БИБЛИОТЕКА ПРОГРАММИСТА

Вильям Спрингер

ГИД ПО COMPUTER SCIENCE

РАСШИРЕННОЕ ИЗДАНИЕ

A Programmer's Guide to Computer Science Volume II

William M. Springer II, PhD

Вильям Спрингер

ГИД ПО COMPUTER SCIENCE

ДЛЯ КАЖДОГО ПРОГРАММИСТА

РАСШИРЕННОЕ ИЗДАНИЕ

Санкт-Петербург • Москва • Минск

2021

Вильям Спрингер

Гид по Computer Science, расширенное издание

Перевел с английского А. Павлов

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректор	*Е. Павлович*
Верстка	*О. Богданович*

ББК 32.973.2-018
УДК 004.3

Спрингер Вильям

С74 Гид по Computer Science, расширенное издание. — СПб.: Питер, 2021. — 304 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1825-0

Колосс на глиняных ногах — так можно назвать программиста без подготовки в области Computer Science. Уверенное владение основами позволяет «не изобретать велосипеды» и закладывать в архитектуру программ эффективные решения. Всё это избавляет от ошибок и чрезмерных затрат на тестирование и рефакторинг. Не беда, если вы чувствуете себя не у дел, когда другие программисты обсуждают аппроксимативный предел. Даже специалисты с опытом допускают ошибки из-за того, что подзабыли Computer Science.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1951204044 англ.
ISBN 978-5-4461-1825-0

© William M. Springer II
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Библиотека программиста», 2021
© Павлов А., перевод с английского языка, 2021

Права на издание получены по соглашению с William Springer. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 07.2021. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214,
тел./факс: 208 80 01.
Подписано в печать 25.06.21. Формат 60×90/16. Бумага офсетная. Усл. п. л. 19,000. Тираж 1000. Заказ 0000.

Оглавление

Часть II. Графы и графовые алгоритмы

Часть III. Неграфовые алгоритмы

Часть VII. Безопасность и конфиденциальность

Часть VIII. Аппаратное и программное обеспечение

Часть IX. Углубленные темы

Приложения

Введение

Зачем нужна эта книга

Многие из моих знакомых разработчиков пришли в профессию из самых разных областей. У одних — высшее образование в области Computer Science; другие изучали фотографию, математику или даже не окончили университет.

В последние годы я заметил, что программисты все чаще стремятся изучить Computer Science по ряду причин:

- чтобы стать хорошими программистами;
- чтобы на собеседованиях отвечать на вопросы про алгоритмы;
- чтобы удовлетворить свое любопытство в области Computer Science или наконец перестать сожалеть о том, что в свое время у них не было возможности освоить этот предмет.

Эта книга для всех вас.

Многие найдут здесь темы, интересные сами по себе. Я попытался показать, в каких реальных (неакадемических) ситуациях эти знания будут полезны. Хочу, чтобы, прочитав эту книгу, вы получили такие же знания, как

после изучения базового курса по Computer Science, а также научились их применять.

Проще говоря, цель этой книги — помочь вам стать более квалифицированным и опытным программистом благодаря лучшему пониманию Computer Science. Мне не под силу втиснуть в одну книгу 20-летний стаж преподавания в колледже и профессиональный опыт... однако я постараюсь сделать максимум того, на что способен. Надеюсь, что вы найдете здесь хотя бы одну тему, о которой сможете сказать: «Да, теперь мне это понятно» — и применить знания в своей работе.

Разослав на рассмотрение черновой вариант книги, я получил множество однотипных отзывов. Слишком много материала. Слишком сложно для восприятия. Слишком устрашающе.

Полученные комментарии способствовали внесению нескольких изменений. Текст был сокращен и упрощен; подробности, не имеющие непосредственного отношения к рассматриваемой теме, опущены. Книга была разбита на части, причем самые важные темы идут в первой половине. В результате она получилась менее устрашающей и более понятной.

Чего вы не найдете в издании

Смысл книги состоит в том, чтобы читатель смог лучше понимать Computer Science и применять знания на практике, а вовсе не в том, чтобы полностью заменить четыре года обучения.

В частности, это не книга с доказательствами. Действительно, начиная с части VI, рассмотрены методы доказательства, однако стандартные алгоритмы обычно приводятся здесь без доказательств. Идея в том, чтобы читатель узнал о существовании этих алгоритмов и научился их использовать, не вникая в подробности. В качестве книги с доказательствами, написанной для студентов и аспирантов, я настоятельно рекомендую *Introduction to Algorithms*[1] («Алгоритмы. Вводный курс») Кормена (Cormen), Лейзерсона (Leiserson), Ривеста (Rivest) и Стейна (Stein) (этих авторов обычно объединяют под аббревиатурой CLRS).

Это и не книга по программированию: вы не найдете здесь рекомендаций, когда использовать числа типа `int`, а когда — `double`, или объяснений, что такое цикл. Предполагается, что читатель сможет разобраться в листингах на псевдокоде, используемых для описания алгоритмов[2]. Цель книги — связать концепции Computer Science с уже знакомыми читателю методами программирования.

Дополнительные ресурсы

Для тех, кто хочет подробнее изучить ту или иную тему, я включил в сноски ссылки на дополнительные

---

[1] *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to Algorithms, 3rd Edition. The MIT Press, 2009.

[2] Все программы в этой книге написаны на псевдокоде на основе языка C.

материалы. Кроме того, по адресу http://www.whatwilliamsaid.com/books/ вы найдете тесты для самопроверки к каждой главе.

Что дальше

Я был намеренно краток и старался опускать детали, которые, будучи интересными сами по себе, не требуются для понимания описываемых концепций. В следующих книгах я собираюсь более подробно остановиться на некоторых областях, представляющих особый интерес.

Чтобы предложить тему для следующей книги, подписаться на рассылку или просто задать вопрос, посетите мой сайт: http://www.whatwilliamsaid.com/books. С нетерпением жду ваших сообщений.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I
Основы Computer Science

1 Асимптотическое время выполнения

1.1. Что такое алгоритм

Предположим, вам нужно научить робота делать бутерброд с арахисовым маслом (рис. 1.1). Ваши инструкции могут быть примерно такими.

1. Открыть верхнюю левую дверцу шкафчика.
2. Взять банку с арахисовым маслом и вынуть ее из шкафчика.
3. Закрыть шкафчик.
4. Держа банку с арахисовым маслом в левой руке, взять крышку в правую руку.
5. Поворачивать правую руку против часовой стрелки, пока крышка не откроется.
6. И так далее...

Это программа: вы описали каждый шаг, который компьютер должен выполнить, и указали всю информацию, которая требуется компьютеру для выполнения каждого шага. А теперь представьте, что вы объясняете

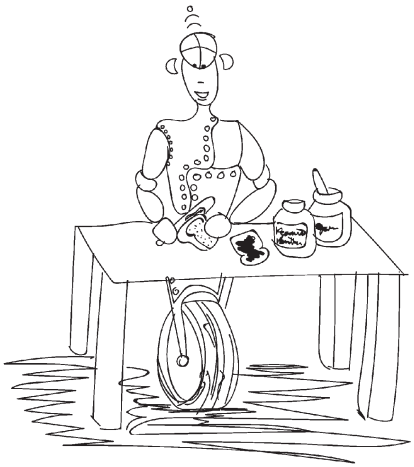


Рис. 1.1. Робот — изготовитель бутербродов

человеку, как сделать бутерброд с арахисовым маслом. Ваши инструкции будут, скорее всего, такими.

1. Достаньте арахисовое масло, джем и хлеб.
2. Намажьте ножом арахисовое масло на один ломтик хлеба.
3. Намажьте ложкой джем на второй ломтик хлеба.
4. Сложите два ломтика вместе. Приятного аппетита!

Это алгоритм: процесс, которому нужно следовать для получения желаемого результата (в данном случае бутерброда с арахисовым маслом и джемом). Обратите внимание, что алгоритм более абстрактный, чем программа. Программа сообщает роботу, откуда именно нужно взять предметы на конкретной кухне, с точным указанием всех необходимых деталей. Это реализация алгоритма, которая предоставляет все важные детали,

но может быть выполнена на любом оборудовании (в данном случае — на кухне), со всеми необходимыми элементами (арахисовое масло, джем, хлеб и столовые приборы).

1.2. Почему скорость имеет значение

Современные компьютеры достаточно быстры, поэтому во многих случаях скорость алгоритма не особенно важна. Когда я нажимаю кнопку и компьютер реагирует за 1/25 секунды, а не за 1/100 секунды, эта разница для меня не имеет значения — с моей точки зрения, компьютер в обоих случаях реагирует мгновенно.

Но во многих приложениях скорость все еще важна, например, при работе с большим количеством объектов. Предположим, что у вас есть список из миллиона элементов, который необходимо отсортировать. Эффективная сортировка занимает одну секунду, а неэффективная может длиться несколько недель. Возможно, пользователь не захочет ждать, пока она закончится.

Мы часто считаем задачу неразрешимой, если не существует известного способа ее решения за разумные сроки, где «разумность» зависит от различных реальных факторов. Например, безопасность шифрования данных часто зависит от сложности разложения на множители (факторизации) больших чисел. Если я отправляю вам зашифрованное сообщение, содержимое которого нужно хранить в секрете в течение недели, то для меня не имеет значения, что злоумышленник перехватит это сообщение и расшифрует его через три года. Задача

не является неразрешимой — просто наш любитель подслушивать не знает, как решить ее достаточно быстро, чтобы решение было полезным.

Важным навыком в программировании является умение оптимизировать только те части программы, которые необходимо оптимизировать. Если пользовательский интерфейс работает на 1/1000 секунды медленнее, чем мог бы, это никого не волнует — в данном случае мы предпочли бы незаметному увеличению скорости удобочитаемую программу. А вот код, расположенный внутри цикла, который может выполняться миллионы раз, должен быть написан максимально эффективно.

1.3. Когда секунды (не) считаются

Рассмотрим алгоритм приготовления бутербродов из раздела 1.1. Поскольку мы хотим, чтобы этот алгоритм можно было использовать для любого количества различных роботов — изготовителей бутербродов, мы не хотим измерять количество секунд, затрачиваемое на выполнение алгоритма, поскольку для разных роботов оно будет различаться. Один робот может дольше открывать шкафчик, но быстрее вскрывать банку с арахисовым маслом, а другой — наоборот.

Вместо того чтобы измерять фактическую скорость выполнения каждого шага, которая будет различаться для разных роботов и кухонь, лучше подсчитать (и минимизировать) количество шагов. Например, алгоритм, который требует, чтобы робот сразу брал и нож и ложку, эффективнее, чем алгоритм, в котором робот открывает

ящик со столовыми приборами, берет нож, закрывает ящик, кладет нож на стол, снова открывает ящик, достает ложку и т. д.

Измерение времени: алгоритм или программа?

Напомню, что алгоритмы являются более обобщенными, чем программы. Для нашего алгоритма приготовления бутербродов мы хотим посчитать число шагов. Если бы мы действительно использовали конкретного робота — изготовителя бутербродов, то нас бы больше интересовало точное время, которое требуется для изготовления каждого бутерброда.

Следует признать, что не все шаги потребуют одинакового времени выполнения; скорее всего, взять нож — быстрее, чем намазать арахисовое масло на хлеб. Поэтому мы хотим не столько узнать точное количество шагов, сколько иметь представление о том, сколько шагов потребуется в зависимости от размера входных данных. В случае с нашим роботом время, необходимое для приготовления бутерброда, при увеличении количества бутербродов не увеличивается (при условии, что нам хватит джема).

Два компьютера могут выполнять алгоритм с разной скоростью. Это зависит от их тактовой частоты, объема доступной памяти, количества тактовых циклов, требуемого для выполнения каждой инструкции, и т. д. Однако обоим компьютерам, как правило, требуется приблизительно одинаковое число инструкций, и мы можем измерить скорость, с которой количество требуемых инструкций увеличивается в зависимости от размера задачи (рис. 1.2). Например, при сортировке

массива чисел, если увеличить его размер в тысячу раз, один алгоритм может потребовать в тысячу раз больше команд, а второй — в миллион раз больше[1].

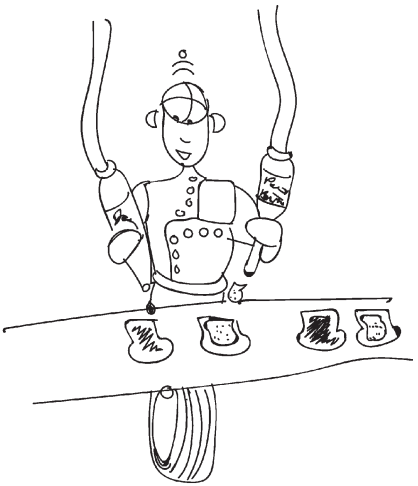


Рис. 1.2. Более эффективный робот — изготовитель бутербродов

Часто мы хотим измерить скорость алгоритма несколькими способами. В жизненно важных ситуациях — например, при запуске двигателей на зонде, который приземляется на Марсе, — мы хотим знать время выполнения при самом неблагоприятном раскладе. Мы можем выбрать алгоритм, который в среднем работает немного медленнее, но зато гарантирует, что его выполнение никогда не займет больше времени, чем мы считаем приемлемым (и наш зонд не разобьется вместо того, чтобы приземлиться). Для более повседневных сценариев мы

[1] Конкретные примеры см. в главе 8.

можем смириться со случайными всплесками времени выполнения, если при этом среднее время не растет; например, в видеоигре мы бы предпочли генерировать результаты в среднем быстрее, смирившись с необходимостью время от времени прерывать длительные вычисления. А бывают случаи, когда мы хотели бы знать наилучшую производительность. Однако в большинстве ситуаций мы просто рассчитываем наихудшее время выполнения, которое все равно часто совпадает со средним временем выполнения.

1.4. Как мы описываем скорость

Предположим, у нас есть два списка целых чисел, которые мы хотим отсортировать: {1, 2, 3, 4, 5, 6, 7, 8} и {3, 5, 4, 1, 2}. Сортировка какого списка займет меньше времени?

Ответ на вопрос: неизвестно. Для одного алгоритма сортировки тот факт, что первый список уже отсортирован, позволит почти сразу завершить работу. Для другого алгоритма решающим фактором может быть то, что второй список короче. Но обратите внимание, что оба свойства входных данных — размер списка и последовательность расположения чисел — влияют на количество шагов, необходимых для сортировки.

Если некоторое свойство, например «отсортированность», изменяет время выполнения конкретного алгоритма только на постоянную величину, мы можем просто его игнорировать, поскольку его влияние незаметно по сравнению с влиянием размера задачи. Например, робот может делать бутерброды с виноградным или

малиновым джемом, но банка с малиновым джемом открывается немного дольше. Если робот делает миллион бутербродов, то влияние выбора джема на время приготовления бутерброда незаметно на фоне количества бутербродов. Поэтому мы можем его игнорировать и просто сказать, что приготовление миллиона бутербродов занимает примерно в миллион раз больше времени, чем приготовление одного бутерброда.

С точки зрения математики мы вычисляем *асимптотическое* время выполнения алгоритма, то есть скорость увеличения времени выполнения в зависимости от размера входных данных. Нашему роботу, который делает бутерброды, требуется некоторое постоянное время c для приготовления одного бутерброда и в n раз больше времени, то есть cn, чтобы сделать n бутербродов. Мы отбрасываем константу и говорим, что наш алгоритм приготовления бутербродов занимает время $O(n)$ (произносится как «O большое от n» или просто «O от n»). Это означает, что время выполнения в худшем случае пропорционально количеству бутербродов, которое будет сделано. Нас интересует не точное количество необходимых шагов, а скорость, с которой это число увеличивается по мере роста размера задачи (в данном случае — количества бутербродов).

1.5. Скорость типичных алгоритмов

Сколько бы бутербродов ни делал наш робот, это не увеличивает время, необходимое для изготовления одного бутерброда. Это *линейный* алгоритм — общее время выполнения пропорционально количеству обрабатываемых элементов. Для большинства задач это лучшее, чего

можно добиться[1]. Типичный пример линейного алгоритма в программировании — чтение списка элементов и выполнение некоторой задачи для каждого элемента списка: время, затрачиваемое на обработку каждого элемента, не зависит от других элементов. Есть цикл, который выполняет постоянный объем работы и осуществляется один раз для каждого из n элементов, поэтому все вместе занимает $O(n)$ времени.

Но чаще количество элементов списка влияет на объем работы, которую необходимо выполнить для отдельного элемента. Алгоритм сортировки может обрабатывать каждый элемент списка, разделяя список на два меньших списка, и повторять это до тех пор, пока все элементы не окажутся в своем собственном списке. На каждой итерации алгоритм выполняет $O(n)$ операций и требует $O(\lg n)$ итераций, что в общей сложности составляет $O(n) \times O(\lg n) = O(n \lg n)$ времени.

Математическое предупреждение

Логарифм описывает, в какую степень необходимо возвести число, указанное в основании, чтобы получить желаемое значение. Например, $\log_{10} 1000 = 3$, так как $10^3 = 1000$.

В компьютерах мы часто делим на 2, поэтому обычно используются логарифмы по основанию 2. Сокращенно $\log_2 n$ обозначается как $\lg n$.[2] Таким образом, $\lg 1 = 0$, $\lg 2 = 1$, $\lg 4 = 2$, $\lg 8 = 3$ и т. д.

---

[1] Поскольку n — это размер входных данных, то в общем случае только для их чтения требуется время $O(n)$.

[2] Строго говоря, lg — это логарифм по основанию 10, но часто именно в Computer Science принимают, что это логарифм по основанию 2. — *Примеч. науч. ред.*

С этого момента время выполнения начинает ухудшаться. Рассмотрим алгоритм, в котором каждый элемент множества сравнивается со всеми остальными элементами множества, — здесь мы выполняем работу, занимающую $O(n)$ времени, $O(n)$ раз, то есть общее время выполнения алгоритма — $O(n^2)$. Это *квадратичное* время.

Все алгоритмы, у которых время выполнения пропорционально количеству входных данных, возведенному в некоторую степень, называются *полиномиальными* алгоритмами; такие алгоритмы принято считать быстрыми. Конечно, алгоритм, решение которого пропорционально количеству входных данных в сотой степени, хоть и является полиномиальным, все же не будет считаться быстрым даже при большом воображении! Однако на практике задачи, о которых известно, что они имеют полиномиальное время решения, как правило, решаются за *биквадратное* (четвертой степени) время или еще быстрее.

Это не означает, что асимптотически более эффективный алгоритм всегда будет работать быстрее, чем асимптотически менее эффективный. Например, ваш жесткий диск вышел из строя и вы потеряли несколько важных файлов. К счастью, вы сделали их резервную копию в Сети[1].

Вы можете загрузить файлы из облака со скоростью 10 Мбит/с (при условии, что у вас хорошее соединение).

---

[1] В этом примере я использовал цифры для резервного копирования из облака Carbonite, но есть и много других. Это ни в коем случае не реклама, а всего лишь первый попавшийся сервис, который я обнаружил при поиске.

Это линейное время — количество времени, которое требуется для извлечения всех потерянных файлов, и оно более или менее прямо пропорционально зависит от размера файлов. Служба резервного копирования также предлагает загрузить файлы на внешний диск и отправить их вам — это действие занимает постоянное время, или $O(1)$, потому что время получения этих данных не зависит от их размера[1]. Если речь идет о восстановлении всего нескольких мегабайтов, то загрузить их напрямую будет намного быстрее. Но, как только файл достигнет такого размера, что его загрузка будет занимать столько же времени, сколько и отправка, удобнее будет использовать внешний диск.

Если полиномиальные алгоритмы быстрые, то какие же алгоритмы медленные? Для некоторых алгоритмов (называемых *экспоненциальными*) число операций ограничено не размером входных данных, возведенным в некоторую постоянную степень, а константой, возведенной в степень, равную размеру входных данных.

В качестве примера рассмотрим попытку угадать числовой код доступа длиной n символов. Если это десять

---

[1] Ну хорошо, почти не зависит — это занимает 1–3 рабочих дня. «Не зависит» в данном случае означает не то, что некая операция всегда занимает одинаковое время, а лишь то, что время выполнения не зависит от количества входных данных. Мы также предполагаем, что компания, занимающаяся резервным копированием, может подготовить вашу копию достаточно быстро, так что они не пропустят отправку почты из-за очень большого размера файла.

цифр от 0 до 9, то количество возможных кодов составляет 10^n. Обратите внимание, что это число растет **намного** быстрее, чем n^{10}: если n равно всего 20, полиномиальный алгоритм работает уже почти в 10 миллионов раз быстрее (рис. 1.3)!

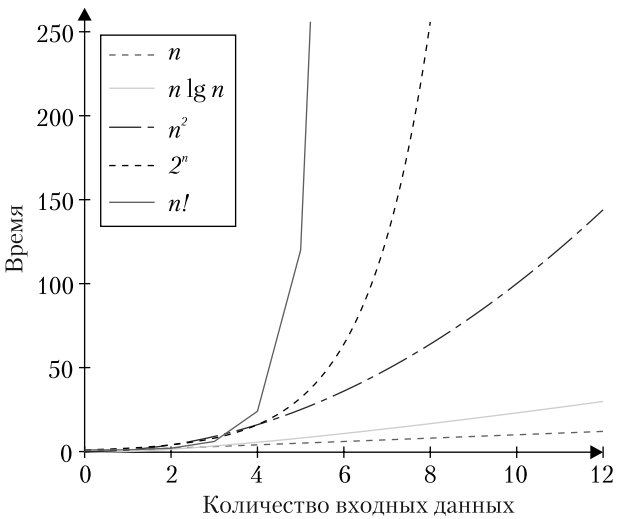


Рис. 1.3. Даже при небольшом количестве входных данных различия в асимптотическом времени выполнения быстро становятся заметными

1.6. Всегда ли полиномиальное время лучше?

Как правило, специалисты по Computer Science заинтересованы получить полиномиальное решение задачи, особенно если оно выполняется за квадратичное время или еще быстрее. Однако для задач разумного

(небольшого) размера экспоненциальные алгоритмы также могут быть приемлемы.

Зачастую мы можем найти приближенное решение задачи за полиномиальное время, однако получить точный (или близкий к точному) ответ можем лишь за экспоненциальное время. Рассмотрим в качестве примера задачу коммивояжера: продавец хотел бы посетить каждый город на своем маршруте ровно один раз и вернуться домой, преодолев минимальное расстояние. (Представьте себе, сколько денег сэкономили бы службы доставки UPS и FedEx, если бы сделали свои маршруты всего лишь немного более эффективными!)

Чтобы получить точный ответ, нужно вычислить все возможные маршруты и сравнить их суммарные расстояния, то есть $O(n!)$ возможных путей. Очень близкое к оптимальному (в пределах до 1 %) решение может быть найдено за экспоненциальное время[1]. Но возможное «достаточно хорошее» (в пределах 50 % от оптимального) решение может быть найдено за полиномиальное время[2]. Это обычный компромисс: мы можем быстро получить достаточно хорошее приближение или медленнее — более точный ответ.

---

[1] Подробный обзор различных подходов вы найдете в статье: *Applegate D. L., Bixby R. E., Chvátal V., Cook W. J.* The Traveling Salesman Problem.

[2] На момент написания этой книги были известны алгоритмы, позволяющие найти решение хуже оптимального не более чем на 50 % за время $O(n^3)$. См., например, статью: *Sebö A., Vygen J.* Shorter Tours by Nicer Ears, 2012.

1.7. Время выполнения алгоритма

Рассмотрим следующий код:

```
foreach (name in NameCollection)
{
    Print "Hello, {name}!";
}
```

Здесь у нас есть коллекция из n строк; для каждой строки выводится короткое сообщение. Вывод сообщения занимает постоянное время[1], то есть $O(1)$. Мы делаем это n раз, то есть $O(n)$. Умножив одно на другое, получим результат: время выполнения кода составляет $O(n)$.

А теперь рассмотрим другую функцию:

```
DoStuff (numbers)
{
    sum = 0;
    foreach (num in numbers)
    {
        sum += num;
    }
    product = 1;
    foreach (num in numbers)
    {
        product *= num;
    }
    Print "The sum is {sum} and the
           product is {product}";
}
```

[1] Это не означает, что для каждой строки требуется одинаковое время; время, необходимое для вывода каждой строки, не зависит от количества строк.

Здесь есть два цикла; каждый из них состоит из $O(n)$ итераций и на каждой итерации выполняет постоянную работу, то есть общее время выполнения каждого цикла составляет $O(n)$. Сложив $O(n)$ и $O(n)$ и отбросив константу, мы снова получим не $O(2n)$, а $O(n)$. Представленную выше функцию также можно написать так:

```
DoStuff (numbers)
{
    sum = 0 , product = 1;
    foreach (num in numbers)
    {
        sum += num;
        product *= num;
    }
    Print "The sum is {sum} and the
           product is {product}";
}
```

Теперь у нас есть только один цикл, который снова выполняет постоянную работу; просто у него константа больше, чем раньше. Помните, что нас интересует, насколько быстро растет время выполнения задачи при увеличении ее размера, а не точное число операций для данного размера задачи.

Теперь попробуем что-нибудь более сложное. Каково время выполнения этого алгоритма?

```
CountInventory (stuffToSell, colorList)
{
    totalItems = 0;
    foreach (thing in stuffToSell)
    {
        foreach (color in colorList)
```

```
        {
            totalItems += thing[color];
        }
    }
}
```

Здесь у нас есть два цикла, причем один вложен в другой. Поэтому мы будем умножать их время выполнения, а не складывать. Внешний цикл запускается один раз для каждого элемента каталога, а внутренний — один раз для каждого предоставленного цвета. Для n элементов и m цветов общее время выполнения равно $O(nm)$. Обратите внимание, что это **не** $O(n^2)$; у нас нет оснований полагать, что между n и m существует взаимосвязь.

Рассмотрим еще одну функцию:

```
doesStartWith47 (numbers)
{
    return (numbers[0] == 47);
}
```

Эта функция проверяет, равен ли 47 первый элемент целочисленного массива, и возвращает результат. Объем работы, который выполняет функция, не зависит от количества входных данных и поэтому равен $O(1)$[1].

Мы часто пишем программы, которые включают в себя двоичный поиск, следовательно, в нашем анализе будут логарифмы.

---

[1] Здесь, конечно, предполагается, что массив передается по ссылке; если массив передается по значению, то время выполнения составит $O(n)$.

Например, рассмотрим следующий код[1]:

```
binarySearch (numarray, left, right, x)
{
    if (left > right) { return -1; }
    int mid = 1 + (right - 1)/2;
    if (numarray[mid] == x) { return mid; }
    if (numarray[mid] > x)
        { return binarySearch (numarray, left,
          mid -1, x); }
    return binarySearch (numarray, mid + 1, right, x);
}
```

Исходя из предположения, что массив отсортирован, мы проверяем, является ли средний элемент массива тем, что мы ищем. Если это так, то возвращаем индекс среднего элемента. Если же нет и средний элемент больше требуемого значения, то мы проделываем то же самое с первой половиной массива, а если больше — то со второй половиной. Для каждого рекурсивного вызова выполняется постоянный объем работы (проверка, что значение `left` не больше, чем `right`; это подразумевает, что мы выполнили поиск по всему массиву и искомое значение не было найдено; затем вычисление средней точки и сравнение ее значения с тем, что мы ищем). Мы выполняем $O(\lg n)$ вызовов, каждый из которых занимает $O(1)$ времени, поэтому общая сложность двоичного поиска составляет $O(\lg n)$.

---

[1] Как показывает практика, лучше перенести проверку left > right в конец, поскольку это менее распространенный случай; мы поступили так, чтобы обойтись без вложенности.

Углубленные темы

Представьте, что у нас есть рекурсивная функция, которая разбивает задачу на две части, размер каждой из них составляет 2/3 от размера оригинала? Формула для вычисления такой рекурсии действительно существует; подробнее о ней и об основной теореме (Master Theorem) вы узнаете в главе 31.

1.8. Насколько сложна задача?

Если есть алгоритм для решения задачи, определить время выполнения этого алгоритма обычно довольно просто. Но что, если у нас еще нет алгоритма, но уже нужно понять, насколько сложно будет решить задачу?

Мы можем это сделать, сравнивая задачу с другими подобными задачами, для которых известно время выполнения. Мы можем разделить задачи на классы — наборы задач, имеющих сходные характеристики. Нас интересуют два основных класса: задачи, которые решаются за полиномиальное время, и задачи, решение которых можно проверить за полиномиальное время. Оба этих класса мы рассмотрим в следующей главе.

2 Структуры данных

2.1. Организация данных

Одно из главных понятий Computer Science — структуры данных. Говоря о времени выполнения алгоритмов, мы предполагаем, что данные хранятся в соответствующей структуре, которая позволяет эффективно их обрабатывать. Какая структура лучше, зависит от типа данных и от того, какой доступ к ним нужен.

- Необходим ли произвольный доступ, или достаточно последовательного?
- Будут ли данные при записи всегда добавляться в конец списка, или нужна возможность вставлять значения в середину?
- Допускаются ли повторяющиеся значения?
- Что важнее: наименьшее возможное время доступа или строгая верхняя граница времени выполнения каждой операции?

Ответы на все эти вопросы определяют то, как должны храниться данные.

2.2. Массивы, очереди и другие способы построиться

Возможно, самая известная структура данных — это массив, набор элементов, проиндексированных ключом. Элементы массива хранятся последовательно, причем ключ имеет форму смещения относительно начальной позиции в памяти, благодаря чему можно вычислить положение любого элемента по его ключу. Именно поэтому индексы массива[1] обычно начинаются с нуля; первый элемент массива находится на нулевом расстоянии от начала, следующий — на расстоянии одного элемента от начала и т. д. «На расстоянии одного элемента» может означать один байт, одно слово и т. д., в зависимости от размера данных. Важно, что каждый элемент массива занимает одинаковое количество памяти.

Польза массивов состоит в том, что получение или сохранение любого элемента массива занимает постоянное время, а весь массив занимает $O(n)$ места в памяти[2]. Если количество элементов заранее известно, то память не расходуется зря; поскольку позиция каждого элемента вычисляется просто по смещению относительно начала, нам не нужно выделять место для указателей. Поскольку элементы массива расположены в смежных областях памяти, перебор значений массива, очевидно,

---

[1] В английском языке есть два варианта множественного числа от слова index (индекс): array indices для индексов массива, но database indexes — для индексов базы данных.

[2] Помните: если не указано иное, буквой n обозначается количество элементов.

выполняется гораздо быстрее, чем для многих других структур данных из-за меньшего количества неудачных обращений к кэш-памяти[1].

Однако требование выделения непрерывного блока памяти может сделать массивы плохим выбором, когда число элементов заранее не известно. С ростом размера массива может понадобиться скопировать его в другое место памяти (при условии, что оно есть). Избежать этой проблемы, предварительно выделив гораздо больше места, чем необходимо, бывает довольно затратно[2]. Другая проблема — вставка и удаление элементов массива занимает много времени ($O(n)$), поскольку приходится перемещать все элементы массива.

На практике массивы используются как сами по себе, так и для реализации многих других структур данных, которые накладывают дополнительные ограничения на манипулирование данными. Например, строка может быть реализована в виде массива символов. Очередь — это последовательный список, в котором элементы добавляются только в один конец списка (постановка в очередь), а удаляются из другого (извлечение из очереди); таким образом, очередь может быть реализована

---

[1] Это так называемая локальность ссылок: как только мы получили доступ к элементу, вполне вероятно, что в ближайшем будущем также понадобится доступ к другим элементам, находящимся поблизости. Поскольку мы уже загрузили страницу, содержащую первый элемент, в кэш, то получение ближайших элементов (которые, вероятно, находятся на той же странице) происходит быстрее.

[2] Это особенно актуально при работе со встроенными системами, которые, как правило, имеют ограниченный объем памяти.

как массив, в котором «начало» перемещается вместе с началом очереди при условии, что максимальное количество элементов в очереди никогда не превышает размер массива. Однако очередь неопределенной длины лучше реализовать на основе двусвязного списка (см. раздел 2.3). К другим структурам, реализуемым на основе массивов, относятся списки, стеки и кучи (см. раздел 2.4), очереди с приоритетом (которые часто создаются на основе куч) и хеш-таблицы (см. раздел 2.5).

2.3. Связные списки

Связный список — это структура данных, в которой каждый элемент содержит данные и указатель на следующий элемент списка (а если это двусвязный список, то также ссылку на предыдущий элемент). Указатель на связный список — это просто указатель на первый элемент, или head, списка; поскольку элементы могут размещаться в разных местах выделенной памяти, для поиска указанного элемента необходимо начать с первого элемента и пройтись по всему списку.

Как уже говорилось, многие структуры данных реализованы на основе массивов или связных списков. Во многом связный список является дополнением массива. Если сильная сторона массива — быстрый доступ к любому элементу (по его ключу), то для того, чтобы найти элемент списка, необходимо пройти по всем ссылкам, пока не будет найден нужный элемент, что в худшем случае займет $O(n)$ времени. С другой стороны, массив имеет фиксированный размер, а элементы связного списка могут размещаться в любом месте

памяти, и список может произвольно увеличиваться до тех пор, пока не будет исчерпана доступная память. Кроме того, вставка и удаление элементов массива очень затратны, а в связном списке эти операции выполняются за постоянное время, если есть указатель на предыдущий узел (рис. 2.1).

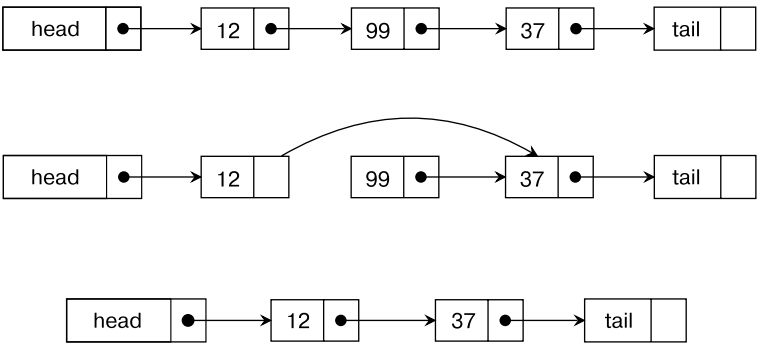


Рис. 2.1. Удаление узла из связного списка

Практическое применение

Представьте себе поезд как пример двусвязного списка: каждый вагон связан с предыдущим и (если он существует) со следующим. В конец поезда можно легко добавить вагоны, но можно вставить вагон и в середину поезда, отсоединив и (пере)присоединив существующие вагоны перед и после добавляемых, или же можно отсоединить вагоны в середине поезда, откатить их на боковой путь и присоединить оставшиеся вагоны. А вот извлечь вагон напрямую не получится; для этого нужно сначала пройти по всему поезду и отделить нужный вагон от предыдущего.

Теоретическая глупость

Как-то мой преподаватель спросил у группы студентов, как определить, содержит ли связный список цикл. Есть несколько классических решений этой задачи. Один из способов — обратить указатели на элементы списка, когда мы их проходим; если цикл существует, то мы в итоге вернемся к началу списка. Другой способ: обходить список с двумя указателями, так что один указатель ссылается на следующий элемент, а второй — через один элемент. Если в списке существует цикл, то в итоге оба указателя когда-нибудь будут ссылаться на один и тот же элемент.

Однако преподаватель добавил условие: он сказал, что ему все равно, сколько времени займет выполнение нашего метода. Я предложил несколько глупый вариант: вычислить объем памяти, необходимый для хранения одного узла списка, и разделить на него общую память системы. Потом вычислить количество посещенных элементов. Если число посещенных элементов превышает количество узлов, которые могут быть сохранены в памяти, список должен содержать цикл.

Преимущество этого решения неоднозначно: оно требует безумно много времени (определяемого размером памяти, а не списка) и является абсолютно правильным.

2.4. Стеки и кучи

2.4.1. Стеки

Стек — это структура данных типа LIFO (Last In, First Out — «последним пришел, первым ушел»), в которой элементы добавляются или удаляются только сверху; это называется «поместить элемент в стек» (push) или «извлечь его из стека» (pop) (рис. 2.2).

Стек можно реализовать на основе массива (отслеживая текущую длину стека) или на основе односвязного списка (отслеживая head списка[1]). Как и в случае с очередями, реализация на основе массива проще, но она накладывает ограничение на размер стека. Стек, реализованный на основе связного списка, может расти до тех пор, пока хватает памяти.

Рис. 2.2. Элементы всегда помещаются в вершину стека

Стеки поддерживают четыре основные операции, каждая из которых может быть выполнена за время $O(1)$: `push` (поместить элемент в стек), `pop` (извлечь элемент из стека), `isEmpty` (проверить, пуст ли стек) и `size` (получить количество элементов в стеке). Часто также реализуют операцию `peek` (просмотреть верхний элемент стека, но

[1] Вместо того чтобы добавлять новые элементы в конец списка, их можно просто помещать в начало. Тогда извлекаемый элемент всегда будет первым в списке.

не удалять его), что эквивалентно извлечению верхнего элемента и его повторному помещению в стек.

При помещении элемента в стек может возникнуть исключение, если размер стека ограничен, и в настоящее время стек заполнен (ошибка переполнения), а при извлечении элемента из стека возникает исключение, если в данный момент стек пуст (ошибка обнуления).

Несмотря на то что в стеках не допускается произвольный доступ, они очень полезны в тех компьютерных вычислениях, для которых требуется ведение истории, от операций Undo до рекурсивных вызовов функций. В этом случае стек обеспечивает возможность обратного перебора, если необходимо вернуться к предыдущему состоянию. В разделе 13.3 вы увидите, как стеки используются для реализации магазинных автоматов, способных распознавать контекстно свободные языки.

Типичным примером использования стека является проверка сбалансированности фигурных скобок. Рассмотрим язык, в котором скобки должны быть парными: каждой правой скобке (`}`) предшествует соответствующая левая скобка (`{`). Мы можем прочитать строку и каждый раз, когда встречается левая скобка, помещать ее в стек. Всегда, когда встречается правая скобка, мы будем извлекать левую скобку из стека. Если попытаться извлечь фигурную скобку из стека, но стек окажется пустым, это будет означать, что у правой фигурной скобки нет соответствующей левой скобки. Если в конце строки стек окажется непустым, это будет означать, что считано больше левых скобок, чем правых. В противном случае все скобки в строке окажутся парными.

2.4.2. Кучи

Кучи, подобно стекам, как правило, реализуются на основе массивов. Как и в стеке, в куче можно удалить за один раз только один элемент. Однако это будет не последний добавленный, а наибольший (для max-кучи) или наименьший элемент (для min-кучи). Куча частично упорядочена по ключам элементов, так что элемент с наивысшим (или низшим) приоритетом всегда хранится в корне кучи (рис. 2.3).

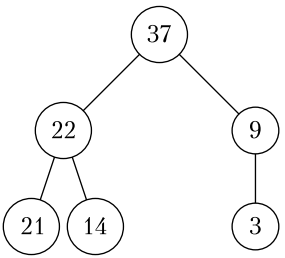


Рис. 2.3. Узлы-братья (узлы с одним и тем же родителем) в куче никак не взаимосвязаны; просто каждый узел имеет более низкий приоритет, чем его родитель

Словом *«куча»* называют структуру данных, которая удовлетворяет свойству упорядоченности кучи: неубывания (min-куча) (значение каждого узла не меньше, чем у его родителя) либо невозрастания (max-куча) (значение каждого узла не больше, чем у родителя). Если не указано иное, то, говоря о куче, мы имеем в виду двоичную кучу, которая является полным двоичным деревом[1], которое удовлетворяет свойству упорядоченности кучи;

[1] Двоичное дерево, в котором каждый уровень, за исключением, возможно, нижнего, имеет максимальное количество узлов. Подробнее о деревьях читайте в главе 5.

в число других полезных видов кучи входят левосторонние кучи, биномиальные кучи и кучи Фибоначчи.

Max-куча поддерживает операции поиска максимального значения (find-max) (`peek`), вставки (insert) (`push`), извлечения максимального значения (extract-max) (`pop`) и увеличения ключа (increase-key) (изменения ключа узла с последующим перемещением узла на новое место на графе). Для двоичной кучи после создания кучи из списка элементов за время $O(n)$ каждая из этих операций занимает время $O(\lg n)$[1].

Кучи используются в тех случаях, когда нужен быстрый доступ к наибольшему (или наименьшему) элементу списка без необходимости сортировки остальной части списка. Именно поэтому кучи используются для реализации очередей с приоритетом: нас интересует не относительный порядок всех элементов, а только то, какой элемент является следующим в очереди, — это всегда текущий корень кучи. Более подробно кучи рассмотрены в разделах 5.3 и 8.3.

2.5. Хеш-таблицы

Предположим, мы хотим определить, содержится ли в массиве некий элемент. Если массив отсортирован, можно выполнить двоичный поиск и найти этот элемент за время $O(\lg n)$; если же нет, можно перебрать весь массив за время $O(n)$. Конечно, если бы мы знали, где находится этот элемент, то мы бы просто обратились туда напрямую за время $O(1)$.

---

[1] Подробнее о временной сложности каждой операции читайте в книге Introduction to Algorithms.

В некоторых случаях, таких как сортировка подсчетом (см. подраздел 8.4.1), в качестве индекса массива используется сам сохраняемый элемент или его ключ, и поэтому можно просто перейти в нужное место без поиска. А что, если бы для произвольного объекта у нас была функция, которая бы принимала ключ этого объекта и преобразовывала его в индекс массива, так что мы бы точно знали, где находится объект? Именно так работают хеш-таблицы.

Первая часть хеш-таблицы — это хеш-функция; она преобразует ключ элемента, который помещается в хеш. Этому ключу соответствует определенное место в таблице. Например, наши ключи представляют собой набор строк, а хеш-функция ставит в соответствие каждой строке количество символов этой строки[1]. Тогда слово *cat* попадет в ячейку 3, а *penguin* — в ячейку 7. Если место ограничено, то мы делим хеш-код по модулю на размер массива; тогда, если массив ограничен десятью ячейками (0–9), строка *sesquipedalophobia* (хеш-код которой равен 18) попадет в ячейку 7[2, 3].

