 0)$ и $p_2 = (3, 3)$, то преобразованные координаты равны $p_1' = (1, -1)$ и $p_2' = (6, 0)$, а манхэттенское расстояние

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

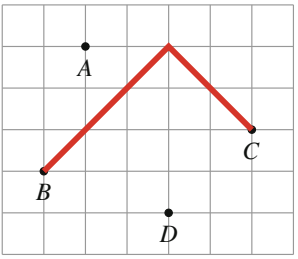


Рис. 13.16. Максимальное манхэттенское расстояние после преобразования координат

Преобразование координат дает простой способ работать с манхэттенскими расстояниями, потому что координаты x и y можно рассматривать порознь. В частности, чтобы максимизировать манхэттенское расстояние, необходимо найти такие две точки, что их преобразованные координаты доставляют максимум величине

$$\max(|x_1' - x_2'|, |y_1' - y_2'|).$$

Это легко, потому что должна быть максимальна разность горизонтальных или вертикальных координат после преобразования.

13.2. Алгоритмы на основе заметающей прямой

Многие геометрические задачи можно решить с помощью заметающей прямой. Идея в том, чтобы представить задачу в виде множества событий, соответствующих точкам на плоскости. Затем события обрабатываются в порядке возрастания координаты *x* или *y*.

13.2.1. *Точки пересечения*

Дано множество *n* горизонтальных и вертикальных отрезков, требуется вычислить количество точек их пересечения. На рис. 13.17 показаны пять отрезков и три точки пересечения.

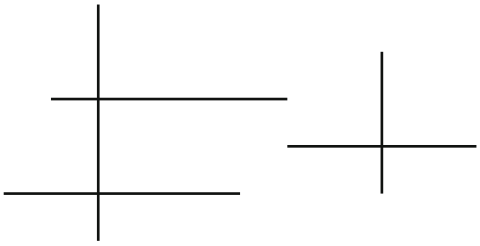


Рис. 13.17. Пять отрезков, пересекающихся в трех точках

Задачу легко решить за время $O(n^2)$, перебрав все пары отрезков и проверив, пересекаются они или нет. Но можно ограничиться временем $O(n \log n)$, если воспользоваться заметающей прямой и структурой данных для запроса по диапазону. Идея в том, чтобы просматривать концевые точки отрезков слева направо, обрабатывая следующие события:

(1) горизонтальный отрезок начался;
(2) горизонтальный отрезок закончился;
(3) встретился вертикальный отрезок.

На рис. 13.18 показаны эти события в нашем примере.

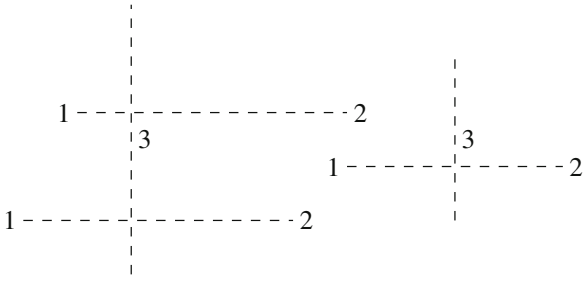


Рис. 13.18. События, соответствующие отрезкам

Создав события, мы пробегаем по ним слева направо, используя структуру данных, в которой хранятся координаты y активных горизонтальных

отрезков. При обработке события 1 координата y отрезка добавляется в структуру, а при обработке события 2 удаляется из нее. При обработке события 3 вычисляются точки пересечения: если координаты концов вертикального отрезка равны y_1 и y_2, то мы подсчитываем количество активных горизонтальных отрезков, для которых координата y лежит между y_1 и y_2, и прибавляем это число к общему числу точек пересечения.

Для хранения координат y горизонтальных отрезков можно использовать двоичное индексное дерево или дерево отрезков, возможно, со сжатием индексов. Обработка одного события занимает время $O(\log n)$, так что временная сложность алгоритма равна $O(n \log n)$.

13.2.2. *Задача о ближайшей паре точек*

Далее рассмотрим такую задачу: дано множество n точек, требуется найти две точки, евклидово расстояние между которыми наименьшее. На рис. 13.19 изображено множество точек, а ближайшая пара выделена черным.

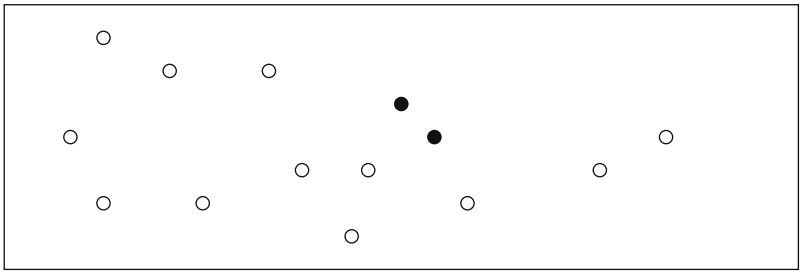


Рис. 13.19. Пример задачи о ближайшей паре точек

Это еще один пример задачи, которую можно решить за время $O(n \log n)$ с помощью алгоритма на основе заметающей прямой[1]. Мы просматриваем точки слева направо и храним значение d: минимальное расстояние между двумя встретившимися до сих пор точками. Если расстояние меньше d, то оно становится новым минимальным расстоянием и значение d обновляется.

Если (x, y) – текущая точка и слева от нее существует точка, удаленная на расстояние меньше d, то координата x такой точки должна принадлежать диапазону $[x - d, x]$, а координата y – диапазону $[y - d, y + d]$. Поэтому достаточно рассмотреть только точки, находящиеся в этой области, что и делает алгоритм эффективным. На рис. 13.20 область внутри штрихового контура содержит точки, которые могут находиться на расстоянии, не большем d от текущей.

[1] Создание эффективного алгоритма нахождения ближайшей пары точек когда-то было важной нерешенной задачей вычислительной геометрии. В конце концов, Шамос и Хоуи [30] придумали алгоритм типа «разделяй и властвуй», работающий за время $O(n \log n)$. Представленный в этом разделе алгоритм на основе заметающей прямой имеет с ним общие черты, но проще в реализации.

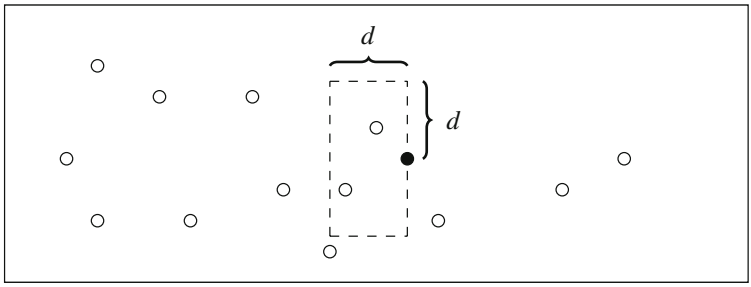


Рис. 13.20. Область, в которой должна находиться ближайшая точка

Эффективность алгоритма основана на том факте, что эта область всегда содержит лишь $O(1)$ точек. Чтобы понять, почему это так, обратимся к рис. 13.21. Поскольку текущее минимальное расстояние между двумя точками равно d, каждый квадрат размера $d/2 \times d/2$ может содержать не более одной точки. А раз так, то во всей области не может быть больше восьми точек.

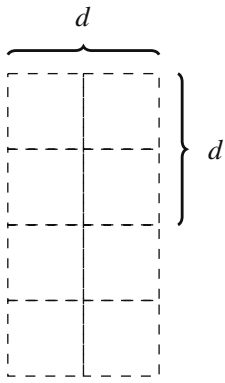


Рис. 13.21. Область ближайших точек содержит $O(1)$ точек

Перебрать все точки можно за время $O(\log n)$, если хранить множество точек с координатой x в диапазоне $[x - d, x]$ отсортированными в порядке возрастания координаты y. Временная сложность алгоритма равна $O(n \log n)$, поскольку нам нужно просмотреть n точек и для каждой определить ближайшую к ней слева, для чего необходимо время $O(\log n)$.

13.2.3. *Задача о выпуклой оболочке*

Выпуклой оболочкой называется наименьший выпуклый многоугольник, содержащий все точки заданного множества. Здесь слово «выпуклый» означает, что если две точки находятся внутри многоугольника, то внутри находится и весь отрезок между ними. На рис. 13.22 показана выпуклая оболочка множества точек.

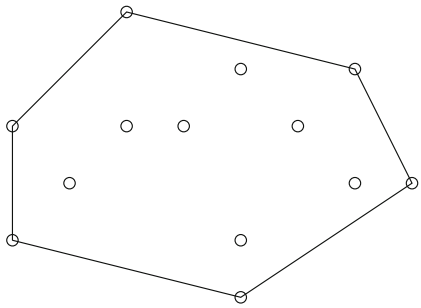


Рис. 13.22. Выпуклая оболочка множества точек

Существует много эффективных алгоритмов построения выпуклой оболочки. Пожалуй, самым простым является описываемый ниже *алгоритм Эндрю* [2]. Сначала алгоритм находит самую левую и самую правую точку множества, затем строит верхнюю и нижнюю части выпуклой оболочки. Обе части похожи, так что можно ограничиться только верхней.

Сначала точки сортируются по координате x, а если координаты x двух точек совпадают, то по координате y. Затем мы просматриваем все точки и добавляем их в оболочку. Добавив очередную точку в оболочку, мы смотрим, поворачивает ли последний отрезок налево. Если да, то мы удаляем предпоследнюю точку из оболочки и повторяем проверку. На рис. 13.23 показано, как алгоритм Эндрю строит верхнюю выпуклую оболочку для нашего множества точек.

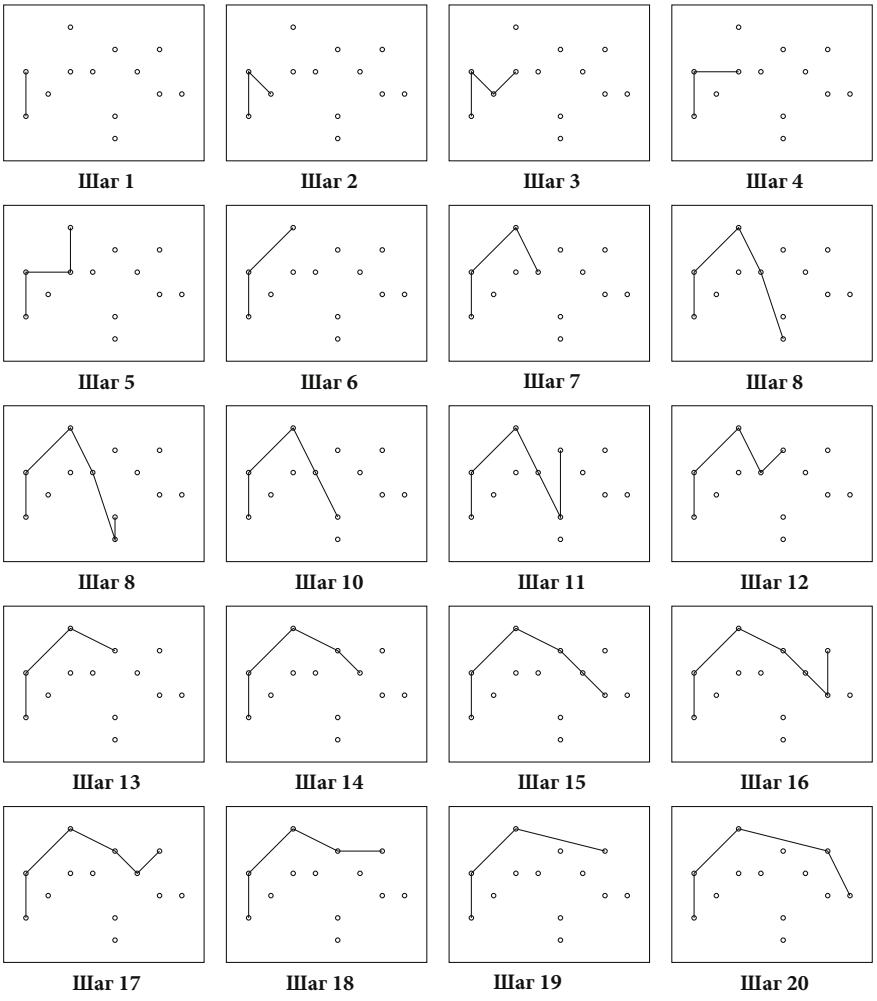


Рис. 13.23. Построение верхней части выпуклой оболочки с помощью алгоритма Эндрю

Глава 14

Алгоритмы работы со строками

В этой главе обсуждаются вопросы обработки строк.

В разделе 14.1 рассмотрена структура префиксного дерева для хранения множества строк. Затем обсуждаются алгоритмы динамического программирования для определения наибольшей общей подпоследовательности и редакторского расстояния.

В разделе 14.2 обсуждаются методы хеширования строк, лежащие в основе эффективных алгоритмов работы со строками. Идея заключается в том, чтобы сравнивать хеш-коды строк вместо символов, это позволяет сравнивать строки за постоянное время.

В разделе 14.3 мы познакомимся с Z-алгоритмом, который для каждой позиции строки находит самую длинную подстроку, одновременно являющуюся префиксом строки. Z-алгоритм – альтернативный подход ко многим задачам, которые можно решить также с помощью хеширования.

В разделе 14.4 обсуждается структура суффиксного массива, используемого для решения многих нетривиальных задач обработки строк.

14.1. Базовые методы

В этой главе предполагается, что индексирование строк начинается с 0, т. е. строка s длины n состоит из символов s[0], s[1], ..., s[n – 1].

Подстрокой называется последовательность соседних символов строки. s[a ... b] обозначает подстроку, которая начинается в позиции a и заканчивается в позиции b. *Префиксом* называется подстрока, начинающаяся первым символом строки, а *суффиксом* – подстрока, заканчивающаяся последним символом строки.

Подпоследовательностью называется любая последовательность символов строки в их исходном порядке. Любая подстрока является подпоследовательностью, но обратное неверно (рис. 14.1).

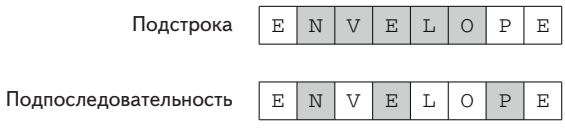


Рис. 14.1. NVELO – подстрока, NEP – подпоследовательность

14.1.1. *Префиксное дерево*

Префиксным деревом (trie) называется корневое дерево для хранения множества строк. Каждая строка хранится в виде цепочки символов, начинающейся в корне. Если у двух строк имеется общий префикс, то у них также будет общая цепочка в дереве. Префиксное дерево на рис. 14.2 соответствует множеству строк {CANAL, CANDY, THE, THERE}. Двойной окружностью обозначены вершины, в которых некоторая строка заканчивается.

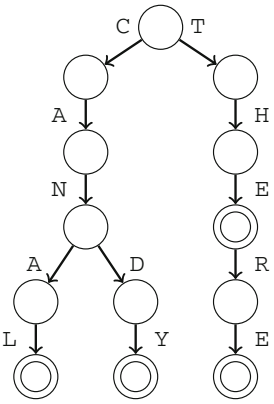


Рис. 14.2. Префиксное дерево, содержащее строки CANAL, CANDY, THE и THERE

Построив префиксное дерево, мы можем легко проверить, содержит ли оно данную строку. Для этого нужно проследовать по цепочке, начинающейся в корневом узле. Чтобы добавить в дерево новую строку, нужно тоже следовать по цепочке, добавляя в нужных местах новые вершины. Временная сложность обеих операций равна $O(n)$, где n – длина строки.

Префиксное дерево можно сохранить в массиве

```
int trie[N][A];
```

где N – максимальное число вершин (суммарная длина всех строк во множестве), а A – размер алфавита. Вершины дерева нумеруются числами 0, 1, 2, ... таким образом, что корень имеет номер 0, а trie[s][c] определяет ту вершину в цепочке, в которую мы переходим из вершины s, встретив символ c.

Структуру префиксного дерева можно обобщить несколькими способами. Пусть, например, требуется вычислить количество строк с определенным префиксом. Это можно сделать эффективно, сохранив в каждой вершине дерева количество строк, чьи цепочки проходят через эту вершину.

14.1.2. *Динамическое программирование*

Методом динамического программирования можно решить много задач, относящихся к строкам. Ниже мы рассмотрим два примера.

Наибольшая общая подпоследовательность. *Наибольшей общей подпоследовательностью* двух строк называется самая длинная строка, встречающаяся в качестве подпоследовательности в обеих строках. Например, OR – наибольшая общая подпоследовательность строк TOUR и OPERA.

Применив динамическое программирование, мы можем определить наибольшую общую подпоследовательность двух строк x и y за время $O(nm)$, где n и m – длины строк. Для этого определим функцию lcs(i, j), ко-

торая возвращает длину наибольшей общей подпоследовательности префиксов x[0 … *i*] и y[0 … *j*]. Тогда имеет место рекуррентное соотношение

$$\texttt{lcs}(i, j) = \begin{cases} \texttt{lcs}(i-1, j-1) + 1 & \texttt{x}[i] = \texttt{y}[j] \\ \max(\texttt{lcs}(i, j-1), \texttt{lcs}(i-1, j)) & \text{в противном случае} \end{cases}$$

Идея в том, что если символы x[*i*] и y[*j*] равны, то мы их учитываем и увеличиваем длину самой длинной общей подпоследовательности на единицу. В противном случае удаляем последний символ из x или y в зависимости от того, какой выбор лучше.

На рис. 14.3 показаны значения функции lcs в нашем примере.

		O	P	E	R	A
T	0	0	0	0	0
O	1	1	1	1	1
U	1	1	1	1	1
R	1	1	1	2	2

Рис. 14.3. Значения функции lcs для определения самой длинной общей подпоследовательности строк TOUR и OPERA

Редакторское расстояние. *Редакторским расстоянием* (или *расстоянием Левенштейна*) между двумя строками называется наименьшее число операций редактирования, преобразующих первую строку во вторую. Разрешены следующие операции:

- вставка символа (например, ABC → ABCA);
- удаление символа (например, ABC → AC);
- замена символа (например, ABC → ADC).

Так, редакторское расстояние между строками LOVE и MOVIE равно 2, потому что мы можем сначала выполнить операцию LOVE → MOVE (замена), а затем операцию MOVE → MOVIE (вставка).

		M	O	V	I	E
L	1	2	3	4	5
O	2	1	2	3	4
V	3	2	1	2	3
E	4	3	2	2	2

Рис. 14.4. Значения функции edit для определения редакторского расстояния между строками LOVE и MOVIE

Вычислить редакторское расстояние между строками x и y можно за время $O(nm)$, где n и m – длины строк. Обозначим edit(*i*, *j*) редакторское расстояние между префиксами x[0 … *i*] и y[0 … *j*]. Значение этой функции можно вычислить, пользуясь рекуррентным соотношением:

$$\begin{aligned} \texttt{edit}(a, b) = \min(&\texttt{edit}(a, b-1) + 1, \\ &\texttt{edit}(a-1, b) + 1, \\ &\texttt{edit}(a-1, b-1) + \texttt{cost}(a, b)), \end{aligned}$$

где cost(*a*, *b*) = 0, если x[*a*] = y[*b*], а в противном случае cost(*a*, *b*) = 1. В этой формуле учтены три способа редактирования строки x: вставка символа в конец x, удаление последнего символа x и совпадение либо замена последнего символа x. В последнем случае, если x[*a*] = y[*b*], последние символы можно оставить без изменения.

На рис. 14.4 показаны значения функции edit в нашем примере.

14.2. Хеширование строк

Хеширование строк позволяет эффективно отвечать на вопрос о равенстве строк, сравнивая их хеш-коды. *Хеш-код* – это целое число, вычисляемое по символам строки. Если две строки равны, то их хеш-коды тоже равны, благодаря чему мы можем сравнивать строки, зная хеш-коды.

