[Алгоритмы]

→ для начинающих

Теория и практика для разработчика

Панос Луридас

Широкий обзор алгоритмических основ, включающий, помимо «лучших хитов», новые интересные направления, такие как системы голосования и алгоритм сжатия текста. Отличное пособие для начинающих, которое стоит прочитать прежде, чем приступить к продвинутому уровню изучения темы.

Стивен Скиена,
автор бестселлера «Алгоритмы. Руководство по разработке»

The MIT Press

БОМБОРА


МИРОВОЙ
КОМПЬЮТЕРНЫЙ
БЕСТСЕЛЛЕР

[Real-World → Algorithms]

A Beginner's Guide

Panos Louridas

The MIT Press

[Алгоритмы]
→ для начинающих

Теория и практика для разработчика

Панос Луридас

БОМБОРА™
Москва 2018

УДК 004.42
ББК 32.973.26-018
Л86

Real-World Algorithms. A Beginner's Guide
Panos Louridas

Луридас, Панос.

Л86 Алгоритмы для начинающих : теория и практика для разработчика / Панос Луридас ; [пер. с англ. Е.М. Егоровой]. — Москва : Эксмо, 2018. — 608 с. — (Мировой компьютерный бестселлер).

Алгоритмы правят миром! Эта книга в простой и наглядной форме дает ответы на целый ряд важнейших для начинающего программиста вопросов, начиная с «Что лежит в основе всех современных языков программирования и по каким принципам они строятся и работают?» и заканчивая «Есть ли способ овладеть всеми языками программирования сразу?».

УДК 004.42
ББК 32.973.26-018

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Панос Луридас

АЛГОРИТМЫ ДЛЯ НАЧИНАЮЩИХ

ТЕОРИЯ И ПРАКТИКА ДЛЯ РАЗРАБОТЧИКА

Директор редакции *Е. Капьёв*. Ответственный редактор *Е. Истомина*
Младший редактор *Е. Минина*. Художественный редактор *С. Власов*

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел.: 8 (495) 411-68-86.
Home page: **www.eksmo.ru** E-mail: **info@eksmo.ru**

Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.

Тауар белгісі: «Эксмо»
Қазақстан Республикасында дистрибьютор және өнім бойынша
арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8(727) 2 51 59 89,90,91,92, факс: 8 (727) 251 58 12 вн. 107; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить по адресу: http://eksmo.ru/certification/ .

Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 05.12.2017. Формат 60x90$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 38,0. Тираж экз. Заказ

ISBN 978-5-04-089834-3

ISBN 978-5-04-089834-3

Содержание

Предисловие

> Подобно многим из моего поколения, я был воспитан на изречении: «Сатана всегда найдет недостойное дело для праздных рук». Будучи глубоко добродетельным ребенком, я верил всему, что мне говорили, и поэтому приобрел чувство моральной ответственности, вплоть до настоящего времени заставлявшее меня прилежно трудиться. Но хотя совесть и контролировала мои действия, взгляды мои претерпели серьезные изменения. Я утверждаю, что в мире сделано чересчур много работы, что вера в то, что работа — это добродетель, нанесла огромный вред и что в современных индустриальных странах следует проповедовать идеи, весьма далекие от тех, что издавна проповедовались.
>
> Бертран Рассел, «Похвала праздности»

Моя книга посвящена алгоритмам; это такая вещь, которая показывает нам, что делать, чтобы ничего не делать. Можно сказать, это работа, которая помогает избежать еще большей работы. Из-за обилия изобретений, сделанных человеком, мы постоянно заполняем мозг большим количеством разнообразной информации. Благодаря же алгоритмам мы можем использовать свой мозг для размышлений.

Облегчение человеческого труда — благородная цель. Мы давно свыклись с мыслью, что машины призваны облегчить человеческий труд и сократить тяжкую, изнурительную работу, которая на протяжении многих веков считалась частью повседневного человеческого уклада. Мысль прекрасна, и нам стоит относить ее не только к физическому труду, но и к умственной работе. Однообразие и рутина напрочь убивают творческое мышление, а нам бы очень хотелось сохранить его. Вот здесь и приходят на помощь алгоритмы.

Современные цифровые технологии способны совершать операции, которые сам человек выполняет не задумываясь, ведь они

воплощают самую суть человеческой природы. Машины распознают и воспроизводят речь, переводят тексты, сортируют и приводят в порядок документы, предсказывают погоду, с поразительной точностью вылавливают закономерности в куче информации, заставляют работать другие машины, производят математические расчеты, становятся оппонентом в играх и помогают в изобретении новых механизмов — все благодаря алгоритмам. Работа алгоритмов облегчает нам работу и освобождает время на получение знаний в других областях, мы даже успеваем придумывать более сложные алгоритмы, которые еще сильнее уменьшают ежедневную рутину.

Алгоритмы появились задолго до компьютеров, еще в древние времена, и их применяют не только в области информационных технологий. Сложно найти сферу деятельности, которая обошлась бы без алгоритмов. Рано или поздно все люди сталкиваются с алгоритмами: однажды к ним приходит осознание, что алгоритмы — неотъемлемая часть их профессии, которая порой даже не связана с компьютерами, и тогда люди начинают интересоваться алгоритмами, стремясь оптимизировать свою работу.

Просто поразительно, как много сил мы тратим даже на повседневные мелкие дела, а все потому, что не знаем верного подхода. Каждый раз, когда вы ловите себя на мысли, что занимаетесь нудной, монотонной работой, скорее всего вы и правда попусту тратите время. Автору не раз доводилось наблюдать работу в офисах, когда сотрудники выполняли целый ряд однообразных действий, которые можно сделать за долю секунды и избежать волокиты, если только знать, как правильно организовать работу: нужно не отказываться от использования компьютеров (многие люди находят по десятку отговорок, чтобы не учиться ими пользоваться), а передать машинам выполнение всех однообразных задач.

Целевая аудитория

Книга написана для тех, кто только начал знакомиться с алгоритмами. Если вы изучаете информатику, то данное руководство поможет вам постигнуть ее азы, а затем перейти к более сложной

литературе. Алгоритмы — основа программирования, и подобное введение способно охватить лишь малую часть темы и дать лишь общее представление о предмете.

Книга также окажется полезной людям, чья профессиональная или учебная деятельность хоть и не связана с информатикой, однако не обходится без алгоритмов. Почти в каждой дисциплине есть свои алгоритмы, и без них никуда; так что данное руководство расскажет о них тем — а таких людей немало, — кто желает их понять и сделать если не центром, то частью своей работы или учебы.

Книга придется по душе и тем, кто захочет прибегнуть к алгоритмам в простых, незначительных ситуациях, чтобы облегчить себе работу и избежать рутины; ведь задачу, выполнение которой обычно занимает несколько часов, можно решить за считанные секунды — стоит только написать программный код на современном скриптовом языке. Порой это становится настоящим откровением для непосвященных людей, которые полагают, что знание алгоритмов — прерогатива некой интеллектуальной элиты, хотя на самом деле алгоритмы доступны всем.

В наши дни никто всерьез не станет спорить с фактом, что без знания основ математики и науки сложно угнаться за современным миром и понять происходящие в нем процессы; точно так же и с алгоритмами: в современном социуме продуктивный труд невозможен, если не имеешь ни малейшего понятия об алгоритмах. Они лежат в основе человеческих жизни и опыта.

Что нужно знать

Не стоит думать, что алгоритмы могут понять одни только программисты. Они состоят из четких инструкций по выполнению задач, и понять их способен каждый человек. В то же время для плодотворного труда и извлечения максимума пользы читатель должен обладать определенным багажом знаний.

Вовсе не обязательно быть опытным математиком, но иметь представление об основных математических понятиях и терминах необходимо. Уровень математики в данной книге не превы-

шает уровня старшей школы. Совсем не обязательно знать высшую математику, но понимать логику математических доказательств нужно, потому что точно таким же образом — с помощью логики — мы будем разбирать алгоритмы. Мы не будем делать сложные выкладки, однако читатель должен понимать, как строятся математические доказательства в этой книге.

Вовсе не обязательно быть профессиональным программистом, но нужно обладать базовыми навыками в работе с компьютером, знать, как пишутся программы, и понимать структуру языков программирования. От читателя не требуется глубокое понимание предмета; вообще-то лучше всего, если он только-только принялся его изучать. Компьютерные системы и алгоритмы очень близки и взаимно дополняют друг друга.

Также нужна любознательность. Алгоритмы призваны решать проблемы и делать это максимально плодотворно. Каждый раз, когда вы спрашиваете себя: «Как лучше решить ту или иную проблему?» — вы на самом деле ищете алгоритм.

Как подан материал

Цель данной книги — объяснить алгоритмы доступным читателю языком, не принижая при этом его интеллекта. Если вы читаете книгу и понимаете, что ничего не понимаете, если в вашу голову закрадываются сомнения, что, возможно, она написана для гениев или профессионалов, если вы смущены тем, что прочитанные слова не складываются в общую картину, значит, книга оскорбляет ваш интеллект. Мы постараемся избежать любых неясностей. Нам придется что-то упростить, поэтому некоторые понятия, приведенные здесь, будут раскрыты не в полной, а лишь в необходимой мере.

Допущения и упрощение сложных вещей вовсе не означают, что читатель окажется избавлен от вдумчивого чтения, ведь мы не сомневаемся в его умственных способностях и не желаем их принижать. Мы полагаем, что читатель искренне жаждет разобраться в алгоритмах, процесс познания требует сил и времени, поэтому чем больше вы вникаете, тем больше извлекаете для себя пользы.

Обратимся на минутку к области литературы. Бывают книги, которые затягивают вас с головой и от которых невозможно оторваться, вы читаете их и не замечаете, как пролетает время. И речь тут не о модных бестселлерах. К примеру, «Чума» Альбера Камю читается очень легко, хотя это и очень серьезное и глубокое произведение.

Бывают книги, которые требуют значительного умственного труда. Их сложно читать, однако те люди, которым удается их понять, испытывают огромное удовольствие и даже чувство собственной уникальности от того, что смогли уловить мысль автора. Например, далеко не все любят «Улисса» Джеймса Джойса, книги Томаса Пинчона или Дэвида Фостера Уоллеса. В то же время вы вряд ли найдете хоть одного человека, который бы пожалел, что прочел их.

Есть третья категория книг, которая находится где-то между первыми двумя. Возможно, «Радуга земного тяготения» слишком специфична для вас, но разве можно сказать что-то подобное о «Братьях Карамазовых» или «Анне Карениной»?

Книга, которую вы держите в руках, как раз из тех, которые находятся где-то между: это не мудреный талмуд, но в то же время ее чтение требует некоторой сообразительности. Автор станет вашим проводником в мир алгоритмов, но не средством передвижения; он протянет вам руку и укажет путь через все алгоритмические хитросплетения, но пройти этот путь вы должны все же своими ногами. Ваш интеллект не будет оскорблен. Напротив, мы полагаем, что вы — любознательная личность, которая желает узнать много новых интересных вещей и прекрасно понимает, что в учении, как в бою, требуются усилия и отдача; что вы тот, кто знает — успех не приходит сам по себе, но добывается с помощью труда.

Псевдокод

Годы назад молодые люди могли начать карьеру программиста, прекрасно владея всего одним программным языком. Теперь все изменилось. Сейчас в ходу множество хороших языков; компью-

теры теперь способны выполнять куда больше операций, чем двадцать лет назад, и в зависимости от конкретной задачи выбирается наиболее подходящий язык. Языковые войны глупы и бесполезны. К тому же из-за многообразия чудес, на которые способны нынешние компьютеры, люди все активнее взаимодействуют с ними, развивая новые сферы и направления, что приводит к созданию новых программных языков и эволюции старых.

Автор честно сознается, что считает некоторые языки более удобными, чем остальные, однако считает несправедливым и недопустимым навязывать читателю свои предпочтения. Более того, мода на языки меняется довольно быстро, и вчерашний фаворит может завтра оказаться не у дел. Чтобы книга стала максимально полезной и универсальной, на ее страницах не приводятся примеры каких-либо конкретных языков программирования — алгоритмы здесь описаны с помощью псевдокода. Его понять куда проще, чем какой-либо действующий язык программирования. С помощью псевдокода проще объяснить материал: когда вы пишете на нем, он позволяет сосредоточиться на понимании алгоритмов, а не на синтаксических конструкциях, которые постоянно требуют внимания и осторожности.

С другой стороны, сложно работать с алгоритмами без написания программного кода, с помощью которого они реализуются. Применение псевдокода в данном руководстве не означает, что читателю нужно так же легко и условно относиться к программным кодам в целом; по возможности следует реализовывать приведенные в книге алгоритмы, выбирая язык программирования на свой вкус. Не стоит забывать, что полное понимание и удовольствие от предмета будет достигнуто только тогда, когда вам удастся создать программу, которая реализует нужный алгоритм так, как вам требуется.

Как читать данную книгу

Лучше читать все по порядку, одно за другим, так как в первых главах даются разъяснения и концепции, употребляемые в даль-

нейшем. В самом начале вы познакомитесь с основными структурами данных, которые используются во всех алгоритмах и проявят себя в следующих главах. Дальше, когда базис знаний будет заложен, вы можете читать главы в любом порядке, выбирая то, что вас больше интересует.

Таким образом, следует начать знакомство с самой первой главы, чтобы понять структуру всех последующих: сперва идет описание проблемы, а затем — разбор алгоритма, который позволяет ее решить. Также первая глава познакомит вас с условностями псевдокода данной книги, основными терминами и первыми структурами данных: массивами и стеками.

Вторая глава освещает тему графов и способы работы с ними. Также она рассказывает о рекурсии, так что, если вам уже доводилось видеть графы, но вы не до конца уловили идею рекурсии, рекомендуем не пропускать эту главу. Во второй главе мы также познакомимся с новыми структурами данных, которые время от времени будут нам попадаться в алгоритмах далее. В третьей главе мы рассмотрим проблему сжатия и то, как работают два различных метода сжатия: глава послужит нам подспорьем в понимании важных структур данных, с которыми мы встретимся позже.

Четвертая и пятая главы расскажут о криптографии. Она не похожа на графы и сжатие, однако является важной частью алгоритмов, особенно в последние годы, когда персональные данные стали храниться в различных местах и устройствах, вызывая у некоторых субъектов желание до них добраться. Эти две главы, в принципе, можно читать отдельно от остальных, хотя некоторые значимые фрагменты, например поиск больших простых чисел, помещены и рассмотрены в главе 16.

Главы 6–10 посвящены проблемам, связанным с графами: порядок заданий, нахождение выхода из лабиринта, определение приоритета соединенных друг с другом частей (например, страниц на сайте) и использование графов в выборах. Нахождение пути в лабиринте применяется куда чаще, чем может показаться на первый взгляд: от набора газетных статей до интернет-маршрутизации и валютного арбитража; его разновидность воз-

никает в контексте выборов, поэтому главы 7, 8 и 10 можно рассматривать как единое целое.

Главы 11 и 12 затрагивают две основные проблемы программирования: поиск и сортировку. По этим темам можно писать отдельные книги. Мы покажем несколько базовых и широко применяемых алгоритмов. Разобравшись с сортировкой, мы получим возможность разобраться с дополнительным материалом, например с интернет-поиском (поиск среди предметов, которые последовательно проходят перед вашими глазами, с условием, что после выбора нельзя изменить свое решение) и безмасштабной сетью, которую внимательный исследователь может увидеть где угодно. В главе 13 описан еще один путь сортировки и выборки данных — с помощью хеширования, очень полезного, распространенного и элегантного способа.

Глава 14 описывает классификацию алгоритмов: алгоритм учится группировать данные, основываясь на заданных примерах, а затем мы используем его, чтобы систематизировать новые, незнакомые случаи. Здесь мы сталкиваемся с машинным обучением, областью, значимость которой стремительно возрастает по мере того, как компьютеры становятся мощнее и мощнее. Глава также охватывает основные принципы теории информации, еще одной прекрасной области, тесно связанной с алгоритмами. Четырнадцатая глава отличается от остальных тем, что показывает, как алгоритм работает с более мелкими алгоритмами, выполняющими долю его работы; точно так же компьютерные программы состоят небольших частей, каждая из которых имеет свое назначение. Глава так же показывает, каким образом структуры данных, описанные в данной книге, играют ключевую роль в выполнении алгоритма классификации. Эта глава особо заинтересует читателей, желающих узнать об устройстве частей высокоуровневых алгоритмов. Это станет большим шагом в понимании того, как алгоритмы превращаются в программы.

Глава 15 рассказывает о последовательности символов, именуемых также строками символов, и том, как в них что-то найти. Тут мы встретим операцию, к которой мы прибегаем каждый

раз, когда ищем что-то внутри текста и хотим добиться плодотворного результата. К счастью, существует несколько быстрых и изящных способов сделать это. Более того, последовательности символов могут представлять множество других вещей, так что сравнение строк применяется в самых разных областях, например в биологии.

В последней, шестнадцатой главе мы затронем алгоритмы, которые работают со случайностями. Существует поразительно большое количество таких рандомизированных алгоритмов, поэтому мы рассмотрим лишь небольшую их группу. Ко всему прочему они дадут нам ответы на поставленные в предыдущих главах вопросы, например поиск больших простых чисел в криптографии. Или опять же выборы: как подсчитать результаты.

Использование в обучении

Материал данной книги рассчитан на семестровый курс, знакомящий учащихся с алгоритмами и объясняющий их основные принципы, не углубляясь в технические подробности. Студенты, изучающие самые разные дисциплины: бизнес и экономику, социальные и прикладные науки или же классические науки вроде математики и статистики, — могут использовать это руководство на начальной ступени обучения, как дополнение к практическому курсу программирования, чтобы сразу на практике оценить всю пользу алгоритмов. Те же, кто изучает информатику, могут использовать книгу для неформального знакомства с предметом: она позволит им оценить всю глубину и красоту алгоритмов, техническая сторона которых более подробно расписана в специализированных учебниках.

Заключительные вступительные слова

Если вглядеться в слово «алгоритм», то можно увидеть в нем фразу «ритм боли», так как «алгос» на греческом значит «боль».

Но не стоит пугаться, слово «алгоритм» придумали не греки, а персидский математик, астроном и географ Аль-Хорезми (около 780–850 гг). Очень надеюсь, что данное руководство принесет вам радость познания и ни капли боли. Давайте же познакомимся с алгоритмами.

1 Разница курсов акций

Представьте, что курс акций меняется каждый день. Таким образом, у вас есть ряды чисел, каждое из которых представляет собой финальную стоимость конкретной акции в конкретный день. Дни идут в хронологическом порядке. Дни, когда фондовый рынок закрыт, не считаются.

Разница стоимостей акций в конкретный день — это число следующих друг за другом дней, от выбранного нами и в обратном направлении, до того дня, в который стоимость была меньше или равна стоимости в выбранный нами день. Таким образом, задача разницы курсов акций представляет следующее условие: даны ряды ежедневных курсов акции, и нам требуется найти разницу стоимостей акций для каждого дня в ряду. Для примера обратимся к рисунку 1.1. Первый день считается за 0. На шестой день нашего примера разница составляет пять дней, на пятый день — четыре дня, а на четвертый день — один день.

В реальной жизни в рядах могут находиться тысячи дней, и нам нужно будет просчитать разницу для множества различных рядов, в каждом из которых описаны изменения для разных курсов акций. Мы хотим поручить эту задачу компьютеру.

Многие задачи, решение которых мы поручаем компьютеру, имеют, как правило, не один способ решения, а несколько. Какие-то решения лучше, какие-то хуже. Впрочем, в данном случае «лучше» — весьма условный термин, который по сути ничего не значит, ведь, когда мы говорим «лучше», мы подразумеваем какой-то контекст: лучше в чем-то конкретном. Речь может быть о скорости, памяти или чем-то еще, что связано с такими ресурсами, как время и пространство. Мы еще коснемся данной темы, однако сейчас главное усвоить сказанное выше, потому что решение задачи может оказаться простым и в то же

время совсем не оптимальным из-за поставленных условий или критериев.

Допустим, из ряда дней вы взяли день *m*. Можно найти разницу курсов акций в день *m*, вернувшись на день назад и оказавшись в дне *m – 1*. Если стоимость в день *m – 1* выше, чем стоимость в день *m*, тогда вы знаете, что разница курсов акций в день *m* равна 1. Но если стоимость в день *m – 1* ниже или равна стоимости в день *m*, то разница курсов акций в день *m* равна 2, а может быть даже больше: в зависимости от стоимости в предыдущий день. Поэтому нам нужно посмотреть цену акций в день *m – 2*. Если стоимость не выше, чем в день *m*, то нам надо проверить более ранний день и так далее. Следовательно, можно получить два итога. Итог первый: у нас закончатся дни (то есть мы вернемся в самое начало ряда). В этом случае получается, что на протяжении всех дней до дня *m* курс акций был ниже или точно таким же, как в день *m*, и значит, разница курсов равна *m*. Итог второй: мы наткнемся на день *k* (где $k < m$), когда цены на акции были выше, чем в день *m*, и получим, что разница равна $m - k$.

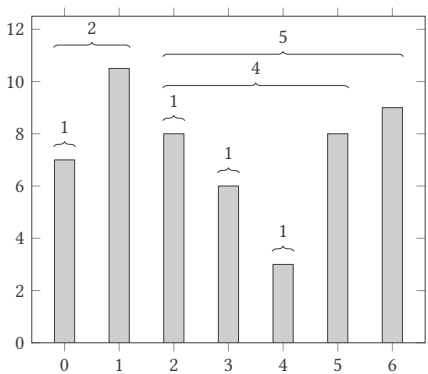


Рисунок 1.1. Пример разницы стоимостей акций

Если ряд состоит из *n* дней, то, чтобы найти разницу для каждого дня, мы повторяем описанную выше процедуру *n* раз. Применив данный метод к ряду, изображенному на рисунке 1.1, можно убедиться, что он работает корректно. Однако такой подход

никуда не годится. Проза — прекрасное средство коммуникации и описания реалий нашего мира, но она совершенно не подходит для описания компьютерных процедур, потому что нам надо четко и ясно описать машине поставленную задачу.

Если мы достаточно точны в своих выражениях, то компьютер сможет понять наши команды, после чего мы сможем написать программу. Однако описание команд в компьютерных программах может оказаться совершенно непонятным человеку, потому что машине требуются всевозможные уточнения и детали, нужные ей для работы, но не связанные с решением поставленной задачи напрямую. То, что для компьютера является минимально подробным описанием, для человека может оказаться сверхподробным, и он его едва ли поймет.

Алгоритм 1.1. Простой алгоритм для задачи о разнице курсов акций

```
SimpleStockSpan(quotes) → spans
    Вводные данные: quotes, массив с n курсами акций.
    Выводимые данные: spans, массив с n разницами курсов акций.
 1  spans ← CreateArray(n)
 2  for i ← 0 to n do
 3      k ← 1
 4      span_end ← FALSE
 5      while i − k ≥ 0 and not span_end do
 6          if quotes[i − k] ≤ quotes[i] then
 7              k ← k + 1
 8          else
 9              span_end ← TRUE
10      spans[i] ← k
11  return spans
```

Поэтому нам стоит выбрать золотую середину и описать команды с помощью структурированного языка, который более точен, чем обычный текст, с пониманием которого у людей не возникнет проблем. Скорее всего, компьютер не поймет заданный структурированный язык как таковой, но если на этом языке написать компьютерную программу, то он без проблем поймет, что мы от него хотим.

1.1. Алгоритмы

Прежде чем мы приступим к поиску решения нашей задачи с разницей курсов акций, хорошо бы познакомиться с основными терминами. Алгоритм — это последовательность операций, но не простая, а особая. Ее можно описать как конечную последовательность множества этапов, которая должна завершиться через определенный промежуток времени. Каждое действие, каждый этап должны быть четко прописаны так, чтобы человек мог самостоятельно разобрать его на бумаге и понять. Алгоритм выполняет действия, основанные на полученных от нас данных, и выдает результат, показывающий выполненную работу. Алгоритм 1.1 реализует описанную ранее процедуру.

Алгоритм 1.1 демонстрирует, каким образом мы будем описывать алгоритмы. Вместо языка программирования, который заставляет нас заниматься подробностями реализации, не относящимися к логике алгоритма, мы используем псевдокод. Это нечто между настоящим программным кодом и неформальным описанием. Он использует структурированный формат и применяет множество слов, которые наделяет особым значением. Так или иначе, псевдокод нельзя считать настоящим программным кодом. Он предназначен не для того, чтобы его понимали машины, а для того, чтобы его понимали люди. К слову, программы вообще должны быть понятны людям, но на деле все далеко не так просто: существует множество рабочих компьютерных программ, написанных совершенно невнятно.

У каждого алгоритма есть название, они берут вводные данные и выдают некий результат. Мы будем записывать название алгоритма в «ВерблюжьемРегистре» (`CamelCase`), а вводные данные — в круглых скобках. Затем мы отметим полученные данные с помощью →. В последующих строчках мы будем описывать вводимые и выводимые данные. Название алгоритма, за которым в скобках следуют вводные данные, может применяться для вызова алгоритма. Как только алгоритм написан, мы можем использовать его в качестве черного ящика, который мы заполняем вводными данными: черный ящик потом выдаст выводимые

данные, результат заданного алгоритма. Реализованный в языке программирования алгоритм представляет собой именованный фрагмент программного кода — функцию. В компьютерной программе мы вызываем функцию, с помощью которой работает алгоритм.

Некоторые алгоритмы выдают результат неявным образом, так как их действия направлены на часть контекста. Например, мы можем выделить алгоритму место, куда будет вписан итог работы. В этом случае алгоритм не вернет привычных выводимых данных, но изменит контекст. Некоторые языки программирования отличают фрагменты именованного программного кода, возвращающие видимый результат, называя их функциями, от фрагментов именованного программного кода, которые не возвращают выводимые данные в явной форме, но имеют иные побочные действия — их называют процедурами. Различие пришло из математики, где функция всегда выдает какое-то значение. Для нас же алгоритм, реализованный в компьютерной программе, может оказаться как функцией, так и процедурой.

В нашем псевдокоде будут встречаться ключевые слова, выделенные **жирным шрифтом**, которые не требуют разъяснений, если вы хоть немного знакомы с компьютерами и языками программирования. Мы используем символ ← для присвоения и символ = для равенства, а также заимствуем пять символов математических операций (+, –, /, x, ·); у нас два знака умножения, и нам понадобятся оба, в зависимости от эстетических предпочтений. Мы не будем использовать ключевые слова или символы для разграничения блоков псевдокода: мы прибегнем к красной строке.

В данном алгоритме мы применяем массив. Это структура, содержащая данные, которые позволяют нам использовать данные особыми способами. Структура, содержащая данные и позволяющая производить особые операции над ними, называется структурой данных. Таким образом, массив является структурой данных.

Массивы для компьютера — то же самое, что ряды объектов для людей. Они состоят из последовательности элементов, которые хранятся в памяти компьютера. Чтобы получить требуемое

для элементов место и создать массив, содержащий n элементов, в первой строчке алгоритма 1.1 мы вызываем алгоритм создания массива, CreateArray. Если вы знакомы с массивами, то вам может показаться странным, что создание массива нуждается в алгоритме. Но все так и есть. Чтобы получить блок памяти для данных, вам нужно найти свободное место в памяти компьютера и отметить его как используемое массивом. Все это делается с помощью вызова CreateArray(n). В результате мы получаем массив с местом для n элементов: сначала там нет никаких элементов, а только место для них. Заполнить свободное пространство нужными данными — как раз задача алгоритма, который вызывает CreateArray(n).

Для массива A с помощью $A[i]$ мы помечаем элемент i и получаем к нему доступ. Позиция элемента в массиве, как у i в $A[i]$, называется индексом. В массиве из n элементов содержатся элементы $A[0], A[1],\ldots, A[n - 1]$. Сперва это может показаться странным, так как первый элемент — нулевой, а последний $(n - 1)$. Наверняка вы ожидали, что на их месте будут первый и n-й. Однако именно так массивы и работают в большинстве программных языков, к чему вам следует поскорее привыкнуть. Как обычно, когда мы просматриваем массив размером n, мы выполняем перебор от 0 до $n - 1$. В наших алгоритмах, когда мы говорим, что нечто будет принимать значения от числа x до числа y (полагая, что x меньше y), мы имеем в виду все значения x и выше, но не включая y; посмотрите вторую строчку нашего алгоритма.

Мы полагаем, что на доступ к элементу i всегда требуется одно и то же время, чем бы ни являлось i, поэтому доступ к $A[0]$ потребует столько же времени, сколько и к $A[n - 1]$. В этом заключается характерная особенность массивов: до каждого элемента можно добраться за одно и то же время; массиву не нужно искать какой-то элемент, когда мы получаем к нему доступ по индексу.

Что касается условных знаков, то при описании алгоритмов мы используем для значения переменных, которые фигурируют в них, строчные символы; однако когда переменные будут связаны со структурой данных, мы будем применять символы верхнего регистра, как в массиве A, чтобы выделить их; но это не всегда необходимо. Когда нужно будет дать переменной имя из несколь-

ких слов, мы используем подчеркивание (_), как в a_connector, что необходимо, поскольку компьютеры не понимают, что слова, разделенные пробелами, могут составлять единую фразу.

Алгоритм 1.1 использует массивы, в которых находятся числа. Массивы могут содержать любой тип элементов, однако в нашем псевдокоде каждый массив может заключать в себе только один тип. То же самое относится к большинству языков программирования. Например, у вас может быть массив с десятичными числами, массив с дробями, массив элементов, обозначающих людей, и другой массив с элементами, обозначающими адреса. У вас не может быть массива, в котором одновременно и десятичные числа, и элементы, обозначающие людей. Чем могут являться «элементы, обозначающие людей» зависит от особенностей программного языка, использованного в программе. Все языки программирования предлагают способы представления важных вещей.

Особо полезный вид массива — тот, который, содержит символы. Массив символов представляет собой строку символов, то есть последовательность букв, чисел, слов, предложений и чего угодно. Во всех массивах каждый отдельный символ может быть найден по индивидуальному индексу. Если у нас есть строка s = “Hello, World”, то $s[0]$ — это буква “H”, а $s[11]$ — буква “d”.

Подытожим сказанное. Массив — это структура данных, в которой содержится последовательность элементов одного и того же типа. Есть две операции, относящиеся к массивам.

- CreateArray(n) создает массив, который может вместить n-е количество элементов. Массив не запущен, в нем пока нет ни одного элемента, но нужное им место создано и готово к использованию.
- Как мы видели, имея массив A и его i элемента, $A[i]$ получает доступ к выбранному элементу, и получение доступа к любому элементу массива занимает одинаковое время. Попытка получения доступа к $A[i]$ при условии $i < 0$ приведет к ошибке.

Вернемся к алгоритму 1.1. Следуя сказанному выше, в строчках 2-10 нашего алгоритма содержится цикл, то есть блок кода, который повторяется. Цикл выполняется n раз; единожды для под-

счета разницы, если у нас есть цена акций за n дней. Нынешний день, разницу в который мы собираемся вычислить, задан переменной i. Изначально мы находимся в дне 0, самой ранней точке временной линии; каждый раз, проходя через вторую строчку нашего цикла, мы будем перемещаться в день 1, 2, …, $n - 1$.

Мы используем переменную k для обозначения длины промежутка, отмеряющего разницу; переменная — это название, присвоенное некоему фрагменту данных в нашем псевдокоде. Содержание этих данных, а точнее значение переменной может меняться в процессе выполнения алгоритма, и, следовательно, меняется ее название. Значение k на момент начала расчета разницы всегда равно 1, что и прописано в строчке 3. Также мы используем индикаторную переменную, span_end. Индикаторные переменные принимают значения истинно (`true`) и ложно (`false`) и указывают, выполняется ли что-то или не выполняется. Переменная span_end примет значение истинно, когда мы достигнем конца промежутка, отмеряющего разницу.

В начале расчета промежутка переменная span_end будет ложной, как в строчке 4. Длина промежутка просчитана во внутреннем цикле строчек 5-9. Строчка 5 говорит вернуться в прошлое насколько возможно далеко и на расстояние, пока промежуток не закончится. Настолько далеко, насколько позволяет условие $i - k \geq 0$, где $i - k$ является индексом дня, в который нам нужно вернуться, чтобы проверить, закончился ли промежуток, обозначающий разницу; к тому же индекс не может равняться нулю, так как он отображает первый день. Проверка конца промежутка отображается в строчке 6. Если промежуток не закончился, то мы увеличиваем его в строчке 7. В противном случае в строчке 9 мы констатируем факт, что промежуток закончился, и, таким образом, цикл остановится, когда процесс вернется к строчке 5. В конце каждой итерации внешнего цикла строчек 2-10 мы помещаем значение k в отведенное в массиве *span* место (строчка 10). Мы возвращаем промежутки (*spans*), в которых находятся результаты алгоритма, в строке 11, после выхода из цикла.

Заметьте, что в начале $i = 0$ и $k = 1$. Это значит, что условие в строчке 5 совершенно точно не будет выполнено на раннем мо-

менте времени. Так и должно быть, потому что разница может быть равна только 1.

Тут самое время вспомнить, что мы только что говорили об алгоритмах и проверке на бумаге. Верный способ понять работу алгоритма — сделать ее самому, своими руками. Если в какой-то момент алгоритм кажется сложным или же вы не уверены, что уловили его суть, тогда возьмите ручку, лист бумаги и распишите его действия в нескольких примерах. Хоть способ и кажется устаревшим, тем не менее он сэкономит вам уйму времени. Если вы что-то не поняли в алгоритме 1.1, разберитесь с ним и возвращайтесь к прочтению книги, лишь когда полностью освоите его.

1.2. Время выполнения работы и сложность

В алгоритме 1.1 мы видим решение задачи о разнице курсов акций, однако мы можем найти способ получше. Лучше в данном случае значит — быстрее. Когда мы говорим о скорости в контексте алгоритмов, мы подразумеваем число этапов, которое требуется для его выполнения. Скорость компьютеров тут не важна; хотя они и будут выполнять расчетные этапы все быстрее и быстрее, количество самих этапов останется неизменным, поэтому имеет смысл задуматься над оценкой сложности выполнения алгоритма в этапах, нужных для выполнения работы. Мы называем количество требуемых этапов временем выполнения работы алгоритма, хотя имеем дело с безразмерным числом, не обусловленным единицами времени. Использование единиц времени сделало бы подсчет времени выполнения чем-то, относящимся к специфичной расчетной модели, которая для нас бесполезна.

Подумайте, как много времени займет расчет разницы n курсов акций. Наш алгоритм включает в себя цикл, который начинается в строчке 2 и повторяется n раз: для каждой цены. Тут есть и внутренний цикл, который начинается в строчке 5 и при каждом повторе внешнего цикла будет пытаться найти промежуток, обозначающий разницу стоимостей. Каждую стоимость он сравнит с ценами всех предыдущих дней. При самом неблагоприятном варианте, если указанная цена окажется самой высокой,

цикл проверит все предыдущие цены. Если стоимость k самая высокая из всех предыдущих, тогда внутренний цикл повторится k раз. Тем не менее в худшем случае, когда все цены идут в порядке возрастания, строчка 7 выполнится следующее число раз:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Если равенство неочевидно, тогда можно добавить числа 1, 2, …, n дважды и увидеть, что все так и есть:

$$\begin{array}{rcccccc} & 1 & + & 2 & + \cdots + & n \\ + & n & + & n-1 & + \cdots + & 1 \\ \hline & n+1 & + & n+1 & + \cdots + & n+1 = n(n+1) \end{array}$$

Так как строка 6 — это этап, который будет повторяться наибольшее число раз, $n(n + 1)/2$ является худшим вариантом для времени выполнения работы алгоритма.

Когда мы говорим о времени выполнения работы алгоритма, нас интересует время выполнения при наличии большого объема вводных данных (в нашем случае число n). Такое время работы называется асимптотическим, потому что оно работает с поведением алгоритма, когда вводные данные постоянно и неограниченно увеличиваются. Для такого случая есть особая запись, нотация. Если для всех значений n бóльших, чем некие изначальные положительные значения, функция $f(n)$ меньше или равна другой функции $g(n)$, имеющий шкалу неких положительных постоянных значений c, которые представляют собой $cg(n)$, то для каждой функции $f(n)$ мы говорим, что $O(f(n)) = g(n)$. Если точнее, мы говорим, что $O(f(n)) = g(n)$, если есть положительные постоянные c и n_0, такие как $0 \le f(n) \le cg(n)$ для всех $n \ge n_0$.

Нотация $O(f(n))$ называется «нотацией большого О». Помните, что нас интересуют большие значения наших вводных данных, потому что именно здесь мы сэкономим больше всего. Взгляните на рисунок 1.2, где изображены две функции: $f_1(n) = 20n + 1000$ и $f_2(n) = n_2$. Для небольших значений n это $f_1(n)$, которая принимает наибольшие значения, но ситуация меняется очень стремительно, после чего n_2 растет значительно быстрее.

Нотация большого O позволяет нам упрощать функции. Если у нас есть функция $f(n) = 3n^3 + 5n^2 + 2n + 1000$, тогда мы упрощаем ее до $O(f(n)) = n^2$. Почему?

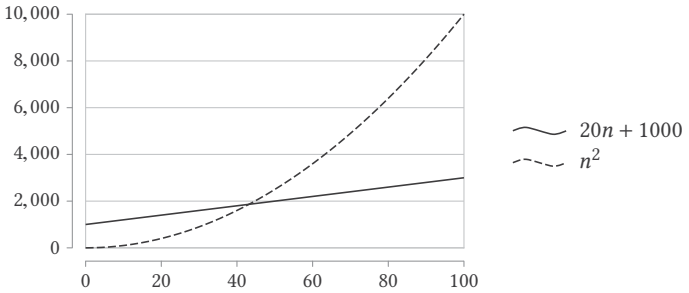


Рисунок 1.2. Сравнение $O(f(n)$

Потому что мы всегда можем подобрать значение c, так что $0 \leq f(n) \leq cn^3$. Обычно, когда у нас есть функция со множеством членов, наибольший член сильно влияет на рост функции, и мы убираем наименьшие члены при работе с большими O. Таким образом, $O(a_1n^k + a_2n^{k-1} + \ldots + a_nn + b) = O(n^k)$.

Описанное нами время выполнения работы алгоритма называется вычислительной сложностью или, коротко, сложностью. Когда мы прибегаем к упрощенным формам функций в изучении времени выполнения работы алгоритмов, оказывается, что время выполнения работы большинства алгоритмов содержит функции, окруженные небольшим количеством упрощенных функций. Это значит, что сложность алгоритмов часто попадает в одну из немногочисленных категорий, или разновидностей.

Прежде всего у нас есть постоянная функция $f(n) = c$. Это всего лишь значит, что данная функция всегда имеет одно и то же значение, c, вне зависимости от того, чем является n. До тех пор, пока c не примет непомерно высокое значение, у нас есть наилучший вариант, который можно желать для нашего алгоритма. Если говорить о нотации большого O, то мы знаем по определению, что существуют положительные постоянные c и n_0, такие как $0 \leq f(n) \leq cg(n) = c \cdot 1$. Действительно, c является константным значением функции, а $n_0 = 1$. Следственно, $O(c) = O(1)$. Алгоритм,

который ведет себя подобным образом, мы называем алгоритмом постоянного времени. На самом деле этот термин употреблен неверно, потому что тут вовсе не имеется в виду, что алгоритм будет всегда требовать неизменного количества времени, независимого от вводных данных. Имеется в виду, что от вводных данных не зависит верхняя граница времени работы алгоритма. Например, простому алгоритму, который добавляет значение y к значению x, если $x > 0$, не всегда требуется одинаковое количество времени для выполнения задачи: если $x > 0$, то он прибавляет нужное значение, а в противном случае ничего не делает. Тем не менее верхняя граница, то есть требуемое для прибавления время, постоянна и попадает в категорию $O(1)$. К сожалению, существует не так уж и много алгоритмов, которые работают за постоянное время. Самые привычные операции, которые исполняются за постоянное время, — это получение доступа к элементу в массиве, который мы считаем постоянным и независимым от индекса выбранного нами элемента; как мы уже видели, в массиве A из n элементов доступ к $A[0]$ занимает столько же времени, сколько и к $A[n - 1]$.

После алгоритмов постоянного времени мы обратимся к алгоритмам, которые выполняются за логарифмическое время. Логарифмическая функция или логарифм — это $\log_a(n)$, которое обозначает степень, в которую мы должны возвести a, чтобы получить n: если $= \log_a(n)$ тогда $n = a^y$. Число a — основа логарифма. Из определения логарифма следует, что $x = a^{log_a x}$, которое показывает, что логарифмирование — это обратное действие от возведения числа в степень. Действительно, $\log_3 27 = 3$ и $3^3 = 27$. Если $a = 10$, то есть логарифм по основанию 10, тогда мы просто пишем $y = \log(n)$. При работе с компьютерами мы часто сталкиваемся с логарифмами по основанию 2, называемые двоичными логарифмами, так что мы используем для них особую запись, $\lg(n) = \log_2(n)$. Они отличны от так называемых натуральных логарифмов с основанием логарифма e, где $e \approx 2{,}71828$. Натуральные логарифмы также имеют особую запись, $\ln(n) = \log_e(n)$.