Что произойдет, если мы уже поместили в хеш-таблицу слово *cat*, но попытаемся вставить туда слово *cat* еще раз? В хеш-таблицах допускается хранить только один

---

1 Это не очень хорошая хеш-функция.

2 Если бы мы хешировали слова sesquipedalophobia и hippopotomonstrosesquippedaliophobia по модулю 9, то оба слова попали бы в точку ноль и образовалась бы коллизия, к удовольствию любителей длинных слов и к ужасу страдающих логофобией.

3 На практике мы будем использовать массив, размер которого является простым числом.

экземпляр каждого элемента; в некоторых реализациях в каждой ячейке хранится счетчик, который увеличивается по мере необходимости, чтобы отслеживать количество копий элемента. Но что, если мы попытаемся вставить слово *dog*, которое попадет в ту же ячейку? Есть два способа решить эту проблему. Можно рассматривать каждую ячейку как группу объектов, представленных в виде связного списка (который нужно пройти, чтобы найти правильное животное); это называется методом цепочек. Или же можно просмотреть близлежащие ячейки, найти свободную и поместить *dog* туда — это называется открытой адресацией. Размер хеш-таблицы с цепочками не ограничен, однако производительность будет снижаться по мере увеличения количества элементов в ячейке. При открытой адресации таблица имеет фиксированный максимальный размер; как только все ячейки будут заполнены, в таблицу нельзя будет вставить новые элементы.

Способ организации хеш-таблицы зависит от того, предпочитаем ли мы свести к минимуму коллизии (несколько значений, которые попали в одну ячейку) или объем памяти. Чем больше места выделено под таблицу относительно количества вставляемых элементов, тем меньше вероятность коллизий. За счет увеличения памяти мы получаем повышение скорости: после того как ключ захеширован, сохранение или извлечение элемента (при условии отсутствия коллизий) занимает $O(1)$ времени. Однако при коллизиях наихудшее время извлечения в хеш-таблице (когда для каждого элемента возникает коллизия) составляет $O(n)$.

Практическое применение

В C# есть класс `Hashtable`, позволяющий хранить произвольные объекты. Каждый объект, добавляемый в `Hashtable`, должен реализовывать функцию `GetHashCode()`, она возвращает значение `int32`, которое можно использовать для хеширования объекта. `Dictionary<TKey, TValue>` предоставляет тот же функционал, но только для объектов типа `TValue`, который (при условии, что `TValue` не установлен в `Object`) позволяет программисту избегать упаковки и распаковки сохраненных элементов. Для внутреннего представления в обоих случаях используется структура данных хеш-таблицы, но с разными методами предотвращения коллизий. В `Hashtable` применяется перехеширование (поиск другой ячейки, если первая занята), а в `Dictionary` — метод цепочек (несколько элементов с одинаковым хешем просто добавляются в одну ячейку).

Как правило, хеш-таблицы используют в тех случаях, когда нужен прямой доступ к неотсортированным данным на основе ключа и при этом существует быстродействующая функция генерации ключа для каждого объекта (при условии, что сами объекты не являются такими ключами). Мы не будем использовать хеш-таблицы, когда нужно сохранять порядок сортировки, или элементы не распределены (то есть много элементов хешируется в малое количество ячеек), или часто необходим доступ к блокам последовательно размещенных элементов. Поскольку элементы (как мы надеемся) равномерно распределены по памяти, выделенной для хеш-таблицы, мы теряем преимущество от локальности ссылки.

2.6. Множества и частично упорядоченные множества

Множество — это неупорядоченная коллекция уникальных предметов. Существует три основные операции над множествами, каждая из которых принимает два множества в качестве аргументов и возвращает еще одно множество в качестве результата.

Union(A, B) — объединение множеств — это множество, содержащее каждый элемент, который принадлежит хотя бы одному из множеств A и B. Обычно объединение обозначается как $A \cup B$ (рис. 2.4).

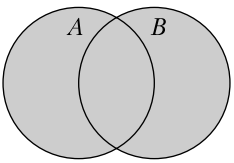


Рис. 2.4. Объединение множеств A и B

Intersection(A, B) — пересечение множеств — это множество элементов, содержащихся как в A, так и в B, обозначается как $A \cap B$ (рис. 2.5).

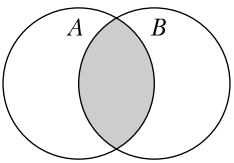


Рис. 2.5. Пересечение множеств A и B

Difference(A, B) — разность множеств (рис. 2.6) — обозначается как $A - B$ — это множество, в которое входят все элементы, содержащиеся в A, но отсутствующие в B.

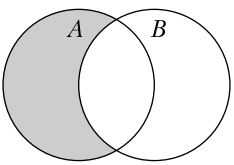


Рис. 2.6. Разность множеств A и B ($A - B$)

Наконец, есть еще одна операция, которая возвращает логическое значение: подмножество

Subset(A, B). Ее результатом будет истина (true), если A является подмножеством B (то есть каждый элемент множества A — элемент множества B). Если A является подмножеством B и не равно B, то такое подмножество называется собственным. Мы используем обозначения $A \subseteq B$ (A является подмножеством B), $A \nsubseteq B$ (A не является подмножеством B), $A \subset B$ (A является собственным подмножеством B) и $A \not\subset B$ (A не является собственным подмножеством B). Если $A \subseteq B$ и $B \subseteq A$, то $A = B$. Существуют также дополнительные операции, которые применяются к множествам строк; они рассмотрены в подразделе 13.2.2.

Существует несколько видов множеств (рис. 2.7). Мультимножество — это множество, в котором допускается дублирование элементов, то есть хранятся несколько копий одного и того же значения либо просто ведется подсчет того, сколько раз данное значение присутствует в множестве.

$$\{4, 7, 2, 16, 1004\}$$

Рис. 2.7. Множество целых чисел

Частично упорядоченное множество — множество, в котором некоторые элементы расположены в определенном порядке. Это означает, что для некоторой двоичной операции[1] $\leq$ между элементами b и c может быть истиной $b \leq c$, в других — $c \leq b$ или же между b и c может не существовать отношений.

---

[1] Двоичная операция — это такая операция, которая работает с двумя операндами. Например, операция «плюс» используется для получения суммы двух значений.

Если все элементы множества связаны операцией ≤, то есть для любых двух элементов f и g множества либо $f \leq g$, либо $g \leq f$, то операция ≤ определяет полный порядок множества. Например, стандартная операция «меньше или равно» — это порядок для множества действительных чисел: любые два числа либо равны, либо одно больше другого.

Необходимо, чтобы отношение было рефлексивным (каждый элемент множества меньше или равен самому себе), антисимметричным (если b меньше или равно c, то c не может быть меньше или равно b, если только b не равно c) и транзитивным (если b меньше или равно c, а c меньше или равно d, то b меньше или равно d).

Практический пример

Пусть ≤ означает отношение «является потомком», причем по определению каждый человек является потомком самого себя. Тогда такое отношение является антисимметричным (если я твой потомок, то ты не можешь быть моим потомком) и транзитивным (если я твой потомок, а ты потомок дедушки, то я тоже потомок дедушки). Это частичное, а не полное упорядочение, поскольку, возможно, ни я не являюсь твоим потомком, ни ты — моим.

Обратите внимание, что куча — частично упорядоченное мультимножество (рис. 2.8): в ней может существовать несколько копий одного и того же значения и каждое значение имеет определенную связь только со своим родителем и потомками.

$$\{3, 3, 6, 3, 9\}$$

Рис. 2.8. Мультимножество целых чисел

Практическое применение

Реляционные базы данных[1] содержат таблицы, в которых хранятся наборы строк. Упорядоченность может быть задана при выводе данных (в SQL это делается с помощью оператора ORDER BY), но такое ограничение не накладывается на фактически сохраненные данные, поэтому нельзя считать, что строки в базе данных хранятся в каком-либо определенном порядке.

2.7. Специализированные структуры данных

Далее в книге мы рассмотрим более специализированные структуры данных. В главе 5 описаны некоторые распространенные графовые структуры данных. В главе 32 представлена концепция амортизированного времени выполнения (общего времени, затрачиваемого на серию операций, а не времени выполнения отдельной операции), а в главе 33 описана структура данных, которую мы проанализируем с использованием амортизированного времени выполнения.

[1] Реляционная база данных структурирована в соответствии с отношениями между хранимыми элементами. Реляционные и иерархические базы данных рассмотрены в главе 30.

3 Классы задач

Специалисты в Computer Science классифицируют задачи по времени, которое занимает их решение, в зависимости от количества входных данных. Благодаря такой классификации задач мы определяем сложность их решения. На практике это позволяет не тратить времени на задачи, которые не решаются достаточно быстро, чтобы ответ был полезным.

Самые простые задачи, класса *P*, могут быть решены за полиномиальное время. Это все задачи, у которых время решения — количество входных данных, возведенное в некоторую постоянную степень. Принято считать, что такие задачи имеют эффективные решения. К этому классу относятся многие широко известные задачи, например такие, как сортировка списков. Задачи класса *P* также называют легкоразрешимыми. Как правило, если можно доказать, что задача относится к классу *P*, значит, ее можно решить за разумное время.

Класс *P* является собственным подмножеством множества задач *EXP*, которые решаются за экспоненциальное время; любая задача, которая может быть решена за время $O(n^2)$, также может быть решена за время $O(2^n)$.

Математическое предупреждение: возвращаясь к главе 2

Для двух множеств A и B множество A является подмножеством B, а B — надмножеством A, если каждый элемент множества A также является элементом множества B. Это обозначается так: $A \subseteq B$ и $B \supseteq A$.

Если множество B содержит все элементы множества A, а также что-то еще, то B является собственным надмножеством множества A, а A — собственным подмножеством B. Это обозначается так: $A \subset B$ и $B \supset A$.

Например, множество {1, 2, 3} является собственным подмножеством множества {1, 2, 3, 4, 5}.

Но в множество *EXP* входят и другие классы, кроме *P*. Один из них — это недетерминированный полином (Nondeterministic Polynomial, NP). Задача относится к классу *NP*, если она может быть решена недетерминированно[1] за полиномиальное время. Другими словами, существует алгоритм, который решает эту задачу путем принятия ряда решений, причем в каждой точке принятия решения алгоритм случайным образом (и удачно) делает правильный выбор. Пока остается ряд шагов,

[1] Детерминированный алгоритм с фиксированными входными данными каждый раз проходит через одну и ту же последовательность состояний и возвращает один и тот же результат. С математической точки зрения такой алгоритм отображает область экземпляров задачи на ряд решений. Недетерминированный алгоритм может демонстрировать различное поведение для одного и того же набора входных данных.

ведущих к ответу, алгоритм выбирает правильные шаги. Другой (более полезный) способ показать, что задача относится к классу *NP*, — убедиться, что ее решение можно проверить (детерминированно, то есть следуя предварительно определенной последовательности шагов) за полиномиальное время. *NP* является надмножеством *P*; любое решение, которое может быть найдено за полиномиальное время, также может быть проверено за полиномиальное время.

Один из важнейших вопросов Computer Science, на который до сих пор не получен ответ: является ли *NP* собственным надмножеством *P*? Существуют ли задачи, которые относятся к *NP*, но не принадлежат *P*, или это одно и то же множество задач? Другими словами: любая ли задача, решение которой быстро проверяется компьютером, также быстро решается компьютером? Большинство специалистов в области Computer Science считают, что $P \neq NP$, но никаких математических доказательств найдено не было[1] (рис. 3.1).

Рассмотрим задачу разбиения множества чисел (рис. 3.2). Для заданного мультимножества (множества, в котором могут присутствовать повторяющиеся элементы) натуральных чисел нужно определить, можно ли разбить такое множество на два подмножества таким образом, чтобы сумма чисел в первом множестве равнялась сумме чисел во втором. Если мультимножество разделено на два подмножества $\{S_1, S_2\}$, то задача решается тривиально: нужно

[1] Вопрос о том, справедливо ли утверждение $P = NP$, является одной из семи задач тысячелетия — важных математических вопросов, ответы на которые не найдены. За решение каждого из них установлен приз 1 миллион долларов.

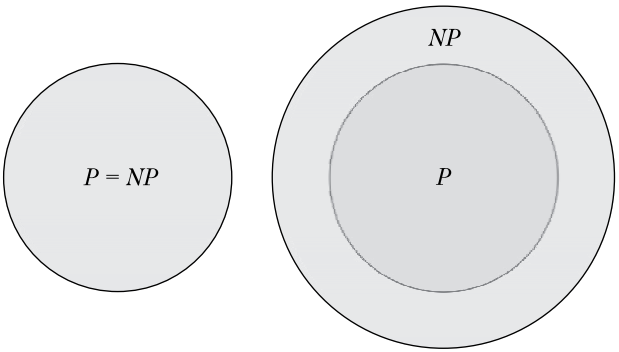


Рис. 3.1. Проблема $P = NP$: это одно и то же множество *(слева)* или же P является собственным подмножеством NP *(справа)*?

сложить значения в каждом множестве и определить, идентичны ли эти две суммы. Но выяснить, какие значения следует поместить в каждое из множеств, за полиномиальное время невозможно (при условии, что $P \neq NP$).

$$S = \{1, 1, 2, 3, 4, 5, 6\}$$
$$S_1 = \{1, 1, 2, 3, 4\}$$
$$S_2 = \{5, 6\}$$

Рис. 3.2. Простой пример задачи разбиения множества чисел. Мультимножество {1, 1, 2, 3, 4, 5, 6} можно разделить на мультимножества {1, 1, 2, 3, 4} и {5, 6}. Сумма элементов каждого из них равна 11

Некоторые задачи класса *NP* называются *NP*-сложными; *NP*-сложная задача (несколько рекурсивно) определяется как любая задача, которая по крайней мере столь же сложна, как и самая сложная задача класса *NP*. Чтобы понять, что это значит, рассмотрим принцип сокращения.

Углубляясь в подробности

Выше мы говорили о том, что задача разбиения не может быть решена за полиномиальное время при условии, что $P \neq NP$. Это верно только в том случае, если обратить внимание на важное различие между значением и размером входных данных.

Сложность задачи часто зависит от того, как входные данные закодированы. С технической точки зрения задача относится к классу P, если ее можно решить с помощью алгоритма, который выполняется за полиномиальное время, в зависимости от длины входных данных, то есть от числа битов, необходимых для представления входных данных. Алгоритм выполняется за псевдополиномиальное время, если он является полиномиальным, в зависимости от числового значения входных данных, длина которых изменяется экспоненциально. Рассмотрим задачу с количеством входных данных 1 миллиард и алгоритмом, который требует n операций. Если n — это количество цифр, необходимое для представления числа (в данном случае 10, при условии, что основанием системы счисления является 10), то алгоритм является полиномиальным. Если n — значение входных данных (1 000 000 000), то алгоритм будет псевдополиномиальным. В последнем случае количество требуемых операций растет намного быстрее, чем количество битов входных данных.

Задача B может быть сокращена до задачи C, если решение задачи C позволило бы решить задачу B за полиномиальное время. Другими словами, если у нас есть оракул, который дает ответ на задачу C, то мы можем (за полиномиальное время) преобразовать его в ответ на задачу B. Решение задачи является NP-сложным, если любая задача класса NP может быть сведена к этому

решению, то есть эффективное решение задачи также приведет к эффективному решению любой задачи класса *NP*.

Обратите внимание, что сама *NP*-сложная задача не обязательно должна принадлежать к классу *NP*; она должна быть как минимум такой же сложной, как все задачи класса *NP*. Это означает, что некоторые *NP*-сложные задачи (не принадлежащие классу *NP*) могут быть намного сложнее, чем другие задачи, которые принадлежат к классу *NP*.

Задача, которая и является *NP*-сложной, и относится к классу *NP*, называется *NP*-полной. Поскольку каждая *NP*-полная задача может быть сокращена до любой другой *NP*-полной задачи, сокращение *NP*-полной задачи к новой задаче и демонстрация того, что новая задача принадлежит к классу *NP*, достаточны, чтобы доказать, что данная задача является *NP*-полной.

Углубляясь в подробности

Если *NP*-полная задача имеет псевдополиномиальное решение, она называется слабо *NP*-полной. Если же это не так (или $P = NP$), то задача называется сильно *NP*-полной.

Для того чтобы ответить на вопрос, действительно ли $P = NP$, нужно либо предоставить алгоритм, который бы решал *NP*-полную задачу за (детерминированное) полиномиальное время (в этом случае $P = NP$), либо доказать, что такого алгоритма не существует (в таком случае $P \neq NP$).

Углубленные темы: NP-полное сокращение

Предположим, что у нас есть следующие задачи.

Сумма подмножества. Для заданного мультимножества S целых чисел и значения w определить, существует ли непустое подмножество множества S, сумма элементов которого равна w?

Разделение. Можно ли разделить мультимножество S целых чисел на два подмножества — S_1 и S_2, суммы элементов которых равны?

Поскольку сумма подмножества — *NP*-полная задача, можно доказать, что разбиение также является *NP*-полной задачей, следующим образом.

Первый шаг — показать, что задача разбиения относится к классу *NP*. Для данных подмножеств S_1 и S_2 можно найти сумму элементов каждого множества и сравнить их за время, пропорциональное общему количеству значений. Поскольку решение может быть проверено за линейное (полиномиальное) время, задача принадлежит к классу *NP*.

Второй шаг — показать, что разбиение является *NP*-сложным. Для этого мы покажем, что решение новой задачи за полиномиальное время также даст решение за полиномиальное время задачи, которая, как известно, является *NP*-сложной, поэтому разбиение — столь же сложная задача, как и эта задача.

Допустим, что у нас есть алгоритм, который решает за полиномиальное время и задачу разбиения, и задачу суммы подмножества, которую мы хотели бы решить. Есть множество S, и мы хотим знать, существует ли у него подмножество, сумма элементов которого равна w. Это эквивалентно вопросу, можно ли разбить множество S с суммой $|S|$ на мультимножество S_1 с суммой w_1 и мультимножество S_2 с суммой $w_2 = |S| - w$.

Пусть x — разность между w_1 и w_2. Добавим это число в множество S, а затем выполним алгоритм разбиения для нового множества. Если задача разрешима, то алгоритм вернет два разбиения с одинаковой суммой, которая равна половине от $|S| + x$. Вследствие того, как мы выбрали значение x, удаление этого элемента теперь дает множества S_1 и S_2, одно из которых является решением исходной задачи суммы подмножества. Поскольку дополнительная работа заняла линейное время, решение задачи суммы подмножества заняло полиномиальное время.

Пример:

$S = \{5, 10, 10, 30, 45\}$, $w = 25$

$|S| = 100$, $w_2 = 100 - 25 = 75$, $x = 75 - 25 = 50$

$S_a = \{5, 10, 10, 30, 45, 50\}$

$S_1 = \{5, 10, 10, 50\}$

$S_2 = \{30, 45\}$

Решение: {5, 10, 10}

Этот пример показывает, что задача разбиения является *NP*-полной, а ее решение за полиномиальное время было бы доказательством того, что $P = NP$.

Существует еще несколько классов сложности, но те два, которые мы обсудили в этом разделе, наиболее известны. Другие классы, как правило, определяются в терминах машин Тьюринга, которые мы рассмотрим позже в этой книге.

Дополнительные источники информации

Некоторые наиболее известные *NP*-полные задачи рассмотрены в приложении Б. Более подробную информацию о классах задач вы найдете в главах 13 и 14.

Часть II
Графы и графовые алгоритмы

4 Введение в теорию графов

4.1. Семь кенигсбергских мостов

Я считаю, что введение в теорию графов обязательно должно начинаться с задачи о кенигсбергских мостах. Эта задача важна не сама по себе, а потому, что с нее начался новый раздел математики — теория графов.

В городе Кенигсберге (ныне Калининград) протекает река. Она делит город на четыре части, которые соединяются семью мостами (рис. 4.1). Вопрос: можно ли прогуляться по городу, проходя каждый мост ровно один раз?

Однажды этот вопрос задали математику Леонарду Эйлеру. Он объявил задачу тривиальной, но она все же привлекла его внимание, поскольку ни одна из существующих областей математики не была достаточной для ее решения. Главным заключением является то, что топологические деформации неважны для решения; другими словами, изменение размера и формы различных деталей не меняет задачу при условии, что не меняются соединения[1].

---

[1] Эйлер назвал это геометрией положения.

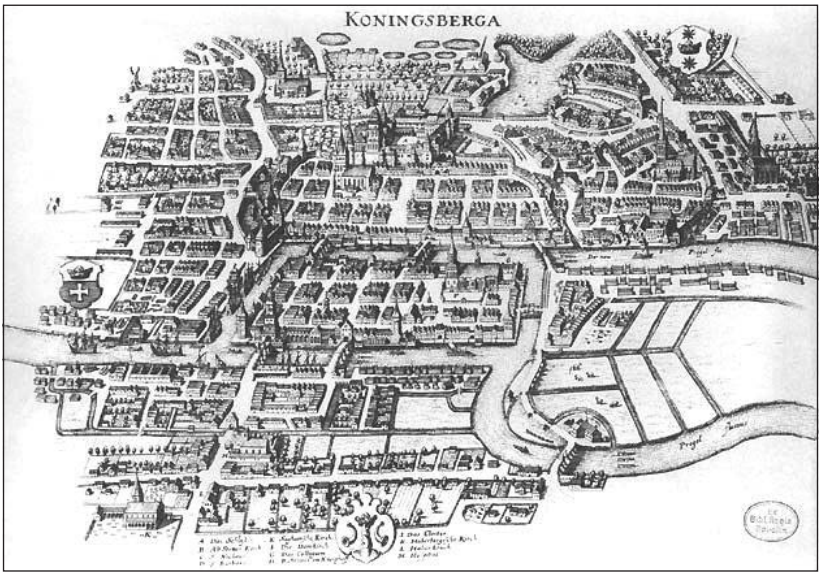


Рис. 4.1. Находящаяся в открытом доступе карта Кенигсберга. Merian-Erben, 1652

Таким образом, мы можем упростить карту, изображенную на рис. 4.1, заменив каждую часть города вершиной, а каждый мост — ребром, соединяющим две вершины. В результате получим граф, показанный на рис. 4.2.

Ключевым логическим заключением[1] является тот факт, что для того, чтобы сначала попасть на сушу, а затем покинуть ее, требуется два разных моста. Поэтому

[1] Solutio Problematis ad Geometriam Situs Pertinentis // Commentarii academiae scientiarum Petropolitanae 8, 1741. P. 128–140. («Решение одной проблемы, относящейся к геометрии положения»). Английский перевод доступен в книге: *Biggs N., Lloyd E. K., Wilson R. J.* Graph Theory, 1736–1936.

любой участок суши, который не является начальной или конечной точкой маршрута, должен быть конечной точкой для четного числа мостов. В случае Кенигсберга к каждой из четырех частей города вело нечетное число мостов, что делало задачу неразрешимой. Путь через граф, который проходит через каждое ребро ровно один раз, теперь называется эйлеровым путем.

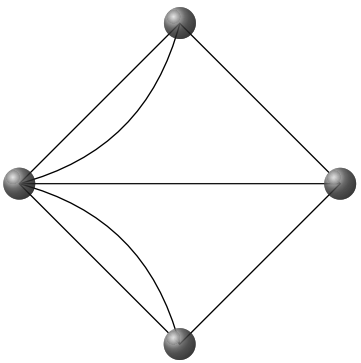


Рис. 4.2. Кенигсбергские мосты в виде графа. Обратите внимание, что каждая вершина имеет нечетную степень, то есть ее касается нечетное количество ребер

4.2. Мотивация

Графы в Computer Science чрезвычайно важны: очень многие задачи можно представить в виде графов. В случае с кенигсбергскими мостами представление города в виде графа позволяет игнорировать неважные детали — фактическую географию города — и сосредоточиться на важном — на связях между различными частями города. Во многих случаях возможность свести

постановку задачи к эквивалентной задаче для определенного класса графов дает полезную информацию о том, насколько сложно ее решить. Некоторые задачи являются *NP*-сложными на произвольных графах, но имеют эффективные (часто $O(n)$) решения на графах, имеющих определенные свойства.

В этой и последующих главах вы познакомитесь с терминами, связанными с графами и некоторыми распространенными структурами данных, которые используют графы. Вы узнаете, как представлять графы визуально, а также в форматах, более удобных для вычислений. После этого вы узнаете об известных графовых алгоритмах и некоторых основных графовых классах. К тому времени, когда вы закончите читать часть II, вы будете понимать, в каких случаях можно применять методы построения графов при решении задач.

Определение

Класс графов — это (как правило, бесконечный) набор графов, который обладает определенным свойством; принадлежность данного графа к этому классу зависит от того, есть ли у этого графа это свойство.

Пример

Двудольные графы — это такие графы, в которых вершины можно разделить на два множества, так что каждое ребро соединяет вершину из одного множества с вершиной из другого множества.

4.3. Терминология

Граф — это способ представления взаимосвязей в множестве данных. Графы часто изображаются в виде кружков, которыми обозначаются вершины, и линий между ними, представляющих ребра. Но вы узнаете и о других способах представления графов. Две вершины являются смежными, если между ними есть ребро, и несмежными, если между ними нет ребра.

Вершины графа также называются узлами; эти два термина, как правило, взаимозаменяемы. Однако точка многоугольника, в которой встречаются два ребра и более, всегда будет вершиной, а участок памяти, в котором содержится вершина и ее набор ребер, — узлом.

4.3.1. Части графов

Я буду часто ссылаться на подграф или порожденный подграф графа. *Подграфом* графа является любое количество вершин графа вместе с любым количеством ребер (которые также принадлежат исходному графу) между этими вершинами. *Порожденный подграф* — это любое подмножество вершин вместе со всеми ребрами графа, соединяющими эти вершины.

Строгое подмножество множества содержит меньше элементов, чем исходное множество; другими словами, собственное подмножество вершин графа содержит меньше вершин, чем исходный граф, в то время как регулярное подмножество может содержать все множество вершин.

4.3.2. Графы со всеми ребрами или без ребер

Полный (под)граф, или клика, — это такой граф, который содержит все возможные ребра между его вершинами. *Независимое* (или внутренне *устойчивое*) множество — это множество вершин без ребер между ними. На рис. 4.3 показан граф K_8; в теории графов буква K с целочисленным индексом означает полный граф с соответствующим количеством вершин.

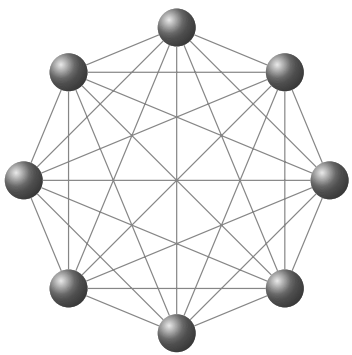


Рис. 4.3. K_8 — полный граф из восьми вершин

Граф или подграф, в котором существует путь от любой вершины к любой другой вершине, называется *связным*; граф, который не является связным, состоит из нескольких компонент связности[1].

[1] Компонентой связности графа является максимально связный подграф — множество вершин графа такое, для которого существует путь от любой вершины множества к любой другой вершине множества, и при этом ни у одной вершины нет ребра, ведущего к вершине, не принадлежащей этому множеству.

Для данного графа G его дополнение G' также является графом с теми же вершинами; для любой пары вершин G' ребро между ними существует тогда и только тогда, когда такого ребра нет в графе G. На рис. 4.4 показан граф, который является дополнением к графу K_6; вместо того чтобы содержать все возможные ребра, он не имеет ни одного ребра.

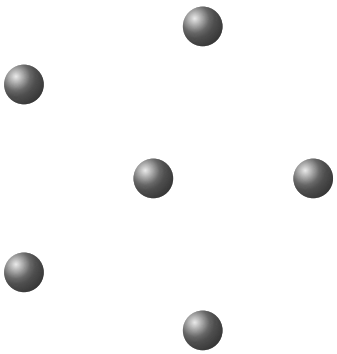


Рис. 4.4. Разъединенный граф с шестью компонентами связности размером 1

Математическое предупреждение

Символ штриха (') используется в математике для представления объекта, который был получен из другого объекта. Здесь я использую этот символ для обозначения графа G', который мы получаем из G, удаляя из него все ребра и добавляя ребра там, где их раньше не было. Это похоже на использование данного символа в теории множеств для обозначения дополнения множества.

При обсуждении языков в главе 13 используется альтернативная нотация — обозначение дополнения языка A как $\bar{A}$. Эта нотация (также широко распространенная) облегчает работу с дополнениями дополнений.

4.3.3. Петли и мультиграфы

Обычно мы работаем с простыми графами, то есть с графами, которые не содержат петель (ребро, которое заканчивается в той же вершине, в которой началось) или кратных ребер (несколько ребер, соединяющих одни и те же вершины). Говоря «граф» и не указывая иное, мы всегда имеем в виду простой граф. Граф с петлями можно назвать непростым графом, а граф с кратными ребрами — мультиграфом. Далее в книге, встретив слово «граф», считайте, что оно означает простой неориентированный граф[1], если не указано иное.

4.4. Представление графов

Когда мы говорим о размере графа, то обычно используем обозначение n для числа вершин и m — для числа ребер[2]. На рис. 4.3 $n = 8$ и $m = 28$, на рис. 4.4 — $n = 6$ и $m = 0$. Количество места в памяти, необходимое для хранения графа, зависит от того, как именно мы его

---

[1] Ориентированные графы мы рассмотрим в разделе 4.5.

[2] Обычно множество вершин обозначают буквой V, а множество ребер — буквой E, поэтому $|V| = n$, а $|E| = m$; то есть n — размер множества V, а m — размер множества E.

храним; как правило, графы хранятся в виде списков смежности и матриц смежности.

4.4.1. Представление графов в виде списков смежности

При использовании представления в виде списка смежности каждая вершина графа сохраняется вместе со списком смежных с ней вершин (рис. 4.5, 4.6). Если реализовать это в виде множества связных списков, то объем занимаемого пространства составит $O(n + m)$[1]. В случае *разреженного* графа (с очень небольшим числом ребер) этот объем сократится до $O(n)$. Для *плотного* графа (графа с большим количеством ребер, такого как полный или почти полный граф) этот объем составит $O(n^2)$.

A: BCDEFGH
B: ACDEFGH
C: ABDEFGH
D: ABCEFGH
E: ABCDFGH
F: ABCDEGH
G: ABCDEFH
H: ABCDEFG

Рис. 4.5. Представление списка смежности графа, показанного на рис. 4.3

A:
B:
C:
D:
E:
F:

Рис. 4.6. Представление списка смежности графа, показанного на рис. 4.4

[1] У нас есть n связных списков, некоторые из них могут быть пустыми (если граф разъединенный). Каждое ненаправленное ребро присутствует в обоих списках, поэтому общее количество узлов в списках составляет $2m$. В сумме это дает $O(n + m)$ объема памяти.

4.4.2. Представление графов в виде матрицы смежности

Еще одним популярным способом хранения графа является матрица смежности (рис. 4.7, 4.8), которая представляет собой матрицу со следующими свойствами:

- каждая ячейка матрицы равна либо 0, либо 1;
- ячейка в позиции (i, j) равна 1 тогда и только тогда, когда существует ребро между вершинами i и j. Это верно и для ячейки в положении (j, i);
- из предыдущего пункта следует, что число единиц в матрице равно удвоенному числу ребер в графе;
- диагональные ячейки всегда равны нулю, потому что ни у одной вершины нет ребра, которое и начинается, и заканчивается в ней[1];

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

Рис. 4.7. Представление матрицы смежности графа, показанного на рис. 4.3

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

Рис. 4.8. Представление матрицы смежности графа, показанного на рис. 4.4

[1] Диагональ матрицы — это ячейки, для которых номер столбца совпадает с номером строки: (0, 0), (1, 1) и т. д. На рис. 4.7 приведен пример матрицы, которая имеет нулевые значения только на диагонали.

- матрица состоит из n строк и n столбцов и поэтому занимает n^2 места. Для плотного графа линейная зависимость от размера матрицы сохраняется[1].

В мультиграфе, где присутствуют петли, некоторые из этих условий не выполняются. В частности, значения могут быть больше 1 (потому что между двумя вершинами может существовать несколько ребер) и диагональ может быть ненулевой (петли могут соединять вершину с самой собой).

4.4.3. Представление графов в памяти

В памяти граф часто хранится в виде набора узлов. Каждый узел представляет одну вершину и содержит набор указателей на другие узлы, так что каждый указатель соответствует ребру, ведущему к другой вершине.

4.4.4. Выбор представлений

Выбор представления графа зависит от плотности графа и от того, как вы планируете его использовать. Для разреженного графа список смежности намного эффективнее по объему памяти, чем матрица смежности, поскольку нам не нужно хранить $O(n^2)$ нулей и можно легко перебрать все существующие ребра. Кроме того, в случае динамического графа (который изменяется со временем) в списке смежности проще добавлять и удалять вершины.

С другой стороны, в матрице смежности более эффективно организован доступ к ребрам. Чтобы определить,

---

[1] Размер матрицы — это сумма количества вершин (n) и количества ребер (m), но для плотного графа $m = O(n^2)$.

являются ли вершины i и j смежными, достаточно проверить, равно ли $A[i][j]$ единице. Не нужно сканировать весь список, что может занять до $O(n)$ времени. Таким образом, в матрице смежности поиск выполняется не только быстрее, но и занимает постоянное количество времени, что делает матрицы смежности лучшими для приложений, где необходима предсказуемость[1].

4.5. Направленные и ненаправленные графы

В задаче о кенигсбергских мостах все ребра графа были *ненаправленными*; если по мосту можно идти в одном направлении, то можно идти и в обратном. Графы, для которых это верно, называются ненаправленными или просто графами и описывают ситуации, в которых если (вершина) A связана с B, то B также связана с A. Например, если Алиса — двоюродная сестра Веры, то Вера также является двоюродной сестрой Алисы.

В *ориентированном* графе, или *орграфе*, каждое ребро имеет направление, в котором следует данная связь. Если Алиса любит проводить время с Верой и мы обозначаем это стрелкой из A в B, это еще не говорит о том, что Вера тоже любит проводить время с Алисой. Если это так, то также существует стрелка из B в A. Орграф является *симметричным*, когда для каждого ориентированного ребра в нем существует ребро, соединяющее те же две вершины, но направленное в противоположную

[1] В приложениях реального времени следует быть готовыми пожертвовать некоторой производительностью, чтобы получить более жесткие ограничения на максимальное время, которое может потребоваться для выполнения операции.

сторону. Такой граф эквивалентен ненаправленному графу с одним ребром между каждой парой вершин, где есть пара направленных ребер, поэтому обычные графы можно рассматривать как особый случай орграфов.

И наоборот, возможна ситуация, при которой между любыми двумя вершинами может существовать только одно ориентированное ребро; такой граф называется *направленным* или *антисимметричным*. Например, если Алиса является родителем Влада, то Влад не может быть родителем Алисы (по крайней мере никаким социально приемлемым способом). Если взять ненаправленный граф и задать направление каждому из ребер (превратить ребро в ориентированное), получим направленный граф.

4.6. Циклические и ациклические графы

Один из способов классификации графов — разделить их на *циклические* или *ациклические*. Ациклический граф имеет не более одного пути между любыми двумя вершинами; другими словами, не существует пути *a*-*b*-*c*-*a*, где {*a*, *b*, *c*} — различные вершины графа[1]. Циклический граф имеет хотя бы один цикл: можно найти путь, который начинается и заканчивается в одной и той же вершине (рис. 4.9). В случае ориентированных графов добавляется условие, что все ребра цикла должны иметь одинаковое направление — по часовой

[1] В данном случае на месте *c* также может быть несколько вершин — это может быть цикл *a*-*b*-*c*-*d*-*e*-*f*-*g*-*h*-*a*. Обратите внимание, что каждое ребро можно использовать только один раз: переход по пути *a*-*b*-*c* и затем возвращение назад по тем же ребрам — это не цикл!

стрелке или против часовой стрелки; другими словами, в циклическом орграфе можно найти путь от вершины к самой себе, следуя по направлению стрелок. При программировании графовых алгоритмов необходимо соблюдать осторожность при обработке циклов, иначе программа может попасть в бесконечный цикл.

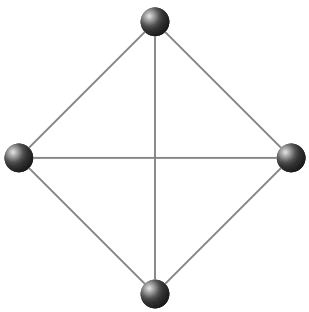


Рис. 4.9. Любой полный граф является циклическим

4.6.1. Деревья

Многие важные алгоритмы в теории графов оперируют с *деревьями*. Дерево — это просто связный граф, не имеющий циклов. Как правило, удобно описывать дерево, начиная с корня; мы назначаем одну вершину корнем дерева и определяем остальные вершины по их отношению к корню. Следующие определения для дерева являются эквивалентными:

- ациклический граф, в котором появится *простой* (без повторяющихся вершин) цикл, если добавить в него любое ребро[1];

[1] Это должен быть связный граф; понимаете ли вы почему?

- связный граф, который перестанет быть связным, если удалить из него любое ребро;
- граф, в котором любые две вершины связаны уникальным простым путем.

Узлы дерева бывают двух типов: внутренние узлы (у которых есть хотя бы один дочерний элемент) и листья (у которых нет дочерних элементов) (рис. 4.10).

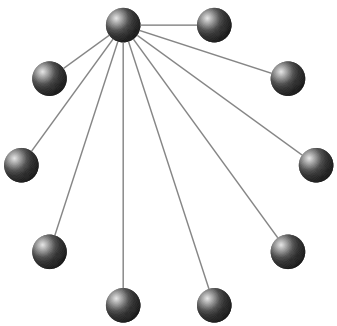


Рис. 4.10. Дерево с десятью вершинами. Это звезда, представляющая собой дерево ровно с одним внутренним узлом

4.6.2. Бесхордовые циклы

В большинстве случаев нас особенно интересуют бесхордовые циклы. *Хорда* — это ребро между двумя вершинами цикла, которое само не является частью цикла. Бесхордовый цикл — это цикл, который состоит хотя бы из четырех вершин и не содержит хорд (рис. 4.11). Другими словами, это такой цикл, в котором нет ребер

между любыми двумя вершинами, не являющимися последовательными в цикле[1].

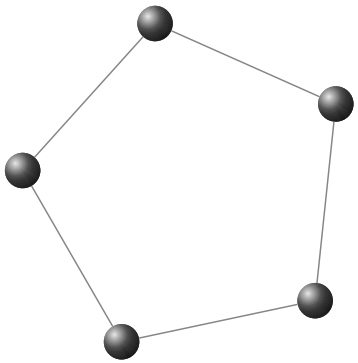


Рис. 4.11. C_5, бесхордовый цикл из пяти вершин

В главе 7 речь пойдет о нескольких классах графов, которые (по крайней мере частично) определяются отсутствием порожденных бесхордовых циклов, являющихся порожденными подграфами, содержащими циклы без хорд. Граф без порожденных бесхордовых циклов называется, что неудивительно, хордальным графом.

4.7. Раскраска графа

Многие графовые задачи связаны с раскраской — способом маркировки вершин (или ребер) графа. Правильная раскраска вершин — такая, при которой смежные

[1] Например, если у нас есть бесхордовый цикл *A-B-C-D-E-A*, то у *A* нет ребра, ведущего к *C* или *D*, поскольку эти вершины не стоят непосредственно перед или после *A* в цикле.

вершины (то есть вершины, между которыми есть ребро) раскрашены в разные цвета. Другими словами, раскраска — это разделение вершин на независимые множества.

Математическое предупреждение

Когда мы говорим о поиске раскраски графа, это не значит, что мы должны в буквальном смысле использовать цвета. Иногда при создании раскраски в роли цветов может выступать набор целых чисел.

Даже если мы используем реальные цвета, компьютер все равно будет обрабатывать их как список целых чисел, а затем при визуализации преобразовывать целые числа в цвета.

Аналогично правильная раскраска ребер — это такая, при которой два ребра, инцидентные[1] одной и той же вершине, раскрашены в разные цвета. Если не указано иное, то, говоря о раскраске графа, мы будем иметь в виду правильную раскраску вершин.

Первые результаты раскраски графов включали в себя раскраску плоских графов[2] в виде карт. В гипотезе о четырех цветах[3], выдвинутой в 1852 году, говорится, что для правильной раскраски любой карты, состоящей

---

1 Два ребра называются инцидентными, если они имеют общую вершину; аналогично две вершины, соединенные общим ребром, называются смежными.

2 Это графы, которые можно нарисовать так, чтобы никакие два ребра не пересекались, кроме как в вершине; подробнее см. в разделе 7.2.

3 Доказано, теперь это теорема о четырех цветах.

только из связных областей[1] с границами конечной длины, требуется не более четырех цветов.

Доказательство этого утверждения было представлено Альфредом Кемпе в 1879 году и считалось общепринятым, пока в 1890 году не было доказано, что оно неверно[2]. В итоге задачу решили с помощью компьютера в 1976 году; теперь у нас есть алгоритмы квадратичного времени для любой четырехцветной карты[3]. До сих пор не существует доказательств, не требующих использования компьютера.

Несмотря на то что задача раскраски карты, в сущности, не представляла особого интереса для создателей карт, она интересна с теоретической точки зрения и имеет практическое применение. Так, судоку — это форма раскраски графа, «цветами» которого являются числа от единицы до девяти.

Хроматическое число графа — это количество цветов, необходимых для его правильной раскраски. Другая формулировка теоремы о четырех цветах состоит в том, что хроматическое число плоского графа — не больше четырех.