14.2.1. *Полиномиальное хеширование*

Чаще всего реализуют *полиномиальное хеширование* строк, при котором хеш-код строки `s` длины n вычисляется по формуле

$$(s[0]A^{n-1} + s[1]A^{n-2} + \ldots + s[n-1]A^0) \bmod B,$$

где `s`[0], `s`[1], ..., `s`[n − 1] интерпретируются как коды символов, а A и B – заранее выбранные константы.

Вычислим, к примеру, хеш-код строки `ABACB`. Коды символов `A`, `B` и `C` равны 65, 66, и 67. В качестве констант выберем $A = 3$ и $B = 97$. Тогда хеш-код равен

$$(65 \cdot 3^4 + 66 \cdot 3^3 + 65 \cdot 3^2 + 66 \cdot 3^1 + 67 \cdot 3^0) \bmod 97 = 40.$$

При использовании полиномиального хеширования хеш-код любой подстроки строки `s` можно вычислить за время $O(1)$ после предобработки, занимающей время $O(n)$. Идея в том, чтобы построить массив `h` – такой, что `h`[k] содержит хеш-код префикса `s`[0 ... k]. Элементы массива вычисляются рекурсивно по формуле:

$$h[0] = s[0]$$
$$h[k] = (h[k-1]A + s[k]) \bmod B.$$

Дополнительно строится массив `p`, для которого `p`[k] = $A^k \bmod B$:

$$p[0] = 1$$
$$p[k] = (p[k-1]A) \bmod B.$$

Для построения этих массивов требуется время $O(n)$. После этого хеш-код любой подстроки `s`[a ... b] можно вычислить за время $O(1)$ по формуле

$$(h[b] - h[a-1]p[b-a+1]) \bmod B$$

в предположении, что $a > 0$. Если $a = 0$, то хеш-код равен просто `h`[b].

14.2.2. *Применения*

С помощью хеширования можно решить много задач, относящихся к строкам, потому что мы можем сравнивать произвольные подстроки строк за время $O(1)$. Фактически хеширование зачастую позволяет сделать эффективным алгоритм полного перебора.

Сопоставление с образцом. Фундаментальной задачей для строк является *сопоставление с образцом*: даны строка *s* и образец *p*, требуется установить, в каких позициях *s* встречается *p*. Например, образец ABC встречается в строке ABCABABCA в позициях 0 и 5 (рис. 14.5).

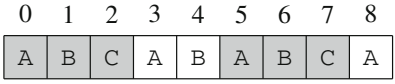


Рис. 14.5. Образец ABC встречается в строке ABCABABCA дважды

Эту задачу можно решить за время $O(n^2)$ методом полного перебора: пробежаться по всем позициям *s*, в которых может встретиться *p*, и сравнить строки посимвольно. Но этот алгоритм можно сделать эффективным за счет хеширования, поскольку тогда каждое сравнение занимает время $O(1)$. Таким образом, мы получаем алгоритм с временной сложностью $O(n)$.

Различные подстроки. Рассмотрим задачу о подсчете числа *различных* подстрок длины *k* заданной строки. Например, в строке ABABAB имеется две различные подстроки длины 3: ABA и BAB. Применив хеширование, мы можем вычислить хеш-коды каждой подстроки и свести задачу к подсчету числа различных целых чисел в списке, что можно сделать за время $O(n \log n)$.

Минимальная циклическая перестановка. *Циклическая перестановка* строки создается путем перемещения первого символа строки в конец. Например, циклическими перестановками строки ATLAS являются ATLAS, TLASA, LASAT, ASATL и SATLA. Рассмотрим задачу о нахождении лексикографически *минимальной* циклической перестановки строки. Например, для строки ATLAS таковой является циклическая перестановка ASATL.

Для эффективного решения задачи мы воспользуемся комбинацией хеширования строк и *двоичного поиска*. Ключевая идея заключается в том, что определить лексикографический порядок двух строк можно за логарифмическое время. Сначала мы вычисляем длину общего префикса строк, применяя двоичный поиск. Здесь хеширование позволяет за время $O(1)$ проверить, совпадают ли префиксы определенной длины. Затем проверяем следующий за общим префиксом символ, который и определяет взаимный порядок строк.

Теперь, чтобы решить задачу, мы строим строку, содержащую две расположенные подряд копии исходной строки (ATLASATLAS), и перебираем ее подстроки длины *n*, запоминая по ходу минимальную подстроку. Поскольку для одного сравнения нужно время $O(\log n)$, полная временная сложность алгоритма равна $O(n \log n)$.

14.2.3. *Коллизии и параметры*

Очевидной опасностью при сравнении хеш-кодов является *коллизия* – ситуация, когда строки различны, но их хеш-коды совпадают. В таком

случае алгоритм, основанный на хеш-кодах, заключает, что строки одинаковы, хотя на самом деле это не так, поэтому алгоритм может давать неправильные результаты.

Избавиться от коллизий невозможно, потому что число различных строк больше числа возможных хеш-кодов. Однако вероятность коллизии мала, если константы A и B выбраны с умом. Обычно выбирают случайные константы, близкие к 10^9, например:

$$A = 911382323,$$
$$B = 972663749.$$

При таком выборе для вычисления хеш-кодов можно использовать тип `long long`, поскольку произведения AB и BB в него укладываются. Но достаточно ли иметь примерно 10^9 различных хеш-кодов?

Рассмотрим три случая, в которых применяется хеширование.

Случай 1. Строки x и y сравниваются между собой. Вероятность коллизии равна $1/B$ в предположении, что все хеш-коды равновероятны.

Случай 2. Строка x сравнивается со строками y_1, y_2, ..., y_n. Вероятность хотя бы одной коллизии равна

$$1 - (1 - 1/B)^n.$$

Случай 3. Строки x_1, x_2, ..., x_n сравниваются попарно. Вероятность хотя бы одной коллизии равна

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

Таблица 14.1. Вероятности коллизий при разных применениях хеширования

Константа B	Случай 1	Случай 2	Случай 3
10^3	0.00	1.00	1.00
10^6	0.00	0.63	1.00
10^9	0.00	0.00	1.00
10^{12}	0.00	0.00	0.39
10^{15}	0.00	0.00	0.00
10^{18}	0.00	0.00	0.00

В табл. 14.1 приведены вероятности коллизий для различных значений B при $n = 10^6$. Как видно, в случаях 1 и 2 вероятность коллизии пренебрежимо мала, если $B \approx 10^9$. Однако в случае 3 все совсем по-другому: при $B \approx 10^9$ коллизия произойдет почти наверняка.

Ситуация в случае 3 известна под названием *парадокс дней рождения*: если в комнате находится n человек, то вероятность того, что *какие-то* два из них родились в один день, велика даже при совсем небольших n. И точ-

но так же, когда все хеш-коды сравниваются попарно, велика вероятность, что какие-то два из них одинаковы.

Вероятность коллизии можно уменьшить, если вычислять *несколько* хеш-кодов с разными параметрами. Крайне маловероятно, что коллизия произойдет одновременно в нескольких хеш-кодах. Например, два хеш-кода с параметром $B \approx 10^9$ эквивалентны одному хеш-коду с параметром $B \approx 10^{18}$, при котором вероятность коллизии очень мала.

Некоторые выбирают $B = 2^{32}$ или $B = 2^{64}$ из-за удобства операций с 32- и 64-разрядными целыми числами по модулю 2^{32} и 2^{64}. Но это *плохой* выбор, поскольку можно сконструировать такие входные данные, что при любой константе вида 2^x коллизии произойдут обязательно [27].

14.3. Z-алгоритм

Z-массив z строки s длины n для каждого $k = 0, 1, \ldots, n - 1$ содержит длину максимальной (самой длинной) подстроки s, которая начинается в позиции k и является префиксом s. Таким образом, z[k] = p означает, что s[0 ... $p - 1$] равно s[k ... $k + p - 1$], но символы s[p] и s[$k + p$] различны (или длина строки равна $k + p$).

На рис. 14.6 показан Z-массив строки ABCABCABAB. В нем z[3] = 5, поскольку подстрока ABCAB длины 5 является префиксом s, а подстрока ABCABA длины 6 таковым не является.

0	1	2	3	4	5	6	7	8	9
A	B	C	A	B	C	A	B	A	B
–	0	0	5	0	0	2	0	2	0

Рис. 14.6. Z-массив строки ABCABCABAB

14.3.1. *Построение Z-массива*

Далее мы опишем *Z-алгоритм* эффективного построения Z-массива за время $O(n)$[1]. Алгоритм вычисляет элементы Z-массива слева направо, при этом используется уже сохраненная в массиве информация и производится посимвольное сравнение строк.

Для эффективного вычисления элементов Z-массива алгоритм хранит диапазон [x, y] – такой, что s[x ... y] – префикс s, значение z[x] уже определено, а значение y максимально возможное. Поскольку мы знаем, что s[0 ... $y - x$] совпадает с s[x ... y], то можем воспользоваться этой информацией при вычислении последующих элементов массива. Предположим, что уже вычислены элементы z[0], z[1], ..., z[$k - 1$] и требуется вычислить z[k]. Возможны три случая.

[1] В работе Gusfield [15] Z-алгоритм представлен как простейший из известных методов сопоставления с образцом за линейное время. При этом авторы ссылаются на оригинальную идею, изложенную в работе Main, Lorentz [26].

Случай 1: $y < k$. В этом случае у нас нет информации о позиции k, поэтому мы вычисляем z[k], сравнивая подстроки посимвольно. Так, на рис. 14.7 диапазона [x, y] еще не существует, поэтому подстроки, начинающиеся в позициях 0 и 3, сравниваются посимвольно. Поскольку z[3] = 5, то новым диапазоном [x, y] становится [3, 7].

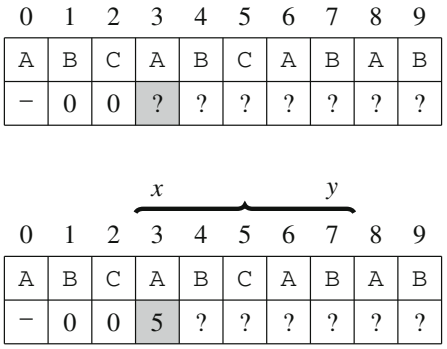


Рис. 14.7. Случай 1: вычисление значения z[3]

Случай 2: $y \geqslant k$ и $k + z[k - x] \leqslant y$. В этом случае мы знаем, что z[k] = z[$k - x$], потому что s[0 … $y - x$] и s[x … y] равны, и мы остаемся внутри диапазона [x, y]. В ситуации на рис. 14.8 мы заключаем, что z[4] = z[1] = 0.

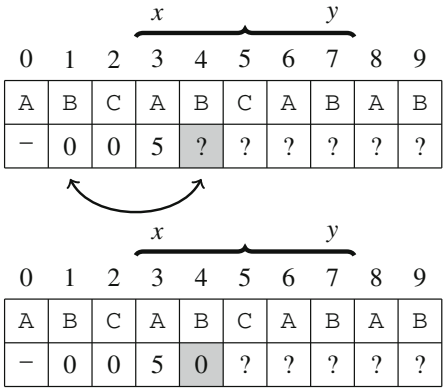


Рис. 14.8. Случай 2: вычисление значения z[4]

Случай 3: $y \geqslant k$ и $k + z[k - x] > y$. В этом случае мы знаем, что $z[k] \geqslant y - k + 1$. Однако поскольку у нас нет никакой информации после позиции y, то мы должны посимвольно сравнить подстроки, начинающиеся в позициях $y - k + 1$ и $y + 1$. Так, на рис. 14.9 мы знаем, что $z[6] \geqslant 2$. А поскольку $s[2] \neq s[8]$, то получается, что z[6] = 2.

Этот алгоритм имеет временную сложность $O(n)$, поскольку всякий раз, как при посимвольном сравнении строки два символа совпадают, значение y увеличивается. Следовательно, полное время работы, необходимое для сравнения подстрок, равно всего лишь $O(n)$.

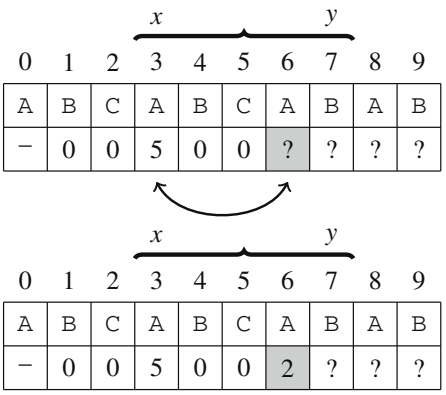


Рис. 14.9. Случай 3: вычисление значения z[6]

14.3.2. *Применения*

Z-алгоритм предлагает альтернативный способ решения многих задач со строками, которые можно решить и с помощью хеширования. Но, в отличие от хеширования, Z-алгоритм всегда дает правильные результаты без риска коллизий. Что применять на практике – хеширование или Z-алгоритм, – зачастую дело вкуса.

Сопоставление с образцом. Снова рассмотрим задачу о сопоставлении с образцом, в которой требуется найти все вхождения образца *p* в строку *s*. Мы уже решили ее с помощью хеширования, а теперь посмотрим, как применить к ней Z-алгоритм.

При обработке строк часто эксплуатируется одна и та же идея: построить строку, состоящую из нескольких частей, разделенных специальными символами. В данном случае мы можем построить строку *p*#*s*, где *p* и *s* разделены специальным символом #, не встречающимся ни в одной из строк. Тогда Z-массив строки *p*#*s* даст нам позиции вхождений *p* в *s*: это те элементы, которые содержат длину *p*.

На рис. 14.10 показан Z-массив для строки *s* = ABCABABCA и образца *p* = ABC. В элементах 4 и 9 находится значение 3, а это значит, что *p* входит в *s*, начиная с позиций 0 и 5.

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	#	A	B	C	A	B	A	B	C	A
–	0	0	0	3	0	0	2	0	3	0	0	1

Рис. 14.10. Сопоставление с образцом с помощью Z-алгоритма

Нахождение границ. *Границей* строки *s* называется строка *b*, являющаяся одновременно префиксом и суффиксом *s*, но не совпадающая с *s*. Например, границами строки ABACABACABA являются A, ABA и ABACABA. Все

границы строки можно эффективно найти с помощью Z-алгоритма, потому что суффикс, начинающийся в позиции k, является границей тогда и только тогда, когда $k + z[k] = n$, где n – длина строки. Так, на рис. 14.11 $4 + z[4] = 11$, т. е. ABACABA – граница строки.

0	1	2	3	4	5	6	7	8	9	10
A	B	A	C	A	B	A	C	A	B	A
–	0	1	0	7	0	1	0	3	0	1

Рис. 14.11. Нахождение границ с помощью Z-алгоритма

14.4. Суффиксные массивы

Суффиксный массив строки описывает лексикографический порядок ее суффиксов. Каждый элемент суффиксного массива содержит начальную позицию некоторого суффикса. На рис. 14.12 показан суффиксный массив строки ABAACBAB.

0	1	2	3	4	5	6	7
2	6	0	3	7	1	5	4

Рис. 14.12. Суффиксный массив строки ABAACBAB

Часто удобно располагать суффиксный массив вертикально вместе с соответствующими суффиксами (рис. 14.13). Отметим, однако, что сам по себе суффиксный массив содержит только начальные позиции суффиксов, а не составляющие их символы.

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	B
5	1	BAACBAB
6	5	BAB
7	4	CBAB

Рис. 14.13. Другой способ представления суффиксного массива

14.4.1. *Метод удвоения префикса*

Простой и эффективный способ построения суффиксного массива строки дает *метод удвоения префикса*, имеющий временную сложность $O(n \log^2 n)$ или $O(n \log n)$ в зависимости от реализации[2]. Алгоритм состоит из раундов с номерами 0, 1, ..., $\lceil \log_2 n \rceil$, на i-м раунде обрабатываются подстроки длины 2^i. В ходе раунда каждой подстроке x длины 2^i назначается целочисленная метка $l(x)$ – такая, что $l(a) = l(b)$, тогда и только тогда, когда $a = b$, и $l(a) < l(b)$ тогда и только тогда, когда $a < b$.

На раунде 0 все подстроки состоят из одного символа, и можно использовать, например, метки A = 1, B = 2 и т. д. Если же $i > 0$, то на i-м раунде для построения меток подстрок длины 2^i используются метки для подстрок длины 2^{i-1}. Чтобы назначить метку $l(x)$ подстроке x длины 2^i, мы разбиваем x на две половины a и b длины 2^{i-1} с метками $l(a)$ и $l(b)$. (Если вторая

[2] Идея метода удвоения префикса изложена в работе Karp, Miller, Rosenberg [20]. Существуют также более эффективные алгоритмы построения суффиксного массива с временной сложностью $O(n)$; в работе Kärkkäinen, Sanders [19] приведен пример довольно простого алгоритма такого рода.

половина начинается за пределами строки, то предполагаем, что ее метка равна 0.) Сначала мы назначаем в качестве *начальной* метки x пару $(l(a), l(b))$. После того как всем подстрокам длины 2^i назначены начальные метки, мы сортируем эти метки и назначаем *конечные* метки – целые числа 1, 2, 3 и т. д. Цель назначения конечных меток состоит в том, чтобы по завершении последнего раунда у каждой подстроки была *уникальная* метка, и эти метки отражали лексикографический порядок подстрок. И наконец, на основе этих меток легко построить суффиксный массив.

На рис. 14.14 показан процесс построения меток для строки ABAACBAB. После раунда 1 мы знаем, что l(AB) = 2 и l(AA) = 1. Поэтому на раунде 2 подстроке ABAA назначается начальная метка (2, 1). Поскольку существуют две меньшие начальные метки ((1, 6) и (2, 0)), то конечная метка будет равна l(ABAA) = 3. Отметим, что в этом примере после раунда 2 все метки уже уникальны, потому что лексикографический порядок подстрок полностью определен первыми четырьмя символами.

Рис. 14.14. Построение меток для строки ABAACBAB

Этот алгоритм работает за время $O(n \log^2 n)$, поскольку число раундов равно $O(\log n)$, и на каждом раунде сортируется список n пар. На самом деле возможна также реализация с временной сложностью $O(n \log n)$, поскольку для сортировки пар можно использовать алгоритм с линейным временем. Тем не менее прямолинейная реализация со временем $O(n \log^2 n)$, для которой не нужно ничего, кроме стандартной функции C++ `sort`, обычно достаточно эффективна.

14.4.2. *Поиск образцов*

Имея суффиксный массив, можно эффективно искать все вхождения произвольного образца в строку. Это делается за время $O(k \log n)$, где

n – длина строки, а k – длина образца. Идея заключается в том, чтобы обрабатывать образец символ за символом и поддерживать диапазон суффиксного массива, который соответствует уже обработанному префиксу образца. Чтобы эффективно обновить диапазон после обработки очередного символа, применяется двоичный поиск.

Найдем, например, вхождения образца BA в строку ABAACBAB (рис. 14.15). Первоначально диапазон поиска равен [0, 7], т. е. покрывает весь суффиксный массив. После обработки символа B диапазон сужается до [4, 6]. Наконец, после обработки символа A остается диапазон [5, 6]. Таким образом, мы заключаем, что образец BA входит в строку ABAACBAB дважды – начиная с позиций 1 и 5.

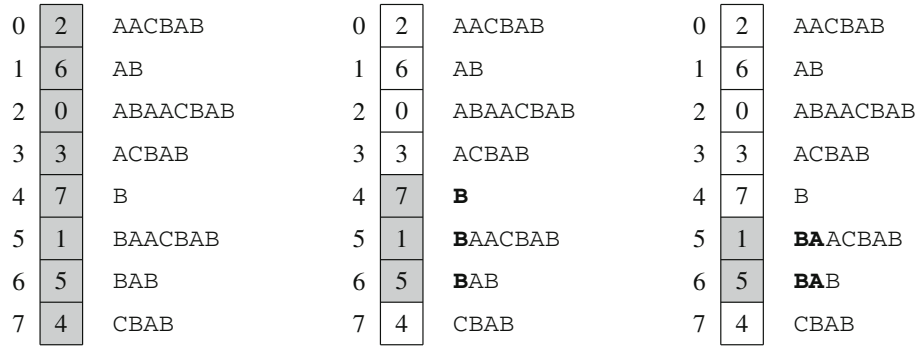


Рис. 14.15. Поиск вхождений образца BA в строку ABAACBAB с помощью суффиксного массива

По сравнению с хешированием строк и Z-алгоритмом, суффиксный массив имеет то преимущество, что позволяет эффективно обрабатывать *несколько* запросов для разных образцов, и знать эти образцы заранее при построении суффиксного массива необязательно.