Сделаем небольшое отступление для тех, кому интересно, откуда взялось число e. Число e часто называется эйлеровым числом, в честь жившего в XVIII веке швейцарского математика

Леонарда Эйлера, оно возникает в различных областях. У числа Эйлера есть форма выражения $(1 + 1/n)^n$, где n стремится к бесконечности. Хоть оно и названо в честь Эйлера, число было открыто другим швейцарским математиком, жившим в XVII веке, Якобом Бернулли. Он пытался вывести формулу для расчета постоянно растущих процентных доходов.

Допустим, вы положили d долларов в банк и вклад приносит вам доход $R\%$. Если процентный доход считается раз в год, тогда через год ваш доход составит $d + d(R/100)$. Берем $r = R/100$, тогда выгода будет $d\ (1 + r)$. Можете удостовериться, что если R = 50, r = 1/2, то ваша выгода вырастет до $1.5 \times d$. Если проценты считаются дважды в год, тогда размер процентной ставки каждые шесть месяцев будет $r/2$. Через шесть месяцев у вас будет $d(1+ r/2)$. Тогда через следующие шесть месяцев, к концу года, у вас будет $d(1 + r/2)(1 + r/2) = d(1 + r/2)^2$. Если процент рассчитывается по сложной процентной ставке, n раз в год, то вы получите $d(1 + r/n)^n$ к концу года. При солидной ставке $R = 100\%$ вы получите $r = 1$; если процент будет бесконечно рассчитываться по сложной процентной ставке, то за ничтожно малое время n возрастет до бесконечности. Тогда если $d = 1$, ваш доллар к концу года вырастет до $(1 + 1/n)^n = e$. Конец отступления.

Основная особенность логарифмов заключается в том, что логарифмы с разными основаниями отличаются друг от друга постоянным множителем, так как $\log_a(n) = \log_b(n)/\log_b(a)$. Например, $\lg(n) = \log_{10}(n)/\log_{10}(2)$. Таким образом, мы объединяем все логарифмические функции в категорию с общей сложностью, которую мы обычно обозначаем $O(\log(n))$, хотя более специфичное $O(\lg(n))$ тоже встречается довольно часто. Сложность $O(\lg(n))$ появляется, когда алгоритм постоянно разделяет задачу пополам, потому что если вы раз за разом разделяете что-то надвое, то по сути вы прибегаете к логарифмической функции. Значимыми алгоритмами логарифмического времени являются алгоритмы, связанные с поиском: быстрейшие поисковые алгоритмы работают за логарифмическое по основанию 2 время.

Еще большего времени, чем алгоритмы логарифмического времени, требуют алгоритмы линейного времени, которые выполняются за $f(n) = n$ время, иными словами, за время пропорциональ-

ное их вводным данным. Для этих алгоритмов сложность составляет $O(n)$. Для получения ответа им приходится просматривать все вводные данные по порядку. Например, если мы ищем ряд случайных элементов, не заданных ни в каком порядке, тогда нам придется перебрать их все, чтобы найти тот, который нам нужен. Таким образом происходит поиск линейного времени.

Медленней алгоритмов линейного времени — линейно-логарифмические, где $f(n) = n \log(n)$, поэтому мы пишем $O(n \log(n))$. Как говорилось выше, у логарифмов могут быть различные основания, хотя на практике намного чаще используются логарифмы с основанием 2. Данные алгоритмы в некотором роде являются смесью алгоритмов линейного времени и алгоритмов логарифмического времени. Они постоянно разделяют задачу надвое и используют алгоритм линейного времени для каждой из частей. Алгоритмы, хорошо справляющиеся с сортировкой, всегда имеют линейно-логарифмическую временную сложность.

Когда функция, описывающая время выполнения алгоритма, полиноминальная $f(n) = (a_1n^k + a_2n^{k-1} + \ldots + a_nn + b)$, мы, как видно выше, имеем дело со сложностью $O(n^k)$ и алгоритмом полиноминального времени. Многие алгоритмы выполняются за полиноминальное время; важный их подвид — алгоритмы, которые работают за $O(n^2)$ время, их называют алгоритмами квадратичного времени. Некоторые методы сортировки квадратичного времени работают непродуктивно, так как используется обычный способ умножения двух чисел с n разрядами у каждого — обратите внимание, что существуют более современные и эффективные способы умножения чисел, и мы используем эти более эффективные способы, ожидая проведения точнейших арифметических расчетов.

Еще медленнее, чем алгоритмы полиноминального времени, алгоритмы экспоненциального времени, где $f(n) = c^n$ и c имеет постоянное значение, поэтому $O(c^n)$. Обязательно обратите внимание на разницу между n^c и c^n. Хотя мы поменяли местами n и показатель степени, это сильно сказывается на результате функции. Как мы говорили, возведение в степень — обратный процесс от логарифмирования и он попросту увеличивает константу до непомерного числа. Будьте внимательны: возведение в степень —

это c^n; а степенная зависимость — это особый случай, где $c = e$, то есть $f(n) = e^x$, где e является числом Эйлера. Возведение в степень случается, когда нам приходится иметь дело с проблемой вводных данных n, где каждые n данных могут принять c разных значений, и нам нужно рассмотреть все возможные случаи. У нас есть c значений для первого ввода данных, и для каждого из случаев у нас есть c значений для второго ввода данных; в общей сложности $c \times c = c^2$. Для каждого из случаев c^2 у нас есть c возможных значений для третьего ввода данных, который делает $c^2 \times c = c^3$; и так до последнего ввода данных, который дает c^n.

Таблица 1.1. Рост функций

функция	размер вводных данных				
	1	10	100	1000	1,000,000
$\lg(n)$	0	3.32	6.64	9.97	19.93
n	1	10	100	1000	1,000,000
$n\ln(n)$	0	33.22	664.39	9965.78	1.9×10^7
n^2	1	100	10,000	1,000,000	10^{12}
n^3	1	1000	1,000,000	10^9	10^{18}
2^n	2	1024	1.3×10^{30}	10^{301}	$10^{10^{5.5}}$
$n!$	1	3,628,800	9.33×10^{157}	4×10^{2567}	$10^{10^{6.7}}$

Еще медленнее, чем алгоритмы экспоненциального времени, алгоритмы факториального времени с $O(n!)$, где факториальное число задано как $n! = 1 \times 2 \times \ldots \times n$ и в вырожденном случае $0! = 1$. Факториал появляется на сцене, когда для решения задачи нам нужно перебрать все возможные комбинации вводных данных. Комбинация — это иное расположение последовательности значений. Например, если мы имеем значения [1, 2, 3], тогда у нас есть следующие комбинации: [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2] и [3, 2, 1]. В первой позиции есть n возможных значений; затем, так как мы использовали одно значение, есть $n - 1$ возможных значений для второй позиции; это создает $n \times (n - 1)$ различных комбинаций для первых двух позиций. Мы в том же духе продолжаем перебор для оставшихся позиций, пока в последней не останется единственно возможное значение. В итоге у нас $n \times (n - 1) \ldots \times 1 = n!$ Мы встречаем факториальное число при раздаче коло-

ды игральных карт: число возможных тасовок в карточной колоде — 52, и это просто заоблачное число.

Практика показывает, что алгоритмы с полиномиальной временной сложностью хороши в работе, так что зачастую наша задача заключается в том, чтобы отыскать алгоритмы с такой вот производительностью. К сожалению, мы не знаем ни одного алгоритма полиномиального времени, который подошел бы для решения целого круга важнейших задач. Посмотрите на таблицу 1.1; вам должно стать понятно, что если для задачи у нас есть только алгоритм со временем выполнения $O(2^n)$, то такой алгоритм довольно бесполезен и годится разве что для пустячных задач с малыми вводными значениями. Также обратите внимание на рисунок 1.3; в нижнем ряду $O(2^n)$ и $O(n!)$ начинают стремительно увеличиваться для малых значений n.

На рисунке 1.3 линиями изображены графики функций, хотя на деле число n, когда мы изучаем алгоритмы, является натуральным числом, так что нам бы следовало ожидать график рассеяния с точками, а не линиями. Логарифмические, линейные, линейно-логарифмические и полиномиальные функции задаются напрямую для действительных чисел, поэтому их совершенно спокойно можно изобразить линиями, используя нормальные функциональные определения. Обычное объяснение возведения в степень касается целых чисел, однако также возможна и степень с рациональным показателем, потому что $x^{(a/b)} = (x^a)^{(1/b)} = \sqrt[b]{x^a}$. Степени с показателями, являющимися действительными числами, определяются как $b^x = (e^{\ln b})^x = e^{x \ln b}$. Что касается факториалов, то, углубляясь чуть дальше в высшую математику, получается, что они тоже могут быть заданы для всех действительных чисел (отрицательные факториалы считаются равными бесконечности). Таким образом, изображение функций сложности с помощью линий вполне оправданно.

Чтобы вы не подумали, что сложность $O(2^n)$ или $O(n!)$ редко встречаются на практике, рассмотрите известную (скорее даже печально известную) задачу коммивояжера. Суть задачи в том, что коммивояжер должен посетить определенное количество городов, заезжая в каждый город только один раз. Каждый го-

род напрямую связан с каждым другим (возможно, коммивояжер путешествует самолетом). Загвоздка в том, что коммивояжер должен найти самый короткий маршрут. Решение в лоб — это перепробовать все возможные комбинации с очередностью городов. Для n городов — $O(n!)$. Есть алгоритм и получше, он решает задачу через $O(n^2 2^n)$ — хоть он и быстрее, однако на практике разница едва ли заметна. Так как же нам решить эту (и прочие похожие) задачи? Оказывается, что, хотя мы и не знаем хорошего алгоритма, который выдаст нам точный ответ, нам могут быть известны хорошие алгоритмы, которые дадут приблизительные результаты.

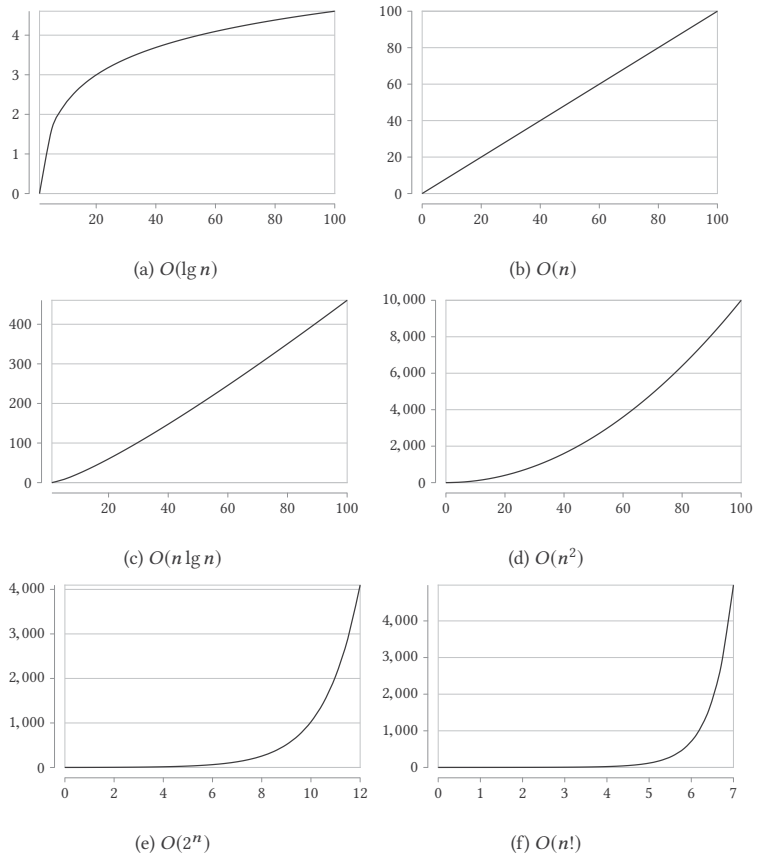


Рисунок 1.3. Различные категории сложности

Большое О задает верхнюю границу производительности алгоритма. Противоположная ей нижняя граница, когда мы знаем, что сложность всегда будет не лучше, чем у заданной функции, после нескольких исходных значений. Такое поведение называется «большая Омега», или $\Omega(f(n))$, а точное ее определение: $\Omega(f(n)) = g(n)$, если существует положительные константы c и n_0, такие как $f(n) \geq cg(n) \geq 0$ для всех $n \geq n_0$. Имея заданные большое О и большую Омегу, мы можем также задать ситуацию, где у нас есть обе, и верхняя, и нижняя границы. Такая ситуация называется «большая Тета», и мы говорим, что $\Theta(f(n)) = g(n)$, если и только если $O(f(n)) = g(n)$ и $\Omega(f(n)) = g(n)$. Тогда мы знаем, что время выполнения алгоритма ограничено сверху и снизу одной и той же функцией, зависящей от константы. Можете считать это временем выполнения алгоритма, лежащим в области этой функции.

1.3. Используем стек для разницы курсов акций

Вернемся к задаче о разнице курсов акций. Мы нашли алгоритм со сложностью $O(n(n + 1/2))$. Из написанного выше следует, что он равноценен $O(n^2)$. Можем ли мы сделать лучше? Вернемся к рисунку 1.1. Обратите внимание: когда мы на шестом дне, нам не надо сравнивать все предыдущие дни до первого, так как мы уже «знаем», что стоимость во второй, третий, четвертый и пятый дни ниже или равна стоимости в день шестой. Если мы каким-то образом сохраним это знание, то вместо того, чтобы заниматься лишними сравнениями, нам надо будет сопоставить только курс акций в шестой и первый дни.

Мы коснулись базового момента. Представьте, что вы находитесь в дне k. Если цена акций в день $k - 1$ ниже или равны цене в день k так, что цены $quotes[k - 1] \leq$ цен $quotes[k]$ или то же самое для цен $quotes[k] \geq quotes[k - 1]$, тогда не имеет смысла еще раз сравнивать с $k - 1$. Почему? Заглянем в будущее, в день $k + 1$. Если курс в день $k + j$ меньше или равен курсу в день k, $quotes[k+ j] <$ цены$[k]$, тогда нам не надо сравнивать с $k - 1$, так как промежуток разницы, который начинается с $k + j$, заканчивается в k. Если цена в $k + j$ больше цены в k, тогда мы понимаем, что она должна быть $quotes[k + j] \geq quotes[k - 1]$,

потому что $quotes[k + j] \geq quotes[k]$ и $quotes[k] \geq quotes[k - 1]$. Так что каждый раз, когда мы возвращаемся в прошлое в поисках конца отрезка, обозначающего разницу, нам нужно отбросить все дни со значениями меньше или равными дню, в который мы высчитываем разницу, и исключить отброшенные дни из последующего анализа.

Для большей ясности приведем метафору. Представьте, что сидите на верхушке колоны, которая изображена на рисунке 1.4 и обозначает шестой день. Вы глядите не вниз, а назад, и видите перед собой только колонну, обозначающую день первый. Она — единственное, с чем вам надо сравнивать стоимость акций в шестой день. Таким образом, вам всегда нужно сравнивать нынешний день только с тем, который попадает в поле вашего зрения.

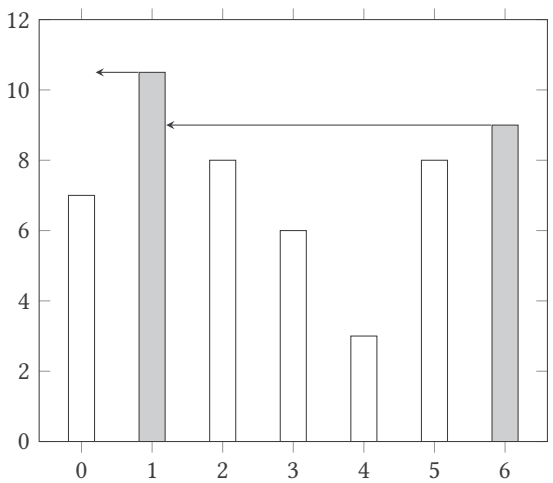


Рисунок 1.4. Оптимизированная разница курсов акций

Получается, мы впустую тратим время в алгоритме 1.1, когда во внутреннем цикле, идущем от строчки 5, начинаем сравнивать все предыдущие дни. Мы можем избежать лишней работы, используя механизм, с помощью которого у нас всегда будут наготове границы самых больших промежутков.

Помочь в нашей работе может специальная структура для хранения данных, которая называется стеком. Стек — простая структура данных. Такая вещь, на которую мы можем сложить

одни за другими все данные, а затем забрать их. Забирать их можно только в обратном порядке, начиная с последних, положенных в стек. Стек устроен так же, как стопка подносов в ресторане: мы можем взять только верхний поднос, а если хотим положить новый, то кладем только сверху. Последний поднос, положенный в стопку, вынимается первым, поэтому стек называют структурой с принципом «последним зашел — первым вышел». На рисунке 1.5 изображены похожие на подносы операции с добавлением и извлечением элементов стека.

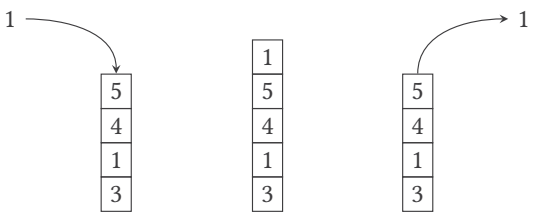


Рисунок 1.5. Добавление и извлечение из стека

Когда мы говорим о структурах данных, нам нужно описать операции, которые мы способны производить над ними. Для массивов мы рассмотрели две операции: одна — создание массива, вторая — добавление элементов. Для стеков, учитывая все вышесказанное, существует пять операций.

- CreateStack() создает пустой стек.
- Push(*S*, *i*) добавляет элемент *i* в самый верх стека *S*.
- Pop(*S*) извлекает верхний элемент стека S. Операция возвращает элемент. Если стек пуст, тогда операция не может быть выполнена (мы получим ошибку).
- Top(*S*) мы получаем значение верхнего элемента стека *S*, не извлекая при этом сам элемент. Стек не изменяется. Если стек пуст, тогда, опять же, операция не может быть выполнена и мы увидим сообщение об ошибке.
- IsStackEmpty(*S*) возвращает `true`, если стек *S* пуст, или `false`, если стек *S* полон.

На самом деле стек не безграничен: мы можем поместить в него определенное количество элементов, пока он не запол-

нится: в конце концов, у памяти компьютера есть свои пределы. При реализации стека есть дополнительные операции для проверки количества элементов в стеке (его размер) и его заполненности: есть ли в нем еще место или нет. К нашим алгоритмами, использующим псевдокод, это не имеет никакого отношения, поэтому в дальнейшем мы не будем их касаться; точно так же мы поступим с прочими схожими операциями для других структур данных.

С помощью стека мы можем справиться с задачей о разнице курсов акций, применив задумку, которую мы развили в алгоритме 1.2. Как и прежде, в строчке 1 мы создаем массив на n элементов. Разница в первый день по умолчанию равна единице, поэтому, согласно строчке 2, мы запускаем *spans*[0]. Теперь мы используем стек, чтобы поместить в него все дни, которые нужно сравнить, поэтому в строчке 3 мы создаем новый пустой стек. В начале нам известен лишь очевидный факт, что цена акций в первый день не ниже их цены в первый день, поэтому в строчке 4 мы кладем в стек индекс первого дня, ноль.

Алгоритм 1.2. Алгоритм вычисления разницы курсов акций с помощью стека

```
StackStockSpan(quotes) → spans
    Вводные данные: quotes, массив с n курсами акций.
    Выводимые данные: spans, массив с n разницами курсов акций.
1   spans ← CreateArray(n)
2   spans[0] ← 1
3   S ← CreateStack()
4   Push(S, 0)
5   for i ← 1 to n do
6       while not IsStackEmpty(S) and quotes[Top(S)] ≤ quotes[i] do
7           Pop(S)
8       if IsStackEmpty(S) then
9           spans[i] ← i + 1
10      else
11          spans[i] ← i − Top(S)
12      Push(S, i)
13  return spans
```

Цикл в строчках 5–12 работает со всеми последующими днями. Внутренний цикл в строчках 6–7 заглядывает в прошлое, чтобы найти последний день, когда курс акций был выше, чем в выбранный нами день. Он проделывает это, вынимая объекты из стека (строчка 7) до тех пор, пока у дня, находящегося вверху стека, не будет меньшая или равная цена по сравнению с выбранным нами днем (строчка 6). Если мы выйдем из внутреннего цикла после того, как стек опустел (строчка 8), и будем находиться в дне *i*, тогда во все предыдущие дни цена акций будет ниже, поэтому разница будет $i + 1$. В строчке 9 мы устанавливаем $spans[i]$ для данного значения. Иначе (строчка 10) промежуток тянется дальше от дня *i* до дня, находящегося наверху стека, так что в строчке 11 мы используем разницу между двумя установками $spans[i]$. Прежде чем вернуться к началу цикла, мы убираем *i* с верхушки стека. Таким образом в конце внешнего цикла в стеке останутся дни, в которые цена акций не ниже, чем в выбранный день. Это позволит нам при следующем проходе цикла сравнивать только нужные дни: те, которые которые попадают в поле нашего зрения и которые мы как раз искали.

В строчке 6 нашего алгоритма есть один момент, на который стоит обратить внимание. Речь об ошибке при оценке $\text{Top}(S)$, если S пуст. Такого не случится благодаря важному свойству, касающемуся оценки условий, которое называют упрощенным вычислением. Данная характеристика означает, что, когда мы вычисляем выражение, в котором предполагаются логические булевы операторы, вычисление такого выражения прекращается, как только мы получаем итоговый результат: мы не затрудняем себя просмотром остальных частей выражения. Например, выражение **если** $x > 0$ **и** $y > 0$. Если мы знаем, что $x \leq 0$, тогда все выражение ложно, вне зависимости от значения y; нам вовсе не нужно вычислять вторую часть выражения. Точно так же с выражением **если** $x > 0$ **или** $y > 0$, если мы знаем, что $x > 0$, то нам не надо вычислять вторую часть выражения с y, так как мы уже знаем, что все выражение истинно, потому что верна первая его часть. Таблица 1.2 показывает общую ситуацию для всех логических выражений, состоящих из двух частей и содержащих операторы **и** либо **или**.

Закрашенные серым ряды показывают, что результат выражения не зависит от второй части и, следовательно, вычисление можно прекратить, как только нам станет известно значение первой части. С помощью упрощенного вычисления, когда IsStackEmpty(*S*) показывает `true`, означающее, что **не** IsStackEmpty(*S*) является `false`, мы не будем пытаться вычислить находящееся справа от **и**, содержащее Top(*S*), тем самым избегая ошибку.

Таблица 1.2. Упрощенное булево вычисление

оператор	*a*	*b*	результат
и	T	T	T
	T	F	F
	F	T/F	F
или	T	T/F	T
	F	T	T
	F	F	F

На рисунке 1.6 вы можете увидеть, как работает наш алгоритм и метафора «взгляда с колонны». На каждом графике мы показываем справа стек в начале каждой итерации цикла; мы также отмечаем закрашенными столбцами дни, находящиеся в стеке, а дни, с которыми мы еще не разобрались, отмечаем пунктирными столбцами. Выбранный нами день, с которым мы работаем, выделен внизу черным кругом.

На первом графике $i = 1$, и мы должны сравнить значение выбранного дня со значениями других дней, находящихся в стеке: а в стеке у нас только нулевой день. В первый день цена выше, чем в нулевой. Это означает, что с нынешнего момента не нужно сравнивать дни до первого дня — наш взгляд с колонны остановится на нем; потому при следующей итерации цикла, когда $i = 2$, в стеке содержится число 1. В день второй цена ниже, чем в день первый. Это означает, что любой промежуток, начавшийся с третьего дня, может закончиться на втором дне, если значение третьего дня ниже, чем значение второго; или же он может закончиться на первом дне, если значение третьего дня не меньше, чем во второй день. Но он никак не может закончиться на нулевом дне, даже если цена в нулевой день меньше, чем в первый день.

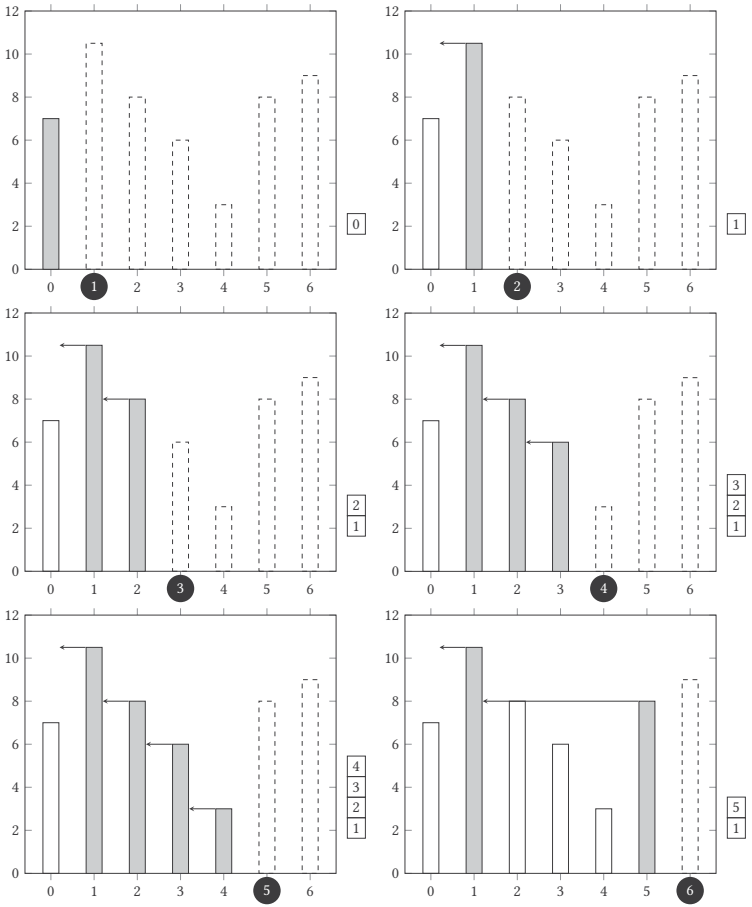


Рисунок 1.6. Взгляд на промежуток, обозначающий разницу курсов акций

Похожая ситуация возникает с $i = 3$ и $i = 4$. Но когда мы доходим до $i = 5$, мы понимаем, что больше не нужно сравнивать второй, третий и четвертый дни — они покрыты тенью пятого дня. Если вернуться к метафоре взгляда с колонны, ничто не закрывает наш обзор вплоть до первого дня. Все, что находится между пятым и первым днем, можно убрать из стека, в котором останутся лишь 5 и 1, таким образом, $i = 6$ нам нужно будет сравнить только с этими двумя днями. Если цена в какой-либо день больше или

равна цене в день пятый, она уж точно больше стоимости в четвертый, третий и второй дни; чего мы не можем сказать наверняка — сможет ли она достигнуть значения первого дня. Когда мы доходим до шестого дня, в стеке оказываются числа 6 и 1.

Получилось у нас сделать лучше? В алгоритме 1.2 цикл, который запускается в строчке 5, выполняется n – 1 раз. Для каждого раза, скажем, при итерации i цикла, во внутреннем цикле, который запускается в строчке 6, операция Pop выполняется p_i раз. Это означает, что всего операция Pop будет выполнена $p_1 + p_2 + \ldots + p_{n-1}$ раз, p_i раз для каждой итерации внешнего цикла. Мы не знаем, что из себя представляет число p_i. Однако если вы присмотритесь к алгоритму, то заметите, что каждый день добавлен в стек единожды, первый день в строчке 3 и последующие дни в строчке 11. Следовательно, каждый день может быть извлечен из стека в строчке 7 только один раз. Таким образом, на протяжении всего выполнения алгоритма, во время всех итераций внешнего цикла, мы не можем выполнить строчку 6 более n раз. Иными словами, $p_1 + p_2 + \ldots + p_{n-1} = n$, и это означает, что весь алгоритм равен $O(n)$; строчка 7 — это операция, которая будет повторяться наибольшее количество раз, так как она находится во внутреннем цикле, в отличие от остального кода в строчках 5–12.

Мы можем продолжить наш анализ и увидеть, что в сравнении с алгоритмом 1.1, в котором мы смогли прийти только к худшему варианту, здесь наша оценка также достигает нижней границы производительности алгоритма: так как нам надо пройти через n-е количество дней, алгоритм не может быть выполнен за менее чем n этапов. Таким образом, вычислительная сложность нашего алгоритма тоже $\Omega(n)$ и, стало быть, $\Theta(n)$.

Стеки, как и все структуры данных, с которыми мы познакомимся, имеют широкий спектр применения. Поведение «последним зашел — первым вышел» довольно часто встречается в программировании, так что стеки будут попадаться вам везде: от простейших программ, написанных на машинных языках, до объемных задач, решаемых с помощью сверхмощных компьютеров. Именно поэтому вообще и существуют структуры данных. Они являются выжимкой многолетнего опыта решения задач

с помощью компьютера. Снова и снова получается так, что алгоритмы используют схожие способы для организации данных, которые они обрабатывают. Люди перевели эти способы в код так, что, когда мы ищем разные подходы к решению задачи, мы прибегаем их возможностям, чтобы разработать алгоритмы.

Примечания

Самой объемной и полнейшей работой, посвященной алгоритмам, является многотомник Дональда Кнута [112, 113, 114, 115] — труд длинною в пятьдесят лет, и он до сих пор не окончен: можно написать еще многие тома, посвященные тому, чего не коснулись уже существующие книги, — они охватывают далеко не все области алгоритмов, но то, что охватывают, изложено в них строгим и непревзойденным стилем.

Исчерпывающим классическим введением в алгоритмы является книга Кормена, Лейзерсона, Ривеста и Штайна [42]. Томан Кормен также написал еще одну популярную книгу [41], которая более коротко и понятно знакомит читателя с основными алгоритмами. Еще одно хорошее введение в алгоритмы для новичков — книга Маккормика [130]. Другие популярные книги по знакомству с алгоритмами — книги Клейберга и Тардоша [107], Дасгупты, Пападимитриу и Вазирани [47], Харела и Фельдмана [86] и Левитина [129].

Есть еще много прекрасных книг, которые рассказывают об алгоритмах и их реализации в разных языках программирования [180, 176, 178, 177, 179, 188, 82].

Стеки стары, как сами компьютеры. Согласно Кнуту [112, с. 229 и 459], Алан Тьюринг описал их при проектировке автоматической вычислительной машины (ACE) в 1945 году и явил миру в 1946 году; операции стека называли BURY и UNBURY («погрузить» и «выгрузить») вместо «push» и «pop» («запихнуть» и «вытолкнуть») [205, с. 11–12 и 30].

Алгоритмы намного древнее компьютеров и известны со времен Вавилона [110].

Упражнения

1. Стек — несложная для реализации структура данных. Простейшая ее реализация использует массив; напишите реализацию стека на основе массива. Ранее мы говорили о том, что на практике стек имеет куда больше операций, чем пять, упомянутых в данной книге: операции, возвращающие размер стека и проверяющие его наполненность. Убедитесь, что их вы тоже реализовали.
2. Мы рассмотрели два решения для задачи о разнице курсов акций: одна использует стек, а другая обходится без него. Мы говорили, что способ с применением стека быстрее. Проверьте это на практике: реализуйте два алгоритма на выбранном вами программном языке и засеките, сколько времени потребуется на каждое решение задачи. Обратите внимание, что при расчете времени выполнения программы вы должны предоставить достаточно данных, чтобы ей потребовалось разумное количество времени на выполнение работы; учитывая, что в компьютере параллельно происходит множество процессов и время выполнения может попадать под влияние разных факторов, вам нужно не один раз замерить время, чтобы в итоге получить достоверный результат. Так что у вас отличная возможность узнать, как тестируются программы.
3. Стеки используются для реализации арифметических вычислений, написанных на обратной польской нотации (ОПН), также известной как постфиксная запись. В ОПН идут сначала операнды, а потом их операторы, в отличие от инфиксной записи, где операторы располагаются между операндами. Так что вместо 1 + 2, мы пишем 1 2 +. «Польской» нотация называется в честь Яна Лукасевича, который в 1924 году изобрел польскую, или префиксную, запись, на ней у нас получится + 1 2. Преимущество ОПН состоит в том, что здесь не нужны скобки: 1 + (2 × 3) из инфиксной записи превращается в 1 2 3 * + постфиксной записи. Чтобы вычислить результат, мы читаем запись слева

направо. Мы добавляем числа в стек. Когда мы сталкиваемся с оператором, мы извлекаем из стека столько элементов, сколько нужно в качестве операндов, мы производим операцию и помещаем результат в стек. В итоге результат оказывается на самом верху стека (и единственным его элементом). Например при вычислении 1 2 3 * + 2 – стек, написанный горизонтально в квадратных скобках, становится [ ], [1], [1 2], [1 2 3], [1 6], [7], [7 2], [5]. Напишите калькулятор, который вычисляет арифметические выражения, заданные пользователем в ОПН.

4. Во многих языках программирования мы имеем дело с выражениями, выделенными символами-разграничителями, такими как круглые скобки ( ), квадратные скобки [ ] или фигурные скобки { }. Напишите программу, которая читает последовательность символов-разграничителей, такую как ( ) { [ ] ( ) { } }, и сообщает, когда разграничители симметричны или когда несимметричны, например (( ); или когда разграничители разных видов стоят в паре, как (}. Используйте стек, чтобы запомнить открытые в настоящий момент символы-разграничители.

2 Исследуя лабиринты

Как найти выход из лабиринта — это одна из древнейших проблем. История рассказывает нам о критском царе Миносе, который приказал афинянам отправлять к нему каждые семь лет по семь юношей и семь девушек. Жертв бросали в подземелье, находившееся под дворцом Миноса, на расправу чудовищу с телом человека и головой быка — Минотавру. Подземелье представляло собой огромный лабиринт, и несчастные пленники были обречены на встречу с Минотавром и смерть от его рук. Когда в Афины в третий раз приплыл корабль, чтобы забрать юношей и девушек, Тесей добровольцем отправился на Крит. Там он очаровал дочь Миноса, Ариадну, и она дала ему клубок. Пока Тесей шел по коридорам лабиринта, он разматывал нить, когда же он нашел и убил Минотавра, нить помогла ему найти обратную дорогу.

Лабиринт интересен не только как мифическая история или красивое украшение для парков. Он представляет собой одну из многочисленных ситуаций, в которых нам приходится исследовать пространство, соединенное особыми путями. Самый простой пример — сеть городских дорог; однако проблема становится еще интересней, когда мы осознаем, что порой таким же образом нам нужно исследовать абстрактные вещи. У нас может быть компьютерная сеть, в которой машины соединены друг с другом, и нам надо узнать, как один из компьютеров соединен с другим. Точно так же у нас может быть окружение, иными словами, люди, которые окружают нас и которые связаны друг с другом, и нам нужно узнать, как через одного человека нам выйти на другого.

Согласно мифу, для того, чтобы найти выход, нам нужно знать места, где мы уже побывали, в противном случае выбраться не удастся. Давайте возьмем примерный лабиринт и попробуем придумать стратегию выхода. На рисунке 2.1 изображен лаби-

ринт, где комнаты обозначены кругами, а коридоры — соединяющими их линиями.

На рисунке 2.2 показано, что происходит, когда мы исследуем лабиринт систематически, следуя особой стратегии, которая называется «рукой по стене».

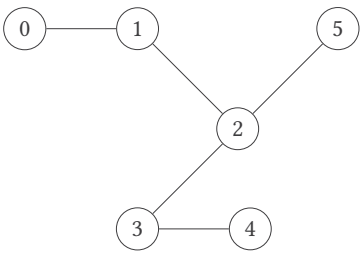


Рисунок 2.1. Лабиринт

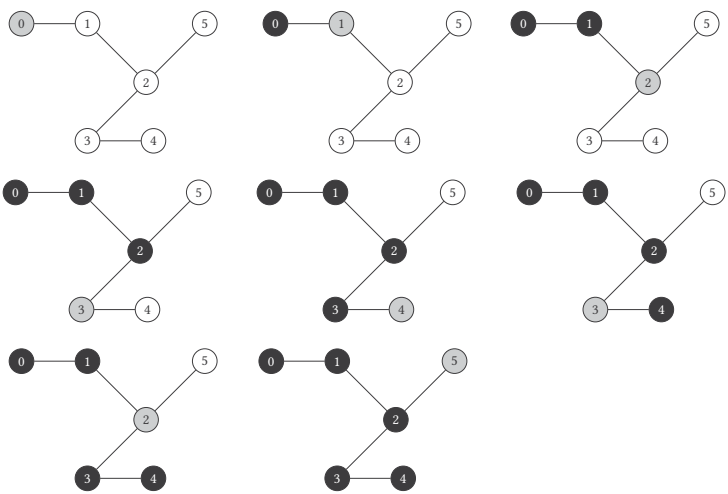


Рисунок 2.2. Стратегия «не отпускай руку от стены»: она работает!

Комната, в которой мы находимся, отмечена серым цветом, а те, в которых мы уже побывали, — черным. Данная стратегия очень проста. Кладем руку на стену и идем, не отпуская ее. Переходя из комнаты в комнату, ни в коем случае не отпускаем руку. Как видно, стратегия работает. Но посмотрим теперь на лабиринт

рисунка 2.3. Если мы будем придерживаться все той же стратегии, мы пройдем лишь по комнатам, расположенным по периметру лабиринта, и пропустим комнату в центре, что и показано на рисунке 2.4.

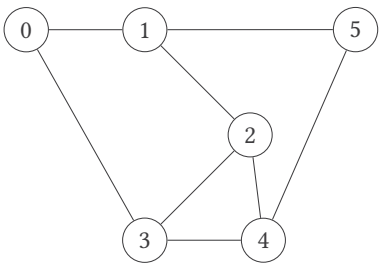


Рисунок 2.3. Еще одни лабиринт

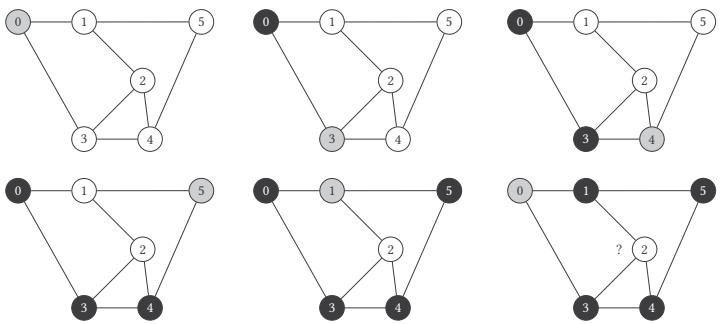
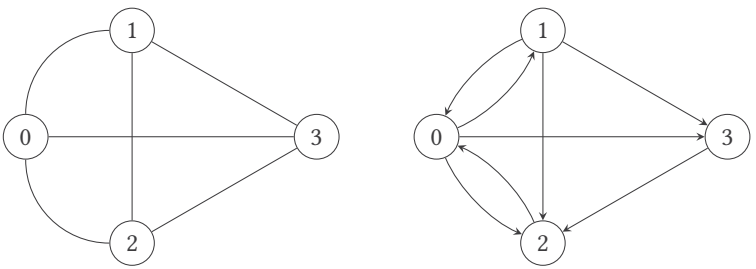


Рисунок 2.4. Стратегия «не отпускай руку от стены»: не работает…

2.1. Графы

Прежде чем перейти к решению задачи с данным лабиринтом, нам нужно рассмотреть лабиринты в целом. Мы сказали, что лабиринты состоят из комнат и коридоров между ними. Мы также отметили, что лабиринты становятся интересней, когда мы замечаем в них сходство с другими знакомыми структурами; на самом деле лабиринты похожи на все, что состоит из объектов и связей между этими объектами. Они являются фундаментальной структурой данных, возможно, самой основной из всех возможных, так

как множество вещей в нашем мире могут быть описаны в виде объектов и связей между этими объектами. Такие структуры называются графами. Коротко говоря, графы — это группа узлов и связок, соединяющих эти узлы. Или, по-другому, вершинами и ребрами. Ребро соединяет две вершины. Последовательность ребер, в которой каждые два ребра соединяются одним узлом, называется путем.