Очевидно, что хроматическое число графа, не имеющего ребер, равно единице (все вершины могут быть одно-

---

1 В соответствии с этим требованием континентальная часть США, Аляска и Гавайи будут считаться отдельными регионами. На самом деле Гавайи были бы несколькими регионами.

2 Первая опубликованная исследовательская работа автора этой книги была посвящена контрпримеру к ошибочному доказательству Кемпе.

3 Или же для любого плоского графа.

го цвета). Для полного графа из n вершин хроматическое число равно n (каждая вершина смежна со всеми остальными вершинами, поэтому все вершины должны быть разных цветов).

Проверка того, может ли произвольный граф быть двухцветным, занимает линейное время[1] — достаточно окрасить одну вершину в красный цвет, затем окрасить все ее соседние вершины в синий, потом все соседние вершины этих вершин, которые еще не были окрашены, — в красный и т. д. Работа прекращается, когда либо все вершины окрашены, либо обнаружена вершина, у которой соседняя вершина имеет тот же цвет. А вот трехцветная раскраска является *NP*-полной задачей! Известно, что алгоритмы, определяющие, является ли граф k-раскрашиваемым, занимают экспоненциальное время. Однако, если знать, что граф принадлежит конкретному классу[2], найти раскраску можно за полиномиальное время.

Алгоритмы раскраски обычно используются в таких приложениях, как планирование, анализ данных, работа в сетях и т. п. Например, рассмотрим задачу назначения времени для собраний длительностью один час, когда на разные собрания приглашаются разные люди и используется разное оборудование. Мы представляем каждое собрание в виде вершины и добавляем ребро между двумя вершинами, если в них задействованы одни и те же человек или оборудование. Нахождение

---

[1] То есть время пропорционально сумме количества вершин и ребер.

[2] Например, идеальные графы, описанные в разделе 7.3.

минимальной раскраски говорит нам о необходимости назначить разное время для встреч (рис. 4.12).

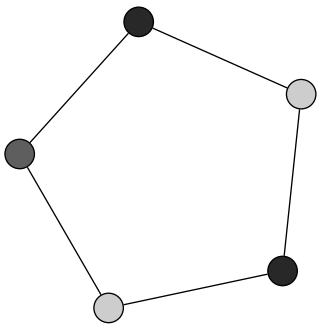


Рис. 4.12. Граф C_5 с минимальной раскраской

4.8. Взвешенные и невзвешенные графы

Вершины графа можно представить как локации, а ребра — как пути между ними, но в действительности не все пути имеют одинаковую длину. В *невзвешенном* графе ребра просто показывают, между какими вершинами есть прямой путь, но существуют также *взвешенные* графы, в которых каждому ребру присвоен вес. Обычно, но не всегда эти веса являются неотрицательными целыми числами. Вес часто воспринимается как стоимость использования этого ребра.

То, что конкретно означает вес, зависит от того, что описывает граф. На графе крупных городов ребра с весами могут представлять расстояния в милях по кратчайшему шоссе между двумя городами. На электрической схеме веса — это максимальный ток через данное соединение.

5 Структуры данных на основе графов

При проектировании алгоритмов мы часто используем абстрактные структуры данных в том смысле, что знаем свойства, которыми должна обладать структура, а не ее точная организация. Например, очередь с приоритетом — это структура данных FIFO (first in, first out — «первым пришел, первым ушел»), в которой элементы с более высоким приоритетом проходят вне очереди и обслуживаются раньше, чем элементы с более низким приоритетом. При выполнении реального алгоритма мы должны превратить эту абстрактную структуру данных в реальную. Так, очередь с приоритетом может быть реализована с помощью двоичного дерева поиска или же с помощью кучи. С кучей вы познакомились в подразделе 2.4.2; в этой главе более подробно рассмотрим ее внутреннее устройство.

5.1. Двоичные деревья поиска

Двоичное дерево поиска (Binary Search Tree, BST) (рис. 5.1) — это корневое двоичное дерево[1], которое рекурсивно определяется следующим образом: ключ

[1] Двоичное дерево — это дерево, каждый узел которого имеет не более двух дочерних узлов.

корня у него больше или равен ключу его левого потомка и меньше или равен ключу правого потомка (если он есть). Это также верно для поддерева, корнем которого является любой другой узел.

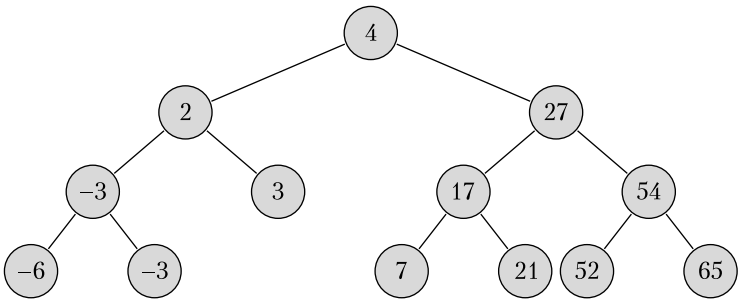


Рис. 5.1. Двоичное дерево поиска

Операции над двоичным деревом поиска занимают время, пропорциональное высоте дерева — длине самой длинной цепочки от корня (высота которого равна нулю) до листа. В общем случае[1] это $\Theta(\lg n)$[2], но в худшем случае (когда у каждого узла есть только один дочерний элемент, что, по существу, превращает дерево в связный список) высота составляет $O(n)$. Некоторые вариации двоичного дерева поиска гарантируют, что высота дерева будет $\Theta(\lg n)$, поэтому мы можем быть уверены, что все

---

[1] Предположим, что вы назначили корнем случайный узел. Тогда в среднем половина оставшихся узлов окажется слева от него, а половина — справа. Таким образом, ожидается, что количество узлов на каждом уровне дерева будет примерно вдвое больше, чем на предыдущем, следовательно, во всем дереве будет примерно $\lg n$ уровней.

[2] Θ означает, что время выполнения ограничено как сверху, так и снизу; высота дерева должна быть не менее $\lg n$.

операции завершатся за $O(\lg n)$ времени. На рис. 5.2 изображено несбалансированное двоичное дерево поиска.

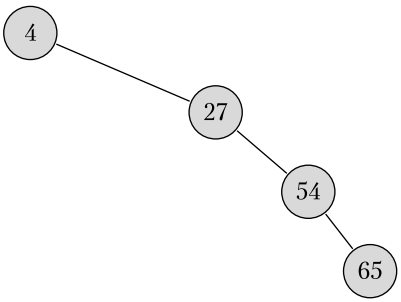


Рис. 5.2. Несбалансированное двоичное дерево поиска. Операции с ним занимают $O(n)$ времени

Двоичное дерево поиска может быть реализовано как коллекция связанных узлов, где каждый узел имеет ключ и указатели на левый и правый дочерние элементы и на родительский элемент. Поскольку узлы расположены так, что любой узел не меньше, чем любой из узлов его левого поддерева, и не больше, чем любой из узлов его правого поддерева, то можно вывести все ключи по порядку, выполнив упорядоченный обход дерева.

```
Процедура PrintInOrder(node x)
begin
    if x ≠ null then
        PrintInOrder(x.left);
        print x.key;
        PrintInOrder(x.right);
    end
end
```

Поиск по двоичному дереву выполняется просто: начиная с указателя на корень дерева и имея заданный ключ, мы сначала проверяем, есть ли у корня этот ключ. Если ключ корня меньше заданного, мы выбираем правый дочерний узел, если же он больше, то левый. Поиск прекращается, когда либо обнаружено совпадение ключей, либо поддерево, для которого мы пытаемся выполнить рекурсию, пусто. Чтобы найти наименьший элемент, мы просто всегда выполняем рекурсию для левого дочернего элемента, а чтобы найти наибольший элемент — для правого. В каждом случае мы будем проверять не более одного узла на каждом уровне дерева, поэтому время выполнения в худшем случае пропорционально высоте дерева.

Как создать и модифицировать двоичное дерево поиска? Первый добавленный узел становится корнем дерева. Чтобы добавить другие узлы, мы ищем значение ключа для вставки; обнаружив нулевой указатель, меняем его так, чтобы он указывал на новый узел, и делаем текущий узел родительским для нового узла[1].

Чтобы удалить узел *d*, нужно найти его в дереве и затем выполнить следующее:

- если у *d* нет потомков, просто присвоить null указателю родительского узла, который сейчас ссылается на *d*;

[1] Если в дереве допускаются дубликаты, нам может встретиться еще один или несколько узлов с таким же значением ключа, прежде чем мы найдем нулевой указатель.

- если у d есть один потомок, этот потомок занимает место d;
- если у d два потомка, то найти его предшественника или преемника (узлы, которые будут появляться непосредственно перед или после d при упорядоченном обходе) и также переместить его вверх, на место d (после чего может потребоваться обработать потомков этого узла так, как если бы он был удален).

5.2. Сбалансированные деревья двоичного поиска

Для того чтобы гарантировать, что операции над двоичным деревом поиска занимают $O(\lg n)$, а не $O(n)$ времени, необходимо ограничить высоту дерева. Если все ключи заранее известны, мы, конечно же, можем построить сбалансированное дерево (имеющее минимально возможную высоту).

На практике элементы дерева время от времени меняются, поэтому мы допускаем, что его высота может быть больше минимальной, но при этом составит $\Theta(\lg n)$.

Двоичное дерево поиска с автоматической балансировкой — это дерево, которое автоматически сохраняет небольшую высоту (по сравнению с количеством требуемых уровней) независимо от добавления или удаления узлов. Деревьями с автоматической балансировкой

являются, в частности, красно-черные деревья[1], косые деревья[2] и декартовы деревья (treaps).

5.3. Кучи

Как показано в подразделе 2.4.2, куча — это структура данных, а именно почти полное двоичное дерево с корнем, ключ которого больше, чем ключ любого из его дочерних элементов (рис. 5.3). И это правило рекурсивно выполняется для любого поддерева, корнем которого является любой дочерний элемент[3]. «Почти полное» означает, что дерево максимально заполнено, за исключением, возможно, самого нижнего уровня, который заполняется слева направо.

---

[1] Красно-черное дерево — это двоичное дерево поиска, в котором все ключи хранятся во внутренних узлах, а листья — это нулевые узлы. Для таких деревьев также существуют дополнительные требования: каждый узел должен быть красным или черным, корень и все листья — черными, потомки красного узла — черными и каждый путь от узла к листу должен содержать одинаковое количество черных узлов. В результате этих ограничений путь от корня до самого дальнего листа не более чем в два раза превышает длину пути от корня до ближайшего листа, поскольку кратчайший возможный путь — это все черные узлы, а в самом длинном из возможных путей красные и черные узлы чередуются, поэтому ни один путь не может быть более чем в два раза длиннее другого.

[2] Косые деревья подробно рассмотрены в главе 33.

[3] Или же корень может быть меньше, чем его дочерние элементы, такая куча называется неубывающей. Для согласованности далее мы будем использовать невозрастающую кучу.

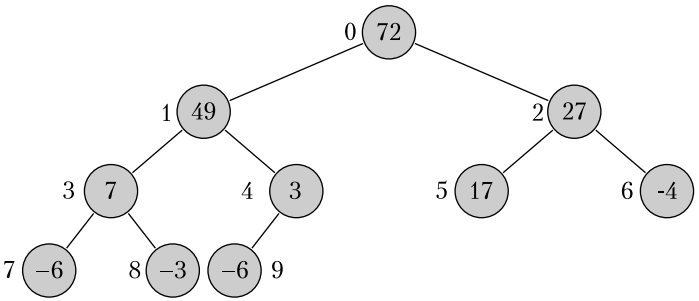


Рис. 5.3. Мах-куча с узлами, помеченными их позицией

Предположим, что мы хотим реализовать кучу. В этой куче вершины нумеруются начиная с корня, а затем на каждом уровне — слева направо: корень — вершина 0, его левый потомок — вершина 1, правый потомок — вершина 2, самый левый внук корня — вершина 3 и т. д. (рис. 5.4).

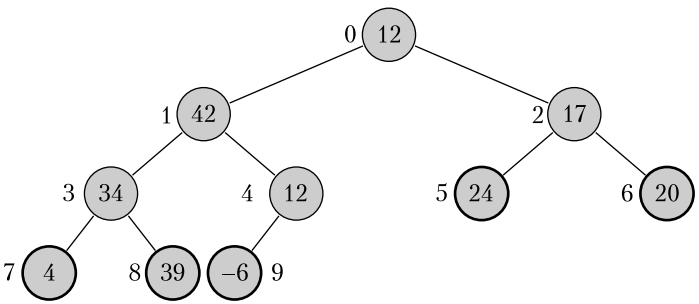


Рис. 5.4. В этом неупорядоченном двоичном дереве узлы с 5-го по 9-й являются корнями кучи размером 1

Обратите внимание, что каждая строка содержит в два раза больше узлов, чем строка, расположенная над ней.

Это означает, что метка левого дочернего элемента каждого узла как минимум вдвое плюс еще на единицу больше, чем номер родительского элемента[1]. Для любого узла k номер его родителя равен $\left\lfloor \frac{k-1}{2} \right\rfloor$, а его дочерние элементы (если они есть), имеют номера $2k + 1$ и $2k + 2$.

Математическое предупреждение

Приведенные выше L-образные операторы представляют функцию округления до ближайшего целого в меньшую сторону, что означает усечение любой дроби; $\lfloor 2.8 \rfloor = 2$. Обратная функция, округление до ближайшего целого в большую сторону, записывается как $\lceil 2.1 \rceil = 3$.

Если вы предпочитаете использовать массив с отсчетом от 1, то родительским узлом для k будет узел номер $\left\lfloor \frac{k}{2} \right\rfloor$, а его дочерние элементы будут иметь номера $2k$ и $2k + 1$.

Можно эффективно определить позицию родительского или дочернего узла, выполнив умножение посредством сдвига битов; если куча реализована на основе массива, то мы сразу получим положение нужного узла.

5.3.1. Построение кучи

Теперь, когда мы определили, как куча хранится в памяти, мы готовы ее построить. Куча строится рекурсивно:

[1] Или же просто удвоить массив с отсчетом от 1.

создаются маленькие кучи и объединяются в большие. Для начала предположим, что у нас есть массив A размером n, который мы интерпретируем как описанное ранее двоичное дерево (см. рис. 5.4). Обратите внимание, что каждый лист дерева (любой элемент в позиции от $\left\lfloor \frac{n}{2} \right\rfloor$ до $n - 1$) является корнем кучи размером 1.

Теперь рассмотрим внутренние узлы на втором снизу уровне дерева. Если ключ такого узла больше, чем ключ любого из его дочерних элементов, то он является корнем невозрастающей кучи. Если же нет, то мы поменяем его с дочерним элементом, ключ которого больше, чтобы этот дочерний элемент стал корнем невозрастающей кучи, и перейдем к новому дочернему элементу, чтобы убедиться, что новое поддерево сохраняет свойство кучи.

Затем мы продолжим объединять кучи таким образом, пока не будет обработан весь массив (рис. 5.5–5.7).

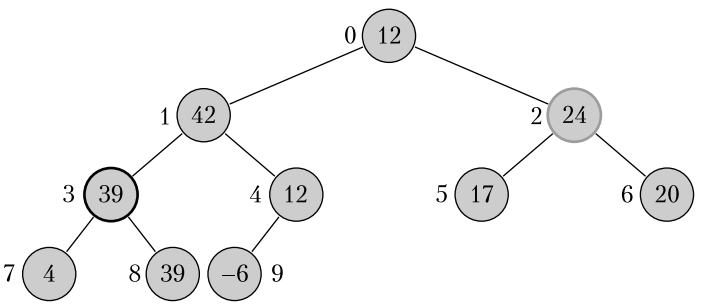


Рис. 5.5. Если лист больше, чем его родитель и сестра, то этот лист меняется местами с родителем

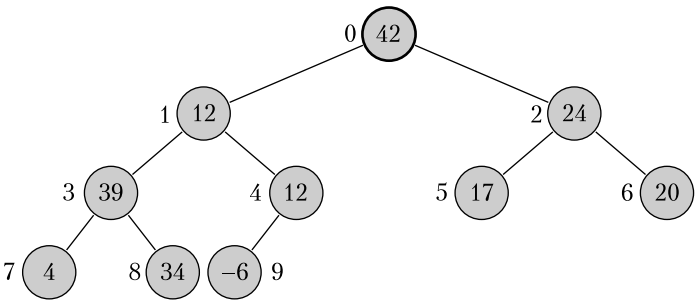


Рис. 5.6. Перенос дочернего элемента на более высокий уровень может уничтожить свойство кучи поддерева, корневым элементом которого был этот дочерний элемент. В данном случае новый узел 1 меньше, чем один из его дочерних узлов

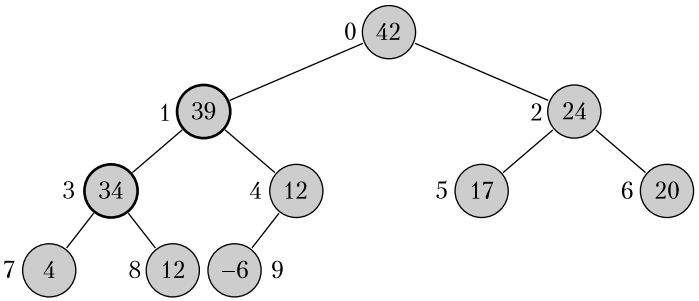


Рис. 5.7. В итоге ни одно значение узла не превышает значение его родительского элемента, так что граф удовлетворяет свойству кучи

Этот процесс можно представить в виде следующего кода (алгоритм 1). Кроме `.length`, который, как обычно,

означает количество элементов массива, мы определяем `.heapSize` — число элементов массива, которые являются частью кучи.

Алгоритм 1. BuildMaxHeap

Входные данные: массив A
Выходные данные: массив A, отсортированный как невозрастающая куча

```
begin
    A.heapSize = A.length
    for i = ⌈A:length / 2⌉ downto 1 do
        MaxHeapify(A, i)
    end
end
```

В `BuildMaxHeap` мы перебираем все внутренние узлы, начиная с самого нижнего. Для каждого выбранного узла мы проверяем, больше ли он, чем каждый из его дочерних узлов, которых может быть не более двух. Если это так, то, поскольку каждый дочерний элемент уже является корнем невозрастающей кучи, этот узел теперь является корнем новой невозрастающей кучи большего размера.

Если это не так, то мы меняем данный элемент на больший из двух его дочерних элементов. Это может привести к тому, что поддерево с корнем в данном дочернем элементе больше не будет удовлетворять свойству кучи (в случае если новый элемент был меньше не только чем его дочерний элемент, но также и дочерний элемент

этого дочернего элемента, как было в случае, показанном на рис. 5.5), поэтому мы снова вызываем алгоритм `MaxHeapify` для этого поддерева (алгоритм 2).

Алгоритм 2. MaxHeapify

```
Входные данные: массив A, индекс i
begin
    left = LeftChild(i)
    right = RightChild(i)
    if left < A.heapSize и A[left] > A[largest]
        then
            largest = left
        else
            largest = i
    end
    if right < A.heapSize и A[right] > A[largest] then
        largest = right
    end
    if largest ≠ i then
        Swap(A[i], A[largest])
        MaxHeapify(A, largest)
    end
end
```

В худшем случае каждый раз при объединении куч новый корень должен перемещаться в самое основание дерева. Каждый узел, который обрабатывается таким образом, может быть сравнен с $O(\lg n)$ потомками, причем каждое сравнение занимает постоянное количество времени. Умножив это на n узлов, получим общее время выполнения алгоритма $O(n \lg n)$.

Алгоритм 3. Heapsort

Входные данные: массив A

Выходные данные: отсортированный массив A

```
begin
    BuildMaxHeap(A)
    for i = A.length downto 2 do
        Swap(A[1], A[i])
        A.heapSize = A.heapSize – 1
        MaxHeapify(A, 1)
    end
end
```

В алгоритмах `BuildMaxHeap` и `MaxHeapify` параметр `.heapSize` может показаться избыточным. Проверка того, является ли какой-либо узел меньше `A.heapSize`, всегда будет возвращать `true`, поскольку размер кучи задан равным размеру массива. Однако при запуске `Heapsort` этот узел отмечает конец оставшихся неотсортированных элементов кучи.

В алгоритме `Heapsort` (см. алгоритм 3) после построения кучи корень (который является самым большим элементом) перемещается в конец массива, где он и остается. Затем мы уменьшаем значение `heapSize` и снова вызываем `MaxHeapify` для остальной части массива.

Каждый раз, когда значение `heapSize` уменьшается, оставшиеся `heapsize – 1` элементов массива, отличные от корня, образуют кучу и вызов `Heapify` для корня восстанавливает свойство кучи. Элемент, перемещенный в конец массива, был самым большим из оставших-

ся элементов, поэтому после k-й итерации в позициях `[heapsize ... n]` находятся k самых больших элементов в отсортированной последовательности.

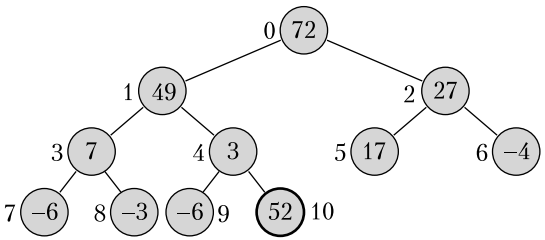
5.3.2. Добавление элементов в кучу

Элемент, добавляемый в кучу, вставляется в ближайшее доступное место (то есть в первую пустую ячейку массива) (рис. 5.8). Если при этом новый элемент больше родительского элемента, это может нарушить свойство неубывания. В таком случае нужно поднять этот элемент вверх, рекурсивно меняя его местами с родительскими элементами, пока они меньше этого элемента.

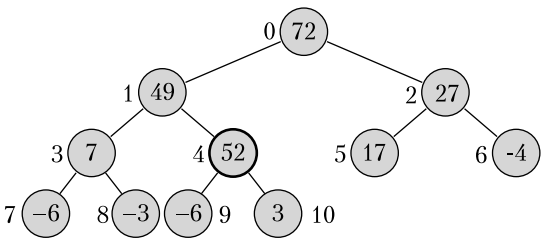
Поскольку максимальная высота дерева равна $O(\lg n)$ и каждое сравнение/обмен занимает время $O(1)$, общее время вставки равно $O(\lg n)$.

5.3.3. Удаление корня из кучи

Чтобы извлечь первый элемент из кучи, достаточно удалить первый элемент массива; однако при этом в куче остается пустое место, которое мы заменяем последним элементом кучи, при необходимости оставляя почти полное дерево (рис. 5.9). Затем мы поднимаем новый корень вверх, заменяя его дочерними узлами с бóльшими значениями, пока дерево снова не приобретет свойство кучи.



Мы помещаем новый элемент (с ключом 52) в первую открытую ячейку, из-за чего куча может потерять (и в данном случае теряет) свойство неубывания



Мы восстанавливаем свойство кучи, поднимая новый узел в правильную позицию

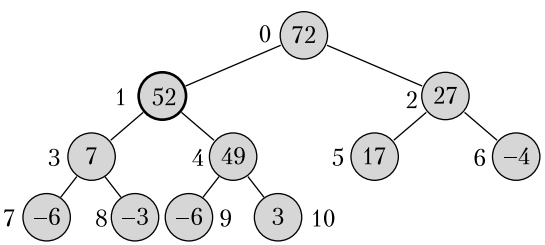
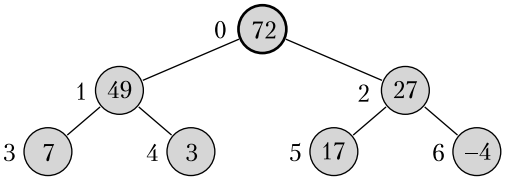
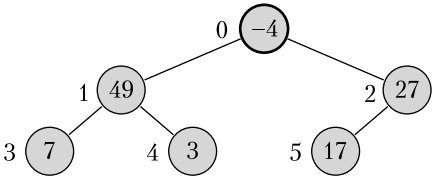


Рис. 5.8. Добавление элемента в кучу



Чтобы удалить корень,
мы заменяем его последним элементом
с самого низкого уровня кучи



Затем мы добавляем новый корень
в нужное место в куче

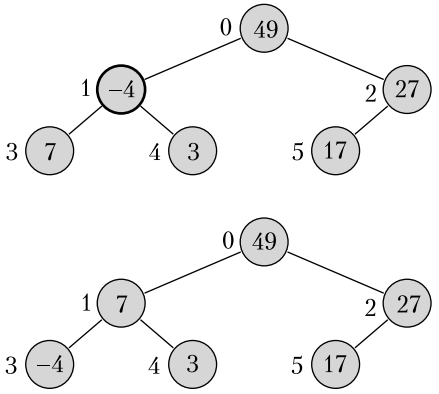


Рис. 5.9. Удаление корня из кучи

6 Хорошо известные графовые алгоритмы

6.1. Введение

Есть алгоритмы на графах, которые встречаются в различных приложениях настолько часто, что знать о них вы просто обязаны. Многие из них представляют собой различные способы сортировки графа или поиска подграфа с определенным свойством. Понимание относительных преимуществ этих фундаментальных алгоритмов — ключ к определению, когда использовать каждый из них в практических приложениях.

Эта глава начинается с описания алгоритмов поиска в ширину (Breadth-First Search, BFS) и в глубину (Depth-First Search, DFS), которые превращают произвольный граф в дерево поиска. В обоих случаях мы назначаем одну из вершин графа корнем дерева и рекурсивно исследуем ребра каждой вершины, пока все вершины графа[1] не будут добавлены в остовное

[1] То есть любая вершина, которой можно достичь, переходя по ребрам от корня; в случае связного графа это будут все вершины.

дерево[1]. Рассмотрим, для каких типов приложений полезен каждый вид дерева поиска, а затем перейдем к поиску кратчайших путей.

При анализе времени выполнения алгоритмов на графах обычно используются две системы обозначений. Одна из них — это обозначение числа вершин буквой n, а числа ребер — буквой m. Другая — указать размеры множеств вершин и ребер.

Для графа G множество вершин обозначается как $V(G)$ [читается «вершины G»), а множество ребер — как $E(G)$ (читается «ребра G»), поэтому $n = |V(G)|$, а $m = |E(G)|$.

Математическое предупреждение

Напомню, что в математике заключение значения в прямые скобки означает получение абсолютного значения или размера; здесь это означает размер множества, поэтому $|V(G)|$ можно прочитать как «размер множества вершин G». Если G подразумевается по умолчанию, то можно обращаться к множествам просто как V и E.

6.2. Поиск в ширину

При поиске в ширину мы сначала исследуем все вершины, которые примыкают к корню; затем исследуем

[1] Остовное дерево графа — это дерево, которое содержит все вершины графа.

все вершины, смежные с соседями корня, которые еще не были исследованы, и т. д. Таким образом, вначале мы исследуем все вершины, которые находятся на расстоянии k от корня (который обычно обозначается буквой s, от слова *source* — «источник»), и лишь затем переходим к вершинам на расстоянии $k + 1$.

Когда алгоритм завершает работу, глубина каждого узла дерева — это минимальное количество ребер (то есть длина кратчайшего пути), по которому можно добраться до этого узла из s как в дереве, так и в исходном графе. Чтобы найти этот путь, нужно идти по указателям на родительские узлы, пока не будет достигнут узел s.

Каково время выполнения поиска в ширину? При инициализации для каждой вершины инициализируются три упомянутых выше свойства, что занимает постоянное время для каждой вершины, в сумме $O(n)$. Затем каждая вершина добавляется в очередь, удаляется из очереди, и каждое из свойств изменяется не более одного раза — каждая из этих операций занимает $O(1)$ времени, так что для всех вершин снова получаем $O(n)$. Наконец, мы перебираем всех соседей каждой вершины, чтобы проверить, отмечены ли они, что занимает еще $O(m)$ времени (алгоритм 4). В сумме получаем общее время выполнения $O(n + m)$.

Для хранения графа требуется $n + m$ места в памяти, а очередь занимает $O(n)$ места, поэтому в сумме требуется пространство объемом $O(n + m)$. Это также означает, что алгоритм поиска в ширину работает за

линейное время по отношению к размеру входных данных (который равен $n + m$).

Алгоритм 4. Поиск в ширину

Входные данные: произвольный граф, корнем которого выбрана одна вершина, и все вершины имеют такие свойства:

- distance (расстояние) — изначально равно бесконечности;
- parent (родитель) — изначально равно нулю;
- marked (помечена) — изначально равно false.

Выходные данные: остовное дерево графа, каждая вершина которого максимально приближена к корню

```
begin
    Инициализировать очередь Q
    Задать s.distance = 0
    Пометить s
    Добавить в очередь s
    while Q не пустая do
        Удалить из очереди u
        foreach вершина v ∈ Adj(u) do
            if v.marked then
                продолжить
            end
            Задать v.parent = u
            Задать v.distance = u.distance + 1
            Пометить v
            Добавить в очередь v
        end
    end
end
```

6.3. Применение поиска в ширину

Как показано ранее, поиск в ширину позволяет найти длину кратчайшего пути от исходной вершины *s* до любой другой вершины графа за линейное время. Затем мы можем найти сам путь, следуя по родительским указателям от исходного узла до корня дерева, за время, пропорциональное длине пути. Это означает, что алгоритм поиска в ширину полезен для всех задач, где нужен поиск кратчайших путей. Вот несколько примеров таких задач.

6.3.1. Навигационные системы

Рассмотрим задачу построения маршрутов с помощью GPS. Если картографическая система хранит данные о локальной области в виде графа, вершинами которого являются адреса (или пересечения), а ребрами — улицы (или, точнее, короткие отрезки улиц), то она может выполнять поиск в ширину для дерева, источник которого — ваше текущее местоположение.

6.3.2. Проверка, является ли граф двудольным

При запуске поиска в ширину, если существует ребро между данной вершиной и уже отмеченной вершиной, расстояние которой либо такое же, либо меньше в степени 2, этим двум вершинам будет присвоен один и тот же цвет и граф не будет двудольным. Кроме того, ребро

вместе с одним или несколькими путями к наименьшему общему предку двух вершин является нечетным циклом, который будет признаком того, что граф не двудольный[1]. Двудольные графы используются в теории кодирования[2], в сетях Петри[3], в анализе социальных сетей и облачных вычислениях.

6.4. Поиск в глубину

Как и при поиске в ширину, при поиске в глубину мы начинаем с исходной вершины и рекурсивно проходим остальную часть графа (алгоритм 5). Но только теперь мы продвигаемся настолько глубоко, насколько можем, по одному пути и только потом исследуем другие пути.

Представьте, что вы изучаете собак. Студент, использующий алгоритм поиска в ширину, может начать с изучения названий всех пород — от австралийской короткохвостой пастушьей собаки до японского шпица — и лишь потом углубиться в детали. Тот же, кто использует поиск в глубину, начнет с австралийской короткохвостой, которая является пастушьей породой, поэтому он прочитает все о пастушьих собаках, что приведет его к чтению

---

[1] Сертификат — это доказательство того, что ответ, возвращаемый программой, является правильным; подробнее об этом вы прочитаете в главе 19.

[2] Теория кодирования изучает, в частности, криптографию, сжатие данных и исправление ошибок.

[3] Сети Петри используются для моделирования поведения систем.

о стадах, это, в свою очередь, — к истории о домашних животных, что приведет к... В общем, идею вы поняли. Если область поиска слишком велика или даже бесконечна, может использоваться модифицированная версия поиска в глубину, которая работает только до указанной глубины.

Алгоритм 5. Поиск в глубину

Входные данные: произвольный граф, у каждой вершины которого есть свойства distance (расстояние) (изначально равно нулю) и marked (помечена) (изначально имеет значение false), а также исходная вершина s

Выходные данные: остовное дерево графа

```
begin
    Инициализировать стек S
    S.Push(s)
    while S не пустой do
        u = S.Pop(s)
        if u.marked then
            продолжить
        end
        Пометить u
        foreach вершина v ∈ Adj(u) do
            if v.marked then
                продолжить
            end
            Задать v.parent = u
            S.Push(v)
        end
    end
end
```

Практическое применение

Недавно меня попросили помочь решить задачу планирования. Было несколько заданий, которые требовалось выполнить, со следующими ограничениями:

- ни одно задание нельзя начать раньше определенного времени;
- каждое задание имеет максимальное время выполнения;
- у задания может быть ноль или более заданий, от которых оно зависит, и ноль или более заданий, которые зависят от него; выполнение задания нельзя начать, пока не будут выполнены все задания, от которых оно зависит. Циклических зависимостей нет (поэтому это орграф частично упорядоченного множества);
- на выполнение всех заданий может уйти не более 24 часов.

Поиск кратчайшего пути — распространенная задача, но в данном случае мы фактически хотели найти самый длинный путь для каждой задачи, что и сделали, используя модификацию алгоритма поиска в глубину. Это позволило нам построить такой порядок выполнения заданий, который гарантировал, что во время выполнения любого задания все предки этого задания в графе зависимостей уже были выполнены, так что самое длительное время выполнения задания было равно сумме самого позднего времени, требуемого для выполнения любого из родителей данного задания, и количества времени, отведенного для самого задания. Повторное выполнение алгоритма в обратном порядке

позволило выявить все цепочки зависимостей, которые будут выполняться после выделенного времени, что позволило выяснить, какие задачи необходимо оптимизировать.

6.5. Кратчайшие пути

Рассмотрим задачу наискорейшего перемещения из одного места в другое. В теории графов это задача кратчайшего пути: найти маршрут между двумя вершинами, имеющий наименьший вес. В невзвешенном графе это просто путь с наименьшим количеством ребер; во взвешенном графе это путь с наименьшим суммарным весом ребер.

Рассмотрим вариации этой задачи.

- **Кратчайший путь из одной вершины.** Найти кратчайший путь от исходного узла ко всем остальным узлам графа. Пример: узнать кратчайший путь от пожарной части до каждой точки города.
- **Кратчайший путь в заданный пункт назначения.** Найти кратчайший путь от каждого узла графа до данной точки назначения. Это просто задача поиска кратчайшего пути из одной вершины, для которой направление всех ребер изменено на противоположное. Например, мы можем захотеть узнать кратчайший путь до больницы из любой точки города.
- **Кратчайший путь между всеми парами вершин.** Найти кратчайший путь между всеми парами узлов графа. В идеале наш GPS сможет построить наилучший маршрут из любой точки в любую точку.

6.5.1. Алгоритм Дейкстры

Алгоритм Дейкстры[1] изначально определял кратчайший путь между двумя узлами графа, но был расширен таким образом, что теперь он решает задачу поиска кратчайшего пути из одной вершины (алгоритм 6). Этот алгоритм применим к любому взвешенному графу (напомню, что невзвешенный граф — это просто взвешенный граф, у которого вес каждого ребра равен 1), у которого все ребра имеют неотрицательный вес.

Алгоритм 6. Алгоритм Дейкстры

Входные данные: граф G и исходная вершина s
Выходные данные: расстояние от s до любого другого узла графа G
Инвариантное: S — множество узлов, для которых определены кратчайшие пути

```
begin
    Инициализировать множество вершин S нулем
    Инициализировать очередь с приоритетами Q
    и поместить в нее все вершины G
    while Q не пустая do
        u = Q.ExtractMin()
        S = S ∪ {u}
        foreach вершина v ∈ Adj(u) do
            Relax(u, v, w)
        end
    end
end
```

[1] www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf.

В алгоритме Дейкстры мы начинаем с исходной вершины *s* и множества узлов *S* (в настоящее время пустого), расстояние до которых было определено. Мы добавляем все элементы графа в очередь с приоритетом на основе их расстояния от *s*; сначала расстояние для *s* равно нулю; у всех остальных узлов расстояние равно бесконечности. Мы удаляем из очереди наименьший элемент (которым первоначально будет *s*) и «ослабляем» все его ребра. Далее мы продолжаем удалять наименьшие элементы и «ослаблять» его ребра до тех пор, пока очередь не станет пустой, после чего каждому узлу присваивается длина его кратчайшего пути (алгоритм 7).

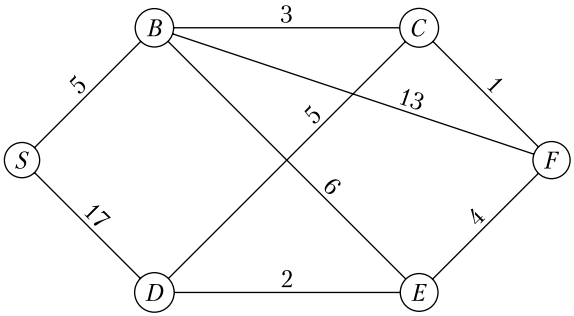
Алгоритм 7. «Ослабление ребра»

Входные данные: смежные вершины u и v и вес w ребра между ними
Выходные данные: v.d — вес найденного кратчайшего пути от s до v; если путь проходит через ребро от u до v, то v.π (родительский элемент v) присваивается значение u

```
begin
    if (v.d > u.d + w(u, v)) then
        v.d = u.d + w(u, v)
        v.π = u
    end
end
```

«Ослабление» ребра в данном случае означает, что мы берем приоритет текущего узла *u*, прибавляем к нему расстояние *w* до нового узла *v* и сравниваем сумму с текущим приоритетом *v*. Если сумма оказывается меньше существующего значения *v.d*, это значит, что мы нашли более короткий путь к *v* и можем заменить старое значение;

мы также помечаем *u* как нового родителя для *v*. Каждый раз, когда узел удаляется из очереди с приоритетом, мы знаем, что уже нашли кратчайший путь к этому узлу, потому что любой более короткий путь должен был бы пройти через уже обработанные узлы.



S	*B*	*C*	*D*	*E*	*F*
0	1	∞	∞	∞	∞
—	$\mathbf{5}_{S}$	∞	17_{S}	∞	∞
—	—	$\mathbf{8}_{B}$	17_{S}	11_{B}	18_{B}
—	—	—	13_{C}	11_{B}	$\mathbf{9}_{C}$
—	—	—	13_{C}	$\mathbf{11}_{B}$	—
—	—	—	$\mathbf{13}_{C}$	—	—

Рис. 6.1. Поиск всех кратчайших путей из множества *S* с помощью алгоритма Дейкстры. Каждая строка таблицы представляет один шаг, на котором мы обрабатываем ранее необработанную вершину с наименьшим весом

Это жадный алгоритм[1], то есть на каждом шаге он делает все, что кажется лучшим на данный момент. Поскольку

[1] Подробнее о жадных алгоритмах и других подходах к решению задач рассказывается в части IV.

мы считаем, что каждый узел в множестве S помечен длиной его кратчайшего пути, алгоритм Дейкстры (в отличие от многих других жадных алгоритмов) гарантированно возвращает оптимальное решение.

«Ослабление» ребра занимает постоянное время, что в сумме дает время $O(m)$ для обработки всех ребер. Поиск элемента с наименьшим приоритетом для последующей обработки занимает время $O(n)$, и это выполняется $O(n)$ раз, что дает общее время $O(n^2)$. В сумме это дает $O(n^2 + m)$, где $m \leq n^2$, поэтому сложность алгоритма Дейкстры составляет $O(n^2)$.

7 Основные классы графов

Многие задачи достаточно сложны для произвольных графов (являются *NP*-сложными), но имеют эффективные (или даже тривиальные) решения для графов определенного класса. И наоборот, задача может не иметь решения на графах определенного класса. Таким образом, мы часто можем избежать проблем, если удастся показать, что данная задача относится к определенному классу графов.

7.1. Запрещенные подграфы

Характеризация *запрещенными подграфами* для класса графов — метод, определяющий набор структур, которые не должны появляться в графе; наличие или отсутствие таких структур говорит о том, принадлежит ли данный граф к этому классу. Такие запрещенные подструктуры могут быть определены несколькими способами.

- **Графы.** Граф может принадлежать к классу, только если он не содержит подграфа из (возможно, бесконечного) множества. Например, двудольные

графы — это только такие графы, которые не содержат нечетных циклов.

- **Индуцированные графы.** То же самое, что и в предыдущем случае, за исключением того, что здесь нас интересуют только индуцированные подграфы (напомню, что это некоторое подмножество вершин графа вместе со всеми ребрами между этими вершинами). Например, хордальные графы — это такие графы, которые не содержат индуцированного бесхордового цикла длиной не менее четырех.
- **Гомеоморфные графы.** Два графа называются гомеоморфными, если один из них можно получить из другого, удалив вершины второй степени[1] и свернув все ребра в одно.
- **Миноры графа.** Минором графа называется подграф, получаемый посредством стягивания ребер, где стягивание ребер происходит, когда мы выбираем две смежные вершины и объединяем их в одну.

Как вы увидите в следующем разделе, класс графов может иметь несколько характеризаций запрещенными подграфами разных типов.

7.2. Планарные графы

В главе 4 описана задача о кенигсбергских мостах, которая включает в себя представление карты в виде графа. Класс графов, представляющих карты на плоскости

[1] Степень вершины — это количество ее ребер. В данном случае это означает, что все вершины имеют ровно два ребра.

(или сфере[1]), имеет специальное название: планарные графы. Формально планарный граф — это граф, который может быть построен на плоскости, следовательно, он может быть нарисован так, что все его ребра пересекаются только в вершинах графа. Любая плоская карта может быть представлена в виде плоского графа: для этого достаточно заменить каждую область вершиной и провести ребра там, где раньше были общие границы.

Теорема Куратовского классифицирует планарные графы в терминах запрещенных подграфов: планарные графы — это только такие графы, которые не содержат подграфов, гомеоморфных K_5 (полный граф из пяти вершин, приведен на рис. 7.1) или $K_{3,3}$ (полный двудольный граф из шести вершин, см. раздел 7.4, рис. 7.2).