14.4.3. *LCP-массивы*

LCP-массив строки для каждого ее суффикса дает *значение LCP*: длину *наибольшего общего префикса* (longest common prefix) этого суффикса и следующего суффикса в суффиксном массиве. На рис. 14.16 показан LCP-массив для строки ABAACBAB. Например, значение LCP суффикса BAACBAB равно 2, поскольку наибольший общий префикс BAACBAB и BAB равен BA. Отметим, что у последнего суффикса в суффиксном массиве нет значения LCP.

Далее мы рассмотрим эффективный алгоритм (Kasai et al. [18]) построения LCP-массива строки в предположении, что уже построен ее суффиксный

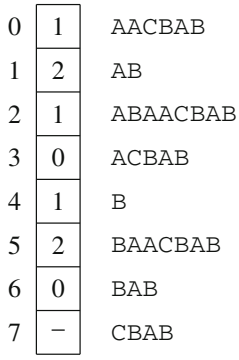


Рис. 14.16. LCP-массив строки ABAACBAB

массив. Алгоритм основан на следующем наблюдении. Рассмотрим суффикс, для которого значение LCP равно x. Если удалить первый символ из суффикса, то значение LCP нового суффикса заведомо должно быть не меньше $x - 1$. Например, на рис. 14.16 значение LCP суффикса BAACBAB равно 2, поэтому значение LCP суффикса AACBAB должно быть не меньше 1. На самом деле оно в точности равно 1.

Воспользовавшись этим наблюдением, мы можем эффективно построить LCP-массив, вычисляя значения LCP в порядке убывания длины суффикса. Для каждого суффикса мы вычисляем его значение LCP, посимвольно сравнивая этот суффикс и следующий за ним в суффиксном массиве. Теперь воспользуемся тем фактом, что нам известно значение LCP суффикса, который на один символ длиннее. Следовательно, текущее значение LCP должно быть не меньше $x - 1$, где x – предыдущее значение LCP, и нам не нужно сравнивать первые $x - 1$ символов суффиксов. Получающийся алгоритм работает за время $O(n)$, поскольку производит всего $O(n)$ сравнений.

С помощью LCP-массива можно эффективно решать некоторые нетривиальные задачи со строками. Например, чтобы вычислить количество различных подстрок строки, мы можем просто вычесть сумму всех элементов LCP-массива из общего числа подстрок, т. е. ответ равен

$$\frac{n(n+1)}{2} - c,$$

где n – длина строки, а c – сумма всех элементов LCP-массива. Например, в строке ABAACBAB имеется

$$\frac{8 \cdot 9}{2} - 7 = 29$$

различных подстрок.

14.5. Строковые автоматы

Автомат[3] – это ориентированный граф, вершины которого называются *состояниями*, а ребра – *переходами*. Одно из состояний является начальным, оно помечается входящим ребром. Может существовать сколько угодно *допускающих состояний*, обозначаемых двойными кружками. Каждому переходу сопоставляется некоторый символ.

Автомат можно использовать для проверки правильности формата строки. Для этого мы стартуем в начальном состоянии, а затем обрабатываем символы слева направо, двигаясь по переходам. Если после обработки всей строки мы оказываемся в допускающем состоянии, то строка допускается, в противном случае отвергается.

[3] Точнее, нас будут интересовать детерминированные конечные автоматы, или ДКА.

В теории автоматов множество строк называется *языком*. Язык автомата состоит из всех допускаемых им строк. Говорят, что автомат *распознает* язык, если он допускает все строки, принадлежащие этому языку, и отвергает все прочие строки.

Например, автомат на рис. 14.17 допускает все строки, состоящие из символов A и B, в которых первый и последний символы различны, т. е. язык этого автомата состоит из строк вида

{AB, BA, AAB, ABB, BAA, BBA, ...}.

В этом автомате состояние 1 начальное, а состояния 3 и 5 допускающие. Если на вход автомата подается строка ABB, то он проходит по цепочке состояний 1 → 2 → 3 → 3 и допускает строку, а если подается строка ABA, то он проходит по цепочке состояний 1 → 2 → 3 → 2 и отвергает строку.

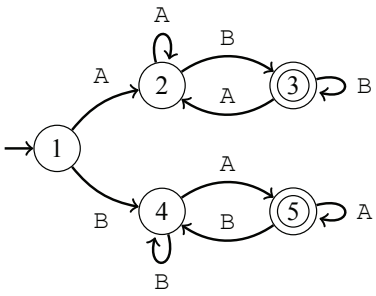


Рис. 14.17. Автомат, допускающий все AB-строки, в которых первый и последний символы различны

Мы предполагаем, что автоматы *детерминированные*, т. е. не существует двух переходов из одного состояния, помеченных одним и тем же символом. Это позволяет эффективно и однозначно применять автомат для обработки любой строки.

14.5.1. *Регулярные языки*

Язык называется *регулярным*, если существует распознающий его автомат. Например, множеств AB-строк, в которых первый и последний символы различны, является регулярным языком, потому что его распознает автомат на рис. 14.17.

Оказывается, что язык является регулярным тогда и только тогда, когда существует *регулярное выражение*, описывающее допустимый формат принадлежащих ему строк. Регулярные выражения состоят из следующих структурных элементов:

- вертикальная черта | означает выбор одного из нескольких вариантов. Например, регулярное выражение AB|BA|C допускает строки AB, BA и C;

- скобки (и) применяются для группировки. Например, регулярное выражение A(A|B)C допускает строки AAC и ABC;
- звездочка * означает, что предшествующую ей часть можно повторять произвольное число раз (в т. ч. нуль). Например, регулярное выражение A(BC)* допускает строки A, ABC, ABCBC и т. д.

Ниже приведено регулярное выражение для автомата на рис. 14.17:

```
A(A|B)*B|B(A|B)*A
```

В этом случае имеется два варианта: либо строка начинается символом A и заканчивается символом B, либо начинается B и заканчивается A. Часть (A|B)* соответствует любой строке, состоящей из символов A и B.

Интуитивно понятно, что язык регулярный, если можно создать алгоритм, который просматривает входную строку слева направо один раз, использует постоянный объем памяти и может определить, принадлежит ли строка языку. Например, язык

{AB, AABB, AAABBB, AAAABBBB, ...}

не является регулярным, потому что мы должны запомнить количество символов A, а затем проверить, что символов B столько же, но для произвольно длинных строк это невозможно сделать в памяти постоянного объема.

Заметим, что в реализациях регулярных выражений в языках программирования часто имеются расширения, позволяющие распознавать языки, не являющиеся строго регулярными, для которых невозможно создать автомат.

14.5.2. *Автоматы для сопоставления с образцом*

Автомат для сопоставления с образцом можно использовать для эффективного обнаружения всех вхождений образца в строку. Идея заключается в том, чтобы создать автомат, допускающий строку тогда и только тогда, когда образец является ее суффиксом. Тогда при обработке строки автомат всегда переходит в допускающее состояние, если обнаружил вхождение образца.

Если задан образец p, содержащий n символов, то автомат для сопоставления с ним имеет $n + 1$ состояний. Состояния нумеруются числами 0, 1, ..., n, так что состояние 0 начальное, а единственным допускающим является состояние n. Если мы находимся в состоянии i, то уже произвели сопоставление с префиксом $p[0 \ldots i - 1]$, т. е. с первыми i символами образца. Далее мы переходим в состояние $i + 1$, если следующий входной символ совпадает с $p[i]$, а в противном случае возвращаемся в какое-то состояние $x \leqslant i$.

Например, на рис. 14.18 показан автомат для сопоставления с образцом ABA. В процессе обработки строки ABABA он перемещается по цепочке состояний $0 \to 1 \to 2 \to 3 \to 2 \to 3$. В допускающее состояние 3 он попадает дважды, это значит, что образец входит в строку два раза.

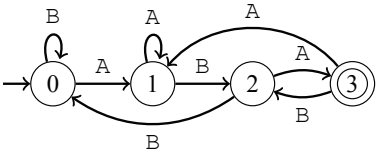


Рис. 14.18. Автомат для сопоставления с образцом ABA

Для построения автомата мы должны определить все переходы между состояниями. Обозначим `nextState[s][c]` состояние, в которое мы переходим из состояния s в результате чтения символа c. Например, на рис. 14.18 `nextState[1][B] = 2`, потому что после чтения B в состоянии 1 мы переходим в состояние 2. Оказывается, что значения `nextState` можно эффективно вычислить, создав сначала *массив граней* для образца, т. е. массив, в котором `border`[i] обозначает длину самой длинной (собственной) грани $p[0 \ldots i]$. Например, на рис. 14.19 показан массив граней образца ABAABABAAA. В частности, `border`[4] = 2, потому что AB – грань ABAAB максимальной длины.

0	1	2	3	4	5	6	7	8	9
A	B	A	A	B	A	B	A	A	A
0	0	1	1	2	3	2	3	4	1

Рис. 14.19. Массив граней ABAABABAAA

Массив граней можно построить за время $O(n)$ следующим образом:

```
border[0] = 0;
for (int i = 1; i < n; i++) {
    int k = border[i-1];
    while (k != 0 && p[k] != p[i]) {
        k = border[k-1];
    }
    border[i] = (p[k] == p[i]) ? k+1 : 0;
}
```

Алгоритм вычисляет значения `border`[i], используя уже вычисленные элементы массива. Идея заключается в том, чтобы обойти грани $p[0 \ldots i-1]$ и выбрать самую длинную грань, которую можно расширить, добавив символ $p[i]$. Время работы алгоритма имеет порядок $O(n)$, потому что `border`[$i+1$] ⩽ `border`[i] + 1, поэтому общее число итераций цикла `while` равно $O(n)$.

После построения массива граней мы можем воспользоваться формулой

$$\texttt{nextState}[s][c] = \begin{cases} s+1 & \text{если } s < n \text{ и } p[s] = c \\ 0 & \text{если } s = 0 \\ \texttt{nextState}[\texttt{border}[s-1]][c] & \text{в противном случае} \end{cases}$$

для вычисления переходов. Если мы можем расширить префикс, сопоставленный в текущий момент, то переходим в следующее состояние. Если не

можем и находимся в состоянии 0, то в нем и остаемся. В противном случае определяем самую длинную грань текущего префикса и следуем по ранее вычисленному переходу. Пользуясь этой формулой, мы можем построить автомат для сопоставления с образцом за время $O(n)$ в предположении, что алфавит не изменяется.

Алгоритм Кнута–Морриса–Пратта [24] – хорошо известный алгоритм сопоставления с образцом, основанный на моделировании автомата для сопоставления с образцом. Его можно рассматривать как альтернативу Z-алгоритму (раздел 14.3).

14.5.3. *Суффиксные автоматы*

Суффиксным автоматом [5] называется автомат, который допускает все суффиксы строки[4] и имеет минимальное число состояний. На рис. 14.20 показан суффиксный автомат для строки `BACA`. Он допускает суффиксы `A`, `CA`, `ACA` и `BACA`. Каждое состояние суффиксного автомата соответствует некоторому множеству строк, т. е. если мы находимся в этом состоянии, значит, произошло сопоставление с одной из строк этого множества. Например, на рис. 14.20 состояние 3 соответствует множеству {`C`, `AC`, `BAC`}, а состояние 5 – множеству {`A`}. Обозначим `length`[x] максимальную длину строки в состоянии x. Тогда `length`[3] = 3 и `length`[5] = 1. Оказывается, что все строки в некотором состоянии являются суффиксами самой длинной из них, а их длины полностью покрывают диапазоны соседних целых чисел. Например, в состоянии 3 все строки являются суффиксами `BAC`, а их длины образуют диапазон 1...3.

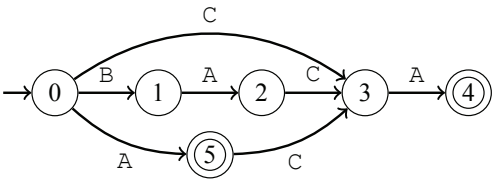


Рис. 14.20. Суффиксный автомат для строки `BACA`

Для данной строки s длины n мы хотим создать ее суффиксный автомат за время $O(n)$, начав с пустого автомата, имеющего только состояние 0, и добавив в него все символы по одному. Для этого мы будем хранить для каждого состояния $x > 0$ *суффиксную ссылку* `link`[x], указывающую на одно из предыдущих состояний автомата. Новый символ c добавляется в автомат следующим образом.

1. Обозначим x текущее последнее состояние автомата, т. е. состояние, из которого не исходит ни один переход. Создадим новое

[4] И только их. – *Прим. ред.*

состояние y и добавим переход из x в y, помеченный c. Положим length[y] = length[x] + 1 и link[y] = 0.

2. Следуем по суффиксным ссылкам, исходящим из x, для каждого посещенного состояния добавляем новый переход в y, помеченный c, пока не найдем состояние s, в которое уже ведет переход, помеченный c. Если такого состояния s нет, то завершаем работу, дойдя до состояния 0. В противном случае переходим к следующему шагу.
3. Обозначим u такое состояние, для которого существует переход из s в u, помеченный c. Если length[s] + 1 = length[u], полагаем link[y] = u и завершаем работу. В противном случае переходим к следующему шагу.
4. Создаем новое состояние z, клонируя состояние u (копируем все исходящие из u переходы в z и полагаем link[z] = link[u]), добавляем переход из s в z, помеченный c, и полагаем length[z] = length[s] + 1. Затем полагаем link[u] = link[y] = z.
5. Наконец, следуем по суффиксным ссылкам, исходящим из s. До тех пор пока из текущего состояния имеется переход в состояние u, помеченный c, заменяем u на z в этом переходе. Если мы нашли состояние, из которого нет перехода в u, помеченного c, или дошли до состояния 0, то завершаем работу.

На рис. 14.21 показан процесс создания суффиксного автомата для строки BACA. После добавления последнего символа мы должны создать дополнительное состояние 5 путем клонирования состояния 2. В этом примере все суффиксные ссылки указывают на состояние 0, и следует ожидать, что в окончательном автомате существует суффиксная ссылка из состояния 4 в состояние 5, обозначенная штриховым ребром. После создания автомата мы можем найти допускающие состояния, начав из последнего состояния и следуя по суффиксным ссылкам, пока не достигнем состояния 0. Все состояния на этом пути (в нашем примере – состояния 4 и 5) являются допускающими.

Заметим, что суффиксная ссылка говорит, в какое состояние мы должны перейти, если хотим найти более короткие строки, являющиеся суффиксами строк в текущем состоянии. В нашем примере состояние 4 соответствует множеству {CA, ACA, BACA}, а состояние 5 – множеству {A}. Таким образом, суффиксную ссылку из состояния 4 в состояние 5 можно использовать для нахождения более короткого суффикса A. На самом деле, следуя по суффиксным ссылкам из состояния x в состояние 0, мы найдем *все* суффиксы самой длинной строки в состоянии x, и каждый суффикс принадлежит ровно одному состоянию.

После того как суффисный автомат создан, мы можем за время $O(m)$ проверить, встречается ли в строке любой образец длины m. Применив дина-

мическое программирование, мы сможем также найти, сколько раз встречается образец, вычислить количество различных подстрок и т. д. В общем случае суффиксные автоматы могут служить альтернативой суффиксным массивам, и с этой новой точки зрения можно рассмотреть многие задачи обработки строк.

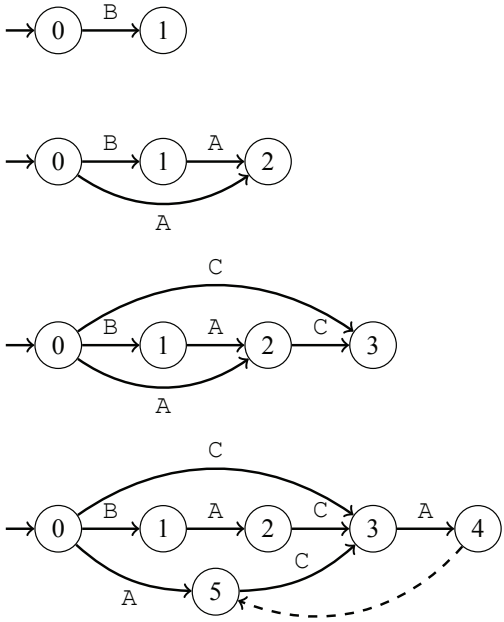


Рис. 14.21. Построение суффиксного автомата

Глава 15

Дополнительные темы

В этой последней главе представлена подборка дополнительных алгоритмов и структур данных. Свободное владение изложенными здесь методами иногда может помочь при решении наиболее трудных олимпиадных задач.

В разделе 15.1 обсуждаются приемы создания структур данных и алгоритмов, в которых встречается квадратный корень. В основе решения зачастую лежит идея о разбиении последовательности n элементов на $O(\sqrt{n})$ блоков, каждый из которых состоит из $O(\sqrt{n})$ элементов.

В разделе 15.2 рассматриваются дополнительные возможности дерева отрезков. Например, мы увидим, как создать дерево отрезков, которое поддерживает как запросы по диапазону, так и обновление диапазона.

В разделе 15.3 представлена структура данных «дуча» (дерево-куча), позволяющая эффективно разбить массив на две части и объединить два массива в один.

Раздел 15.4 посвящен оптимизации метода динамического программирования. Сначала мы рассмотрим трюк с выпуклой оболочкой, который применяется для линейных функций, а затем обсудим его оптимизацию методом «разделяй и властвуй» и оптимизацию Кнута.

В разделе 15.5 рассматриваются различные приемы проектирования алгоритмов, в частности встреча в середине и параллельный двоичный поиск.

15.1. Квадратный корень в алгоритмах

Квадратный корень можно рассматривать как «логарифм для бедных»: сложность $O(\sqrt{n})$ лучше, чем $O(n)$, но хуже, чем $O(\log n)$. Так или иначе, многие структуры данных и алгоритмы, в которых участвуют квадратные корни, оказываются быстрыми и практичными. В этом разделе приведено несколько примеров использования квадратного корня в проектировании алгоритмов.

15.1.1. *Структуры данных*

Иногда эффективная структура данных получается, если разбить массив на *блоки* размера $\sqrt{n}$ и хранить информацию о значениях элементов

массива внутри каждого блока. Например, предположим, что требуется обрабатывать запросы двух типов: модификация элементов массива и нахождение минимального значения в диапазоне. Выше мы видели, что дерево отрезков способно поддержать выполнение обеих операций за время $O(\log n)$, а сейчас решим эту задачу проще, согласившись на временную сложность $O(\sqrt{n})$.

Разобьем массив на блоки по $\sqrt{n}$ элементов и для каждого блока будем хранить минимальное значение в нем. На рис. 15.1 показан массив из 16 элементов, разбитый на блоки по 4 элемента. При изменении значения любого элемента необходимо модифицировать соответствующий блок. Это можно сделать за время $O(\sqrt{n})$, просмотрев все элементы блока, как показано на рис. 15.2. Затем для вычисления минимального значения по диапазону мы разобьем диапазон на три части: одиночные элементы по краям и блоки между ними. На рис. 15.3 показан пример такого разбиения. Ответом на запрос является либо значение какого-то из одиночных элементов, либо минимум в одном из блоков. Поскольку и число одиночных элементов, и число блоков равно $O(\sqrt{n})$, то запрос выполняется за время $O(\sqrt{n})$.