(a) Неориентированный граф (b) Ориентированный граф

Рисунок 2.5. Ориентированный и неориентированный графы

Таким образом, на рисунке 2.2 изображен путь, проходящий через узел 1 и соединяющий узлы 0 и 2. Число ребер пути называется длинной. Одно ребро — это путь с длинной 1. Если путь пролегает между двумя узлами, то мы говорим, что два узла соединены. В некоторых графах нам требуется задать направление ребрам; такие графы называются ориентированными графами, или орграфами. В противном же случае мы имеем дело с неориентированными графами. На рисунке 2.5 слева изображен неориентированный граф, а справа — ориентированный. Как вы видите, тут может быть несколько разных ребер, которые начинаются и заканчиваются в одном и том же узле. Число ребер, прилежащих к узлу, называется степенью вершины. В ориентированных графах у нас есть входящая степень, для входящих ребер, и исходящая степень, для исходящих ребер. На рисунке 2.5(a) все ребра со степенью 3. На рисунке 2.5(b) входящая степень самого правого узла 2, а исходящая — 1.

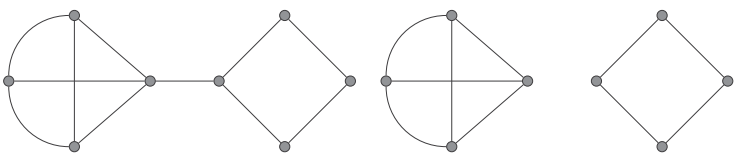
Применению графов можно посвятить целые собрания сочинений: с их помощью можно описать невероятное количество

вещей и обрисовать множество задач, существует огромное число алгоритмов, которые выполняют задачи, связанные с графами. Все потому, что в нашем мире многие вещи состоят из объектов и связей между ними; данная тема требует дальнейшего рассмотрения.

Возможно, самое распространенное применение графов — это описание сетей. Точки сети — это узлы графа, а связки — ребра между ними. Существует большое количество разных видов сетей; в первую очередь компьютерные сети, где все компьютеры соединены друг с другом, тут же и транспортные сети, которые с помощью автомагистралей, авиа- и железнодорожных путей соединяют друг с другом города. В плане компьютерных сетей самым ярким примером будет Интернет, да и название Сеть говорит само за себя: она состоит из страниц-узлов, соединенных друг с другом гиперссылками. «Википедия» — особо огромная сеть, разновидность глобальной Сети. Если заглянуть в область электроники, электросхемы состоят из электротехнических компонентов, таких как транзисторы, соединенных цепями. В биологии мы встречаем метаболические сети, которые, помимо прочего, содержат метаболические пути: химические вещества связаны химическими реакциями. Социальные сети представлены как графы, в которых узлами являются люди, а ребрами — их отношения. Распределение работы между людьми и машинами тоже можно смоделировать с помощью графов: задания — это узлы, а взаимосвязи между ними, например распределение приоритетов заданий, можно считать ребрами.

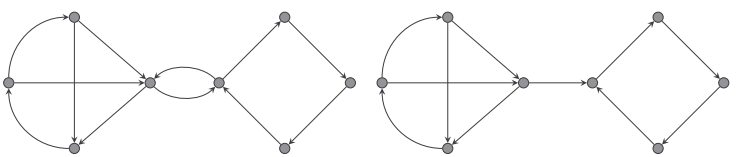
При бесчисленном множестве применений существуют еще другие виды графов, которые подходят для представления различных ситуаций. Если в графе имеется путь от одного узла к другому, то такой граф называется связным. В противном случае речь идет о несвязном графе. На рисунке 2.6 изображены связный и несвязный графы, оба неориентированные. Обратите внимание, что в ориентированных графах мы должны учитывать направление ребер, чтобы определять, с какими узлами они соединены. Ориентированный граф, имеющий ориентированный путь между двумя узлами, называется сильно связным. Если мы

забудем о направлениях и нам станет интересно, есть ли между двумя узлами неориентированный путь, то речь пойдет о слабо связном графе. Если ориентированный граф не сильно и не слабо связный, то мы имеем дело с несвязным графом. На рисунке 2.7 изображены подобные варианты. Вопрос связности графов встает каждый раз, когда мы хотим рассмотреть что-либо, смоделированное в виде графа, что представляет собой целостный объект или же нечто составленное из отдельных элементов. Соединенные элементы в неориентированных графах и сильно связные элементы в ориентированных графах называются связными компонентами. Таким образом, граф является связным (или, если речь идет об ориентированном графе, сильно связном), когда он включает в себя одну связную компоненту. Схожий вопрос касается доступности: возможности дотянуться от одного узла к другому.

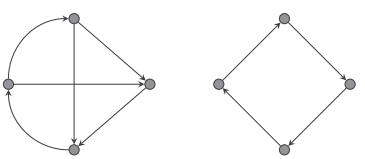


(a) Связный неориентированный граф (b) Несвязный неориентированный граф

Рисунок 2.6. Неориентированные связные и несвязные графы



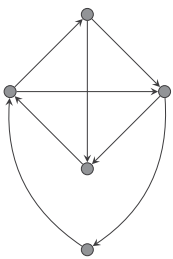
(a) Сильно связный ориентированный граф (b) Слабо связный ориентированный граф



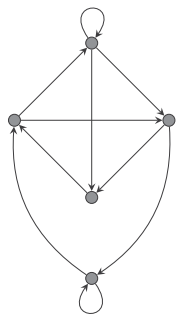
(c) Несвязный ориентированный граф

Рисунок 2.7. Связные и несвязные ориентированные графы

В ориентированном графе, или орграфе, может случиться так, что мы начнем движение от одного узла, будем скакать с ребра на ребро и в итоге окажемся на том же узле, с которого начинали путь. В таком случае мы сделали круг, и наш путь называется циклом. В неориентированных графах мы всегда можем сделать шаг назад и вернуться к точке старта, поэтому мы говорим, что сделали круг только в том случае, если вернулись к начальному узлу, не проходя по ребру в обратном направлении. Графы с циклами называются циклическими, а графы без циклов — ациклическими. На рисунке 2.8 изображены два ориентированных графа с несколькими циклами. Заметьте, что у правого графа есть несколько ребер, которые начинаются и заканчиваются в одном и том же узле. Длина таких циклов равна единице, и мы зовем их петлями. Они бывают и в неориентированных графах, но встречаются не так уж и часто. У ориентированных ациклических графов широкое применение, и у них есть отдельное короткое название: ориентированные ациклические графы сокращаются до ОАГ. На рисунке 2.9 изображены два ациклических графа: неориентированный и ОАГ.



(a) Ориентированный циклический граф

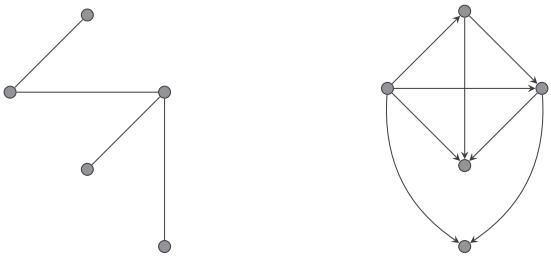


(b) Ориентированный циклический граф без петель

Рисунок 2.8. Циклические ориентированные графы

Если узлы графа возможно разделить на две доли таким образом, что все ребра соединяют узел из одной доли с узлом из другой доли, то речь у нас идет о двудольном графе. Самое известное использование двудольного графа — отождествление, когда нам надо сравнить между собой две доли с элементами (например,

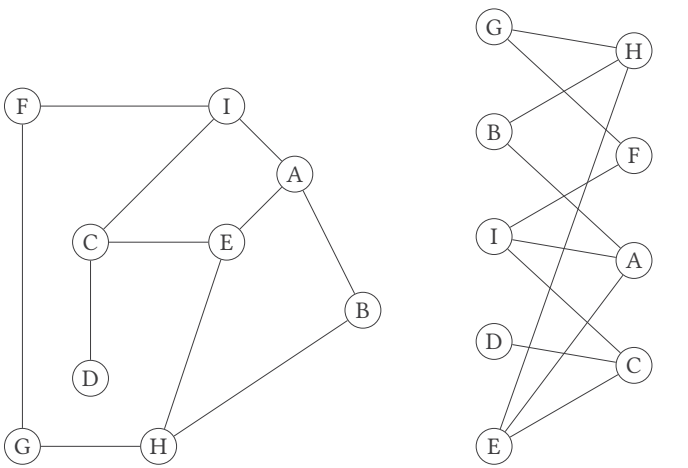
люди и поручения, которые надо распределить между ними). Элементы изображаются узлами, а совместная связность между ними является ребрами. Чтобы избежать неприятностей, нужно точно соотнести один элемент с другим. Это не всегда неочевидно, пока мы не переставим узлы и не превратим граф в двудольный, как показано на рисунке 2.10.



(a) Ациклический неориентированный граф

(b) Ориентированный ациклический граф

Рисунок 2.9. Ациклические графы



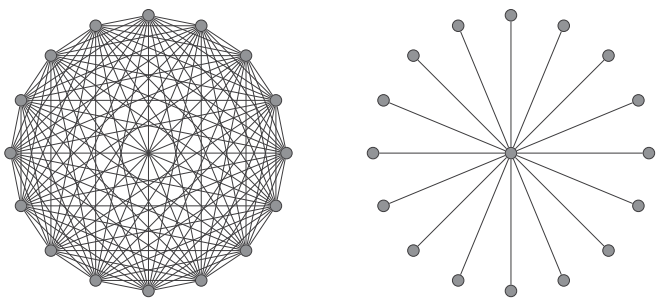
(a) Граф

(b) Тот же самый граф

Рисунок 2.10. Двудольный граф

Графы с большим и малым количеством ребер существенно отличаются друг от друга. Графы с большим числом ребер называются насыщенными графами, а с малым — разреженным. Бы-

вают такие графы, в которых все узлы напрямую соединены друг с другом. Подобные графы называются полными, такой граф изображен на рисунке 2.11: тут совершенно ясно видно огромное количество ребер. Для n узлов (так как каждый узел соединен с со всеми остальными n – 1 узлами) у нас есть $n(n-1)/2$ ребер. В целом мы можем сказать, что граф с n-м количеством узлов насыщенный, если у него примерно n^2 ребер, и что он разреженный, если у него примерно n ребер. Область между n и n^2 остается размытой, и чаще всего ее проясняет заданный областью применения контекст, поясняя имеем ли мы дело с насыщенным или разреженным графом; в большинстве приложений используются разреженные графы. Например, представьте, что у нас граф, отображающий дружбу между людьми. Граф включает в себя семь миллиардов узлов, то есть $n = 7 \times 10^9$, полагая, что у всех людей на Земле есть друзья. Мы так же допускаем, что у каждого человека 1000 друзей, что на самом деле сложно представить. Таким образом, число ребер 7×10^{12}, или же семь биллионов. Количество $n(n-1)/2$ для $n = 7 \times 10^9$ равно почти что $2{,}5 \times 10^{25}$, или же семи септильонам, что намного больше семи биллионов. У графа может быть несметное количество ребер, однако он все равно будет оставаться разреженным.



(a) Насыщенный граф (b) Разреженный граф

Рисунок 2.11. Насыщенный и разреженный графы

2.2. Представление графов

Прежде чем приступить к работе с графами на компьютере, нам нужно посмотреть, как графы реализуются в компьютерных про-

граммах. Но сначала устроим небольшую экскурсию и узнаем, как определяются графы в математике. Как правило, все вершины мы обозначаем как V, а все ребра — E. В таком случае граф G представляет собой $G = (V, E)$. В неориентированных графах множество E — это множество, состоящее из двух наборов элементов $\{x, y\}$ для каждого ребра между двумя узлами графа: между x и y. Чаще всего мы пишем (x, y) вместо $\{x, y\}$. В том и другом случае расположение x и y не имеет значения. Таким образом, граф на рисунке 2.5(a) отображается так:

$V = \{0, 1, 2, 3\}$.

$E = \{\{0, 1\}, \{0, 2\}, \{0,3\}, \{1,2\}, \{1,3\}, \{2,3\}\}$

$= \{(0, 1), (0, 2) (0, 3), (1, 2), (1, 3), (2, 3)\}$

В ориентированных графах множество E — это множество, состоящее из двух кортежей элементов (x, y) для каждого ребра между двумя узлами графа, x и y. Теперь же расположение x и y важно, оно отвечает за порядок изображенных ребер. Таким образом, граф на рисунке 2.5(b) отображается как:

$V = \{0, 1, 2, 3\}$

$E = \{(0, 1), (0, 2), (0, 3), (1, 0), (1, 2), (1, 3), (2, 0), (3, 2)\}$

Таблица 2.1. Матрица смежности для графа, изображенного на рисунке 2.3

	0	1	2	3	4	5
0	0	1	0	1	0	0
1	1	0	1	0	0	1
2	0	1	0	1	1	0
3	1	0	1	0	1	0
4	0	0	1	1	0	1
5	0	1	0	0	1	0

Математическое определение графа показывает, что для того, чтобы представить граф, нам надо каким-то образом отобразить вершины и ребра. Самый простой способ представить G — с помощью матрицы. Такая матрица, названная матрицей смежности, представляет собой квадратную матрицу, в которой для каждой вершины отведен ряд и столбец. В матрице содержатся 0 и 1. Если вершина, представленная рядом i, соединена с вершиной, связ-

ной рядом *j*, тогда матричный элемент (*i*, *j*) будет 1, в противном случае он будет 0. В матрице смежности вершины представлены индексами столбцов и рядов, а вершины представлены содержимым матрицы.

Согласно приведенным правилам, матрица смежности для графа с рисунка 2.2 приведена в таблице 2.1. Вы можете заметить, что матрица смежности симметрична. К тому же по диагонали будут идти нули, кроме случая, когда в графе есть петли. Если мы назовем матрицу *A*, тогда для двух узлов *i* и *j* $A_{ij} = A_{ji}$. Это верно для всех неориентированных графов, однако неверно для всех орграфов (за исключением случаев, когда для каждого ребра от узла *i* до узла *j* существует ребро из узла *j* к узлу *i*). Также видно, что множество значений в матрице равны нулю, что типично для разреженных графов.

Даже если граф не разреженный, нам нужно осторожно обходиться с пространством для всех этих нулевых элементов в матрице смежности. Чтобы избежать подобного положения дел, мы можем прибегнуть к альтернативному представлению графов, которое требует куда меньше места. Так как графы, существующие в нашем мире, могут иметь миллионы ребер, чаще всего мы пользуемся альтернативным представлением, чтобы сохранить все возможные детали. При таком подходе для представления вершин графа мы используем массив. Каждый элемент массива обозначает одну из вершин и запускает кортеж, состоящий из вершин, соседствующих с выбранной вершиной. Такой кортеж называется списком смежных вершин графа.

Что же из себя представляет список? Это структура данных, в которой содержатся элементы. Каждый находящийся в нем элемент — узел — состоит из двух частей. Первая часть содержит данные, описывающие элемент, а вторая часть содержит связку со следующим элементом списка. Вторая часть часто является указателем, который указывает на следующий элемент; указатель в компьютерах — это нечто, указывающее на определенное местоположение в компьютерной памяти, их еще называют сносками, так как они направляют к тому самому местоположению. Итак, вторая часть списка элементов часто является указателем,

содержащим адрес места, в котором находится следующий узел. У списка есть самый первый элемент, голова. Мы поочередно перебираем элементы списка, как если бы перебирали звенья цепи. Когда доходим до такого элемента, после которого нет других элементов, мы говорим, что он указывает в никуда или же на ноль; в компьютерной сфере мы используем термин «ноль» для обозначения пустоты; так как ноль — особое значение, мы будем отмечать его в тексте и псевдокоде как `null`. Список, построенный подобным образом, называется связным списком. Такой список изображен на рисунке 2.12. Мы использовали перечеркнутый квадрат для обозначения `null`.

Рисунок 2.12. Связной список

Базовые операции, которые нам нужны для работы со списками, таковы:

- `CreateList`() создает и возвращает новый, пустой список.
- `InsertListNode`(*L*, *p*, *n*) добавляет в список *L* узел *n* после узла *p*. Если *p* — null, тогда мы рассматриваем *n* как новую голову списка. Функция возвращает указатель к *n*. Мы полагаем, что узел *n* уже создан с некоторыми данными, которые нам надо добавить в список. Мы не будем вдаваться в подробности, каким образом создаются узлы. Вкратце, часть памяти должна быть выделена и инициализирована, таким образом, чтобы в узле содержались нужные нам данные и указатель. `InsertListNode` нужна, чтобы изменить указатели, с ее помощью указатель *n* станет указывать на следующий узел или же на `null`, если *p* — последний узел списка. Функция также должна изменить указатель *p*, чтобы он указывал на *n*, если *p* не null.
- `InsertInList`(*L*, *p*, *d*) добавляет в список *L* узел, содержащий данные *d*, после узла *p*. Если *p* — `null`, тогда нам нужен новый узел, который станет новой головой списка. Функция вернет указатель к новому интересующему нас узлу. Отличие от `InsertListNode` в том, что

`InsertInList` создает узел, в котором будут *d*, в то время как `InsertListNode` берет уже созданный узел и вставляет его в список. `InsertListNode` вставляет узлы, в то время как `InsertInList` вставляет данные, содержащиеся в созданных узлах. Это означает, что `InsertInList` может использовать `InsertListNode`, чтобы вставить в список созданный узел.

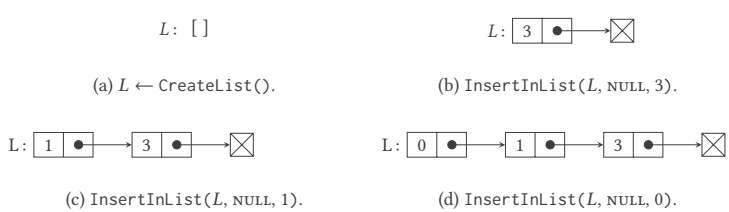
- `RemoveListNode`(*L*, *p*, *r*) убирает узел *r* из списка и выдает этот узел; *p* указывает на узел, предшествующий *r*, или на `null`, если *r* является головой. Нам нужно знать *p*, чтобы успешно извлечь объект, на который указывает *r*. Если *r* нет в списке, тогда мы получим `null`.
- `RemoveFromList`(*L*, *d*) убирает из списка первый узел, содержащий *d*, и выдает этот узел. Отличие от `RemoveListNode` в том, что данная функция будет искать в списке узел, содержащий *d*, найдет его и уберет; *d* указывает не на сам узел, а на данные, содержащиеся в узле. Если ни один узел в списке не содержит *d*, тогда `RemoveFromList` выдает `null`.
- `GetNextListNode`(*L*, *p*) выдает узел, следующий за *p* в списке *L*. Если *p* в конце списка, тогда она выдает `null`. Если *p* — это `null`, то функция выдаст первый узел в *L*, то есть голову. Полученный узел из списка не убирается.
- `SearchInList`(*L*, *d*) просматривает список в поисках первого узла, содержащего *d*. Выдает нужный узел или же `null`, если нужного узла не существует; узел из списка не убирается.

Чтобы создать представление графа с помощью списка смежных вершин, необходимы только `CreateList` и `InsertInList`. Чтобы просмотреть элементы списка *L*, нам нужно вызвать *n* ← `GetNextListNode`(*L*, `null`) и получить первый элемент; затем, пока n ≠ `null`, мы можем повторно назначить *n* ← `GetNextListNode`(*L*, *n*). Обратите внимание, что нам нужно получить доступ к данным внутри узла, например с помощью функции `GetData`(*n*), которая возвращает данные *d*, хранящиеся в узле *n*.

Чтобы увидеть, как работает добавление, представьте пустой список, в который мы добавляем три узла, каждый из которых содержит число. Когда мы напишем список в виде текста, мы сможем пронумеровать его элементы в квадратных скобках, вот так: [3, 1, 4, 1, 5, 9]. Пустой же список будет выглядеть так [ ]. Если есть числа 3, 1 и 0 и мы в таком порядке добавляем их в начало списка, то список увеличится от [ ] до [0, 1, 3], как показано на рисунке 2.13. Добавление в начало списка работает благодаря периодическому прохождению `null` в качестве второго аргумента в `InsertInList`.

Как вариант, если нам надо прикрепить ряд узлов к концу списка, мы производим ряд вызовов в `InsertInList`, передавая в качестве второго аргумента возвращаемое значение предыдущих вызовов; вы можете видеть эту последовательность на рисунке 2.14.

Далее идет извлечение узла из списка: один узел извлекается, после чего предыдущий (если он есть) начинает указывать на узел, следовавший за изъятым. Если мы убираем голову списка, то нет никакого предыдущего узла и следующий узел становится главой списка. На рисунке 2.15 показано, что происходит, когда мы извлекаем узлы с данными 3, 0, 1 из списка, который мы создали на рисунке 2.13. Если предыдущий узел нам неизвестен, нам придется начать с головы списка и просмотреть все последующие узлы, пока мы не найдем тот, что указывает на узел, который мы хотим изъять. Поэтому мы включаем это в вызовы `RemoveListNode`.



(a) $L \leftarrow$ `CreateList()`. (b) `InsertInList(`L, NULL, 3`)`.

(c) `InsertInList(`L, NULL, 1`)`. (d) `InsertInList(`L, NULL, 0`)`.

Рисунок 2.13. Добавление узлов в начало списка

Хотя в наших примерах узлы из списка содержат только числа, на самом деле список — куда более масштабная структура данных и может содержать совершенно разную информацию.

Мы сейчас рассматриваем самые простые примеры, однако следует помнить, что можно составить списки чего угодно, главное, чтобы они состояли из элементов, содержащих данные, и чтобы каждый элемент указывал на последующий элемент списка или же `null`.

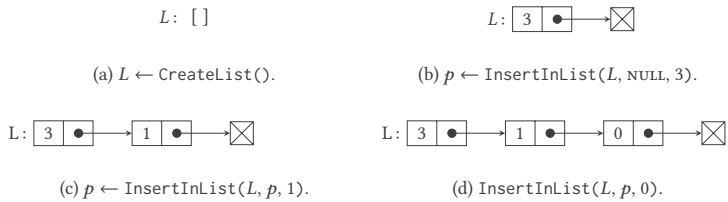


Рисунок 2.14. Прикрепление узлов к списку

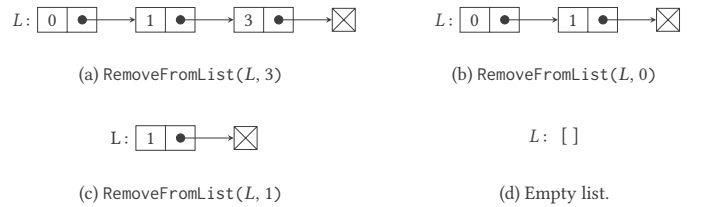


Рисунок 2.15. Извлечение узлов из списка

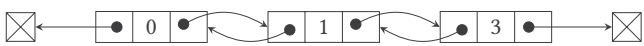


Рисунок 2.16. Двусвязный список

Рассмотренные нами списки довольно примитивны, всего лишь с одной связкой от одного элемента к другому. Существуют и другие виды списков. У нашего элемента могут быть две связки: одна указывает на предыдущий элемент списка, а другая — на следующий; такой список называется двусвязным. И напротив, для большей конкретики мы применяем термин «односвязный список», чтобы дать понять, что речь идет о списках с одной-единственной связкой. Пример двусвязного списка приведен на рисунке 2.16. В двусвязном списке и первый, и последний элементы указывают на `null`. Двусвязные списки позволяют нам проходится по ним туда и обратно, в то время как односвязные списки

позволяют нам проходить только в одном направлении, вперед. В то же время двусвязным требуется куда больше места, так как каждому узлу списка нужны две связки: одна к следующему элементу, а другая к предыдущему. Более того, добавление и извлечение происходят чуть сложнее, потому что нам нужно убедиться, что обе связки (вперед и назад) актуальны и назначены как надо. С другой стороны, чтобы извлечь узел, нам больше не надо знать указывающий на него узел, так как мы тут же можем найти его с помощью обратной связки.

Еще одни вид списка — список, в котором последний элемент не указывает на `null`, а возвращает к первому элементу списка. Такой список называется циклическим, он изображен на рисунке 2.17.

Все это полезно знать, и множество разных списков широко применяются в разных областях, однако здесь и сейчас мы уделим внимание именно односвязным спискам.

С помощью списков мы создаем представление графа в виде списка смежности. В данном представлении у нас есть один список смежности для каждой вершины. Все они объединены в массив, который указывает в начало списков смежных вершин. Если есть массив A, то объект $A[i]$ данного массива указывает в начало списка смежности узла i. Если рядом с узлом i нет других узлов, $A[i]$ будет указывать на `null`.

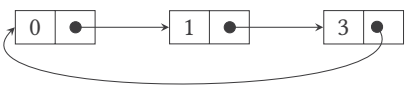


Рисунок 2.17. Циклический список

Представление графа для списка смежности рисунка 2.4 показано на рисунке 2.18. В правом верхнем углу мы для удобства разместили упрощенную версию графа. Слева на рисунке 2.18 размещен массив, содержащий головы списка смежности графа; каждому объекту массива соответствует определенная вершина. Список смежности содержит ребра вершины, стоящей во главе. Таким образом, третий элемент массива содержит голову списка смежности для узла 2. Каждый список смежности составлен из соседних узлов, расположенных в числовом порядке.

Например, чтобы создать список смежности для узла 1, мы вызываем `Insert` три раза: для узлов 0, 2 и 5, именно в таком порядке. Это объясняет, почему узлы появляются в обратном порядке что здесь, что в любом другом списке смежности, изображенном на рисунке: узлы добавляются в самое начало каждого списка, так что, добавив их в порядке 0, 2, 5, мы в итоге получим список [5, 2, 0].

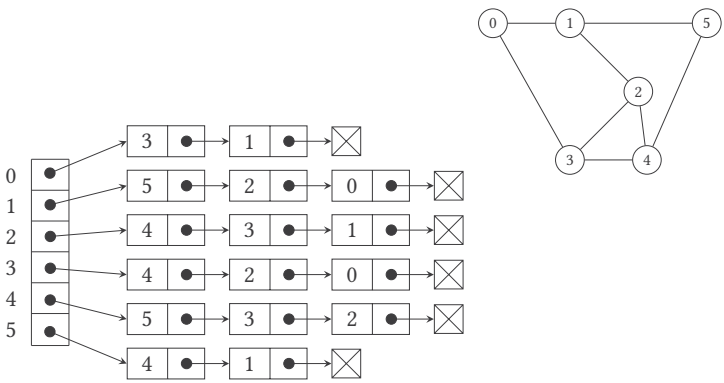


Рисунок 2.18. Изображение графа для списка смежных вершин

Теперь сравним требуемое место для матрицы смежности и для представления списка смежности, составленных для графа $G = (V, E)$. Если $|V|$ — число вершин в графе, то матрица смежности будет содержать $|V|^2$ элементов. Точно так же, если $|E|$ — число ребер в графе, то список смежности будет содержать массив размером $|V|$ и тогда $|V|$ определяет, что в них всех будет содержаться $|E|$ ребер. Таким образом, списку смежности надо будет $|V| + |E|$ элементов, что намного меньше, чем $|V|^2$, если только граф не насыщенный и не обладает множеством вершин, соединенных друг с другом.

Вам может показаться, что в таком случае от матриц смежности нет никакого проку. Но прок есть. Во-первых, матрицы смежности проще. Вам надо знать только матрицы и больше ничего другого, никаких хлопот со списками. Во-вторых, матрицы смежности быстрее. Мы считаем само собой разумеющимся, что доступ к элементу в матрице — это операция, которой требуется

постоянное время, то есть мы достаем любое ребро или элемент одинаково быстро, вне зависимости от их расположения: что в верхнем левом углу матрицы, что в нижнем правом. Когда мы пользуемся списками смежности, нам нужно получить доступ к правому элементу массива с вершинами, который расположен с левой стороны рисунка 2.18, плюс отыскать требуемое ребро путем просмотра всего списка, в главе которого стоит конкретная вершина. Таким образом, чтобы увидеть, соединены ли узлы 4 и 5, нам надо сперва найти в матрице вершин узел 4, а затем перескочить к 2, 3 и, наконец, к 5. Чтобы узнать, соединены ли узлы 4 и 0, нам надо пройтись по всему списку, озаглавленному 4, до самого конца, а затем сообщить, что мы не нашли 0 и значит, у узлов нет общей связки. Вы можете возразить, что гораздо быстрее просмотреть список, озаглавленный 0, так как он короче, но нам об этом никак не узнать.

Использование нотации большого O, которая определяет соединена ли одна вершина с другой, занимает постоянное время, поэтому, если мы прибегаем к матрице смежности, сложность составляет $\Theta(1)$. Та же самая операция, использующая списки смежности, занимает $O(|V|)$ времени, так как есть возможность, что вершина в графе соединена со всеми другими вершинами, и нам придется просмотреть весь список смежности в поисках соседней. Вы же знаете, что не бывает бесплатного сыра в мышеловке. В компьютерной сфере то же самое, и это называется компромиссом: мы жертвуем местом ради скорости. Нам часто приходится идти на такую сделку, и у нее даже есть собственное название: пространственно-временной компромисс.

2.3. Обход графа в глубину

Итак, вернемся к исследованию лабиринтов. Чтобы полностью обойти лабиринт, нам нужны две вещи: способ отследить пройденные нами места и некий систематический подход к посещению комнат, в которых мы еще не бывали. Представим, что комнаты расположены в каком-то порядке. Судя по графу, который

мы видели, комнаты пронумерованы, поэтому, чтобы пройтись по всем комнатам, мы заходим в первую комнату и отмечаем, что мы в ней побывали. Затем мы идем в первую комнату, соединенную с той, в которой мы только что побывали. Отмечаем, что мы здесь были. Снова идем в первую непосещенную комнату, которая соединена с той, в которой мы находимся. Далее процедура повторяется: мы делаем в комнате отметку, что были здесь, а потом направляемся к первой непосещенной комнате, соединенной с той, в которой мы находимся. Если не осталось непосещенных комнат, соединенных с нынешней, мы возвращаемся туда, откуда пришли, и смотрим, остались ли комнаты, в которых мы еще не побывали. Если да, мы заходим в первую непосещенную комнату, и продолжаем все ту же процедуру. Если нет, мы возвращаемся еще на одну комнату назад. Так мы идем до тех пор, пока не вернемся в начальную комнату и не обнаружим, что посетили все соединенные с ней комнаты.

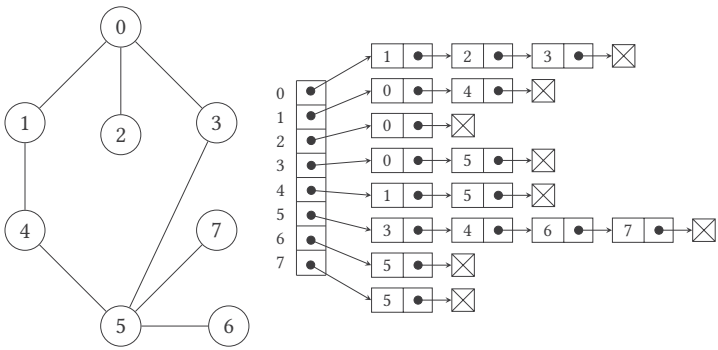


Рисунок 2.19. Исследуемый в глубину лабиринт

На практике все выглядит намного понятней. Данная процедура называется поиском в глубину, так как мы исследуем лабиринт скорее вглубь, нежели вширь. На рисунке 2.19 изображен примерный лабиринт, а справа от него — его отображение в виде списка смежности. Еще раз обратите внимание, что узлы в списке расположены в обратном порядке. Например, в списке смежности для узла 0 нас интересуют соседние узлы 3, 2, 1, поэтому в списке они будут представлены как [0, 1, 2, 3].

Следуя поиску в глубину на рисунке 2.20, мы начинаем путь в узле, или комнате, 0. Серым цветом мы закрашиваем узел, в котором находимся, а черным — узлы, в которых уже побывали. Двойные линии изображают виртуальную нить, которую мы держим в руках, пока идем по лабиринту. Подобно Тесею, мы используем нить, чтобы вернуться обратно и изменить направление движения, когда упираемся в тупик.

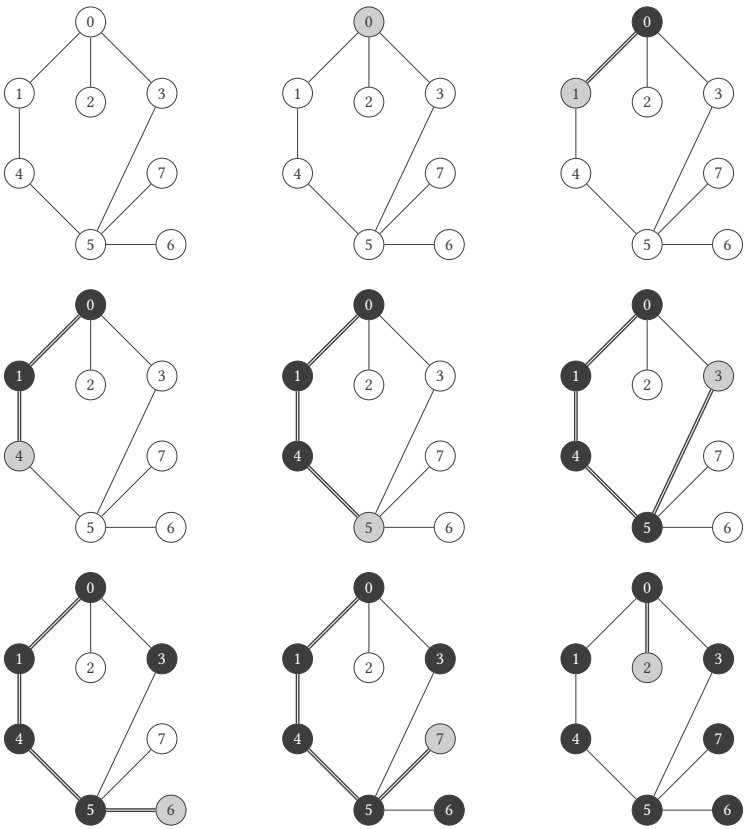


Рисунок 2.20. Исследование лабиринта в глубину

Первая непосещенная комната у нас — комната 1, и мы заходим в нее. Первая непосещенная комната из комнаты 1 — комната 4. Далее первая непосещенная комната в комнате 4 — комната 5. Тем же путем мы попадаем из 5 в 3. Здесь мы находим, что не-

посещенных комнат больше нет, и используем нашу нить, чтобы вернуться назад, в комнату 5. Тут мы обнаруживаем непосещенную комнату 6, мы заходим в нее и снова возвращаемся в 5. В комнате 6 есть одна непосещенная комната, 7. Мы заходим в 7, а затем возвращаемся в 5. Теперь у комнаты 5 не осталось непосещенных комнат, поэтому мы возвращаемся в комнату 4. У комнаты 4 нет непосещенных комнат, так что мы снова делаем шаг назад, в комнату 1. Там нам снова приходится отступать, и мы возвращаемся в 0. Там мы видим, что осталась непосещенной комната 2; мы заходим в нее и возвращаемся в 0. Теперь, когда мы побывали в 1, 2 и 3, наше исследование закончено. Если вы просмотрите проделанный нами путь, то отметите, что мы шли вглубь, а не вширь. Из комнаты 0 мы направились в 1, а затем в 4 вместо того, чтобы пойти в 2 и 3. Несмотря на то, что комната 2 находилась рядом с нашим стартом, мы посетили ее в последнюю очередь. Так что мы сперва копаем вглубь до упора, прежде чем рассматривать альтернативы.

Алгоритм 2.1 реализует поиск в глубину. Алгоритм берет в качестве вводных данных граф G и узел, с которого он начнет исследование графа. Он также использует массив *visited* («посещенные»), который учитывает каждый посещенный узел.

Алгоритм 2.1. Поиск в глубину по рекурсивному графу

```
DFS(G, node)
    Вводные данные: G = (V, E), граф; node, узел в G.
    Данные: visited, массив размером |V|.
    Результат: visited[i] будет true, если мы уже посетили узел i,
    а в противном случае — false
1   visited[node] ← TRUE
2   foreach v in AdjacencyList(G, node) do
3       if not visited[v] then
4           DFS(G, v)
```

В самом начале пути мы не посетили ни одного узла, поэтому *visited* будет `false`. Хотя *visited* и нужен нашему алгоритму, мы не включаем его в вводные данные, так как, когда мы вызываем его, мы не передаем его алгоритму; он представляет собой мас-

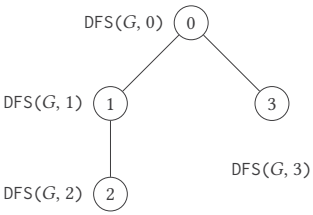
сив, созданный и инициализированный вне алгоритма, алгоритм может получить к нему доступ, прочесть его и преобразовать. Так как *visited* преобразуется с помощью алгоритма, он по сути является выводимыми данными алгоритма, даже если мы его так не называем. Мы специально не описываем выводимые данные для алгоритма, потому что он ничего не выводит; но все же его работа с помощью массива *visited* оказывает влияние на окружение, поэтому мы можем сказать, что изменения в *visited* — это и есть результаты работы алгоритма. *Visited* можно считать эдакой новой чистой доской, на которой алгоритм записывает свой процесс выполнения.

Алгоритм поиска в глубину — `DFS`(*G*, *node*) — рекурсивный. Рекурсивные алгоритмы — это алгоритмы, которые вызывают сами себя. В строчке 1 `DFS`(*G*, *node*) отмечает нынешнюю вершину как посещенную и затем вызывает сама себя для каждой непосещенной вершины, соединенной с ней, путем прохождения вниз по списку смежности; мы полагаем, что у нас есть функция `AdjacencyList`(*G*, *node*), которой задали граф и узел и которая возвращает список смежности. В строчке 2 мы проходимся по узлам списка смежности; это легко сделать, потому что от каждого узла мы можем перейти прямо к следующему узлу, благодаря определению списка, так как каждый узел в списке связан с последующим. Если мы не посетили тот соседний узел (строчка 3), тогда мы вызываем `DFS`(*G*, *v*) для соседнего узла *v*.

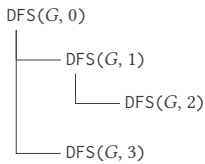
Самая заковыристая часть в понимании алгоритма — понимание рекурсии, которую он выполняет. Давайте возьмем простенький граф, состоящий из четырех узлов, изображенных на рисунке 2.21(а). Мы начинам в узле 0. Если граф назван *G*, то мы вызываем `DFS`(*G*, *0*). Функция получает список смежности для узла 0, который представляет собой [1, 3]. Цикл в строчек 2–4 выполняется дважды: сначала для узла 1, а потом для узла 3. При первом проходе цикла, когда узел 1 еще не посещен, мы доходим до строчки 4.

Теперь идет самая важная часть. В строчке 4 мы вызываем `DFS`(*G*, 1), но текущий вызов пока еще не завершен. Мы начинаем выполнение `DFS`(*G*, 1) и в то же время будто бы откладыва-

ем на полку DFS(*G*, 0), мы заберем ее с полки, когда завершится DFS(*G*, 1), и продолжим ее выполнение сразу после вызова DFS(*G*, 1). Когда мы работаем с DFS(*G*, 1), мы берем ее список смежности, [2]. Цикл в строчках 2–4 будет выполнен единожды, для узла 2. Так как узел 2 еще не посещен, мы доходим до строчки 4. Как и прежде, мы вызываем DFS(*G*, 2), но DFS(*G*, 1) еще не завершена и отложена на полку в ожидании завершения DFS(*G*, 2). DFS(*G*, 2) имеет пустой список смежности, поэтому цикл в строчках 2–4 вовсе не выполняется и функция тут же возвращается.



(a) Граф и порядок посещения



(b) Трассировка вызова при поиске в глубину

Рисунок 2.21. Углубленный поиск в глубину

Куда? Мы забираем DFS(*G*, 1) с полки и направляемся в место в DFS(*G*, 1), где она остановилась, то есть в строчку 4, сразу после вызова DFS(*G*, 2). Так как в списке смежности узла 1 нет других узлов кроме узла 2, цикл в DFS(*G*, 1) прекращается и функция завершается. Мы возвращаемся к DFS(*G*, 0), забираем ее с полки и отправляемся к месту, где мы остановились: в строчку 4, тут же после вызова DFS(*G*, 1). Здесь выполнение первой итерации цикла заканчивается, и мы запускаем вторую итерацию цикла. Мы взываем DFS(*G*, 3), а DFS(*G*, 0) снова откладываем на полку. Узел 3 имеет пустой список смежности и, как и в узле 2, DFS(*G*, 3) завершится и вернется к DFS(*G*, 0). Это позволит DFS(*G*, 0) закончить вторую итерацию цикла, полностью завершить процесс выполнения и вернуться.

Отслеживание ряда вызовов функции мы называем, как ни тривиально, трассировкой вызова. На рисунке 2.21(b) изображена трассировка вызова, которая начинается с DFS(*G*, 0). Трасси-

ровка — это считывание данных сверху донизу, слева направо. Она похожа на дерево, в котором вызовы, порожденные другим вызовом, являются ответвлениями от этого самого вызова, его детьми.

Алгоритм 2.2. Факториальная функция

`Factorial(`n`)` $\rightarrow !n$

Вводные данные: n, натуральное число.