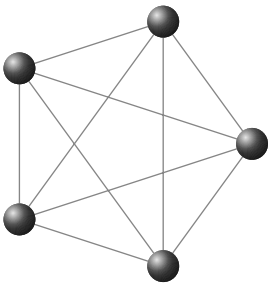


Рис. 7.1. K_5 — полный граф из пяти вершин

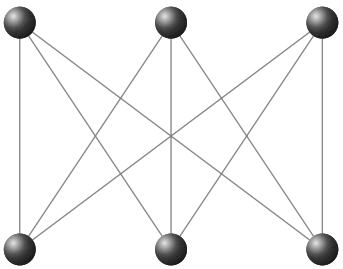


Рис. 7.2. Граф $K_{3,3}$, также известный как «домики и колодцы»

Теорема Вагнера классифицирует планарные графы как те, что не содержат те же подграфы в качестве миноров.

---

[1] Плоскость — это просто сфера с удаленным северным полюсом. Прости, Санта.

Рассмотрим следующее

Предположим, у нас есть граф K_5, который не является планарным, — нет возможности нарисовать его на плоскости хотя бы без одного пересечения ребер. Если добавить вершину в середину только одного ребра, это не устранит пересечение, поэтому новый граф также не будет планарным. Именно поэтому запрещенным является любой подграф, гомеоморфный K_5 или $K_{3,3}$.

Формула Эйлера гласит, что если конечный связный граф из v вершин, e ребер и f граней можно представить на плоскости без пересечений ребер[1], то $v - e + f = 2$.

Определение планарного графа (такого, который может быть построен на плоскости без пересечения ребер), естественно, означает, что такие графы удобны для отображения, так как легче увидеть, куда ведут все ребра (и фактически каждый такой граф можно нарисовать так, чтобы все ребра были отрезками прямых линий[2]). Приложение для рисования графов может разбить граф на плоские компоненты и объединить их, чтобы получить «более приятную» визуализацию. Это также имеет практическое применение в таких областях, как проектирование электронных схем.

Планарные графы можно разбить на более мелкие части, удалив $O(\sqrt{n})$ вершин, что помогает в разработке алгоритмов типа «разделяй и властвуй» и при динамическом программировании (см. главу 10) на планарных графах.

---

1 Это, конечно, доказывает, что граф планарный.

2 Теорема Фари.

7.3. Совершенные графы

В разделе 4.7 представлена концепция раскраски графа, где хроматическое число графа — это минимальное количество цветов, необходимое для его правильной раскраски. Для графа, который содержит полный подграф (порожденный подграф, содержащий все возможные ребра) размером k, очевидно, хроматическое число должно быть не менее k. Совершенным графом называется такой граф, для которого это строгое равенство (хроматическое число строго равно k) и в котором это строгое равенство для хроматического числа также выполняется для всех порожденных подграфов. Другими словами, граф совершенен тогда и только тогда, когда для этого графа и для всех его порожденных подграфов хроматическое число равно размеру наибольшей клики[1].

Свойство совершенности является *наследуемым*; это означает, что если граф совершенен, то все его порожденные подграфы также совершенны. Наследуемость совершенных графов следует из приведенного выше определения: если все порожденные подграфы графа удовлетворяют условию совершенности, то очевидно, что все порожденные подграфы этих подграфов также должны удовлетворять этому условию.

В действительности существует несколько эквивалентных определений совершенного графа[2]. Сильная

---

[1] Словами «клика» и «полный подграф» обозначается одно и то же понятие. Так бывает.

[2] Чтобы доказать, что эти определения действительно эквивалентны, потребовалось приложить усилия. Подробности читайте в книге Мартина Голамбика (Martin Golumbic) Algorithmic Graph Theory and Perfect Graphs.

теорема о совершенных графах характеризует их как такие графы, которые не содержат нечетных отверстий (порожденных бесхордовых циклов нечетной длины) или антиотверстий (дополнений этих циклов). Слабая теорема о совершенных графах характеризует их как дополнения к совершенным графам (то есть дополнение каждого совершенного графа само по себе является совершенным). Эта теорема непосредственно следует из доказательства сильной теоремы о совершенных графах.

Еще одной характеристикой совершенных графов является то, что для таких графов произведение размеров наибольшей клики и наибольшего независимого множества больше или равно числу вершин графа. Это также верно для всех порожденных подграфов.

В число задач, которые являются *NP*-сложными на произвольных графах, но решаются за полиномиальное время на совершенных графах, входит раскраска графа, определение максимальной клики и максимального независимого множества. Подклассы совершенных графов (некоторые из них рассмотрены в этой главе) включают в себя двудольные графы (то есть такие, у которых хроматическое число равно двум), хордальные графы (такие, у которых нет хордовых циклов длиной больше трех), графы сравнимости (которые отражают частично упорядоченное множество) и подмножества этих классов[1]. Как мы увидим в следующем разделе,

[1] Эти подмножества включают в себя интервальные графы — хордальные графы, которые можно изобразить в виде набора последовательных интервалов и лесов — графов, в которых каждый связный компонент представляет собой дерево.

эти подклассы являются совершенными графами с дополнительными ограничениями.

7.4. Двудольные графы

Двудольные графы — это графы, для которых хроматическое число равно двум. Здесь они рассмотрены главным образом для того, чтобы вы увидели, насколько легко можно распознать класс графа. Для этого нужно выбрать одну вершину графа из каждого связного компонента, присвоить ей первый цвет и добавить ее в очередь. Каждый раз, когда вершина удаляется из очереди, рассматриваются все ее соседки и, если какая-либо из них еще не окрашена, ей присваивается противоположный цвет, после чего она добавляется в очередь. Если соседняя вершина уже раскрашена в тот же цвет, что и выбранная вершина, то граф не является двудольным (и две вершины образуют часть нечетного цикла); в противном случае мы получаем на выходе 2-раскраску графа.

Двудольные графы применяются в сетях Петри (которые используются для описания распределенных систем) и для множества других известных задач, включая задачу об устойчивом браке[1] и задачу назначения[2].

---

[1] В задаче об устойчивом браке каждый человек из группы *A* сопоставляется с человеком из группы *B* таким образом, чтобы не было пар, в которых бы он предпочитал того, кто также предпочитает его.

[2] В задаче назначения каждый агент выполняет ровно одно задание (разные задания могут стоить по-разному, в зависимости от того, какой агент их выполняет); задания назначаются таким образом, чтобы свести к минимуму общую стоимость.

7.5. Интервальные графы

Интервальные графы — это графы, которые можно изобразить в виде набора пересекающихся отрезков, расположенных вдоль общей линии, где каждый отрезок представляет вершину; две вершины являются смежными тогда и только тогда, когда соответствующие сегменты линии пересекаются.

Как и совершенные графы, подклассом которых они являются, интервальные графы имеют несколько характеризаций. Характеризация запрещенными подграфами формулируется в терминах *астероидальных троек*. Они представляют собой наборы из трех вершин графа, где между любыми двумя вершинами существует такой путь, который избегает окрестности третьей вершины. Интервальные графы — это такие графы, которые являются хордальными и не содержат астероидальных троек. На рис. 7.3 представлена астероидальная тройка.

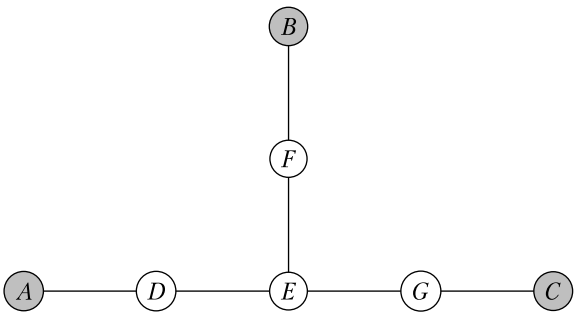


Рис. 7.3. Вершины *A*, *B* и *C* образуют астероидальную тройку; между любыми двумя из этих вершин существует путь, который не содержит никаких вершин, смежных с третьей

Вероятно, самым известным является применение интервальных графов в генетическом анализе; интервальные графы были использованы для определения того, что субэлементы гена с высокой вероятностью связаны друг с другом, образуя линейную структуру[1].

Интервальные графы обычно используются в задачах выделения ресурсов: каждый интервал представляет запрос ресурса, а раскраска графа соответствует выделению этих ресурсов. Например, если каждый интервал представляет урок, а два интервала пересекаются, если уроки в какой-то момент совпадают, то хроматическое число графа — это минимальное количество необходимых классных комнат, а раскраска распределяет уроки по этим комнатам.

7.6. Графы дуг окружности

Надмножеством интервальных графов[2] являются графы дуг окружности — такие графы, которые можно изобразить в виде набора пересекающихся дуг окружности (рис. 7.4). Две вершины такого графа будут смежными тогда и только тогда, когда соответствующие дуги пересекаются. Если граф дуг окружности можно представить так, чтобы окружность не была замкнута, то такой

---

1 О топологии тонкой генетической структуры читайте работы Сеймура Бензера (Seymour Benzer), 1959.

2 Другими словами, каждый интервальный граф также является графом пересечения дуг окружности. Это строгое отношение надмножеств, поскольку обратное неверно: многие графы пересечения дуг окружности не являются интервальными.

граф также интервальный; если разорвать круг в незамкнутой части и растянуть его в линию, то получим интервальное представление.

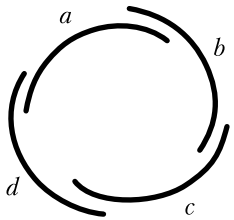


Рис. 7.4. C_4, бесхордовый цикл из четырех вершин, представленный в виде графа дуг окружности

В то время как интервальные графы являются строгим подмножеством как совершенных графов, так и графов дуг окружности, последние классы пересекаются. Класс графов пересечения дуг окружности содержит как совершенные графы (например, интервальные), так и графы, которые не являются совершенными (нечетные бесхордовые циклы). Подобно интервальным графам, графы дуг окружности можно классифицировать по их запрещенным подграфам.

Часть III
Неграфовые алгоритмы

8 Алгоритмы сортировки

Сортировка относится к тем задачам, о которых большинству программистов никогда не приходится думать; за них это делает язык программирования или библиотеки. Как правило, подходящий алгоритм сортировки выбирается автоматически в зависимости от объема входных данных. Например, на платформе .NET массивы сортируются методом вставки, пирамидальной или быстрой сортировки, в зависимости от задачи (https://msdn.microsoft.com/en-us/library/85y6y2d3(v=vs.110).aspx).

Так зачем же создавать собственный алгоритм сортировки? Возможно, вам нужна устойчивая сортировка (то есть такая, в которой два элемента с одинаковым ранжированием будут размещаться в одинаковом порядке друг относительно друга), а предоставляемые готовые алгоритмы сортировки не являются устойчивыми. Или же вы обладаете дополнительной информацией о сортируемых данных, что может значительно сократить время выполнения алгоритма.

Наиболее эффективно изменить алгоритм сортировки можно, если:

- данные и так уже почти отсортированы;
- данные расположены в обратном или почти в обратном порядке;
- данные представляют собой конечное число дискретных значений, относительно малое по сравнению с количеством сортируемых элементов.

В этой главе рассмотрены некоторые характерные алгоритмы сортировки. Сравним их преимущества и недостатки.

Большинство алгоритмов сортировки — сортировка сравнением, когда два элемента сортируемого списка сравниваются с помощью некоторой операции, которая определяет, что один из них меньше или равен другому (размещается перед ним). Сложность алгоритмов сортировки сравнением обычно определяется количеством требуемых операций сравнения.

8.1. Малые и большие алгоритмы сортировки

Первые два вида сортировки, которые мы рассмотрим, имеют неэффективную среднюю сложность. Это означает, что на больших наборах данных они будут слишком медленными, поэтому бесполезными. Однако для небольших наборов данных асимптотически неэффективный алгоритм может работать быстрее, чем асимптотически эффективный алгоритм.

Множество из восьми элементов алгоритм со сложностью $O(n^2)$ отсортирует примерно за 64 сравнения,

а алгоритм со сложностью $O(n \lg n)$ — всего за 24 сравнения, однако каждое сравнение может потребовать меньше дополнительной работы, и в результате асимптотически медленный алгоритм на практике работает быстрее. Однако при сортировке тысячи элементов вряд ли удастся преодолеть стократное асимптотическое ускорение более эффективного алгоритма.

Затем мы рассмотрим три алгоритма со сравнением, которые выполняются за среднее время $O(n \lg n)$, и два алгоритма, в которых невозможно подсчитать количество сравнений, потому что в них значения не сравниваются между собой.

Время выполнения сортировки сравнением в худшем случае составляет $O(n \lg n)$ — это лучшее, что можно получить, и это можно продемонстрировать следующим образом. Каждое сравнение определяет относительный порядок двух значений и поэтому может рассматриваться как внутренний узел сбалансированного двоичного дерева, каждый лист которого является одним из $n!$ возможных вариантов размещения элементов. В сбалансированном двоичном дереве с k листьями любой путь от корня до листа состоит из $\lg k$ внутренних узлов или вариантов выбора (в данном случае сравнений), поэтому существует $\lg n!$ сравнений; согласно приближению Стирлинга, $\lg n!$ имеет сложность $O(n \lg n)$.[1]

[1] Это упрощенное объяснение; более подробное объяснение с точными цифрами вы найдете в подразделе 5.3.1 книги «Искусство программирования» Дональда Кнута (Donald Knuth).

8.2. Сортировки для малых наборов данных

8.2.1. Сортировка пузырьком

Сортировка пузырьком (bubble sort) не имеет практической пользы — ее обычно приводят в качестве примера наивного алгоритма сортировки. Однако благодаря своей простоте она дает хорошее представление о сортировке. Алгоритм состоит в том, что каждая пара соседних элементов массива сравнивается и, если эти элементы не расположены в должном порядке, их меняют местами.

Пример: сортировка пузырьком

$$\begin{matrix}3\\1\\4\\2\end{matrix} \rightarrow \begin{matrix}1\\3\\4\\2\end{matrix} \rightarrow \begin{matrix}1\\3\\4\\2\end{matrix} \rightarrow \begin{matrix}1\\3\\2\\4\end{matrix} \qquad (8.1)$$

$$\begin{matrix}1\\3\\2\\4\end{matrix} \rightarrow \begin{matrix}1\\3\\2\\4\end{matrix} \rightarrow \begin{matrix}1\\2\\3\\4\end{matrix} \qquad (8.2)$$

$$\begin{matrix}1\\2\\3\\4\end{matrix} \rightarrow \begin{matrix}1\\2\\3\\4\end{matrix} \qquad (8.3)$$

На k-й итерации k самых больших значений гарантированно находятся в конце списка, поэтому после $n - 1$ итераций список отсортирован с общим временем

выполнения $O(n^2)$. Интересно, что это не только сложность наихудшей ситуации (по которой обычно измеряется эффективность алгоритмов), но и средняя сложность. Однако в лучшем случае время выполнения составит всего $O(n)$. Если список уже полностью или почти отсортирован, то, поскольку сортировка пузырьком способна это обнаружить (при этом не выполняется никаких перестановок), она эффективно работает для таких наборов данных. Это свойство присуще также сортировке вставками, о которой мы поговорим позже. Такая сортировка была бы более полезной во времена ленточных накопителей, когда последовательный доступ выполнялся намного быстрее, чем произвольный.

8.2.2. Сортировка вставками

Сортировка вставками (insertion sort) — еще один способ сортировки, который прост в реализации, но, в отличие от сортировки пузырьком, он действительно полезен для небольших наборов данных. Этот метод также обладает тем преимуществом, что он интуитивно понятен для большинства людей[1].

Сортировка вставками выполняется путем перебора элементов массива. Если размер массива равен единице, то такой массив явно упорядочен. Каждый элемент, добавляемый в конец массива (то есть при просмотре

[1] Вы когда-нибудь играли в бридж или преферанс? Большинство людей берут карты по одной и вставляют каждую карту на свое место среди уже отсортированных карт, которые держат в руке. Это и есть сортировка вставкой!

следующего элемента массива), либо больше, чем все остальные элементы, либо его можно переместить в правильное место массива, сместив все расположенные после него отсортированные элементы в конец на одну ячейку.

Пример: сортировка вставками

$$\begin{matrix}3\\4\\1\\2\end{matrix} \rightarrow \begin{matrix}3\\4\\1\\2\end{matrix} \rightarrow \begin{matrix}1\\3\\4\\2\end{matrix} \rightarrow \begin{matrix}1\\2\\3\\4\end{matrix} \qquad (8.4)$$

Подобно сортировке пузырьком, наилучшая сложность сортировки вставкой (когда массив уже отсортирован) равна $O(n)$, а наихудшая сложность (когда массив отсортирован в обратном порядке) равна $O(n^2)$. На практике алгоритм быстро работает для небольших массивов и обычно применяется для сортировки небольших участков массивов в рекурсивных алгоритмах, таких как быстрая сортировка и сортировка слиянием[1].

У сортировки вставками есть еще несколько приятных особенностей. В частности, она стабильна, выполняется на месте и *онлайн*. Онлайн-алгоритм — это такой алгоритм, в котором значения могут обрабатываться по мере их поступления; они могут быть недоступны все сразу с самого начала.

---

[1] После того как однажды мне пришлось проверить 160 ученических работ и выставить оценки, я привык сортировать вставкой стопки из 8–10 работ, объединять эти стопки методом сортировки слиянием, а затем записывать оценки.

8.3. Сортировка больших наборов данных

8.3.1. Пирамидальная сортировка

Подобно предыдущим алгоритмам сортировки, пирамидальная сортировка делит входные данные на отсортированную и неотсортированную части и поэтапно перемещает элементы в отсортированную часть. В отличие от предыдущих методов сортировки пирамидальная сортировка требует предварительной обработки данных. Сначала из данных строится куча, а затем самый большой элемент кучи многократно извлекается и вставляется в массив. Построение кучи требует $O(n)$ операций[1]. Извлечение из кучи максимального элемента и восстановление кучи требуют $O(\lg n)$ операций. Поскольку нужно извлечь все n элементов, наилучшая и наихудшая производительность составляет $O(n \lg n)$[2]. Обратите внимание: хотя асимптотическая сложность для наилучшего и наихудшего случаев одинакова, на практике наилучшее время выполнения будет примерно в два раза быстрее[3].

8.3.2. Сортировка слиянием

При сортировке слиянием (merge sort) список сортируется рекурсивно: массив делится на несколько мас-

---

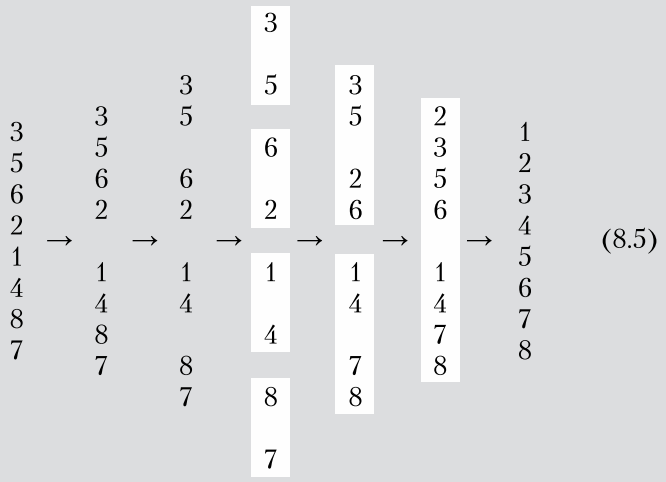
[1] $O(n \lg n)$ для некоторых вариаций.

[2] *Schaffer R., Sedgewick R.* The Analysis of Heapsort // Journal of Algorithms. Vol. 15. Issue 1, July, 1993.

[3] *Bolloba's B., Fenner T. I., Frieze A. M.* On the Best Case of Heapsort // Journal of Algorithms. Vol. 20. Issue 2, March, 1996.

сивов меньшего размера, которые сортируются, а затем объединяются. При чистом слиянии можно продолжать деление до тех пор, пока каждый набор не будет содержать только одно значение. Затем множества объединяются; имея два отсортированных набора размером k, мы можем создать полностью упорядоченный объединенный набор, при этом число сравнений составит от k до $2k - 1$.

Пример: сортировка слиянием



(8.5)

Это дает $O(\lg n)$ комбинированных шагов, каждый из которых занимает $O(n)$ времени, так что общее время выполнения алгоритма составляет $O(n \lg n)$.

На практике вместо того, чтобы делить исходный набор данных на отдельные элементы, обычно прекращают деление, когда наборы достаточно малы, чтобы

использовать алгоритм с наилучшим временем выполнения для небольших наборов данных (например, сортировку вставками).

8.3.3. Быстрая сортировка

Алгоритм быстрой сортировки (quick sort) интересен тем, что его наихудшее время выполнения $O(n^2)$ хуже, чем у сортировки слиянием и пирамидальной сортировки, но на практике он обычно работает в 2–3 раза быстрее этих алгоритмов, его среднее время достигает $O(n \lg n)$[1].

Быстрая сортировка работает так: выбирается опорный элемент, после чего остальные элементы массива упорядочиваются так, чтобы все элементы слева от опорного были меньше или равны, а все элементы справа — больше или равны опорному элементу. Затем алгоритм рекурсивно вызывается для левого и правого сегментов массива и так далее до достижения базового случая (когда каждый сегмент содержит одно или ноль значений).

Фактическое количество требуемых итераций зависит от выбора опорного элемента и от значений, которые необходимо отсортировать. Например, метод Ломуто состоит в том, чтобы просто выбрать в качестве опорного последний элемент массива (или сегмента массива); это легко реализовать, но тогда, если массив уже отсор-

---

[1] *Скиена С.* Алгоритмы. Руководство по разработке. — СПб.: BHV, 2017.

тирован, время выполнения составит $O(n^2)$, потому что на каждой итерации сегмент будет уменьшаться только на один элемент. Чтобы этого избежать, можно выбрать в качестве опорного элемент, находящийся в середине массива, случайный элемент или медиану из трех случайных элементов. В идеале опорный элемент будет точной медианой сортируемых данных, а каждый из двух новых разделов будет содержать половину оставшихся данных. При известном (неслучайном) методе выбора опорного элемента злоумышленник может увеличить время выполнения до $O(n^2)$, намеренно размещая данные таким образом, чтобы опорный элемент каждый раз находился с одной и той же стороны от элемента, с которым он сравнивается.

Пример: быстрая сортировка

$$\begin{matrix}4\\5\\7\\2\\1\\3\\8\\\boxed{6}\end{matrix} \rightarrow \begin{matrix}4\\5\\2\\1\\\boxed{3}\\6\\7\\\boxed{8}\end{matrix} \rightarrow \begin{matrix}2\\\boxed{1}\\3\\4\\\boxed{5}\\6\\7\\8\end{matrix} \rightarrow \begin{matrix}1\\2\\3\\4\\5\\6\\7\\8\end{matrix} \qquad (8.6)$$

По аналогичным причинам быстрая сортировка также будет плохо работать с массивами, содержащими

большое количество повторяющихся значений[1], но это можно исправить, используя сортировку не на два, а на три раздела (меньше, равно, больше). В этом случае средний раздел не нуждается в дальнейшей сортировке, поэтому при сортировке массивов с большим количеством дубликатов можно получить более быстрые результаты (а если все элементы одинаковы, то сортировка займет линейное время).

Преимущества быстрой сортировки, по сравнению с сортировкой слиянием, включают в себя сортировку на месте и более простой внутренний цикл, благодаря чему в большинстве случаев алгоритм работает быстрее[2]. Недостатками этого метода являются наихудшее время выполнения и отсутствие стабильности[3]; переупорядочение на месте также не работает, если данные представлены не в виде массива, а в виде связного списка.

8.4. Сортировки без сравнения

Как уже упоминалось, минимальное время выполнения в худшем случае, которое мы можем получить для сортировки сравнением, составляет $O(n \lg n)$. Чтобы улучшить время выполнения, нужна дополнительная инфор-

---

1 Эта задача известна как задача о голландском национальном флаге.

2 *Скиена С.* Алгоритмы. Руководство по разработке. — СПб.: BHV, 2017.

3 Причина нестабильности быстрой сортировки, как правило, состоит в том, что во время разбиения могут переставляться элементы с одинаковым ключом.

мация (метаданные) о сортируемых данных. Иногда это позволяет улучшить время сортировки до $O(n)$.

В частности, даже если количество сортируемых элементов достаточно велико, может существовать ограниченное количество ключевых значений, по которым эти элементы должны быть отсортированы. На этом принципе основаны следующие методы сортировки.

8.4.1. Сортировка подсчетом

Предположим, что каждый из n элементов сортируемых данных имеет ключ, который является положительным целым числом со значением не более k. Тогда можно создать массив длиной k и перебрать в цикле все элементы, которые должны быть отсортированы, используя этот массив для отслеживания количества вхождений каждого ключа (выражаясь более формально, мы строим частотную гистограмму для значений ключа).

Теперь у нас есть массив значений, соответствующих количеству появлений каждого ключа. Затем остается еще раз пройти по массиву и заменить каждое значение на количество ключей с меньшими значениями. После этого каждая ячейка массива будет содержать правильную позицию для первого элемента с этим ключом.

Наконец, остается переместить каждый элемент исходных данных в позицию, которая определяется значением, записанным в соответствующей ячейке массива ключей, и увеличить это значение на единицу (чтобы следующий элемент с таким ключом, если он есть, разместить в следующей ячейке выходного массива).

Пример: сортировка подсчетом

Рассмотрим следующий массив, в котором цифра является ключом для сортировки, а буква — прикрепленным значением: [1a, 3b, 2a, 4c, 1c, 4b, 2a, 2c, 2b, 1b, 3c].

Пройдя по массиву, мы обнаружили три единицы, четыре двойки, две тройки и две четверки и получили гистограмму [3, 4, 2, 2].

Это говорит о том, что первый элемент с каждым ключом должен появляться в следующих позициях: [0, 3, 7, 9].

Теперь снова пройдем по исходному массиву. Первый элемент равен 1 и перейдет в ячейку 0. Второй элемент имеет ключ 3, он пойдет в ячейку 3 и т. д. Каждый раз, помещая элемент в выходной массив, мы увеличиваем соответствующее значение в массиве ключей на единицу. Например, после размещения первых семи элементов массив ключей принял вид [2, 5, 8, 11], и мы знаем, что следующий элемент, 2c, попадет в ячейку 5 выходного массива.

Весь процесс требует двух циклов для входного массива (один для заполнения массива ключей и один — для перемещения элементов в выходной массив) и одного цикла для массива ключей, чтобы просуммировать значения и определить конечные позиции ключей. Кроме того, должны быть инициализированы массив ключей (размером k) и выходной массив (размером n). Тогда общее время выполнения, как и занимаемое место в памяти, составит $O(n + k)$. Если k мало по сравнению с n (например, для большого количества элементов, которые ранжируются по шкале от 1 до 100), то это $O(n)$. Это стабильный метод сортировки.

8.4.2. Поразрядная сортировка

Поразрядная сортировка — это более старый метод сортировки[1], который до сих пор остается полезным; в зависимости от допущений, он может быть более эффективным, чем лучшие варианты сортировки сравнением. Его сложность составляет $O(wn)$ для ключей с длиной слова w; если w постоянна, это сводится к $O(n)$. Естественно, это требует дублирования ключей; если каждый сортируемый элемент имеет уникальный ключ, то длина ключа должна быть не менее $\lg n$[2]. Существует несколько вариантов поразрядной сортировки; здесь мы рассмотрим сортировку по младшему разряду (least significant digit, LSD).

Рассмотрим алфавит, из которого взят ключ, и создадим по одной ячейке для каждой буквы этого алфавита. Например, если ключ является десятичным числом, то у нас будет десять ячеек, помеченных цифрами от 0 до 9. Каждый элемент списка мы поместим в ячейку, соответствующую младшему разряду ключа этого элемента. Например, элемент с ключом 378 попадает в ячейку номер 8. Другими словами, мы сгруппировали ключи по младшему разряду, но сохранили их относительный порядок (чтобы это была стабильная сортировка).

---

[1] Метод использовался Германом Холлеритом (Herman Hollerith) для счетно-аналитических машин в 1887 году; одна из компаний, основанных Холлеритом, в итоге стала IBM.

[2] Для основания b и ключа длиной w количество различных ключей равно b^w. Если w меньше $\lg_b n$, то количество возможных ключей меньше количества сортируемых элементов.

Затем мы снова сгруппируем ключи, но теперь по второму с конца разряду; теперь элемент с ключом 378 окажется в ячейке номер 7. И так далее. После того как будут перебраны все цифры в ключе, список отсортирован. Один из способов реализовать это — представить каждую ячейку в виде очереди; после каждого прохода элементы можно возвращать обратно в список.

Почему это работает? Рассмотрим два ключа на последнем этапе процесса. Если их старшие разряды различны, то ключ, который должен стоять первым при сортировке, попадет в предшествующую ячейку, поэтому они будут правильно расположены друг относительно друга. Если старшие разряды ключей одинаковы, то они попадут в одну ячейку в том же порядке, в котором были отсортированы для предыдущего разряда, что по индукции также будет правильным, если эти цифры отличаются, и т. д. Если все цифры одинаковы, то порядок не имеет значения (но элементы расположатся в том же относительном порядке, в котором были представлены в исходном списке).

Каждый элемент помещается в ячейку (за постоянное время) один раз для каждого символа ключа, что дает максимальное время выполнения $O(wn)$.

Часть IV
Методы решения задач

9 А если в лоб?

Computer Science можно рассматривать как способ найти решения задач с помощью компьютеров. Затем работа программиста сводится к трем действиям: определить, какие задачи может решить компьютер, объяснить компьютеру, как их решать, и спрогнозировать, сколько времени займет этот процесс.

Выбор метода решения задачи зависит от того, какие атрибуты решения наиболее важны. У каждого алгоритма есть своя стоимость выполнения, которую можно выразить как сложность по времени и пространству. Сложность по времени — это время выполнения алгоритма в зависимости от объема входных данных. Сложность по пространству — требуемый объем памяти (опять же относительно объема входных данных). *Вычислительная сложность* алгоритма — это количество ресурсов (времени и пространства), необходимых для его выполнения; вычислительная сложность задачи — это минимальная сложность алгоритма, который мог бы ее решить.

Однако ресурсы, необходимые для выполнения программы, — это не единственные расходы (хотя в научных статьях, как правило, упоминают только их). Приходит-

ся также учитывать сложность написания и отладки программы. Иногда лучше реализовать менее эффективный алгоритм, пожертвовав некоторой скоростью в обмен на время программиста (потому что реализация более быстрого алгоритма займет значительно больше времени) или точностью (потому что в более быстром алгоритме выше вероятность программных ошибок).

Практическое применение

Однажды я проводил обзор кода (code review) джуниора. Рецензент предложил внести некое изменение, чтобы сделать код более эффективным.

Я согласился, что предложенный код будет более эффективным, но наложил вето на изменение, потому что из-за этого код стал бы более сложным. В том случае код не относился к чувствительной ко времени части приложения (долю секунды, которую могло бы сэкономить изменение, никто бы не заметил), но дополнительное усложнение повысило бы вероятность того, что в итоге мы бы получили ошибку — либо сразу, либо в будущем, при очередном обновлении кода.

В конце шкалы «эффективность — сложность» находится метод решения задач в лоб, или метод грубой силы. Он очень прост: перебрать все возможные решения и проверить каждое, пока одно из них не окажется правильным. Этот метод годится, если задачу нужно решить один раз, и дополнительное время, необходимое для поиска эффективного решения, превысит экономию времени при выполнении этого решения. Конечно, иногда мы применяем грубую силу просто потому, что не знаем лучшего решения!

Например, если на сейфе установлен пульт из десяти кнопок и вы знаете, что комбинация представляет собой четырехзначное число и больше никакой информации нет (и количество попыток не ограничено), можно найти комбинацию, просто перебрав все возможные числа от 0000 до 9999. Это может занять очень много времени — в среднем 5000 попыток, — но сама процедура очень проста. В цифровом мире это означает, что, если на сайте не используются надежные пароли, а учетная запись не блокируется после некоторого количества неправильных попыток, то с помощью известного имени пользователя можно перебрать все возможные пароли, пока один из них не подойдет.

Решение методом грубой силы, как правило, очень просто реализовать. В то же время его очень сложно реализовать неправильно, что делает его полезным, если размер задачи невелик, а точность крайне желательна.

10 Динамическое программирование

10.1. Задача недостающих полей

Предположим, у нас есть шахматная доска размером $n \times n$, на которой недостает нескольких полей. Как найти наибольший участок доски размером $k \times k$ без недостающих полей?[1]

Если вы раньше не сталкивались с такой задачей, потратьте несколько минут на то, чтобы написать решение и определить время выполнения вашего алгоритма.

Столкнувшись с этой задачей, я рассуждал следующим образом. Каждое поле шахматной доски может принадлежать ко многим наибольшим участкам, но только в одном из них оно может быть верхним левым углом. Если я помечу каждое поле размером наибольшего

---

[1] Мне задали эту задачу несколько лет назад на собеседовании в известной компании по разработке программного обеспечения; быстро предложив эффективное решение, я вышел на следующий уровень собеседований. С тех пор прошло достаточно времени, чтобы я мог спокойно изложить ее здесь.

неповрежденного участка, верхним левым углом которого он является, то поле с наибольшей такой меткой будет соответствовать искомому участку.

Предположим, что доска представлена в виде матрицы $n \times n$, в которой каждая ячейка содержит 1, если соответствующее поле присутствует, и 0, если оно пропущено. Если текущее значение ячейки равно 0, то соответствующее поле отсутствует и не может быть частью непрерывного участка, поэтому его не нужно изменять. Если же значение равно 1, то мы можем заменить его числом, которое на единицу больше, чем минимальное значение из трех ячеек, расположенных ниже и справа.

Мы изменяем каждую ячейку матрицы один раз, включая проверку, равно ли значение ячейки нулю, проверку значений еще максимум трех ячеек и запись нового значения ячейки. Каждая из этих операций занимает время $O(1)$, поэтому время, необходимое для обработки всей шахматной доски, составляет $O(n^2)$.

Обратите внимание, что это линейное, а не квадратичное время выполнения алгоритма — на шахматной доске n[1] полей (некоторые из них отсутствуют), поэтому общее время, затрачиваемое алгоритмом, пропорционально количеству полей. Если более точно обозначить размер шахматной доски как $\sqrt{n} \times \sqrt{n}$, то получим n полей и общее время выполнения $O(n)$.

---

[1] Мы считаем, что n — это общее количество полей, и придерживаемся обычного соглашения о том, что n — это объем входных данных.

10.2. Работа с перекрывающимися подзадачами

Подход, использованный в этом разделе, называется динамическим программированием. Он применяется, когда задачу можно разделить на несколько подзадач, решение каждой из которых будет использовано несколько раз. Этот подход отличается от принципа «разделяй и властвуй», когда задачу разделяют на подзадачи, которые решаются независимо друг от друга. В задаче с шахматной доской каждая подзадача зависела от решений трех других задач, а решения всех подзадач сохранялись для дальнейшего использования.

Динамическое программирование обычно выполняется путем построения таблиц, как показано выше. Это означает решение задачи методом «снизу вверх», когда мы начинаем с решения наименьших подзадач и продвигаемся вверх до тех пор, пока не придем к ответу (алгоритм 8). Другой метод — это мемоизация, при которой мы идем сверху вниз, решая подзадачи по мере необходимости и кэшируя результаты для повторного использования[1].

Построение таблиц — предпочтительный вариант, когда нужно решить каждую подзадачу (в моем примере с шахматной доской нужно было найти наибольший

---

[1] Некоторые считают, что динамическое программирование ограничивается построением таблиц, а мемоизация к нему не относится. Какую бы семантику вы ни предпочли, сами методы остаются такими, как описано здесь.

неповрежденный участок для каждого поля доски), поскольку затраты у этого метода меньше, чем при мемоизации. Если некоторые подзадачи из области решения не обязательно решать, то мемоизация позволяет решать только те подзадачи, которые действительно необходимы.

Алгоритм 8. В каждой ячейке может быть записано число больше 1, только если начальные значения этой ячейки, а также ячеек справа, снизу и снизу справа от нее равны 1

Входные данные: матрица M, каждая ячейка которой содержит либо 1, если соответствующее поле присутствует на доске, либо 0, если его нет

Выходные данные: матрица M, в каждой ячейке которой записан размер самого большого неповрежденного участка доски, для которого соответствующее поле находится в верхнем левом углу

```
begin
    for i = n – 2 to 0 do
        for j = n – 2 to 0 do
            if M[i][j] == 0 then
                продолжить;
            else
                M[i][j] = 1 + min(M[i + 1][j],
                M[i][j + 1], M[i + 1][j + 1]);
            end
        end
    end
end
```

Ключевой момент

Там, где метод «разделяй и властвуй» подразумевает разделение задачи на несколько *независимых* подзадач, динамическое программирование подразумевает разделение задачи на несколько *перекрывающихся* подзадач. Решение каждой подзадачи кэшируется для последующего повторного использования.

10.3. Динамическое программирование и кратчайшие пути

Рассмотрим задачу поиска кратчайшего пути: для заданного графа со взвешенными ребрами нужно найти такой путь из одного узла в другой, который имеет наименьшую стоимость.

Определение

Граф со взвешенными ребрами — это граф, в котором каждое ребро имеет свой вес (стоимость). Стоимость пути из одного узла в другой определяется суммой стоимости всех пройденных ребер.

Предположим, что мы нашли путь между узлами s и t, который содержит третий узел v. Тогда путь из s в t должен содержать кратчайший путь из s в v, поскольку в противном случае мы могли бы заменить этот участок более коротким путем и уменьшить длину кратчайшего пути из s в t, что противоречит начальному условию[1].

[1] Это принцип оптимальности Беллмана.

Ключевой момент

Динамическое программирование (и жадные алгоритмы) полезно для решения задач, имеющих оптимальную подструктуру. Это означает, что оптимальное решение задачи может быть эффективно построено из оптимальных решений ее подзадач. Если самый короткий путь из Мэдисона в Денвер проходит через Омаху, то этот маршрут также должен содержать самый короткий путь из Мэдисона в Омаху и из Омахи в Денвер.

Если задача имеет и оптимальную подструктуру, и перекрывающиеся подзадачи, то она становится кандидатом на решение методом динамического программирования.

Задачи поиска кратчайшего пути представляют собой яркие примеры динамического программирования, поскольку оптимальное свойство подструктуры интуитивно понятно — очевидно, что самый быстрый способ перехода из точки *A* в точку *C* через точку *B* — это также самый быстрый способ перехода из точки *A* в точку *B* и из точки *B* в точку *C*. В число алгоритмов, основанных на этом принципе, входит метод Беллмана — Форда, который находит кратчайший путь из исходной точки в любую вершину графа (или от любой вершины графа до конечной точки) и метод Флойда — Уоршелла — с его помощью вычисляется кратчайший путь между каждой парой вершин графа (алгоритм 10). В обоих случаях идея состоит в том, чтобы начать с небольшого подмножества узлов, близких к интересующим нас узлам, и постепенно расширять это множество, используя уже найденные узлы для вычисления новых расстояний.

Алгоритм 9. В начале алгоритма в ячейке M[s][t] хранится длина ребра между вершинами s и t, если оно существует. Если сумма расстояний от s до другой вершины i и от i до t меньше расстояния от s до t, то мы заменяем M[s][t] новым значением

Алгоритм: Флойда — Уоршелла
Входные данные: матрица M, в каждой ячейке которой записана длина ребра между соответствующими вершинами, с нулями по диагонали и ∞, если такого ребра не существует
Выходные данные: матрица M, в каждой ячейке которой записана длина кратчайшего пути между соответствующими вершинами

```
begin
    for i = 1 to n do
        foreach s, t do
            M[s][t] = min(M[s][t], (M[s][i] + M[i][t]);
        end
    end
end
```

10.4. Примеры практического применения

10.4.1. Git merge

Еще одна задача, на примере которой обычно демонстрируют возможности динамического программирования, — поиск наибольшей общей подпоследовательности (Longest Common Subsequence). Задача состоит в том, чтобы для двух заданных строк *A* и *B* найти самую

длинную последовательность, которая встречается в обеих строках с сохранением последовательности символов. Символы в строках не обязательно должны стоять подряд; например, если $A = \{acdbef\}$ и $B = \{babdef\}$, то $\{adef\}$ будет их общей подпоследовательностью.

При слиянии изменений в Git (merge) выполняется поиск наибольшей общей подпоследовательности для master и рабочей веток. Символы, присутствующие в master, но отсутствующие в наибольшей общей подпоследовательности, удаляются; символы, которые есть в рабочей ветке, но отсутствуют в этой подпоследовательности, добавляются в master.

10.4.2. LaTeX

Систему подготовки документов LaTeX часто используют для создания технических документов. Одна из задач системы набора текста — выравнивание текста одновременно по правому и левому краю; для этого интервалы между словами и символами растягиваются или сжимаются таким образом, чтобы все строки имели одинаковую длину. Другой способ выровнять текст состоит в переносе последнего слова, так что часть слова оказывается на следующей строке. LaTeX [1] пытается найти оптимальные точки разрыва строки, чтобы текст выглядел красиво. Если это сделать не удастся, то несколько строк подряд будут заканчиваться переносом слова или же расстояние между словами в разных

[1] С технической точки зрения практически всю работу выполняет система ввода текста TeX; LaTeX построена на основе TeX. Здесь я использую LaTeX из соображений простоты.

строках будет различаться. В LaTeX существует набор правил для оценки «неудачности» выравнивания. Программа стремится найти «наименее плохой» вариант.

Если в абзаце есть n возможных точек разрыва строки, то существует 2^n возможных вариантов разбиения текста. «Неудачность» выбора для каждой точки разрыва зависит от того, какие точки разрыва были выбраны до нее. Следовательно, у нас снова есть перекрывающиеся подзадачи. Использование методов динамического программирования сокращает время выполнения до $O(n^2)$, которое может быть улучшено с помощью дополнительных методов[1].

[1] Подробнее см. статью: *Donald E. Knuth, Michael F.* Plass Breaking paragraphs into lines // Software: Practice and Experience. Vol. 11. Issue 11, 1981.