3				2				1				2			
5	8	6	3	4	7	2	6	7	1	7	5	6	2	3	2

Рис. 15.1. Структура на основе квадратного корня для поиска минимального значения в диапазоне

3				4				1				2			
5	8	6	3	4	7	5	6	7	1	7	5	6	2	3	2

Рис. 15.2. При обновлении элемента массива необходимо также обновить минимальное значение в соответствующем блоке

3				2				1				2			
5	8	6	3	4	7	2	6	7	1	7	5	6	2	3	2

Рис. 15.3. Для определения минимального значения в диапазоне этот диапазон разбивается на одиночные элементы и блоки

Насколько эта структура эффективна на практике? Чтобы выяснить это, мы поставили эксперимент: создали массив, содержащий n случайных чисел типа `int`, и обработали n случайных запросов о минимуме. Мы реализовали три структуры данных: дерево отрезков с временной сложностью запроса $O(\log n)$, описанную выше структуру на основе квадратного корня с временной сложностью $O(\sqrt{n})$ и простой массив с временной сложно-

стью $O(n)$. Результаты сведены в табл. 15.1. Оказалось, что для этой задачи структура на основе квадратного корня вполне эффективна для значений n, не превышающих 2^{18}, но потом время выполнения заметно уступает дереву отрезков.

Таблица 15.1. Время выполнения запроса о минимуме в диапазоне при использовании трех структур данных: дерева отрезков ($O(\log n)$), структуры на основе квадратного корня ($O(\sqrt{n})$) и простого массива ($O(n)$)

Размер входных данных n	Время выполнения $O(\log n)$ (с)	Время выполнения $O(\sqrt{n})$ (с)	Время выполнения $O(n)$ (с)
2^{16}	0.02	0.05	1.50
2^{17}	0.03	0.16	6.02
2^{18}	0.07	0.28	24.82
2^{19}	0.14	1.14	> 60
2^{20}	0.31	2.11	> 60
2^{21}	0.99	9.27	> 60

15.1.2. *Подалгоритмы*

Далее мы обсудим две задачи, которые можно эффективно решить путем создания двух *подалгоритмов*, специализированных для различных ситуаций, которые могут иметь место в процессе выполнения алгоритма. И хотя каждый подалгоритм может решить задачу без использования другого, их комбинирование повышает общую эффективность.

Расстояние между буквами. Первая задача ставится следующим образом: дана сетка $n \times n$, в каждой клетке которой находится буква. Каково минимальное манхэттенское расстояние между двумя клетками, содержащими одну и ту же букву? На рис. 15.4 минимальное расстояние между клетками, содержащими букву «D», равно 2.

A	C	E	A
B	D	F	D
E	A	B	C
C	F	E	A

Рис. 15.4. Пример задачи о расстоянии между буквами

Для решения задачи мы можем перебрать все буквы в сетке и для каждой буквы c определить минимальное расстояние между клетками, содержащими эту букву. Рассмотрим два алгоритма обработки фиксированной буквы c.

Алгоритм 1. Перебрать все пары клеток, содержащих букву c, и определить пару, для которой расстояние минимально. Этот алгоритм требует времени $O(k^2)$, где k – число клеток, содержащих букву c.

Алгоритм 2. Выполнить поиск в ширину, одновременно начинающийся во всех клетках с буквой c. Этот поиск занимает время $O(n^2)$.

Для обоих алгоритмов имеются худшие случаи. Для алгоритма 1 это случай, когда буквы во всех клетках одинаковы, тогда $k = n^2$, и время работы алгоритма составляет $O(n^4)$. Для алгоритма 2 худшим является случай, когда буквы во всех клетках разные. Тогда алгоритм придется выполнить $O(n^2)$ раз, и общее время работы составит $O(n^4)$.

Однако мы можем *скомбинировать* эти алгоритмы, так чтобы они работали как подалгоритмы одного алгоритма. Идея в том, чтобы для каждой буквы c по отдельности определить, какой алгоритм использовать. Очевидно, что алгоритм 1 хорошо работает, если k мало, а алгоритм 2 лучше приспособлен для больших k. Следовательно, мы можем зафиксировать константу x и применять алгоритм 1, если k не больше x, и алгоритм 2 в противном случае.

В частности, если выбрать $x = \sqrt{n^2} = n$, то получится алгоритм с временной сложностью $O(n^3)$. Сначала каждая клетка, обрабатываемая алгоритмом 1, сравнивается не более чем с n другими клетками, так что обработка занимает время $O(n^3)$. Затем, поскольку существует не более n букв, которые могут встречаться более чем в n клетках, алгоритм 2 выполняется самое большее n раз, и общее время работы также равно $O(n^3)$.

Черные клетки. В качестве еще одного примера рассмотрим следующую игру. На доске $n \times n$ имеется ровно одна черная клетка, а все остальные белые. На каждом ходе выбирается одна белая клетка, и мы должны вычислить минимальное манхэттенское расстояние между ней и черной клеткой. Затем выбранная белая клетка перекрашивается в черный цвет. Этот процесс повторяется $n^2 - 1$ раз, после чего все клетки оказываются черными.

На рис. 15.5 показан один ход в игре. Минимальное расстояние от выбранной клетки X до черной клетки равно 3 (надо сделать два шага вниз и один шаг вправо). После этого клетка перекрашивается в черный цвет.

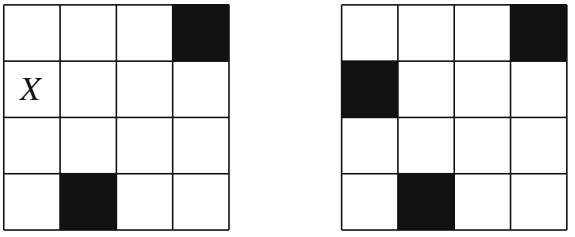


Рис. 15.5. Ход в игре в черные клетки.
Минимальное расстояние от X до черной клетки равно 3

Эту задачу можно решить, обрабатывая ходы *порциями* по k ходов. В начале каждой порции мы для каждой клетки доски вычисляем минимальное расстояние до черной клетки. Это можно сделать за время $O(n^2)$, применив поиск в ширину. Далее в процессе обработки порции мы храним

список всех клеток, которые были перекрашены в черный цвет в текущей порции. Таким образом, минимальное расстояние до черной клетки либо предварительно вычислено, либо является расстоянием до одной из клеток в списке. Поскольку список содержит не более k значений, то для его обхода требуется время $O(k)$.

Следовательно, взяв $k = \sqrt{n^2} = n$, мы получим алгоритм, работающий за время $O(n^3)$. Во-первых, существует $O(n)$ порций, поэтому полное время, затраченное на поиски в ширину, равно $O(n^3)$. Во-вторых, список клеток в порции содержит $O(n)$ значений, поэтому вычисление минимальных расстояний для $O(n^2)$ клеток также требует времени $O(n^3)$.

Оптимизация параметров. На практике необязательно использовать в качестве параметра точный квадратный корень, нужно лишь настроить производительность алгоритма, поэкспериментировав с различными значениями параметров и выбрав то, которое дает наилучший результат. Разумеется, оптимальное значение зависит от алгоритма и от свойств тестовых данных.

В табл. 15.2 приведены результаты эксперимента, в котором алгоритм игры в черные клетки с временной сложностью $O(n^3)$ выполнялся для различных значений k при $n = 500$. Порядок перекрашивания клеток в черный цвет выбирался случайным образом. В данном случае оптимальным, похоже, является параметр k в районе 2000.

Таблица 15.2. Оптимизация значения параметра k в алгоритме игры в черные клетки

Параметр k	Время работы (с)
200	5.74
500	2.41
1000	1.32
2000	1.02
5000	1.28
10 000	2.13
20 000	3.97

15.1.3. *Целые разбиения*

Пусть имеется палочка длины n, разрезанная на несколько частей целочисленной длины. На рис. 15.6 показано несколько таких разбиений для $n = 7$. Каково максимальное число *различных* длин в таком разбиении?

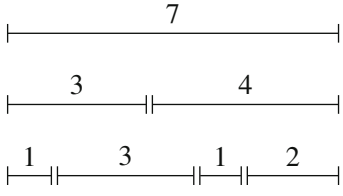


Рис. 15.6. Несколько целых разбиений палочки длины 7

Оказывается, что различных длин может быть не более $O(\sqrt{n})$. А оптимальный способ

получить максимально возможное число различных длин – взять длины 1, 2, …, k. Тогда, поскольку

$$1 + 2 + \cdots + k = \frac{k\,(k+1)}{2},$$

мы можем сделать вывод, что k не может быть больше $O(\sqrt{n})$. Далее мы увидим, как воспользоваться этим наблюдением при проектировании алгоритмов.

Задача о рюкзаке. Пусть дан список целых весов $[w_1, w_2, \ldots, w_k]$ – таких, что $w_1 + w_2 + \ldots + w_k = n$, а наша задача – найти все возможные суммы, составленные из этих весов. На рис. 15.7 показаны возможные суммы, составленные из весов [3, 3, 4].

0	1	2	3	4	5	6	7	8	9	10
✓			✓	✓		✓	✓			✓

Рис. 15.7. Возможные суммы, составленные из весов [3, 3, 4]

Применив стандартный алгоритм рюкзака (раздел 6.2.3), мы можем решить задачу за время $O(nk)$, так что если $k = O(n)$, то временная сложность окажется равной $O(n^2)$. Но поскольку различных весов не более $O(\sqrt{n})$, задачу можно решить эффективнее, если одновременно обрабатывать все веса, имеющие определенное значение. Например, если имеются веса [3, 3, 4], то мы сначала обработаем два веса, равных 3, а затем вес 4. Стандартный алгоритм рюкзака нетрудно модифицировать таким образом, что обработка каждой группы одинаковых весов займет всего лишь время $O(n)$, и в результате получается алгоритм с временной сложностью $O(n\sqrt{n})$.

Составление строки. Рассмотрим еще один пример. Пусть даны строка длины n и словарь, состоящий из слов суммарной длины m. Наша задача – посчитать, сколько существует способов составить строку из слов. Например, имеется четыре способа составить строку `ABAB` из слов `{A, B, AB}`:

- `A + B + A + B`;
- `AB + A + B`;
- `A + B + AB`;
- `AB + AB`.

Применив динамическое программирование, мы для каждого $k = 0, 1, \ldots, n$ сможем вычислить, сколькими способами можно составить префикс строки длиной k. Одно из возможных решений – воспользоваться префиксным деревом, которое содержит обращения всех словарных слов, что дает алгоритм с временной сложностью $O(n^2 + m)$. Но есть и другой подход: воспользоваться хешированием строк и тем фактом, что существует не более $O(\sqrt{m})$ различных длин слов. Следовательно, можно ограничиться фактически

существующими длинами слов. Для этого мы можем создать множество, содержащее все хеш-коды слов, что приводит к алгоритму с временной сложностью $O(n\sqrt{m} + m)$ (если воспользоваться классом `unordered_set`).

15.1.4. *Алгоритм Мо*

Алгоритм Мо[1] обрабатывает множество запросов по диапазону *статического* массива (т. е. значения элементов массива не изменяются между запросами). Для ответа на каждый запрос требуется вычислить нечто, зависящее от элементов массива в некотором диапазоне $[a, b]$. Поскольку массив статический, запросы можно обрабатывать в любом порядке, а изюминка алгоритма Мо состоит в том, чтобы выбрать специальный порядок, гарантирующий эффективность алгоритма.

В алгоритме поддерживается *активный диапазон* массива, и в каждый момент ответ на запрос, касающийся активного диапазона, известен. Алгоритм обрабатывает запросы по одному и каждый раз перемещает концевые точки активного диапазона, вставляя и удаляя элементы. Массив разбивается на блоки по $k = O(\sqrt{n})$ элементов, и запрос $[a_1, b_1]$ всегда обрабатывается раньше запроса $[a_2, b_2]$, если

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$ или
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$ и $b_1 < b_2$.

Таким образом, все запросы, для которых левые концевые точки находятся в некотором блоке, обрабатываются один за другим в порядке возрастания правых концевых точек. При таком порядке алгоритм выполняет только $O(n\sqrt{n})$ операций, поскольку левая концевая точка сдвигается на $O(n) \times O(\sqrt{n})$ шагов, а правая концевая точка – на $O(\sqrt{n}) \times O(n)$ шагов. Следовательно, за время работы алгоритма обе концевые точки суммарно сдвигаются на $O(n\sqrt{n})$ шагов.

Пример. Пусть дано множество диапазонов массива, и требуется вычислить количество *различных* значений в каждом диапазоне. В алгоритме Мо запросы всегда сортируются одинаково, но способ хранения ответа на запрос зависит от задачи.

Чтобы решить эту задачу, мы заведем массив `count`, в котором элемент `count`[x] показывает, сколько раз элемент x встречается в активном диапазоне. При переходе от одного запроса к другому активный диапазон изменяется. Рассмотрим, к примеру, два диапазона на рис. 15.8. Для перехода от первого диапазона ко второму надо сделать три шага: левая концевая точка сдвигается на один шаг вправо, а правая концевая точка – на два шага вправо.

После каждого шага массив `count` следует обновить. Добавив элемент x, мы увеличиваем значение `count`[x] на 1, и если после этого окажется, что

[1] Согласно [6], алгоритм Мо назван в честь китайского олимпиадного программиста Мо Тао.

count[x] = 1, то также прибавляем 1 к ответу на запрос. Аналогично, удалив элемент x, мы уменьшаем значение count[x] на 1, и если после этого окажется, что count[x] = 0, то также вычитаем 1 из ответа на запрос. Поскольку каждый шаг выполняется за время $O(1)$, временная сложность алгоритма равна $O(n\sqrt{n})$.

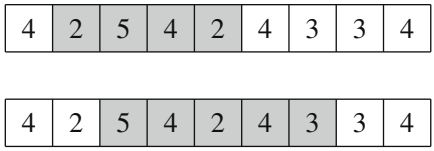


Рис. 15.8. Переход от одного диапазона к другому в алгоритме Мо

15.2. И снова о деревьях отрезков

Дерево отрезков – многоцелевая структура данных, применимая для решения многих разных задач. До сих пор мы видели лишь небольшую часть ее возможностей. Пора обсудить более развитые варианты деревьев отрезков, позволяющие решать задачи посложнее.

Выше при реализации операций с деревом отрезков мы поднимались *снизу вверх*. Так, для вычисления суммы значений в диапазоне [a, b] (раздел 9.2.2) использовалась следующая функция:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Но в деревьях отрезков с дополнительными возможностями часто бывает необходимо реализовывать операции, выполняемые *сверху вниз*, например:

```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

С помощью этой функции мы можем вычислить сумму в диапазоне $[a, b]$ следующим образом:

```
int s = sum(a,b,1,0,n-1);
```

Параметр k обозначает текущую позицию в дереве. Первоначально k равно 1, поскольку начинаем мы с корня дерева. Диапазон $[x, y]$ соответствует k и вначале равен $[0, n-1]$. Если при вычислении суммы $[x, y]$ оказывается вне $[a, b]$, то сумма равна 0, а если $[x, y]$ целиком расположен внутри $[a, b]$, то сумму можно обнаружить в дереве. Если же $[x, y]$ расположен частично внутри $[a, b]$, то поиск продолжается рекурсивно в левой и правой половинах $[x, y]$. Левой половиной является диапазон $[x, d]$, а правой – диапазон $[d + 1, y]$, где $d = \lfloor (x + y)/2 \rfloor$.

На рис. 15.9 показано, как продвигается поиск при вычислении значения $\text{sum}_q(a, b)$. Серым цветом закрашены вершины, в которых рекурсия останавливается, поскольку сумму можно найти в дереве. При такой реализации операция занимает время $O(\log n)$, потому что общее число посещенных вершин равно $O(\log n)$.

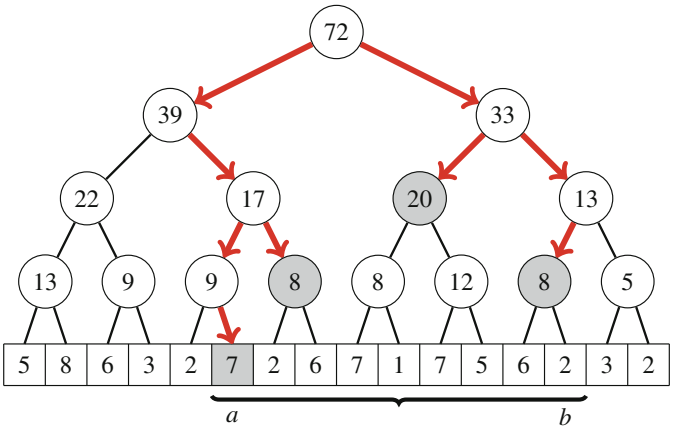


Рис. 15.9. Обход дерева отрезков сверху вниз

15.2.1. *Ленивое распространение*

С помощью ленивого распространения мы можем построить дерево отрезков, которое будет поддерживать как запросы, *так и* обновления по диапазону за время $O(\log n)$. Идея в том, чтобы выполнять обновления и запросы сверху вниз, причем обновления производить *лениво*, чтобы они распространялись вниз по дереву, только когда это необходимо.

В вершинах ленивого дерева отрезков хранится информация двух типов. Как и в обычном дереве отрезков, в каждой вершине хранятся сумма, минимальное значение и другие данные, относящиеся к соответствую-

щему диапазону массива. Дополнительно в вершине может храниться информация о ленивом обновлении, которое еще не было распространено на потомков. Ленивое дерево отрезков способно поддержать два типа обновления диапазона: каждый элемент диапазона либо *увеличивается* на некоторую величину, либо ему *присваивается* новое значение. Для реализации обеих операций используются схожие идеи, и можно даже построить дерево, которое будет поддерживать обе операции.

Рассмотрим случай, когда наша цель – построить дерево отрезков, поддерживающее две операции: увеличение каждого элемента в диапазоне $[a, b]$ на постоянную величину и вычисление суммы элементов $[a, b]$. Для этого будем хранить в каждой вершине два значения s/z, где s – сумма элементов диапазона, а z – величина ленивого обновления, т. е. все элементы диапазона следует увеличить на z. На рис. 15.10 показан пример такого дерева, в котором $z = 0$ во всех вершинах, т. е. невыполненных ленивых обновлений нет.

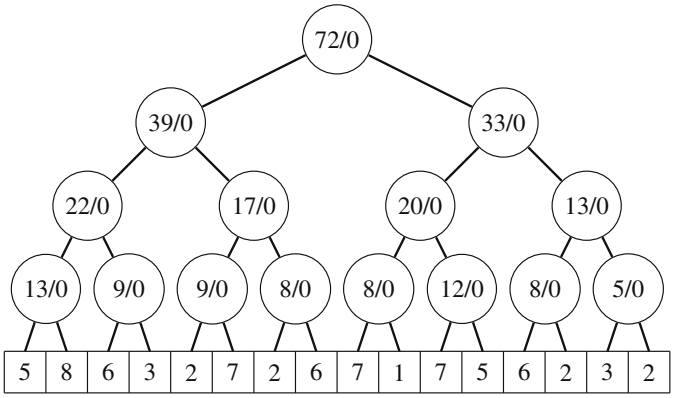


Рис. 15.10. Ленивое дерево отрезков для обновления диапазонов и запросов к ним

Будем реализовывать операции с деревом сверху вниз. Чтобы увеличить элементы диапазона $[a, b]$ на u, модифицируем вершины следующим образом: если диапазон $[x, y]$ вершины расположен целиком внутри $[a, b]$, то увеличиваем значение z в этой вершине на u и останавливаемся. Если $[x, y]$ частично расположен внутри $[a, b]$, то продолжаем рекурсивный обход дерева и после этого вычисляем новое значение s в вершине. На рис. 15.11 показано дерево после увеличения элементов диапазона $[a, b]$ на 2.

И при обновлении, и при выполнении запросов ленивое обновление распространяется вниз по мере спуска по дереву. Перед тем как обратиться к вершине, мы всегда проверяем, существует ли в ней ленивое обновление. Если да, то мы изменяем хранящееся в ней значение s, распространяем обновление на дочерние вершины, после чего сбрасываем значение z. На рис. 15.12 показано, как изменяется дерево при вычислении $\text{sum}_q(a, b)$.

Прямоугольник содержит вершины, значения которых изменяются, когда ленивое обновление распространяется вниз.

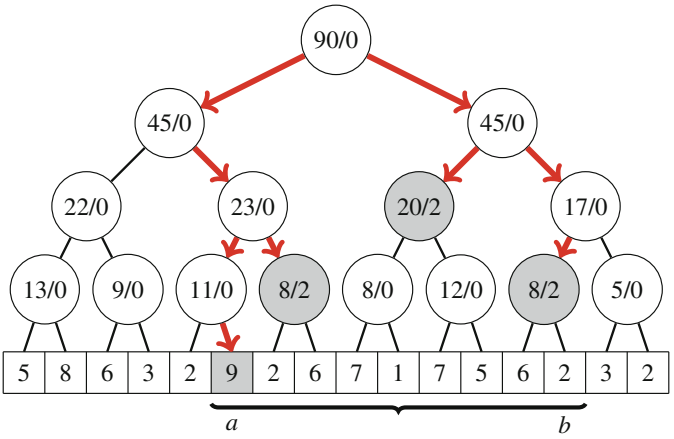


Рис. 15.11. Увеличение элементов диапазона [*a*, *b*] на 2

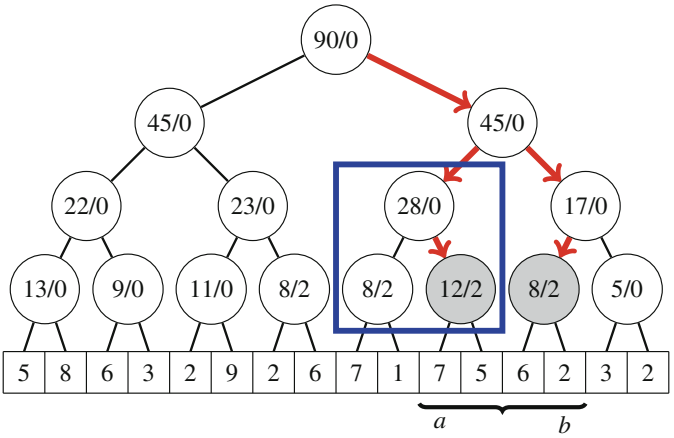


Рис. 15.12. Вычисление суммы элементов диапазона [*a*, *b*]

Полиномиальные обновления. Мы можем обобщить показанное выше дерево отрезков, так чтобы можно было обновлять диапазоны, используя полиномы вида

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \ldots + t_0.$$

В этом случае после обновления i-й элемент диапазона [a, b] принимает значение $p(i - a)$. Например, прибавление полинома $p(u) = u + 1$ к [a, b] означает, что элемент в позиции a увеличивается на 1, элемент в позиции $a + 1$ – на 2 и т. д.

Для поддержки полиномиальных обновлений каждой вершине сопоставляется $k + 2$ значений, где k – степень полинома. Значение s равно сум-

ме элементов диапазона, а z_0, z_1, ..., z_k – коэффициенты полинома, описывающие отложенное обновление. Тогда сумма элементов диапазона [x, y] равна

$$s + \sum_{u=0}^{y-x} (z_k u^k + z_{k-1} u^{k-1} + \cdots + z_1 u + z_0),$$

а значение такой суммы можно эффективно вычислить, применяя формулы суммирования. Например, член z_0 соответствует сумме $z_0(y - x + 1)$, а член $z_1 u$ – сумме

$$z_1(0 + 1 + \cdots + y - x) = z_1 \frac{(y - x)(y - x + 1)}{2}.$$

При распространении обновления по дереву индексы $p(u)$ изменяются, поскольку в каждом диапазоне [x, y] значения вычисляются для $u = 0, 1, \ldots, y - x$. Однако с этим легко справиться, потому что $p'(u) = p(u + h)$ – полином той же степени, что и $p(u)$. Например, если $p(u) = t_2 u^2 + t_1 u + t_0$, то

$$p'(u) = t_2(u + h)^2 + t_1(u + h) + t_0 = t_2 u^2 + (2ht_2 + t_1)u + t_2 h^2 + t_1 h + t_0.$$

15.2.2. *Динамические деревья*

Обычное дерево является статическим, т. е. все вершины занимают фиксированное положение в массиве, представляющем дерево отрезков, и для размещения структуры требуется память фиксированного размера. В *динамическом дереве отрезков* память выделяется только для вершин, к которым действительно производятся обращения в процессе работы алгоритма. Это может заметно сэкономить память.

Вершины динамического дерева можно представить такой структурой:

```
struct node {
   int value;
   int x, y;
   node *left, *right;
   node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

Здесь `value` – значение в вершине, [`x`, `y`] – соответствующий диапазон, а `left` и `right` – указатели на левое и правое поддеревья. Вершины создаются следующим образом:

```
// создать вершину со значением 2 и диапазоном [0,7]
node *x = new node(2,0,7);
// изменить значение
x->value = 5;
```

Разреженные деревья отрезков. Динамическое дерево отрезков особенно полезно, когда стоящий за ним массив *разреженный*, т. е. диапазон его индексов [0, n − 1] велик, но большинство элементов – нули. Если для хранения обычного дерева отрезков понадобилась бы память объемом $O(n)$, то для динамического – только объемом $O(k \log n)$, где k – число выполненных операций.

В *динамическом дереве отрезков* первоначально имеется только одна вершина [0, n − 1] с нулевым значением; это означает, что все элементы массива равны нулю. По мере обновления в дерево динамически добавляются новые вершины. Любой путь от корня к листу содержит $O(\log n)$ вершин, так что всякая операция с деревом добавляет не более $O(\log n)$ новых вершин. Следовательно, после k операций дерево будет содержать $O(k \log n)$ вершин. На рис. 15.13 показано разреженное дерево отрезков, для которого n = 16, а элементы в позициях 3 и 10 модифицированы.

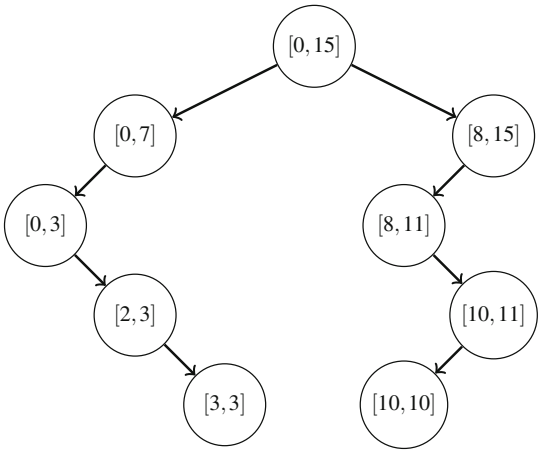


Рис. 15.13. Разреженное дерево отрезков, в котором элементы в позициях 3 и 10 модифицированы

Отметим, что если заранее известны все элементы, которые будут модифицированы в процессе работы алгоритма, то использовать динамическое дерево отрезков необязательно, т. к. можно обойтись обычным деревом отрезков со сжатием индексов (раздел 9.2.3). Однако это не получится, если индексы порождаются в ходе работы алгоритма.

Персистентные деревья отрезков. Динамическая реализация позволяет также создать *персистентное дерево отрезков*, в котором хранится *история модификации*. Имея такую реализацию, мы можем эффективно обращаться ко всем версиям дерева, существовавшим за время выполнения алгоритма. Если доступна история модификации, то запросы к любой предыдущей версии дерева можно выполнять как к обычному дереву отрезков, поскольку сохранена полная структура каждого дерева. Мы также

можем создавать новые деревья на базе предыдущих и модифицировать их независимо.

Рассмотрим последовательность обновлений на рис. 15.14, где помеченные вершины изменяются, а все остальные – нет. После каждого обновления большинство вершин остаются неизменными, поэтому компактный способ хранения истории модификации заключается в том, чтобы представить каждую историческую версию дерева в виде комбинации новых вершин и поддеревьев предыдущих деревьев. На рис. 15.15 показано, как можно сохранить историю. Структуру каждого предыдущего дерева можно воссоздать, следуя по указателям, начиная с соответствующего корня. Поскольку при каждой операции в дерево добавляется только $O(\log n)$ новых вершин, мы можем сохранить полную историю модификации дерева.

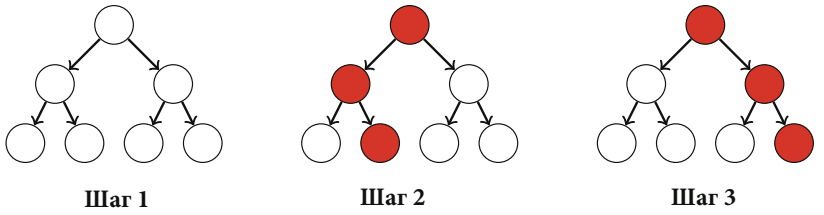


Рис. 15.14. История модификации дерева отрезков: начальное дерево и два обновления

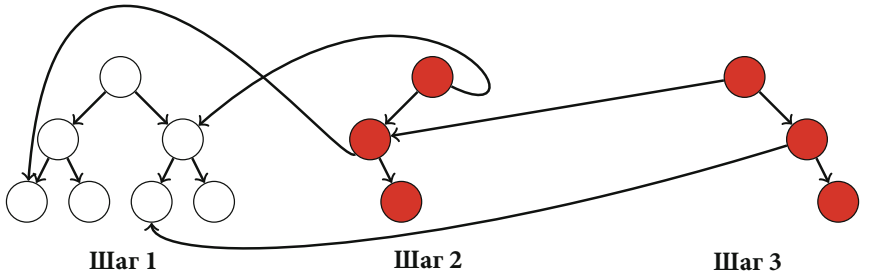


Рис. 15.15. Способ компактного хранения истории модификаций

15.2.3. *Структуры данных в вершинах*

В вершинах дерева отрезков можно хранить не только одиночные значения, но и целые *структуры данных*, содержащие информацию о соответствующих диапазонах. Предположим, к примеру, что требуется эффективно подсчитывать количество вхождений элемента x в диапазон $[a, b]$. Для этого можно создать дерево отрезков, каждой вершине которого сопоставлена структура данных, поддерживающая запросы о количестве вхождений в соответствующий ей диапазон. Тогда для вычисления ответа на запрос можно комбинировать результаты, полученные от вершин, принадлежащих диапазону.

Остается выбрать отвечающую задаче структуру данных. Вполне подойдет отображение `map`, в котором ключом является элемент массива, а значением – число его вхождений в диапазон. На рис. 15.16 показаны массив и соответствующее ему дерево отрезков. Корень дерева говорит, что элемент 1 встречается в массиве 4 раза.

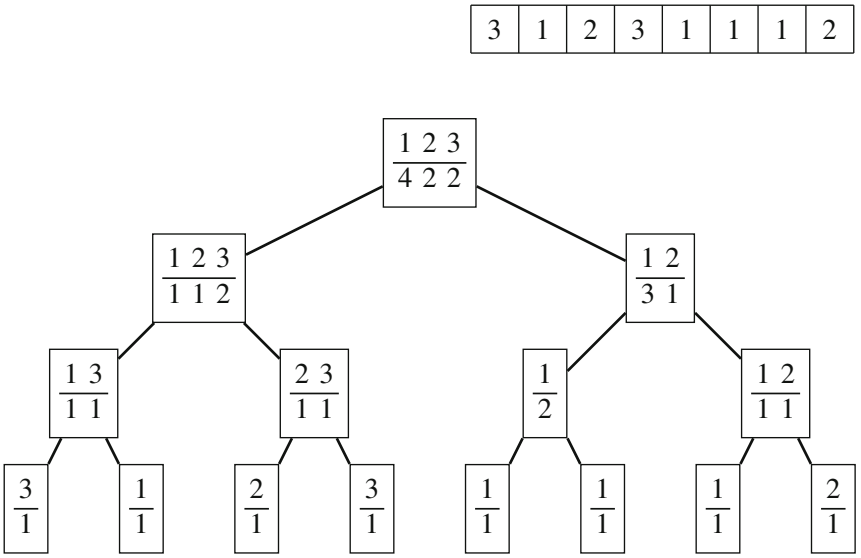


Рис. 15.16. Дерево отрезков для вычисления количества вхождений элемента в диапазон массива

Для выполнения каждого запроса с помощью этого дерева отрезков требуется время $O(\log^2 n)$, поскольку в каждой вершине хранится структура `map`, операции с которой имеют временную сложность $O(\log n)$. Дерево потребляет память объема $O(n \log n)$, т. к. состоит из $O(\log n)$ уровней, и на каждом уровне находится n элементов, распределенных по структурам `map`.

15.2.4. *Двумерные деревья*

Двумерное дерево отрезков позволяет обрабатывать запросы, относящиеся к прямоугольным подмассивам двумерного массива. Идея заключается в том, чтобы создать дерево отрезков, соответствующее столбцам массива, а затем сопоставить каждой его вершине дерево сегментов, соответствующее строкам массива.

На рис. 15.17 показано двумерное дерево отрезков, поддерживающее запросы двух видов: вычисление суммы элементов подмассива и обновление одного элемента массива. Оба запроса занимают время $O(\log^2 n)$, поскольку производится обращение к $O(\log n)$ вершин основного дерева отрезков, а обработка каждой вершины занимает время $O(\log n)$. Структура потребляет память объема $O(n^2)$, потому что основное дерево отрезков

содержит $O(n)$ вершин, и в каждой вершине хранится дерево с $O(n)$ вершинами.

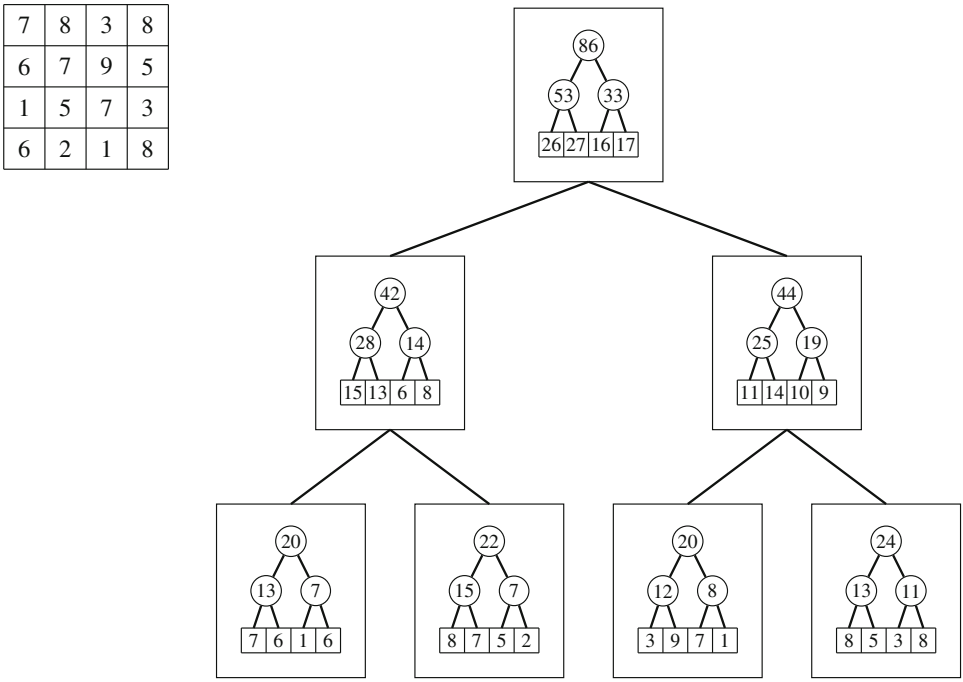


Рис. 15.17. Двумерный массив и соответствующее дерево отрезков для вычисления суммы прямоугольных подмассивов

15.3. Дучи

Дуча (treap) – это двоичное дерево, в котором содержимое массива хранится таким образом, что массив можно эффективно разбить на два других, а два массива объединить в один. В каждой вершине дучи хранятся две величины: *вес*[2] и *значение*. Вес вершины не превосходит веса любой из ее дочерних вершин, а в массиве любая вершина расположена *после* всех вершин ее левого поддерева и *до* всех вершин ее правого поддерева.

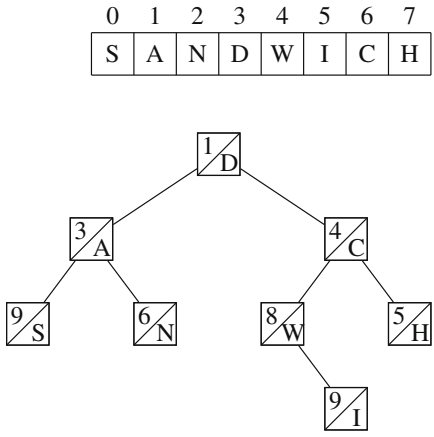


Рис. 15.18. Массив и соответствующая ему дуча

На рис. 15.18 приведен пример массива и соответствующей ему дучи. В частности, корневая вершина имеет

[2] В русскоязычном сообществе эта величина носит название «приоритет». – *Прим. ред.*

вес 1 и значение D. Левое поддерево содержит три вершины, поэтому элемент массива в позиции 3 имеет значение D.

15.3.1. *Разбиение и слияние*

При добавлении в дучу новой вершины ей приписывается *случайный* вес. Из-за этого дерево с высокой вероятностью будет сбалансированным (высота равна $O(\log n)$), и операции с ним выполняются эффективно.

Разбиение. Операция разбиения создает из одной дучи две, в результате чего массив разбивается на два таким образом, что первые k элементов принадлежат первому массиву, а остальные – второму. Для этого мы создаем две новые, первоначально пустые дучи и обходим исходную дучу, начиная с корня. Если текущая вершина принадлежит левой дуче, то она сама и ее левое поддерево добавляются в левую дучу, после чего процедура рекурсивно применяется к правому поддереву. Наоборот, если текущая вершина принадлежит правой дуче, то она сама и ее правое поддерево добавляются в правую дучу, после чего процедура рекурсивно применяется к левому поддереву. Поскольку высота дучи равна $O(\log n)$, эта операция занимает время $O(\log n)$.

На рис. 15.19 показано, как массив из примера выше разбивается на два массива таким образом, что первый содержит первые пять элементов, а второй – последние три. Вершина D принадлежит левой дуче, поэтому мы добавляем ее вместе с левым поддеревом в левую дучу. Далее, вершина C принадлежит правой дуче, поэтому мы добавляем ее вместе с правым поддеревом в правую дучу. Наконец, вершина W добавляется в левую дучу, а вершина I – в правую.

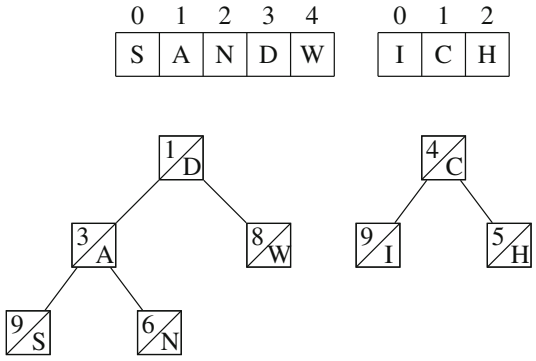


Рис. 15.19. Разбиение массива на два

Слияние. Операция слияния двух дуч создает новую дучу, соответствующую конкатенации двух массивов. Обе дучи обрабатываются одновременно, и на каждом шаге выбирается дуча с наименьшим весом корня. Если вес корня левой дучи меньше, то сам корень и его левое поддерево перемещаются в новую дучу, а его правое поддерево становится новым корнем

левой дучи. Аналогично, если корень правой дучи меньше, то сам корень и его правое поддерево перемещаются в новую дучу, а его левое поддерево становится новым корнем правой дучи. Поскольку высота дучи равна $O(\log n)$, эта операция занимает время $O(\log n)$.

Например, мы можем поменять местами два массива, а затем снова конкатенировать их. На рис. 15.20 показаны массивы до слияния, а на рис. 15.21 – конечный результат. Сначала вершина D и ее правое поддерево добавляются в новую дучу. Затем вершина A и ее правое поддерево становятся левым поддеревом вершины D. После этого вершина C и ее левое поддерево становятся левым поддеревом вершины A. Наконец, в новую дучу добавляются вершины H и S.