Выводимые данные: $n!$, факториал n

```
1  if n = 0 then
2      return 1
3  else
4      return n · Factorial(n − 1)
```

Контроль переходит от родителя к детям, от детей ко внукам и так далее, пока не останется потомков, после чего начинается обратный путь, от потомков к предкам. Получается так: вниз `DFS`(G, 0) → `DFS`(G, 1) → `DFS`(G, 2); вверх `DFS`(G, 2) → `DFS`(G, 1) → `DFS`(G, 0); снова вниз `DFS`(G, 0) → `DFS`(G, 3); снова вверх `DFS`(G, 3) → `DFS`(G, 0); завершение. Заметьте, когда мы посещаем узел, мы рекурсивно идем к его детям, если таковые имеются; мы посещаем его братьев только после того, как побывали у его детей. Например, мы проделали путь от узла 1 к узлу 2, но не к узлу 3. Используя рекурсию подобным образом, мы все равно идем вглубь, выполняя обход в глубину.

Если вы не совсем уверены, что поняли рекурсию, то мы сейчас сделаем небольшое, но очень полезное отступление, чтобы подробней осветить тему. Первоначальная рекурсивная функция — это факториал, $n! = n \times (n - 1) \times \ldots \times 1$ с тупиковой ситуацией $0! = 1$. Мы можем отобразить факториальную функцию в виде алгоритма 2.2; на рисунке 2.22 изображена трассировка вызова для вычисления 5!.

В данной трассировке вызова мы включаем в каждый вызов описание, которое создает рекурсивный вызов; таким образом мы можем показать с помощью стрелок куда именно возвращается вызываемая функция. Например, когда мы находим-

ся в `Factorial(1)`, мы достигаем строчки 4 алгоритма 2.2. Тут вы можете представить `Factorial(0)` на рисунке 2.22 как поле для заполнения, которое заполняется, когда `Factorial(0)` возвращается. Обратный путь в трассировке вызова, заполняющей поля, обозначен идущими снизу-вверх стрелками, которые поднимают рекурсивную серию вызовов. Мы спускаемся вниз по трассировке вызова, следуя по расположенным слева ступенькам, а с помощью стрелок справа возвращаемся наверх.

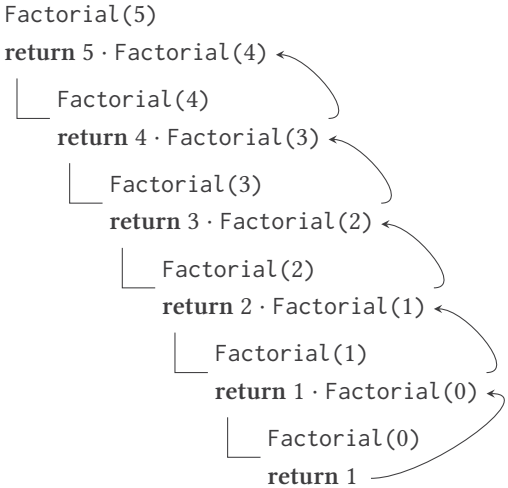


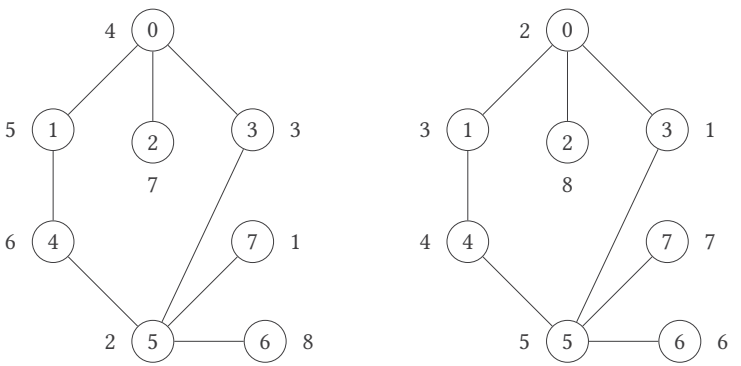
Рисунок 2.2. Факториальная трассировка вызова

Ключевую роль во всех рекурсивных функциях играет условие, которое прерывает рекурсию. В факториальной функции, когда мы достигаем $n = 0$, мы прекращаем рекурсивные вызовы и отправляемся назад. В алгоритме 2.1 условие прерывания прописано в строчке 3. Если мы посетили все соседние с выбранным узлы, тогда у нас больше не остается рекурсивных вызовов для выбранного узла, и время отправляться в обратный путь. *Это очень важный момент.* Если забудете указать, когда прерывается рекурсия, — потом проблем не оберетесь. Рекурсивная функция без условия прерывания будет бесконечно вызывать сама себя. Программисты, забывающие об этом условии, ищут себе неприятностей. Компьютер будет все вызывать и вызывать

функцию, пока в памяти не останется места, после чего программа выдаст ошибку. В конце концов вас ждет сообщение «переполнение стека» или что-то в этом роде. Почему? Объясним дальше.

Надеемся, что с рекурсией теперь все понятно (если нет, то перечитайте главу еще раз). Заметьте, что алгоритм поиска в глубину работает с любого узла. В нашем примере мы использовали узел 0 только потому, что он находится сверху. Мы точно так же можем начать алгоритм на узле 7 или 3, как изображено на рисунке 2.23, где рядом с узлами мы указали порядок их посещения. Каким бы ни был порядок посещения узлов, мы полностью исследовали граф.

У нас бы ничего не получилось, будь граф неориентированным и несвязным или же ориентированным и не сильно связным. В таких графах нам надо вызывать алгоритм 2.1 для каждого узла, который может оказаться непосещенным. При таком раскладе даже узлы, до которых не достать с помощью одного узла, могут оказаться стартовыми.



(a) Поиск в глубину с узла 7 (b) Поиск в глубину с узла 3

Рисунок 2.23. Исследование в глубину с различных узлов

Как рекурсия работает на компьютере? Как реализуется вся эта работа с функциями, отложенными на полку, с вызовами других функций и возвращениями? Компьютер знает, куда вернуться из функции, потому что использует внутрен-

ний стек — стек вызовов. Наверху данного стека находится нынешняя функция. Внизу — функция, которая вызвала ее, со всей информацией для продолжения работы с того места, где мы остановились. Еще ниже находится функция, которая вызвала эту функцию, и так далее. Именно поэтому, если рекурсия пойдет неправильно, случится программная ошибка: у нас не бесконечный стек. На рисунке 2.24 вы можете видеть состояние стека при работе алгоритма 2.1. Содержимое стека, когда алгоритм посещает узел, размещено внизу и закрашено серым. Когда он упрется в тупик, то есть наткнется на узел, не имеющий непосещенных соседних узлов, функция вернется к инициатору вызова; процесс смены пути называется поиском с возвратом. Извлечение верхнего вызова в стеке мы называем раскруткой (хотя в случае исследования лабиринта мы, наоборот, сматываем нашу нить в клубок). Таким образом, от узла 3 мы направляемся к узлу 5, закрашивая черным уже посещенные места. На рисунке во втором ряду видна раскрутка, когда мы проделывали весь путь от узла 7 до узла 0, чтобы посетить узел 2.

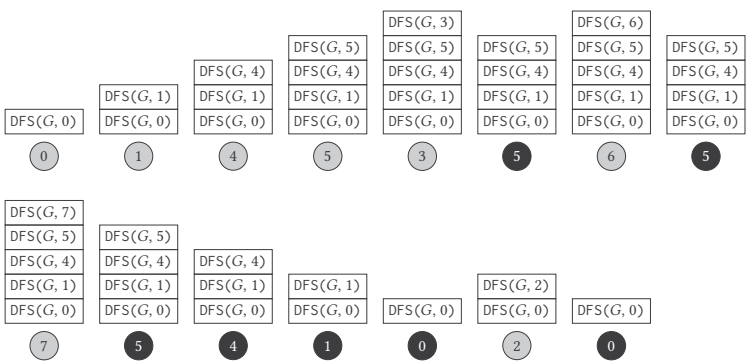


Рисунок 2.24. Поиск в глубину по стеку для рисунка 2.20

Все движение в стеке происходит автоматически. Впрочем, вы и сами можете делать подобные чудеса. Вместо рекурсии, нужной для поиска в глубину, который использует стек неявно, вы можете использовать стек явно, в этом случае вы полностью избавитесь

от рекурсии. Суть в том, что к каждому узлу мы добавляем непосещенные узлы в стеке и, когда мы ищем узлы, которые надо посетить, мы попросту извлекаем элементы из стека. Алгоритм 2.3 демонстрирует стековую реализацию поиска в глубину. Вы можете посмотреть содержимое стека на рисунке 2.25(b); в целях экономии места в книге мы изобразили только стек после прохождения узлов, без возврата.

Данный алгоритм работает точно так же, как алгоритм 2.1, однако использует явно указанный стек вместо того, чтобы полагаться на рекурсию. В строчке 1 мы создаем стек. Мы теперь не полагаемся на внешне заданный массив, который записывает наш прогресс, а собственноручно создаем массив *visited* в строчке 2; затем в строчках 3–4 мы инициализируем его для всех значений `false`.

Алгоритм 2.3. Поиск в глубину по графу с использованием стека

`StackDFS`(*G, node*) → *visited*

Вводные данные: граф $G = (V, E)$; *node*, стартовая вершина в G.
Выводимые данные: *visited*, массив размером $|V|$ такой, чтобы *visited*[*i*] был `true`, если мы уже посетили узел *i*, а иначе — `false`)

```
1  S ← CreateStack()
2  visited ← CreateArray(|V|)
3  for i ← 0 to |V| do
4      visited[i] ← FALSE
5  Push(S, node)
6  while not IsStackEmpty(S) do
7      c ← Pop(s)
8      visited[c] ← TRUE
9      foreach v in AdjacencyList(G, c) do
10         if not visited[v] then
11             Push(S, v)
12 return visited
```

Чтобы эмулировать рекурсию, мы кладем в стек узлы, которые нам надо посетить, и, когда мы ищем узел, чтобы посетить его, мы обращаемся к самому верхнему в стеке. Чтобы привести все в движение, в строчке 5 мы кладем в стек стартовый узел. За-

тем, поскольку у нас уже что-то есть в стеке (строчка 6), мы извлекаем его (строчка 7), отмечаем как посещенный (строчка 8) и помещаем в стек каждый узел в списке смежности, который мы еще не посетили (строчки 9–11). Когда мы закончим, то вернем массив *visited* и таким образом составим отчет, до каких узлов нам удалось дотянуться.

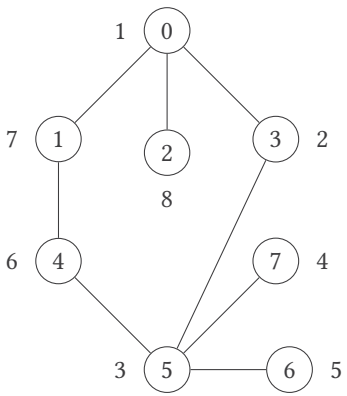
Из-за порядка узлов, сложенных в стек, обход графа идет в глубину, но идет от узлов с наибольшими значениями к узлам с наименьшими. Рекурсивный алгоритм в нашем лабиринте идет против часовой стрелки, в то время как данный алгоритм идет по часовой.

Посмотрите на рисунок 2.25(b), на третий столбец справа — узел 1 добавлен в стек дважды. Эту мелочь нельзя назвать ошибкой, но мы все же можем ее исправить. Нам нужен стек без дубликатов, в который добавляется лишь тот объект, которого еще нет в стеке. Для этого нам понадобится дополнительный массив. Элемент данного массива будет истинным, если такой элемент уже есть стеке, и будет ложным, если его нет. В результате мы получаем алгоритм 2.4. Он очень похож на алгоритм 2.3, однако использует дополнительный массив *instack*, в котором записаны уже имеющиеся в стеке узлы. На рисунке 2.25(c) показано, что происходит в стеке.

Возможно, вы задаетесь вопросом: зачем нам разрабатывать алгоритм 2.4, если у нас уже есть алгоритм 2.1? Помимо того, что он просто интересен, он еще и наглядно показывает нам работу рекурсии: неявная рекурсия требует, чтобы компьютер при каждом рекурсивном вызове клал все необходимое состояние памяти функции в стек. Таким образом он перепрaвит в стек больше данных, как на рисунке 2.24 (где показаны только вызовы функции), а не только однозначные числа, как на рисунке 2.25. Алгоритм, выполненный с помощью явного стека, скорее всего, окажется более экономным, чем рекурсивный алгоритм.

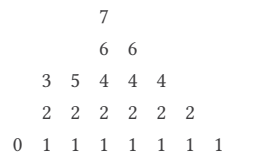
В завершение нашего исследования в глубину давайте вернемся к алгоритму 2.1 и посмотрим на его сложность. Сложность алгоритмов 2.3 и 2.4 будет такой же, как в алгоритме 2.1, потому

что в них меняется только реализация рекурсивного механизма, а сама стратегия исследования остается прежней. Строчка 2 выполняется $|V|$ раз, единожды для каждой вершины. После чего в строчке 4 `DFS`(*G*, *node*) вызывается единожды для каждого ребра, то есть $|E|$ раз. В конечном счете сложность поиска в глубину — $\Theta(|V| + |E|)$. Логично, что мы можем исследовать граф за время, пропорциональное его размеру.



(a) Порядок посещения комнат

(b) Модификация стека



(c) Модификация стека без дубликатов

Рисунок 2.25. Прохождение и содержимое стека алгоритмов 2.3 и 2.4

2.4. Поиск в ширину

Как мы уже видели, исследование графа в глубину идет вглубь, а не вширь. Теперь же давайте исследуем наш лабиринт иным способом: начнем с узла 0 и посетим узлы 1, 2 и 3 прежде, чем доберемся до узла 4. Это означает, что наша сеть будет расти не в глубину, а в ширину. Такая стратегия, как ни банально, называется поиском в ширину.

Во время поиска в ширину мы больше не полагаемся на спасительную нить (метафорическую или настоящую). Мы никак не можем попасть напрямую из узла 3 в узел 4, так как они напрямую никак не связаны, поэтому аналогия с настоящими лабиринтам, как лабиринт Миноса, здесь не работает. Чтобы осуществить обход в ширину, нам надо предположить, что из узла, в котором мы находимся в данный момент, мы можем попасть в узел, который существует где-то там и в котором мы еще не бывали. Окажись мы в настоящем лабиринте, мы бы никак не смогли испариться из узла 3 и внезапно оказаться в узле 4, однако такое возможно в алгоритме, если мы знаем, что узел 4 вообще существует. В данной версии исследования лабиринта мы можем перемещаться от одного узла к другому: известному, но еще непосещенному.

Алгоритм 2.4. Поиск в глубину по графу с использованием стека без дубликатов

`NoDuplicatesStackDFS(`G, $node$`)` $\rightarrow$ $visited$

Вводные данные: граф $G = (V, E)$; $node$, стартовая вершина в G.

Выводимые данные: $visited$, массив размером $|V|$ такой, чтобы $visited[i]$ был `true`, если мы уже посетили узел i, а иначе — `false`)

```
1  S ← CreateStack()
2  visited ← CreateArray(|V|)
3  instack ← CreateArray(|V|)
4  for i ← 0 to |V| do
5      visited[i] ← FALSE
6      instack[i] ← FALSE
7  Push(S, node)
8  instack[node] ← TRUE
9  while not IsStackEmpty(S) do
10     c ← Pop(S)
11     instack[c] ← FALSE
12     visited[c] ← TRUE
13     foreach v in AdjacencyList(G, c) do
14         if not visited[v] and not instack[v] then
15             Push(S, v)
16             instack[v] ← TRUE
17 return visited
```

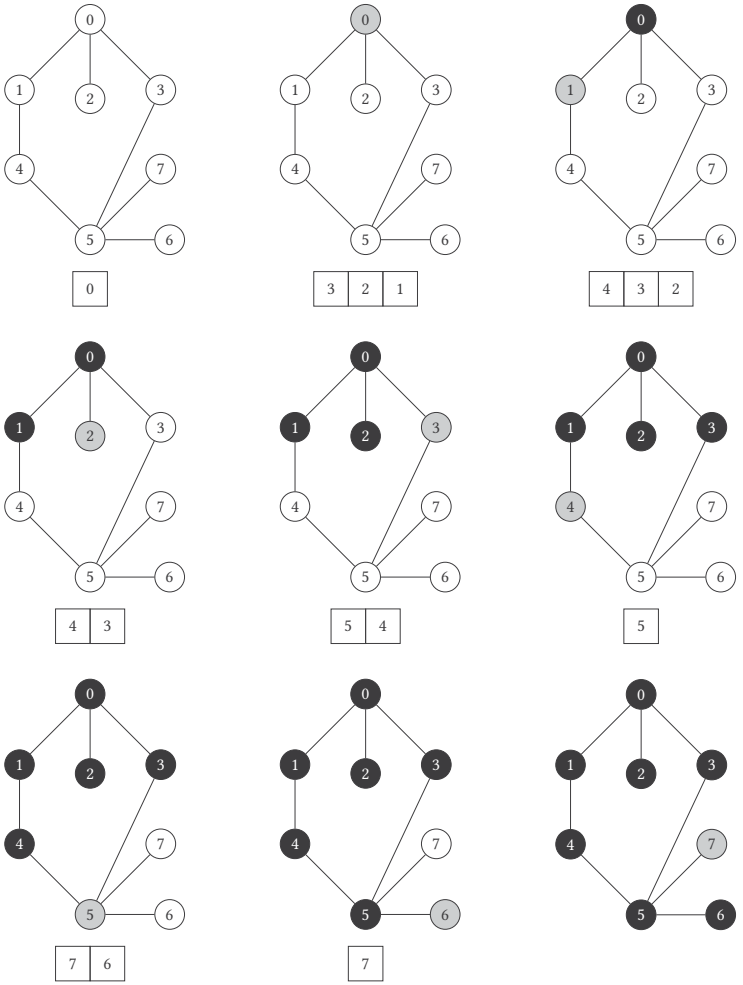


Рисунок 2.26. Исследование лабиринта в ширину

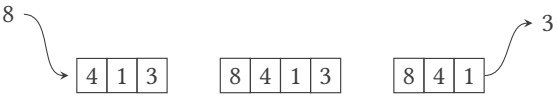


Рисунок 2.27. Добавление и извлечение из очереди

Вы можете увидеть пример поиска в ширину на рисунке 2.26. Под каждой картинкой с примером исследования указаны узлы,

которые, как мы знаем, надо посетить; просмотрите все картинки справа налево — и сразу поймете, зачем мы указали узлы. Когда мы начинаем поиск в ширину с узла 0, мы знаем лишь о существовании этого самого узла 0. В нем мы узнаем о трех соседних узлах: 1, 2 и 3.

В таком порядке мы их и посетим. Когда мы посещаем узел 1, мы узнаем о его соседнем узле, 4; теперь мы знаем, что нам надо побывать в узлах 2, 3 и 4. Мы посещаем узел 2, у которого нет соседних непосещенных узлов, и мы направляемся в узел 3, где мы узнаем о соседнем узле 5. Теперь мы знаем, что еще не побывали в узлах 4 и 5. Мы посещаем 4, а потом 5. Когда мы оказываемся в узле 5, мы узнаем, что нам еще надо посетить узел 6 и 7. В таком порядке мы их и посещаем, после чего исследование нашего лабиринта завершено.

Для выполнения поиска в ширину нам понадобится новая структура данных — очередь, которая снабжает нас полезными возможностями, нужными, чтобы отслеживать непосещенные узлы, указанные внизу каждой картинки исследования графа на рисунке 2.26. Очередь — это последовательность элементов. Мы добавляем элементы в конец последовательности и забираем их из ее начала; все как в реальной жизни: если ты первый в очереди, то и освободишься первым (если только кто-то не влезет вперед тебя). Если точнее, то речь у нас идет об очередях, построенных по принципу «первым пришел — первым ушел». Конец очереди мы называем хвостом, а начало — головой (и у списков, и у очередей есть головы). На рисунке 2.27 изображено, как в очереди работают добавление и извлечение. У очереди имеются следующие операции.

- `CreateQueue`() создает пустую очередь.
- `Enqueue`(*Q*, *i*) добавляет элемент *i* в хвост очереди *Q*.
- `Dequeue`(*Q*) убирает элемент из начала очереди. По сути, она убирает нынешнюю голову и делает головой следующий элемент очереди. Если очередь пуста, тогда операция не может быть выполнена (мы получим ошибку).
- `IsQueueEmpty`(*Q*) возвращает `true`, если очередь пуста, в противном случае возвращает `false`.

Имея под рукой такие операции, мы можем написать алгоритм 2.5, выполняющий по графу поиск в ширину. Так как очередь заполняется с хвоста и опустошается с головы, мы посещаем содержащиеся в ней узлы, прочитывая их справа налево, как изображено на рисунке 2.26.

Алгоритм 2.5. Поиск по графу в ширину

`BFS(`G, *node*`)` $\rightarrow$ *visited*

Вводные данные: граф $G = (V, E)$; *node*, стартовая вершина в G.

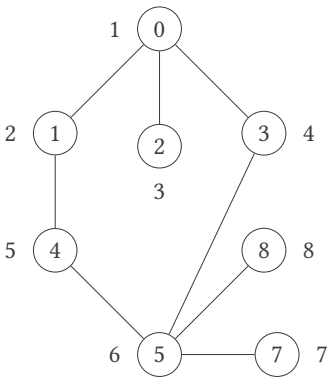
Выводимые данные: *visited*, массив размером $|V|$ такой, чтобы *visited*[i] был `true`, если мы уже посетили узел i, а иначе — `false`

```
1  Q ← CreateQueue()
2  visited ← CreateArray(|V|)
3  inqueue ← CreateArray(|V|)
4  for i ← 0 to |V| do
5      visited[i] ← FALSE
6      inqueue[i] ← FALSE
7  Enqueue(Q, node)
8  inqueue[node] ← TRUE
9  while not IsQueueEmpty(Q) do
10     c ← Dequeue(Q)
11     inqueue[c] ← FALSE
12     visited[c] ← TRUE
13     foreach v in AdjacencyList(G, c) do
14         if not visited[v] and not inqueue[v] then
15             Enqueue(Q, v)
16             inqueue[v] ← TRUE
17 return visited
```

Данный алгоритм похож на алгоритм 2.4. Он возвращает массив *visited*, который помечает узлы, до которых мы можем дотянуться.

Он использует массив *inqueue*, который отслеживает узлы, находящиеся в данный момент в очереди. В начале алгоритма, в строке 1, мы инициализируем очередь, которую будем потом использовать, затем в строчках 2–6, мы создаем и инициализируем *visited* и *inqueue*.

В очереди находятся узлы, о которых мы знаем, что они существуют, но мы них еще не побывали. В начале у нас есть только стартовый узел, поэтому мы добавляем его в очередь (строчка 7) и отслеживаем его в *inqueue*. Пока очередь не пустует (строчки 9–16), мы забираем из ее головы элементы (строчка 10), фиксируем событие (строчка 11) и помечаем элемент как посещенный (строчка 12). Затем каждый узел в списке смежности (строчки 13–16), непосещенный и не находящийся в очереди (строка 14), мы помещаем в очередь (строчка 15) и фиксируем, что узел добавлен в очередь (строчка 16). В результате узел покинет очередь при будущей итерации основного цикла алгоритма. Рисунок 2.28 представляет собой сжатую версию рисунка 2.26.



(a) Порядок посещения комнат

	0		
0:	3	2	1
1:	4	3	2
2:	4	3	
3:	5	4	
4:	5		
5:	7	6	
6:	7		
7:			

(b) Модификация очереди

Рисунок 2.28. Прохождение и содержимое очереди в алгоритме 2.5

Если мы обратим внимание на сложность алгоритма, увидим, что строка 9 будет выполнена $|V|$ раз. Затем цикл, начавшийся в строчке 13, будет выполнен единожды для каждого ребра в графе, то есть $|E|$ раз. Таким образом, сложность поиска в ширину — $\Theta(|V| + |E|)$, точно такая же, как с поиском в глубину, что очень приятно, так как, получается, у нас в распоряжении два алгоритма для поиска по графу, у них одинаковая сложность и каждый из них, пусть и по-своему, но безошибочно справляется с исследованием графа. Нам остается лишь выбрать, какой из алгоритмов больше подойдет для решения той или иной задачи.

Примечания

Основы теории графов заложены Леонардом Эйлером в 1736 году, когда он опубликовал свои исследования, отвечавшие на вопрос: можно ли за раз пройтись по всем семи мостам Кенигсберга, не проходя по одному и тому же мосту дважды (в те времена Кенигсберг входил в состав Пруссии, теперь же город переименован в Калининград и входит в состав России, в нем осталось всего пять мостов)? Эйлер доказал, что нельзя [56]; так как его оригинальный труд написан на латыни, которая, возможно, не является вашей сильной стороной, можно заглянуть в книгу Биггса, Ллойда и Уилсона [19], где приведен перевод, а также содержится множество других интересных исторических материалов по теории графов.

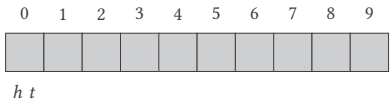
Довольно просто описывает теорию графов книга Бенжамина, Чартранда и Жанга [15], она познакомит вас с азами. Если хотите углубиться в предмет, советуем прочитать книгу Бонди и Мурти [25]. В последние годы ответвления теории графов затрагивают различные аспекты всех видов сетей; посмотрите, к примеру, книги Барабаши [10], Ньюмана [150], Дэвида и Клейберга [48] и Уоттса [214]. Изучение графов в сетях (в самых различных), во всемирной паутине и Интернете можно рассматривать как три разных направления [203], где с помощью графов объясняются различные феномены, возникающие в огромных взаимосвязанных структурах.

У поиска в глубину длинная история; его версия была опубликована в XIX веке французским математиком Шарлем Пьером Тремо; для подробного изучения данного и прочих аспектов теории графов, рекомендуем прочесть книгу Эвана [57]. Хопкрофт и Тарьян описали компьютерный поиск в глубину и высказывались в пользу представления графов с помощью списков смежности [197, 96]; советуем также посмотреть небольшую известную работу Тарьяна, посвященную структурам данных и графам [199].

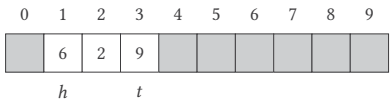
Исследование лабиринта является основой поиска в глубину, об этом в 1950 году писал Э. Ф. Мур [145]. Независимо от него к тому же выводу пришел Ч. И. Ли [126] с помощью алгоритма для дискретного рабочего поля.

Упражнения

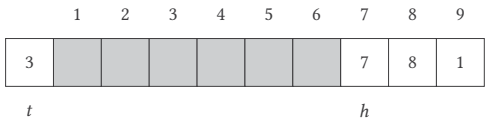
1. Во всех известных программных языках есть хорошие реализации списков, однако интересно и не сложно сделать собственную реализацию. Базовые идеи изображены на рисунках 2.13 и 2.15. Пустой список — указатель на `null`. Когда вы добавляете элемент в голову списка, вам нужно откорректировать список так, чтобы он указывал на новую голову и чтобы новая добавленная голова указывала туда же, куда указывал список до ее добавления: на предыдущую голову или `null`. Чтобы добавить элемент перед уже существующим элементом, вам надо согласовать связку с предыдущим элементом так, чтобы она указывала на новый добавленный элемент и чтобы этот элемент указывал туда же, куда указывал предыдущий элемент до добавления. Чтобы найти элемент в списке, вам нужно начать с головы и проверить, следуя указателям, каждый элемент, пока вы не найдете тот, что искали, или же `null`. Чтобы извлечь элемент, вам сперва надо его найти; как только вы его нашли, нужно сделать так, чтобы указатель, ведущий к нему, указывал на следующий элемент или `null`, если вы извлекли последний элемент.
2. Очередь можно реализовать с помощью массива, в котором вы отслеживаете индекс ее головы, *h*, и индекс ее хвоста, *t*. Изначально голова и хвост равны 0:



Когда вы добавляете элемент в очередь, то повышаете индекс хвоста; точно так же, когда вы убираете из очереди элемент, то повышаете индекс головы. После добавления 5, 6, 2, 9 и после изъятия 5 массив будет:



Если в массиве n элементов, когда голова или хвост достигают (n – 1) элемента, они переходят к позиции 0. Таким образом, после нескольких добавлений и извлечений очередь может выглядеть так:



Реализуйте очередь таким вот способом. Очередь опустеет, когда голова доберется до хвоста, и заполнится, когда хвост почти что упрется в голову.

3. Реализуйте поиск в глубину (рекурсивный и не рекурсивный) и поиск в ширину, используя вместо списка смежности, который мы опробовали, матрицу смежности.
4. Поиск в глубину можно использовать не только для исследования лабиринтов, но и для их создания. Мы начинаем с графа с узлами $n \times n$, приведенными в таблице; например, если $n = 10$ и мы называем узлы согласно их (x, y) позиции, мы получаем:

Мы начинаем с узла в графе. Помечаем узел как посещенный и в случайном порядке переходим к соседним узлам. Для каждого из узлов, что мы еще не посетили, фиксируем

связку, с помощью которой мы переходим в новый узел, и, добравшись до узла, снова повторяем всю процедуру. Это означает, что мы выполняем обход в глубину, при котором посещаем соседние узлы в произвольном порядке, фиксируя связки, по которым мы переходим. Когда поиск в глубину завершится, у нас получится новый граф: в нем будут все те же $n \times n$ узлов и подмножество связок, по которым мы прошли. Это и есть наш лабиринт. Напишите программу, которая создаст такой вот лабиринт. Будет полезно изучить библиотеку изображений, чтобы визуализировать полученные результаты.

5. Как видно на рисунке 2.10, не так уж просто опознать с первого взгляда двудольный граф. Но мы можем выполнить следующую процедуру. Мы пройдем через граф, раскрасив узлы в два разных цвета. Когда мы раскрашиваем узел, мы, конечно же, не красим его на самом деле, а просто отмечаем как окрашенный. Возьмем «красный» и «зеленый» цвета, покрасим первый узел в красный, следующий — в зеленый и так далее. Если во время прохождения нам попадется соседний узел, окрашенный в тот же цвет, что и нынешний наш узел, то можно смело сказать, что граф у нас не двудольный. Если же мы закончим прохождение без подобных приключений, тогда наш граф двудольный. Реализуйте данную процедуру в виде алгоритма для обнаружения двудольных графов.

3 Сжатие

Мы храним цифровые данные с помощью рядов нулей и единиц, то есть с помощью битов. Каждый встреченный вами текст, включая данную книгу, — это последовательность битов. Задумайтесь над фразой «I am seated in an office» («Я сижу в кабинете»). Как ее воспринимает компьютер?

Компьютеру нужно закодировать каждую букву предложения в набор битов. Существует множество способов, но в английском языке чаще всего используется кодировка ASCII. Она расшифровывается как «американский стандартный код для информационного обмена» и представляет собой кодировку, которая использует 128 символов для воплощения английского алфавита, плюс знаки препинания и управляющие символы. Это не новомодный стандарт; кодировка используется с 1960-х годов и успела претерпеть различные изменения. Она прекрасно работает с языками, использующими латиницу, но не приживается во всех других языках, поэтому для них нужна другая кодировка, такая как Юникод, в которой представлены более 110 тысяч символов из различных языков. ASCII предоставляет только 128 символов, так как в ней используется по семь бит на каждый знак, и число возможных знаков, которые могут быть воплощены с помощью семи битов, $2^7 = 128$. Это вытекает из факта, что с помощью семи битов мы можем представить 2^7 различных чисел, и каждому из этих чисел соответствует свой знак.

Число различных знаков, которые мы можем представить с помощью семи битов, равно числу различных комбинаций битов, возможных при семи битах; вы можете это увидеть на рисунке 3.1. Каждый бит, изображенный на рисунке в виде прямоугольника, может принимать значение 0 или 1, поэтому существует две разные комбинации, 0 и 1 для каждого бита. Если мы начнем с конца ряда битов и возьмем два последних, то мы увидим, что

для каждого есть две возможные комбинации, таким образом, для обоих получается 2 × 2 возможные комбинации (00, 01, 10, 11). С тремя битами у нас есть две возможные комбинации для каждой возможной комбинации из двух битов, все вместе получается 2 × 2 × 2, и так далее, пока мы не закроем все семь бит. В целом, если у нас есть *n* бит, мы можем выразить 2^n различных чисел.

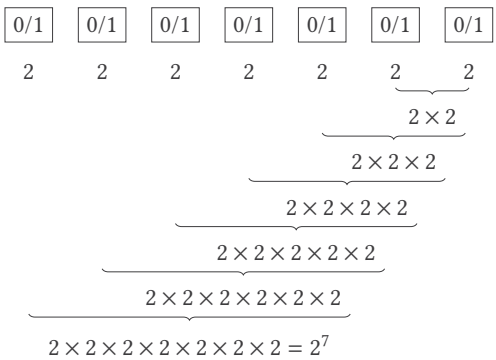


Рисунок 3.1. Число возможных знаков, представленных семью битами

Вы можете посмотреть кодировку ASCII в таблице 3.1. Каждому знаку соответствует уникальная комбинация битов, которой, в свою очередь, соответствует уникальное число: значение набора данных в двоичной системе. В таблице расположены 8 рядов по 16 элементов в каждом; ради удобства мы используем для столбцов шестнадцатеричную систему счисления. В шестнадцатеричных числах нет ничего особенного. Вместо девяти цифр 0, 1 …, 9, мы прибегаем к шестнадцати однозначным числам 0, 1, …, 9, A, B, C, D, E, F. Число A в шестнадцатеричной системе — это 10, число B — 11, и так до числа F, которое 15. Помните, что в десятичных числах значением числа, такого как 53, является 5 × 10 + 3, и, как правило, число, образованное из однозначных чисел $D_n D_{n-1} \ldots D_1 D_0$ имеет значение $D_n \times 10^n + D_{n-1} \times 10^{n-1} + \ldots + D_1 \times 10^1 + D_0 \times 10^0$. В шестнадцатеричной системе логика все та же, только в наших расчетах мы вместо основания степени 10 используем основание 16. Число $H_n H_{n-1} \ldots H_1 H_0$ в шестнадцатеричной системе имеет значение $H_n \times 16^n + H_{n-1} \times 16^{n-1} + \ldots + H_1 \times 16^1 + H_0 \times 16^0$.

Например, число 20 в шестнадцатеричной системе имеет значение $2 \times 16 + 0 = 32$, и число 1B в той же системе имеет значение $1 \times 16 + 11 = 27$. Во избежание недоразумений мы обычно добавляем к шестнадцатеричным числам префикс 0× для пояснения, что мы используем шестнадцатеричную систему счисления. Таким образом, мы пишем 0×20, давая понять, что мы используем не десятичную, а шестнадцатеричную систему. Мы точно так же пишем 0×1B, хотя и так ясно, что число не десятичное; мы записываем его так для общего порядка.

Кстати, описанная нами логика работает не столько с основаниями 10 и 16, но и с другими основаниями степени; точно так же и с двоичной системой счисления, когда основой в наших расчетах становится 2. Значение двоичного числа, образованного из битов $B_n B_{n-1} \ldots B_1 B_0$, представляет собой $B_n \times 2^n + B_{n-1} \times 2^{n-1} + \ldots + B_1 \times 2^1 + B_0 \times 2^0$.

Это все примеры позиционных систем счисления, то есть числовых систем, в которых значение числа определяется позицией однозначных чисел и основой степени заданной числовой системы. Основное правило для нахождения значения в системе с основой степени b таково:

$$X_n X_{n-1} \ldots X_1 X_0 = X_n \times b^n + X_{n-1} \times b^{n-1} + \ldots + X_1 \times b^1 + X_0 \times b^0$$

Если вы вместо b подставите 2, 10 или 16, то получите формулы, которые мы использовали выше. Как правило, в работе с числами в различных системах счисления используется запись $(X)_b$. Например $(27)_{10} = (1B)_{16}$.

Наверняка у вас возник вопрос, зачем нам все эти шестнадцатеричные числа. Данные хранятся в памяти компьютера в виде множества байтов, в каждом байте по 8 битов. Если вы еще раз взглянете на рисунок 3.1, то заметите, что четыре бита образуют комбинации $2 \times 2 \times 2 \times 2 = 2^4 = 16$. С помощью одной шестнадцатеричной цифры мы можем представить все комбинации, образованные из четырех битов. Затем, путем разделения байта на две части (каждой по четыре бита), мы можем представить все возможные байты, разделив их напополам и пользуясь лишь двумя шестнадцатеричными символами, от 0×0 до 0×FF. Возьмем,

к примеру, один байт 11100110. Если мы разделим его надвое, мы получим 1110 и 0110. Мы используем каждый из них как двоичное число. Двоичное число 1110 имеет значение 14, так как оно $2^3 + 2^2 + 2^1$. У двоичного числа 0110 значение 6, так как оно $2^2 + 2^1$. Шестнадцатеричное число со значением 14 — это 0×E, а шестнадцатеричное число со значением 6 — это 0×6. Следовательно, мы можем записать байт 11100110 как 0×E6. Такая запись более изящна, чем ее десятичный вариант — 230; к тому же более простого способа образовать 230 не существует: только подробный расчет $2^7 + 2^6 + 2^5 + 2^2 + 2^1$ и все; в то же время с помощью эквивалентной шестнадцатеричной системы мы сразу получаем $E \times 16 + 6 = 14 \times 16 + 6$.

Если вы все еще сомневаетесь в пользе шестнадцатеричной системы, обратите внимание, что с ней вы можете записывать «волшебные числа» вроде 0×CAFEBABE. Как только вы его напишите, 0×CAFEBABE поможет распознать скомпилированные файлы программ, написанных на программном языке Java. Написание английских слов с помощью шестнадцатеричных символов называется Hexspeak, и, если вы поищете, то непременно найдете интересные примеры.

Вернемся к ASCII; первые 33 символа и 128-й символ являются управляющими. Их изначальная функция — управление устройствами, например принтерами, в которых используется ASCII. Большинство из них устарели и больше не используются, хотя есть несколько исключений. Символ 32 (0×20, 33-й символ, так как мы начинаем счет с нуля) — это пробел; символ 127 (0×7F) — команда удаления; символ 27 (0×1B) — команда выхода, а символы 10 (0×A) и 13 (0×D) используются, соответственно, для разрыва строки и перевода строки: они начинают новую строку (в зависимости от операционной системы компьютера используется либо только разрыв строки, либо то и другое). Остальные символы встречаются намного реже. Например, символ 7 нужен для звонка на телетайпах.

Из таблицы 3.1 вы можете узнать, что предложение «I am seated in an office» («Я сижу в кабинете») соответствует рядам ASCII в шестнадцатеричной и двоичной системах, представленных

в таблице 3.2. Так как каждому символу соответствует двоичное число с семью битами, а в предложении содержится 24 символа, нам понадобится 24 × 7 = 168 бит.

Таблица 3.1. Кодировка ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Таблица 3.2. Пример кодировки ASCII

I		a	m		s	e	a
0x49	0x20	0x61	0x6D	0x20	0x73	0x65	0x61
1001001	100000	1100001	1101101	100000	1110011	1100101	1100001
t	e	d		i	n		a
0x74	0x65	0x64	0x20	0x69	0x6E	0x20	0x61
1110100	1100101	1100100	100000	1101001	1101110	100000	1100001
n		o	f	f	i	c	e
0x6E	0x20	0x6F	0x66	0x66	0x69	0x63	0x65
1101110	100000	1101111	1100110	1100110	1101001	1100011	1100101

3.1. Сжатие

Можем ли мы сделать лучше? Если мы найдем способ представить текст в более компактной форме, у нас останется больше свободных битов; а если учесть горы цифровых текстовых данных, которые у нас скапливаются за день, то сэкономленного места у нас будет за глаза. В самом деле, вся хранящаяся на компьютере информация находится в сжатом состоянии, и, когда нам нужно ее изучить, мы ее разворачиваем.

Если точнее, сжатие — это процесс, с помощью которого мы шифруем некий объем информации и после которого данным требуется меньшее количество битов, чем обычно. Существует два вида сжатия, в зависимости от способа сокращения битов.

Если сокращение происходит путем нахождения и устранения избыточной информации, мы говорим о сжатии без потери данных. Простейшая форма сжатия без потери — кодирование

по длинам серий. Например, представьте черно-белое изображение. Каждая его строка представляет собой серию черных и белых пикселей, как здесь:

□□□□■□□□□□□□□■■□□□□□□□□□□□□□□□■■□□□□□□□□□□□□□

Кодирование по длинам серий использует серии данных, то есть ряды с одним и тем же значением, представленные в виде значения и его количества. Серия, изображенная выше, может быть представлена как:

□4■1□9■2□15■2□13

Как видно, места нам потребовалось намного меньше и при этом вся информация сохранена: мы без проблем можем восстановить первоначальный вид строки.