11 Жадные алгоритмы

Жадный подход к решению задачи состоит в том, чтобы каждый раз, когда нужно принять решение, выбирать тот вариант, который является локально оптимальным. Другими словами, выбирается тот вариант, который дает лучшее решение текущей подзадачи, даже если это не будет лучшим решением всей задачи.

В задаче о коммивояжере из раздела 1.6 это означало, что нужно всегда ехать в ближайший город, который еще не посещался (то есть выбрать ребро с наименьшим весом, которое приведет к еще не посещавшейся вершине). Такой метод работает быстро (нужно только перебрать все ребра, выходящие из текущей вершины, и выбрать то, которое имеет наименьший вес), но не гарантирует, что будет найдено наилучшее решение всей задачи.

Допустим, мы ходим по некоторому участку земли и хотим найти самую высокую точку. Жадный алгоритм предложил бы постоянно идти в гору, пока это возможно. Когда все допустимые пути будут направлены вниз, это будет означать, что мы нашли локальный

максимум — любая точка, в которую мы сможем перейти непосредственно из текущего положения, окажется ниже того места, в котором мы находимся в данный момент. Это не означает, что мы действительно нашли самое высокое доступное место, — мы достигли лишь самой высокой точки в локальной области (места, где вы сейчас находитесь, и соседних мест, с которыми вы его сравниваете).

Существуют задачи, для которых жадные алгоритмы дают оптимальное решение; одним из примеров является алгоритм поиска кратчайших путей Дейкстры. Как и в случае динамического программирования, жадные алгоритмы лучше всего работают в тех случаях, когда задача имеет оптимальную подструктуру.

Разница между этими подходами заключается в том, что динамическое программирование позволяет гарантированно найти решение задачи для оптимальной подструктуры, поскольку оно учитывает все возможные подзадачи и объединяет их для достижения оптимального решения, тогда как жадный алгоритм просто выбирает ту подзадачу, которая лучше всего выглядит в данный момент.

Рассмотрим такую задачу: добраться домой с работы в час пик. Жадный подход заключается в том, чтобы выбрать любой маршрут домой с наименьшей загруженностью возле работы. В этом случае вы будете двигаться быстрее, но рискуете столкнуться с большим трафиком ближе к дому. Алгоритм динамического программирования будет учитывать весь трафик и выберет маршрут с наименьшими затратами

в среднем, даже если начальный участок этого пути будет сильнее загружен.

Жадные алгоритмы, как правило, являются быстрыми. Поэтому их предпочтительно выбирать, когда они гарантированно найдут либо оптимальное решение задачи, либо хотя бы одно достаточно хорошее решение.

Часть V
Теория сложности вычислений

12 Что такое теория сложности

В главе 1 вы узнали, как оценить время выполнения алгоритма, чтобы можно было выбрать наилучший алгоритм для решения данной задачи (или определить, существует ли достаточно хороший алгоритм для ее решения). В той главе не имело особого значения, какой язык программирования использовать. Там описаны алгоритмы на псевдокоде, однако использование C#, Java или Python дало бы такую же асимптотическую скорость выполнения.

Это не означает, что программы, написанные на любом языке, всегда будут одинаково эффективны, однако все они (теоретически) работают на одном и том же оборудовании и используют одни и те же структуры данных, поэтому мы ожидаем увидеть одинаковое увеличение времени выполнения по мере роста объема задачи. Основное предположение состоит в том, что существует некий теоретический компьютер, во всем подобный тем, что применяются в настоящее время, за исключением бесконечного объема памяти (что, увы, редко встречается в реальных компьютерах).

Одно из применений теории сложности состоит в определении того, что может и чего не может сделать компьютер. Иногда удается значительно сократить время, необходимое для решения задачи, либо используя более эффективный алгоритм, либо вводя дополнительные ресурсы (например, процессоры). Более сложные по своей природе задачи невозможно решить без значительных ресурсов даже при наличии оптимального алгоритма (NP-полные задачи считаются сложными). Иногда задача действительно не решается на определенном типе компьютеров, даже при наличии неограниченного количества ресурсов.

При смене вычислительной модели задача, которая прежде была очень сложной, может стать тривиальной, а тривиальная задача, наоборот, — неразрешимой. Например, существуют квантовые алгоритмы (алгоритмы, работающие на квантовом компьютере), которые выполняются экспоненциально быстрее, чем самые известные классические алгоритмы (те, что выполняются на компьютерах, используемых сегодня повсеместно, не только в исследовательских лабораториях), для одной и той же задачи.

Дополнительная информация

Квантовый компьютер — это сравнительно новая модель вычислений, основанная на квантовой механике. Там, где в классических компьютерах используются биты, которые могут быть равны 0 или 1, в квантовых компьютерах применяются *кубиты*. Последние существуют

в суперпозиции состояний: каждый бит равен 0 и 1 одновременно. Если байт выражает одно значение от 0 до 255, то квантовый байт выражает все 256 значений одновременно.

Если прочитать кубиты, то они свернутся в одно состояние. Другими словами, вместо того, чтобы находиться во всех возможных состояниях сразу, они имеют ровно одно значение. При этом нет гарантии, что это значение будет правильным; алгоритм просто манипулирует кубитами так, что при их измерении они с большой вероятностью придут в правильное состояние.

И наоборот, существуют модели вычислений, неспособные решить задачи, которые тривиально решаются с помощью современных компьютеров. Эти модели имеют такие ограничения, как способ доступа к памяти. Продемонстрировав, что конкретная задача может быть решена с помощью данной модели, мы устанавливаем верхнюю границу вычислительной мощности, необходимой для ее решения, а также получаем доступ к инструментам, созданным для работы с этой моделью.

13 Языки и конечные автоматы

13.1. Формальные языки

Любой человеческий язык, например английский, представляет собой набор букв (в письменной речи) или звуков (в устной речи), а также правил, описывающих, как комбинировать эти буквы или звуки в слова и предложения. Точно так же в математике и Computer Science язык — это набор символов и правила их комбинации.

Языки классифицируются в зависимости от того, насколько мощным должен быть компьютер для их распознавания. Под распознаванием в данном случае понимается способность компьютера точно определить по заданной строке, принадлежит ли она некоторому языку. Такие свойства, как тип доступной памяти, определяют мощность компьютера и, следовательно, сложность языков, которые он способен распознавать.

Формально язык L с алфавитом Σ — это (возможно, бесконечное) множество всех допустимых слов, которые могут быть составлены из символов этого алфавита. Например, мы можем определить алфавит $\Sigma = \{a\}$ для языка $L = a^{2n}$, другими словами, все строки четной

длины, состоящие только из буквы *a*. В этот язык будут входить строки λ (пустая строка — ноль, а ноль — это четное число!), *aa*, *aaaa* и т. д. В языке может быть конечное количество слов, бесконечное количество слов или даже не быть слов вообще ($L = \{\varnothing\}$[1]).

Класс языков может определяться по типу машины, которая способна его распознать, по типу грамматики, генерирующей этот язык, а также в терминах теории множеств. Рассмотрим все способы для каждого класса языков.

13.2. Регулярные языки

Регулярные языки — это такие языки, которые могут быть поняты *конечными автоматами*. Конечный автомат (Finite State Machine), — это машина, которая имеет одно начальное состояние, одно или несколько допустимых состояний и переходы между этими состояниями. Чтобы определить, есть ли в языке данная строка, вы начинаете со стартового состояния и следуете переходу для каждой буквы этой строки. Если состояние, достигнутое после последней буквы строки, является допустимым состоянием, то строка принадлежит этому языку.

В конечном автомате, представленном на рис. 13.1, работа начинается с состояния *S*, которое также является допустимым (поскольку нам еще не встретилось никаких букв, а ноль — четное число). Встретив одну букву *a*, мы переходим в состояние q_1, а встретив вторую букву *a* — возвращаемся к *S*. И так далее: продолжаем переходить по ребрам, пока не прочитаем всю строку.

---

[1] Это пустое множество — не то же самое, что множество, содержащее только пустую строку ($\{\lambda\}$).

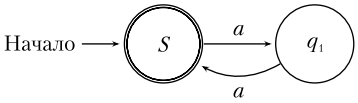


Рис. 13.1. Конечный автомат, который принимает любую строку четной длины, составленную из алфавита $\Sigma = \{a\}$. Двойной круг указывает на то, что S — допустимое состояние

Существует два вида конечных автоматов: детерминированные и недетерминированные. В *детерминированных конечных автоматах*, ДКА (Deterministic Finite Automata), если алфавит содержит более одной буквы (как в большинстве языков), каждое состояние автомата должно содержать ребро для каждой буквы алфавита. На рис. 13.2 показан детерминированный конечный автомат для языка, который построен на алфавите $\Sigma = \{a, b\}$ и состоит из любого количества букв a, после которых стоит ровно одна буква b.

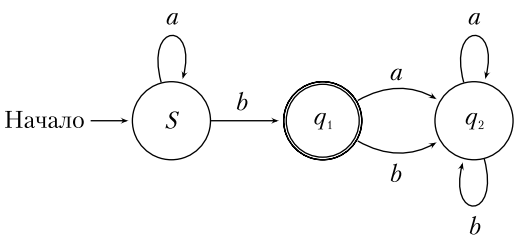


Рис. 13.2. Детерминированный конечный автомат, принимающий строки, состоящие из любого (включая нулевое) числа букв a, после которых стоит одна буква b

Состояние q_2 на рис. 13.2 — то, что называют мертвым состоянием: это состояние не является допустимым и, поскольку, какую бы букву мы ни добавили, мы все равно вернемся в эту же точку, если мы сюда попали, то

уже не сможем достичь какого-либо допустимого состояния. В данном случае строка, которая содержит любые символы после буквы *b*, отсутствует в данном языке. Обратите внимание, что, поскольку автомат является детерминированным, переход на каждом шаге однозначно определяется вводимым символом и текущим состоянием.

Альтернативой мертвому состоянию является использование *недетерминированного конечного автомата*, НКА (Nondeterministic Finite Automaton). Это конечный автомат, в котором у каждого состояния для любой буквы может быть ноль, один или несколько переходов. Поскольку в недетерминированном конечном автомате каждое состояние не должно иметь переход для каждой буквы, мертвое состояние не является обязательным. Если для следующей буквы в строке нет перехода, то данная строка отсутствует в этом языке.

С точки зрения описываемых языков автоматы НКА эквивалентны ДКА. Каждый ДКА также является НКА (ДКА должен соблюдать ограничение по одному переходу на одну букву, в отличие от НКА), а любой НКА может быть преобразован в соответствующий ДКА. На рис. 13.3 представлен НКА, который описывает тот же язык, что и ДКА на рис. 13.2. Как правило, предпочтительно использовать НКА вместо ДКА, поскольку НКА часто требуют гораздо меньше переходов.

Предположим, мы хотим распознать более сложный язык: он состоит из одной или нескольких букв *a*, после которых стоит ноль или более букв *c*, ноль или более букв *b*, а затем ровно одна буква *c*. Мы могли бы распознать этот язык с помощью недетерминированного

конечного автомата на рис. 13.4. Обратите внимание: в этом автомате используется лямбда-переход[1], чтобы гарантировать, что *c*-цикл не будет повторяться после *b*-цикла.

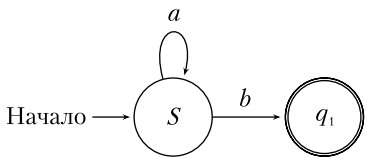


Рис. 13.3. Недетерминированный конечный автомат, который принимает строки, состоящие из любого (включая нулевое) количества букв *a*, после которых стоит ровно одна буква *b*

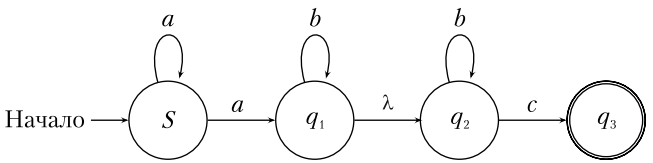


Рис. 13.4. НКА для языка *a* + *c* * *b* * *c*

Какие типы языков не распознаются конечными автоматами? Обратите внимание, что у этих конечных автоматов нет отдельной памяти, что ограничивает длину сравнений числом состояний автомата. Например, мы хотим построить автомат, который бы распознавал язык, состоящий из любого количества парных скобок, причем после левой скобки всегда следует соответствующая правая скобка. Такой автомат мог бы выглядеть так, как показано на рис. 13.5.

---

[1] Лямбда-переход позволяет переходить из одного состояния в другое, не считывая входные данные.

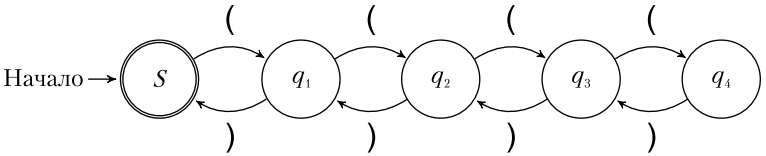


Рис. 13.5. Недетерминированный конечный автомат, который принимает наборы парных скобок и поддерживает до четырех одновременно открытых пар

Такой автомат будет делать именно то, что мы хотим, если число одновременно открытых пар скобок не превысит четырех (четыре левые скобки без соответствующих правых скобок). Если этот предел будет превышен, автомат не сможет распознать строку. Мы можем увеличить автомат, добавив еще несколько состояний, но число состояний всегда будет конечным. В случае $n + 1$ состояний можно отслеживать до n переходов, то есть конечный автомат не сможет распознать любую строку языка, в котором допускаются сравнения любой длины. Позже вы узнаете о методе, позволяющем показать, что язык, который требует такого сравнения, не может быть распознан никаким конечным автоматом, следовательно, такой язык не является регулярным.

13.2.1. Регулярные грамматики

Грамматика — это набор правил для генерации языка. Там, где автоматы для языка позволяют проверить, принадлежит ли данная строка языку, грамматика позволяет генерировать любую строку, которая является частью языка. Грамматика состоит из переменных (обозначаемых заглавными буквами), терминалов (которые явля-

ются символами алфавита) и переходов, преобразующих переменные в символы (которые могут быть переменными или терминалами). Например, грамматика, показанная на рис. 13.6, генерирует язык, представленный на рис. 13.5 и состоящий из наборов скобок, в которых одновременно может быть открыто до четырех пар.

$$
\begin{aligned}
S &\to (A \\
S &\to \lambda \\
A &\to (B \\
A &\to)S \\
B &\to (C \\
B &\to)A \\
C &\to (D \\
C &\to)B \\
D &\to)C
\end{aligned}
$$

Рис. 13.6. Грамматика, генерирующая язык парных скобок, в котором не допускается более четырех открытых пар одновременно

Мы будем использовать грамматику, в которой построение строки начинается с S и затем на каждом шаге переменная заменяется до тех пор, пока получившаяся строка полностью не будет состоять из терминалов. Например: $S \to (A \to ((B \to (() A \to (() (B \to (() () A \to$ $\to (() ()) S \to (() ())$.

Такая грамматика называется праволинейной, потому что при каждом преобразовании после переменной (если она есть) стоит терминальный символ. Каждое порождающее правило в праволинейной регулярной грамматике имеет одну из трех форм: переменная присоединяется к терминалу, переменная ставится после

терминала или λ. Леволинейная регулярная грамматика идентична праволинейной, только переменная присоединяется к терминалу, терминал ставится после переменной или λ. Обратите внимание, что эти два вида грамматик не смешиваются; для представления любого регулярного языка можно использовать только леволинейную регулярную или праволинейную регулярную грамматику, но не обе одновременно.

Подводные камни на практике

Казалось бы, еще один способ работы с регулярными языками — это регулярные выражения (regex). Действительно, с точки зрения Computer Science это так. Регулярные выражения описывают регулярные языки и обладают теми же возможностями описания, что и регулярные грамматики.

В программировании регулярные выражения расширены и поддерживаются многими языками, которые на самом деле не являются регулярными, поскольку включают в себя запоминание введенных ранее данных (обратные ссылки). Благодаря этому regex в программировании гораздо мощнее регулярных выражений в Computer Science.

Зато теоретические регулярные выражения по крайней мере не подвержены катастрофическим возвратам.

13.2.2. Свойства замыкания

Все слова языка образуют множество[1]. Множества могут быть замкнуты относительно определенных операций.

[1] Напомню, что множество — неупорядоченная коллекция элементов.

Это означает, что, если применить эту операцию к членам множества, мы получим другой элемент данного множества. Например, целые числа замкнуты относительно сложения, вычитания и умножения, потому что сложение, вычитание и умножение двух целых чисел дает еще одно целое число. Целые числа не замкнуты относительно деления, так как результат деления одного целого числа на другое не обязательно будет целым числом.

Аналогично класс языков является замкнутым относительно операции $\circ$, если для любых языков из этого класса результатом применения операции является язык этого же класса. Регулярные языки замкнуты относительно таких операций, как:

- **объединение** $A \cup B$: каждое слово, которое входит хотя бы в один из языков A или B;
- **пересечение** $A \cap B$: каждое слово, которое входит и в A, и в B;
- **конкатенация** A с B: множество слов, каждое из которых представляет собой любую строку из A, после которой идет любая строка из B;
- **дополнение** $A-$: все слова, составленные из того же алфавита Σ, что и A, но не принадлежащие A;
- **разность** $A - B$: все слова, которые есть в A, но которых нет в B;
- **звезда Клини** A^*: ноль копий A и более;
- **плюс Клини** A^+: одна копия A и более.

Следующие свойства либо являются очевидными, либо логически следуют из определения регулярных языков в терминах теории множеств:

- пустой язык $L = \{\varnothing\}$ является регулярным;
- язык, содержащий только пустую строку $L = \lambda$, является регулярным;
- для каждой буквы в алфавите Σ язык, содержащий только эту букву, является регулярным;
- если A и B являются регулярными языками, то $A \cup B$, конкатенация A и B, а также A^* являются регулярными.

Пример

Конкатенация. Если A и B являются регулярными языками, то добавление перехода λ из каждого допустимого состояния НКА для A в начальное состояние НКА для B[1] дает НКА для конкатенации A с B.

Дополнение. Если заменить любое допустимое состояние ДКА для языка L на недопустимое состояние (и наоборот), то получим ДКА для дополнения L.

Пересечение. Пересечение множеств A и B (элементы, присутствующие в обоих множествах) — это те же самые элементы, которых нет в множестве элементов, отсутствующих в A, или элементов, отсутствующих в B. Другими словами, это $\overline{\overline{A} \cup \overline{B}}$. Таким образом, зная, что регулярные языки замкнуты относительно объединения, из дополнения следует, что они также замкнуты относительно пересечения.

Ни один язык с алфавитом Σ, который не может быть построен по этим правилам, не является регулярным.

Чтобы доказать, что регулярный язык замкнут относительно определенной операции, нужно либо показать,

что можно создать конечный автомат (или регулярное выражение), представляющий операцию замыкания, либо показать, что эта операция является объединением уже проверенных операций замыкания.

13.2.3. Лемма о накачке

Чтобы показать, что язык является регулярным, я использую либо конечные автоматы, либо регулярную грамматику, которая описывает этот язык. Как показать, что язык не является регулярным? Неспособность найти подходящий автомат или грамматику не является доказательством того, что, возможно, она существует, просто я ее не нашел.

Вместо того чтобы доказывать, что язык не является регулярным, мы используем то, что называется леммой о накачке. Она не доказывает, что язык является регулярным, а лишь показывает, что это не так. Это способ доказательства от противного — мы предполагаем, что язык является регулярным, и показываем, что любой автомат, который допустим для этого языка, также будет считать допустимыми строки, которых нет в данном языке.

Рассмотрим язык $L = \{a^n b^n\}$ (то есть любое количество букв a, после которого следует такое же количество букв b). Предположим, что этот язык — регулярный; тогда существует конечный автомат, для которого этот язык является допустимым. Данный автомат имеет конечное число состояний, обозначим его k. Далее, если этот автомат считает допустимой строку длиной k

[1] Следовательно, эти состояния в A больше не являются допустимыми.

и более символов, то одно и то же состояние должно посещаться более одного раза (в автомате есть цикл).

Рассмотрим строку $a^k b^k$, которая заведомо есть в языке. Поскольку для нашего автомата этот язык допустимый, то и строка $a^k b^k$ является допустимой для этого автомата. Однако, поскольку число букв a в строке совпадает с количеством состояний, в автомате должен существовать цикл размером j[1], который необходимо выполнить, чтобы распознать эту строку. При чтении букв a мы проходим этот цикл еще один раз, после чего попадаем в допустимое состояние для строки $a^{k+j} b^k$, которой в языке заведомо нет. Это противоречит предположению о том, что для данного языка существует конечный автомат, и показывает, что этот язык не регулярный.

13.3. Контекстно свободные языки

Контекстно свободные языки — это надмножество регулярных языков: любой регулярный язык также является контекстно свободным языком. Оба типа языков входят в состав иерархии грамматик Хомского (рис. 13.7), в которой каждая грамматика является собственным надмножеством менее мощных грамматик иерархии.

Подобно регулярным языкам, контекстно свободные языки могут быть описаны в терминах автоматов, которые могут их распознавать, грамматик, которые могут их создавать, и свойств замыкания множества. Контекстно свободные языки более мощные, чем регулярные языки, поскольку в них допускается использование памяти,

---

[1] $|j| > 0$.

в частности стека (сколь угодно большого размера). Интуитивно понятно, что контекстно свободными языками являются те, в которых нужно помнить не более одного элемента за раз, — так (как вскоре будет доказано), язык $a^n b^n$ будет контекстно свободным, а $a^n b^n c^n$ — нет.

Рис. 13.7. Иерархия Хомского. Регулярные языки являются собственным подмножеством контекстно свободных языков, которые, в свою очередь, являются собственным подмножеством контекстно зависимых языков, а те — собственным подмножеством рекурсивно перечислимых языков

13.3.1. Магазинные автоматы

Магазинные (стековые) автоматы (МП-автоматы) (PushDown Automata) очень похожи на конечные автоматы, с двумя исключениями. Чтобы определить, какой переход выполнить, такие автоматы, кроме чтения очередного элемента входных данных, могут использовать верхний элемент стека. Автоматы также могут управлять стеком в процессе перехода.

Подобно конечным автоматам, магазинные состоят из конечного множества состояний (одно или несколько из которых могут быть допустимыми) и переходов между ними. Однако, кроме чтения буквы из входных данных, в процессе этих переходов автомат может извлекать переменную из стека и помещать одну или несколько переменных в стек. В дополнение к алфавиту терминалов, здесь есть множество символов стека (в которое иногда входит специальный символ, обычно *Z*, обозначающий дно стека).

Магазинный автомат может проверить строку на допустимость двумя способами. Первый способ — дойти до заключительного состояния, как в случае конечных автоматов: МП-автомат считает строку допустимой, если после чтения строки автомат находится в допустимом состоянии. Кроме того, МП-автомат также считает строку допустимой в случае пустого стека. Одни и те же языки можно выразить, используя любой метод доказательства допустимости. Но, как правило, удобно предположить, что МП-автомат должен находиться в допустимом состоянии, а также иметь пустой стек; именно этому соглашению мы и будем следовать.

Рассмотрим автомат, показанный на рис. 13.8.

В первом состоянии *S* мы можем прочитать любое количество букв *a* (включая нулевое). Для каждой прочитанной буквы *a* мы ничего не извлекаем из стека, но помещаем туда *A*. Закончив чтение букв *a*, мы выполняем λ-переход (при котором ничего не читаем, не помещаем в стек и не извлекаем оттуда) в состояние q_1. В состоянии q_1 мы считываем столько букв *b*, сколько это возможно, с учетом

того, что при каждом чтении буквы b мы извлекаем из стека A. Закончив чтение входных данных, мы выполняем последний λ-переход в допустимое состояние, извлекая в процессе перехода символ дна стека.

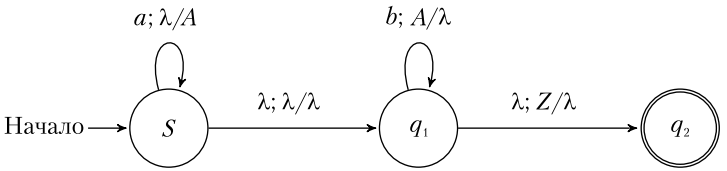


Рис. 13.8. Недетерминированный магазинный автомат, для которого является допустимым язык $L = \{a^n b^n\}$. Предполагается, что в начале стек содержит Z

Как и в случае конечных автоматов, МП-автоматы могут быть детерминированными или недетерминированными. МП-автомат является недетерминированным, если для данной ситуации (текущего состояния, следующего считываемого символа и переменной вверху стека) существует несколько возможных переходов. Детерминированный магазинный автомат имеет не более одного возможного перехода из каждого состояния, после которого можно читать следующий символ из входных данных и верхнего символа стека.

В отличие от конечных автоматов детерминированные и недетерминированные магазинные автоматы не являются взаимозаменяемыми. Недетерминированные МП-автоматы могут реализовать любой контекстно свободный язык, в то время как детерминированные МП-автоматы реализуют собственное подмножество детерминированных контекстно свободных языков.

13.3.2. Контекстно свободная грамматика

Контекстно свободная грамматика, которая генерирует контекстно свободный язык, — это грамматика, левая часть которой всегда представляет собой одну переменную, а правая часть может быть любым числом переменных и терминалов. Это надмножество регулярных грамматик, для которых левая часть также должна представлять собой единственную переменную, но правая часть всегда является только одним терминалом, после которого следует ноль или одна переменная (или перед ним стоит леволинейная регулярная грамматика). Интуитивно понятно, что контекстно свободными грамматиками являются такие, в которых (поскольку левая часть в правиле контекстно свободной грамматики всегда является единственной переменной) переменная всегда может быть преобразована в любую из возможных замен, независимо от чего-либо (от контекста), что может стоять до или после нее (рис. 13.9).

$$S \rightarrow aSb$$
$$S \rightarrow \lambda$$

Рис. 13.9. Контекстно свободная грамматика, которая генерирует язык $a^n b^n$

13.3.3. Лемма о накачке

Как и в случае с регулярными языками, можно доказать, что язык не является контекстно свободным. Для этого нужно выбрать строку, принадлежащую языку, и показать, что любой магазинный автомат, который генерирует эту строку, также должен генерировать строки,

отсутствующие в данном языке. Лемма о накачке для контекстно свободных языков гласит, что если язык L контекстно свободный, то любая строка s из L, имеющая длину p или более символов, может быть записана как $s = uvwxy$, так что выполняются следующие условия:

- uv^nwx^ny принадлежит к языку L для всех $n \geq 0$.

  Это условие говорит о том, что, если v и x повторяются одинаковое число раз (включая ноль), мы все равно получим строку, которая принадлежит языку. Чтобы отследить, сколько раз мы выполняем первый цикл, и повторить то же количество итераций во втором цикле, можно использовать стек;

- $|vwx| \leq p$.

  Отсюда следует, что количество символов в цикле, включая неповторяющуюся часть (x), не больше p;

- $|vx| \geq 1$.

  Данное условие показывает, что повторяющаяся часть цикла содержит хотя бы один символ.

Предположим, что язык L является контекстно свободным; тогда существует автомат с p состояниями, для которого этот язык будет допустимым. Если строка языка имеет длину не менее p, то путь, который проходит строка, должен содержать цикл длиной не менее единицы. Если выполнить этот цикл еще один раз или не выполнять его вообще, то мы должны закончить в том же конечном состоянии. Следовательно, есть другая строка, которая также должна принадлежать этому языку. Если условие не соблюдается, то это противоречит нашему первоначальному предположению

о существовании магазинного автомата, для которого L — допустимый язык и L не является контекстно свободным языком.

Хитрость использования леммы о накачке для контекстно свободных языков состоит в том, что обычно существует несколько способов разбить строку таким образом, чтобы она соответствовала приведенным выше правилам. Мы должны показать, что любой способ разбиения строки все равно выбрасывает нас за пределы языка. Таким образом, главным условием эффективного использования леммы о накачке является выбор хороших строк, у которых не очень много вариантов разбиения.

Рассмотрим язык $L = \{a^n b^n c^n\}$ и выберем строку $s = a^P b^P c^P$. Если строка целиком состоит из букв a, то в результате накачки мы получим строку, в которой букв a больше, чем b и c, поэтому строка не относится к данному языку. Это верно и для случая, когда строка полностью состоит из букв b или c. Если же начальная строка содержит два символа, то результат будет содержать еще несколько этих символов, после которых будет стоять третий (или меньше, если использовать обратную накачку, то есть вообще не выполнять цикл). Исходная строка не может содержать все три символа, поскольку ее длина не может быть больше p, а между последней a и первой c должно стоять p букв b.

Таким образом, любой способ, которым теоретически можно разбить строку, позволяет превратить ее во что-то, чего нет в языке. Следовательно, такой язык не является контекстно свободным.

13.4. Контекстно зависимые языки

Контекстно зависимые языки — это следующий уровень в иерархии Хомского, и они включают в себя контекстно свободные языки. Если регулярные языки реализуются конечными автоматами, а контекстно свободные языки — магазинными автоматами, то контекстно зависимые языки реализуются *линейно ограниченными автоматами* (Linear Bounded Automata). Такой автомат — это недетерминированная машина Тьюринга (см. главу 14), в которой длина ленты ограничена и зависит от длины входных данных[1].

Контекстно зависимые языки генерируются неукорачивающими грамматиками — это грамматики, которые не содержат каких-либо правил, в которых левая часть длиннее правой. Другими словами, при использовании такой грамматики длина перезаписываемой строки никогда не уменьшается.

$L = \{a^n b^n c^n, n \geq 1\}$ является контекстно зависимым языком, поскольку его можно сгенерировать с помощью следующей грамматики (рис. 13.10).

$$\begin{aligned} S &\rightarrow abc \\ S &\rightarrow aSBc \\ cB &\rightarrow Bc \\ bB &\rightarrow bb \end{aligned}$$

Рис. 13.10. Грамматика, которая генерирует язык $a^n b^n c^n, n \geq 1$

[1] Формально длина ленты является линейной функцией длины входных данных.

13.5. Рекурсивные и рекурсивно перечислимые языки

Рекурсивные языки также реализуются машинами Тьюринга, но они снимают ограничение на длину используемой ленты. Если существует машина Тьюринга, которая в итоге прекращает работу на любом заданном наборе входных данных и правильно признает допустимыми или отклоняет строки некоторого языка, то этот язык является рекурсивным.

Рекурсивно перечислимые языки — это те языки, для которых машина Тьюринга может перечислить все допустимые строки. Другими словами, мы снимаем требование, чтобы машина Тьюринга прекращала работу, если строка не относится к данному языку. Если язык, комплементарный рекурсивно перечислимому языку, также является рекурсивно перечислимым, то такой язык является рекурсивным.

14 Машины Тьюринга

14.1. Чисто теоретический компьютер

Машина Тьюринга — это абстрактная машина, способная моделировать любой алгоритм. Возможно, другие модели вычислений могут работать быстрее, использовать меньше памяти, их легче программировать, но если машина Тьюринга не способна решить какую-то задачу, значит, никакая другая машина также не способна это сделать (насколько мы знаем, это тезис Черча — Тьюринга).

> **Предупреждение**
>
> Говоря о компьютере без каких-либо уточнений, я имею в виду классический (не квантовый) компьютер.

Машина Тьюринга — это конечный автомат, который работает с лентой памяти бесконечной длины, разделенной на отдельные ячейки. У машины есть головка, которая расположена над определенной ячейкой; при запуске алгоритма машина считывает значение этой ячейки. Затем она может выполнить запись в эту ячейку, переместить

головку влево или вправо и перейти в новое состояние. Другой способ представить машину Тьюринга — как магазинный автомат с двумя стеками, в котором один стек представляет часть ленты, расположенную слева от головки, а второй — остальную часть ленты.

Установка различных ограничений на машину Тьюринга может сделать ее эквивалентной другим моделям вычислений. Например, если машина осуществляет только чтение и может перемещать головку только вправо, то она эквивалентна НКА. И наоборот, различные ослабления ограничений на машины Тьюринга — сделать их недетерминированными, добавляя дополнительные ленты и т. д., — не расширяют класс задач, которые может решить машина (хотя это влияет на число операций, необходимых для решения этих задач).

Универсальная машина Тьюринга, способная моделировать другие машины Тьюринга, эквивалентна по мощности (то есть по задачам, которые она способна решать) реальному компьютеру (при условии, что у реального компьютера бесконечная память).

14.2. Построение машины Тьюринга

Машину Тьюринга можно представить так же, как конечные автоматы: в виде последовательности состояний и переходов между этими состояниями (рис. 14.1). Каждый переход включает в себя считываемый символ (если он есть), записываемый символ (если есть) и то, должна ли головка машины передвигаться влево, вправо или остаться в той же ячейке ленты. Строка принадлежит языку, который является допустимым для машины

Тьюринга, если обработка ленты, содержащей эту строку, приведет к остановке машины в допустимом состоянии. Строка не будет допустимой, если не существует корректного перехода, который можно было бы выполнить.

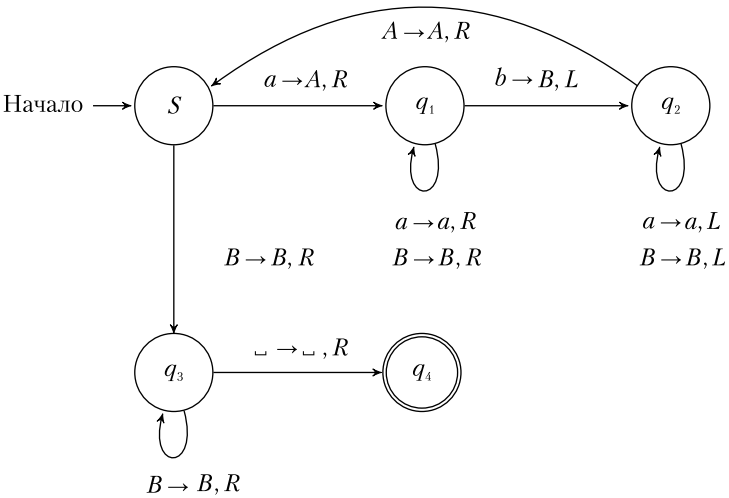


Рис. 14.1. Машина Тьюринга для $a^n b^n$. Обратите внимание, что при последнем переходе читается пустая ячейка. Машина находит буквы a и соответствующие буквы *b*, повторяя так до тех пор, пока в строке не закончатся символы

14.3. Полнота по Тьюрингу

Система правил, такая как язык программирования, называется Тьюринг-полной, если ее можно использовать для моделирования любой машины Тьюринга. Поскольку машина Тьюринга способна решить любую задачу, которая в принципе может быть решена компьютером, это означает, что Тьюринг-полный язык также способен

решить любую задачу, которая может быть решена компьютером с использованием любого языка. Обратите внимание, что здесь ничего не говорится о том, сколько времени потребуется для решения задачи, а лишь то, что машина в итоге придет к решению задачи.

Процедурным языком является Тьюринг-полный язык, если в нем есть условное ветвление и он способен обрабатывать сколь угодно большой объем памяти. Это означает, что (без учета аппаратных ограничений) большинство языков программирования являются Тьюринг-полными.

14.4. Проблема остановки

Если машина Тьюринга способна решить любую задачу, которая в принципе может быть решена компьютером, это означает, что существуют задачи, которые не может решить ни один компьютер.

Одной из таких является проблема остановки: если даны описание произвольной программы и входные данные, как определить, закончит ли когда-нибудь работу эта программа? Задача считается неразрешимой — не существует машины Тьюринга, которая могла бы ответить на этот вопрос для всех возможных входных данных. Машина для языка L всегда останавливается на любой строке этого языка, но может работать вечно, если строка не принадлежит данному языку и при этом язык не рекурсивный.

Как и в случае *NP*-полноты, можно продемонстрировать, что задача неразрешима, показав, что ее решение также позволило бы решить еще одну задачу, относительно которой уже было доказано, что она неразрешимая.

Часть VI
Доказательства

15 Приемлемые доказательства

15.1. Введение в доказательства

Математика замечательна тем, что позволяет выстраивать доказательства, которые никогда не утрачивают своей силы. Если аксиомы[1] верны, как и логическая цепочка, связывающая эти аксиомы с данным результатом, то он всегда является верным.

Математическое доказательство должно демонстрировать, что утверждение верно всегда, то есть не просто во многих случаях, а во всех.

Студенты часто используют такие фразы, как «это очевидно» или «это легко показать». Однако математическое доказательство не допускает пропуска шагов. В учебнике простые шаги могут быть опущены для краткости, но в общем случае доказательство требует выстраивания всей логической цепочки.

---

[1] Аксиома — утверждение, принимаемое истинным без доказательств, из которого логически выводятся дополнительные утверждения. Например, согласно одной из аксиом Евклида, любые две точки могут быть соединены прямой линией.

Пример

Числа-близнецы — это пары простых чисел, отличающихся друг от друга на 2 (например, 5 и 7, 11 и 13, 17 и 19). Согласно гипотезе[1] о простых числах-близнецах существует бесконечное количество простых чисел, отличающихся друг от друга на 2. Это кажется разумным, поскольку мы знаем, что существует бесконечно много простых чисел[2], однако это неочевидно, так как до сих пор нам не удавалось обнаружить никакой закономерности в порядке их появления.

На момент написания этой книги самыми большими числами-близнецами были $2\,996\,863\,034\,895 * 2^{1\,290\,000} +/- 1$, состоящие из 388 342 цифр. Множество чисел-близнецов, обнаруженных до настоящего момента, убедительно свидетельствует о том, что они будут появляться бесконечно, однако это не есть доказательство!

15.2. Терминология

- **Аксиома** — утверждение, которое принимается без доказательств; остальная часть доказательства зависит от правильности аксиом.
- **Математическая гипотеза** (conjecture) — утверждение, которое представляется верным, но для которого еще не получено математическое доказательство.

---

[1] Математическая гипотеза — это утверждение, которое мы считаем верным, но пока не можем доказать.

[2] Вторая теорема Евклида.

Когда математическая гипотеза доказана, она становится теоремой[1].

- **Следствие** — утверждение, которое непосредственно вытекает из другой теоремы или определения. Нередко представляет собой просто частный случай теоремы.
- **Гипотеза** (hypothesis) — предположение, которое считается верным, несмотря на отсутствие доказательств. Иногда гипотеза полагается верной и используется для выстраивания условных доказательств других утверждений.
- **Лемма** — вспомогательная теорема, используемая для доказательства другой теоремы. Если искомое доказательство — это вершина горы Эверест, то лемма — это остановка на пути к ней. Доказательство леммы должно приблизить вас к желаемому пункту назначения[2].
- **Доказательство** (математическое доказательство) — это цепочка логических умозаключений, показывающая, что при условии истинности некоего набора

---

[1] Опровергнутая гипотеза считается просто ложным предположением.

[2] Например, согласно лемме Евклида, если произведение двух целых чисел a и b делится на простое число p, то по крайней мере один из этих сомножителей делится на p. Эта простая лемма используется для доказательства основной теоремы арифметики, согласно которой любое целое число, превышающее 1, либо является простым, либо может быть выражено в виде произведения простых чисел, причем это представление является единственным, если не учитывать порядок следования множителей.

предположений утверждение верно. Часто для обозначения конца доказательства используется аббревиатура QED[1] или символ в виде заполненного квадрата.

- **Теорема** — утверждение, которое было доказано на основе аксиом и/или ранее доказанных теорем. Доказанная математическая гипотеза становится теоремой.

[1] Quod erat demonstrandum — латинская фраза, означающая «что и требовалось доказать».

16 Методы доказательства

16.1. Конструктивное доказательство, доказательство методом исчерпывания вариантов

Эти два метода доказательства часто путают.

Конструктивное доказательство — это доказательство, в котором существование объекта доказывается путем построения его примера.

Если я хочу доказать гипотезу о существовании двух простых чисел, отличающихся друг от друга на 2, то могу сделать это, предоставив пару таких чисел (например, 17 и 19). В данном случае цель — просто продемонстрировать факт существования искомого объекта, а для этого достаточно привести хотя бы один пример.

При использовании метода исчерпывания вариантов мы разбиваем задачу на некое количество случаев и до-

казываем каждый из них отдельно[1]. Лемма о накачке, обсуждаемая в главе 13, доказывается с помощью именно этого метода: мы берем строку, принадлежащую некоему языку, и показываем, что любой автомат, генерирующий эту строку, также будет генерировать строки, которых нет в данном языке.

Пример

Согласно теореме о четырех цветах[2], для правильной раскраски планарного графа требуется не более четырех цветов. Кеннет Аппель (Kenneth Appel) и Вольфганг Хакен (Wolfgang Haken) доказали эту теорему методом исчерпывания вариантов, продемонстрировав, что минимальный контрпример должен содержать одну из 1936 возможных конфигураций, каждая из которых была проверена с помощью компьютера.

16.2. Доказательство от противного[2]

При использовании данного метода доказательства мы предполагаем, что утверждение, которое мы хотим опровергнуть, на самом деле верно, а затем показываем, что это предположение приводит к логическому противоречию, то есть к заведомо ложному утверждению.

---

[1] Этот метод доказательства также называется методом «разбора случаев», поскольку мы разбиваем задачу на конечное число частных случаев (каждый из которых может содержать бесконечное количество примеров) и отдельно доказываем каждый.

[2] См. раздел 4.7.

Пример

Иррациональное число — это действительное число, которое нельзя представить в виде отношения двух целых чисел.

Предположим, нам нужно доказать, что число $\sqrt{2}$ является иррациональным. Допустим, оно является рациональным; тогда $\sqrt{2}$ можно представить в виде несократимой дроби a / b, где a и b — взаимно простые числа. Выполнив преобразование, получим $b\sqrt{2} = a$. Возведя обе части уравнения в квадрат, получим $b^2 * 2 = a^2$.