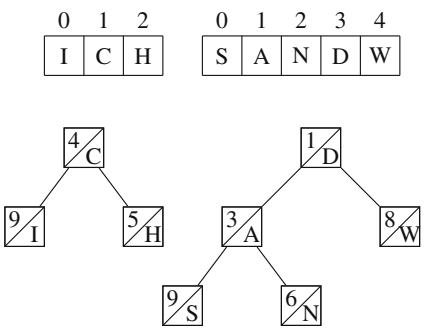


Рис. 15.20. Слияние двух массивов в один: до начала слияния

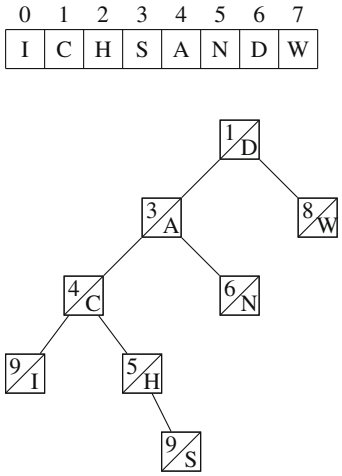


Рис. 15.21. Слияние двух массивов в один: после слияния

15.3.2. *Реализация*

Далее мы рассмотрим удобный способ реализации дучи. Ниже показана структура для хранения вершины дучи:

```
struct node {
    node *left, *right;
    int weight, size, value;
    node(int v) {
        left = right = NULL;
        weight = rand();
        size = 1;
        value = v;
    }
};
```

В поле `size` хранится размер поддерева вершины. Поскольку вершина может быть равна `NULL`, полезна следующая функция:

```
int size(node *treap) {
   if (treap == NULL) return 0;
   return treap->size;
}
```

Показанная ниже функция `split` реализует операцию разбиения. Она рекурсивно разбивает дучу `treap` на дучи `left` и `right`, так что левая дуча содержит первые *k* вершин и правая – все остальные.

```
void split(node *treap, node *&left, node *&right, int k) {
   if (treap == NULL) {
      left = right = NULL;
   } else {
      if (size(treap->left) < k) {
         split(treap->right, treap->right, right,
               k-size(treap->left)-1);
         left = treap;
      } else {
         split(treap->left, left, treap->left, k);
         right = treap;
      }
         treap->size = size(treap->left)+size(treap->right)+1;
      }
}
```

Функция `merge` реализует операцию слияния. Она создает дучу `treap`, которая содержит сначала вершины дучи `left`, а затем – вершины дучи `right`.

```
void merge(node *&treap, node *left, node *right) {
   if (left == NULL) treap = right;
   else if(right == NULL) treap = left;
   else {
      if (left->weight < right->weight) {
         merge(left->right, left->right, right);
         treap = left;
      } else {
         merge(right->left, left, right->left);
         treap = right;
      }
         treap->size = size(treap->left)+size(treap->right)+1;
      }
}
```

Например, в следующем фрагменте создается дуча, соответствующая массиву [1, 2, 3, 4]. Затем она разбивается на две дучи размера 2, которые переставляются местами, в результате чего создается дуча, соответствующая массиву [3, 4, 1, 2].

```
node *treap = NULL;
merge(treap, treap, new node(1));
merge(treap, treap, new node(2));
merge(treap, treap, new node(3));
merge(treap, treap, new node(4));
node *left, *right;
split(treap, left, right, 2);
merge(treap, right, left);
```

15.3.3. *Дополнительные методы*

Операции разбиения и слияния дуч весьма эффективны, поскольку позволяют произвольным образом «вырезать и копировать» массивы за логарифмическое время. Но дучи можно еще обобщить, так что они будут работать, почти как деревья отрезков. Например, помимо хранения размера каждого поддерева, мы можем также хранить сумму значений в нем, минимальное значение и т. д.

В частности, с помощью дуч можно выполнить интересный трюк: *обратить* массив. Для этого нужно переставить местами левого и правого потомков каждой вершины в дуче. На рис. 15.22 показано, что получается после обращения массива на рис. 15.18. Чтобы сделать это эффективно, мы можем ввести поле, которое показывает, следует ли обращать поддерево вершины, и выполнять операции перестановки лениво.

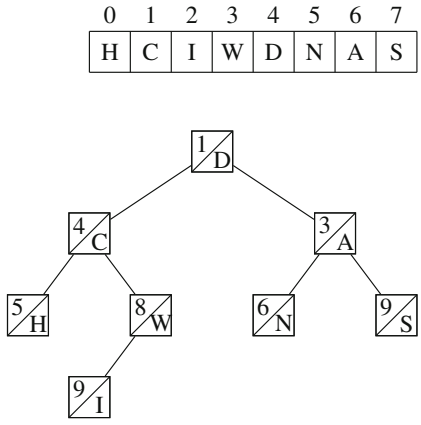


Рис. 15.22. Обращение массива с помощью дучи

15.4. Оптимизация динамического программирования

В этом разделе обсуждаются методы оптимизации динамического программирования. Сначала мы рассмотрим трюк с выпуклой оболочкой, которым можно воспользоваться для эффективного нахождения минималь-

ного значения семейства линейных функций. Затем мы обсудим еще два приема, основанных на свойствах функций стоимости.

15.4.1. *Трюк с выпуклой оболочкой*

Трюк с выпуклой оболочкой (convex hull trick) позволяет эффективно найти минимальное значение в заданной точке x семейства n линейных функций вида $f(x) = ax + b$. На рис. 15.23 показаны функции $f_1(x) = x + 2$, $f_2(x) = x/3 + 4$, $f_3(x) = x/6 + 5$ и $f_4(x) = -x/4 + 7$. Минимальное значение в точке $x = 4$ принимает функция $f_2(4) = 16/3$.

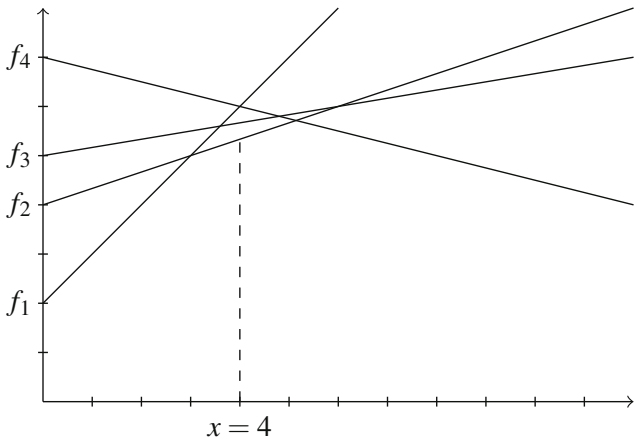


Рис. 15.23. Минимальное значение семейства функций в точке $x = 4$ равно $f_2(4) = 16/3$

Идея заключается в том, чтобы разбить ось x на диапазоны, в которых некоторая функция меньше всех остальных. Оказывается, что для каждой функции есть не более одного такого диапазона, и мы можем сохранить их в отсортированном списке, содержащем не более n диапазонов. На рис. 15.24 показаны диапазоны для нашего примера. Сначала наименьшей является функция f_1, затем f_2 и, наконец, f_4. Отметим, что f_3 ни в одной точке не принимает значение, которое было бы меньше значений всех остальных функций.

Имея список диапазонов, мы можем найти минимальное значение семейства функций в точке x за время $O(\log n)$, применив двоичный поиск. Например, поскольку на рис. 15.24 точка $x = 4$ принадлежит диапазону функции f_2, мы сразу заключаем, что минимальное значение в этой точке принимает функция f_2, и оно равно $f_2(4) = 16/3$. Таким образом, множество k запросов можно обработать за время $O(k \log n)$. Кроме того, если запросы следуют в порядке возрастания, то их можно обработать за время $O(k)$, просто обходя диапазоны слева направо.

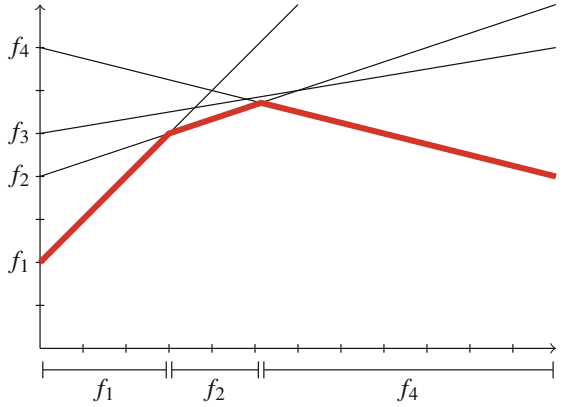


Рис. 15.24. Диапазоны, в которых минимальны функции f_1, f_2 и f_4

Но как определить эти диапазоны? Если функции заданы в порядке убывания коэффициентов наклона, то диапазоны найти легко, поскольку мы можем вести стек диапазонов. Тогда амортизированная стоимость обработки каждой функции равна $O(1)$. Если функции можно задавать в произвольном порядке, то потребуется более сложная структура множества, и обработка каждой функции займет время $O(\log n)$.

Пример. Пусть имеется n последовательных концертов. Билет на i-й концерт стоит p_i рублей, а если мы посещаем этот концерт, то получаем скидочный купон величиной d_i ($0 < d_i < 1$). Впоследствии этот купон можно использовать для покупки билета за $d_i p$ рублей вместо p. Известно также, что $d_i \geqslant d_{i+1}$ для всех последовательных концертов i и $i + 1$. Мы хотим обязательно посетить последний концерт, но не против посетить также и другие. В какую минимальную цену мы можем уложиться?

Эту задачу легко решить методом динамического программирования, если вычислять для каждого концерта i значение u_i – минимальную цену посещения концерта i и, возможно, некоторых других. Чтобы найти оптимальный выбор для некоторого концерта, мы можем просто перебрать все предыдущие концерты за время $O(n)$, так что в итоге получается алгоритм с временной сложностью $O(n^2)$. Однако, воспользовавшись трюком с выпуклой оболочкой, мы сможем найти оптимальное решение за время $O(\log n)$ и получить алгоритм с временной сложностью $O(n \log n)$.

Идея заключается в том, чтобы построить семейство линейных функций, первоначально содержащее только функцию $f(x) = x$, означающую, что скидочного купона нет. Чтобы вычислить значение u_i для некоторого концерта, мы найдем в нашем семействе функцию f, для которой достигается минимум $f(p_i)$; это можно сделать за время $O(\log n)$, применив трюк с выпуклой оболочкой. Затем мы добавляем в семейство функцию $f(x) = d_i x + u_i$ и можем использовать ее для посещения какого-то из последующих концертов. Этот алгоритм работает за время $O(n \log n)$.

Отметим, что если дополнительно известно, что $p_i \leqslant p_{i+1}$ для всех соседних концертов i и $i + 1$, то задачу можно решить еще эффективнее за время $O(n)$, поскольку мы можем не использовать двоичный поиск, а обработать диапазоны слева направо и найти оптимальное решение за амортизированное постоянное время.

15.4.2. *Оптимизация методом «разделяй и властвуй»*

Оптимизация методом «разделяй и властвуй» применима к некоторым задачам динамического программирования, когда последовательность n элементов $s_1, s_2, \ldots, s_n$ нужно разбить на k подпоследовательностей, состоящих из соседних элементов. Задана функция $\texttt{cost}(a, b)$, которая определяет стоимость создания подпоследовательности $s_a, s_{a+1}, \ldots, s_b$. Общая стоимость разбиения равна сумме стоимостей отдельных подпоследовательностей, а наша задача состоит в том, чтобы найти разбиение, минимизирующее общую стоимость.

Предположим, к примеру, что имеется последовательность положительных целых чисел и $\texttt{cost}(a, b) = (s_a + s_{a+1} + \ldots + s_b)^2$. На рис. 15.25 показан оптимальный способ разбить последовательность на три подпоследовательности при такой функции стоимости. Общая стоимость этого разбиения равна $(2 + 3 + 1)^2 + (2 + 2 + 3)^2 + (4 + 1)^2 = 110$.

Для решения задачи определим функцию $\texttt{solve}(i, j)$, которая дает минимальную общую стоимость разбиения первых i элементов $s_1, s_2, \ldots, s_i$ на j подпоследовательностей. Очевидно, что значение $\texttt{solve}(n, k)$ и есть ответ на вопрос. Чтобы вычислить $\texttt{solve}(i, j)$, мы должны найти индекс $1 \leqslant p \leqslant i$, при котором достигается минимум выражения

$$\texttt{solve}(p - 1, j - 1) + \texttt{cost}(p, i).$$

На рис. 15.25 оптимальным решением для $\texttt{solve}(8, 3)$ является $p = 7$. Чтобы найти его, можно просто проверить все индексы $1, 2, \ldots, i$, что займет время $O(n)$. Если таким образом вычислять все значения $\texttt{solve}(i, j)$, то мы получим алгоритм динамического программирования с временной сложностью $O(n^2k)$. Однако, применив оптимизацию методом «разделяй и властвуй», мы можем улучшить временную сложность до $O(nk \log n)$.

1	2	3	4	5	6	7	8
2	3	1	2	2	3	4	1

Рис. 15.25. Оптимальный способ разбить последовательность на три блока

Такую оптимизацию можно использовать, если функция стоимости удовлетворяет *неравенству четырехугольника*

$$\texttt{cost}(a, c) + \texttt{cost}(b, d) \leqslant \texttt{cost}(a, d) + \texttt{cost}(b, c)$$

для любых $a \leqslant b \leqslant c \leqslant d$. Обозначим $\texttt{pos}(i, j)$ наименьший индекс p, при котором достигается минимум стоимости разбиения для $\texttt{solve}(i, j)$. Если

приведенное выше неравенство удовлетворяется, то гарантируется, что pos(i, j) ⩽ pos(i + 1, j) для всех i и j, что позволяет вычислять значения solve(i, j) более эффективно.

Идея заключается в том, чтобы определить функцию calc(j, a, b, x, y), которая вычисляет все значения solve(i, j) для $a \leqslant i \leqslant b$ и фиксированного j, пользуясь тем фактом, что $x \leqslant$ pos(i, j) $\leqslant y$. Функция сначала вычисляет solve(z, j), где $z = \lfloor (a + b)/2 \rfloor$, а затем рекурсивно вызывает calc(j, a, $z - 1$, x, p) и calc(j, $z + 1$, b, p, y), где p = pos(z, j). Тот факт, что pos(i, j) ⩽ pos(i + 1, j), используется для ограничения области поиска. Для вычисления всех значений solve(i, j) мы выполняем вызов calc(j, 1, n, 1, n) для каждого $j = 1, 2, \ldots, k$. Поскольку каждый такой вызов занимает время $O(n \log n)$, построенный алгоритм имеет временную сложность $O(nk \log n)$.

Наконец, докажем, что используемая в нашем примере функция стоимости – сумма квадратов – удовлетворяет неравенству четырехугольника. Обозначим sum(a, b) сумму значений в диапазоне [a, b], и пусть x = sum(b, c), y = sum(a, c) − sum(b, c), z = sum(b, d) − sum(b, c). В этих обозначениях неравенство четырехугольника принимает вид

$$(x + y)^2 + (x + z)^2 \leqslant (x + y + z)^2 + x^2,$$

что эквивалентно неравенству

$$0 \leqslant 2yz.$$

Поскольку y и z неотрицательны, утверждение доказано.

15.4.3. *Оптимизация Кнута*

Оптимизацию Кнута[3] можно использовать для решения некоторых задач динамического программирования, в которых требуется разбить последовательность n элементов $s_1, s_2, \ldots, s_n$ на отдельные элементы, применяя операции разделения. Функция стоимости cost(a, b) определяет стоимость обработки последовательности $s_a, s_{a+1}, \ldots, s_b$, а наша задача – найти решение, минимизирующее суммарную стоимость разделения.

Пусть, например, cost(a, b) = $s_a + s_{a+1} + \ldots + s_b$. На рис. 15.26 показан оптимальный способ обработки последовательности в этом случае. Полная стоимость этого решения равна 19 + 9 + 10 + 5 = 43.

Эту задачу можно решить, определив функцию solve(i, j), которая дает минимальную стоимость разбиения последовательности $s_i, s_{i+1}, \ldots, s_j$ на отдельные элементы. Тогда значение solve(1, n) дает искомый ответ. Чтобы вычислить значение solve(i, j), мы должны найти индекс $i \leqslant p < j$, при котором достигает минимума величина

$$\text{cost}(i, j) + \text{solve}(i, p) + \text{solve}(p + 1, j).$$

[3] Кнут [23] применил эту оптимизацию для построения оптимального двоичного дерева поиска. Впоследствии Яо [36] обобщил предложенный подход на другие похожие задачи.

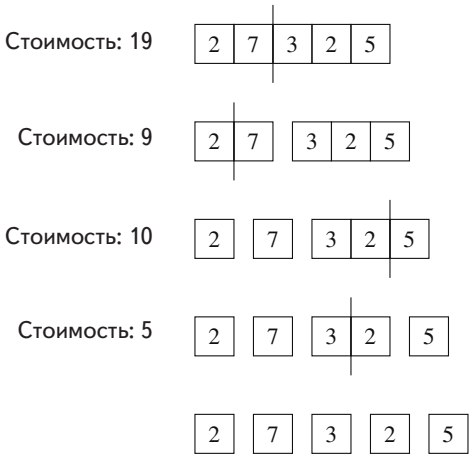


Рис. 15.26. Оптимальный способ разбиения массива на отдельные элементы

Если мы будем проверять все индексы между i и j, то получим задачу динамического программирования с временной сложностью $O(n^3)$. А с помощью оптимизации Кнута мы сможем вычислить значения $\texttt{solve}(i, j)$ более эффективно за время $O(n^2)$.

Оптимизация Кнута применима, если

$$\texttt{cost}(b, c) \leqslant \texttt{cost}(a, d)$$

и

$$\texttt{cost}(a, c) + \texttt{cost}(b, d) \leqslant \texttt{cost}(a, d) + \texttt{cost}(b, c)$$

для любых $a \leqslant b \leqslant c \leqslant d$. Отметим, что второе неравенство – не что иное, как неравенство четырехугольника, которое встречалось и в оптимизации методом «разделяй и властвуй». Обозначим $\texttt{pos}(i, j)$ наименьший индекс p, при котором достигается минимум стоимости вычисления $\texttt{solve}(i, j)$. Если приведенные выше неравенства удовлетворяются, то

$$\texttt{pos}(i, j - 1) \leqslant \texttt{pos}(i, j) \leqslant \texttt{pos}(i + 1, j).$$

Теперь можно выполнить n раундов – $1, 2, \ldots, n$ – вычисляя на k-м раунде значения $\texttt{solve}(i, j)$, где $j - i + 1 = k$, т. е. обрабатывать подпоследовательности в порядке возрастания длины. Поскольку известно, что $\texttt{pos}(i, j)$ должно быть заключено между $\texttt{pos}(i, j - 1)$ и $\texttt{pos}(i + 1, j)$, то каждый раунд можно выполнить за время $O(n)$, и полная временная сложность алгоритма оказывается равна $O(n^2)$.

15.5. Методы перебора с возвратом

В этом разделе мы покажем, как ускорить работу алгоритмов перебора с возвратом. Сначала рассмотрим задачу о подсчете количества путей на

сетке и попробуем улучшить алгоритм с помощью отсечения ветвей дерева поиска. Затем решим задачу об игре в 15, применив алгоритм IDA* и эвристические функции.

15.5.1. *Отсечение ветвей дерева поиска*

Многие алгоритмы перебора с возвратом можно улучшить путем *отсечения ветвей* дерева поиска: заметив, что частичное решение нельзя продолжить до полного, не имеет смысла продолжать поиск.

Рассмотрим задачу о вычислении количества путей на доске 7×7 таких, что каждый путь начинается в левом верхнем углу, заканчивается в правом нижнем углу и заходит в каждую клетку ровно один раз. Один такой путь показан на рис. 15.27, а всего их, как можно показать, 111 712.