При сжатии без потери данных сокращение происходит за счет того, что мы находим второстепенную информацию и убираем ее, поэтому наши потери совершенно незначительны. Например, изображения в формате JPEG — это изображения претерпевшие некоторые потери по сравнению с оригиналом, точно так же, как и видео в формате MPEG-4 и музыкальные файлы формата MP3: MP3 файлы весят намного меньше, чем оригиналы, а потеря в звуке едва ли заметна для человеческого уха (хотя кое-кто способен заметить разницу).

Прежде чем продолжить, давайте обратимся к прошлому и вспомним старейшие кодировки, которыми пользовались для хранения информации. В таблице 3.3 изображена азбука Морзе, придуманная в 1836 году Самюэлем Ф. Б. Морзе, Джозефом Генри и Альфредом Вейлом для отправки сообщений с помощью телеграфа (на самом деле речь идет об усовершенствованной азбуке Морзе, изначальная версия азбуки Морзе несколько отличалась от нее). Азбука Морзе шифрует символы и числа с помощью точек и тире. Если вы посмотрите, то увидите, что для разных символов в ней используется разное количество точек и тире. Когда Вейл задумался, как бы отобразить послания, ему пришло в голову использовать короткие шифры для самых распространенных символов и длинные шифры для менее употребляемых. Таким образом общее количество точек и тире должно было сократиться. Чтобы

найти наиболее употребляемые в английском языке буквы, Вейл отправился в местное газетное издательство Морристауна, штат Нью-Джерси. Там он принялся считать символы в шрифт-кассах, которые использовали наборщики. Наиболее распространенные символы должны были встречаться в шрифт-кассах намного чаще, потому что они ведь фигурировали в текстах куда чаще, чем все прочие символы. В таблице 3.3 также представлена частота повторения букв английского языка, актуальная по сей день. Вы можете убедиться, что Вейл и наборщики трудились не зря.

Таблица 3.3. Азбука Морзе

A	.-	8.04%	J	.---	0.16%	S	...	6.51%	2	..---
B	-...	1.48%	K	-.-	0.54%	T	-	9.28%	3	...--
C	-.-.	3.34%	L	.-..	4.07%	U	..-	2.73%	4	....-
D	-..	3.82%	M	--	2.51%	V	...-	1.05%	5	.....
E	.	12.49%	N	-.	7.23%	W	.--	1.68%	6	-....
F	..-.	2.4%	O	---	7.64%	X	-..-	0.23%	7	--...
G	--.	1.87%	P	.--.	2.14%	Y	-.--	1.66%	8	---..
H	....	5.05%	Q	--.-	0.12%	Z	--..	0.09%	9	----.
I	..	7.57%	R	.-.	6.28%	1	.----		0	-----

Таблица 3.4. Частота повторения символов в слове «effervescence»

E: 5	F: 2	R: 1	V: 1	S: 1	C: 2	N: 1

Мы и сегодня можем позаимствовать эту идею для более экономного представления текста: нам понадобится меньшее число бит для часто встречающихся в тексте букв и большее число бит — для редко попадающихся.

Предположим теперь, что нам надо зашифровать слово «effervescence» («вспенивание»). Мы будем использовать короткие комбинации битов для часто повторяющихся букв. В таблице 3.4 показана частота повторения букв данного слова. Удобней всего, если кратчайшая комбинация битов будет у буквы E, затем у F и C, а потом у всех остальных.

Прежде чем приступить к поиску нужного кода, подумайте, сколько места вам понадобится для шифровки «effervescence» в обычном ASCII. В слове 13 символов, значит, вам понадобится $13 \times 7 = 91$ битов. Теперь же обратите внимание, что в слове всего

семь разных символов, поэтому вам не нужен весь ASCII код, чтобы представить слово. У вас всего семь разных символов, которые вы можете представиться с помощью кодировки, используя лишь три бита: $2^3 = 8 > 7$. Следовательно, вы могли бы пронумеровать разные комбинации битов с помощью трех битов, создав кодировку, как на таблице 3.5; так как требования у всех символов одинаковые, кодировка с фиксированной длиной привязана к длине. Чтобы представить слово в данной кодировке, нам понадобится лишь $13 \times 3 = 39$ битов, что намного меньше, чем 91 бит в ASCII.

Кодирование, представленное в таблице 3.5, использует для всех символов то же количество битов. Так что же с нашей целью использовать разное количество битов в зависимости от частоты повторения символов? То есть где кодирование с переменной длиной?

Таблица 3.5. Кодирование с фиксированной длинной для слова «effervescence»

E: 000	F: 001	R: 010	V: 011	S: 100	C: 101	N: 110

Таблица 3.6. Кодирование с переменной длиной для слова «effervescence»: неверно!

E: 0	F: 1	R: 11	V: 100	S: 101	C: 10	N: 110

Таблица 3.7. Беспрефиксное кодирование с переменной длиной для слова «effervescence»: верно!

E: 0	F: 100	R: 1100	V: 1110	S: 1111	C: 101	N: 1101

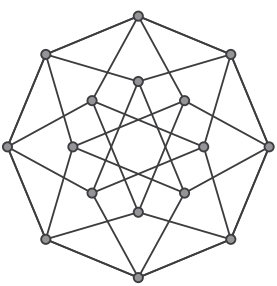
Вы можете приступить к кодировке, как в таблице 3.6, вот только она неверна. Зашифрованное слово будет 011011, и мы никак не распознаем его, когда возьмемся за расшифровку: оно начинается с E (0), а далее идут две F (1 и 1) или же R (11). Чтобы верно расшифровать наше слово, нам понадобится кодирование с переменной длиной, в котором все символы начинаются поразному, то есть ни одна из комбинаций кода символа не является префиксом другой комбинации того же кода. Такое кодирование называется беспрефиксным.

В таблице 3.7 показано беспрефиксное кодирование с переменной длиной для слова «effervescence». Слово зашифровано в 01001000110011100111101011011010, в 32 бита, и еще компактней, чем прежде. Действительно, нам приходится использовать четыре бита для некоторых символов, для которых мы используем три бита в кодировании с фиксированной длиной, однако такая расточительность компенсируется фактом, что мы используем всего один бит для самой часто повторяющейся буквы и два бита для следующей самой распространенной буквы, так что в итоге мы выигрываем. Как у нас получилась таблица 3.7? Наверняка с помощью какого-то алгоритма, который создает беспрефиксную кодировку с переменной длиной.

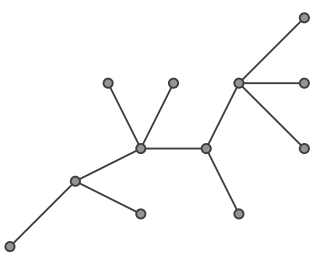
Так и есть, однако прежде, чем мы опишем его, вам следует познакомиться с некоторыми структурами данных.

3.2. Деревья и очереди с приоритетом

Дерево — это неориентированный связный граф, в котором нет циклов, то есть ациклический неориентированный связный граф. На рисунке 3.2 изображены два графа. Левый граф не является деревом, а правый — как раз дерево и есть.



(a) Граф, но не дерево



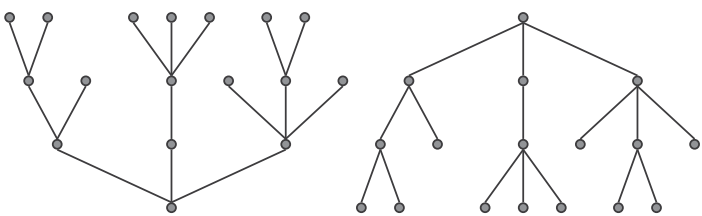
(b) Граф, являющийся деревом

Рисунок 3.2. Графы и деревья

Обычно, когда мы рисуем деревья подобным образом, они напоминают нам обычные, живые деревья. Мы назначаем один из узлов корнем дерева, и все соседние узлы, соединенные с ним, становятся его детьми, будь они вверху или внизу. Узлы, не имею-

щие детей, называются листьями. Все дети корневого узла подчиняются одному и тому же правилу, что приводит нас к еще одному, обратному определению дерева: дерево — это структура, содержащая корневой узел, который может иметь несколько узлов, соединенных с ним. Каждый такой узел является корнем другого дерева. Посмотрите на дерево рисунка 3.3. Слева дерево растет вверх, как и полагается обычным деревьям. Справа же мы видим, что то же самое дерево растет вниз. Большинство деревьев, которые вы встретите в компьютерной области, растут именно вниз.

Количество детей узла, или дочерних узлов, называется степенью узла. Сейчас мы будем иметь дело с бинарными, то есть двоичными деревьями. Бинарное дерево — это дерево, в котором каждый узел имеет не более двух детей, то есть самая большая степень узлов — 2. Более точное определение звучит так: бинарное дерево — это структура, содержащая корневой узел, который соединен максимум с двумя узлами. Каждый из этих узлов является корнем другого бинарного дерева.



(a) Дерево растет вверх (b) Дерево растет вниз

Рисунок 3.3. Деревья, растущие вверх и вниз

Дерево полезно не только из-за своей структуры, но и потому, что в каждом узле содержатся данные. Эти данные — полезное содержимое узла, и они связаны с данными, содержащимися в дочерних узлах. Их связь иерархична и отображает иерархичную структуру дерева.

Деревья — простая структура данных и возникает при различных условиях. У них есть множество операций, такие как добавление узлов, удаление узлов и поиск узла, но в данный момент нас интересует лишь операция, которая поможет нам создать дерево с корнем и двумя детьми:

- `CreateTree`(*d*,*x*,*y*) берет фрагмент данных, *d*, и два узла, *x* и *y*, чтобы создать новый узел с полезными данными *d*, а справа и слева детей — *x* и *y*; затем функция вернет новый узел. *x* и *y* (поодиночке или оба) могут быть `null`, поэтому у нас может получиться дерево с нулем, одним или двумя детьми.

Так как деревья являются графами, по ним можно пройтись в ширину, посещая все узлы сперва на одном уровне, а затем спускаясь на следующий; или же можно применить поиск в глубину, спускаясь к листьям, а затем возвращаясь назад. Деревья можно представить в виде графов, однако чаще всего мы прибегаем к иным способам представления. Самый простой — связное представление, когда в каждом узле имеются две связки, ведущие к их детям или же к `null`: связки листьев и узлов с одним ребенком ведут к `null`. Операция `CreateTree` в таком случае просто создает узел и инициализирует его связки с *x* и *y*.

На рисунке 3.4 показано, как представить бинарное дерево с помощью соединенных узлов. Для создания такого дерева мы прокладываем путь снизу вверх, от детей к родителям. Левое поддерево данного дерева появится вследствие следующих операций:

$n_1 \leftarrow$ `CreateTree`(7, `null`, `null`)
$n_2 \leftarrow$ `CreateTree`(1, `null`, `null`)
$n_3 \leftarrow$ `CreateTree`(5, n_1, n_2)

Продолжая в том же духе, мы построим целое дерево.

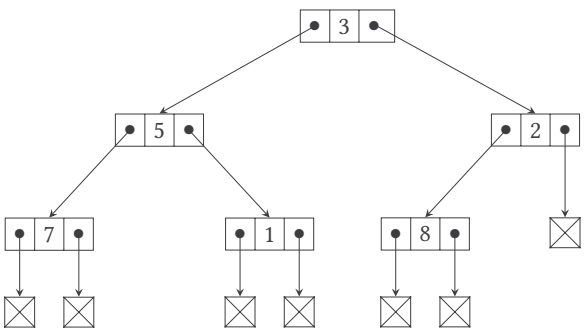


Рисунок 3.4. Бинарное дерево, представленное в виде соединенных узлов

Следующая структура данных, которая понадобится нам для создания нашего кодирования, — очередь с приоритетом. Это структура данных, в которой при извлечении элементов мы убираем элемент с наибольшим значением или же элемент с наименьшим значением, в зависимости от того, какая у нас очередь: с максимальным приоритетом или же с минимальным. Если вы убираете элемент из очереди с максимальным приоритетом, вы убираете элемент с наибольшим значением, или с высшим приоритетом. Затем, если вам снова надо изъять элемент, вы убираете элемент с наибольшим значением среди оставшихся элементов, то есть элемент со вторым по счету высшим приоритетом, и так далее. В то же время в очереди с минимальным приоритетом вы извлекаете элемент с низшим приоритетом, а затем из оставшихся элементов снова убираете элемент с низшим приоритетом, и так далее. Таким образом, очередь с приоритетом имеет следующие операции:

- `CreatePQ()` создает новую, пустую очередь с приоритетом.
- `InsertInPQ(`*pq*, *i*`)` добавляет элемент *i* в очередь с приоритетом *pq*.
- `FindMinInPQ(`*pq*`)` или `FindMaxInPQ(`*pq*`)` возвращает минимум (для очереди с минимальным приоритетом) или максимум (для очереди с максимальным приоритетом) из всех элементов очереди. `FindMinInPQ(`*pq*`)` и `FindMaxInPQ(`*pq*`)` возвращают только минимальное или максимальное значение, но саму очередь не изменяют: значение остается в очереди.
- `ExtractMinFromPQ(`*pq*`)` или `ExtractMaxFromPQ(`*pq*`)` убирает и возвращает минимум (для очереди с минимальным приоритетом) или максимум (для очереди с максимальным приоритетом) из элементов очереди.
- `SizePQ(`*pq*`)` возвращает число элементов в очереди с приоритетом *pq*.

3.3. Код Хаффмана

Теперь у нас есть очередь с минимальным приоритетом, так что мы можем создать беспрефиксный код в виде бинарного дерева.

Такой код называется кодом Хаффмана, в честь Дэвида Хаффмана, который разработал его в 1951 году, будучи аспирантом. Код Хаффмана, как мы увидим дальше, является эффективным алгоритмом сжатия без потерь, который использует повторяющиеся символы, находящиеся в информации, которую мы собираемся сжать.

Начнем с очереди с приоритетом, чьи элементы представляют собой бинарные деревья. В листьях этих бинарных деревьев будет находиться буква и ее частота в тексте. Изначально в каждом бинарном дереве будет по одному узлу, которые в таком случае будут являться одновременно и листом, и корнем. Все для того же слова «effervescence» мы запускаем очередь с приоритетом, как в первом ряду рисунка 3.5, где минимальное значение находится слева.

Мы дважды берем минимальный элемент очереди. Они образуют два дерева, состоящих из одного узла с минимальной частотой, — для букв R и N. Мы создаем новое бинарное дерево с новым корнем, чьими детьми становятся два дерева, которые мы только что забрали из очереди с приоритетом. У корня нового дерева частой служит сумма частот повторений букв R и N, и этот корень обозначает общую частоту обеих букв. Мы помещаем только что созданное дерево в нашу очередь, что показано на рисунке 3.5. То же самое мы проделываем в третьем ряду со следующими двумя узлами, V и S. После мы объединяем два дерева, полученных в двух предыдущих этапах, и создаем дерево со всеми четырьмя узлами: R, N, S, V. Мы продолжаем процесс до тех пор, пока все узлы не окажутся в одном дереве.

Данная процедура описана в алгоритме 3.1. Мы задаем алгоритму очередь с приоритетом. Каждый элемент приоритетного дерева представляет собой дерево, состоящее из одного элемента и содержащее в себе букву и ее частоту. В алгоритме мы полагаем, что у нас есть функция `GetData`(*node*), которая возвращает данные, хранящиеся в узле; в данном случае частоту, хранящуюся в каждом узле. Пока в очереди с приоритетом содержится более одного элемента (строчка 1), алгоритм вытаскивает из очереди с приоритетом два элемента (строчки 2 и 3), добавляет их частоту (строчка 4), создает новое бинарное дерево с корнем, представля-

ющим их сумму, и двумя элементами-детьми (строчка 5), а затем помещает это дерево обратно в очередь (строчка 6).

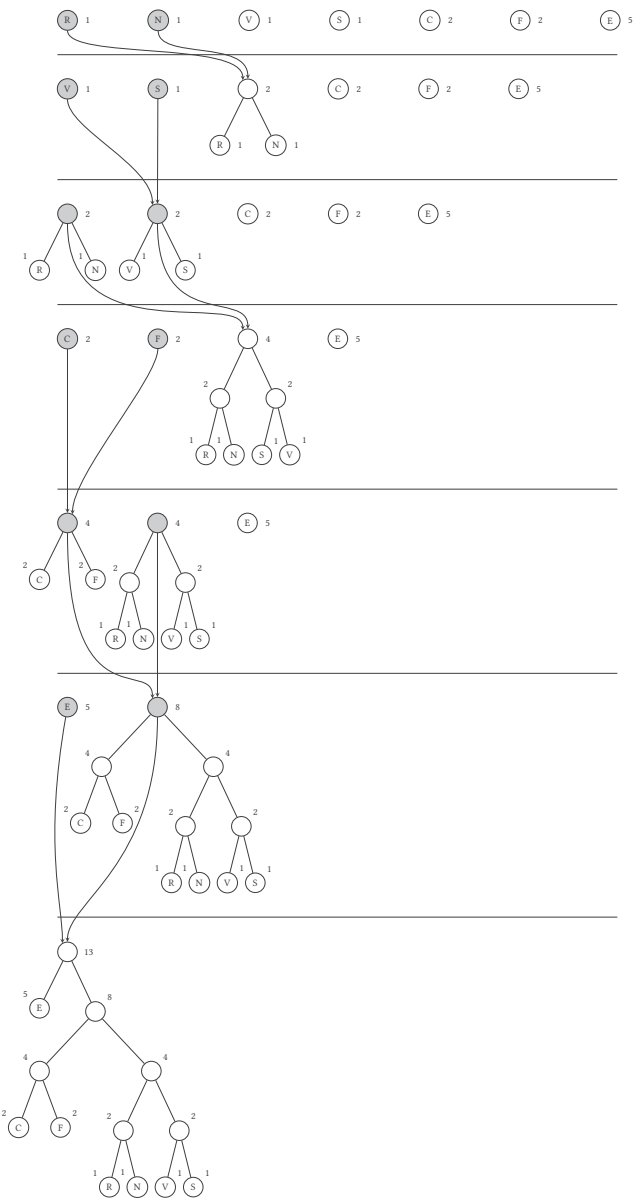


Рисунок 3.5. Составление кода Хаффмана для слова «effervescence»

В конце алгоритма в очереди с приоритетом находится одно бинарное дерево, описывающее наш код, который мы извлекаем из очереди и возвращаем (строчка 7). Чтобы найти кодирование нужной нам буквы, мы проходимся по дереву от корня до листа, отвечающего за эту самую букву. Каждый раз, когда мы идем налево, мы добавляем в кодировку 0, каждый раз, когда мы идем направо, — 1. Конечный ряд и будет кодом Хаффманна для выбранной нами буквы. Таким образом, в нашем примере E — 0, F — 100, C — 101, и так далее для остальных букв, пока наша таблица 3.7 не заполнится; все пути изображены на рисунке 3.6.

Алгоритм 3.1. Написание кода Хаффмана

`CreateHuffmanCode(`pq`)` $\rightarrow$ hc

Вводные данные: pq, очередь с приоритетом.

Выводимые данные: hc, бинарное дерево, представляющее код Хаффмана.

```
1 while SizePQ(pq) > 1 do
2     x ← ExtractMinFromPQ(pq)
3     y ← ExtractMinFromPQ(pq)
4     sum ← GetData(x) + GetData(y)
5     z ← CreateTree(sum, x, y)
6     InsertInPQ(pq, z)
7 return ExtractMinFromPQ(pq)
```

Обратите внимание, чтобы закодировать строки символов с помощью кода Хаффмана, нам нужно дважды пройтись по ним, поэтому данный метод называется двухпроходный. Мы проходим первый раз, чтобы узнать частоту букв и построить код Хаффмана. Затем мы проходимся по второму разу и зашифровываем с помощью кода Хаффмана каждую букву. В результате мы получаем сжатый текст.

Чтобы развернуть данные, сжатые кодом Хаффмана, нам понадобится дерево Хаффмана, которое мы использовали во время сжатия. Оно должно сохраниться как часть сжатого файла. Сперва мы достаем из файла дерево Хаффмана (оно должно храниться в знакомом формате), а затем приступаем к чтению ряда битов в конце файла. Мы спускаемся по дереву Хаффмана, используя для навигации биты, которые мы прочли в фай-

ле. Каждый раз, когда мы доходим до листа, мы выводим букву и возвращаемся к корню дерева, а потом считываем из файла следующий бит.

Например, если нам надо развернуть бинарную последовательность битов 01001000110011100111101011011010 и дерево на рисунке 3.6, мы начинаем наш путь с корня дерева. Мы считываем бит 0, который приводит нас к букве E, первой букве нашего итога. Затем мы возвращаемся к корню. Считываем бит 1, который ведет нас направо, потом к биту 0 и снова к биту 0, что приводит нас к букве F, второй букве нашего слова. Мы возвращаемся к корню дерева и повторяем тот же процесс, пока не разберемся с оставшимися битами.

Алгоритм 3.1 использует очередь с приоритетом, в которой содержится бинарное дерево. Теперь вы можете поинтересоваться, как очередь с приоритетом творит такие чудеса и каким образом работают `ExtractMinFromPQ`(*pq*) и `InsertInPQ`(*pq*, *i*)?

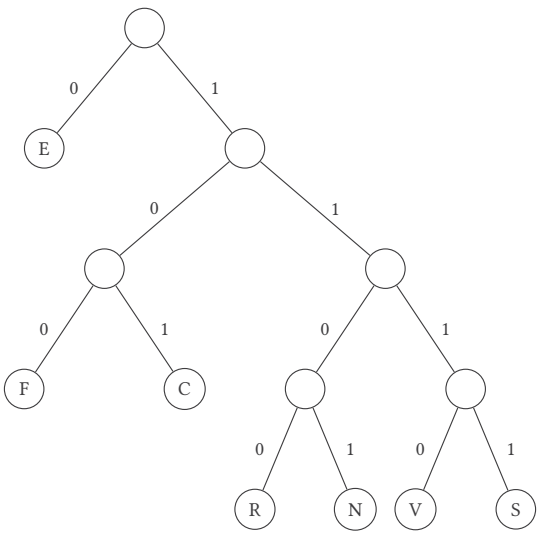
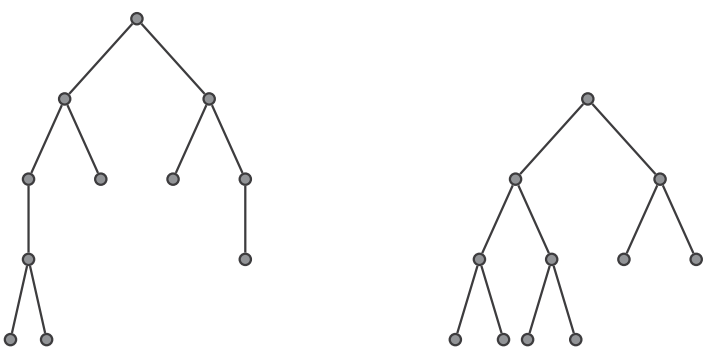


Рисунок 3.6. Код Хаффмана для слова «effervescence»

Очереди с приоритетами могут быть реализованы в виде деревьев. Чтобы увидеть как, обратите сперва внимание на два дерева на рисунке 3.7. Оба дерева — бинарные, у обоих одина-

ковое количество узлов, 11. Однако левое уходит глубже, чем правое. Дерево справа имеет минимальное количество уровней для числа его узлов. Дерево с минимальным количеством уровней называется полным деревом. В бинарном дереве самый правый узел на самом последнем уровне называется последним узлом.

Очереди с приоритетом реализуются в виде куч. Это полные бинарные деревья, в которых каждый узел больше либо равен или же меньше либо равен своим дочерним узлам. Максимальная куча — бинарное дерево, в котором значение каждого узла больше или равно значению дочерних узлов. Это означает, что из всех узлов дерева корень кучи всегда будет иметь наибольшее значение. И наоборот, бинарное дерево, в котором значение каждого узла меньше или равно значению дочерних узлов, представляет собой минимальную кучу. В минимальной куче корень всегда имеет наименьшее значение. Таким образом, очередь с минимальным приоритетом реализуется в виде минимальной кучи.



(a) Бинарное дерево (b) Полное бинарное дерево

Рисунок 3.7. Бинарное дерево и полное бинарное дерево

Чтобы лучше понять, представьте минимальную кучу как набор грузов с разным весом, плавающие в жидкости. Узлы с наибольшим весом погрузятся ниже, чем узлы с весом полегче. Чтобы добавить узел к минимальной куче, мы добавляем его к самому нижнему уровню и делаем последним узлом. Затем, если

он больше своего родителя, он всплывет наверх, поменявшись с ним местами. При необходимости так повторяется, пока узел не займет свое место. Его место может быть в корне или же где-то внизу, если мы найдем более легкий узел, который затем станет его родителем. Все это описано в алгоритме 3.2 и изображено на рисунке 3.8. В данном алгоритме мы полагаем, что у нас есть функция `AddLast`(*pq*, *c*), добавляющая *c* в качестве последнего узла в очередь с приоритетом *pq*, функция `Root`(*pq*), возвращающая корень очереди с приоритетом, `Parent`(*c*), которая возвращает родителя узла *c* в очередь с приоритетом, `getData`(*x*), возвращающая данные, хранящиеся в узле *x*, и `Exchange`(*pq*, *x*, *y*), которая меняет значения узлов *x* и *y* в дереве. Обратите особое внимание: мы меняем значения узлов, но не сами узлы. Если бы мы переместили сами узлы, тогда пришлось бы перемещать все поддерево, корнем которого является перемещенный нами узел. В первой строчке алгоритма мы добавляем элемент в конец очереди; затем цикл в строчках 2–5 поднимает его в нужное место. Элемент поднимается до тех пор, пока не достигнет корня дерева и пока его значение ниже родительского (строчка 2); если данные условия соблюдены, мы меняем местами его и родительские значения (строчки 3–5). Для этого мы в строчке 3 используем *p*, чтобы указать на родительский узел *c*, далее в строчке 4 мы вызываем `Exchange`(*pq*, *c*, *p*), а в строчке 5 делаем так, чтобы *c* указывал на *p* (родителя *c*). С помощью последней операции мы поднимаемся по дереву на один уровень выше.

Алгоритм 3.2. Очередь с приоритетом, добавление минимальной кучи

`InsertInPQ`(*pq*, *c*)

Вводные данные: *pq*, очередь с приоритетом, и элемент *c*, который нужно добавить в очередь.

Выводимые данные: элемент *c* добавлен в *pq*

```
1  AddLast(pq, c)
2  while c ≠ Root(pq) and GetData(c) < GetData(Parent(c)) do
3      p ← Parent(c)
4      Exchange(pq, c, p)
5      c ← p
```

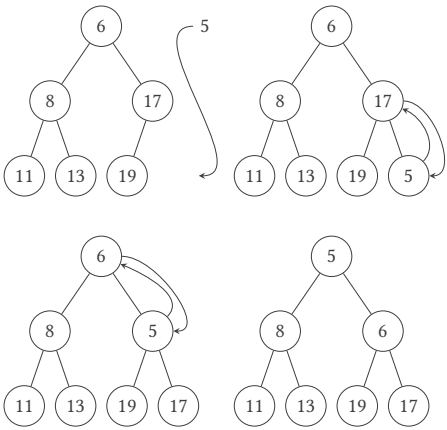


Рисунок 3.8. Добавление в очередь с приоритетом

Таким же образом мы извлекаем из очереди с приоритетом минимум. Мы вынимаем корень очереди, который по определению является узлом с наименьшим значением. Затем мы берем последний узел и кладем его в корень. Если он меньше дочернего узла, тогда наша работа завершена. Если же нет, узел погрузится на уровень ниже: он поменяется с самым маленьким из детей. Мы будем повторять процедуру до тех пор, пока не обнаружим, что наш узел больший из детей, или же пока не достигнем дна. Алгоритм 3.3 описывает нам, как все осуществилось, а рисунок 3.9 изображает алгоритм в действии. Мы встречаем здесь несколько новых функций: `ExtractLastFromPQ`(*pq*), которая извлекает последний элемент из очереди *pq*, `Children`(*i*), которая возвращает детей узла, `HasChildren`(*i*), которая возвращает `true`, если у узла есть дети, и `false`, если нет, и `Min`(*values*), которая возвращает минимальное из всех пройденных значений.

`ExtractMinFromPQ` назначает корнем очереди с приоритетом переменную *c* (строчка 1); это тот самый минимум, который мы хотим извлечь из очереди. Вся работа алгоритма заключается в том, чтобы перестроить очередь после извлечения минимума таким образом, чтобы она продолжала оставаться минимальной кучей. В строчке 2 мы кладем последний элемент в корневую по-

зицию и в строчке 3 назначаем новым, временным корнем переменную *i*. Затем цикл в строчках 4–9 ставит *i* в нужную позицию. Пока у *i* есть дети (строчка 4), берется дитя с минимальным значением и обозначается *j* (строчка 5).

Алгоритм 3.3. Очередь с приоритетом, минимальная куча извлекает минимум

```
ExtractMinFromPQ(pq) → c
    Вводные данные: pq, очередь с приоритетом.
    Выводимые данные: c, минимальный элемент очереди.
1   c ← Root(pq)
2   Root(pq) ← ExtractLastFromPQ(pq)
3   i ← Root(pq)
4   while HasChildren(i) do
5       j ← Min(Children(i))
6       if GetData(i) < GetData(j) then
7           return c
8       Exchange(pq, i, j)
9       i ← j
10  return c
```

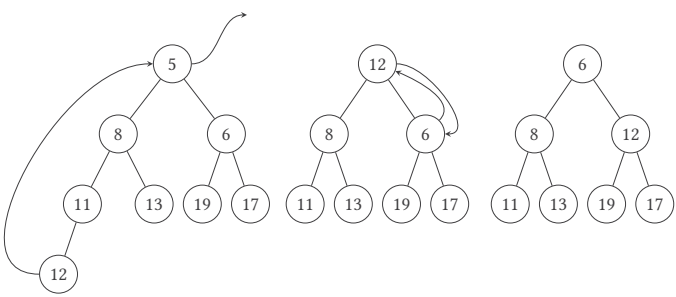


Рисунок 3.9. Изъятие минимума из очереди с приоритетом

Если значение *j* не меньше значения *i*, тогда наша работа окончена, перестройка завершена, и мы можем вернуть *c* из строчки 7. Если нет, тогда значения *i* и *j* меняются местами, как прописано в строчке 8, и *i* указывает на *j* (строчка 9). Если мы выходим из цикла, потому что добрались до дна, мы возвращаем *c* (строчка 10).

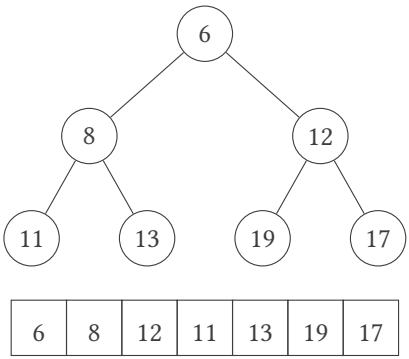


Рисунок 3.10. Изображение массива для очереди с приоритетом

Пара заключительных слов по данным алгоритмам. Мы упомянули несколько вспомогательных функций в алгоритмах 3.2 и 3.3. Оказывается, что написать их довольно просто, если мы реализуем очередь с приоритетом в виде массива, корень кучи которого находится в позиции 0. Затем для каждого узла i, его левое дитя находится в позиции $2i + 1$, а его правое дитя — в позиции $2i + 2$; взгляните на рисунок 3.10. В то же время, если мы находимся в узле i, его родитель находится в позиции $\lfloor i - 1/2 \rfloor$, где $\lfloor x \rfloor$ — это целая часть числа x (то же самое, что его дно). Если очередь с приоритетом содержит n элементов, тогда для того, чтобы проверить, есть ли у узла в позиции i дети, нам надо всего лишь проверить, что $2i + 1 < n$. В конце, чтобы извлечь последний элемент из очереди с приоритетом, состоящей из n элементов, нам нужно лишь взять n-й элемент и уменьшить размер очереди на один. Помните, мы говорили, что деревья обычно представлены с помощью связной структуры? Как правило, но не всегда, и сейчас мы видим, что нам гораздо удобней представить кучу, которая является деревом, с точки зрения массива.

При таком подходе для создания кода Хаффмана мы используем два разных представления деревьев. С помощью матрицы мы представляем дерево, лежащее в основе очереди с приоритетом; с помощью связных деревьев мы представляем дерево, содержащее код Хаффмана. Верхние ряды рисунка 3.5 можно реализовать в виде массива, элементами которого будут связные деревья.

Эффективен ли наш алгоритм для построения кодов Хаффмана? Если вы еще раз взглянете на алгоритм 3.1, вы заметите, что цикл выполнен ровно $n - 1$ раз, где n — это количество символов, которые мы хотим зашифровать. Объяснение этому следующее. При каждом проходе цикла мы делаем два извлечения и одно добавление, поэтому размер очереди с приоритетом сокращается на один. Итерации прекращаются, когда в очереди остается только один элемент. Изначально в ней было n элементов, по одному на каждый символ, который мы хотим зашифровать, поэтому после $n - 1$ итераций цикла размер очереди будет равен единице, и мы можем вернуть этот самый один элемент.

Теперь мы должны коснуться извлечений и добавлений. В алгоритме 3.2 процесс добавления может потребовать, самое большое, столько добавлений, сколько нужно на всю глубину кучи. Поскольку наша куча является полным бинарным деревом, у нас имеется бинарный логарифм ее размера, $\lg n$, потому что, чтобы пройти от узла до его родителя, нам надо поделить на два, и чтобы добраться до корня, нам нужно повторить эту процедуру для каждого уровня до самого верха. Почти то же самое в алгоритме 3.3, процесс извлечения может потребовать, самое большое, столько изъятий, сколько нужно на всю глубину кучи. Таким образом, построение кода Хаффмана может потребовать два извлечения и одно добавление $n - 1$ количество раз, или же $O((n - 1)3 \lg n$, что равно $O(n \lg n)$.

3.4. Сжатие Лемпеля — Зива — Велча

Основная идея кода Хаффмана в том, чтобы для часто встречающихся элементов использовать коды покороче, а для редко встречающихся — подлиннее. А что если вместо длины кодирования мы станем изменять длину элементов, которые хотим зашифровать? Именно такая идея лежит в основе схемы сжатия Лемпеля — Зива — Велча (LZW), эффективного и очень простого способа реализации алгоритма, придуманного Абрахамом Лемпелем, Якобом Зивом и Терри Велчем.

Представим, что у нас есть текст, закодированный в ASCII, что, как мы видели, требует семи битов на каждый символ. Но вот мы

вдруг немного сошли с проторенной дорожки и решили использовать для нашего кодирования побольше битов: например, будем зашифровывать каждый элемент с помощью восьми битов, вместо минимальных семи. Казалось бы, дурь и блажь, но все же тут есть здравый смысл. С помощью восьми битов мы можем представить $2^8 = 256$ различных элементов. Мы используем числа от 0×00 (0) до 0×7F (127), чтобы представить символы ASCII (вспомните таблицу 3.1), и у нас еще остаются числа от 0×80 (128) до 0×FF (255), которыми мы можем представить еще что угодно. Мы используем эти 128 чисел для представления групп из двух, трех или более символов, вместо одного. Группы из двух букв называются биграммы, группы из трех букв — триграммы, названия групп с бóльшим количеством символов составляется из значения числа и суффикса «грамм», например четыреграмма, а общее их название — *n*-граммы. *N*-граммы с одним элементом — это униграммы. Следовательно, числа от 0 до 127 будут использоваться для представления униграмм, в то время как числа от 128 до 255 будут использоваться для представления *n*-граммов размером больше, чем один, то есть *n*-граммами, которые не являются униграммами.

А какими *n*-граммами? Неизвестно, мы не можем предвидеть, какие *n*-граммы появятся в нашем тексте. Для 26 букв английского алфавита существует $26 \times 26 = 26^2 = 676$ возможных биграммов, $26^3 = 17\,576$ триграммов и 26^n *n*-граммов для любого *n*. Мы сможем выбрать лишь небольшое подмножество из них всех. В частности, мы встретим *n*-граммы, когда будем проходить через текст, который собираемся сжать. Это значит, что мы будем строить наше кодирование по ходу процесса, шаг за шагом: мы будем создавать коды сжатия и тут же осуществлять сжатие — очень ловкий и изящный способ.

Рассмотрим на примере. Допустим, мы хотим сжать фразу MELLOW YELLOW FELLOW. Как и говорилось, мы вдруг решаем для каждой униграммы, каждой встреченной нами буквы использовать в восьмибитное число. Нижние семь битов будут соответствовать кодированию ASCII для заданного символа; крайний левый бит будет нулевым. Мы используем таблицу, в ко-

торой будут содержаться преобразования элементов в числовые значения.

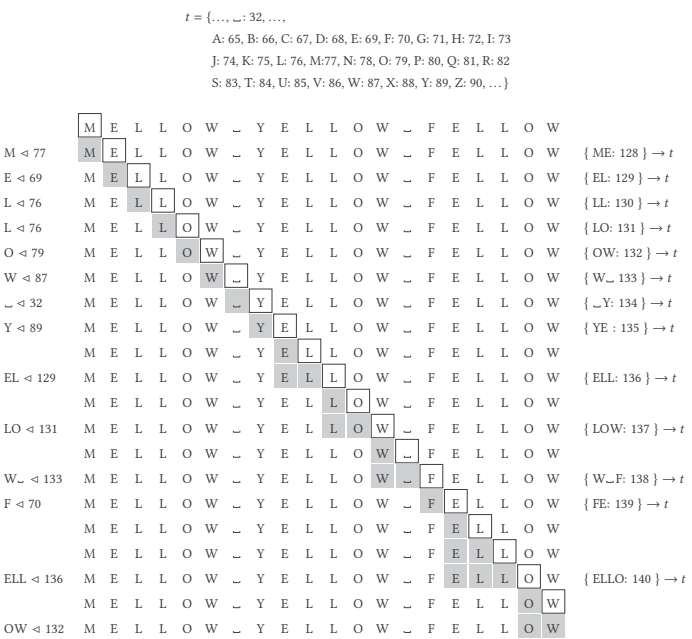


Рисунок 3.11. LZW-сжатие

На рисунке 3.11 в самом верху можно увидеть таблицу *t* с этими самыми преобразованиями. Для краткости мы показываем преобразования для заглавных символов ASCII и пробела (_). Под ними в каждой строке рисунка показано, как мы, символ за символом, читаем фразу; символ, который мы читаем в данный момент, выделяется рамками.

В первом ряду мы читаем символ, униграмму *M*. Мы проверяем, присутствует ли *M* в таблице. Да, есть. Вместо ввода ее числового значения мы решаем подождать, попадется ли нам дальше в таблице *n*-грамма подлиннее. Сейчас в нашей таблице нет *n*-грамм длиннее, но это общая логика, применимая к процессу в целом, поэтому мы будем применять ее постоянно.

Во второй строке рисунка мы читаем символ *E*. Теперь у нас появилась биграмма *ME*; на рисунке мы помечаем серым цветом

символы, являющиеся частью *n*-грамм, о которых мы знаем, что они есть в нашей таблице. Мы проверяем, присутствует ли *ME* в таблице, — ее нет. Мы выводим значение для *n*-граммы, которая есть в таблице — *M*, 77, — и добавляем в таблицу новую биграмму *ME*, приписывая ей следующее доступное числовое значение, 128. С помощью символа ◀ мы обозначаем полученный результат; можно представить, что это такой условный рупор. Полученный результат отображен в левой части рисунка: *M* ◀ 77 означает, что для *M* мы получили результат 77. В правой же части рисунка отображаются добавления в таблицу: {*ME* : 128} → *t* означает, что мы в качестве шифровки для *ME* добавили 128 в таблицу *t*. Теперь, как только нам где-то в тексте попадется *ME*, мы сразу используем 128 для обозначения обоих символов. Так как мы получили результат кодирования *M*, данная *n*-грамма нам больше не нужна. Мы сохраняем только последний прочитанный символ, *E*, используя его для начала следующей *n*-граммы. Как и в прошлый раз, *E* — униграмма, находящаяся в таблице, но нам хочется отыскать более длинные *n*-граммы, начинающиеся с *E*.

В третьей строке нашего рисунка мы читаем символ *L*. Он следует за *E*, и вместе они образуют биграмму *EL*. Такой биграммы нет в таблице, поэтому мы выводим значение *E*, 69, вставляем *EL* в таблицу со значением 129 и сохраняем символ *L* для начала следующей *n*-граммы.

Мы повторяем процесс, даже когда натыкаемся на пробел в конце MELLOW. Биграммы *W_* нет в таблице, поэтому мы выводим кодирование для *W* и добавляем в таблицу преобразование для *W_*. Мы убираем *W* и читаем *Y* из YELLOW, создавая биграмму *_Y*, которой нет в таблице, поэтому мы выводим кодирование для пробела, убираем пробел и, после прочтения *E*, формируем новую биграмму *YE*.