Поскольку a^2 в два раза больше b^2, число a должно быть четным, поэтому мы можем представить его в виде произведения $a = 2c$, где c — некое третье число. Итак, мы имеем: $2b^2 = (2c)^2 = 4c^2$. Тогда $b = 2c$, а это значит, число b — четное, что противоречит предположению о том, что дробь a / b является несократимой[1].

Таким образом, наше первоначальное предположение было неверным, то есть число $\sqrt{2}$ не является рациональным.

Если мы хотим использовать конструктивное доказательство, но не знаем, как подойти к построению объекта, то можем показать, что утверждение о том, что этот объект несуществует, приводит к противоречию. [1]

Доказательство, демонстрирующее существование объекта без его фактического построения, называется неконструктивным.

---

[1] Если числа a и b являются взаимно простыми, то у них не может быть никакого общего делителя, кроме +/–1.

16.3. Доказательство методом индукции

Доказательство методом индукции состоит из двух этапов. На первом из них, называемом базой (или базисом) индукции, мы доказываем истинность гипотезы для некоего начального случая или случаев.

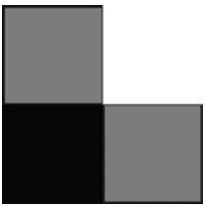
На втором этапе, называемом индуктивным переходом (или шагом индукции), мы показываем, что если утверждение верно для случая k, то оно верно и для следующего случая $k + 1$. О нескольких базовых случаях речь идет тогда, когда данный этап доказательства зависит от нескольких предыдущих шагов или когда для доказательства необходимо, чтобы предыдущий шаг был на некую минимальную величину больше того базового случая, который мы хотим доказать.

Итак, наше рассуждение строится следующим образом. Утверждение верно для случая 1. Поскольку оно верно для случая 1, оно должно быть верно для случая 2. Поскольку оно верно для случая 2, оно должно быть верно для случая 3. Поскольку оно верно для случая n, оно должно быть верно для случая $n + 1$.

Как только мы показали, что утверждение верно для базового случая, а также то, что из истинности утверждения для каждого случая вытекает его истинность для каждого последующего случая, мы можем сказать, что утверждение является истинным для любого случая n, а значит, теорему можно считать доказанной.

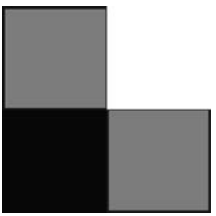
16.3.1. Пример

Представьте, что у нас есть шахматная доска, из которой вырезана одна клетка. Подобная шахматная доска 2×2 могла бы выглядеть так.

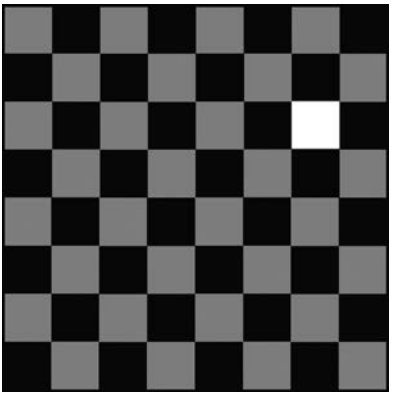


Назовем эту фигуру L. Теперь представьте, что у нас есть шахматная доска $2^n \times 2^n$ с вырезанной клеткой. Нам нужно доказать, что ее можно замостить (то есть покрыть все клетки доски) только L-образными плитками.

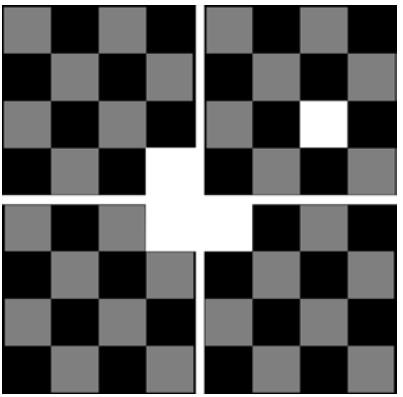
Базовый случай — показанная выше доска 2×2 с вырезанной клеткой. Форма данной доски по определению совпадает с формой L-образной плитки, поэтому очевидно, что ее можно замостить.



Предположим, данное утверждение верно для доски размером до $2^k \times 2^k$; это наша *индуктивная гипотеза* (ИГ). Рассмотрим доску размером $2^{k+1} \times 2^{k+1}$, которую можно представить в виде сетки 2×2, состоящей из досок $2^k \times 2^k$.



Положите плитку так, чтобы каждый из составляющих ее квадратов располагался на одной из трех четвертей доски, в которых нет вырезанных клеток. Теперь у нас есть четыре доски размером $2^k \times 2^k$, в каждой из которых отсутствует (или закрыта) одна клетка, которые, согласно нашей индуктивной гипотезе, можно выложить плиткой.



Такой способ покрытия всей шахматной доски $2^{k+1} \times 2^{k+1}$ доказывает, что подобную доску можно замостить плитками при любом положительном целом k.

16.3.2. Ошибочное доказательство

Предположим, я хочу доказать, что все лошади одного цвета[1].

База индукции: очевидно, что в множестве, состоящем из одной лошади, все лошади одного цвета.

Индуктивная гипотеза: в любом множестве лошадей, состоящем из k особей, все лошади одного цвета.

Индуктивный шаг: рассмотрим множество, состоящее из $k + 1$ лошадей. Согласно ИГ, первые k из них одного цвета. Согласно ИГ, последние k из них тоже одного цвета. Поскольку эти два множества пересекаются, все $k + 1$ лошадей имеют одинаковый цвет, следовательно, все лошади одного цвета!

Что не так с этим доказательством? В данном примере нам нужно было доказать два базовых случая. Индуктивный шаг предполагает пересечение множеств, состоящих из первых и последних k лошадей, что неверно, если $k = 1$. Нам также следовало доказать базовый случай для двух лошадей, но мы не можем показать, что две лошади имеют одинаковый цвет, поэтому данное доказательство является ошибочным.

---

[1] На самом деле это не так.

16.4. Доказательство на основе закона контрапозиции

Зачастую для того, чтобы доказать то или иное утверждение, гораздо легче доказать его модифицированную (возможно, более сильную) версию, которая подразумевает то, что мы хотели доказать с самого начала. Согласно закону контрапозиции, если некая посылка A влечет за собой некое следствие B ($A \Rightarrow B$), то отрицание следствия B влечет за собой отрицание посылки A ($\bar{B} \Rightarrow \bar{A}$). Если контрапозиционное утверждение истинно, то исходное утверждение также должно быть истинным, иначе посылка A и ее отрицание $\bar{A}$ будут иметь место одновременно.

Пример

Предположим, мы хотим доказать, что если число k является иррациональным, то число $\sqrt{k}$ также должно быть иррациональным.

Контрапозиционное утверждение состоит в том, что если число $\sqrt{k}$ является рациональным, то число k также должно быть рациональным.

Предположим, что $\sqrt{k}$ — это рациональное число; тогда его можно представить в виде отношения двух целых чисел: $\sqrt{k} = a / b$.

Возведя обе части уравнения в квадрат, получаем $k = a^2 / b^2$. Это уравнение показывает, что число k также представляет собой отношение двух целых чисел и, следовательно, является рациональным.

Эта цепочка рассуждений доказывает, что если $\sqrt{k}$ — рациональное число, то число k также должно быть рациональным, и наоборот, если k является иррациональным числом, то число $\sqrt{k}$ тоже должно быть иррациональным.

17 Сертификаты

Допустим, у нас есть проверенный алгоритм, позволяющий решить ту или иную задачу. Если мы реализуем этот алгоритм в виде программы, то можем ли быть уверены в том, что она будет выдавать правильное решение?

> Опасайтесь ошибок в приведенном выше коде;
> я доказал его корректность, но еще
> не запускал его.
>
> *Дональд Кнут*[1]

Сложность заключается в следующем: несмотря на корректность алгоритма, мы не можем быть уверены в том, что его реализация не содержит ошибок. Поэтому нам необходимо каким-то образом проверить, действительно ли программа возвращает правильный ответ.

Доказательство корректности называется *сертификатом*. Это то, что мы можем легко проверить, чтобы убе-

---

[1] *Knuth D. E.* Notes on the van Emde Boas construction of priority deques: An instructive use of recursion, classroom notes. March 1977.

диться в правильности ответа. Например, рассмотрим программу, определяющую, является ли то или иное число простым. Если ответ отрицательный, то сертификат будет представлять собой набор целых чисел, произведением которых является проверяемое нами число.

Наша цель — удостовериться в отсутствии ошибок в программе, поэтому нам необходимо добиться того, чтобы проверка сертификата происходила быстрее и проще (то есть отличалась меньшим асимптотическим временем выполнения и меньшей сложностью), чем решение исходной задачи. Сертификат считается сильным, если алгоритм верификации имеет лучшие

временные показатели по сравнению с процессом решения исходной задачи, и слабым, если это не так; на практике алгоритмы часто имеют сильный сертификат отвержения и слабый сертификат принятия или наоборот[1].

Пример

Проверка графа на двудольность занимает время $O(n+m)$. Если граф является двудольным, то *сертификат принятия* представляет собой раскраску графа в два цвета. В данном случае проверка также занимает время $O(n+m)$, что делает сертификат слабым. Если граф не является двудольным (содержит нечетный цикл), то *сертификат отвержения* может быть проверен за время $O(n)$, что делает этот сертификат сильным.

[1] *Kratsch, D. McConnell R. M., Mehlhorn K., Spinrad J. P.* Certifying algorithms for recognizing interval graphs and permutation graphs // ACM-SIAM SODA. № 14 (2003). P. 866–875.

Часть VII
Безопасность и конфиденциальность

18 Введение в безопасность

Обеспечение информационной безопасности — это комплекс мер, направленных на предотвращение угроз, связанных с нарушением конфиденциальности, целостности и доступности данных.

18.1. Конфиденциальность

Рассмотрим следующие виды информации:

- переписку пациента и врача на медицинском портале;
- электронную таблицу с суммой расходов на предвыборную рекламу;
- список логинов и паролей пользователей для входа на популярные торговые площадки.

В каждом из этих случаев очень важно, чтобы информация оставалась конфиденциальной[1]. При нарушении

[1] Иногда даже само наличие информации должно оставаться в тайне. Например, факт общения пациента с онкологом раскрывает личную медицинскую информацию.

конфиденциальности информация становится доступна третьим лицам, не уполномоченным на ее получение. Это может произойти по нескольким причинам, изложенным ниже.

- В компании не внедрена политика, запрещающая раскрытие информации, или ее сотрудники не следуют данной политике. Например, конфиденциальные документы хранятся в каталоге сайта, к которому может получить доступ поисковый робот[1].
- Злоумышленник получил доступ к информации, замаскировавшись под доверенного пользователя (сбой аутентификации). Законный пользователь может щелкнуть на ссылке в электронном письме, якобы отправленном Amazon, и, оказавшись на поддельном сайте, ввести свои учетные данные, после чего злоумышленник сможет совершать покупки от его имени.
- Злоумышленник сумел прочитать файл, который должен был быть нечитаемым (сбой шифрования). Файл мог быть зашифрован с помощью устаревшей схемы шифрования или легко угадываемого пароля.
- Злоумышленник смог перехватить данные во время их передачи от одного авторизованного пользователя другому.

Безопасное хранение и передача информации часто требует шифрования, о котором мы поговорим в следующих двух главах.

---

[1] Это не шутка. Я случайно обнаружил целые налоговые декларации в ходе поиска совершенно другой информации.

18.2. Целостность

Понятие «целостность» имеет отношение к достоверности информации и охватывает две области: целостность данных (сохранение информации в том виде, в каком она была создана) и целостность источника (наличие сведений о том, откуда поступила эта информация).

Чтобы обеспечить целостность данных, нам необходимы механизмы предотвращения или выявления фактов их несанкционированного изменения[1]. Кроме того, мы хотим знать об источнике данных, поскольку от этого зависит их достоверность. Например, в медицинских вопросах мы склонны больше доверять врачу, чем администратору клиники. Благодаря механизмам обеспечения целостности данных мы можем не только узнать, кто отправил сообщение, но и доказать это. Подробнее об этом мы поговорим в главе 22.

18.3. Доступность

Если злоумышленнику и не удастся скомпрометировать систему, то он все равно может сделать ее недоступной для использования. Его действия могут варьироваться от простой попытки сделать сервис недоступным для всех (например, проведя атаку типа «отказ в обслуживании» на популярный сайт) до вызова задержек, позволяющих провести повторную атаку (приводящую к аварийному переключению с безопасного сервера на взломанный).

---

[1] Данное изменение не обязательно должно быть умышленным. С помощью контрольной суммы можно выявить как признаки преднамеренного изменения, так и случайные ошибки.

18.4. Цели

Различные механизмы обеспечения безопасности преследуют три основные цели:

- предотвращение успешной атаки. Политика использования надежных паролей направлена на защиту системы от несанкционированного доступа;
- восстановление после атаки. Это может означать восстановление удаленного или измененного файла из резервной копии или судебное преследование злоумышленника;
- выявление факта атаки. Это может означать обнаружение выполняющейся атаки с целью ее отражения (например, блокирование пользователя после нескольких неудачных попыток входа в систему) или выявление уже произошедшего вторжения и принятие мер по восстановлению системы.

19 Введение в криптографию

Криптография — это наука о способах обеспечения безопасности коммуникации, охватывающая широкий спектр математических методов[1]. Криптографию можно считать одним из самых опасных разделов Computer Science, учитывая частоту случаев некачественной реализации и возможные последствия использования небезопасной криптографической системы.

Предотвратить перехват сообщения можно несколькими способами. Самый простой (если не в плане реализации, то по крайней мере в концептуальном плане) заключается в том, чтобы скрыть это сообщение. В идеале противник вообще не должен знать о его отправке.

Люди использовали *стеганографию* (практику сокрытия факта передачи сообщения) на протяжении многих тысячелетий. Согласно греческому историку Геродоту, тиран Гистией велел обрить голову своему слуге и вытатуировать на ней послание, а когда волосы отросли,

---

[1] Более подробно с историей криптографии вы можете ознакомиться, прочитав увлекательную «Книгу шифров» Саймона Сингха (Simon Singh).

отослал его с поручением снова обрить голову по прибытии на место. В наше время изображение или другой файл может содержать скрытое послание, часто прямо на виду. Если криптография позволяет скрыть содержимое сообщения, то стеганография позволяет скрыть сам факт его отправки.

Сообщение можно защитить двумя способами: с помощью кодов и с помощью шифров. При использовании шифра буквы (или группы букв) заменяются другими буквами (или группами букв). Например, широко известный шифр Цезаря предполагает замену каждой буквы другой буквой, находящейся на несколько позиций левее или правее в алфавите[1]. При использовании кода словам или фразам придается другое, заранее оговоренное значение, не имеющее прямого отношения к сказанному; например, фраза «куриный суп» может означать «атака на рассвете».

19.1. Современная криптография

В прошлом сохранение информации в тайне в значительной степени опиралось на принцип «безопасность через неясность», предполагающий сокрытие сообщения или отсутствие у противника ключей для его расшифровки.

В основе современной криптографии лежит математика. Мы предпочитаем использовать криптосистему, спроектированную таким образом, что, даже имея

[1] Говорят, при переписке со своими генералами сам Цезарь использовал сдвиг на три символа.

копию зашифрованного текста *(шифротекст)* и зная, как именно он был зашифрован, злоумышленник все равно не сможет прочитать сообщение, не зная общего секретного *ключа*. Наличие этого ключа позволяет расшифровать сообщение и восстановить *открытый (исходный) текст*.

19.2. Терминология

Рассмотрим стандартную ситуацию. Алиса отправляет сообщение Бобу. Третье лицо, Ева, хочет перехватить это сообщение.

У Алисы и Боба есть некий фрагмент информации, обеспечивающий безопасность их коммуникации. Если они применяют алгоритм *симметричного шифрования*, то имеют копию ключа, который используется как для шифрования, так и для расшифровки. Симметричное шифрование предполагает, что у участников есть возможность безопасным образом согласовать ключ, прежде чем задействовать его для передачи сообщений. Этот обязательный обмен ключами — основной недостаток симметричного шифрования, поскольку требует того, чтобы две стороны заранее договорились относительно алгоритма шифрования или нашли уже существующий метод безопасной коммуникации. Данную проблему можно обойти с помощью *квантового распределения ключей,* предполагающего использование квантовых явлений для безопасной передачи ключей.

Алгоритмы *асимметричного шифрования*, также известные как алгоритмы *с открытым ключом*, используют разные ключи для шифрования и расшифровки.

Сообщение, зашифрованное с помощью открытого ключа, можно расшифровать с помощью закрытого.

В некоторых ситуациях верно обратное: один человек может зашифровать сообщение с помощью закрытого ключа, а любой обладатель открытого ключа может его расшифровать. В этом случае закрытый ключ может использоваться для аутентификации: обладатель закрытого ключа может подписать документ, а любой обладатель открытого может проверить данную подпись[1].

19.3. Абсолютно безопасный обмен данными

Теоретически, если времени и вычислительной мощности достаточно, можно взломать практически любой шифр. Чем больше объем передаваемого шифротекста, созданного с помощью одного и того же ключа, тем проще его расшифровать. Например, при использовании шифра подстановки (когда каждая буква заменяется другой) по мере увеличения длины текста в нем будет наблюдаться частотное распределение букв, характерное для языка, на котором написано сообщение, что упрощает его расшифровку методом частотного анализа.

Исключение и единственный способ шифрования, который нельзя взломать с помощью криптографических методов, — одноразовый блокнот (рис. 19.1). При его

[1] Однако этот подход может ослабить безопасность схемы шифрования, вследствие чего обладателю ключей рекомендуется использовать разные ключи для расшифровки и подписи.

использовании у Алисы и Боба есть общий ключ, длина которого не может быть меньше длины передаваемого текста, и каждая буква этого текста комбинируется с соответствующей буквой блокнота для получения шифротекста. При условии, что блокнот содержит действительно случайные символы, хранится в секрете и применяется только один раз, полученный с его помощью шифротекст не может быть расшифрован. Сложность заключается в создании и распространении этих одноразовых блокнотов (это одна из тех задач, которые решает квантовая криптография).

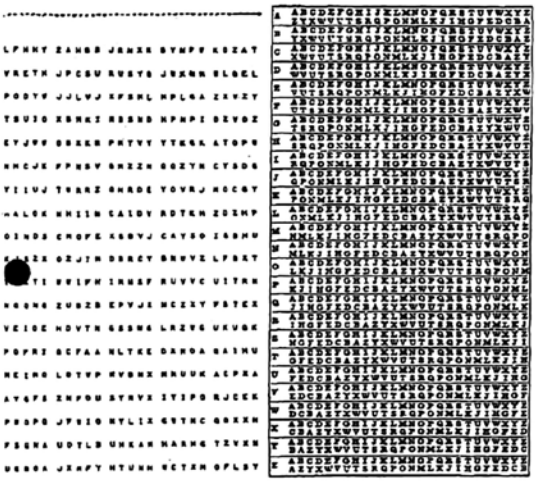


Рис. 19.1. Одноразовый блокнот, использовавшийся Агентством национальной безопасности США. Слева сам блокнот, а справа — инструкции по шифрованию каждой буквы открытого текста с учетом соответствующей буквы блокнота. Изображение взято из рассекреченного документа, доступного по адресу https://bit.ly/3d6byxX

19.4. Квантовое распределение ключей

Самый известный способ применения квантовой криптографии — квантовое распределение ключей, которое позволяет Алисе и Бобу сгенерировать ключ, не прибегая к личной встрече или любой другой форме безопасной коммуникации.

При этом требуется, чтобы Алиса и Боб обменивались данными по аутентифицированному классическому каналу, то есть отправляли сообщения (не обязательно конфиденциальные) и были уверены в том, что они общаются друг с другом, а не с кем-то посторонним.

Технология квантового распределения ключей опирается на тот факт, что перехватчик должен будет измерить передаваемые квантовые состояния, а значит, изменить их. А такие изменения можно выявить.

В классическом примере используются состояния поляризации фотона. Алиса посылает поток фотонов и измеряет их состояния, но для каждого из них она случайным образом выбирает прямолинейную или диагональную поляризацию. Получая фотоны, Боб делает то же самое. Затем они сравнивают поляризации, выбранные ими для каждого фотона (задействуя для этого классический канал), и отбрасывают те, которые не совпадают.

После этого они выбирают случайное подмножество фотонов и сравнивают свои результаты. Если информация не была перехвачена и квантовый канал является

достаточно надежным, то результаты почти всех измерений должны совпасть. Если же Еве удалось перехватить фотоны, то примерно в 50 % случаев она не сможет определить состояние полученного фотона и отправит фотон Бобу в выбранном наугад состоянии. В результате его измерения будут отличаться от измерений Алисы примерно в 25 % случаев, из чего они смогут сделать вывод о небезопасности квантового канала.

Коммерческие системы квантового распределения ключей доступны уже в течение некоторого времени.

20 Криптографическая система с открытым ключом

Асимметричная криптография предполагает использование разных ключей для шифрования и расшифровки сообщения. Этот тип шифрования лежит в основе криптографии с открытым ключом, при которой открытый ключ передается по незащищенному каналу, а расшифровка сообщения выполняется с помощью закрытого ключа.

20.1. Использование открытого и закрытого ключей

Если Алиса хочет отправить Бобу сообщение, то сначала находит его открытый ключ и использует его для шифрования. Затем она может отправить Бобу зашифрованное сообщение или даже опубликовать его на открытом форуме. Получив сообщение, Боб задействует для его расшифровки свой закрытый ключ.

Суть работы этой системы заключается в том, что злоумышленник не может (с помощью известных методов)

получить закрытый ключ из открытого ключа **в разумные сроки**[1].

Использование закрытых ключей также позволяет Алисе подтвердить, что сообщение отправила именно она. После шифрования сообщения Алиса снова шифрует его с помощью своего закрытого ключа и прикрепляет эту копию в качестве подписи. Любой обладатель ее открытого ключа может расшифровать подпись, чтобы убедиться в ее соответствии исходному зашифрованному сообщению, а значит, в том, что отправителю был известен закрытый ключ Алисы[2].

В асимметричной криптографии используются так называемые односторонние функции с лазейкой — математические функции, которые легко вычислить, но трудно обратить, не имея определенной секретной информации.

Математика

Пусть дана такая функция f, что $f(x) = y$. Если f представляет собой хорошую одностороннюю функцию с лазейкой, то вычислить y из x легко, однако вычислять x из y при отсутствии ключа непрактично.

---

1 Если сообщение должно оставаться в секрете на протяжении недели, а на взлом шифра требуется три года, то мы можем говорить о достаточной степени безопасности данных.

2 Использование такой подписи обеспечивает и невозможность отказа от авторства. В дальнейшем Алиса не сможет заявить, будто не отправляла сообщение, не заявив при этом о том, что кто-то посторонний знает ее закрытый ключ.

20.2. Алгоритм RSA

RSA[1] — это один из наиболее распространенных криптографических алгоритмов с открытым ключом.

При использовании этого алгоритма Алиса выбирает два больших простых числа p и q сопоставимой длины и вычисляет их произведение $n = pq$. Данная функция является односторонней, поскольку перемножить два целых числа легко, а разложить результат на простые множители — нет.

20.2.1. Создание ключей

Алиса использует два больших простых числа p и q и их произведение n, чтобы сгенерировать открытый ключ e и закрытый ключ d, который нелегко получить из e. Затем она публикует ключ шифрования e и произведение n, поэтому они оказываются известны всем.

20.2.2. Шифрование с помощью открытого ключа

Если Боб хочет отправить Алисе сообщение M, то преобразует M в целое число m, которое подлежит шифрованию, а затем выполняет следующие операции.

1. Возводит число m в степень e: $m' = m^e$.

---

[1] RSA — это аббревиатура, состоящая из первых букв фамилий создателей алгоритма: Rivest (Ривест), Shamir (Шамир) и Adleman (Эдлман).

2. Берет результат по модулю n, чтобы получить шифротекст c: $c = m' \bmod n$.

20.2.3. Расшифровка с помощью закрытого ключа

Получив сообщение (c), Алиса расшифровывает его, используя свой закрытый ключ.

1. Сначала она возводит зашифрованное число в степень d: $c' = c^d$.
2. Затем, чтобы восстановить исходное целое число, выполняет операцию взятия по модулю: $m = c' \bmod n$.
3. Наконец, преобразует m в сообщение M, которое теперь может прочитать.

Математика

Почему это работает? Согласно малой теореме Ферма, если p — простое число, а a — целое число, то $a^p \equiv a \bmod p$. Символ ≡ означает, что обе части конгруэнтны (сравнимы) по модулю. Например, числа 3 и 15 сравнимы по модулю 12, поэтому 3-й час (3 часа ночи) и 15-й час (3 часа дня) находятся на одной позиции 12-часового циферблата.

В данном случае сообщение m возводится в степень e, а затем в степень d, $\bmod n$. При возведении степени в степень основание остается без изменений, а показатели степеней перемножаются, поэтому после шифрования и расшифровки мы получаем $m^{ed} \bmod n$. Алиса выбирает такие числа e и d, что в результате у нас остается исходное сообщение m.

20.2.4. Попытки перехвата сообщения

Чтобы расшифровать сообщение, Ева должна знать значение *d*. Для вычисления *d* ей нужно разложить *n* на простые множители (а значит, воссоздать закрытый ключ), что является сложной задачей. Таким образом, сообщение считается защищенным, поскольку современные технологии и математические методы не позволяют Еве взломать шифр в разумные сроки.

20.3. Соображения производительности

Поскольку симметричное шифрование происходит быстрее, чем асимметричное, асимметричные ключи, как правило, используются для обмена симметричными ключами, также известными как сеансовые ключи, которые применяются для шифрования и расшифровки данных в ходе сеанса. Этот метод позволяет объединить удобство асимметричной криптографии, скорость симметричной криптографии и безопасность, обусловленную частой заменой ключа (каждый симметричный ключ применяется только для одного сеанса связи).

По такому же принципу работает протокол SSL. Браузер получает копию асимметричного ключа сервера, создает сеансовый ключ, зашифровывает его и отправляет серверу. Тот расшифровывает сообщение, используя свой закрытый ключ, а в ходе остальной части сеанса данные шифруются с помощью общего секретного ключа.

Причина, по которой симметричное шифрование выполняется быстрее, заключается в том, что асимметричное шифрование предполагает использование более длинного ключа[1]. Поскольку симметричный ключ может представлять собой любое случайное число, надежность такого ключа зависит от его длины. Асимметричный ключ выбирается с помощью известного алгоритма, поэтому данное пространство ключей содержит меньше уникальных значений, кроме того, они подчиняются определенным закономерностям, которыми может воспользоваться злоумышленник. Как следствие, в целях обеспечения сопоставимого уровня надежности пространство ключей для асимметричного алгоритма должно быть больше, чем для симметричного.

Например, в случае описанного выше алгоритма RSA закрытый ключ состоит из двух простых чисел, а значит, все остальные числа можно игнорировать, поскольку они не могут быть частью этого ключа.

---

[1] Алгоритмы симметричного шифрования обычно используют ключи от 128 до 256 бит, в то время как асимметричные обычно применяют ключи от 1024 до 4096 бит.

21 Аутентификация пользователя

Пароли использовались для получения доступа к секретной информации на протяжении многих тысячелетий. Но в эпоху Интернета пароли часто не гарантируют должный уровень безопасности по нескольким причинам.

- **Слабые пароли.** Если система допускает неограниченное количество попыток ввода пароля, то злоумышленник может просто перепробовать все возможные пароли, пока не найдет правильный. В данном случае речь может идти об исчерпывающем поиске ключей (переборе всех допустимых комбинаций) или об атаке по словарю (переборе списка распространенных паролей).
- **Повторное использование паролей.** Даже надежный пароль можно выяснить. Если злоумышленник получит пароль, связанный с одним и тем же именем пользователя или адресом электронной почты на нескольких сайтах, то сможет получить доступ к каждому из них.

Введение дополнительных требований к длине и сложности паролей помогает решить первую проблему, но

повышает вероятность того, что пользователь повторно применит пароль (кому захочется запоминать несколько длинных и сложных комбинаций символов?), сохранит его в доступном злоумышленнику месте[1] или забудет. Одно из решений — менеджеры паролей (которые генерируют и сохраняют надежные пароли для разных сайтов, при этом пользователю требуется запомнить только один мастер-пароль). Правда, в случае взлома менеджера паролей злоумышленник получит доступ ко всем учетным записям пользователя[2].

Ниже перечислены другие средства повышения надежности паролей.

- **Принудительная смена пароля.** До недавнего времени корпорация Microsoft рекомендовала компаниям побуждать сотрудников к частой смене паролей с целью предотвращения возможной утечки последних. Однако частая смена паролей лишь усугубляет описанные выше проблемы (пользователи, которым приходится часто менять пароли, как правило, выбирают более легкие для запоминания), поэтому данная мера больше не рекомендуется[3]. Пароли, которые не были украдены, менять не обязательно, однако если

---

1 В 2018 году сотрудник Белого дома наделал много шума, когда записал свои пароли от зашифрованной электронной почты на официальном бланке и забыл его на автобусной остановке.

2 В 2019 году выяснилось, что ошибка кэширования в популярном менеджере паролей LastPass допускала утечку последних использованных паролей на вредоносный сайт.

3 https://bit.ly/3gLon38.

вы подозреваете, что они были скомпрометированы, то их следует незамедлительно заменить.

- **Списки запрещенных паролей.** Помимо предъявления требований к минимальной длине пароля, его можно сравнить со списком распространенных паролей и запретить его использование в случае совпадения с одним из присутствующих в этом списке вариантов.
- **Хеширование паролей.** Безопасный сайт не позволит вам восстановить утерянный пароль, поскольку никогда его не сохраняет. Вместо этого он объединяет пароль с так называемой солью (уникальной для каждого пользователя секретной строкой)[1], хеширует полученное значение[2] и сохраняет хешированное значение. Таким образом, даже если база данных паролей будет скомпрометирована, злоумышленнику все равно придется взламывать каждый пароль по отдельности.
- **Многофакторная аутентификация.** Как правило, тремя главными факторами аутентификации являются: знание (например, пароля для входа на сайт), облада-

---

[1] Применение уникальной для каждого пользователя соли позволяет предотвратить применение радужных таблиц — особых таблиц поиска для обращения криптографических хеш-функций. При наличии базы данных с хешированными паролями мы можем восстановить пароль, хранящийся в ней в виде хеш-значения, и таким образом получить доступ к нужной учетной записи. Добавление соли, уникальной для каждого пользователя, делает радужную таблицу бесполезной.

[2] Чтобы познакомиться с темой хеширования, обратитесь к разделу 2.5.

ние (смарт-картой для доступа в здание) и свойство (отпечаток пальца для разблокировки телефона). Использование кредитной карты с чипом и PIN-кодом требует двух факторов: обладания (самой картой) и знания (PIN-кода)[1]. В настоящее время многие сайты задействуют двухфакторную аутентификацию[2], для прохождения которой требуется пароль и временный код, позволяющий вам подтвердить наличие доступа к телефону или устройству, связанному с учетной записью.

---

1 В качестве еще одного примера можно привести вход в Disney World по многодневному билету, который предполагает использование сканера отпечатков пальцев, поэтому зависит как от обладания (билетом), так и от вашего свойства (отпечатка пальца).

2 Двухфакторная аутентификация, или 2FA, представляет собой разновидность многофакторной аутентификации, которая использует комбинацию двух факторов.

Часть VIII
Аппаратное и программное обеспечение

22 Аппаратные абстракции

Большинство из нас не занимается написанием машинного кода для непосредственного управления оборудованием компьютера. Вместо этого мы используем набор абстракций, позволяющих получить доступ к ресурсам определенного типа, не запоминая их физические детали. Эти абстракции могут иметь место на нескольких уровнях.

- При обращении к операционной системе нам нужно знать только ее API, а не конкретный набор команд процессора, на базе которого она работает.
- При написании кода на высокоуровневом языке программирования мы можем использовать общие команды, которые компилятор затем преобразует в специфичные для процессора инструкции.

22.1. Физическое хранилище

Данные хранятся в памяти компьютера в виде набора нулей и единиц.

Традиционный жесткий диск состоит из одной или нескольких круглых пластин; их поверхность разделена на

дорожки (концентрические круги) и секторы (клинья) (рис. 22.1). Каждый сектор содержит несколько[1] очень мелких областей, которые можно намагничивать или размагничивать по отдельности. По мере вращения пластин головки чтения/записи перемещаются к областям, в которых требуется считать или записать биты информации.

В твердотельном накопителе используется флеш-память типа NAND[2], хранящая данные в массиве транзисторов, то есть в ячейках памяти (на пересечении строки и столбца), которые могут находиться в одном из двух состояний в зависимости от подаваемого напряжения.

Каждый диск предусматривает контроллер — компонент, который управляет физическим доступом к памяти, позволяя ЦП просто запросить нужные данные, не прибегая к необходимости непосредственно управлять рычагом привода жесткого диска или подавать напряжение на транзисторы твердотельного накопителя. Помимо этого, контроллер может выполнять такие дополнительные функции, как выявление и переназначение поврежденных секторов, обнаружение и исправление ошибок, а также выравнивание износа[3].

---

[1] Традиционно размер сектора жесткого диска составлял 512 байт, однако в 2011 году был введен новый стандарт и размер сектора был увеличен до 4096 байт за счет объединения восьми 512-байтовых секторов.

[2] Сокращение NAND (Not AND) обозначает логический элемент «И-Не» (см. врезку).

[3] Для выравнивания износа контроллер пытается организовать данные таким образом, чтобы обеспечить равномерную запись во все ячейки памяти, поскольку они рассчитаны на ограниченное количество циклов записи.

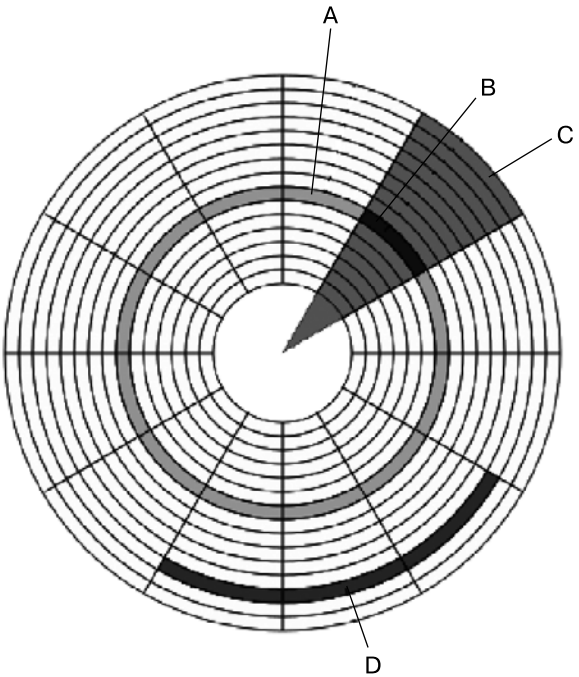


Рис. 22.1. Схематическое изображение пластины жесткого диска. Источник изображения: Wikimedia Commons

Каждый из концентрических кругов [A] называется дорожкой, а каждый клин в форме куска пирога [B] — геометрическим сектором. Пересечение дорожки и геометрического сектора называется сектором дорожки [C] и представляет собой наименьший элемент данных, который может быть прочитан с диска или записан на него. Кластер [D] — наименьшая логическая единица дискового пространства, которая фактически используется операционной системой (например, для хранения файла).

Забираемся в дебри

Любой тип флеш-памяти предполагает использование транзисторов для хранения данных, но эти транзисторы могут быть организованы по-разному. Два основных типа такой памяти, NOR и NAND, названы в честь соответствующих логических элементов. Различное расположение транзисторов обусловливает различные физические свойства.

Флеш-память типа NOR (Not OR) обеспечивает возможность высокоскоростного произвольного доступа, отличается высокой надежностью и позволяет считывать и записывать отдельные байты.

Для флеш-памяти типа NAND (Not AND) характерны ячейки меньшего размера, что обусловливает гораздо более высокую скорость записи и стирания данных, а также меньшую стоимость. Однако флеш-память NAND считывает и записывает данные отдельными страницами или блоками и не позволяет работать напрямую с байтами. Использование косвенного интерфейса усложняет получение доступа к такой памяти, а наличие плохих блоков требует дополнительных функций исправления ошибок, в которых не нуждается флеш-память NOR.

На практике флеш-память NOR используется для хранения и выполнения кода (например, прошивка мобильных телефонов), а флеш-память NAND — для хранения данных (карты памяти и твердотельные накопители).

22.2. Данные и методы ввода/вывода

Физическая структура жестких дисков и флеш-памяти типа NAND не позволяет считывать или записывать

отдельные байты. Вместо этого мы имеем дело с целым блоком. *Блок* — наименьшая логическая единица данных, которая может быть прочитана или записана операционной системой, а *сектор* — наименьшая физическая единица хранения данных на жестком диске. Таким образом, размер блока равен размеру одного или нескольких секторов. В зависимости от диска он, как правило, составляет 512 или 4096 байт[1].

SSD-накопители не имеют секторов, но их ячейки памяти организованы в страницы, а несколько страниц (обычно 128) составляют блок. Данные могут быть прочитаны и записаны на уровне страницы, но их стирание происходит только на уровне блока.

Каждый физический сектор имеет заголовок, включающий информацию, которую задействует контроллер диска, и область данных, содержащую как сохраненную пользователем информацию, так и коды коррекции ошибок (error-correcting code, ECC). Более крупные секторы более эффективны, поскольку их заголовок занимает меньше места, а значит, для хранения пользовательских данных может быть выделено больше дискового пространства. Более крупные секторы также нуждаются в большем объеме ECC-кода для поддержания той же эффективности исправления ошибок. Кроме того, поскольку жесткие диски большей емкости отличаются более плотной упаковкой данных, любой

[1] Диски с 4096-байтовыми секторами обычно способны эмулировать 512-байтовые секторы. Однако обновление логических 512-байтовых секторов все равно предполагает чтение и запись всего 4096-байтового сектора, что сопровождается уменьшением скорости.

физический недостаток может повлиять на большее количество битов, что опять же требует большего объема ECC-кода для поддержания уровня эффективности исправления ошибок.

22.3. Память

Когда речь заходит о компьютерной памяти, помимо объема, программистов больше всего интересует ее распределение. При выполнении программы данные хранятся либо в (контрольном) стеке, либо в куче. В любом случае они хранятся в памяти (с одной оговоркой, о которой чуть позже); стек и куча — это просто структуры данных, которые используются для их организации.

Когда запущенная программа вызывает функцию, все переменные этой функции помещаются в стек. После выполнения функции они удаляются из стека, и память освобождается для повторного использования. Кроме того, стек содержит указатели на функции, что обеспечивает возврат управления в нужное место.

Стек представляет собой простую структуру данных типа LIFO (Last In, First Out — «последним пришел, первым ушел»; см. раздел 2.4), поэтому выделение и освобождение памяти происходит более эффективно, чем при использовании кучи[1]. В многопоточном приложении каждый поток имеет свой стек. Поскольку стек, как правило, содержит (относительно) небольшое количество элементов, а доступ к стековой памяти

[1] После выхода из функции вся память, которая была выделена для нее в стеке, автоматически освобождается.

осуществляется достаточно часто, значения в стеке, вероятно, будут кэшироваться. Кроме того, согласно результатам исследований, использование небольшого стекового кэша (отдельного от основного кэша) позволяет значительно повысить производительность[1].

В стек помещаются только примитивы и ссылки; объекты всегда помещаются в кучу[2]. В отличие от стековой памяти, доступ к которой строго регулируется, любой элемент кучи может быть доступен в любое время ввиду ее неупорядоченности. И куча, и стек занимают место в памяти; часто стек растет сверху вниз, от самого высокого адреса к самому низкому, а куча — снизу вверх, от самого низкого к самому высокому.

В отличие от стека, память для кучи выделяется динамически и может быть освобождена в любое время[3]. Доступ к памяти кучи отличается меньшей эффективностью, и эту память необходимо отслеживать, поскольку ее освобождение не происходит автоматически. Куча способна вмещать объекты разного размера, которые могут храниться в ней, пока в стеке существуют ссылки на них.

---

1 *Schoeberl M., Nielsen C.* A Stack Cache for Real-Time Systems // IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC). New York, 2016. P. 150–157.

2 Здесь под кучей подразумевается не структура данных, а область памяти, выделяемая не так, как в случае стека. Когда для выделения памяти используется функция new (в C++) или malloc (в C), эта память поступает из кучи.

3 Когда вы создаете объект, для него выделяется память в куче. Когда вы освобождаете память (или объект обрабатывается сборщиком мусора), эта память снова становится доступной для использования.

22.4. Кэш

Несмотря на то что оперативная память работает быстрее, чем жесткий диск, она не может незамедлительно предоставлять данные процессору. Отчасти это объясняется предельной скоростью света: за тот или иной промежуток времени свет (а значит, и данные) может распространиться лишь на определенное расстояние[1]. В вакууме свет преодолевает примерно 30,48 см за одну наносекунду[2]. Это значит, если ЦП работает на частоте 3 ГГц, то максимальное расстояние, на которое может распространиться сигнал за один тактовый цикл, составляет 10,16 см. Таким образом, если память находится на расстоянии более 6 см от ЦП, то данные не могут быть получены за один тактовый цикл, даже если эта память реагирует мгновенно.

Однако память, разумеется, не реагирует мгновенно. Еще один важный фактор — количество времени, необходимое модулю памяти для доступа к определенной ячейке[3]. Даже при очень быстром ОЗУ эта задержка означает, что между запросом информации и получением доступа к ней пройдет несколько наносекунд.

---

1 Разумеется, скорость света — абсолютный предел; в действительности мы не передаем данные настолько быстро, это максимальная скорость, с которой их в принципе можно передавать.

2 Объяснение понятия наносекунды адмиралом Грейс Хоппер (Grace Hopper) можно найти по адресу: https://www.youtube.com/watch?v=9eyFDBPk4Yw.