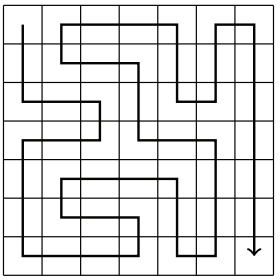


Рис. 15.27. Путь из левого верхнего угла в правый нижний

Мы начнем с простого алгоритма перебора с возвратом, а затем постепенно оптимизируем его, заметив, как можно сократить поиск. После каждой оптимизации будем измерять время работы алгоритма и количество рекурсивных вызовов, чтобы оценить влияние оптимизации на эффективность поиска.

Базовый алгоритм. В первой версии алгоритма нет никаких оптимизаций. Мы просто пользуемся перебором с возвратом, чтобы породить все возможные пути из левого верхнего угла в правый нижний, и подсчитываем число таких путей.

- Время работы: 483 с.
- Количество рекурсивных вызовов: $76 \cdot 10^9$.

Оптимизация 1. В любом решении мы сначала делаем один шаг вниз или вправо, и существует два пути, симметричных относительно диагонали доски. Например, пути на рис. 15.28 симметричны. Поэтому мы можем всегда делать первый шаг вниз (или, наоборот, вправо), а в конце умножить количество решений на два.

- Время работы: 244 с.
- Количество рекурсивных вызовов: $38 \cdot 10^9$.

Оптимизация 2. Если путь заходит в правый нижний угол, не посетив все остальные клетки, то очевидно, что получить корректное решение уже не удастся. Пример показан на рис. 15.29. Следовательно, поиск можно завершать немедленно, если мы попали в правую нижнюю клетку слишком рано.

- Время работы: 119 с.
- Количество рекурсивных вызовов: $20 \cdot 10^9$.

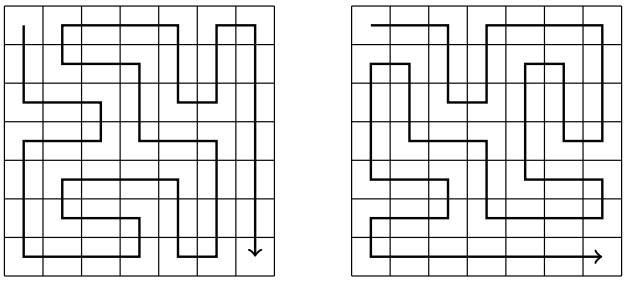


Рис. 15.28. Два пути, симметричных относительно диагонали

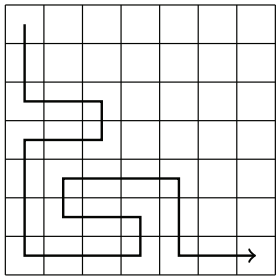


Рис. 15.29. Мы дошли до правого нижнего угла, не посетив все остальные клетки

Оптимизация 3. Если путь упирается в край и может повернуть как налево, так и направо, то доска оказалась разбита на две части, в каждой из которых есть непосещенные клетки. Например, путь на рис. 15.30 может повернуть налево или направо. В таком случае нам точно не удастся посетить все клетки, поэтому можно завершать поиск. Эта оптимизация оказалась очень полезной.

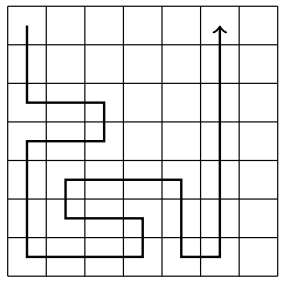


Рис. 15.30. Путь разбивает доску на две части, содержащие непосещенные клетки

- Время работы: 1,8 с.
- Количество рекурсивных вызовов: $221 \cdot 10^6$.

Оптимизация 4. Идею предыдущей оптимизации можно обобщить: если путь нельзя продолжить вперед, но можно повернуть налево или направо, то доска разбивается на две части, каждая из которых содержит непосещенные клетки. На рис. 15.31 приведен пример такого случая. Очевидно, что нам не удастся посетить все клетки, поэтому можно завершать поиск. После этой оптимизации поиск становится очень эффективным.

- Время работы: 0,6 с.
- Количество рекурсивных вызовов: $69 \cdot 10^6$.

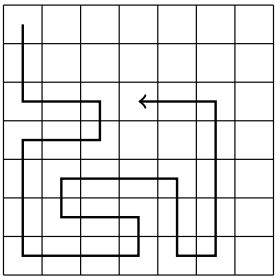


Рис. 15.31. Более общая ситуация, когда путь разбивает доску на две части

Вывод. Теперь самое время прекратить попытки оптимизации алгоритма и посмотреть, чего мы достигли. Время работы первой версии алгоритма составляло 483 с, а после всех оптимизаций уменьшилось до 0,6 с. То есть благодаря оптимизациям алгоритм стал работать почти в 1000 раз быстрее.

Это типично для перебора с возвратом, потому что дерево поиска обычно велико, и даже простые наблюдения могут заметно уменьшить количество ветвей в нем. Особенно полезны оптимизации на первых шагах алгоритма, т. е. в самом начале дерева поиска.

15.5.2. *Эвристические функции*

В некоторых задачах перебора с возвратом мы ищем оптимальное решение, например самую короткую последовательность ходов. В таких случаях поиск можно улучшить, воспользовавшись *эвристической функцией*, которая оценивает расстояние от текущего состояния поиска до конечного состояния.

В игре в 15 имеется доска 4×4, а на ней 15 костяшек (с числами от 1 до 15), так что одна клетка пустая. На каждом ходе мы можем выбрать любую костяшку, соседнюю с пустой клеткой, и переместить ее в пустую клетку. Мы хотим найти минимальное количество ходов, требуемое для получения конечной позиции, показанной на рис. 15.32.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Рис. 15.32. Конечная позиция в игре в 15

Для решения задачи мы воспользуемся алгоритмом IDA*, который состоит из нескольких поисков методом перебора с возвратом. В каждом поиске мы пытаемся найти решение с количеством ходов не более k. Вначале k равно 0, и после каждого неудачного поиска увеличивается на 1.

В алгоритме используется эвристическая функция, которая оценивает количество ходов, оставшееся до конца игры. Эвристическая функция должна быть *допустимой*, т. е. она никогда не должна давать завышенной оценки числа ходов. Таким образом, с ее помощью мы получаем нижнюю границу числа ходов.

11	3	12	9
8	15	6	5
14		10	2
7	13	1	4

Рис. 15.33. Минимальное количество ходов в этой позиции равно 61

Для примера рассмотрим позицию на рис. 15.33. Как выясняется, минимальное число ходов в ней равно 61.

Поскольку в каждом состоянии имеется от 2 до 4 возможных ходов в зависимости от положения пустой клетки, бесхитростный алгоритм перебора с возвратом занял бы слишком много времени. По счастью, алгоритм IDA* включает эвристическую функцию, которая значительно ускоряет поиск.

Далее мы рассмотрим несколько эвристических функций и измерим время работы алгоритма и количество рекурсивных вызовов. Во всех эвристиках мы реализуем перебор с возвратом таким образом, что предыдущий ход никогда не отменяется, потому что это не привело бы к оптимальному решению.

Эвристика 1. Простая эвристика заключается в том, чтобы для каждого квадратика вычислить манхэттенское расстояние от его текущего положения до конечного. Манхэттенское расстояние вычисляется по формуле $|x_c - x_f| + |y_c - y_f|$, где (x_c, y_c) – текущее положение квадратика, а (x_f, y_f) – его конечное положение. Нижнюю границу числа ходов мы получим, просуммировав все такие расстояния, поскольку каждый ход изменяет положение костяшки по вертикали или по горизонтали на 1.

- Время работы: 126 с.
- Число рекурсивных вызовов: $1{,}5 \cdot 10^9$.

Эвристика 2. Эвристику получше мы создадим, сконцентрировав внимание на костяшках, которые уже находятся в правильной строке или правильном столбце. Рассмотрим костяшки 6 и 8 в нашем примере. Они уже стоят в правильной строке, правда, не в том порядке. Чтобы поменять порядок, нужно будет сдвинуть одну из них по вертикали, что добавляет два хода.

Таким образом, мы можем улучшить эвристику следующим образом: сначала вычислить сумму манхэттенских расстояний, а затем добавить два дополнительных хода для каждой строки или столбца, где две костяшки, находящиеся в своей строке (своем столбце), но неправильно расположенные друг относительно друга.

- Время работы: 22 с.
- Число рекурсивных вызовов: $1{,}43 \cdot 10^8$.

Эвристика 3. Мы можем еще улучшить предыдущую эвристику: если имеется больше двух костяшек, находящихся в правильной строке или в правильном столбце, то, возможно, мы сможем добавить еще два дополнительных хода. Например, рассмотрим костяшки 5, 6 и 8. Поскольку они стоят в порядке, обратном нужному, нам придется сдвинуть по крайней мере две из них по вертикали, что даст четыре дополнительных хода.

Вообще, если имеется c_1 костяшек, занимающих правильную строку или столбец, и не более c_2 из них стоят в нужном порядке, то можно добавить $2(c_1 - c_2)$ дополнительных ходов.

- Время работы: 39 с.
- Число рекурсивных вызовов: $1{,}36 \cdot 10^8$.

Вывод. А что случилось? Мы улучшили эвристику, но время работы алгоритма увеличилось с 22 с до 39 с.

Хорошая эвристика обладает двумя свойствами: она дает нижнюю границу, близкую к истинному расстоянию, и может быть эффективно вычислена. Последняя эвристика точнее предпоследней, но ее труднее вычислить, поэтому в данном случае более простая эвристика предпочтительнее.

15.6. Разное

В этом разделе представлена подборка методов проектирования алгоритмов. Мы обсудим технику встречи в середине, алгоритм динамического программирования для подсчета подмножеств, метод параллельного двоичного поиска и пакетное решение задачи о динамической связности.

15.6.1. *Встреча в середине*

Идея техники *встречи в середине* состоит в том, чтобы разбить пространство поиска на две части приблизительно равного размера, выполнить поиск в каждой части, а затем объединить результаты двух поисков. Эта техника позволяет ускорить некоторые алгоритмы с временной сложностью $O(2^n)$, уменьшив время работы до $O(2^{n/2})$. Отметим, что $O(2^{n/2})$ гораздо быстрее, чем $O(2^n)$, поскольку $2^{n/2} = \sqrt{2^n}$. С помощью алгоритма с временной сложностью $O(2^n)$ мы можем обработать входные данные, для которых $n \approx 20$, а если временная сложность уменьшается до $O(2^{n/2})$, то оценка n возрастает до $n \approx 40$.

Пусть дано множество n целых чисел и требуется определить, существует ли в нем подмножество с суммой x. Например, если дано множество $\{2, 4, 5, 9\}$ и $x = 15$, то мы можем выбрать подмножество $\{2, 4, 9\}$, поскольку $2 + 4 + 9 = 15$. Эту задачу легко решить за время $O(2^n)$, перебрав все возможные подмножества, но мы решим ее более эффективно за время $O(2^{n/2})$, воспользовавшись техникой встречи в середине.

Идея в том, чтобы разделить наше множество на два подмножества A и B, так чтобы в каждом была примерно половина чисел. Мы выполняем два поиска: первый порождает все подмножества A и сохраняет их суммы в списке S_A, второй создает аналогичный список S_B для B. После этого достаточно проверить, можно ли выбрать один элемент из S_A и другой элемент из S_B, так чтобы их сумма была равна x; это возможно тогда и только тогда, когда в исходном множестве имеется подмножество с суммой x.

Посмотрим, к примеру, как обрабатывается множество $\{2, 4, 5, 9\}$. Сначала мы разбиваем множество на два подмножества $A = \{2, 4\}$ и $B = \{5, 9\}$.

После этого создаем списки S_A = [0, 2, 4, 6] и S_B = [0, 5, 9, 14]. Поскольку S_A содержит сумму 6, а S_B содержит сумму 9, мы заключаем, что в исходном множестве имеется подмножество с суммой 6 + 9 = 15.

При хорошей реализации списки S_A и S_B можно создать за время $O(2^{n/2})$ таким образом, что оба списка будут отсортированы. После этого можно воспользоваться алгоритмом двух указателей и за время $O(2^{n/2})$ проверить, можно ли составить сумму x из двух слагаемых, взятых по одному из S_A и S_B. Таким образом, полная временная сложность алгоритма равна $O(2^{n/2})$.

15.6.2. *Подсчет подмножеств*

Пусть $X = \{0 \ldots n-1\}$, и каждому подмножеству $S \subset X$ сопоставлена ценность – целое число value[S]. Требуется для каждого S вычислить

$$\texttt{sum}[S] = \sum_{A \subset S} \texttt{value}[A],$$

т. е. сумму ценностей подмножеств S.

Пусть, например, $n = 3$ и ценности таковы:

- value[∅] = 3;
- value[{0}] = 1;
- value[{1}] = 4;
- value[{0, 1}] = 5;
- value[{2}] = 5;
- value[{0, 2}] = 1;
- value[{1, 2}] = 3;
- value[{0, 1, 2}] = 3.

Тогда

$$\begin{aligned}\texttt{sum}(\{0, 2\}) &= \texttt{value}[\varnothing] + \texttt{value}[\{0\}] + \texttt{value}[\{2\}] + \texttt{value}[\{0, 2\}] \\ &= 3 + 1 + 5 + 1 = 10.\end{aligned}$$

Ниже мы покажем, как решить эту задачу за время $O(2^n n)$, применив динамическое программирование и поразрядные операции. Идея в том, чтобы рассмотреть подзадачи, в которых имеются ограничения на то, какие элементы можно удалять из S.

Обозначим partial(S, k) сумму ценностей подмножеств S с тем ограничением, что удалять из S можно только элементы 0 … k. Например,

$$\texttt{partial}(\{0, 2\}, 1) = \texttt{value}[\{2\}] + \texttt{value}[\{0, 2\}],$$

поскольку разрешено удалять лишь элементы 0 … 1. Отметим, что, зная partial, мы можем вычислить любое значение sum(S), потому что

$$\texttt{sum}(S) = \texttt{partial}(S, n-1).$$

Чтобы воспользоваться динамическим программированием, мы должны найти рекуррентное соотношение для partial. Базой рекурсии является случай

$$\texttt{partial}(S, -1) = \texttt{value}[S],$$

поскольку из S нельзя удалить ни одного элемента. А в общем случае имеет место соотношение:

$$\texttt{partial}(S, k) = \begin{cases} \texttt{partial}(S, k-1) & k \notin S \\ \texttt{partial}(S, k-1) + \texttt{partial}(S \setminus \{k\}, k-1) & k \in S \end{cases}$$

Здесь наше внимание направлено на элемент k. Если $k \in S$, то есть два варианта: либо мы оставляем k в подмножестве, либо удаляем из подмножества.

Реализация. Существует очень красивый способ реализовать решение методом динамического программирования с помощью поразрядных операций. А именно объявим массив:

```
int sum[1<<N];
```

который будет содержать суммы по каждому подмножеству. Массив инициализируется следующим образом:

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

После этого массив заполняется так:

```
for (int k = 0; k < n; k++) {
    for (int s = 0; s < (1<<n); s++) {
        if (s&(1<<k)) sum[s] += sum[s^(1<<k)];
    }
}
```

Здесь мы записываем вычисленные значения partial(S, k) для $k = 0 \ldots n-1$ в массив sum. Поскольку partial(S, k) зависит только от partial(S, $k-1$), то мы можем использовать массив sum повторно, что дает очень эффективную реализацию.

15.6.3. *Параллельный двоичный поиск*

Техника *параллельного двоичного поиска* позволяет повысить эффективность некоторых алгоритмов, основанных на двоичном поиске. Общая идея заключается в том, чтобы выполнять несколько двоичных поисков одновременно, а не по отдельности.

Рассмотрим следующую задачу. Имеется n городов, пронумерованных $1, 2, \ldots, n$. Первоначально между городами нет дорог. Затем каждый день в течение m дней между какими-то двумя городами строится новая дорога. И наконец, дано k запросов вида (a, b), а наша задача – для каждого запроса определить самый ранний момент, когда можно будет проехать между го-

родами a и b. Мы можем предполагать, что все встречающиеся в запросах пары городов будут соединены через m дней.

На рис. 15.34 приведен пример для четырех городов. Предположим, что предъявлены запросы $q_1 = (1, 4)$ и $q_2 = (2, 3)$. Ответ на запрос q_1 равен 2, поскольку города 1 и 4 оказываются соединены на второй день, а на запрос q_2 – 4, поскольку города 2 и 3 будут соединены на четвертый день.

Сначала рассмотрим более простую задачу, когда имеется всего один запрос (a, b). В этом случае можно воспользоваться системой непересекающихся множеств для моделирования процесса добавления дорог. После строительства каждой новой дороги мы проверяем, соединены ли города a и b, и если да, то останавливаемся. Как добавление новой дороги, так и проверка наличия соединения занимают время $O(\log n)$, поэтому алгоритм имеет временную сложность $O(m \log n)$.

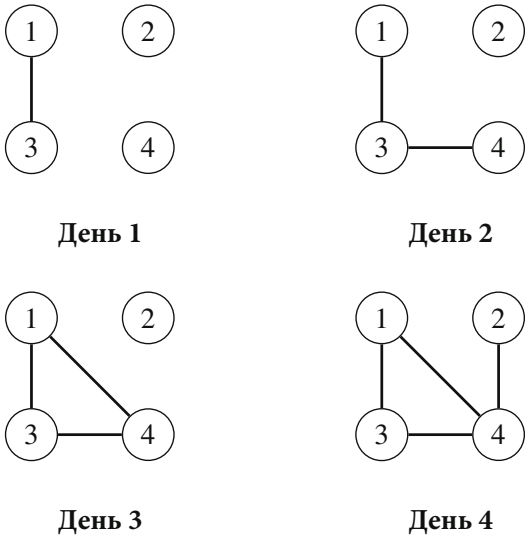


Рис. 15.34. Пример задачи о строительстве дороги

Как обобщить это решение на k запросов? Конечно, можно было бы обрабатывать каждый запрос по отдельности, но такой алгоритм потребовал бы времени $O(km \log n)$ – слишком много, если k и m велики. Ниже мы покажем, как решить задачу более эффективно, воспользовавшись параллельным двоичным поиском.

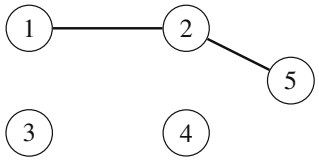
Идея в том, чтобы сопоставить каждому запросу диапазон $[x, y]$, означающий, что первое соединение между городами появляется не раньше, чем на x-й день, и не позже, чем на y-й день. Вначале все диапазоны равны $[1, m]$. Теперь $\log m$ раз смоделируем процесс добавления всех дорог в сеть, применяя систему непересекающихся множеств. Для каждого запроса проверяем, соединены ли города в момент $u = \lfloor (x + y)/2 \rfloor$. Если да, то новым диапазоном становится $[x, u]$, иначе $[u + 1, y]$. После $\log m$ раундов каждый

диапазон состоит всего из одного момента, который и является ответом на запрос. На каждом раунде мы добавляем в сеть m дорог за время $O(m \log n)$ и проверяем, соединены ли k пар городов, за время $O(k \log n)$. Поскольку всего раундов $\log m$, получившийся алгоритм имеет временную сложность $O((m + k) \log n \log m)$.

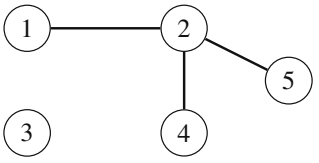
15.6.4. *Динамическая связность*

Пусть имеется граф, содержащий n вершин и m ребер. И пусть задано q запросов вида «добавить ребро между вершинами a и b» или «удалить ребро между вершинами a и b». Требуется эффективно вычислять количество компонент связности графа после каждого запроса.