Теперь обратите внимание, что происходит, когда мы читаем первую *L* в YELLOW. Наша нынешняя *n*-грамма — *EL*. Когда мы проверяем, есть ли она в таблице, то узнаем, что да, есть, так как мы ее уже добавляли прежде. Поэтому мы можем попробовать удлинить существующую *n*-грамму на один символ; мы читаем вторую *L* и получаем триграмму *ELL*, которой нет в таблице,

а значит, мы добавляем ее туда со значением 136. Затем мы выводим значение биграммы *EL* (129), убираем его и начинаем новую *n*-грамму с последнего прочтенного символа, *L*.

Общая логика такова: сперва читаем символ, затем используем его для продления нынешней *n*-граммы. Если получившаяся в результате *n*-грамма есть в таблице, переходим к прочтению следующего символа и повторяем процесс. Если же полученной *n*-граммы в таблице нет, тогда вставляем ее туда, выводим код для предыдущей *n*-граммы, начинаем новую *n*-грамму с только что прочтенным символом, считываем следующий символ и снова повторяем процесс. Вкратце, мы пытаемся закодировать самую длинную из возможных *n*-грамму. Каждый раз, когда нам попадается *n*-грамма, которую мы еще не видели, мы шифруем ее, и в следующий раз, когда мы на нее наткнемся, мы спокойно сможем использовать ее. Через какое-то время у нас наберется целая группа *n*-грамм, которые, будем надеяться, еще много раз встретятся нам в тексте, и мы сэкономим на месте, нужном для нашего сообщения. Данная идея прекрасно реализуется на практике: *n*-граммы, которые мы находим, и в самом деле повторяются по ходу текста, и мы действительно экономим место.

Фраза из нашего примера состоит из 20 символов, так что для представления в ASCII ей требуется $20 \times 7 = 140$ битов. С использованием LZW, где на каждый код нужно по 8 битов, фраза кодируется в [77, 69, 76, 76, 79, 87, 32, 89, 129, 131, 133, 70, 136, 132], где содержится 14 чисел по 8 бит каждое, итого $14 \times 8 = 122$ бита. Мы ужали фразу на 80% от ее исходного размера.

Вы можете подумать, что 80% — это не так уж и много, особенно если учесть, что фраза YELLOW MELLOW FELLOW надуманная и содержит намного больше повторяющихся *n*-грамм, чем можно на самом деле ожидать от трех слов. Вы правы, однако фраза специально подбиралась так, чтобы ярко проиллюстрировать работу алгоритма даже в таком коротком тексте. Мы использовали семь битов для исходного алфавита и восемь битов для каждого кодирования. В действительности данные числа могут быть намного больше, и тогда мы можем использовать более объемные алфавиты и большее количество *n*-грамм: скажем,

8-битные алфавиты и 12-битные кодирования. Тогда коды от 0 и до 255 включительно представляют индивидуальные символы, а значения от 256 до 4095 используются для представления *n*-грамм. Чем больше текст, тем больше возможностей представить *n*-граммы, и они будут становиться все длиннее и длиннее. Если мы применим LZW с такими параметрами к «Улиссу» Джеймса Джойса, то мы сократим размер текста на 53% по сравнению с исходником. На рисунке 3.12 изображено распределенное выполнение *n*-грамм, полученных в ходе такого сжатия. Большинство *n*-грамм состоит из трех-четырех символов, но есть *n*-грамма длиной в десять символов («Stephen's_») и две *n*-граммы длиной в девять («Stephen_b» и «Stephen's»). Стивен Дедал — один из главных персонажей книги.

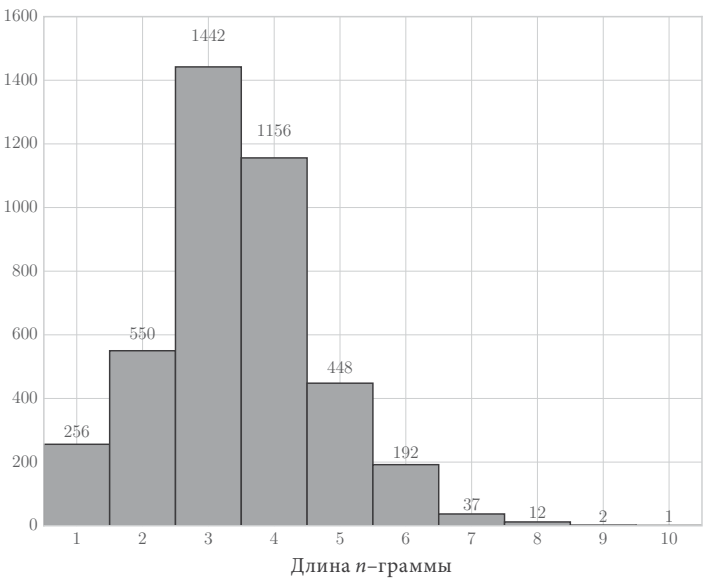


Рисунок 3.12. Распределенное выполнение LZW *n*-грамм в «Улиссе» Джеймса Джойса

Алгоритм 3.4 показывает LZW-алгоритм. Данный алгоритм полагает, что у нас есть таблица, которая работает, как и было описано, позволяя нам просматривать числовые значения, соответствующие строкам символов, и вставлять новые преобразо-

вания из строк символов в значения. Такая таблица реализуется с помощью структуры данных, которая называется картой, словарем или ассоциативным массивом. Она зовется картой, потому что она связывает элементы (или, по-другому, ключи) с соответствующими значениями. Она очень напоминает поиск по словарю, где слово — это ключ, а определение — это его значение, хотя словарям в компьютерах не нужны ни слова в качестве ключей, ни определения в качестве значений. Перед нами ассоциативный массив, который работает как обычный массив, но вместо того, чтобы связывать со значением числовой индекс, он может связывать со значением любые другие типы данных, те же строки символов. Так как нам довольно часто приходится сохранять связанные с элементами значения, словари в программах — дело обычное. Мы подробней рассмотрим словари и расскажем об их работе в главе 13. В частности, важно, что с помощью словарей мы можем выполнять операции просмотра и добавления за постоянное время, $O(1)$, поэтому они работают так же быстро, как и обычные массивы. На данный момент нам достаточно знать лишь базовые операции словарей, выполняющих следующие функции, которые мы используем в `LZWCompress`.

- `CreateMap`() создает новую пустую карту.
- `InsertInMap`(*t*, *k*, *v*) добавляет элемент *k* в карту *t* со значением *v*. Нотация {ME: 128} → *t* на рисунке 3.11 означает, что мы вызываем `InsertInMap`(*t*, “ME”, 128).
- `Lookup`(*t*, *k*) выполняет поиск элемента *k* в карте *t*; возвращает связанное с *v* значение, если он существует, или же `null`, если в карте *t* не содержится *k*. Нотация OW ◀ 132 на рисунке 3.11 означает, что вызов `Lookup`(*t* , “OW”) возвращает значение 132.

`LZWCompress` влияет на строки символов, поэтому там так же нужны какие-то функциональные возможности для строк символов. Функция `CreateString` создает новую пустую строку. Если у нас есть две строки *a* и *b*, мы можем объединить их в еще один символ: *a* + *b*. Если у нас есть строка символов, мы можем просмотреть все ее символы с помощью инструкции **foreach**. В заключение длина строки *a* задается с помощью $|a|$.

В качестве вводных данных алгоритм берет строку символов *s*, которую надо сжать, число битов, *nb*, которые мы используем для кодирования каждого элемента, и число элементов, *n* в алфавите строки, которую мы хотим сжать. В строчках 1–6 мы закладываем основу для главной части алгоритма. Сперва создаем пустой список, *compressed*, в котором будут содержаться результаты сжатия (строчка 1). Возможные значения кодирования от 0 до $2^{nb} - 1$, которые мы сохраняем в *max_code* в строчке 2. В нашем примере диапазон значений кодирования охватывает числа от 0 до 255 включительно, потому что мы имеем 8 битов для каждого из них, следовательно, *max_code* будет установлен на 255. Затем в строчке 3 мы создаем пустую карту *t*. В цикле строчек 4–6 мы вставляем в *t* связи, полученные из *k*: *v*, где *k* — строка, состоящая из одной буквы, а *i* — соответствующий числовой код. Для этого мы используем функцию `Char`(*i*), которая возвращает ASCII символы, у которых ASCII значение *i*; `Char`("A") возвращает значение 65. В строчке 6 мы помещаем в переменную *code* кодирование для следующего элемента, который будет добавлен в таблицу; когда мы приступим, он будет равняться размеру алфавита. В нашем примере в *code* будет установлена на 128. К концу строчки 6 наша таблица заполнится, как показано вверху рисунка 3.11, и мы сможем приступить к добавлению новых сопоставлений.

Мы используем *w*, чтобы сохранить прочтенную нами *n*-грамму и определить, есть ли она в таблице. Эта *n*-грамма выделена на рисунке 3.11 серым цветом. Изначально у нас вовсе не было *n*-грамм, так что в строчке 7 мы помещаем ее в пустую строку символов. Мы также прибегнем к помощи указателя *p*, чтобы указать на последний элемент списка *compressed*; изначально список пуст, поэтому он `null` (строчка 8). А теперь в цикле строчек 9–20 начинается настоящая работа. Для каждого символа *c* в строке *s*, которую мы хотим сжать (строчка 9), мы формируем новую *n*-грамму, *wc*, путем добавления к ней нашей нынешней *n*-граммы (строчка 10). Мы пробуем найти ее в нашей карте (строчка 11). Если новая *n*-грамма есть в карте (строчка 12), мы пробуем найти *n*-грамму подлиннее, поэтому мы присваиваем *w* значение *wc* (строчка 13) и повторяем цикл.

Если новой n-граммы нет в карте (строчка 14), то мы просматриваем нынешнюю n-грамму w (строчка 15), о которой мы знаем, что она есть в словаре. Откуда мы знаем? Единственный способ удлинить w — это присвоить ей значение wc в строчке 13 во время предыдущей итерации цикла, *после* того, как мы установили, что wc находится в таблице t.

Алгоритм 3.4. LZW-сжатие

`LZWCompress(`*s, nb, n*`)` → *compressed*

Вводные данные: *s*, строка, которую надо сжать; *nb*, число битов, использованных для представления элемента; *n*, число элементов в алфавите.

Выводимые данные: *compressed*, список, в котором содержатся числа представляющие *s* согласно сжатию LZW.

```
1  compressed ← CreateList()
2  max_code ← 2^nb − 1
3  t ← CreateMap()
4  for i ← 0 to n do
5      InsertInMap(t, Char(i), i)
6  code ← n
7  w ← CreateString()
8  p ← NULL
9  foreach c in s do
10     wc ← w + c
11     v ← Lookup(t, wc)
12     if v ≠ NULL then
13         w ← wc
14     else
15         v ← Lookup(t, w)
16         p ← InsertInList(compressed, p, v)
17         w ← c
18         if code ≤ max_code then
19             InsertInMap(t, wc, code)
20             code ← code + 1
21 if |w| > 0 then
22     InsertInList(compressed, p, v)
23 return compressed
```

Мы добавляем значение кодирования для w в конец списка *compressed* (строчка 16). Обратите внимание, что `InsertInList`

возвращает указатель к только что добавленному элементу, поэтому *p* будет указывать в новый конец списка, позволяя нам при следующем вызове присоединить следующее значение кодирования к концу списка, вслед за только что добавленным элементом.

Покончив с этим, мы разобрались с *w*, поэтому мы сбрасываем ее содержимое и возвращаем ее к последнему прочтенному нами символу (строчка 17), чтобы подготовиться к следующей итерации цикла. Но прежде, если возможно, мы сохраняем удлиненную *n*-грамму *wc*. Это случится, если мы не исчерпали все возможные кодирования (строчка 18). Если так, мы добавляем нашу *n*-грамму в *t* (строчка 19) и увеличиваем значение, которое мы используем для следующей закодированной *n*-граммы (строчка 20).

Строчки 21–22 заботятся о ситуации, когда во время поиска наидлиннейшей *n*-граммы мы вдруг понимаем, что наш текст, который мы собираемся сжать, закончился. Это случается в последней строке рисунка 3.11. В такой ситуации мы выводим кодирование нынешней *n*-граммы, и на этом работа окончена.

Чтобы развернуть сообщение, сжатое с помощью LZW, мы идем обратным путем. У нас есть последовательность кодирований, и мы хотим извлечь зашифрованный текст. Сначала мы не знаем, какие *n*-граммы там зашифрованы и с помощью каких кодирований, нам известны лишь кодирования для униграмм, которые представляют собой перевернутую версию таблицы, что мы использовали в начале процесса сжатия.

Если мы снова посмотрим на рисунок 3.11, мы увидим, что каждый раз, когда мы выводим результат, мы создаем *n*-грамму с этим самым результатом и следующий для прочтения символ.

Для воссоздания перевернутой таблицы кодирования, нам, следовательно, нужно зафиксировать результаты, прочесть следующее кодирование, найти соответствующее зашифрованное значение и добавить в таблицу *n*-грамму, сформированную с помощью наших предыдущих результатов и первого символа расшифрованного значения, которое мы только что нашли.

dt = {…, 32: ␣, …,
65: A, 66: B, 67: C, 68: D, 69: E, 70: F, 71: G, 72: H, 73: I
74: J, 75: K, 76: L, 77: M, 78: N, 79: O, 80: P, 81: Q, 82: R
83: S, 84: T, 85: U, 86: V, 87: W, 88: X, 89: Y, 90: Z, …}

77 ◁ M	77	69	76	76	79	87	32	89	129	131	133	70	136	132	
69 ◁ E	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 128: ME } → *dt*
76 ◁ L	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 129: EL } → *dt*
76 ◁ L	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 130: LL } → *dt*
79 ◁ O	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 131: LO } → *dt*
87 ◁ W	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 132: OW } → *dt*
32 ◁ ␣	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 133: W␣ } → *dt*
89 ◁ Y	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 134: ␣Y } → *dt*
129 ◁ EL	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 135: YE } → *dt*
131 ◁ LO	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 136: ELL } → *dt*
133 ◁ W␣	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 137: LOW } → *dt*
70 ◁ F	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 138: W␣F } → *dt*
136 ◁ ELL	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 139: LOW } → *dt*
132 ◁ OW	77	69	76	76	79	87	32	89	129	131	133	70	136	132	{ 140: ELLO } → *dt*

Рисунок 3.13. LZW-развертывание

Вы можете увидеть процесс развертывания на рисунке 3.13. Сверху дана таблица декодирования, *dt*, в исходном состоянии. Из каждой строки мы считываем кодировку, а затем ищем ее в таблице. Нам надо быть осторожными так, чтобы в процессе мы заполнили таблицу декодирования кодировками встреченных нами *n*-грамм. В первой строчке нам нечего делать, так как у нас пока есть только одна униграмма. Из второй же строчки нам нужно добавить в таблицу развертывания *n*-грамму, которая образовалась из данных нашего предыдущего результата и первого символа нынешнего результата. Таким образом, таблица декодирования будет пополняться нашими входными данными, и, когда, например, мы считываем 129, мы обнаруживаем, что ранее уже добавили его в таблицу развертывания.

Но неужели таблица декодирования всегда так работает с кодировками, которые мы считываем в данный момент, чтобы мы суметь их в ней найти? В нашем примере каждая *n*-грамма, которую мы просматриваем, была внесена в таблицу декодирова-

ния несколькими этапами (рядами) ранее. Так происходит, потому что в обратной ситуации кодирование для соответствующей *n*-граммы было точно так же создано несколькими этапами ранее. Что происходит, когда во время сжатия мы создаем кодировку для *n*-граммы и тут же на следующем этапе выводим результат?

Мы имеем дело с примером тупиковой ситуации. Тупиковая ситуация — это такая чрезвычайная ситуация, которую улаживает алгоритм или компьютерная программа. Всегда полезно испытать наши алгоритмы на тупиковых ситуациях, так как они могут оказаться непростыми. Поэтому нам всегда следует быть настороже, когда мы имеем дело с тупиковыми ситуациями и проверяем нашу работу на наличие ошибок. Например, наша программа работает со множеством значений, но не с наименьшими или наибольшими из них — вот тут в тупиковой ситуации притаилась ошибка.

Как мы уже говорили, в LZW-развертывании тупиковая ситуация случается, когда во время сжатия мы создаем *n*-грамму и тут же выводим результат. Такая ситуация показана на рисунке 3.14(a). Мы сжимаем строку ABABABA. Шифруем *AB*, затем *BA*, а потом *ABA*, чья кодировка как раз и является нашим следующим результатом. Сжатый итог представляет собой список [65, 66, 128, 130].

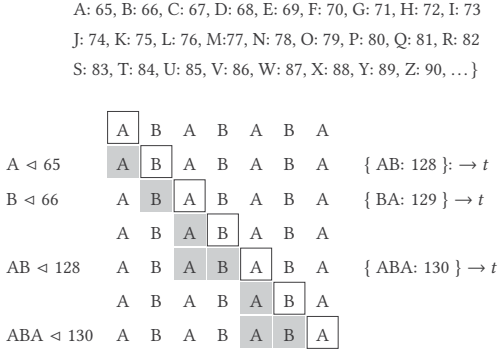
Если нам нужно развернуть список [65, 66, 128, 130], мы начинаем с числа 65, которое, согласно таблице *dt*, равно *A*; посмотрите на рисунок 3.14b. Затем мы берем число 66, которым зашифровано *B*. Мы добавляем в таблицу декодирования биграмму *AB*, ключом которой служит значение 128. Далее берем из списка 128, которое при развертывании становится *AB*. Пока все идет хорошо, хотя мы даже сумели добавить *AB* в таблицу декодирования. Последним берем значение 130; и вот тут все не так радужно, потому что 130 еще не добавлено в таблицу декодирования.

Чтобы уладить проблему, нам нужно вернуться на шаг назад и вспомнить, что такое случается, только когда в процессе кодирования мы что-то шифруем и тут же выводим результат этого шифрования. Предположим, мы прочли строку $x[0],x[1],\ldots,x[k]$, которую мы нашли в таблице декодирования. Затем мы чита-

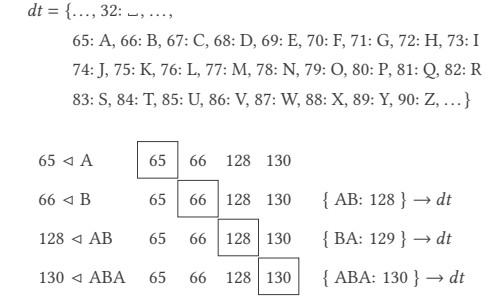
ем $x[k + 1]$ и не можем найти $x[0],x[1],\ldots,x[k],x[k + 1]$ в таблице. Мы сжимаем $x[0],x[1],\ldots,x[k]$, используя ее кодировку, а затем создаем новую кодировку для $x[0],x[1],\ldots,x[k],x[k + 1]$. Если следующее, что мы сжимаем, как раз $x[0],x[1],\ldots,x[k],x[k + 1]$, тогда процесс возможен только, если мы вводимые строки имеют вид:

$$\ldots\ x[0]\ x[1]\ \ldots\ x[k]\ x[k+1]$$
$$x[k] \qquad\qquad x[1]\ \ldots\ x[k]\ x[k+1]\ \ldots$$

Иными словами, $x[0] = x[k +1]$ и новая, только что созданная n-грамма равны предыдущей n-грамме с ее первым символом, добавленным в конец. Что и отображено на рисунке 3.14.



(a) Тупиковый случай при LZW-сжатии



(b) Тупиковый случай при LZW-развертывании

Рисунок 3.14. Тупиковая ситуация при LZW-сжатии и развертывании

Вернемся к развертыванию. Когда нам попадается зашифрованное значение, которого еще нет в нашей таблице декодирова-

ния, мы можем добавить в таблицу новую компоненту, чьим ключом будет последняя *n*-грамма, которую мы добавили, с первым ее символом, добавленным туда. Тогда мы в качестве результата сможем вывести эту новоиспеченную *n*-грамму. В нашем примере, когда мы считываем 130, мы отмечаем, что его нет в таблице декодирования. Мы только что добавили в нее *AB*. Мы прибавляем *A* к *AB* и создаем *ABA*, которую мы добавляем в таблицу декодирования и тут же выводим в виде результата.

Мы используем для этого алгоритм 3.5, дубликат алгоритма 3.4. Алгоритм берет в качестве входных данных список, содержащий зашифрованные значения для сжатых строк, количество битов, которое мы используем для представления элемента, и общее число элементов в алфавите. В строчке 1 мы высчитываем максимальное зашифрованное значение. В строчках 2–4 создаем начальную версию таблицы декодирования. Она похожа на таблицу кодирования, только отображает содержимое задом наперед: от кодировок к строкам. В строчке 5 мы отмечаем, что к данному моменту мы израсходовали *n* кодировок.

В строчке 6 мы создаем пустую строку символов, в которую будем помещать результаты развертывания. Чтобы запустить процесс развертывания сам по себе, мы берем первую кодировку (строчка 7), убираем ее из списка (строчка 8) и ищем в таблице декодирования (строчка 9). Мы совершенно точно найдем ее там, потому что первая кодировка всегда соответствует одному символу алфавита. Мы добавляем расшифрованное значение к результатам развертывания (строчка 10). В оставшейся части алгоритма мы используем переменную *pv*, которая будет хранить последнее значение развертывания; мы инициализируем ее в строчке 11.

Цикл в строчках 12–20 развертывает оставшийся список, то есть весь список за исключением первого элемента, о котором мы уже позаботились. Мы выискиваем каждый элемент списка в таблице декодирования (строчка 13). Если мы находим элемент, то снова добавляем расшифрованное значение в результаты развертывания (строчка 16). Если же элемента в таблице нет (строчка 14), мы имеем дело с тупиковой ситуацией, описанной выше,

следовательно, наше расшифрованное значение точно такое же, как предыдущее расшифрованное значение с первым символом, добавленным в конец (строчка 15); таким образом, в строчке 16, мы добавляем значение к результатам развертывания.

Алгоритм 3.5. LZW-развертывание

`LZWDecompress(`*compressed, nb, n*`)` → *decompressed*

Вводные данные: *compressed*, список представляющий сжатую строку; *nb*, число битов, использованных для представления элемента; *n*, число элементов в алфавите.

Выводимые данные: *decompressed*, исходная строка.

```
1  max_code ← 2^nb − 1
2  dt ← CreateMap()
3  for i ← 0 to n do
4      InsertInMap(dt, i, Char(i))
5  code ← n
6  decompressed ← CreateString()
7  c ← GetNextListNode(compressed, NULL)
8  RemoveListNode(compressed, NULL, c)
9  v ← Lookup(dt, GetData(c))
10 decompressed ← decompressed + v
11 pv ← v
12 foreach c in compressed do
13     v ← Lookup(dt, c)
14     if v = NULL then
15         v ← pv + pv[0]
16     decompressed ← decompressed + v
17     if code ≤ max_code then
18         InsertInMap(dt, code, pv + v[0])
19         code ← code + 1
20     pv ← v
21 return decompressed
```

В строчках 17–19, если в таблице декодирования есть свободное место (строчка 17), мы добавляем в нее новое соответствие (строчка 18) и помечаем, что у нас в таблице стало на одну компоненту больше (строчка 19). Перед итерацией следующего цикла мы сохраняем в *pv* в новое расшифрованное значение (строч-

ка 20). И наконец мы возвращаем сконструированную строку (строчка 21).

LZW-алгоритм можно реализовать с больши́м успехом, так как ему требуется всего один проход через его вводные данные: это однопроходный метод. Он шифрует данные во время считывания и развертывает, проходя через сжатые значения. Операции сжатия и развертывания выполняются очень просто; по сути скорость алгоритма зависит от скорости, с которой мы управляемся с таблицами кодирования и декодирования. Можно использовать словарь, и тогда добавление и просмотр будут выполняться постоянно, за $O(1)$; следственно, сжатию, равно как и развертыванию, требуется линейное время, $O(n)$, где n — это длина вводных данных.

Примечания

Историю и развитие ASCII и схожих с ним кодов вплоть до 1980 года вы можете узнать из доклада Маккензи [132]. Сжатие данных плотно вплетено в нашу повседневную жизнь, и существует множество источников, которые следует изучить, если вы желаете познакомиться с темой сжатия поближе; посмотрите книги Саломона [170], Саломона и Мотты [171], а также книгу Сайеда [172]. Ранние исследования описаны в работе Лельюер и Хиршберга [128].

О развитии азбуки Морзе хорошо рассказано в истории Рассела В. Берна [32, с. 68]. Частота употребления английских букв, показанная в таблице 3.3, рассчитана Питером Норвигом, ее можно посмотреть по ссылке http://norvig.com/mayzner.html. Сведение в таблицы частоты употребления букв — само по себе интересное занятие; мы рассмотрим его в четвертой главе.

В 1951 году Дэвид Хаффман был студентом-электротехником, он учился в магистратуре Массачусетского технологического института и проходил курс теории информации. Студентам предоставили на выбор, либо писать курсовую работу, либо сдавать итоговый экзамен; в курсовой работе требовалось найти наиболее эффективный метод представления чисел, текста и прочих

символов с использованием двоичного кода. Код, который описал Хаффман в своей курсовой, и стал его великим изобретением. Его история живо и интересно описана в журнале «В мире науки» [192]. Код Хаффмана широко применим, благодаря его простоте и факту, что Хаффман никогда не пытался запатентовать свое открытие. Первая статья Хаффмана, касающаяся его изобретения, была опубликована в 1952 году [99].

Абрахам Лемпель и Якоб Зив опубликовали алгоритм LZ77, также известный как LZ1, в 1977 году [228], а алгоритм сжатия LZ78, также известный как LZ2, в 1978 году [299]. Терри Уэлч в 1984 году улучшил LZ78 [215], подарив нам LZW. В самых первых работах LZW для данных использовались 8-битные символы (вместо 7-битных ASCII, которые мы описывали) и их шифровали 12-битными кодами. Алгоритм LZW был запатентован, но срок действия патентов уже истек. Но еще до этого компания «Юнисис», которой принадлежал патент на LZW, пыталась взимать плату за использование GIF-изображений, в которых использовано LZW-сжатие, но такое поведение вызвало лишь общественное негодование и насмешки.

Упражнения

1. В фильме «Марсианин» астронавт Марк Уотни (сыгранный Мэттом Деймоном) брошен на Марсе. Единственный способ общения с центром управления на Земле — с помощью вращающейся камеры. Тогда в голову Уотни приходит идея. Так как каждый символ ASCII может быть зашифрован двумя шестнадцатеричными символами, он размещает по кругу листы с шестнадцатеричными символами. Таким образом, каждое слово, которое он собирается отослать на Землю, он разбивает на символы ASCII и затем для каждого символа посылает с помощью камеры две картинки: по одной для каждого шестнадцатеричного символа. Уотни по профессии ботаник, так что ему пригодилась бы помощь. Напишите программу, которая в качестве входных

данных берет ASCII-сообщение и выдает ряды углов поворота камеры.

2. Реализуйте минимальную и максимальную очередь с приоритетом, используя массив с условием, описанным в тексте: элемент в позиции 0 будет минимальным (или максимальным), для каждого узла i его дитя слева будет находиться в позиции $2i + 1$, а его дитя справа — в позиции $2i + 2$. Попробуйте сконструировать ваш код так, чтобы его можно было использовать многократно и чтобы в реализациях минимальной и максимальной очереди с приоритетом было как можно больше общего кода.
3. Еще одно условие для реализации очередей с приоритетом с использованием массива: оставьте пустым элемент в позиции 0, таким образом, для каждого узла его дитя слева будет находиться в позиции $2i$, а дитя справа — в позиции $2i + 1$. Выполните предыдущее упражнение, учитывая данное условие.
4. Мы объяснили построение кодов Хаффмана, но не вдавались в подробности, как они работают на компьютере. Прежде чем вызвать алгоритм 3.1, нам надо пройтись по тексту, который мы хотим зашифровать, и рассчитать частоту повторения символов в тексте. После того как алгоритм 3.1 завершен, мы можем создать таблицу, подобную таблице 3.7, которую мы используем для кодирования каждого символа текста с помощью кода Хаффмана. Результат сжатия, представляющий собой, как правило, файл, должен содержать две вещи: созданную нами таблицу и зашифрованный текст. Нам нужна таблица, иначе потом мы не сможем узнать, как расшифровывается текст. Закодированный текст содержит ряд битов, не являющихся символами. Вернувшись к таблице 3.7, в результате вы должны получить не строку *1110* для *V*, а четыре однозначных числа 1, 1, 1, 0. Возможно, это не очень очевидно, так как большинство языков программирования по умолчанию выводят в результате байты, поэтому вам нужно самим объединить биты в байты, которые и будут выводимыми

данными. С помощью всего этого вы можете написать свой собственный шифратор и дешифратор Хаффмана.

5. С помощью шифратора Хаффмана закодируйте объемный кусок английского текста и сравните длину кодировки для каждой буквы с кодировками азбуки Морзе.
6. Напишите программу, которая будет генерировать случайные последовательности символов так, чтобы каждый символ с одинаковой вероятностью появился в выводимых данных. В качестве вводных данных программа будет брать размер выводимых данных. Пропустите выводимые данные программы через шифратор Хаффмана и убедитесь, что с такими вводными данными шифратор Хаффмана будет работать не лучше, чем кодирование с фиксированной длинной.
7. Мы описали LZW-развертывание с помощью таблицы декодирования. В то же время она связывает численные значения со строками, так что мы можем использовать вместо нее массив строк. Перепишите и реализуйте алгоритм 3.5, применяя к таблице декодирования *dt* массив.

4 Секреты

Вы умеете хранить секреты? Скажем, вы хотите кое-что написать, но прочесть вашу запись должен только получатель сообщения и никто больше. В нашей жизни мы сталкиваемся с подобными ситуациями довольно часто, например покупки через Интернет. Вам надо сообщить продавцу данные своей кредитки, и вы хотите, чтобы информация осталась конфиденциальной, только между вами и продавцом, поэтому вам нужна твердая уверенность, что, вмешайся в сделку кто-то лишний, ваша личная информация никуда не утечет.

Чтобы сохранить информацию в тайне, мы используем криптографию. В криптографии мы прибегаем к механизму шифрования, с помощью которого берем исходное сообщение, называемое открытым текстом, и зашифровываем его; иными словами, мы превращаем открытый текст в нечто, что нельзя прочесть и что называется криптограммой. Чтобы прочитать криптограмму, ее надо расшифровать; этот процесс называется дешифрацией. Дешифровать сообщение должен только тот, кому мы доверяем. Если же сообщение расшифровал кто-то другой, можно сказать, что криптографическая защита взломана. Шифрование и дешифрация работают с помощью ключа шифрования — главного инструмента, помогающего нам засекретить и рассекретить информацию, которую мы желаем уберечь.

Шифрование стоит на страже нашей личной жизни. Как сказал Филип Циммерман, основоположник криптографии: «Личная информация принадлежит только вам. Она конфиденциальна и касается только вас. Собираетесь ли вы участвовать в политической кампании, высказываете ли недовольство налогами или же завели тайный роман. А, может, вы рассуждаете о политике с диссидентом, живущим в стране репрессивного режима. О чем бы ни шла речь, вам совсем не хочется, чтобы ваша личная пере-

писка или секретные документы попали в чужие руки. Вы защищаете свое личное пространство — и вы правы. Личная жизнь — наше все, это краеугольный камень американской конституции».

Каждый раз, когда вы вводите пароль, вы используете криптографию. Каждый раз, когда совершаете финансовую транзакцию в Интернете, вы тоже используете криптографию. Если хотите совершить защищенный видео- или аудиозвонок, где под «защитой» подразумевается тот факт, что никто кроме вас и ваших собеседников не услышит разговор, — значит, вам нужна криптография. Все тот же Циммерман очень точно сказал: у вас должна быть возможность «шептать собеседнику на ухо, даже если он находится в тысяче километрах от вас».

Лучше всего начать знакомство с криптографией с самого очевидного способа шифровки сообщений, который знает каждый ребенок; он используется в самых разных вариантах еще с древних времен. Способ заключается в том, чтобы выдумать свой алфавит и заменить каждую букву нашего сообщения буквой вымышленного алфавита. Или еще мы можем заменить каждую букву нашего сообщения другой буквой обычного алфавита. Такой метод шифрования называется подстановочным шифром. Самый известный подстановочный шифр — это так называемый шифр Цезаря, которым, как говорят, пользовался Юлий Цезарь. В этом шифре каждая буква заменяется буквой, находящейся в нескольких позициях правее или левее выбранной для замены буквы. Количество позиций, нужных, чтобы найти букву для замены, является ключом шифра. Если ключ равен 5, тогда А превращается в Е, Б превращается в Ё и так далее, пока Я не превратится в Д. Можете проверить, что открытый текст «Я сижу в кабинете» зашифровывается в открытый текст Д ЦНЛШ Ж ПЁЕНТЙЧЙ. Чтобы расшифровать сообщение, вам всего лишь надо сдвинуть позиции букв на 5 назад. Такие шифры называются шифрами со сдвигом.

4.1. Попробуйте расшифровать

Представьте, что вам дали зашифрованный текст на картинке 4.1. Вам что-нибудь понятно? Сперва это кажется абракадаброй.

Но вы догадываетесь, что за ней скрывается внятный, читабельный текст. Так как прежде мы говорили о шифрах с заменами, вы можете подумать, что и данный текст зашифрован тем же способом: путем замены английских букв буквами из выдуманного алфавита.

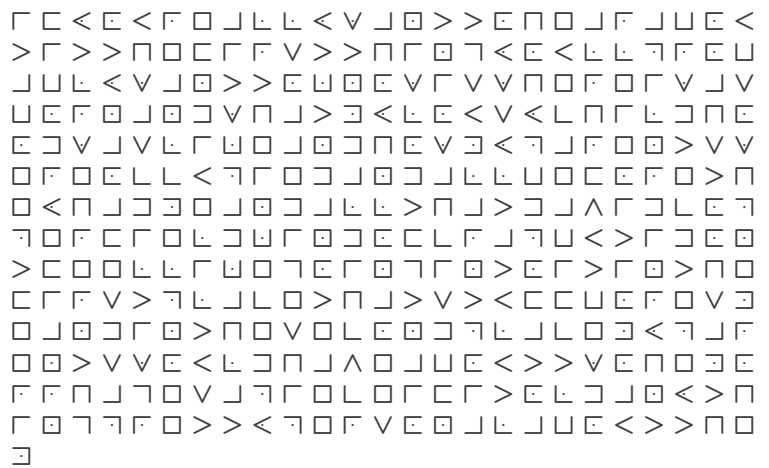


Рисунок 4.1. Задачка на расшифровку

Все языки и тексты подчиняются определенным правилам. Слова состоят из букв и следуют грамматическим правилам; предложения состоят из слов и следуют синтаксическим правилам. Одни слова встречаются чаще, другие — реже. Какие-то буквы используются чаще, какие-то — реже, тоже самое и с буквенными последовательностями (скажем, две или три идут подряд). Если у нас есть большой кусок текста, то мы можем определить частоту употребления буквы в языке. Марк Майзнер в 1965 году опубликовал таблицы с частотой букв. Используя технологии того времени, он сумел обработать корпус из 20 000 слов. 17 декабря 2012 года он связался с директором по исследованиям в корпорации «Гугл», Питером Норвигом, спрашивая у того, могут ли средства «Гугла» помочь обновить его таблицы с частотами букв. Норвиг ответил утвердительно и выполнил запрос, после чего опубликовал результаты. Из них составилась таблица 4.1, полученная после подсчета в базе данных Гугла 3 563 505 777 820 букв.

Число напротив каждой буквы измеряется в миллиардах, а дальше указана употребляемость букв в процентах.

Теперь вернемся к рисунку 4.1. Если вы посчитаете количество зашифрованных в тексте символов, то увидите, что самый распространенный — □, он встречается нам 35 раз. Далее идет > — 33 раза, и ⌟ — 32 раза. Если произведете в зашифрованном тексте замену □ → E, > → T и ⌟ → A, то вы получите три текста, как на рисунке 4.2.

Но все равно непонятно, что там написано. Поэтому нужно совершить еще несколько замен. Следующим по частоте употребления в зашифрованном тексте идет ⊏ — 28 раз, ⌜ — 24 раза и ⊡ — 22 раза. Совершив замены ⊏ → O, ⌜ → I, ⊡ → N, на место встали еще несколько символов. Взгляните на рисунок 4.3(a).

Таблица 4.1. Частота английских букв

E	445.2	12.49%	M	89.5	2.51%
T	330.5	9.28%	F	85.6	2.40%
A	286.5	8.04%	P	76.1	2.14%
O	272.3	7.64%	G	66.6	1.87%
I	269.7	7.57%	W	59.7	1.68%
N	257.8	7.23%	Y	59.3	1.66%
S	232.1	6.51%	B	52.9	1.48%
R	223.8	6.28%	V	37.5	1.05%
H	180.1	5.05%	K	19.3	0.54%
L	145.0	4.07%	X	8.4	0.23%
D	136.0	3.82%	J	5.7	0.16%
C	119.2	3.34%	Q	4.3	0.12%
U	97.3	2.73%	Z	3.2	0.09%

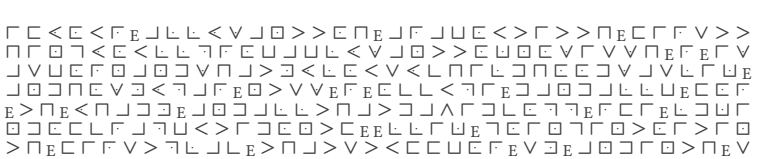
Теперь обратите внимание, что не только одиночные символы повторяются с разной частотой, но и последовательности из двух букв, биграммы, тоже повторяются с разной частотой, как показано в таблице 4.2, предоставленной Норвигом: числа в ней все так же подразумевают миллиарды. Самая распространенная биграмма в английском языке — TH; в нашем зашифрованном тексте самая частая — биграмма, которая повторяется 9 раз, T⊓, так что вполне логично предположить, что ⊓ — это H.

Теперь текст стал понятней; посмотрев на рисунок 4.3(b), вы уже можете прочесть некоторые слова, например THE и THAT.

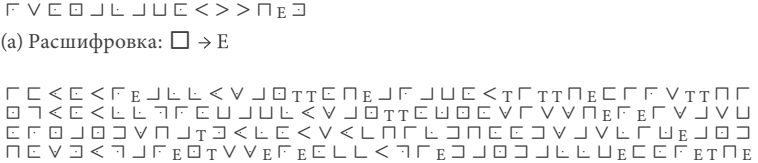
Вторая из самых частых биграмм нашего зашифрованного текста — ⌐E, она встречается 8 раз. В английских текстах очень распространена биграмма RE, так что мы можем попробовать подставить вместо ⌐ букву R.

Таблица 4.2. Десять самых распространенных биграмм

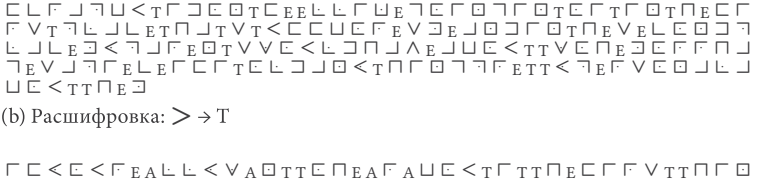
TH	100.3	3.56%
HE	86.7	3.07%
IN	68.6	2.43%
ER	57.8	2.05%
AN	56.0	1.99%
RE	52.3	1.85%
ON	49.6	1.76%
AT	41.9	1.49%
EN	41.0	1.45%
ND	38.1	1.35%



(a) Расшифровка: □ → E



(b) Расшифровка: > → T



(c) Расшифровка: ⌟ → A

Рисунок 4.2. Загадка на расшифровку

Теперь мы смотрим на рисунок 4.4(a), где в первой строчке вы можете увидеть последовательность REA∟∟<. Может быть, это слово REALLY? Совершив необходимую замену, мы оказываемся у рисунка 4.4(b), и, кажется, мы пошли в нужном направлении, так как теперь мы можем угадать еще больше букв. В первых двух строчках нам дважды попадается YO<, что скорее всего означает YOU. В последней строчке мы видим слово ANYTHIN┐, которое, вероятно, значит ANYTHING. Затем еще ·┐ — видимо, это PRETTY. Путем таких вот догадок мы переходим к рисунку 4.4.

Мы продолжаем угадывать буквы, чем больше слов мы расшифровываем, тем проще становится процесс, таким образом мы доходим до рисунка 4.5, где мы видим исходный открытый текст. Он может показаться странным, но это все из-за отсутствия знаков препинания и заглавных букв: мы убрали их, чтобы не усложнять объяснение. Добавив в текст нужные знаки препинания, мы получим:

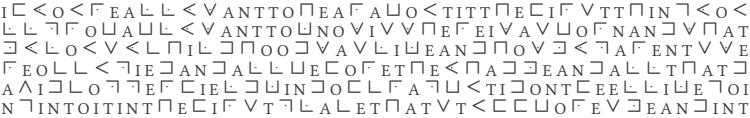
> “If you really want to hear about it, the first thing you’ll probably want to know is where I was born, and what my lousy childhood was like, and how my parents were occupied and all before they had me, and all that David Copperfield kind of crap, but I don’t feel like going into it. In the first place, that stuff bores me, and in the second place, my parents would have about two hemorrhages apiece if I told anything pretty personal about them”.