3 Это так называемая CAS-латентность, или CAS-задержка (от Column Address Strobe — строб адреса столбца).

Решение обеих этих задач заключается в расположении небольшого объема очень быстрой памяти на самом кристалле[1]. Близость к процессору позволяет минимизировать задержку, связанную с предельной скоростью света, а использование небольшого объема очень быстрой памяти вдобавок помогает сократить время ожидания. Быстрая память, размещаемая на кристалле, отличается ограниченным объемом и высокой стоимостью[2], поэтому применяется для кэширования самых часто используемых данных. Эта встроенная в процессор память называется кэшем L1 (или кэшем первого уровня). Существует также кэш L2, который находится на материнской плате, но может отсутствовать на самом чипе[3].

22.5. Регистры

В то время как кэш-память используется компьютером для хранения данных, которые могут понадобиться

---

[1] Кристалл — это небольшой блок полупроводникового материала, на котором размещены транзисторы, составляющие ЦП; вместе со встроенной электроникой он образует интегральную схему (как правило, называемую микросхемой или чипом). Сначала производится кремниевая полупроводниковая пластина с несколькими процессорами, которая затем разделяется на отдельные устройства. Каждый кристалл может содержать несколько ядер.

[2] В качестве кэш-памяти задействуется память типа SRAM, которая работает намного быстрее, чем DRAM, используемая в качестве основной памяти, однако она занимает больше места и стоит дороже.

[3] Помимо большей удаленности от процессора, кэш L2, как правило, использует более медленную память типа DRAM, однако его объем обычно в несколько раз превышает объем кэша L1.

в ближайшем будущем, регистры служат для хранения данных и инструкций, необходимых в настоящий момент. Процессор работает с регистрами напрямую, поэтому они представляют собой области памяти, находящейся в непосредственной близости от ЦП и имеющей минимальное время отклика. В разделе 25.1 мы рассмотрим пример инструкции машинного кода, указывающей регистр, в который должно быть помещено значение.

Обычно ЦП имеет как регистры общего назначения (которые хранят временные данные и могут быть доступны пользовательским программам), так и регистры специального назначения наподобие аккумулятора, счетчика команд и регистра команд[1].

[1] Аккумулятор служит для хранения промежуточных результатов арифметических операций. Его можно представить в виде блокнота, в котором записывается результат одного вычисления, используемого в качестве входных данных для следующего. Счетчик команд содержит адрес следующей инструкции, которую требуется извлечь из памяти. Регистр команд содержит инструкцию, выполняемую в настоящий момент.

23 Программные абстракции

23.1. Машинный код и язык ассемблера

Вначале был машинный код. Машинный код — это код, который выполняется непосредственно процессорами и является специфичным для того или иного их семейства. Двоичное (или, для удобства программиста, шестнадцатеричное) число представляет действие, которое требуется выполнить, и данные, над которыми его нужно произвести. Язык ассемблера упрощает процесс программирования, заменяя числа мнемокодами.

Например[1], вместо записи 10110000 01100001 (в двоичной системе счисления) или B0 61 (в шестнадцатеричной) мы можем написать MOV AL, 61h. В данном случае B0 — машинная инструкция (для процессора

[1] Я взял данный пример из «Википедии» (https://en.wikipedia.org/wiki/Machine_code), поскольку не писал на языке ассемблера уже два десятилетия и не хочу заниматься этим снова.

x86/IA-32), которая приказывает системе переместить следующее значение в регистр AL. Запись MOV AL означает то же самое, но является более простой для восприятия. Ассемблер преобразует программу на языке ассемблера в машинный код, который затем выполняет процессор.

23.2. Низкоуровневые языки программирования

Машинный код и язык ассемблера считаются языками программирования низкого уровня, поскольку абстракция между написанной командой и фактически выполняемым кодом практически отсутствует, а компилятор или интерпретатор не требуется. Код, написанный на низкоуровневом языке, может быть очень эффективным, но, как правило, непереносимым, поскольку приспособлен для конкретного семейства процессоров.

23.3. Высокоуровневые языки программирования

В отличие от низкоуровневых языков программирования высокоуровневые имеют более высокий уровень абстракции, позволяющий компилировать или интерпретировать код для обеспечения его работоспособности на различных машинах. Эта абстракция означает, что программист не может оптимизировать код, исходя из свойств конкретного оборудования, на котором он будет выполняться. В ситуации, когда производительность имеет критическое значение,

а о целевом оборудовании известно все, программист может написать самый важный фрагмент кода на языке ассемблера, а остальную часть программы — на высокоуровневом языке. В других случаях программист, умеющий работать как с языком ассемблера, так и с компилятором для высокоуровневого языка, может написать высокоуровневый код таким образом, чтобы скомпилированный код, генерируемый компилятором, был достаточно эффективным.

Когда следует использовать язык ассемблера

Программисты, которые пишут код на языке ассемблера, говорят, что у такого подхода есть три преимущества. Первые два очевидны: код занимает меньше места и работает быстрее. Третье не столь очевидно: если в коде есть ошибка, то она, как правило, приводит к катастрофическому сбою. На первый взгляд это не кажется преимуществом, однако сразу дает понять, что вы сделали что-то не так.

В подавляющем большинстве случаев проще работать на высокоуровневом языке. Язык ассемблера стоит использовать при работе со встроенными системами с ограниченной вычислительной мощностью, особенно когда экономия места позволяет задействовать более дешевые процессоры. В случае создания крупномасштабных приложений экономия на процессорах может более чем компенсировать дополнительное время разработчика.

Например, программы DSP (цифровой обработки сигналов) часто пишутся на языке ассемблера.

24 Компьютерная арифметика

Представьте приложение, которое должно очень быстро выполнять миллионы вычислений.

Подойти к его разработке можно несколькими способами. Один из них заключается в создании специализированного оборудования, оптимизированного для выполнения необходимых операций[1]. Другой — в особом структурировании вычислений, позволяющем воспользоваться преимуществами тех операций, которые компьютер выполняет очень быстро[2].

---

[1] Например, графические процессоры (ГП) способны выполнять определенные типы вычислений во много раз быстрее, чем ЦП общего назначения. По этой причине они оказываются полезными и для неграфических приложений, например для добычи биткойна.

[2] Рассмотрим задачу вычисления обратных квадратных корней, которые используются, например, при нормализации векторов. Существует быстрый приближенный алгоритм вычисления обратного квадратного корня с помощью битового сдвига, применявшегося в видеоиграх, таких как Quake. В то время это было гораздо быстрее по сравнению с выполнением точных расчетов. Для получения дополнительной информации введите запрос 0x5f3759df в строку поисковой системы.

24.1. Битовый сдвиг

Операции битового сдвига (<< и >> в языках семейства C) сдвигают все биты в регистре на одну позицию.

При сдвиге влево биты, освободившиеся справа от операнда, заполняются нулями. Сдвиг влево на n бит соответствует умножению числа на 2^n.

Сдвиг вправо может быть арифметическим или логическим[1]. При арифметическом сдвиге вправо крайний левый бит дублируется. В случае двоичного числа, записанного в дополнительном коде[2], сдвиг вправо на n бит эквивалентен делению на 2^n с округлением в меньшую сторону (то есть в сторону отрицательной бесконечности).

При логическом сдвиге вправо освободившиеся слева разряды заполняются нулями (что вполне уместно в случае беззнаковых двоичных чисел).

---

[1] То же касается и сдвига влево, просто между арифметическим и логическим сдвигом влево нет никакой разницы.

[2] Обратный код двоичного числа (дополнение до единицы) получается путем инверсии каждого бита, а дополнительный код (дополнение до двух) получается из обратного кода числа путем добавления единицы. Как правило, для представления знаковых чисел используется дополнительный код: положительное значение имеет знаковый бит, равный нулю, и хранится в прямом коде, а отрицательное — знаковый бит, равный единице, и хранится в дополнительном коде.

Зачем использовать битовый сдвиг, если он логически эквивалентен умножению или делению на два? Главная причина — скорость. Раньше из-за дефицита пространства или пропускной способности данные упаковывались максимально плотно и считывались полубайтами путем битового сдвига.

Пример — C#

В C# битовые операции определены для типов int, uint, long и ulong. Другие типы преобразуются в int (и битовая операция возвращает значение типа int).

В случае любой переменной *num* подходящего типа операция *num* << *x* приведет к сдвигу *num* влево на *x* бит. Например, если *num* = 3 (0000 0011), то операция *num* << 4 будет эквивалентна $3 \times 2^4 = 48$ (0011 0000).

Операция сдвига вправо выполняет арифметический сдвиг, если операнд является знаковым, и логический, если это не так.

24.2. Битовые И и ИЛИ

В то время как логические операторы И и ИЛИ (&& и || в языке C) принимают два логических значения (или значения, которые могут рассматриваться в качестве таковых) и возвращают логическое значение, битовые операторы И и ИЛИ (& и |) принимают и возвращают биты. Битовое И возвращает 1, если оба входных бита равны 1, и 0, если это не так. А битовое ИЛИ возвращает 0, если оба входных бита равны 0, в противном случае возвращается 1.

Пример — маскирование

Битовые операции часто используются в целях маскирования. Предположим, у объекта есть несколько логических свойств. Мы можем объединить их в одну переменную, присвоив каждому степень двойки и применив к ним оператор ИЛИ:

```
PropertyOne = 2⁰ = 1 << 0 = 1
PropertyTwo = 2¹ = 1 << 1 = 2
PropertyThree = 2² = 1 << 2 = 4
PropertyFour = 2³ = 1 << 3 = 8
```

Если объект обладает свойствами один, два и четыре, то переменная-флаг будет иметь значение 0001 | 0010 | 1000 = 1011.

Позднее мы можем захотеть выяснить, обладает ли объект свойствами три и четыре.

```
PropertyThree: (0100 & 1011) = 0
PropertyFour: (1000 & 1011) = 1
```

В коде мы определили бы эти свойства как константы и написали бы следующее[1]:

```
if ((PropertyThree & flagVar) !=0)
if ((PropertyFour & flagVar) !=0)
```

Если мы хотим проверить несколько свойств сразу, то можем выполнить над ними операцию с помощью ИЛИ, объединив их в новую переменную `sumVar`, а затем применить к переменным `sumVar` и `flagVar` оператор И. Если в результате будет возвращено значение `sumVar`, значит, каждое из исходных логических значений истинно.

---

[1] Да, в языке C выражение !=0 использовать не обязательно, поскольку ноль интерпретируется как «ложь», а все остальное — как «истина», однако при обращении с числом как с логическим значением может возникнуть ошибка.

24.3. Битовое НЕ

Битовое НЕ (~) инвертирует состояние каждого бита исходного числа, то есть единицы становятся нулями, а нули — единицами.

Пример — маскирование

Продолжим рассмотрение приведенного выше примера. Теперь мы хотим установить для четвертого свойства значение «ложь». Для этого мы применяем оператор НЕ к `PropertyFour` и получаем ~1000 = 0111. После этого применяем к `PropertyFour` и `flagVAR` оператор И: 0111 & 1011 = 0011. Теперь у нас установлены только два первых свойства.

24.4. Битовое исключающее ИЛИ

Логический оператор «исключающее ИЛИ» (XOR или ^) возвращает значение «истина», если истинным является лишь один из операндов; его битовая версия возвращает 1, если только один из операндов равен 1.

Следствием этого является обратимость операции XOR, то есть ее повторное применение возвращает исходное значение. Например, мы хотим зашифровать открытый текст 10100101 с помощью одноразового блокнота[1] 10110111.

- Чтобы зашифровать сообщение, мы выполняем над текстом и блокнотом операцию XOR: 10100101 XOR 10110111 и получаем шифротекст 00010010.

[1] Одноразовые блокноты обсуждались в разделе 19.3.

- Для расшифровки шифротекста мы выполняем над ним и блокнотом операцию XOR: 00010010 XOR 10110111 и получаем исходное сообщение 10100101.

Вне сферы криптографии эта операция применяется для решения задач в случае использования минимального объема памяти. Один из популярных примеров таких задач — нахождение единственного неповторяющегося элемента в списке повторяющихся элементов. Для этого мы выполняем операцию XOR над всеми значениями в списке, в результате чего все значения, кроме уникального, обнуляются[1] (поскольку при одинаковых значениях соответствующих битов исходных переменных операция XOR устанавливает значение бита результата в 0). Еще один пример — обмен значений двух переменных без использования временной переменной: let $x = x$ XOR y, $y = y$ XOR x, а затем $x = x$ XOR y[2].

---

[1] Разумеется, для того, чтобы это сработало, количество повторяющихся значений должно быть четным.

[2] Первая операция XOR дает нам содержимое одной переменной — x или y. Вторая дает содержимое одной или трех переменных — x, или y, или y, то есть x. Третья операция XOR дает нам содержимое одной или трех переменных — x, или y, или x, то есть y. Однако, как правило, для решения такой задачи лучше использовать временную переменную.

25 Операционные системы

Операционную систему можно рассматривать в качестве еще одного уровня абстракции, отделяющего пользователя (или программиста) от физической структуры компьютера. Вместо того чтобы напрямую управлять оборудованием, код приложения может просто обратиться к API ОС, чтобы задействовать те или иные системные функции.

25.1. Управление процессами

Как правило, пользователи предпочитают делать несколько дел одновременно[1]. Подобно человеческому мозгу, процессор компьютера не может решать больше одной задачи за раз[2], но способен переключаться между разными задачами, чтобы создать видимость одновременного выполнения множества действий.

---

[1] Этот факт требует проверки.

[2] Современные процессоры, как правило, имеют несколько ядер, каждое из которых является полноценным процессором, способным решать задачи независимо от остальных.

Это переключение позволяет нескольким задачам совместно использовать системные ресурсы (в том числе процессоры и память), но требует дополнительной работы, связанной с переключением контекстов[1] и обеспечением того, чтобы разные *процессы* (выполнение задачи) не мешали друг другу.

25.1.1. Многозадачность

Многозадачная система работает в режиме разделения времени, при котором процессорное время распределяется между потоками различных процессов. Современные компьютеры обычно используют так называемую вытесняющую многозадачность, при которой ядро прерывает выполняющийся процесс по истечении некоего кванта времени или при получении сигнала от задачи с более высоким приоритетом. Поскольку распределение времени контролирует операционная система, каждый процесс в конечном итоге гарантированно получает квант времени ЦП. Тем не менее любой процесс может когда угодно потерять управление. Кроме того, может возникнуть ситуация взаимной блокировки, при которой несколько процессов находятся в состоянии ожидания ресурсов, занятых остальными, и ни один не может продолжить свое выполнение.

---

[1] Контекст процесса — это его текущее состояние, включающее содержимое всех регистров, которое потребуется после возобновления его выполнения. Данная информация сохраняется в блоке управления процессом (Process Control Block, PCB) и используется для восстановления его состояния при перезапуске.

Пример

Для иллюстрации данной проблемы часто используется задача об обедающих философах, в которой пять безмолвных (то есть не разговаривающих друг с другом) философов сидят за круглым столом. Перед каждым из них стоит тарелка спагетти, и между каждой парой сидящих рядом философов лежит по одной вилке. Каждый из философов может либо есть, либо размышлять, причем есть он может, лишь взяв две вилки, лежащие с обеих сторон. Философ, у которого есть только одна вилка, не ест, а ждет, пока ему не достанется вторая вилка; таким образом, если каждый философ возьмет вилку слева от себя, все они будут голодать, ожидая возвращения второй вилки.

При использовании другого типа многозадачности, называемой совместной или кооперативной, каждый процесс сам контролирует свое выполнение и добровольно отдает процессорное время другим задачам либо периодически, либо при логической блокировке (например, при ожидании завершения ввода/вывода). Совместная многозадачность предполагает, что все процессы регулярно передают управление другим, и является непрактичной для большинства систем, поскольку существует вероятность того, что один из процессов не отдаст ресурсы, из-за чего остальные приложения не смогут продолжить работу. Тем не менее этот тип многозадачности часто применяется во встроенных системах[1].

[1] Подробнее о встроенных системах мы поговорим в главе 27.

25.1.2. Многопроцессорность и многопоточность

Если компьютер предусматривает несколько процессоров, то на нем могут одновременно выполняться несколько процессов. Кроме того, один процесс может состоять из нескольких потоков, выполняющихся на разных процессорах.

Многопоточность позволяет ускорить отклик программы, оптимизировать использование системных ресурсов и обеспечить возможность распараллеливания. Тем не менее процесс написания многопоточных приложений может быть более сложным, поскольку требует тщательной синхронизации потоков в целях предотвращения состояний гонки[1].

25.1.3. Многопоточность и состояния гонки

Состояния гонки возникают тогда, когда два или более потока пытаются одновременно изменить общие данные. Результат этого изменения зависит от порядка, в котором эти потоки обращаются к данным. Как правило, происходит нечто наподобие этого.

1. Поток A встречает оператор `if` и проверяет его условие.

---

[1] Я по собственному опыту знаю, как сложно бывает решить проблему, связанную с состояниями гонки в многопоточных программах.

2. Поток B выполняет код, который должен изменить результат условного выражения.
3. Поток A продолжает выполнение тела оператора `if`, даже если это уже неактуально.

Для того чтобы предотвратить возникновение этого гейзенбага[1], необходимо выполнить одно из следующих действий:

- заблокировать общие данные перед обращением к ним любого из потоков и тем самым предотвратить их изменение во время использования[2];
- превратить всю единицу работы (условное выражение и тело оператора `if`) в атомарную операцию, то есть в операцию, которую нельзя прервать[3].

25.2. Управление хранилищем

Ранее мы обсуждали[4], как происходит выделение памяти в стеке или в куче. Этим процессом управляет операционная система, распределяя память между процессами и решая, когда ее следует освободить.

---

1 Ошибка, которая исчезает или меняет свои свойства, когда вы пытаетесь ее обнаружить.

2 В языке C# существует класс *Interlocked*, позволяющий выполнять атомарные операции.

3 В языке Java существует ключевое слово *synchronized*, которое блокирует доступ к блоку кода, если тот уже используется другим потоком.

4 См. раздел 22.3.

25.2.1. Логические и физические адреса

Предположим, некоему вновь созданному процессу разрешено использовать 4 Гбайт памяти. Если мы предоставим этому процессу прямой доступ к памяти, то столкнемся с несколькими проблемами.

- Как мы можем гарантировать, что этот процесс не обратится к памяти за пределами выделенного блока?
- Если у нас нет непрерывного блока оперативной памяти размером 4 Гбайт, то как мы можем его создать?
- Если процесс приостановлен, а выделенная для него оперативная память используется другим процессом, то нужно ли нам ждать освобождения этого конкретного блока памяти перед возобновлением данного процесса?

Решение заключается в замене физической памяти логической. Вместо того чтобы предоставлять каждому процессу прямой доступ к диапазону физических адресов, мы назначаем ему блок логических адресов, которые ЦП может затем отобразить в физические. Благодаря такому подходу процессу не требуется знать о том, где в действительности хранятся используемые им данные[1]. Их можно как угодно перемещать, не опасаясь того, что процесс (намеренно или случайно)

[1] Подробнее о динамическом распределении памяти вы можете узнать в разделе 2.5 книги Дональда Кнута «Искусство программирования».

получит доступ к памяти за пределами выделенного ему блока.

Редко используемые данные можно переместить в область подкачки, при этом процесс будет воспринимать их так, будто они находятся в оперативной памяти. Если нескольким процессам требуется копия одних и тех же инструкций, то одна и та же физическая память может отображаться в несколько логических адресов.

25.2.2. Пейджинг и свопинг

Объем данных растет так, чтобы
заполнить все место на носителе.

Закон информации Паркинсона

Допустим, объем оперативной памяти системы не позволяет одновременно хранить данные для всех запущенных процессов, что тогда? Один из вариантов решения этой задачи — свопинг: процесс загружается в память целиком, выполняется до тех пор, пока управление не перейдет к другому процессу, а затем выгружается обратно на диск, чтобы освободить память для следующего процесса. В качестве альтернативы в память могут загружаться только те части программы, которые требуются в настоящий момент, а остальные при этом остаются на диске до тех пор, пока в них не возникнет необходимость.

Пейджинг, или подкачка страниц, предполагает разбиение виртуального адресного пространства на бло-

ки фиксированного размера, называемые страницами, которые отображаются в страничные кадры или фреймы (того же размера) в памяти. Всякий раз, когда программа обращается к памяти, мы индексируем таблицу страниц, чтобы выяснить местоположение соответствующего страничного кадра. Если данная страница в настоящее время не загружена, то возникает ее отказ, после чего мы загружаем соответствующий кадр (при необходимости заменяя другую страницу, находящуюся в памяти в настоящее время).

Часто процесс работает с небольшим набором доступных ему страниц[1]. Если это *рабочее множество* находится в памяти, то процесс может выполняться быстро, несмотря на небольшое количество загруженных страниц. Если же в каждый момент времени загруженной оказывается лишь часть рабочего множества, то результатом будут частые отказы страниц, поскольку процесс будет постоянно сталкиваться с отсутствием в памяти нужной ему страницы (такое состояние называется *пробуксовкой*). Чтобы этого избежать, мы можем разрешить запуск процесса только при загрузке в память всего рабочего множества[2].

Если памяти недостаточно для хранения рабочих множеств всех запущенных процессов, то некоторые про-

---

[1] Речь идет о так называемой локальности ссылок, упомянутой в разделе 2.2.

[2] Чтобы не допустить отказов страниц, мы можем загрузить все рабочее множество, прежде чем процессу будет позволено возобновить работу. Этот подход называется опережающей подкачкой страниц (prepaging).

цессы должны быть выгружены во избежание пробуксовки.

Практические соображения, связанные с подкачкой страниц

Как можно освободить место для загрузки страницы, когда его оказывается недостаточно? Идеальным решением была бы замена страницы, которая нам больше не понадобится. Однако, поскольку задача определения таких страниц, как правило, является трудновыполнимой, вместо этого мы можем заменять страницы, не использовавшиеся в последнее время.

На каждой странице установлены флаги, сообщающие о факте ее чтения или изменения тем или иным процессом, причем бит чтения периодически очищается. Когда нам нужно освободить место, мы удаляем случайным образом выбранную страницу, к которой уже какое-то время не обращались, исходя из предположения о том, что она не потребуется нам в ближайшем будущем[1]. Если страница была изменена, то выгружаем эти изменения на диск.

[1] На самом деле существует четыре категории страниц: те,которые не считывались и не изменялись; изменялись, ноне считывались; считывались, но не изменялись; и считывались, и изменялись. Существование страницы, которая изменялась, но не считывалась, обусловлено тем, что бит чтения периодически очищается, а бит изменения (сообщающий о необходимости выгрузки страницы на диск) — нет. Когда нам нужно удалить страницу, мы выбираем ее из нижнего непустого класса, поэтому измененная страница, к которой давно не обращались, будет удалена раньше, чем неизмененная страница, к которой недавно был получен доступ.

25.3. Ввод/вывод

Одна из важных функций операционной системы — унифицированное представление устройств ввода/вывода. С точки зрения процесса не имеет значения, откуда считываются данные (из памяти, с диска или из сети) и куда выводятся (на экран, в файл или на принтер). Операционная система предоставляет интерфейс, позволяющий процессу считывать данные из стандартного потока ввода и записывать их в стандартный поток вывода.

С точки зрения операционной системы ввод и вывод данных происходит чрезвычайно медленно по сравнению со скоростью работы процессора[1]. Для простой системы, выполняющей одно действие за раз, может быть достаточно того, чтобы процессор запросил выполнение операции ввода/вывода, а затем находился в режиме *активного ожидания*[2] до тех пор, пока соответствующее устройство не закончит обработку данных. В большинстве случаев предпочтительным является переключение контекста и возобновление прерванного процесса после завершения операции ввода/вывода. Один из способов возврата к исходно-

---

[1] Существуют исключения, например серверы Oracle Exadata, однако мало кто из нас готов потратить на сервер сотни тысяч долларов (или даже больше).

[2] В режиме активного ожидания (busy waiting) ресурсы ЦП бесполезно тратятся на многократную проверку некоего условия вплоть до его выполнения; это также называется вращением процесса в пустом цикле (spinning).

му состоянию заключается в том, чтобы разрешить аппаратному обеспечению посылать процессору запросы на прерывание всякий раз, когда оно готово к обработке очередного символа, однако на это тратится достаточно много времени. Более предпочтителен вариант использования контроллера прямого доступа к памяти (direct memory access, DMA). В таком случае ЦП просто инициирует передачу данных, делегирует ее контроллеру DMA и прерывается только после завершения этого процесса.

25.4. Безопасность

В главе 18 мы обсудили три составляющие информационной безопасности: конфиденциальность, целостность и доступность. Когда речь идет об операционных системах, мы можем ожидать от ОС гарантии того, что без соответствующего разрешения к данным пользователя или процесса не получит доступ другой пользователь или процесс, что они не подвергнутся несанкционированному изменению и будут доступны по запросу.

В предыдущих разделах мы говорили о том, как сделать так, чтобы процесс мог получить доступ только к ячейкам памяти, выделенным для него операционной системой, и не мог монополизировать ЦП. Обычно мы хотим, чтобы каждый процесс имел доступ лишь к тем ресурсам, которые ему разрешено использовать, причем исключительно допустимыми способами.

Для этого требуется, чтобы процессы обращались к ресурсам путем отправки запроса в операционную систему, а не контролировали оборудование напрямую. ОС, в свою очередь, должна проверить уместность полученного запроса. Для этого она может использовать списки управления доступом (access control lists, ACL), которые определяют, какие процессы могут задействовать ресурс и как именно, или перечни возможностей, определяющие права доступа, предоставленные тому или иному процессу.

В целях предоставления гарантий того, что процессы не смогут обойти меры контроля доступа, крайне важно избежать повреждения самой операционной системы. Надежные системы предполагают создание доверенной вычислительной базы (trusted computing base, TCB), состоящей из аппаратных и программных компонентов, отвечающих за поддержание безопасности. Размер TCB должен быть минимальным, чтобы можно было периодически проверять корректность функционирования ее компонентов; все запросы на получение доступа к системным ресурсам должны проверяться монитором обращений, выполняющим функцию барьера между доверенной и недоверенной частями системы.

В более общем смысле в основе компьютерной безопасности лежит принцип наименьших привилегий, согласно которому каждый субъект (будь то процесс, пользователь или программа) должен иметь доступ только к тем ресурсам, которые минимально необходимы для выполнения его цели. Благодаря этому

ядро[1] остается максимально компактным и изолированным от остальных частей системы.

Программы выполняются как процессы на уровне пользователя, которые не имеют разрешения на прямой доступ к ресурсам. Когда процессу требуется доступ, он выполняет системный вызов через прерывание, передающее управление ядру. После того как права процесса на доступ к запрошенному ресурсу будут проверены, ядро его предоставляет.

[1] Ядро — это часть ОС, которая всегда находится в оперативной памяти и имеет полный доступ ко всем системным ресурсам.

26 Распределенные системы

Представьте процесс создания популярной поисковой системы. По мере ее развития могут возникнуть проблемы, связанные с такими аспектами, как:

- масштабируемость — система должна обрабатывать постоянно растущий объем данных;
- производительность — работа системы не должна замедляться по мере увеличения нагрузки;
- доступность — система всегда должна быть доступна; ее простои недопустимы.

Мы можем до определенной степени повысить производительность и масштабируемость, добавив дополнительное оборудование, однако рано или поздно достигнем пределов возможностей одной системы[1]. Решение заключается в разделении работы между несколькими недорогими компьютерами. Это обеспечивает масштабируемость (по мере увеличения нагрузки можно добавить больше серверов), производительность (на каждый

[1] Возможности недорогого компьютера можно исчерпать очень быстро.

сервер приходится ограниченная часть нагрузки) и доступность (резервные серверы позволяют продолжить работу в случае отказа одного из них). Даже небольшая система может выиграть от разделения компонентов, например, на веб-сервер, сервер базы данных и т. д.

Масштабирование Google

В 1998 году, когда компания Google начала свою деятельность, у ее соучредителей было всего четыре компьютера и несколько сотен гигабайтов дискового пространства. Сейчас ее работу обеспечивают более двух миллионов серверов в нескольких центрах обработки данных по всему миру.

26.1. Ложные допущения относительно распределенных вычислений

Распределенные вычисления имеют свои сложности. Л. Питер Дойч в соавторстве с другими специалистами[1] составил список ложных допущений, которые часто делают программисты, приступающие к работе над распределенными системами[2].

---

[1] Краткое изложение этой истории вы можете найти по адресу https://web.archive.org/web/20070811082651/http://java.syscon.com/read/38665.htm.

[2] Дополнительную информацию и примеры вы можете найти в этой статье: https://arnon.me/wp-content/uploads/Files/fallacies.pdf.

1. **Допущение:** работа сети стабильна.

   Реальность: сеть (или ее критически важные компоненты) отключается в самое неподходящее время.

2. **Допущение:** задержка равна нулю.

   Реальность: пакеты задерживаются.

> **Исторический экскурс: латентность**
>
> В 1990-х годах сотрудники одного из факультетов Университета Северной Каролины выяснили, что не могут отправить электронную почту на расстояние более 500 миль. Системный администратор решил, что этого не может быть, но тестирование подтвердило существование данной проблемы. В конце концов он понял, что после обновления почтовый сервер перестал понимать файл конфигурации и отключался по тайм-ауту, спустя ровно столько времени, сколько было необходимо для того, чтобы сообщение было получено системой, находящейся на расстоянии 500 миль.

3. **Допущение:** пропускная способность безгранична.

   Реальность: в каналах передачи данных будут возникать перегрузки.

4. **Допущение:** сеть безопасна.

   Реальность: хакеры попытаются украсть ваши данные.

5. **Допущение:** топология не меняется.

   Реальность: узлы сети будут добавляться и/или удаляться.

6. **Допущение:** есть только один администратор.

Реальность: разные администраторы внедряют разные политики.

7. **Допущение:** затраты на перемещение данных равны нулю.

   Реальность: перемещение данных между узлами требует затрат.

> **Исторический экскурс: безопасность**
>
> В 2013 году после публикации документов Эдвардом Сноуденом выяснилось, что Агентство национальной безопасности США перехватывало информацию, передаваемую между центрами обработки данных (ЦОД), принадлежащими Google (и Yahoo!). Ответом компании Google стало шифрование каналов связи между своими ЦОД.
>
> В 2014 году несколько инженеров по безопасности независимо друг от друга обнаружили ошибку (названную Heartbleed) в криптографическом программном обеспечении OpenSSL, которая позволяет злоумышленнику украсть информацию, как правило защищаемую с помощью протоколов SSL/TLS. Согласно отчету Shodan Heartbleed Report, по состоянию на ноябрь 2019 года 91 063 общедоступных веб-сервера по-прежнему были уязвимы для этой ошибки.

8. **Допущение:** сеть однородна.

   Реальность: разные части системы используют разное оборудование.

   К уже перечисленным я могу добавить следующее.

9. **Допущение:** мы можем восстановить порядок событий.

   Реальность: серверы могут спорить относительно того, что произошло раньше.

Все эти проблемы мы обсудим в следующих разделах.

Исторический экскурс: затраты на перемещение данных

В 2013 году клиенты интернет-провайдеров Comcast и Verizon столкнулись с ухудшением качества потокового видео, в результате чего многие из них отказались от подписки на сервис Netflix. В начале 2014 года компания Netflix заключила соглашения с обоими провайдерами относительно прямого подключения к их сетям.

26.2. Коммуникация

Процессы, образующие распределенную систему, взаимодействуют путем передачи сообщений. Любые два процесса могут быть запущены на одной машине или на двух серверах, находящихся на расстоянии тысяч километров друг от друга. Разработчики системы должны ответить на следующие вопросы.

- Следует ли нам требовать подтверждения получения сообщений, и если да, то как долго мы должны его ожидать?
- Хотим ли мы получать каждое сообщение «хотя бы один раз» или «только один раз»?

- Что мы будем делать, если некоторые элементы системы станут недоступными?
- Как мы будем реагировать на добавление или удаление серверов?

Проще говоря, распределенная система должна предусматривать план действий на случай неспособности обеспечения своевременной, надежной или безопасной доставки сообщений, а также на случай принципиальной невозможности их доставки.

26.3. Синхронизация и согласованность

Рассмотрим, что происходит, когда данные копируются (реплицируются) на несколько географически удаленных друг от друга серверов. Благодаря этому сокращается латентность (ближайший сервер в среднем всегда будет находиться ближе, чем единственный центральный сервер) и обеспечивается избыточность (если один сервер выйдет из строя, то остальные попрежнему будут доступны). В связи с этим также возникает вопрос поддержания синхронизации данных между серверами.

Если изменения данных допускает лишь единственный сервер (а все остальные доступны только для чтения), то он представляет собой слабое звено. Если все серверы допускают как чтение, так и запись данных, то существует вероятность того, что несколько серверов обновятся приблизительно в одно и то же

время, в результате чего будут содержать противоречивую информацию. Даже если данные изменяются только на одном сервере, другие серверы будут получать обновления в разное время, поэтому пользователь может увидеть разные значения в зависимости от того, к какому серверу обращается.

На случай получения серверами конфликтующих данных при обновлении необходимо предусмотреть политику, определяющую те данные, которые следует принять. Из-за отсутствия универсальных часов определение последовательности обновлений может оказаться невозможным, и различные серверы могут получать их в разном порядке.

Почему не существует универсальных часов?

Работать с распределенными системами было бы намного проще, если бы мы точно знали время наступления того или иного события. Однако каждая система предусматривает собственные часы, скорость работы которых может слегка различаться. Мы можем синхронизировать все часы с одними доверенными часами, но, поскольку время, необходимое для получения сообщения от доверенных часов, может варьироваться, наши распределенные системы по-прежнему не будут идеально синхронизированы.

27 Встроенные системы

Обычный человек (не программист) под словом «компьютер» подразумевает компьютер общего назначения, то есть устройство, на котором можно запускать разные программы. Однако компьютеры общего назначения составляют лишь небольшую часть[1] всех компьютеров. Большинство составляют встроенные системы, специально созданные для выполнения конкретной задачи. Эти крошечные компьютеры управляют всем, начиная от вашей микроволновой печи и заканчивая зеркалами заднего вида и фарами некоторых автомобилей[2]. Программирование встроенных систем позволяет сделать так, чтобы ваше программное обеспечение напрямую управляло оборудованием.

В отличие от разработки большинства приложений общего назначения, при программировании встроенных систем вы вынуждены иметь дело с более ограничен-

---

1 По некоторым оценкам, их процент приближается к нулю.

2 С примерами вы можете ознакомиться по адресу https://patents.google.com/patent/US8625815B2/en и https://www.cs.cmu.edu/smartheadlight/.

ным количеством ресурсов. Встроенная система, как правило, отличается медленным процессором, ограниченным количеством энергии, памяти и периферийных устройств. Возможно, ей придется работать бесконечно без какого-либо обслуживания (например, если система встроена в космический спутник или спущена в Марианскую впадину). Она должна уметь либо восстанавливать работу после сбоев, либо полностью отказывать при первом же возникновении ошибки. Она должна выдавать либо детерминированную реакцию (то есть всегда одинаково реагировать на те или иные входные данные), либо реагировать в пределах некоего временного периода[1]. Проектирование встроенной системы обычно предполагает нахождение компромисса между различными ограничениями.

Одна из сложностей программирования встроенных систем заключается в том, что их разработка, как правило, не может выполняться на том же оборудовании, на котором им предстоит работать. Встроенная система обычно не имеет дополнительных возможностей для запуска отладчика и, вероятно, не может напрямую управлять монитором или типичными устройствами ввода, поэтому мы программируем их с помощью компьютера общего назначения.

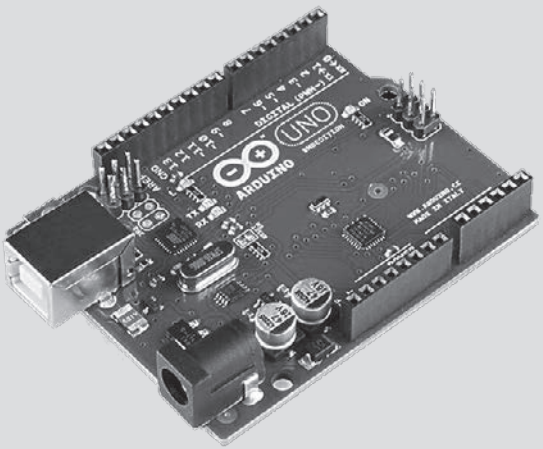
Ограниченность ресурсов повышает полезность методов, описанных в главе 24, позволяющих увеличить скорость работы программы за счет некоторой потери в ясности.

---

[1] Многие встроенные системы используют ОС реального времени, которые должны обрабатывать данные в рамках требуемых временных ограничений. Чтобы гарантировать их производительность, при разработке приоритет отдается скорости и предсказуемости.

Знакомство со встроенными системами

На протяжении длительного времени стандартом в области разработки встроенных систем являлась физическая вычислительная платформа с открытым исходным кодом Arduino[1]. Несмотря на то что эта платформа имеет низкую стоимость (плата начального уровня стоит около 20 долларов США) и подходит для студентов и любителей[2], она представляет собой эффективный инструмент для создания реальных продуктов[3].



[1] Плата Arduino Uno R3. Изображение предоставлено SparkFun по лицензии Creative Commons.

[2] Всевозможные примеры проектов, от игр до роботов, вы можете найти на сайте https://create.arduino.cc/projecthub.

[3] О различных областях профессионального использования платформы Arduino вы можете узнать на сайте https://www.arduino.cc/pro/verticals.

28 Сети и Интернет

Многие из современных программ предназначены для работы в компьютерной сети (как правило, в Интернете). Вне зависимости от того, идет ли речь о модели SaaS (software as a service — программное обеспечение как услуга), MMORPG-играх, сетевых принтерах или камерах видеонаблюдения с функцией автоматического резервного копирования, работа наших программ и устройств зависит от их способности быстро, надежно и безопасно обмениваться данными с другими устройствами, которые могут находиться в том же помещении или в другой стране.

Существует две широко используемые модели описания процесса построения сетей, разделяющие функциональность на различные уровни, которые можно настроить независимо от других. Например, вашему почтовому клиенту все равно, как электронное письмо достигает места назначения. Маршрутизатору, пересылающему пакеты данных, нет дела до их содержимого. Кабель Ethernet предназначен только для передачи битов из конца в конец. Каждый уровень имеет собственные *протоколы*, то есть правила, определяющие порядок обработки данных на этом уровне.

Модель TCP/IP состоит из четырех уровней, построенных на базе протоколов TCP/IP, лежащих в основе работы сети Интернет. Более универсальная теоретическая модель OSI вместо стандартов предлагает семь уровней протоколов и рекомендации (рис. 28.1).

28.1. Уровни протоколов

28.1.1. Прикладной уровень

Прикладной уровень, который является верхним в стеке протоколов, обеспечивает межпроцессное взаимодействие, давая возможность приложениям обмениваться данными по любой имеющейся сети. К этому уровню принадлежат такие протоколы, как HTTP, FTP и SMTP. Нижние уровни в основном рассматриваются в качестве «черного ящика», обеспечивающего соединение, по которому прикладной уровень может передавать данные.

Этот уровень модели TCP/IP разделен на три уровня в модели OSI:

- уровень 7, или прикладной, — то, с чем непосредственно взаимодействует пользователь (например, браузер);
- уровень 6, или уровень представления, отвечает за форматирование данных прикладного уровня, то есть за их подготовку к передаче по сети (а также преобразует полученные из сети данные в формат, понятный приложениям). Например, на этом уровне данные могут быть зашифрованы или расшифрованы;

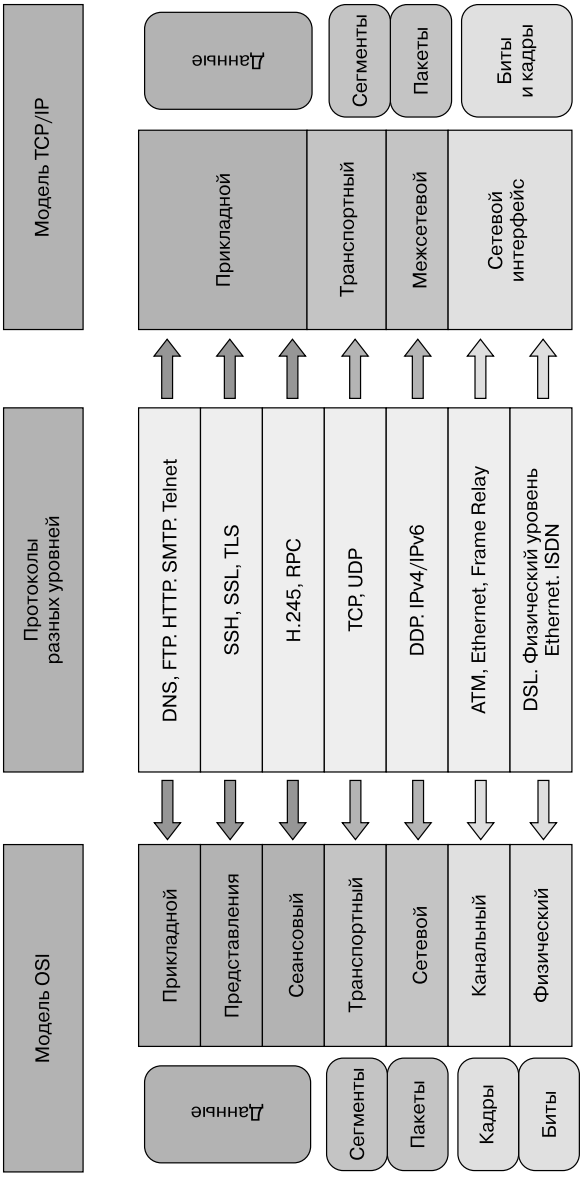


Рис. 28.1. Сравнение моделей OSI и TCP/IP

- уровень 5, или сеансовый, отвечает за поддержание сеанса связи между двумя взаимодействующими устройствами.