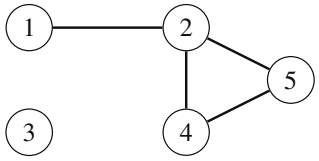
На рис. 15.35 приведен пример выполнения этого процесса. Вначале граф состоит из трех компонент. Затем добавляется ребро 2–4, соединяющее две компоненты. После этого добавляется ребро 4–5 и удаляется ребро 2–5, но количество компонент не изменяется. Далее добавляется ребро 1–3, соединяющее две компоненты, и, наконец, удаляется ребро 2–4, в результате чего одна компонента разделяется на две.



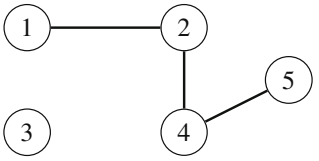
Начальный граф
Количество компонент: 3



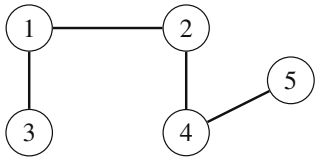
Шаг 1: добавление ребра 2—4
Количество компонент: 2



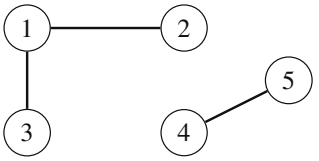
Шаг 2: добавление ребра 4—5
Количество компонент: 2



Шаг 3: добавление ребра 2—5
Количество компонент: 2



Шаг 4: добавление ребра 1—3
Количество компонент: 1



Шаг 5: добавление ребра 2—4
Количество компонент: 2

Рис. 15.35. Задача о динамической связности

Если бы было разрешено только добавлять ребра, то задачу было бы легко решить с помощью системы непересекающихся множеств, но операции удаления сильно усложняют дело. Мы обсудим алгоритм типа «разделяй и властвуй» решения пакетной версии этой задачи, когда все запросы известны заранее, а результаты разрешено выдавать в любом порядке. Описанный алгоритм основан на работе Kopeliovich [25].

Идея заключается в том, чтобы создать временную шкалу, на которой каждое ребро представлено интервалом, показывающим моменты его вставки и удаления. Временная шкала охватывает диапазон [0, q + 1], а ребро, добавленное на шаге a и удаленное на шаге b, представлено интервалом [a, b]. Если ребро принадлежало начальному графу, то a = 0, а если ребро не удалялось, то $b = q + 1$. На рис. 15.36 показана временная шкала для нашего примера.

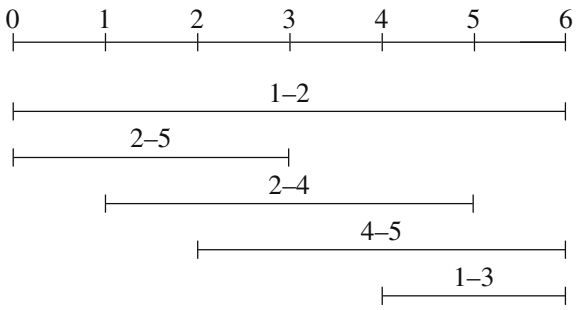


Рис. 15.36. Временная шкала вставок и удалений

Для обработки интервалов мы создадим граф, имеющий n вершин и ни одного ребра, и воспользуемся рекурсивной функцией, которая вызывается с диапазоном [0, q + 1] в качестве параметра. Для диапазона [a, b] функция работает следующим образом. Сначала если [a, b] целиком расположен внутри интервала некоторого ребра и это ребро не принадлежит графу, то оно добавляется в граф. Затем если размер [a, b] равен 1, то мы выводим количество компонент связности, а в противном случае рекурсивно обрабатываем диапазоны [a, k] и [k, b], где $k = \lfloor (a + b)/2 \rfloor$. Наконец, мы удаляем все ребра, которые были добавлены в начале обработки диапазона [a, b].

Всякий раз при добавлении или удалении ребра мы также обновляем количество компонент. Это можно сделать с помощью системы непересекающихся множеств, поскольку мы всегда удаляем ребро, добавленное последним. Следовательно, достаточно реализовать для системы непересекающихся множеств операцию *отмены* (undo), и это вполне возможно, если хранить информацию об операциях в стеке. Поскольку каждое ребро добавляется и удаляется не более $O(\log q)$ раз и каждая операция занимает время $O(\log n)$, полное время работы алгоритма составляет $O((m + q) \log q \log n)$.

Отметим, что, помимо подсчета количества компонент связности, мы можем поддержать любую информацию, которую можно объединить с системой непересекающихся множеств. Например, можно подсчитывать количество вершин в самой большой компоненте или определять двудольность каждой компоненты. Эту технику можно также обобщить на другие структуры данных, поддерживающие операции вставки и отмены.

Приложение

Сведения из математики

Формулы сумм

Любую сумму вида

$$\sum_{x=1}^{n} x^k = 1^k + 2^k + 3^k + \cdots + n^k,$$

где k – целое положительное число, можно записать в виде полинома степени $k + 1$. Например[1]:

$$\sum_{x=1}^{n} x = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

и

$$\sum_{x=1}^{n} x^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Арифметической прогрессией называется такая последовательность чисел, что разность между любыми двумя соседними членами постоянна. Например:

$$3, 7, 11, 15$$

– арифметическая прогрессия с разностью 4. Сумма арифметической прогрессии вычисляется по формуле

$$\underbrace{a + \cdots + b}_{n \text{ чисел}} = \frac{n(a+b)}{2},$$

где a – первый член, b – последний член, а n – количество членов. Например:

$$3 + 7 + 11 + 15 = \frac{4 \cdot (3 + 15)}{2} = 36.$$

[1] Существует даже общая формула для таких сумм, называемая *формулой Фаульхабера*, но она слишком сложная, чтобы приводить ее здесь.

Эта формула основана на том факте, что сумма состоит из n слагаемых и каждое слагаемое в среднем равно $(a + b)/2$.

Геометрической прогрессией называется такая последовательность чисел, что отношение любых двух соседних чисел постоянно (оно называется *знаменателем* прогрессии). Например:

$$3, 6, 12, 24$$

– геометрическая прогрессия со знаменателем 2. Сумма геометрической прогрессии вычисляется по формуле

$$a + ak + ak^2 + \ldots + b = \frac{bk - a}{k - 1},$$

где a – первый член, b – последний член, а k – знаменатель. Например:

$$3 + 6 + 12 + 24 = \frac{24 \cdot 2 - 3}{2 - 1} = 45.$$

Приведем вывод этой формулы. Положим

$$S = a + ak + ak^2 + \ldots + b.$$

Умножив обе стороны равенства на k, получим

$$kS = ak + ak^2 + ak^3 + \ldots + bk,$$

и, решив уравнение

$$kS - S = bk - a,$$

приходим к указанной выше формуле.

Частным случаем формулы суммы геометрической прогрессии является формула

$$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1.$$

Частичной суммой гармонического ряда называется сумма вида

$$\sum_{x=1}^{n} \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Эту сумму можно оценить сверху величиной $\log_2(n)+1$. Действительно, заменим каждый член $1/k$ числом вида $1/m$, где m – ближайшая степень двойки, не превосходящая k. Например, при $n = 6$ получаем следующую оценку:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \le 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

Эту оценку сверху можно представить в виде $\log_2(n) + 1$ частей ($1, 2 \cdot 1/2, 4 \cdot 1/4$ и т. д.), причем значение каждой части не превосходит 1.

Множества

Множеством называется набор элементов. Например, множество

$$X = \{2, 4, 7\}$$

содержит элементы 2, 4 и 7. Символом $\emptyset$ обозначается пустое множество, а символом $|S|$ – размер множества S, т. е. количество элементов в нем. Так, для приведенного выше множества $|X| = 3$. Если множество S содержит элемент x, то мы пишем $x \in S$, в противном случае – $x \notin S$. Так, в примере выше $4 \in X$, а $5 \notin X$.

Новые множества можно строить, применяя операции над множествами:

- *пересечение* $A \cap B$ содержит элементы, принадлежащие обоим множествам A и B. Например, если $A = \{1, 2, 5\}$ и $B = \{2, 4\}$, то $A \cap B = \{2\}$;
- *объединение* $A \cup B$ содержит элементы, принадлежащие либо A, либо B, либо тому и другому. Например, если $A = \{3, 7\}$ и $B = \{2, 3, 8\}$, то $A \cup B = \{2, 3, 7, 8\}$;
- *дополнение* $\overline{A}$ содержит элементы, не принадлежащие A. Интерпретация дополнения зависит от *универсального множества*, которое содержит все возможные элементы. Например, если $A = \{1, 2, 5, 7\}$, а универсальным считается множество $\{1, 2, \ldots, 10\}$, то $\overline{A} = \{3, 4, 6, 8, 9, 10\}$;
- *разность* $A \setminus B = A \cap \overline{B}$ содержит элементы, принадлежащие A, но не принадлежащие B. Отметим, что B может содержать элементы, не принадлежащие A. Например, если $A = \{2, 3, 7, 8\}$ и $B = \{3, 5, 8\}$, то $A \setminus B = \{2, 7\}$.

Если любой элемент A принадлежит также S, то A называется *подмножеством* S и обозначается $A \subset S$. Любое множество S имеет $2^{|S|}$ подмножеств, включая пустое множество. Например, подмножествами множества $\{2, 4, 7\}$ являются

$$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\} \text{ и } \{2, 4, 7\}.$$

Часто встречаются множества $\mathbb{N}$ (натуральные числа), $\mathbb{Z}$ (целые числа), $\mathbb{Q}$ (рациональные числа) и $\mathbb{R}$ (вещественные числа). Множество $\mathbb{N}$ определяется одним из двух способов в зависимости от ситуации: $\mathbb{N} = \{0, 1, 2, \ldots\}$ или $\mathbb{N} = \{1, 2, 3, \ldots\}$.

Для определения множеств применяется специальная нотация. Например, множество

$$A = \{2n : n \in \mathrm{Z}\}$$

состоит из всех четных чисел, а множество

$$B = \{x \in \mathbb{R} : x > 2\}$$

– из всех вещественных чисел, больших двух.

Математическая логика

Логическое выражение может принимать одно из двух значений: *true* (1) или *false* (0). Наиболее важны следующие логические операторы: ¬ (*отрицание*), ∧ (*конъюнкция*), ∨ (*дизъюнкция*), ⇒ (*импликация*), ⇔ (*эквиваленция*). В табл. А.1 приведены определения этих операторов.

Таблица А.1. Логические операторы

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Выражение $\neg A$ означает противоположность A. Выражение $A \wedge B$ равно true, если A и B одновременно равны true, а выражение $A \vee B$ равно true, если либо A, либо B, либо A и B одновременно равны true.

Выражение $A \Rightarrow B$ равно true, если всякий раз, как A равно true, B также равно true. Выражение $A \Leftrightarrow B$ равно true, если A и B одновременно равны true или false.

Предикатом называется выражение, принимающее значение true или false в зависимости от параметров. Обычно предикаты обозначаются заглавными буквами. Например, можно определить предикат $P(x)$, равный true тогда и только тогда, когда x – простое число. В соответствии с этим определением $P(7)$ равно true, но $P(8)$ равно false.

Квантор связывает логическое выражение с элементами множества. Наиболее важны кванторы *всеобщности* ∀ (для любого) и *существования* ∃. Например, запись

$$\forall x(\exists y(y < x))$$

означает, что для любого элемента x множества существует элемент y множества – такой, что y меньше x. Это высказывание истинно для множества целых чисел, но ложно для множества натуральных чисел.

Описанная выше нотация позволяет выразить различные виды логических высказываний. Например, запись

$$\forall x((x > 1 \wedge \neg P(x)) \Rightarrow (\exists a(\exists b(a > 1 \wedge b > 1 \wedge x = ab))))$$

означает, что если число x больше 1 и не является простым, то существуют числа a и b, большие 1, произведение которых равно x. Это высказывание истинно для множества целых чисел.

Функции

Функция $\lfloor x \rfloor$ округляет число x до целого с недостатком, а функция $\lceil x \rceil$ округляет x до целого с избытком. Например:

$$\lfloor 3/2 \rfloor = 1 \text{ и } \lceil 3/2 \rceil = 2.$$

Функции $\min(x_1, x_2, \ldots, x_n)$ и $\max(x_1, x_2, \ldots, x_n)$ возвращают соответственно наименьшее и наибольшее значения среди $x_1, x_2, \ldots, x_n$. Например:

$$\min(1, 2, 3) = 1 \text{ и } \max(1, 2, 3) = 3.$$

Факториал $n!$ определяется по формуле

$$\prod_{x=1}^{n} x = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n .$$

Или рекуррентно:

$$0! = 1$$
$$n! = n \cdot (n - 1)!$$

Числа Фибоначчи возникают во многих ситуациях. Их можно определить следующим рекуррентным соотношением:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n - 1) + f(n - 2)$$

Вот первые числа Фибоначчи:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Существует также замкнутая формула для вычисления чисел Фибоначчи, которую иногда называют *формулой Бине*:

$$f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}.$$

Логарифмы

Логарифм числа x обозначается $\log_b(x)$, где b – основание логарифма. По определению, $\log_b(x) = a$ тогда и только тогда, когда $b^a = x$. *Натуральным логарифмом* $\ln(x)$ числа x называется логарифм по основанию $e \approx 2.71828$.

Полезным свойством логарифмов является тот факт, что $\log_b(x)$ говорит о том, сколько раз нужно разделить x на b для достижения 1. Например, $\log_2(32) = 5$, потому что необходимо 5 делений на 2:

$$32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1.$$

Логарифм произведения равен сумме логарифмов:

$$\log_b(xy) = \log_b(x) + \log_b(y)$$

и, следовательно,

$$\log_b(x^n) = n \cdot \log_b(x).$$

Кроме того, логарифм частного равен разности логарифмов:

$$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y).$$

Приведем еще одну полезную формулу:

$$\log_u(x) = \frac{\log_b(x)}{\log_b(u)},$$

позволяющую вычислять логарифмы по любому основанию, если известен способ вычисления их по какому-то фиксированному основанию.

Системы счисления

Обычно числа записываются в десятичной системе счисления, в которой имеются цифры 0, 1, ..., 9. Но существует много других систем счисления, например с основанием 2 (двоичная система), в которой есть всего две цифры: 0 и 1. Вообще, в системе счисления с основанием b в качестве цифр используются целые числа от 0 до $b - 1$.

Для преобразования числа из десятичной системы в систему с основанием b нужно делить это число на b, пока не получится нуль. Последовательность остатков, выписанных в обратном порядке, и является записью числа в системе с основанием b. Например, преобразуем число 17 в систему с основанием 3:

- 17/3 = 5 (остаток 2);
- 5/3 = 1 (остаток 2);
- 1/3 = 0 (остаток 1).

Таким образом, число 17 в системе с основанием 3 равно 122. Обратно, чтобы преобразовать число из системы с основанием b в десятичное, достаточно умножить каждую цифру на b^k, где k – позиция цифры, отсчитываемая справа, начиная с нуля, и сложить результаты. Например, преобразовать 122 из систем с основанием 3 в десятичную систему можно следующим образом:

$$1 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 17.$$

Количество цифр в записи целого числа x в системе с основанием b вычисляется по формуле $\lfloor \log_b(x) + 1 \rfloor$. Например, $\lfloor \log_3(17) + 1 \rfloor = 3$.

Библиография

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network Flows: Theory, Algorithms, and Applications, Pearson, 1993.
2. A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. Information Processing Letters, 9 (5): 216–219, 1979.
3. M. A. Bender and M. Farach-Colton. The LCA problem revisited. Latin American Symposium on Theoretical Informatics, 88–94, 2000.
4. J. Bentley and D.Wood. An optimal worst case algorithm for reporting intersections of rectangles. IEEE Transactions on Computers, C-29 (7): 571–577, 1980.
5. A. Blumer et al., The smallest automation recognizing the subwords of a text. Theoret. Comput. Sci. 40, 31–55 (1985).
6. Codeforces: On «Mo's algorithm», http://codeforces.com/blog/entry/20032.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, MIT Press, 2009 (3rd edition).
8. K. Diks et al. Looking for aChallenge? TheUltimate Problem Set fromthe University of Warsaw Programming Competitions, University of Warsaw, 2012.
9. J. Edmonds, R. Karp, Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM. 19 (2), 248–264 (1972).
10. D. Fanding. A faster algorithm for shortest-path – SPFA. Journal of Southwest Jiaotong University, 2, 1994.
11. P. M. Fenwick. A new data structure for cumulative frequency tables. Software: Practice and Experience, 24 (3): 327–336, 1994.
12. J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. Annual Symposium on Combinatorial Pattern Matching, 36–48, 2006.
13. F. Le Gall. Powers of tensors and fast matrix multiplication. International Symposium on Symbolic and Algebraic Computation, 296–303, 2014.
14. A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. Annual Symposium on Foundations of Computer Science, 621–630, 2014.
15. D. Gusfield. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
16. S. Halim and F. Halim. Competitive Programming 3: The New Lower Bound of Programming Contests, 2013.
17. The InternationalOlympiad in Informatics Syllabus, https://people.ksp.sk/~misof/ioi-syllabus/.
18. D. Johnson, Efficient algorithms for shortest paths in sparse networks. J. ACM 24 (1), 1–13 (1977).

19. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. International Colloquium on Automata, Languages, and Programming, 943–955, 2003.
20. R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. Annual ACM Symposium on Theory of Computing, 125–135, 1972.
21. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. Annual Symposium on Combinatorial Pattern Matching, 181–192, 2001.
22. J. Kleinberg and É. Tardos. Algorithm Design, Pearson, 2005.
23. D. E. Knuth. Optimum binary search trees. Acta Informatica 1 (1): 14–25, 1971.
24. D. E. Knuth, J. H. Morris Jr., V. R. Pratt, Fast pattern matching in strings. SIAM J. Comput. 6 (2), 323–350 (1977).
25. S. Kopeliovich. Offline solution of connectivity and 2-edge-connectivity problems for fully dynamic graphs. MSc thesis, Saint Petersburg State University, 2012.
26. M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. Journal of Algorithms, 5 (3): 422–432, 1984.
27. J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. Olympiads in Informatics, 7 (1): 90–100, 2013.
28. D. Pearson.Apolynomial-time algorithm for the change-making problem. Operations Research Letters, 33 (3): 231–234, 2005.
29. 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. https://github.com/preusser/q27.
30. M. I. Shamos and D. Hoey. Closest-point problems. Annual Symposium on Foundations of Computer Science, 151–162, 1975.
31. S. S. Skiena. The Algorithm Design Manual, Springer, 2008 (2nd edition).
32. S. S. Skiena andM. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
33. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. Journal of Computer and System Sciences, 26 (3): 362–391, 1983.
34. P. Stańczyk. Algorytmika praktyczna w konkursach Informatycznych. MSc thesis, University of Warsaw, 2006.
35. V. Strassen. Gaussian elimination is not optimal. Numerische Mathematik, 13 (4): 354–356, 1969.
36. F. F. Yao. Efficient dynamic programming using quadrangle inequalities. Annual ACM Symposium on Theory of Computing, 429–435, 1980.

Предметный указатель

P

C

Книги изд тельств «ДМК Пресс» можно з к з ть в торгово-изд тельском холдинге «Пл нет Альянс» н ложенным пл тежом, высл в открытку или письмо по почтовому дресу: **115487, г. Москв , 2-й Н г тинский пр-д, д. 6А**.

При оформлении з к з следует ук з ть дрес (полностью), по которому должны быть высл ны книги; ф милию, имя и отчество получ теля. Жел тельно т кже ук з ть свой телефон и электронный дрес.

Эти книги вы можете з к з ть и в интернет-м г зине: **www.alians-kniga.ru**.

Оптовые з купки: тел. +7(499) 782-38-89

Электронный дрес: **books@alians-kniga.ru**.

Антти Л ксонен

Олимпиадное программирование

Изучение и улучшение алгоритмов на соревнованиях

2-е изд ние, обновленное и дополненное

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100$^{1}/_{16}$. Печать цифровая.
Усл. печ. л. 24,38. Тираж 400 экз.

Веб-с йт изд тельств : www.dmkpress.com