> «Если вам на самом деле хочется услышать эту историю, вы, наверно, прежде всего захотите узнать, где я родился, как провел свое дурацкое детство, что делали мои родители до моего рождения, — словом, всю эту давидкопперфилдовскую муть. Но, по правде говоря, мне неохота в этом копаться. Во-первых, скучно, а во-вторых, у моих предков, наверно, случилось бы по два инфаркта на брата, если б я стал болтать про их личные дела»*.

Перед нами начало романа Дж. Д. Сэлинджера «Над пропастью во ржи».

---

* Перевод Р. Райт-Ковалевой.

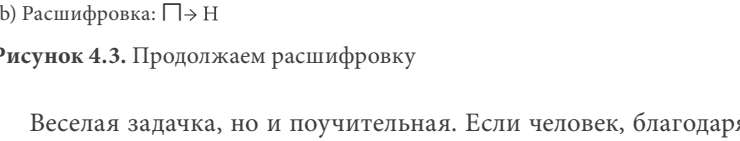


(a) Расшифровка: ⊏ → O, ⌐ → I, ⊡ → N

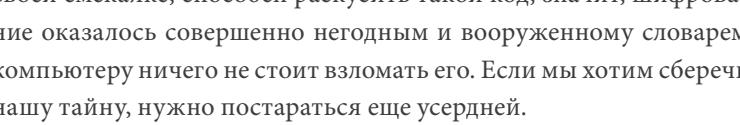
(b) Расшифровка: ⊓ → H

Рисунок 4.3. Продолжаем расшифровку

Веселая задачка, но и поучительная. Если человек, благодаря своей смекалке, способен раскусить такой код, значит, шифрование оказалось совершенно негодным и вооруженному словарем компьютеру ничего не стоит взломать его. Если мы хотим сберечь нашу тайну, нужно постараться еще усердней.



(a) Расшифровка: ⌐ → R



(b) Расшифровка: ⌞ → L, ≺ → Y

Рисунок 4.4. И снова расшифровка

Кстати, символы, использованные для шифрования нашего текста, мы позаимствовали из так называемого масонского шифра, который известен еще с XVIII века. Данный шифр до

сих пор популярен и часто встречается в детских головоломках. Если вам интересно, откуда взялись эти символы — они получились из-за расположения букв алфавита в сетке: буквы очертили линиями и добавили точки. Взгляните на рисунок 4.6: каждая буква замещается окружающими ее линиями и точками.

```
IFYOUREALLYWANTTOHEARABOUTITTHEFIRSTTHINGYOULLPROBABL
YWANTTOKNOWISWHEREIWASBORNANDWHATMYLOUSYCHILDHOOD
WASLIKEANDHOWMYPARENTSWEREOCCUPIEDANDALLBEFORETHEYH
ADMEANDALLTHATDAVIDCOPPERFIELDKINDOFCRAPBUTIDONTFEELL
IKEGOINGINTOITINTHEFIRSTPLACETHATSTUFFBORESMEANDINTHESE
CONDPLACEMYPARENTSWOULDHAVEABOUTTWOHEMORRHAGESAPIE
CEIFITOLDANYTHINGPRETTYPERSONALABOUTTHEM
```

Рисунок 4.5. Задачка решена

Мы смогли расколоть шифр, потому что смогли уловить в зашифрованном тесте закономерности, отражающие общие закономерности, присущие языку. Именно с их помощью и взламываются коды: нашел закономерность — применяешь ее для расшифровки. Так что если мы хотим получить шифр, который нельзя взломать, нам нужно избавиться от любых закономерностей. Иными словами, мы должны зашифровать текст случайным, насколько возможно, образом. Поиск закономерностей в совершенно случайном наборе символов не сработает, так как никаких закономерностей в нем нет, поэтому зашифрованный текст нельзя будет развернуть в исходный, открытый.

4.2. Шифрование одноразовым ключом, или одноразовый блокнот

Существует способ шифрования, который создает закодированный текст случайным, и только он может гарантировать, что ваше секретное послание никто не расшифрует. Мы присваиваем каждой букве алфавита число: A — это 0, B — 1 и так до Z — 25. Мы берем из исходного текста по одной букве за раз. В качестве ключа шифрования мы используем случайную последовательность букв. Мы проходим вместе с открытым текстом по случайному ряду и берем по одной букве за раз. На каждом этапе у нас

имеется одна буква из открытого текста и одна буква из ключа шифрования. Например, у нас есть буква открытого текста W и буква зашифрованного текста G, то есть чи́сла 22 и 6. Мы складываем их: 22 + 6 = 28. Так как Z — это 25, мы переходим к началу алфавита, отсчитываем три буквы и получаем C. Вот этот самый символ и отправляется в зашифрованный текст.

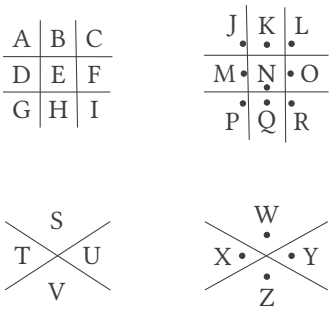


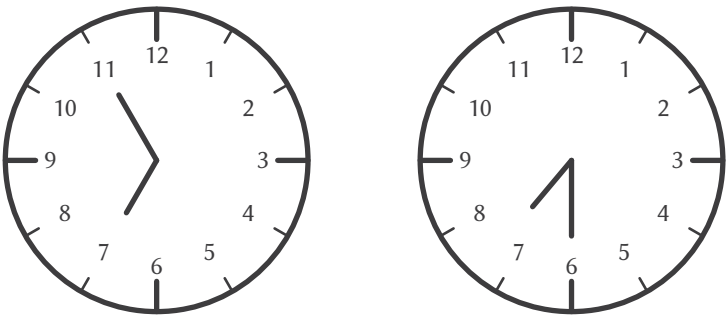
Рисунок 4.6. Масонский шифр

Такое сложение с переходом к началу алфавита часто встречается криптографии и называется сложением по модулю, потому как оно равноценно сложению двух чисел и нахождению остатка после деления полученной суммы. Мы называем это делением по модулю. То же самое мы делаем, когда складываем минуты в час: если мы достигаем 60, мы берем остаток от деления минут на 60, который мы называем минутами по модулю 60; на рисунке 4.7 показан пример. Символ, который используется для обозначения модуля, — mod, таким образом, 23 mod 5 = 3. В нашем примере дано (22 + 6) mod 26 = 28 mod 26 = 2, а числу 2 соответствует литера C.

В математике модуль, x mod y, определяется как число $r \geq 0$, такой остаток, что $x = qy + r$, где q является целой частью числа, полученного при делении x/y, $[x/y]$. Следовательно, мы получаем $r = x - y[x/y]$. Данное определение касается и модуля отрицательного делимого. В самом деле, – 6 mod 10 = 4, потому что [– 6/10] = – 1 и r = – 6 – 10(– 1) = – 6 + 10 = 4.

В шифровании почти все то же самое, только вместо сложения используется вычитание. Мы берем зашифрованный текст и тот

же ключ шифрования, который использовали при шифровке, а затем символ за символом проходимся по ним. Если в зашифрованном тексте у нас C, а в ключе шифрования G, тогда мы получаем (2 – 6) mod 26 = – 4 mod 26 = 22, что и есть буква W.



(a) Начало отсчета (b) Начало отсчета плюс 35 минут

Рисунок 4.7. Добавление минут происходит по модулю 60

Такой метод шифрования называется одноразовым блокнотом, который показан на рисунке 4.8. Данный метод полностью надежен, потому что мы получаем случайно зашифрованный текст. Каждая буква открытого текста, $m[i]$, добавляется по модулю 26, если сообщение состоит лишь из букв английского алфавита, где каждой из них соответствует случайная буква одноразового блокнота $t[i]$, нужная для создания зашифрованного текста, $c[i]$; таким образом, у нас получается $c[i] = (m[i] + t[i]) \bmod 26$. Если $t[i]$ случайный, то и $c[i]$ случайный. Как видите, в зашифрованном тексте нет никаких закономерностей. Одна и та же буква закодирована разными буквами зашифрованного текста, поэтому поиск закономерностей тут бесполезен: он не выявит никаких частот использования букв. Если у нас есть одноразовый блокнот, то расшифровка происходит легко. Если шифрование $c[i] = (m[i] + t[i]) \bmod 26$, то развертывание — $m[i] = (c[i] - t[i]) \bmod 26$. Однако, если у нас нет одноразового блокнота, мы бессильны и никакие угадывания тут не помогут. Хуже того, мы можем угадать неверный одноразовый блокнот, который подходит к расшифровке другого сообщения: каждый символ в одноразовом блокноте случаен, а значит, все одноразовые блокноты разные. Мы можем попробо-

вать ткнуть пальцем в небо и в результате, скорее всего, попадем впросак, как показано на рисунке 4.9. На рисунке 4.9(a) у нас есть зашифрованный текст и нужный одноразовый блокнот, поэтому мы легко получаем открытый текст. На рисунке 4.9(b) у нас схожий с предыдущим, но другой одноразовый блокнот, которым мы пытаемся вскрыть кода, результаты говорят сами за себя.

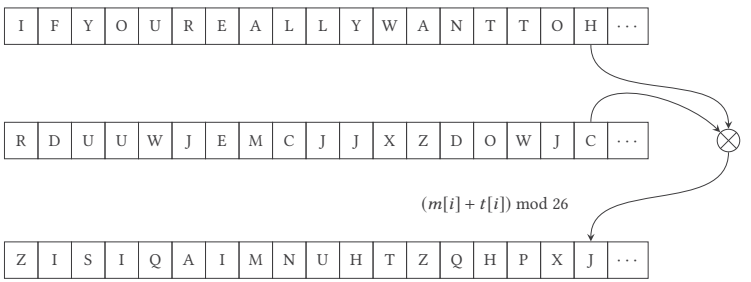


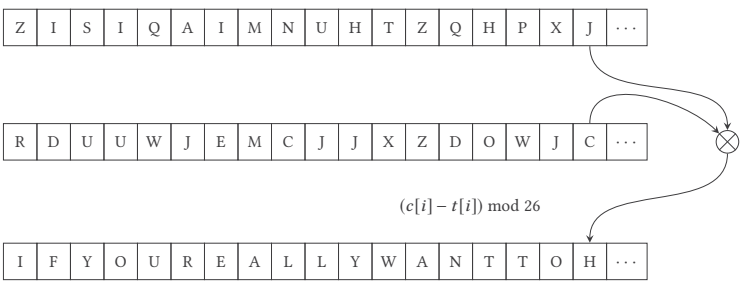
Рисунок 4.8. Одноразовый блокнот

Таблица 4.3. Операция исключающего ИЛИ (`xor`)

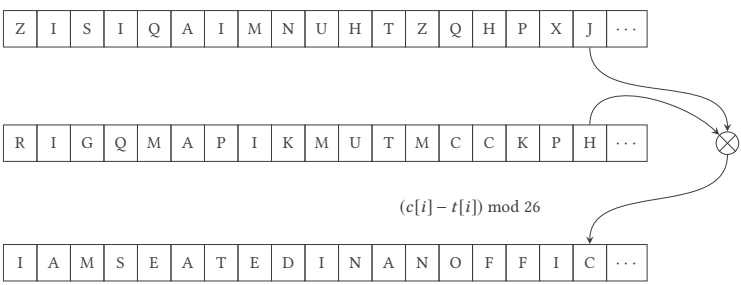
		x	
		0	1
y	0	0	1
	1	1	0

Мы можем упростить операции в одноразовых блокнотах, используя вместо сложения по модулю двоичную операцию, исключающее ИЛИ (`xor`). Операция `xor`, которую обычно обозначают символом $\oplus$, берет в качестве вводных данных двоичные цифры. Если цифры одинаковые, то есть две единицы или два нуля, тогда вывод будет 0. Если они различны, вывод будет 1. Иными словами, у нас есть $1 \oplus 1 = 0$, $0 \oplus 0 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$. Операция «исключающее ИЛИ» приведена в таблице 4.3. Для использования `xor` мы берем открытый текст и представляем его в виде последовательности двоичных цифр. Такое всегда возможно, потому что каждый символ представлен в виде двоичного числа; например, A при использовании ASCII обычно представляется как 1100001. В таком случае одноразовый блокнот — это случайная двоичная последовательность, такая как 1101011… Мы бит за битом при-

меняем `xor` к открытому тексту. В результате получается зашифрованный текст, в нашем случае 0001010. Что любопытно, помимо того, что `xor` — очень простая операция, она еще мгновенно обратимая. Если $c = a \oplus b$, то $c \oplus b = a$. Для расшифровки мы снова применяем `xor` к зашифрованному тексту с одноразовым блокнотом. Вы можете проверить, что 0001010 ⊕ 1101011 = 1100001. Работа с `xor` более общая, чем работа с модульной операцией, потому что `xor` будет иметь дело с двоичными строками, а не с кодированием одной конкретной буквы. К тому же операция `xor` выполняется очень быстро.



(a) Верная расшифровка



(b) Ложная расшифровка

Рисунок 4.9. Одноразовый блокнот, верная и ложная расшифровки

К сожалению, одноразовые блокноты не практичны. Случайный набор букв всегда должен быть совершенно случайным. Но ведь довольно сложно создать по-настоящему случайные наборы букв в больших количествах. Существуют машинные методы, которые должны создавать случайные последовательности, но на деле все эти последовательности псевдослучайные. Подумайте,

разве можно создать беспорядок, следуя строго прописанной процедуре? А именно такие процедуры и выполняет компьютер. Вам же нужен настоящий хаос, который не вычислишь и не предскажешь никакими методами. Именно этой теме будет посвящен раздел 16.1.

К тому же одноразовый блокнот можно использовать лишь единожды. Если мы станем повторять последовательность, то после прохода по всей длине последовательности столкнемся с одинаковой заменой: тогда шифрование ухудшится и станет напоминать шифр со сдвигом. К тому же одноразовый блокнот должен быть не короче самого сообщения, иначе мы, опять-таки, столкнемся с повторными заменами.

Из-за всех этих недостатков одноразовые блокноты не применяются на практике, за исключением особых случаев. Как правило, длинные случайные последовательности сохраняются на запоминающем носителе и отсылаются получателю. Отправитель и получатель применяют эти случайные последовательности, пока не дойдут до конца сохраненной последовательности. Затем им понадобится новая. Таким образом, организация работы выходит слишком сложной и запутанной: огромные объемы случайных последовательностей пересылаются туда-сюда, плюс надо не забывать о постоянной их генерации.

4.3. AES-шифрование

Современная криптография создает зашифрованные тексты, используя особые математические методы. Эти методы применяют относительно небольшой ключ: длиной в сотни или тысячи битов. Они берут открытый текст и ключ, а затем трансформируют текст сложными процессами, которые можно обратить вспять лишь имея ключ шифрования. Мы опишем метод, AES*, который используется повсеместно. Каждый раз, когда вы передаете зашифрованные данные через браузер, вы используете AES.

---

* Advanced Encryption Standard — усовершенствованный стандарт шифрования.

Это стандарт, принятый в 2001 году американским Национальным институтом стандартов и технологий (НИСТ) в результате открытого конкурса по замене устаревшего стандарта DES*. Конкурс длился три года, с 1997 по 2000. НИСТ попросил криптографическое сообщество предоставить варианты замены; после серьезных исследований, 2 октября 2000 года НИСТ объявил, что на замену выбран алгоритм двух бельгийских криптографов, Йоана Даймена и Винсента Рэймена, названный Рэндалом.

AES — сложный алгоритм. Взглянув на него, новички хватаются за голову, но в этом нет ничего постыдного. Читатель не обязан запоминать каждый этап AES. Но вместе мы ждем, что читатель оценит усилие, которое нужно приложить, чтобы разобраться в том, как строится надежный шифр и что нужно знать о шифре в эпоху компьютерных технологий, чтобы защититься от компьютерной расшифровки. Таким образом читатель узнает правду о разных волшебных технологиях и инструментах, которые отвечают за безопасность личных данных и которые широко рекламируются как самые полезные, однако не выдерживают никакой серьезной критики криптографов и программистов. Так что, милый читатель, приготовься к встряске.

AES выполняет серию операций с открытым текстом. Сначала мы разбиваем открытый текст на блоки по 128 битов, или 16 байт. AES работает с блоками битов, отсюда и название — блочным шифр. В отличие от него поточный шифр работает с заданным количеством байтов или битов. Он использует ключи длиной 128, 192 или 256 битов. Байты помещаются в матрицу столбцами, то есть мы постепенно заполняем матрицу столбцами, один за другим. Такая матрица называется матрицей состояний. Если блок состоит из байтов p_0, p_1,…, p_{15}, тогда байт p_i добавляется в матрицу состояний как байт $b_{j,k}$, где $j = i \bmod 4$ и $k = i/4$. Данная трансформация изображена на рисунке 4.10; мы полагаем, что она реализована с помощью операции `CreateState`, которая берет в качестве вводных данных блок b и возвращает матрицу состояний.

* Data Encryption Standard — стандарт шифрования данных.

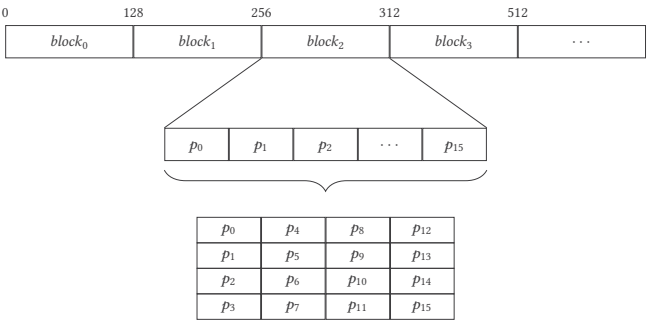


Рисунок 4.10. AES-операция `CreateState`

Затем мы берем ключ шифрования и используем его для извлечения ряда дополнительных ключей из байтов, упорядоченных в столбцах, похожих на определенное состояние. Такие ключи называется цикловыми ключами, потому что, как мы увидим дальше, в каждом цикле, или итерации, базового AES-алгоритма используется по одному ключу. На самом деле ключей, которые нам понадобятся, будет не столько же, сколько циклов, а на один больше, потому что перед началом всех итераций нам тоже нужен ключ. Генерация дополнительных ключей называется расширением ключа, `KeyExpansion`. Идея изменения ключа затем, чтобы шифрование стало более устойчивым к определенным видам атак, называется отбеливанием ключа. Мы не будем дальше углубляться в `KeyExpansion`; он сам по себе небольшой и заковыристый алгоритм.

Представим, что мы сгенерировали ключ и берем первый цикловой ключ. Выполняем `xor` для каждого байта циклового ключа с соответствующим байтом состояния. Такая операция называется `AddRoundKey`. Мы получаем новое состояние с элементами $x_{i,j} = p_{i,j} \oplus k_{i,j}$, показанное на рисунке 4.11.

Теперь мы выполняем серию циклов, чтобы получить результат операции `AddRoundKey`. Количество итераций зависит от длины ключа. 10 итераций для 128-битных ключей, 12 итераций для 192-битных ключей и 14 — для 256-битных ключей.

Первая операция для каждого цикла называется `SubBytes`, она берет нынешнее состояние и заменяет каждый его байт бай-

том из другой матрицы, зовущейся блоком подстановки. Блок подстановки — это матрица 16 × 16, чье содержимое вычисляется при помощи функций с особыми криптографическими свойствами. Такая матрица изображена в виде таблицы 4.4. Если матричный элемент $x_{i,j}$ равен числу X, то из-за того, что X — это байт, оно может быть представлено в виде двух шестнадцатеричных цифр, $h_1 h_2$. Результатом операции `SubBytes` для $x_{i,j}$ будет элемент (h_1, h_2) из блока подстановки, или $s_{h1,h2.}$. Данная процедура показана рисунке 4.12.

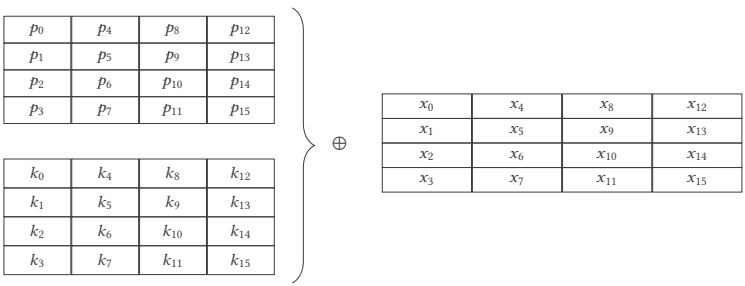


Рисунок 4.11. Опервация `AddRoundKey`

Таблица 4.4. Блок подстановки AES

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Все кажется гораздо сложнее и запутанней, чем есть на самом деле. Предположим, что у нас на рисунке 4.12 x_4 = 168; 168 в десятичном виде — это A8 в шестнадцатеричном. Мы переходим к ряду A и столбцу 8, изображенным в таблице 4.4, и находим

число C2 в шестнадцатеричном виде, или 194 в десятичном. Это значит, что $sb_4 = 194$. Мы проделываем то же самое для всех чисел матрицы.

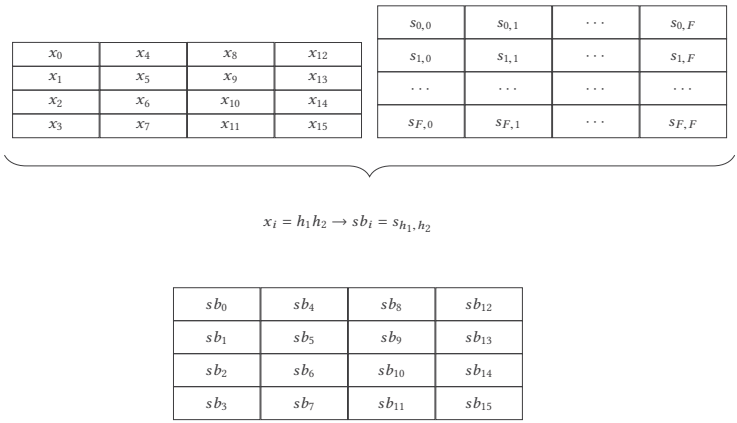


Рисунок 4.12. Операция `SubBytes`

Вторая операция для каждой итерации, следующая за операцией `SubBytes`, заключается в сдвиге рядов полученного состояния. Данная операция называется `ShiftRows`. Она сдвигает каждый ряд состояния влево, прочь от первого, путем увеличения числа позиций. Ряд 2 сдвинут влево на одну позицию, ряд 3 сдвинут влево на две позиции, и ряд 4 сдвинут влево на три позиции, в случае необходимости, перестановка идет по кругу, как показано на рисунке 4.13.

Третья операция для каждого цикла называется `MixColumns`, и тут мы работаем со столбцами, забирая каждый столбец нынешнего состояния и превращая его в новый столбец. Такая трансформация осуществляется путем умножения каждого столбца фиксированной матрицей, как на рисунке 4.14, на котором показана операция для второго столбца; для всех столбцов матрица одна и та же. Вы можете обратить внимание, что вместо привычной нотации для сложения и умножения мы используем $\oplus$ и $\bullet$ соответственно. Все потому, что мы выполняем не обычные арифметические операции. Сложение и умножение над многочленами выполняются по модулю; модулем служит неприводимый

многочлен в степени 8 в конечном поле GF(2^8). Муторная фраза. К счастью, для понимания работы AES вам не обязательно знать, что она означает. Сложение — это простая побитовая операция `xor`. С умножением посложнее, но в целом почти так же просто.

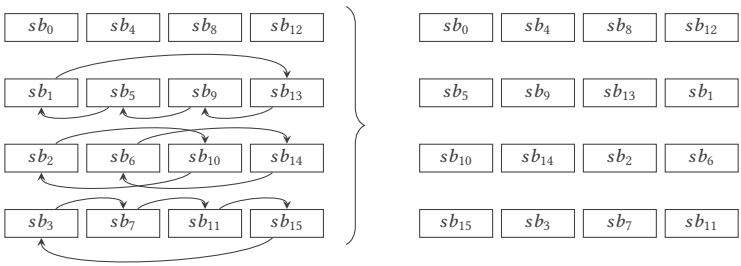


Рисунок 4.13. AES ShiftRows operation

И правда, оказывается, что, хотя базовая теория сложна для неискушенных читателей, на деле операция умножения довольно незатейлива. Если a обозначает одно из значений sb_i, то, как видно на рисунке 4.14, нам нужно лишь определить умножение на 1, 2 или 3. Мы наблюдаем следующее:

$1 \bullet a = a$

$3 \bullet a = 2 \bullet a \oplus a$

Это означает, что нам надо лишь знать, как посчитать $2 \bullet a$. Мы рассматриваем двоичное представление a, в котором $a = (a_7, a_6, \ldots, a_0)$, где каждое из a_i представляет собой отдельный бит. Тогда у нас получается:

$$2 \bullet a = \begin{cases} (a_6, \ldots, a_0, 0) & \text{if } a_7 = 0 \\ (a_6, \ldots, a_0, 0) \oplus (0, 0, 0, 1, 1, 0, 1, 1) & \text{if } a_7 = 1 \end{cases}$$

Для тех, кто реализует AES, вся операция `MixColumns` может быть написана несколькими строчками оптимизированного программного кода. Хоть описание `MixColumns` довольно сложное, ее реализация вовсе не такая громоздкая и запутанная.

Последняя операция в цикле — повторное добавление циклического ключа в состояние, то есть выполнение операции `AddRoundKey` над состоянием, какое оно есть в данный момент.

Мы выполняем одинаковые серии операций для всех итераций за исключением последней, в которой мы выполняем операцию `MixColumns`. В общем, AES работает, как показано в алгоритме 4.1. В строчке 1 мы создаем состояние, затем, в строчке 2, генерируем ключи. Ключи хранятся в массиве *rk* размером $n + 1$, где n — это количество циклов. В строчке 3 добавляем в состояние первый цикловой ключ. Строчки 4–8 отвечают за первые $n - 1$ циклов, в то время как строчки 9–11 отвечают за последний цикл.

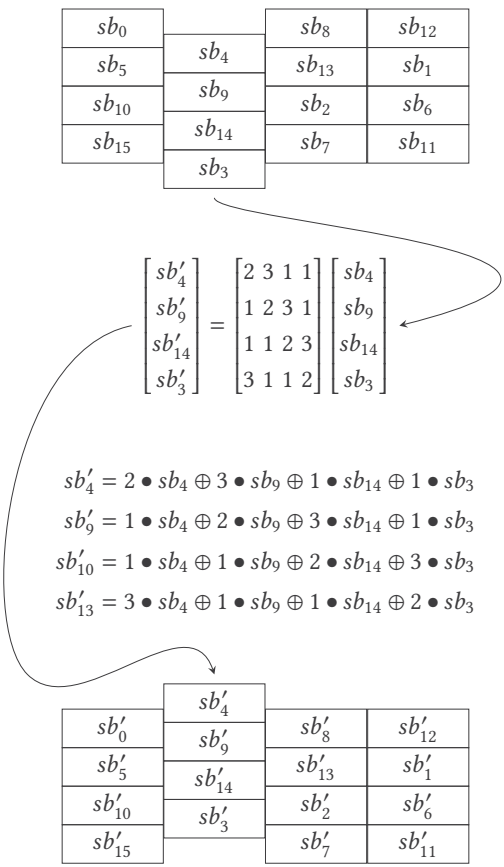


Рисунок 4.14. Операция `MixColumns`

Конечно же, всякий алгоритм шифрования абсолютно бесполезен, если к нему нет алгоритма расшифровки. Оказывается,

что расшифровка в AES довольно проста. Все этапы, описанные в алгоритме 4.1, работают в обратную сторону, с условием, что в операции `SubBytes` мы используем особый, обратный блок подстановки, который отображен в таблице 4.5. Обратный алгоритму AES-шифрования — алгоритм 4.2. Операции носят префикс `Inv`, чтобы показать, что они являются простыми вариантами операций, использованных при шифровании. Если не брать в расчет некоторые изменения в порядке выполнения операций, логика алгоритма точно такая же; обратите внимание, что цикловые ключи здесь применяются в обратной последовательности.

Алгоритм 4.1. Алгоритм AES-шифрования

`AESCipher`(b, k, n) $\rightarrow s$

Вводные данные: b, блок из 16 байтов; k, ключ шифрования; n, число итераций.

Выводимые данные: s, зашифрованный текст, соответствующий b.

```
1  s ← CreateState(b)
2  rk ← ExpandKey(k)
3  s ← AddRoundKey(s, rk[0])
4  for i ← 1 to n do
5      s ← SubBytes(s)
6      s ← ShiftRows(s)
7      s ← MixColumns(s)
8      s ← AddRoundKey(s, rk[i])
9  s ← SubBytes(s)
10 s ← ShiftRows(s)
11 s ← AddRoundKey(s, rk[n])
12 return s
```

В общем, если вам нужно надежное шифрование, вы со спокойной душой можете использовать AES. Берете ключ шифрования, скармливаете его реализованному алгоритму и посылаете полученный в результате текст адресату. Он использует точно такой же ключ, развернет AES-шифровку и прочтет ваше сообщение.

AES используется уже много лет, и пока никто не смог взломать его. Иными словами, еще никто не придумал способ, как

извлечь открытый текст из зашифрованного, не прибегая к подходящему ключу шифрования. AES является примером симметричного шифра; для шифрования и расшифровки требуется один и тот же ключ. Когда же для шифрования нужен один ключ, а для дешифрования — другой, речь идет об асимметричном шифре.

Алгоритм 4.2. Алгоритм AES-расшифровки

`AESDecipher`(b, k, n) $\rightarrow s$

Вводные данные: b, блок из 16 байтов; k, ключ шифрования; n, число итераций.

Выводимые данные: s, открытый текст, соответствующий b.

```
1  s ← CreateState(b)
2  rk ← ExpandKey(k)
3  s ← AddRoundKey(s, rk[n])
4  for i ← 1 to n do
5      s ← InvShiftRows(s)
6      s ← InvSubBytes(s)
7      s ← AddRoundKey(s, rk[n − i])
8      s ← InvMixColumns(s)
9  s ← InvShiftRows(s)
10 s ← InvSubBytes(s)
11 s ← AddRoundKey(s, rk[0])
12 return s
```

Вся безопасность строится на защитном ключе. Это значит, что сохранность вашей личной информации зависит от надежности вашего ключа. Как только он попадет в чужие руки, AES станет бессилен. Это не изъян AES, каждый шифр использует тот или иной ключ. Даже больше: то, что надежность AES и других хороших шифров базируется исключительно на конфиденциальности ключа, — это не минус, а только плюс. В 1883 году голландский лингвист и криптограф Огюст Керкгоффс, преподававший языковедение в парижской школе предпринимательства, доказывал, что принцип работы шифрования не должен быть тайной, и ничего страшного, если он окажется в руках врага. Сегодня мы говорим то же самое: надежность любого шифро-

вания должна базироваться на его ключе, а не самом принципе шифрования.

Таблица 4.5. Обратный блок подстановки AES

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Перед нами четкое правило конструирования и предостережение против подхода «безопасность путем сокрытия», суть которого в том, что, если враг не знает, как работает вся система, он не способен ее взломать. Но это не так; если вы надеетесь, что такой подход защитит ваши тайны, вам стоит задуматься о том, что противник может обзавестись сообразительными подручными, которые рано или поздно раскусят рабочий алгоритм вашей системы. Или же враг просто подкупит кого-то, кто его уже знает. Как бы там ни было, защита, в основе которой лежит ключ, способна гарантировать конфиденциальность, покуда ключ не попал в посторонние руки, к тому же куда проще держать в тайне ключ, а не весь рабочий принцип системы.

Помимо AES существуют и другие симметричные шифры, но в них точно так же все строится вокруг ключа, который надо держать в секрете, и так же требуется соглашение обеих сторон коммуникации на использование общего ключа. И тут есть одна загвоздка. Если вы с партнером или собеседником находитесь недалеко друг от друга, то все просто: вам надо встретиться и обменяться ключами. Но если вы друг от друга далеко, вы не можете записать ключ на бумажке и послать его второй стороне. Сам ключ никак не зашифрован, и, если его перехватят на пути к получателю, всей вашей конфиденциальности конец.

4.4. Обмен ключами по методу Диффи — Хеллмана

Решение задачи с обменом ключей — вот что гарантирует безопасную передачу цифровой информации. На первый взгляд может показаться, что все работает не иначе как по волшебству. Две стороны, Алиса и Боб (в криптографии имена сторон обычно берутся в алфавитном порядке), хотят обменяться ключами, чтобы суметь зашифровать и расшифровать сообщение. Однако прежде они обмениваются несколькими другими сообщениями — сообщениями открытого текста, в которых нет никакого ключа, нужного обеим сторонам. Но, когда обмен сообщениями завершен, у Алисы и Боба есть общий ключ. Никто никому не посылал ключ, и, следовательно, его никто не перехватил.

Давайте посмотрим, как такое возможно. Алиса и Боб выполняют ряд этапов. Сначала они договариваются о двух числах. Простое число, p, и еще одно число, не обязательно простое, g, такие, что $2 \leq g \leq p - 2$ (если вам кажется, что ограничения не имеют смысла, потерпите немного, очень скоро смысл появится). Так как все расчеты, которые они будут выполнять, сделаны по модулю p, p является модулем системы; g называется основанием. Допустим, Алиса и Боб выбрали $p = 23$ и $g = 14$. Им не нужно хранить эти числа в секрете; они могут прилюдно оговорить их и где-нибудь опубликовать.

Затем Алиса выбирает секретное число a, $1 \leq a \leq p - 1$. Допустим, она выбрала $a = 3$. Она делает расчет:

$$A = g^a \bmod p = 14^3 \bmod 23 = 2744 \bmod 23 = 7$$

Так как расчеты выполняется по модулю p, Алисе не имеет смысла выбирать $a \geq p$. Алиса посылает Бобу число A, то есть 7. Теперь Боб выбирает секретное число b, опять же $1 \leq b \leq p - 1$. Допустим, он выбрал $b = 4$. Он выполняет с секретным числом те же операции, что и Алиса. Иными словами, он делает расчет:

$$B = g^b \bmod p = 14^4 \bmod 23 = 38\,416 \bmod 23 = 6$$

Боб посылает Алисе число B, то есть 6. Алиса высчитывает:

$$B^a \bmod p = 6^3 \bmod 23 = 216 \bmod 23 = 9$$

Боб высчитывает:

$$A^b \bmod p = 7^4 \bmod 23 = 2401 \bmod 23 = 9$$

Вот оно: секретное число Алисы и Боба — 9. Обратите внимание, они никогда не обменивались им, но, тем не менее, оба нашли его в результате расчетов. Более того, если кто-то вдруг перехватит их переписку, он никак не сможет рассчитать их секретное число: если только в теории. То есть врагу никак не узнать значения p, g, A и B. Вы можете посмотреть на рисунок 4.15 и убедиться, что Алиса и Боб не посылали друг другу секретное число.

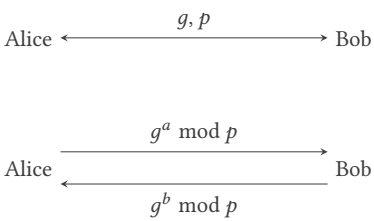


Рисунок 4.15. Обмен информацией по методу Диффи — Хеллмана

Такой способ для обмена ключами называется обменом ключей по методу Диффи — Хеллмана, в честь Уитфилда Диффи и Мартина Хеллмана, опубликовавших его в 1976 году. Данный метод был разработан несколькими годами ранее Малколмом Уильямсоном, работавшим в Центре правительственной связи Великобритании, спецслужбе, ответственной за разведку средств связи, но его открытие было засекречено, поэтому о нем никто не знал. Таблица 4.6 показывает обмен ключами по методу Диффи — Хеллмана. Вы можете увидеть, почему он работает: Алиса и Боб высчитывали одно и то же число, так как $g^{ba} \bmod p = g^{ab} \bmod p$. Чтобы прийти к этому, нам надо знать, что, согласно основным законам арифметики, по модулю мы имеем:

$$(u \bmod n)(v \bmod n) \bmod n = uv \bmod n$$

из чего следует:

$$(u \bmod n)^k \bmod n = u^k \bmod n$$

Таким образом:

$$(g^b \bmod p)^a \bmod p = g^{ba} \bmod p$$

и

$$(g^a \bmod p)^b \bmod p = g^{ab} \bmod p$$

Обмен ключей по методу Диффи — Хеллмана надежен до тех пор, пока Алиса и Боб хранят a и b в секрете, а им совершенно незачем разбалтывать свои тайны. На самом деле, после обмена они даже могут выкинуть эти данные об a и b, так как они им больше не нужны.

В основе безопасности данного метода лежит сложность, связанная со следующей задачей. Если у нас есть простое число p, число g и $y = g^x \bmod p$, задача дискретного логарифма состоит в том, чтобы найти целое число x в данном равенстве, где $1 \le x \le p - 1$. Целое число x называется дискретным логарифмом y по основанию g, и мы можем записать $x = \log_g y \bmod p$. Задача вызывает сложность, потому что $y = g^x \bmod p$ — пример односторонней функции. Без проблем можно вычислить y, имея g, x и p (скоро мы увидим эффективный расчетный метод), но у нас нет эффективного метода для вычисления x при имеющихся y, g и p. Все, что мы можем, — это подставлять различные значения x до тех пор, пока не найдем верное.

Таблица 4.6. Обмен ключом по методу Диффи — Хеллмана

Alice	Bob
Alice and Bob agree on p and g	
Choose a Calculate $A = g^a \bmod p$ Send A to Bob	Choose b Calculate $B = g^b \bmod p$ Send B to Alice
Calculate $s = B^a \bmod p$ $= (g^b \bmod p)^a \bmod p$ $= g^{ba} \bmod p$	Calculate $s = A^b \bmod p$ $= (g^a \bmod p)^b \bmod p$ $= g^{ab} \bmod p$

Таблица 4.7. Возведение в степень и взятие остатка при $g = 2$, $p = 13$

x	1	2	3	4	5	6	7	8	9	10	11	12
g^x	2	4	8	16	32	64	128	256	512	1024	2048	4096
$g^x \bmod p$	2	4	8	3	6	12	11	9	5	10	7	1

И правда, хотя поведение показательной функции предсказуемо при создании увеличивающихся значений для увеличивающихся степеней числа, ее поведение при возведении в степень по модулю целого числа совершенно непредсказуемо; взгляните на таблицу 4.7. Вы с легкостью можете получить x из g^x, взяв логарифм. Но вы не можете получить x из $g^x \bmod p$ путем взятия логарифма или же с помощью каких-либо других известных формул.

Из таблицы 4.7 мы видим, что обычные степени двух получаются путем последовательного удваивания, в то время как степени двух по модулю 13 дают нам различные значения от 1 до 12 включительно, без какой-либо закономерности. Далее они начинают повторяться и идут по кругу. Действительно, из равенства $2^{12} \bmod 13 = 1$ мы получаем $2^{13} \bmod 13 = (2^{12} \times 2) \bmod 13 = ((2^{12} \bmod 13) \times (2 \bmod 13)) \bmod 13 = (1 \times 2) \bmod 13 = 2$; мы снова используем правило вычисления по модулю, которое позволяет выполнить наше деление по модулю. В целом вы можете видеть, что функция $2^x \bmod 13$ периодична, ее период равен 12, так как $2^{12+k} \bmod 13 = ((2^{12} \bmod 13) \times (2^k \bmod 13)) \bmod 13 = (1 \times 2^k) \bmod 13 = 2^k \bmod 13$. Более того, 12 является базовым периодом; меньше него периода нет.

Таблица 4.8. Возведение в степень и взятие остатка при $g = 3$, $p = 13$

x	1	2	3	4	5	6	7	8	9	10	11
g^x	3	9	27	81	243	729	2187	6561	19683	59049	177147
$g^x \bmod p$	3	9	1	3	9	1	3	9	1	3	9

Обратите внимание, что так бывает не всегда. Таблица 4.8 описывает ситуацию с последовательными степенями трех по модулю 13. В этот раз модульные степени не проходят через все значения от 1 до 12, а только через некоторые из них; основной период для $3^x \bmod 13$ будет 3. При такой подборке для g и p нам, чтобы решить задачку дискретного алгоритма, нужно попробовать подставить всего 3 различных значения вместо всех 12 возможных.