28.1.2. Транспортный уровень

Транспортный уровень предоставляет механизм передачи сообщений, не зависящий ни от передаваемых данных, ни от логистических аспектов их перемещения по сети. Этот уровень обеспечивает передачу данных либо с установкой соединения (TCP), либо без него (UDP), а также отвечает за качество обслуживания (то есть за то, что происходит с потерянными или неправильно обработанными пакетами данных).

28.1.3. Межсетевой или сетевой уровень

Межсетевой (модель TCP/IP) или сетевой (модель OSI) уровень отвечает за маршрутизацию пакетов данных от одного устройства к другому. На этом уровне для передачи данных используется интернет-протокол (IPv4 или IPv6).

28.1.4. Канальный уровень

На канальном уровне работают все хосты (устройства), к которым можно получить доступ без прохождения через маршрутизатор. Этот уровень отвечает за перемещение пакетов между хостами, подключенными к одному и тому же сетевому сегменту. В модели OSI он разделен на два подуровня:

- уровень 2, или уровень канала передачи данных, обеспечивает передачу данных между узлами и исправляет ошибки, которые возникли на физическом уровне;
- уровень 1, или физический, имеет отношение к проводам, напряжению и параметрам радиосигналов, обеспечивающих передачу данных.

28.2. Протоколы TCP/IP и UDP

Как говорилось выше, транспортный уровень обеспечивает передачу данных либо с установкой соединения, либо без нее. При использовании ориентированного на соединение протокола сеанс связи между двумя устройствами устанавливается до передачи данных. Для поддержания этого сеанса задействованные устройства должны хранить информацию о его состоянии, чтобы соединение оставалось открытым. Задействуя протокол без установки соединения, отправитель просто отправляет сообщение, как только оно будет готово, не дожидаясь никаких подтверждений.

Протоколы, ориентированные на соединение, отличаются надежностью и гарантируют доставку данных (или по крайней мере сообщают отправителю, если они не могут быть доставлены). Мы используем эти протоколы, когда хотим быть уверенными в том, что данные не потеряются и будут доставлены в правильном порядке.

Протоколы без установки соединения не гарантируют, что данные не будут потеряны, искажены, продублированы или доставлены в неверном порядке (хотя не-

которые протоколы более высокого уровня дают такие гарантии). Благодаря своему минимальному функционалу протоколы без установки соединения отличаются низкими накладными расходами и используются в тех случаях, когда пропускная способность оказывается важнее надежности.

Протокол пользовательских датаграмм (User Datagram Protocol, UDP) и межсетевой протокол (Internet Protocol, IP) относятся к протоколам без установки соединения, а протокол управления передачей (Transmission Control Protocol, TCP) ориентирован на соединение. Мы, как правило, говорим о TCP/IP[1].

28.3. Доставка сообщения

Как говорилось выше, на сетевом уровне происходит маршрутизация пакетов от источника к месту назначения. Межсетевой протокол определяет способ доставки пакетов адресату.

Допустим, некоторые данные должны быть отправлены через Интернет с помощью TCP/IP. На каждом уровне этого стека протоколов передаваемые данные называются *блоком данных протокола*, или БДП (Protocol Data Unit, PDU), но форматируются по-разному.

С точки зрения приложения отправитель передает данные, и они доставляются получателю. На текущем

[1] Протоколы HTTP/1.1 и HTTP/2 работают поверх TCP; HTTP/3 использует протокол QUIC, основанный на UDP, но обеспечивает аналогичную TCP функциональность с уменьшенной латентностью.

этапе БДП представляет собой полное сообщение (файл/электронное письмо и т. д.), подлежащее передаче.

Транспортный уровень устанавливает логическое соединение между двумя процессами, выполняемыми на разных хостах; притом фактическое местонахождение этих хостов и физические характеристики канала связи между ними являются несущественными. На текущем уровне сообщение разбивается на сегменты, размер которых соответствует требованиям расположенного ниже сетевого уровня. Каждый сегмент начинается с заголовка, содержащего различную информацию, в первую очередь номер порта, на который должен быть доставлен данный сегмент.

В то время как транспортный уровень обеспечивает логическое соединение между двумя процессами, сетевой обеспечивает логическое соединение между двумя хостами (на каждом из которых может выполняться множество процессов). На этом уровне сегмент транспортного уровня упаковывается в пакет, включающий различные данные, необходимые для его доставки в место назначения, в том числе IP-адреса отправляющего и принимающего хостов. В то время как сегмент обрабатывается только в пункте отправки и в пункте назначения, пакет обрабатывается маршрутизаторами вдоль всего пути следования.

Задача маршрутизатора состоит в обработке пакетов, поступающих на его входящие интерфейсы, и в их перенаправлении на соответствующие исходящие интерфейсы. Путь от источника до пункта назначения вычисляется с помощью специальных алгоритмов

маршрутизации. От используемой сетевой модели зависит, могут ли различные пакеты, составляющие сообщение, быть переданы по одному и тому же маршруту и будут ли доставлены в том же порядке, в котором были отправлены.

Наконец, канальный уровень отвечает за перемещение данных по одному каналу, то есть между двумя узлами (где узлом может быть маршрутизатор или хост). На этом уровне пакет инкапсулируется в кадр, который физически передается от одного узла к другому.

28.4. Алгоритмы маршрутизации

На сетевом уровне алгоритмы маршрутизации используются для определения наилучших путей от отправителя до получателя. Наилучшим считается путь, имеющий наименьшую стоимость[1], учитывая проблемы политики (например, когда компания А отказывается пересылать пакеты, поступающие от компании Б).

Сеть можно представить в виде взвешенного графа[2], вершинами которого являются точки переадресации пакетов к месту назначения, а веса ребер определяют стоимость использования соответствующего канала связи[3].

---

1 См. раздел 6.5 «Кратчайшие пути».

2 Взвешенные графы были рассмотрены в разделе 4.8.

3 Под стоимостью здесь может подразумеваться множество вещей, от количества времени, необходимого для передачи данных по этому каналу связи, до стоимости его использования в денежном выражении.

Алгоритмы маршрутизации классифицируются по нескольким направлениям.

- Они могут быть *централизованными*, то есть обладающими всей информацией о сети и вычисляющими полный путь, или *децентрализованными*, то есть предполагающими то, что каждый узел знает только о своих непосредственных соседях (соседних вершинах графа), но не обо всем пути, который предстоит пройти.
- Они могут быть *статическими*, если маршруты практически не меняются (и часто конфигурируются вручную), или *динамическими*, если маршруты меняются в зависимости от изменения параметров сетевого трафика и топологии сети[1].
- Они могут быть *адаптивными*, если стоимость изменяется в зависимости от нагрузки на сеть, или *неадаптивными*, если это не так[2].

---

[1] Динамические алгоритмы адаптируются к изменениям быстрее, но могут проявлять склонность к образованию петель маршрутизации.

[2] Первые алгоритмы маршрутизации ARPAnet были адаптивными, в отличие от таких современных алгоритмов, как BGP.

29 Базы данных

Рассмотрим такое типичное приложение для управления данными, как система инвентаризации. Обычно оно должно предоставлять возможность создавать или извлекать записи, изменять их, а затем снова сохранять для последующего извлечения. Это классическое CRUD-приложение (сокр. от англ. create, read, update, delete — «создать, прочесть, обновить, удалить»). База данных — способ организации структурированной информации (данных) с целью ее хранения и извлечения.

29.1. Реляционные базы данных (РБД)

Практически все современные базы данных являются реляционными. РБД организована в виде набора таблиц, связи между которыми установлены с помощью ключей.

Например, у компании может быть таблица «Клиенты» (табл. 29.1), которая содержит информацию о каждом клиенте, в том числе ID (идентификатор) или код.

Идентификатор представляет собой первичный ключ — поле[1], которое требуется для каждой записи и является уникальным для нее.

Таблица 29.1. Таблица «Клиенты» в реляционной базе данных. Код клиента — это первичный ключ

Имя клиента	Код клиента	Адрес
Капитан Крюк	1904	Веселый Роджер
Билли Кид	1859	Форт-Самнер, штат Нью-Мексико
Мордред	1136	Камелот

Вся информация о заказах клиентов находится в таблице «Заказы» (табл. 29.2). Вместо того чтобы включать в нее информацию о клиенте (что привело бы к дублированию большого количества данных, особенно если один клиент сделал много заказов), каждая строка содержит код клиента, соответствующий коду в таблице «Клиенты». Это так называемый внешний ключ — идентификатор, который может использоваться для многих записей в таблице «Заказы», но при этом должен соответствовать только одной записи в таблице «Клиенты».

Таблица 29.2. Таблица «Заказы» в реляционной базе данных. Код заказа — это первичный ключ, а код клиента — внешний ключ

Код заказа	Код клиента	Стоимость	Товар
1	1904	$27	Крюк
2	1904	$4	Повязка на глаз
3	1136	$0	Меч (украден)

[1] Или набор полей.

Правильно спроектированная реляционная база данных обеспечивает точность, предотвращая дублирование информации, как видно в приведенном выше примере. Мы сохраняем часть информации только в соответствующей таблице и ссылаемся на нее по мере необходимости.

К наиболее распространенным системам управления реляционными базами данных (СУРБД) относятся Microsoft SQL Server, Oracle и MySQL. Как и большинство других подобных систем, они используют SQL (structured query language — язык структурированных запросов), который является стандартным языком для создания запросов к реляционным базам данных. Язык SQL — декларативный: вы описываете, какой результат хотите получить, а способ его достижения находит компьютер.

Пример

Предположим, я хочу просмотреть все заказы, сделанные в период с 15 марта по 1 апреля прошлого года. Я запрашиваю эту информацию, а затем компьютер определяет, как следует отсортировать данные в таблице «Заказы», чтобы найти искомое.

Обычно СУРБД поддерживает как язык SQL (который является стандартом ANSI), так и дополнительные проприетарные команды. Системы Microsoft (с T-SQL) и Oracle (с PL/SQL) допускают использование процедурного программирования и переменных в дополнение к обычному декларативному программированию, поддерживаемому в стандартном SQL.

29.2. Иерархические базы данных (ИБД)

ИБД организует данные в виде древовидной структуры, а не таблиц. Такие базы используются в тех случаях, когда приложению требуется быстро собрать всю информацию, имеющую отношение к конкретной записи, а не подмножество доступных данных из множества записей. Например, электронные медицинские карты часто хранятся в ИБД, что обеспечивает быстрый доступ к истории болезни пациента (рис. 29.1). Если нам понадобится создать аналитический отчет (основанный на данных о популяциях, а не об отдельных пациентах), то мы можем извлечь нужную информацию в реляционную базу данных.

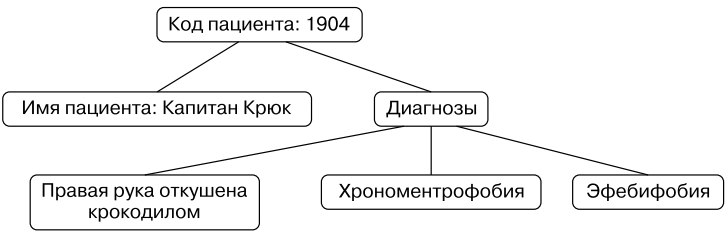


Рис. 29.1. Пример записи в иерархической базе данных пациентов

Часть IX
Углубленные темы

30 Основная теорема о рекуррентных соотношениях

В разделе 1.7 мы говорили об оценке времени выполнения простых алгоритмов. Здесь мы поговорим об оценке времени выполнения рекурсивных алгоритмов.

Рассмотрим алгоритм «разделяй и властвуй», разделяющий исходную задачу на несколько независимых подзадач, которые решаются рекурсивно, а решения объединяются для получения решения исходной задачи.

Время выполнения этого алгоритма можно описать с помощью трех параметров:

- a — количество подзадач, на которые будет разделена исходная задача;
- n / b — размер подзадач, где b — целое число больше единицы. Предполагается, что размер каждой из подзадач не будет превышать это значение;
- $f(n)$ — функция, определяющая количество времени, необходимое для разделения задачи на подзадачи и объединения результатов их решения.

Основная теорема (Master Theorem) применяется к алгоритмам, время выполнения которых может быть выражено следующим образом:

$$T(n) = aT(n / b) + f(n),$$

где $T(n)$ — общее время выполнения (остальные параметры описаны выше).

В зависимости от значения $f(n)$ мы можем получить оценку времени выполнения для трех случаев.

Математическое предупреждение

Выражение $\log_b a$ означает логарифм числа a по основанию b.

Если $b = 2$, то это двоичный логарифм, также сокращенно обозначаемый lg. Например, $\log_2 8 = 3$.

Другими распространенными основаниями являются 10 (десятичный логарифм, log) и e (натуральный логарифм, ln).

1. Если $f(n)$ асимптотически меньше[1], чем $n^{\log_b a}$, то время выполнения алгоритма равно $\Theta(n^{\log_b a})$.
2. Если $f(n) = \Theta(n^{\log_b a})$, то общее время выполнения равно $\Theta(n^{\log_b a} \cdot \log n)$.
3. Если $f(n)$ асимптотически больше, чем $n^{\log_b a}$, то есть общее время решения всех подзадач меньше, чем время, необходимое для объединения этих решений, то время выполнения равно $\Theta(f(n))$.

---

[1] Математически это записывается в виде: $f(n) = O(n^{\log_b a - \varepsilon})$, где ε (эпсилон) — произвольное сколь угодно малое значение.

Здесь мы видим, что общее время выполнения алгоритма определяется путем сравнения объема работы, необходимой для разделения задачи и последующего объединения результатов решения подзадач, с объемом работы, необходимой для фактического решения каждой подзадачи[1].

Мы можем изобразить дерево рекурсии глубиной $\log_b n$, содержащее a^i узлов на уровне i. Тогда количество листьев составляет $a^{\log_b n} = n^{\log_b a}$. Если сравнить количество листьев с объемом работы, производимой на каждом уровне, то большее из двух значений определит решение.

В первом случае объем работы, производимой на каждом уровне, ограничен сверху значением $n^{\log_b a}$. Если допустить, что значение $f(n)$ ничтожно мало, то время выполнения асимптотически равно количеству листьев: $n^{\log_b a}$.

В третьем случае объем работы, производимой на каждом уровне, ограничен снизу значением $n^{\log_b a}$, то есть превосходит количество листьев, поэтому общее время выполнения составляет $\Theta(f(n))$.

Во втором случае количество листьев и значение $f(n)$ асимптотически равны, поэтому время выполнения определяется объемом работы, производимой на каждом уровне, умноженным на количество уровней в дереве, или $\Theta(n^{\log_b a} \cdot \log n)$.

---

[1] Более строгое доказательство этой теоремы вы можете найти в разделе 4.6 книги Introduction to Algorithms (*Cormen et al.*)

Пример: сортировка слиянием

Как говорилось в подразделе 8.3.2, сортировка слиянием предполагает разделение массива на два массива меньшего размера и их рекурсивную сортировку.

При разделении или рекомбинации массивов мы не производим никакой специальной обработки, поэтому процесс занимает $O(n)$ времени. На каждом шаге мы делим массив пополам, так что количество шагов составляет $O(\lg n)$. Перемножив эти значения, мы получаем общее время выполнения $O(n \lg n)$.

Если мы применим основную теорему, то $f(n) = \Theta(n)$. Это второй случай из описанных выше (где и a, и b равны 2), и потому время выполнения сортировки слиянием равно $\Theta(n \lg n)$.

31 Амортизированное время выполнения

Допустим, у нас есть пустой массив размером n, и мы хотим добавить в него значения. Каждая из первых n вставок занимает $O(1)$ времени. К моменту вставки $n + 1$ массив будет заполнен, и его размер придется изменить, что потребует копирования всех значений в новый, более крупный массив и займет $O(n)$ времени.

Операция вставки занимает время $O(n)$ в тех случаях, когда размер массива необходимо изменить; в остальных случаях время этой операции является постоянным. Сколько времени в среднем занимает каждая операция вставки?

Если мы будем каждый раз увеличивать размер массива на постоянную величину, то количество копируемых элементов в какой-то момент начнет доминировать над значением этой константы и операции вставки будут в среднем занимать $O(n)$ времени[1].

---

[1] Если мы каждый раз увеличиваем размер массива на c мест, то общая стоимость добавления c новых элементов составляет $n + c$. Таким образом, каждая вставка в среднем занимает $(n + c) / c$ времени, а поскольку n доминирует над c, то это время равно $O(n)$.

Однако если мы будем каждый раз удваивать размер массива, то общее время, затрачиваемое на добавление n элементов, будет равно $n + n$, а каждая вставка будет в среднем занимать $2n / n = O(1)$ времени.

Здесь важно понять, что зачастую нас интересует не время выполнения отдельной операции, а общее время, затрачиваемое на серию операций. Когда каждая дорогостоящая операция сочетается с множеством дешевых, ее стоимость можно амортизировать. Таким образом, несмотря на то, что та или иная операция может занять $O(n)$ времени, ее амортизированная стоимость может оказаться намного ниже. В описанном выше примере с массивом, если мы просто посмотрим на наихудшее время выполнения данной операции, мы придем к выводу о том, что серия из n вставок займет $O(n^2)$ времени, однако амортизационный анализ показывает, что (при условии удвоения количества элементов при изменении размера массива) фактическое значение будет равно $O(n)$.

32 Расширяющееся дерево

В разделе 5.1 мы говорили о двоичных деревьях поиска, операции над которыми в общем случае занимают время $\Theta(\lg n)$ при условии, что высота дерева остается равной $O(\lg n)$. Расширяющееся, или splay-дерево[1], — это саморегулирующееся двоичное дерево поиска, которое перестраивается так, что узлы, к которым недавно обращались, перемещаются ближе к корню, что в дальнейшем обеспечивает более быстрый доступ при сохранении средней высоты $O(\lg n)$. Таким образом, часто используемые узлы оказываются более доступными, а доступ ко всем узлам в среднем занимает время $O(\lg n)$.

34.1. Концепции

При каждом обращении к узлу x splay-дерева мы производим операцию расширения (серию вращений дерева),

---

[1] Подробнее о splay-деревьях вы можете узнать в разделе 4.3 книги Роберта Тарьяна (Robert Tarjan) Data Structure and Network Algorithms. Если вас интересует тема сбалансированных деревьев, то обратитесь к подразделу 6.2.3 книги Дональда Кнута (Donald Knuth) «Искусство программирования».

чтобы переместить этот узел в корень. Когда дерево перестает быть сбалансированным, поиск узла, находящегося ниже в иерархии, может занять $O(n)$ времени, однако при расширении происходит балансировка дерева. В результате выполнение всех основных операций занимает амортизированное время $O(\lg n)$.

Отступление

Поворот изменяет структуру двоичного дерева, обновляя родительско-дочерние отношения узлов, но не меняя порядок следования элементов. Если узел a является левым дочерним элементом b и мы производим поворот дерева так, чтобы узел b стал правым дочерним элементом a, то узел b по-прежнему больше узла a. Поэтому выполнение центрированного обхода дерева до и после поворота даст одинаковые результаты.

Существует три возможные операции, каждая из которых выполняется в зависимости от следующих факторов:

- узел x является левым или правым дочерним элементом своего родителя p;
- узел p является корнем;
- узел p является правым или левым дочерним элементом своего родителя g (прародителя x).

До тех пор пока узел x не станет корнем, мы используем указанные выше факторы для выбора одного из трех возможных вариантов поворота дерева.

32.2. Zig

Когда узел *p* является корнем, мы поворачиваем дерево по ребру между *x* и *p* (рис. 32.1).

32.3. Zig-zig

Когда узел *p* не является корнем, а *p* и *x* оба являются правыми или левыми потомками, мы поворачиваем дерево по ребру между *p* и *g*, а затем по ребру между *x* и *p* (рис. 32.2).

32.4. Zig-zag

Если не применим ни один из вышеперечисленных случаев, то мы поворачиваем дерево по ребру между *p* и *x*, а затем по ребру между *x* и *g* (рис. 32.3).

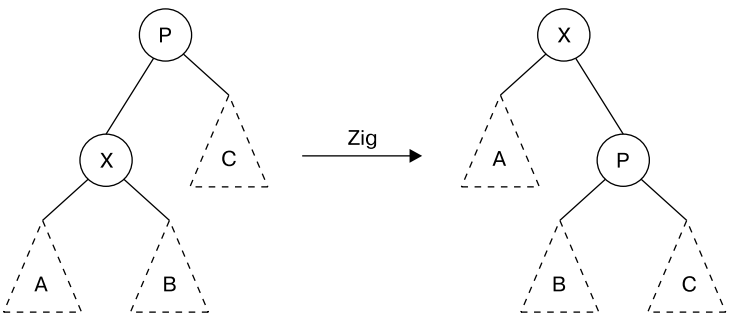


Рис. 32.1. После выполнения операции Zig узел *x* становится корнем

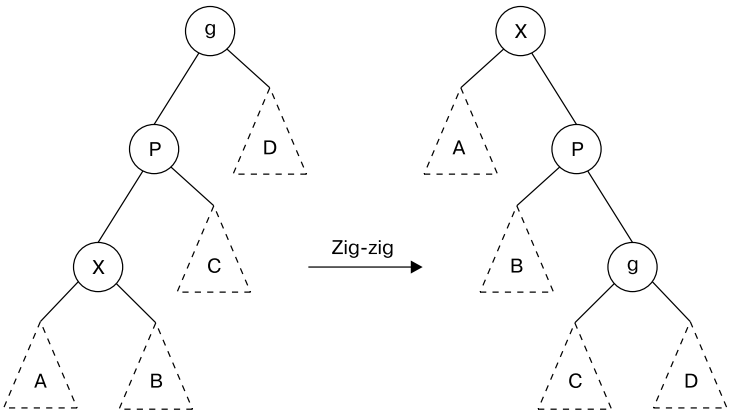


Рис. 32.2. После выполнения операции Zig-zig узел *x* становится на место своего прародителя

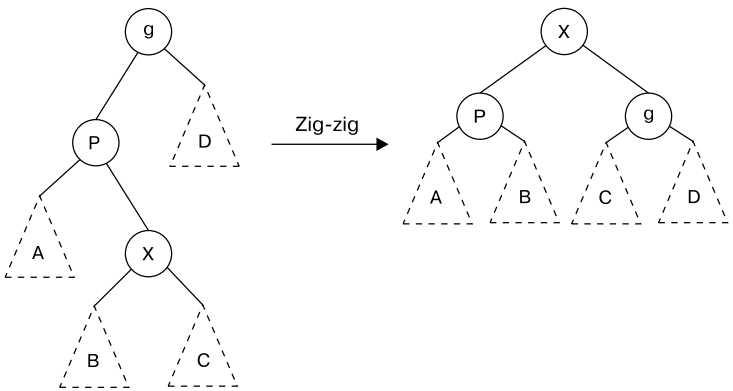


Рис. 32.3. После выполнения операции Zig-zag узел *x* снова становится на место своего прародителя

33 Декартово дерево

Декартово дерево (treap)[1] — еще одна форма самобалансирующегося двоичного дерева поиска, представляющая собой комбинацию дерева и кучи. Все узлы имеют ключи, для которых выполняется свойство двоичного дерева. Кроме того, каждый узел имеет приоритет, для которого выполняется свойство кучи.

Приоритет может присваиваться либо случайным образом, и тогда декартово дерево будет иметь форму случайного двоичного дерева (и с большой долей вероятности будет иметь высоту $O(\log n)$), либо в зависимости от частоты обращений, и в этом случае часто используемые узлы будут располагаться ближе к корню.

Поиск в декартовом дереве происходит так же, как и в любом другом двоичном дереве поиска, при этом приоритеты игнорируются. Добавление нового узла может потребовать вращения дерева для сохранения свойства кучи (рис. 33.1). Чтобы удалить узел, нужно сначала превратить его в лист, если он им еще не

---

[1] *Seidel R., Aragon C. R.* Randomized searchtrees // Algorithmica. 1996. № 16.

является. Для этого ему присваивается приоритет $-\infty$ (отрицательная бесконечность), а затем выполняются необходимые вращения. Как только узел станет листом, его можно будет удалить.

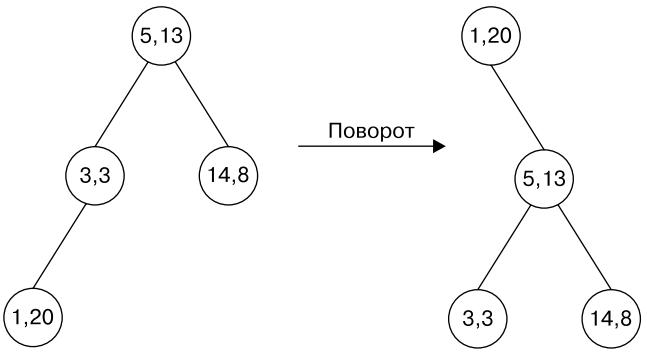


Рис. 33.1. После вставки нового узла с ключом 1 и приоритетом 20 выполняются вращения дерева, позволяющие новому узлу занять полагающееся ему место

Помимо обычных поиска/вставки/удаления, декартовы деревья поддерживают такие операции, как объединение, пересечение и разность множеств[1]. Для их выполнения нам понадобятся две вспомогательные операции.

- **Разделение.** Эта операция разделяет декартово дерево на два дерева меньшего размера: одно с ключами

[1] Описание этих операций приведено в разделе 2.6. Более подробную информацию о выполнении операций над множествами и временных рамках можно найти в работе: *Blelloch G. E., Reid-Miller M.* Fast Set Operations Using Treaps // In Proc. 10th Annual ACM SPAA, 1998.

меньше x, а другое с ключами больше x. Для этого требуется вставить узел с ключом x и максимальным приоритетом. После того как он окажется вверху в результате вращений, просто удалите его, чтобы получить два отдельных декартовых дерева из его бывших потомков.

- **Соединение.** Чтобы соединить два упорядоченных декартовых дерева (где все ключи одного дерева не превосходят ключи другого), создайте новый узел, который будет родителем корней этих деревьев. Назначьте ему соответствующий ключ (который больше ключа его левого дочернего элемента, но меньше ключа правого дочернего элемента) и минимальный приоритет. Затем выполните вращение дерева, чтобы этот узел стал листом, и удалите его, чтобы получить одно декартово дерево[1].

В плане производительности декартовы деревья аналогичны расширяющимся деревьям.

[1] Очевидно, что соединение и разделение являются обратными операциями.

34 Искусственный интеллект

Словарь Merriam-Webster определяет интеллект как способность учиться, понимать, абстрактно мыслить и применять знания для оказания воздействия на окружающую среду. Искусственный интеллект — попытка придать эти свойства машине (или по крайней мере имитировать их).

Здесь используется два подхода. Один из них предполагает имитацию мышления человека (или животного) для того, чтобы заставить компьютер решать задачи так же, как это делаем мы. Другой состоит в нахождении способов, позволяющих компьютерам решать «интеллектуальные» задачи по крайней мере так же хорошо, как люди, вне зависимости от того, как именно они рассуждают при этом. Первый подход больше фокусируется на понимании принципов работы интеллекта, а второй — на решении задач, требующих его наличия.

34.1. Типы искусственного интеллекта

Итак, каким образом искусственный интеллект (ИИ) решает задачи?

Символический ИИ предусматривает набор символов и правил манипулирования ими, совокупность которых представляет человеческие знания в декларативной форме. Такой ИИ использует правила для манипулирования символами с целью достижения желаемого результата. В 1959 году данный подход был использован для создания универсального решателя задач (General Problem Solver, GPS)[1]. Теоретически программа GPS может решить любую достаточно формализованную задачу. В данном случае задача представлена ориентированным графом, а компьютер должен найти способ достижения одной из вершин, представляющих возможный ответ. Например, программе для игры в шахматы требуется достичь конечного состояния, в котором королю соперника объявлен мат; способы перемещения между состояниями определяются правилами системы.

Универсальный решатель задач мог справляться с такими простыми головоломками, как «Ханойская башня», но решение реальных задач оказалось ему не под силу из-за проблемы комбинаторного взрыва.

Когда непрактичность универсальных решателей задач стала очевидной, исследователи переключились на разработку специализированных *экспертных систем*, работа которых основана на базах знаний в определенной области и правилах логического вывода, что позволяет им частично заменить человека-эксперта при решении

---

[1] Подробное описание принципа работы программы GPS можно найти в отчете: *Newell A., Shaw J. C. Simon H. A.* Report on a general problem-solving program // Proceedings of the International Conference on Information Processing, 1959. P. 256–264.

той или иной задачи. Подобные системы оказываются особенно полезными в таких областях, как медицина[1], где для решения задачи можно задействовать огромный объем существенной информации. Такая система может делать выводы, которые в противном случае потребовали бы участия человека-эксперта.

Отступление

Трудноразрешима такая задача, для решения которой существует алгоритм, но он слишком неэффективен, учитывая количество вариантов, которые необходимо перебрать. Комбинаторный взрыв — термин, описывающий эффект резкого увеличения количества вариантов по мере роста размерности задачи.

Рассмотрим игру в шахматы. У белых, как и у черных, есть 20 возможных вариантов первого хода. Таким образом, после того как каждый из игроков сделает первый ход, количество допустимых состояний доски будет равно 400, а после второго хода общее количество возможных вариантов будет выражаться шестизначным числом.

Субсимволический искусственный интеллект делает выводы, используя уравнения, а не следуя конкретным правилам. В то время как символический ИИ опирается на правила, разработанные людьми (и, таким образом, может продемонстрировать ход рассуждений, позволяющих ему прийти к заключению), субсимволический ИИ «учится» решать задачу, фактически не

[1] В области медицины существует целый ряд экспертных систем. Некоторые из них более эффективны, чем другие.

имея готовых правил нахождения ее решения. В основу субсимволического ИИ положен принцип обработки информации нейронами в человеческом мозге. Нейрон получает входные сигналы от других связанных с ним нейронов, и если взвешенная сумма[1] входных сигналов достигает порогового значения, данный нейрон передает сигнал дальше.

Субсимволический ИИ можно обучить, предварительно инициализировав веса входов случайными значениями. При получении неправильного ответа ИИ скорректирует значения весов, чтобы приблизиться к правильному результату, соответствующему входным данным. После проработки множества примеров эффективность ИИ достигает такого уровня, когда ему удается хорошо решать задачу, обходясь без алгоритма и имея лишь набор значений весов. В этом заключается суть машинного обучения, при котором ИИ учится на собственном опыте, не будучи явно запрограммированным на решение задачи, а люди просто поправляют машину, когда она допускает ошибку.

Практический пример

В 2020 году исследователи из Калифорнийского университета в Сан-Диего разработали алгоритм машинного обучения для распознавания признаков COVID-19 на рентгеновских снимках грудной клетки, которому удалось выявить случаи этого заболевания, не замеченные радиологами.

[1] Взвешенная сумма означает, что входные сигналы имеют разную значимость, зависящую от силы связи между нейронами.

34.2. Подобласти ИИ

> Компьютеры способны сравнительно легко достичь уровня взрослого человека в прохождении тестов на интеллект или в игре в шашки, но когда речь идет о восприятии или мобильности, они практически неспособны достичь уровня годовалого ребенка.
>
> *Ханс Моравек, 1988 год*

Парадокс Моравека заключается в том, что компьютеры зачастую весьма эффективно решают задачи, с которыми плохо справляются взрослые люди, но оказываются беспомощными в том, что без труда дается даже ребенку. Наиболее сложные области искусственного интеллекта часто связаны с задачами, которые люди решают совершенно бессознательно. К ним, например, относится распознавание лиц, ловля мяча и понимание речи. Многие из этих задач породили целые подобласти ИИ, в их числе:

- **компьютерное зрение** — разработки в данной области связаны с попытками научить компьютер распознавать объекты в изображениях или видео (рис. 34.1);
- **обработка естественного языка** — эта область посвящена попыткам научить компьютер понимать человеческий язык;
- **нейронные сети и машинное обучение.** Нейронная сеть — это набор алгоритмов, используемых для распознавания связей. Такие сети используются в машинном обучении;

Рис. 34.1. Робот, практикующий компьютерное зрение

- **планирование** — компьютер решает, какие действия необходимо предпринять для достижения поставленной цели;
- **робототехника** — обучение роботов навыкам ориентирования в пространстве и реагирования на неожиданные события в реальном мире;
- **обработка речи** — к этой области относится как распознавание, так и синтез речи.

34.3. Примеры

ИИ уже превратился в повседневное явление. Ниже перечислены некоторые примеры, наверняка знакомые многим читателям.

- Голосовые помощники, такие как Siri и Алиса, распознают и обрабатывают речь.
- Спам-фильтры учатся распознавать спам и пропускать обычные электронные письма, адаптируясь к изменяющимся методам рассылки спама.
- Чтобы вы могли сканировать чек с помощью своего телефона, он должен уметь распознавать буквы и цифры в изображении.
- Беспилотные автомобили распознают препятствия и избегают их в режиме реального времени. Робот-пылесос определяет оптимальный маршрут для уборки комнаты и возвращения к зарядному устройству.
- Компания Amazon составляет прогнозы относительно того, какие товары могут нас заинтересовать, демонстрирует их нам и даже обеспечивает их доставку на склад к тому времени, когда мы, по ее мнению, можем решить их приобрести[1].

[1] В 2014 году компания Amazon получила патент на систему упреждающей доставки, предполагающую отправку на местный склад тех товаров, которые вы, по ее мнению, скорее всего, купите. То есть Amazon обеспечивает доставку товара еще до совершения вами фактической покупки.

35 Квантовые вычисления[1]

До сих пор мы в основном говорили о классических компьютерах, хотя я и упомянул квантовую механику в главе 19.

Терминология

Классический компьютер — это просто компьютер, который не является квантовым. Его работа основана на принципах классической механики, в то время как квантовый компьютер использует явления квантовой механики.

Квантовый компьютер использует квантовомеханические эффекты для вычисления вероятностей. При этом он оперирует не битами, а кубитами (квантовыми битами), способными находиться в двух состояниях одновременно. Для данной задачи мы вычисляем вероятность всех возможных ответов и выбираем тот, который с наибольшей вероятностью является правильным.

[1] Существенный вклад в эту главу внес научный редактор данной книги, занимающийся исследованиями в области квантовых вычислений.

35.1. Физика

Квантовая физика изучает поведение материи на субатомном уровне. Для понимания принципа квантовых вычислений важно познакомиться со следующими концепциями.

- **Суперпозиция** — это способность квантовой системы находиться в нескольких состояниях одновременно. До проведения измерения кубит может находиться в состоянии суперпозиции, то есть в состоянии 0 и 1 (с некоторой вероятностью каждого из них).
- **Запутанность** — когда две или больше частицы являются запутанными, их квантовые состояния оказываются взаимозависимыми. Измерение состояния (спи́на, поляризации, положения, импульса) одной частицы влияет на другие, даже если они находятся на огромном расстоянии друг от друга[1].
- **Квантовое измерение** — при измерении состояния квантовой системы оно коллапсирует в одно из классических состояний. После этого каждый кубит имеет значение либо 0, либо 1, а не находится в состоянии суперпозиции.

35.2. Теоретические соображения

Квантовые компьютеры в действительности предназначены для решения тех же задач, что и классические, и вполне могут быть смоделированы машиной Тьюринга. Однако они позволяют справиться с задачами,

[1] Эйнштейн называл это явление «жутким дальнодействием».

решать которые на классическом компьютере было бы нецелесообразно из-за огромного количества вариантов, требующих проверки при использовании наиболее известных алгоритмов[1].

35.3. Практические соображения

До недавнего времени квантовые компьютеры использовались только в лабораториях и содержали лишь несколько кубитов, однако теперь они коммерчески доступны[2]. На момент написания этой книги квантовые компьютеры все еще в основном применялись для исследований, а не для решения практических задач. Однако технология быстро совершенствуется, и уже разработано множество языков квантового программирования.

---

[1] В 2019 году компания Google использовала 53-кубитный квантовый компьютер для решения весьма специфической задачи, на что у него ушло 200 секунд. Результат был проверен суперкомпьютером Summit, находящимся в Окриджской национальной лаборатории. Исследователи компании IBM подсчитали, что в идеальных условиях и при наличии дополнительной памяти Summit смог бы решить эту задачу за два с половиной дня.

[2] Вы можете запускать квантовые схемы на квантовом компьютере IBM по адресу https://quantum-computing.ibm.com, а также использовать квантовый компьютер D-Wave 2000Q, доступный на платформе Leap: https://www.dwavesys.com/take-leap.

Послесловие

Вы дошли до конца книги! (Если не считать приложения.)

Сейчас вы, возможно, чувствуете некую незавершенность — как будто вам нужно еще многое узнать. И вы правы.

Эта книга неполна в двух смыслах. Во-первых, ни одна из описанных тем не рассмотрена глубоко (если бы это было так, книга была бы втрое толще). Тем не менее теперь у вас есть общее представление, которое позволит вам осмысленно участвовать в обсуждении вопросов и по мере необходимости искать дополнительную информацию.

Я надеюсь, что вы посетите мой сайт http://www.whatwilliam-said.com/books/. Там вы найдете тесты для самопроверки, которые позволят вам узнать, хорошо ли вы усвоили каждую главу. Вы можете подписаться на мои рассылки, чтобы получить уведомление о выходе следующего издания (а также бесплатные дополнительные материалы), и связаться со мной, если у вас возникнут вопросы.

Если вы считаете эту книгу полезной, я буду очень признателен, если вы потратите пару минут и оставите отзыв о ней. Обратная связь с читателями очень важна для авторов!

Меня спросили, планирую ли я написать продолжение, и я ответил отрицательно. Я уже рассказал все, что хотел. При этом я не исключаю, что в серии руководств для программистов появятся мои новые книги. Как человек, испытывающий проблемы со слухом, я надеюсь написать книгу, посвященную теме доступности. Как старший разработчик, обладающий почти десятилетним опытом работы над масштабными и сложными программами, я хотел бы написать книгу для начинающих программистов, посвященную профессиональной разработке программного обеспечения. Ну а там будет видно.

Приложения

A Необходимая математика

Насколько глубокие познания в математике нужны для изучения Computer Science?

Обычно для поступления на специальности, связанные с Computer Science, нужно сдавать экзамен по алгебре; это необходимо потому, что студенты, у которых возникают сложности с пониманием переменных в алгебре, скорее всего, также с трудом поймут переменные в программировании. Поскольку эта книга ориентирована на программистов-практиков, предполагается, что читатель знаком с концепцией и использованием переменных.

При анализе времени выполнения алгоритма понадобятся алгебра и логарифмы. Логарифм числа — это степень, в которую нужно возвести основание логарифма (обычно для компьютеров это 2), чтобы получить данное число; например, логарифм 16 по основанию 2 равен 4, потому что $2^4 = 16$.

Более сложные темы могут также потребовать углубленного знания математики. В компьютерной графике

используются мнимые числа, а в машинном обучении — численные методы и статистика. Эти темы выходят за рамки данной книги.

Если вы не работаете в специализированной области, то вам, скорее всего, будет достаточно алгебры, логарифмов и теории графов (которые подробно рассмотрены в части II).

Б Классические NP-полные задачи

В этом приложении представлен краткий обзор некоторых классических *NP*-полных задач. Здесь не приводятся доказательства *NP*-полноты, а только дана информация, чтобы вы сами могли распознать эти задачи, когда встретитесь с ними. Обратите внимание, что задачи описаны так, что интересен не размер наилучшего решения, а то, существует ли решение размером *s*. Это связано с тем, что по определению *NP*-полные задачи — это задачи принятия решений (ответом на которые является «да» или «нет»).

Б.1. SAT и 3-SAT

Задача выполнимости булевых формул, ВЫП (boolean satisfiability problem, SAT), отвечает на вопрос: можно ли для заданной формулы, состоящей из логических переменных, подобрать такие значения этих переменных, чтобы результат формулы был истинным? Например, формула «*b* и не *c*» принимает значение «истина», если *b* истинно, а *c* — ложно. А вот утвер-

ждение «b и не b» не станет истинным ни при каких значениях b.

Как правило, формула представлена в виде набора условий. Например, если задана формула, значение которой истинно, если истинно любое из приведенных выше утверждений, то ее можно записать как $(b \wedge \neg c) \vee (b \wedge \neg b)$ (это читается так: «b и не c или b и не b»). В данном случае каждое из утверждений содержит два литерала, но вообще оно может содержать любое число литералов. 3-SAT — это та же задача с дополнительным ограничением: каждое условие ограничено максимум тремя литералами.

Б.2. Клика

Для заданного графа G и размера k определите, содержит ли G клику (то есть полный подграф) размером k.

Б.3. Кликовое покрытие

Для заданного графа G и размера k определите, можно ли разбить G на k клик таким образом, чтобы каждая вершина графа принадлежала хотя бы одной из полученных клик.

Б.4. Раскраска графа

Для заданного графа G и размера k определите, можно ли правильно раскрасить G, используя только k цветов.

Б.5. Гамильтонов путь

Для заданного графа G определите, существует ли путь между вершинами графа, который проходит через каждую вершину ровно один раз.

Б.6. Укладка рюкзака

Для заданного набора предметов, каждый из которых имеет вес и ценность, и рюкзака с максимальной вместимостью c определите, можно ли найти набор предметов с общей ценностью не менее v, которые не превышают вместимость рюкзака.

Б.7. Наибольшее независимое множество

Для заданного графа G определите, существует ли независимое множество (множество вершин, в котором нет двух смежных вершин) размером k.

Б.8. Сумма подмножества

Для заданного множества или мультимножества (множества, в котором допускаются повторяющиеся значения) целых чисел и значения s определите, существует ли непустое подмножество, сумма которого равна s? Например, для множества $\{-7, -5, -3, -1, 4, 8, 156\}$ и $s = 0$ таким подмножеством было бы $\{-7, -5, 4, 8\}$.