Если последовательные значения $g^x \bmod p$ покрывают все числа от 1 до $p - 1$ включительно, тогда мы говорим, что g — это генератор или примитивный элемент. Если точнее, его называют

групповым генератором или групповым примитивным элементом, потому что числа 1, 2,…, $p - 1$, где p является простым числом, формируют мультипликативную группу, когда мы умножаем их по модулю p; это основной принцип в алгебре и теории чисел. Следовательно, нам надо выбрать генератор в виде g. На самом деле мы можем обойтись и без генератора, если последовательные значения $g^x \bmod p$ являются достаточно большой выборкой чисел 1, 2,…, $p - 1$, поэтому поиск решения для задачи дискретного логарифма невозможен.

Значения начнут повторяться, как только степень $g^x \bmod p$ станет 1. Если у нас $g = p - 1$, то $g^1 \bmod p = (p - 1) \bmod p = p - 1$; $g^2 \bmod p = (p - 1)^2 \bmod p = 1$, так как $(p - 1)^2 = p(p - 2) + 1$, поэтому все степени будут чередоваться между $p - 1$ и 1. Если у нас $g = 1$, тогда все степени будут равны 1. Именно поэтому мы требуем, чтобы Алиса и Боб выбрали g с условием $2 \leq g \leq p - 2$.

Вернемся к методу Диффу — Хеллмана. Чтобы убедиться, что никто не вторгнется в процесс обмена ключом, нам нужно убедиться, что из $g^x \bmod p$ нельзя узнать x. Это означает, что p должно быть очень большим. Мы можем задумать p простым числом с двоичным представлением в 4096 битов, которое в десятичном виде имеет 1233 знака. Есть прекрасные способы, позволяющие подобрать такие простые числа, так что мы не будем искать вслепую. Мы рассмотрим распространенный метод нахождения простых чисел в разделе 16.4. Также нам надо выбрать подходящее g. Чтобы получить генератор (или что-то близкое к генератору), не обязательно выбирать для g большое, как у p, значение; оно может быть совсем маленьким, например 2. После всех приготовлений мы начинаем обмен ключами.

В итоге получается, что, если Алиса и Боб хотят общаться без лишних свидетелей, они, используя метод Диффи — Хаффмана, сперва обзаводятся общим секретным ключом, который известен только им двоим. Затем они используют этот ключ, чтобы зашифровать с помощью AES свои сообщения. Когда процесс закончен, Алиса и Боб избавятся от секретного ключа, так как вся последовательность может быть выполнена повторно в любой желаемый момент.

Тут есть одна маленькая оговорка. Все, о чем мы сейчас говорили, верно для компьютеров какими мы их знаем. Алгоритм, разработанный для квантовых компьютеров, решает задачу дискретного логарифма за полиноминальное время. Если квантовые компьютеры войдут в нашу жизнь, метод Диффи — Хеллмана, как и всякие другие криптографические методы, окажутся бесполезны. Исследователи помнят об этом и уже вовсю работают над криптографическими алгоритмами, способными противостоять квантовым вычислениям.

4.5. Быстрое возведение в степень и возведение в степень по модулю

При обмене ключами по методу Диффи — Хеллмана, мы выполняем возведение в степень по модулю простого числа; такая операция называется возведением в степень по модулю. Мы, конечно, можем вычислить $g^x \bmod p$ путем возведения g в степень x и подсчетом остатка от деления на p. Однако, по небольшом размышлении мы можем увидеть, что это неэффективно. Число g^x может оказаться очень большим; в то же время конечный результат всегда будет меньше p. Было бы замечательно найти способ вычислить выражение, избежав расчетов потенциально огромных степеней, которые все равно в конце сократятся по модулю p. Используя свойства модульной арифметики, мы имеем:

$$\begin{aligned} g^2 \bmod p &= g \cdot g \bmod p = ((g \bmod p)(g \bmod p)) \bmod p \\ g^3 \bmod p &= g^2 \cdot \mathrm{g} \bmod p = ((g^2 \bmod p)(g \bmod p)) \bmod p \\ &\vdots \\ g^x \bmod p &= g^{x-1} \cdot \mathrm{g} \bmod p = ((g^{x-1} \bmod p)(g \bmod p)) \bmod p \end{aligned}$$

Таким образом, мы можем избежать расчета больших степеней по модулю p, начав с квадратного модуля p, а затем используя полученный результат для расчета кубического модуля p, мы будем так продолжать, пока не дойдем до степени x.

Существует еще более эффективный способ, который представляет собой стандартный подход к возведению в степень по модулю. Чтобы добраться до него, нам надо найти общий быстрый способ возведения в степень (не по модулю). Это даст нам

общий инструмент для расчета больших степеней, который мы приспособим к расчету больших модульных степеней.

Чтобы увидеть, как все работает, мы начинаем с записи степени в двоичной системе счисления:

$$x = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_0 2^0$$

где каждое b_i — это один бит в двоичном представлении x. Зная это, мы можем вычислить g^x следующим образом:

$$g^x = g^{b_{n-1}2^{n-1}+b_{n-2}2^{n-2}+\cdots+b_0 2^0}$$

Данное равенство эквивалентно:

$$g^x = (g^{2^{n-1}})^{b_{n-1}} \times (g^{2^{n-2}})^{b_{n-2}} \times \cdots \times (g^{2^0})^{b_0}$$

Мы берем выражение справа налево и начинаем вычислять $(g^{2^0})^{b_0}$. Затем мы вычисляем $(g^{2^1})^{b_1}$, $(g^{2^2})^{b_2}$, $(g^{2^3})^{b_3}$ и так далее. Но $g^{2^0} = g^1 = g$, g^{2^1} является квадратом g, g^{2^2} — это квадрат g^{2^1}, g^{2^3} — квадрат g^{2^2}, и в общем виде $g^{2^k} = (g^{2^{k-1}})^2$, так как $(g^{2^{k-1}})^2 = g^{2\cdot 2^{k-1}}$. Это означает, что мы можем вычислить основание возведения в степень, g^{2^i} для $i = 1, 2, \ldots, n-1$, для каждого множителя справа налево путем возведения в квадрат основания предыдущего множителя. Что приводит нас к алгоритму 4.3, возведению в степень путем поочередного возведения в квадрат.

Алгоритм берет в качестве входных данных g и x и возвращает g^x. Он работает, выполняя описанные нами вычисления в порядке справа налево. В первой строчке мы задаем основание, c, равное g, которое равно g^{2^0}. Мы используем переменную d, которой в строчке 2 изначально присваиваем значение x, чтобы получить двоичное представление x. Результат рассчитан в переменной r, которой в строчке 3 изначально присваивается значение 1. Цикл в строчках 4–8 выполняется столько же раз, сколько имеется бит в двоичном представлении x. Самый правый бит d — это 1, что мы и проверяем в строчке 5, затем умножаем наш результат на множитель, c, который мы вычислили. Затем в двоичном представлении x нам нужно передвинуть один бит влево. Для этого

мы делим d на 2; таким образом, в строчке 7 отрезается один бит справа. В конце каждой итерации цикла мы возводим нынешнее c в квадрат, как видно в строчке 8. Таким образом мы начинаем итерацию цикла k при c равным $g^{2^{k-1}}$, и это покрывает первую итерацию, в которой мы видели, что у нас $g^{2^0} = g^1 = g$.

Алгоритм 4.3. Возведение в степень путем поочередного возведения в квадрат

`ExpRepeatedSquaring(`g, x`)` $\rightarrow r$

Вводные данные: g, целочисленное основание; x, целочисленный показатель степени.

Выводимые данные: r, равное g^x.

```
1  c ← g
2  d ← x
3  r ← 1
4  while d > 0 do
5      if d mod 2 = 1 then
6          r ← r × c
7      d ← ⌊d/2⌋
8      c ← c × c
9  return r
```

Операция по модулю 2 в строчке 5 на самом деле может обойтись и без деления. Число кратно 2, если его последний бит — 0; в противном случае оно не кратно. Так что нам лишь нужно проверить, является ли последняя цифра d нулем. Делается это очень просто, с помощью операции «побитовое И», *bitwise* `and`, которая бит за битом перебирает два числа и возвращает число, чей бит i — 1, если оба бита i двух чисел — это 1; в противном случае возвращает 0. В нашем случае мы применяем «побитовое И» к d и числу с равным количеством битов, где все биты представляют собой 0 за исключением самого последнего, который является 1. Мы еще встретимся с операцией «побитовое И» в разделе 13.7. Можете пролистать вперед и заглянуть в таблицу 13.4, чтобы увидеть, как она работает.

Точно так же с делением на 2 в строчке 7, оно там не обязательно. Нам всего-то нужно отделить самый правый бит. Это то же самое, что сдвинуть все биты вправо на одну позицию (и тогда

самый правый свалится вниз). Такое действие реализовано в операции названной сдвигом вправо. Мы подробно рассмотрим побитовую операцию сдвига вправо в разделе 16.1, а пока вы можете обратиться к рисунку 16.1.

Пример операции алгоритма можно увидеть в таблице 4.9, где мы использовали его, чтобы вычислить 13^{13}. Каждый ряд, кроме последнего, соответствует значениям c, r и d в строчке 5 цикла алгоритма. Последний ряд показывает конечный результат на момент выхода из цикла. Вы можете убедиться, что, когда последняя цифра в d — 1, значение r в следующем ряду является произведением c и r из нынешнего ряда; в противном же случае остается неизменным. Для вычисления требуются четыре итерации цикла; это куда меньше, чем привычные 13 операций умножения для вычисления 13^{13}.

Мы можем судить о выполнении алгоритма по количеству итераций цикла. Как мы говорили, оно равно количеству битов в двоичном представлении степени x, которая представляет собой $\lg x$. Поэтому возведение в степень путем поочередного возведения в квадрат требует $O(\lg x)$ итераций.

Таблица 4.9. Возведение в степень путем поочередного возведения в квадрат: 13^{13}

$c = g^{2^i} = 13^{2^i}$	r	d
13	1	1101
169	13	110
28561	13	11
815730721	371293	1
	302875106592253	

Вопрос теперь в том, сколько времени занимает каждая итерация цикла. Вычисление по модулю и деление на 2 не требуют много времени, так как выполняются с помощью простых бинарных операций. Умножение в строчке 6 и возведение в квадрат в строчке 8 могут занять некоторое время. Вообще для представления целого числа компьютеры используют фиксированное количество битов, скажем, 32 или 64. Операции, включающие в себя такие числа, выполняются очень быстро и называются операциями с одинарной

точностью; расчеты с такими операциями называются арифметикой одинарной точности. Если же наши числа не укладываются в количество бит, предложенных компьютером, тогда компьютеру нужно прибегнуть к арифметике многократной точности, также известной как арифметика произвольной точности, или арифметика больших чисел. Арифметика многократной точности требует гораздо больше вычислительных ресурсов, чем арифметика одинарной точности. Проведем аналогию с людьми. В младших классах школы мы заучиваем наизусть таблицу умножения и в итоге можем мгновенно умножать однозначные числа. А вот чтобы перемножить многозначные числа, нам приходится выполнять длинные операции умножения, которые занимают куда больше времени (если вдуматься, то для двух n-значных чисел требуется n^2 операций умножения, плюс еще операции сложения). То же самое с компьютерами. Многократное умножение двух чисел a и b из n и m битов, соответственно, с использованием подходящего алгоритма (адаптации известного со школьной скамьи длинного традиционного способа умножения) требует nm операций одинарного умножения, таким образом, $O(nm) = O(\lg a \lg b)$. Возведение в квадрат может быть вдвое быстрее умножения, со сложностью $O((n^2 + n)/2)$ для числа из n битов. Хотя сокращение времени вдвое может быть значимо в практическом применении, сложность все же остается на $O(n^2)$ и совсем не меняет общую сложность алгоритма 4.3.

Мы полагаем, что при каждой итерации цикла мы получаем числа с удвоенным количеством битов, которые у них были в предыдущей итерации. Строчка 6 выполняется меньше раз, чем строчка 8, и r меньше, чем c, поэтому нам нужно разобраться только со строчкой 8. Первая итерация цикла включает в себя умножение чисел до g, требуя $O((\lg g)^2)$ времени. Вторая итерация цикла включает в себя умножение чисел размером g^2, требуя $O((\lg g^2)^2) = O((2\lg g)^2)$ времени. Последняя итерация цикла включает в себя умножение чисел размером $g^{x/2}$, требуя $O((\lg g^{x/2})^2) = O((x/2 \lg g)^2)$. Подведя итог, мы получаем $O((\lg g)^2) + O((2\lg g)^2) + \ldots + O((x/2 \lg g)^2)$. Каждое слагаемое в сумме имеет вид $O((2^{i-1} \lg g)^2)$, для $i = 1, 2, \ldots, \lg x$ (у нас $\lg x$ итераций цикла). У нас множество слагаемых, влияющих на общую сложность вычисле-

ния, но именно наибольшее слагаемое отвечает за рост функции сложности, поэтому общая сложность $O((x/2 \lg g)^2) = O((x \lg g)^2)$.

Алгоритм 4.3. Возведение в степень по модулю путем поочередного возведения в квадрат

`ModExpRepeatedSquaring(`g, x, p`)` $\rightarrow r$

Вводные данные: g, целочисленное основание; x, целочисленный показатель степени; p, делитель.

Выводимые: r, равное $g^x \bmod p$.

```
1  c ← g mod p
2  d ← x
3  r ← 1
4  while d > 0 do
5      if d mod 2 = 1 then
6          r ← (r × c) mod p
7      d ← ⌊d/2⌋
8      c ← (c × c) mod p
9  return r
```

Тут у нас возникает небольшое затруднение, так как мы пришли к двум разным мерам сложности. Какую же из них оставить? Сложность $O(\lg x)$ или $O((x \lg g)^2)$? Ответ зависит от наших ожиданий. $O(\lg x)$ — количество итераций цикла. Если же нам важна сложность операций умножения, которые нужно выполнить с помощью арифметики многократной точности, тогда мы используем $O((x \lg g)^2)$.

Теперь вспомним, что нам нужен эффективный способ выполнения возведения в степень по модулю; это всего короткий этап из алгоритма 4.3. Благодаря арифметическим свойствам деления с остатком мы можем выполнять деление по модулю каждый раз, когда мы вычисляем c и r. Так мы приходим к алгоритму 4.4, взятию в степень по модулю путем поочередного возведения в квадрат. Это тот же, что и прежде, алгоритм, однако мы выполняем все операции умножения по модулю делителя p. Примером алгоритма в действии служит таблица 4.10; просто потрясающе, что мы можем вычислить такое число, как $155^{235} \bmod 391$, всего-то при помощи нескольких операций умножения и ни разу не столкнувшись

с большими числами. Цикл в строчках 4–8 выполняется $\lg x$ раз; если нас не заботит время расчета умножения в строчке 6 и возведения в квадрат в строчке 8, сложность алгоритма будет $O(\lg x)$.

Таблица 4.10. Взятие в степень по модулю путем поочередного возведения в квадрат: $155^{235} \bmod 391$

$c = g^{2^i} = 155^{2^i} \bmod 391$	r	d
155	1	11101011
174	155	1110101
169	382	111010
18	382	11101
324	292	1110
188	229	111
154	42	11
256	232	1
	314	

Однако время может являться для нас важным фактором, так как операции умножения и возведения в квадрат могут касаться арифметики многократной точности. Отметим еще раз: так как строчка 6 выполняется меньше раз, чем строчка 8, и r меньше, чем c, нас интересует только строчка 8. К тому же из-за деления по модулю мы имеем $c < p$. Возведение в квадрат числа с $\lg p$ однозначных чисел может быть выполнено, как мы видели, за $O((\lg p)^2)$ времени. Это дает нам общую сложность $O(\lg x(\lg p)^2)$. Если мы полагаем, что $x \leq p$, то получаем $O((\lg p)^3)$.

Примечания

У криптографии длинная и захватывающая история. Прекрасный доклад о ней написал Дэвид Кан [104]. Еще одна популярная история кодов и их взломов была написана Саймоном Сингхом [187]. Первая цитата Филипа Циммермана взята из статьи, которую он написал как часть пользовательского руководства по PgP [224]; PgP* стало первым надежным и широко распростра-

* Pretty good Privacy — неплохая защита конфиденциальности.

ненным средством шифрования. Вторая цитата взята из аудиоинтервью газете «Гардиан» [225].

Исследовательские результаты Питера Норвига, содержащие частоту употребления букв, опубликованы на сайте: http://norvig.com/mayzner.html.

Если хотите знать больше об устройстве AES, посмотрите документацию по конструкции Рэндала, написанную его создателями [46]. Подробно о том, как реализовать AES на специальных платформах, можно узнать в [74]. AES был опубликован Национальным институтом стандартов и технологий [206]. Шифрование с открытым ключом стало известно благодаря революционной научной работе Уитфилда Диффи и Мартина Хеллмана [50].

Брюс Шнайер написал множество книг, посвященных криптографии и секретности. Его «Прикладная криптография» является классическим произведением в данной области [174]; для более глубокого изучения советуем посмотреть [63]. Учебное пособие Каца и Линделла освещает многие аспекты криптографии, соединяя теоретические и практические соображения [105]. Хорошим другом и проводником в мир криптографии и информатики может стать «Руководство по прикладной криптографии» [137], которое лежит в свободном доступе в Сети.

Если вам интересны математические концепции, лежащие в основе криптографии, прочитайте [153]. Развернутое описание теоретических основ вы найдете в двухтомнике Одеда Голдрейха [79], [80]. Глубинное понимание криптографии требует хорошего знания теории чисел; введение Сильвермана — прекрасная книга для знакомства с ней [186].

Согласно Кнуту [113, с. 461–462], идея расчета степени путем поочередного возведения в квадрат стара как мир, она появилась в Индии за 200 лет до нашей эры. Алгоритм возведения в степень по модулю, который мы использовали, был опубликован в Персии в 1427 году нашей эры.

В 1994 году Питер Шор представил общественности алгоритм, решающий задачу дискретного логарифма за полиномиальное время (алгоритм был издан в виде журнальной статьи 1997 году) [185]. Для знакомства с квантовыми вычислениями советуем [151].

Упражнения

1. Реализуйте программу расшифровывающую зашифрованные тексты, созданные с помощью подстановочного шифра. Используйте таблицу частоты употребления, доступную в Интернете. Для угадывания слов можете воспользоваться списками слов, они есть в виде орфографических словарей в операционных системах, или же можете найти их в Сети.
2. Реализуйте программу шифрования и дешифрования с помощью одноразового блокнота. Сделайте так, чтобы она работала в двух разных режимах: в одном она должна зашифровывать и расшифровывать, используя арифметические операции по модулю, а во втором шифровать и расшифровывать с помощью `xor`. Оцените эффективность выполнения каждого режима.
3. Чтобы понять, как работает AES, хорошо бы выяснить, как он меняет сообщение всего за один цикл. Следите за ходом изменения состояния AES в течение одного цикла, где все вводимые биты — нули. Проделайте то же самое со всеми введенными битами, которые, за исключением последнего, являются нулями.
4. Напишите программу, которая, с заданным простым p, находит примитивные элементы для мультипликативной группы. Чтобы выполнить это задание, можете взять случайное число из 2, 3,…, p – 1 и проверить, является ли оно генератором. Генераторы не так уж редки, поэтому вы наверняка наткнетесь на них.
5. Реализуйте возведение в степень путем поочередного возведения в квадрат двумя способами. В первом используйте обычные операторы деления и вычисления по модулю, предусмотренными избранным вами языком. Во втором используйте «побитовое И» и сдвиг вправо. Оцените эффективность выполнения каждой реализации.

5 Делимся секретами

Представьте, что вы — директор компании «Супернадежные хранилища» (СНХ), разрабатывающей новейший вид сейфов. Обычно у сейфов есть замок и ключ. Сейфы СНХ поставляются с замком и двумя ключами. Ключи сделаны таким образом, что, если вы заперли сейф одним ключом, то открыть его может только второй.

Какое же преимущество у ваших сейфов по сравнению с обычными? Если кто-нибудь, например Алиса, захочет послать что-то кому-то, например Бобу, она может поместить посылку в обычный сейф и отправить его Бобу. По пути никто не сможет взломать сейф, и только Боб сможет открыть его, если у него будет копия ключа. Вот тут и проблема, потому что вы не можете послать ключ вместе с сейфом; кто-нибудь, скажем, Ева (шпионка), может перехватить посылку и спокойно открыть ключом сейф. Так что вам надо найти способ передать Бобу ключ, чтобы он не попал в третьи руки.

Теперь возьмем сейфы СНХ. Алиса может послать Бобу вместе с сейфом первый ключ. Второй ключ она надежно хранит у себя. Боб кладет сообщение в сейф, запирает его и отправляет обратно Алисе. Второй ключ есть только у Алисы, она никому его не давала и не отсылала вместе с сейфом Бобу, поэтому открыть сейф и достать сообщение Боба может только она одна.

К тому же сейфы СНХ дают вашим клиентам дополнительное преимущество. Алиса может положить сообщение в сейф и запереть его с помощью второго ключа. Она посылает Бобу сейф вместе с первым ключом. Боб открывает его, зная, что закрыть сейф мог только тот, у кого хранится второй ключ. Если он знает, что второй ключ у Алисы, то он может быть уверен, что именно Алиса прислала ему сейф, а не кто-то еще.

5.1. Шифрование с открытым ключом

Перенесем наш пример в компьютерный мир, а точнее — в область криптографии. Когда вы хотите защитить сообщение, вы с помощью своего ключа зашифровываете его, а получатель с помощью своего ключа расшифровывает его. Обратите внимание, что у каждого свой ключ. Если ключи для шифрования и дешифрования одинаковые, тогда мы имеем дело с симметричным шифрованием. Но это не обязательно. Можете представить систему, как с сейфами СНХ, в которой используются разные ключи. Здесь у нас асимметричное шифрование. Используются два ключа: один для шифрования, а другой для дешифрования. Чтобы система работала, один ключ становится публичным, или открытым, а другой — личным, или закрытым. Алиса всегда держит в тайне закрытый ключ, а открытый спокойно посылает. Кто угодно может использовать открытый ключ Алисы, чтобы зашифровать сообщение. Но только Алиса сможет расшифровать его, потому что только у нее есть соответствующий закрытый ключ. Так как один из ключей открытый, сам метод называется шифрованием с открытым ключом.

Шифрование с открытым ключом решает проблему рассылки ключей, то есть задачу, каким образом Алиса и Боб могут обменяться ключами, чтобы зашифровать свои сообщения. Все просто: им не нужен никакой такой обмен. Боб может использовать открытый ключ Алисы, чтобы зашифровать сообщения и послать их Алисе, а она, в свою очередь, расшифрует их с помощью закрытого ключа. Алиса может использовать открытый ключ Боба, чтобы зашифровать сообщения для Боба, которые он потом расшифрует своим закрытым ключом. Ключи идут парами, поэтому мы можем назвать $P(A)$ — публичный ключ из пары A и $S(A)$ — личный ключ из пары A. Одному закрытому ключу соответствует один открытый, так что каждая пара уникальна. Если M — исходное незашифрованное сообщение, то операция шифрования сообщения с помощью открытого ключа $P(A)$ такова:

$$C = E_{P(A)}(M)$$

Обратная операция, позволяющая расшифровать сообщение, зашифрованное с помощью открытого ключа $P(A)$, с использованием закрытого ключа $S(A)$ такова:

$$M = D_{S(A)}(C).$$

Процесс шифрования следует из написанного выше.

1. Боб генерирует пару ключей $B = (P(B), S(B))$.
2. Алиса получает открытый ключ Боба $P(B)$. Есть множество разных способов. Например, Боб может разместить его на публичном сервере или переслать напрямую Алисе по электронной почте.
3. Алиса зашифровывает свое сообщение с помощью $P(B)$:

$$C = E_{P(B)}(M).$$

4. Алиса отправляет C Бобу.
5. Боб расшифровывает C с помощью своего закрытого ключа:

$$M = D_{S(B)}(C).$$

То, что открытому ключу соответствует только один закрытый ключ, означает, что наподобие того, как Алиса запирает сейф, мы так же можем использовать личный, закрытый ключ для шифрования сообщения:

$$C = E_{S(A)}(M).$$

Получившееся зашифрованное сообщение может быть расшифровано лишь с помощью соответствующего открытого ключа:

$$M = D_{P(A)}(C).$$

Зачем это нужно? Так как открытому ключу соответствует только один закрытой ключ, Боб может быть уверен, что полученное им сообщение зашифровано владельцем закрытого ключа. Поэтому, если он знает, что владелица — Алиса, он знает, что сообщение пришло от нее. Таким образом, шифрование с помощью открытого ключа и дешифрование с помощью закрытого ключа является хорошим способом узнать, откуда пришло сообщение. Это то же самое, как если бы мы оставили на бумаге свою подпись. Так как наша подпись неповторима, каждый, кто знает ее,

может удостовериться, что бумагу подписали именно мы. Поэтому шифрование при помощи закрытого ключа называется подписанием сообщения, а зашифрованное с помощью закрытого ключа сообщение — цифровой подписью. Весь процесс подписания проходит тем же образом, что и шифрование.

1. Алиса генерирует пару ключей $A = (P(A), S(A))$. Она любым удобным способом передает $P(A)$ Бобу.
2. Алиса с помощью личного ключа $S(A)$ подписывает свое сообщение M:

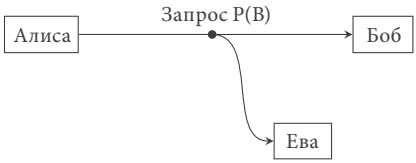
$$C = E_{S(A)}(M).$$

3. Алиса посылает (M, C) Бобу.
4. Боб проверяет C, используя открытый ключ Алисы $P(A)$:

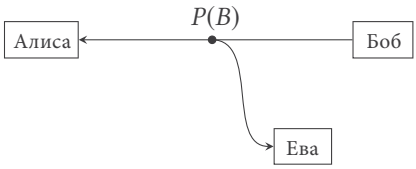
$$M \stackrel{?}{=} D_{P(A)}(C).$$

Итого в шифровании с открытым ключом у каждой стороны вместо одного ключа набор из двух. Один можно показывать кому угодно и выкладывать в публичный доступ, второй же хранится в секрете. Любой пользователь может применять открытый ключ для шифровки сообщения, которое может быть расшифровано только с помощью закрытого ключа. Если шпион, скажем, Ева, следит за перепиской Алисы и Боба, она может получить лишь открытый ключ и зашифрованные сообщения, поэтому ей никак не добраться до открытого текста, что мы и видим на рисунке 5.1. К тому же с помощью закрытого ключа мы можем шифровать сообщения, то есть подписывать их. Кто угодно может достать открытый ключ и удостовериться, что сообщения подписаны с помощью закрытого ключа. Если владелец закрытого ключа известен, то известно, кем подписаны сообщения.

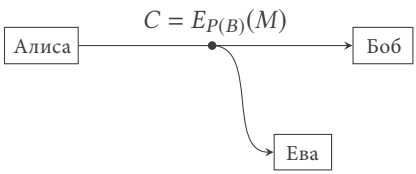
Имея в распоряжении две операции, шифрование и подписание, вы можете их совмещать, одновременно шифруя и подписывая сообщение. Первым делом, используя закрытый ключ, вы подписываете сообщение. Затем шифруете сообщение и подпись с помощью открытого ключа получателя сообщения. Получатель сначала расшифровывает с помощью соответствующего закрытого ключа ваше сообщение и получает открытый текст с подписью.



(a) Алиса просит у Боба его открытый ключ



(b) Боб посылает открытый ключ Алисе



(c) Алиса отправляет Бобу зашифрованное сообщение

Рисунок 5.1. Шифрование с открытым ключом между Алисой и Бобом

Потом он расшифровывает с помощью вашего открытого ключа подпись и проверяет, подходит ли она к расшифрованному открытому тексту сообщения. Алиса и Боб делают следующее:

1. Алиса генерирует пару ключей $A = (P(A), S(A))$. Она отсылает $P(A)$ Бобу любым удобным способом.
2. Боб генерирует пару ключей $B = (P(B), S(B))$. Он отсылает $P(B)$ Алисе любым удобным способом.
3. Алиса подписывает свое сообщение M своим закрытым ключом $S(A)$:
$$C_1 = E_{S(A)}(M).$$
4. Алиса с помощью открытого ключа Боба $P(B)$ зашифровывает свое сообщение и подпись, которую сгенерировала, то есть (M, C_1):
$$C_2 = E_{P(B)}(M, C_1).$$
5. Алиса посылает C_2 Бобу.

6. Боб расшифровывает C_2 с помощью своего закрытого ключа:

$$(M, C_1) = D_{S(B)}(C_2).$$

7. Боб проверяет C_1, используя открытый ключ Алисы $P(A)$:

$$M \stackrel{?}{=} D_{P(A)}(C_1).$$

Чтобы увидеть общую картину, нам теперь надо рассмотреть, как на самом деле происходят все эти шифрования, дешифрования и подписания. Они выполняются с помощью простых процессов, которые основаны на результатах, полученных из теории чисел.

5.2. Криптосистема RSA

Одна из первых криптосистем с открытым ключом, примененная на практике и широко используемая сегодня, — это криптосистема RSA. Она названа в честь ее изобретателей, Рона Ривеста, Ади Шамира и Леонарда Адлемана, которые впервые опубликовали данную систему в 1977 году, хотя независимо от них она была открыта еще раньше, в 1973 году, Клиффордом Коксом, работавшим в Центре правительственной связи Великобритании; однако до 1997 года работа Кокса была засекречена. Мы видели в разделе 4.4, что то же самое случилось обменом ключами по методу Диффи — Хеллмана, который еще до Диффи и Хеллмана был разработан Малкомом Уильямсоном. На самом деле открытые Кокса случилось раньше. Уильямсон был его другом и поэтому знал об открытии, которое потом и подтолкнуло его к разработке обмена ключами Диффи — Хеллмана.

RSA позволяет генерировать пары открытых-закрытых ключей. Прежде чем мы пошагово рассмотрим данный метод, следует привести определение. Если у двух чисел нет общих делителей, кроме 1, они называются взаимно простыми. Помня данное определение, рассмотрим следующие шаги, которые включает в себя RSA.

1. Выберем два больших простых числа, скажем, p и q, для которых $p \neq q$.

2. Найдем их произведение, $n = pq$.
3. Выберем целое число e, взаимно простое для $(p - 1)(q - 1)$.
4. Найдем число d, $1 \leq d < (p - 1)(q - 1)$ такое, чтобы:

$$(e \cdot d) \bmod [(p - 1)(q - 1)] = 1.$$

5. Пара из открытого и закрытого ключей — это $A = (P(A), S(A)) = ((e, n), (d, n))$.
6. Кортеж $P(A) = (e, n)$ является вашим открытым ключом RSA.
7. Кортеж $S(A) = (d, n)$ является вашим закрытым ключом RSA.

Следует пояснить, как мы выбрали d в четвертом пункте. Мы уделим этому моменту внимание, но сначала давайте посмотрим, как работает RSA на практике.

Мы сказали, что оба числа, p и q, должны быть большими. Чем больше, тем лучше, однако чем числа больше, тем больше вычислительные затраты на шифрование и расшифровку. Однако каждое из них вполне может быть 2084-битным, а еще лучше — 4096-битным.

Для e нет никаких ограничений в размере; оно даже может быть равно 3. Чаще всего выбирают $e = 2^{16} + 1 = 65537$; у данного числе хорошие параметры, благодаря которым сообщения, зашифрованные с помощью RSA, вряд ли будут взломаны.

Шифрование и дешифрование используют одну и ту же функцию:

$$f(m, k, n) = m^k \bmod n.$$

В то же время параметры, передающиеся функции, при шифровании и дешифровании различны. Так что, когда мы шифруем, m — это открытый текст сообщения, а k равняется e из открытого ключа участника коммуникации. Когда же мы дешифруем, m — это зашифрованный текст, а k равно d из закрытого ключа участника коммуникации.

Иными словами, в данном методе шифрование сообщения M выполнено путем получения C из M, применяя:

$$C = E_{P(A)}(M) = M^e \bmod n.$$

Расшифровка текста C выполняется благодаря вычислению:

$$M = D_{S(A)}(C) = C^d \bmod n.$$

Подписание сообщения

$$C = E_{S(A)}(M) = M^d \bmod n,$$

а процесс удостоверения подписи

$$D_{P(A)}(C) = C^e \bmod n.$$

Так как для любых целых u, v:

$$(u \bmod n)(v \bmod n) \bmod n = uv \bmod n,$$

расшифровка представляет собой:

$$M = D_{S(A)}(C) = C^d \bmod n = (M^e \bmod n)^d \bmod n = M^{ed} \bmod n.$$

и похожим образом происходит удостоверение подписи:

$$D_{P(A)}(C) = C^e \bmod n = (M^d \bmod n)^e \bmod n = M^{de} \bmod n.$$

Вы можете заметить, что на самом деле они вычисляют одно и то же, как и положено. Когда вы расшифровываете зашифрованное сообщение, то извлекаете открытый текст. Когда проверяете подлинность подписи, вы снова вынимаете оригинальное сообщение. Единственное, в чем различие, — в последовательности использования ключей. В шифровании и расшифровке сперва применяется открытый ключ, а затем закрытый. При подписании и проверке подлинности наоборот, сперва закрытый ключ, а потом открытый.

Теперь представим, что Алиса хочет зашифровать сообщение M. Если p и q по 2048 бит каждый, то $p - 1$ и $q - 1$ должны быть по 2047 каждый, а n должно быть 2047 + 2047 = 4094. Таким образом, M должно быть меньше 4094 бит. Если же оно больше, тогда Алисе придется разделить M так, чтобы каждый кусок был меньше 4094 бит. Алисе известен открытый ключ Боба $P = (e, n)$. Чтобы послать Бобу M, Алиса рассчитает C, как указано выше, а Боб получит C и расшифрует его.

Чтобы вычисления работали, открытый текст сообщения M должен быть целым числом. Это никак не сказывается на работо-

способности RSA. Текстовое сообщение в компьютере представляет собой ряд битов, шифрующих текст с помощью подходящего кодирования, такого как ASCII. Мы затем разбиваем текст на куски с требуемым количеством битов. Десятичным значением куска является число M.

Вы на примере можете убедиться, что RSA действительно работает. Боб хочет зашифровать сообщение $M = 314$ и послать его Алисе.

1. Алиса берет $p = 17$ и $q = 23$.
2. Алиса делает расчет $n = pq = 17 \times 23 = 391$.
3. Она выбирает $e = 3$. Вы можете проверить, что e — число взаимно простое с
$$(p - 1)(q - 1) = (17 - 1)(23 - 1) = 16 \times 22 = 352$$
4. Алиса находит $d = 235$, которое удовлетворяет условию $1 \le d < 352$ и
$$(e \cdot d) \bmod [(p - 1)(q - 1)] = 1,$$
или эквивалентному $(3 \times 235) \bmod 352 = 1$.
5. Пара из открытого и закрытого ключей — $A = ((3, 391), (235, 391))$.
Кортеж $P(A) = (3, 391)$ — это открытый ключ RSA.
Кортеж $S(A) = (235, 391)$ — это закрытый ключ RSA.
6. Боб получает от Алисы $P(A)$ и шифрует M с помощью операции:
$$C = M^e \bmod n = 314^3 \bmod 391 = 155.$$
7. Боб посылает Алисе C.
8. Алиса расшифровывает C с помощью операции:
$$C^d \bmod n = 155^{235} \bmod 391 = 314.$$
и получает исходный открытый текст.

Если выражение вроде $155^{235} \bmod 391 = 314$ вас пугает, можете заглянуть в таблицу 4.10 раздела 4.5 и узнать, как осуществить эффективный расчет.

Все замечательно, вот только этап 4 до сих пор остается загадкой. Чтобы понять, как мы находим d, нужно обладать багажом знаний, касающимся теории чисел. Мы знаем, что мультиплика-

тивная инверсия, или обратная величина, числа x — это число $1/x$ или x^{-1}, такое, что $xx^{-1} = 1$. В модульной арифметике модульная мультипликативная инверсия целого числа x по модулю n ($n > 0$) есть целое число x^{-1}, $1 \leq x^{-1} \leq n - 1$, такое, что $xx^{-1} \bmod n = 1$. Оно эквивалентно $xx^{-1} = kn + 1$, или $1 = xx^{-1} - kn$ для целого числа k. Поэтому, когда в пункте 4 мы находим число d, $1 \leq d < (p - 1)(q - 1)$, чтобы $(e \cdot d) \bmod [(p - 1)(q - 1)] = 1$, мы на самом деле находим модульную мультипликативную инверсию e по модулю $(p - 1)(q - 1)$, для которого

$$1 = ed + k(p - 1)(q - 1)$$

для целого числа k. Это напрямую вытекает из $1 = xx^{-1} - kn$, с противоположным знаком перед k, что не имеет значения, потому что k является целым числом, поэтому нам подойдет даже отрицательное значение. В мире действительных чисел мы всегда можем найти инверсию любому числу $x \neq 0$. Но каким образом мы находим модульную инверсию? Более того, всегда ли существует модульная инверсия?

Модульная мультипликативная инверсия x по модулю n существует только лишь при условии, что x и n взаимно простые. Именно поэтому, выбирая e, мы требуем, чтобы оно было взаимно простым с $(p - 1)(q - 1)$. С таким e мы точно знаем, что существует d с нужными нам параметрами. Посмотрим, как нам его найти.

Наибольший общий делитель (НОД) двух целых чисел является наибольшим целым числом, на которое делится и то и другое. Чтобы найти НОД для двух положительных целых чисел, мы прибегаем к старому доброму алгоритму, найденному в «Началах» Евклида; он изображен в алгоритме 5.1. Алгоритм Евклида вытекает из утверждения, что НОД двух целых чисел $a > 0$ и $b > 0$ — это НОД b и остатка от деления a на b, если только b не делит a — в этом случае, согласно определению, НОД — это само b. Алгоритм 5.1 работает благодаря тому, что делает рекурсивный вызов, заменяя в каждом вызове b и $a \bmod b$ на a и b (строчка 4). Рекурсия прекращается, когда $b = 0$ (строчки 1–2), где b является результатом $a \bmod b$ предыдущего рекурсивного вызова.

Доказательство того, что у a и $a \bmod b$ один и тот же НОД, опирается на базовые свойства деления. Если d — общий делитель a и b, то $a = k_1 d$ и $b = k_2 d$ для целых положительных чисел k_1, k_2. К тому же, если $r = a \bmod b$, мы имеем $r = a - k_3 b$ для целого положительного числа k_3. Заменяя значения a и b, мы получаем $r = k_1 d - k_3 k_2 d = (k_1 - k_3 k_2)d$, таки образом, r делится на d, и это означает, что все делители для a и b также являются делителями для b и $a \bmod b$.

С другой стороны, если d — общий делитель и для b, и для $r = a \bmod b$, тогда $b = z_1 d$ и $r = z_2 d$ для целый положительных чисел z_1, z_2. В то же время мы имеем $a = z_3 b + r$ для целого положительного числа z_3. Заменяя значения r и b, мы получаем $a = z_3 z_1 d + z_2 d = (z_3 z_1 + z_2)d$, таким образом, a делится на d, и это означает, что все делители для b и $a \bmod b$ также являются делителями для a и b. Следовательно, у a и b точно такие же делители, как у b и $a \bmod b$, и значит, у них должен быть точно такой же наибольший общий делитель.

Алгоритм 5.1. Алгоритм Евклида

`Euclid`(a, b) $\rightarrow d$

Вводные данные: a, b, положительные целые числа.

Выводимые данные: d, наибольший общий делитель a и b.

```
1  if b = 0 then
2      return a
3  else
4      return Euclid(b, a mod b)
```

В таблице 5.1 представлен пример операции евклидова алгоритма, который определяет НОД для 160 и 144, который, как можно видеть, равен 16. В каждом ряду таблицы отображено то, что происходит при каждом рекурсивном вызове. Можно увидеть, что при $a > b$ количество рекурсивных вызовов, необходимых алгоритму, есть $O(\lg b)$. Если же $b > a$, то $O(\lg a)$. Все потому, что самый первый вызов переставляет a и b: чтобы увидеть это, проверьте операцию алгоритма, определяющую НОД для 144 и 160.

Таблица 5.1. Пример работы алгоритма Евклида

a	b	$a \bmod b$
160	144	16
144	16	0
16	0	

Обобщение алгоритма Евклида — расширенный алгоритм Евклида — может помочь найти (помимо НОД двух целых положительных чисел) способ, который позволит нам получить *gcd*, то есть НОД этих чисел путем умножения и сложения. Особенно, если $r = gcd(a, b)$, тогда мы можем найти целое число x и y, вследствие чего:

$$r = gcd(a, b) = xa + yb.$$

Расширенный алгоритм Евклида — это алгоритм 5.2, и он является путеводной нитью в нахождении модульных мультипликативных инверсий. Количество рекурсивных вызовов опять-таки $O(\lg b)$, если $a > b$, или $O(\lg a)$, если $a < b$. Можно доказать, что:

$$|x| \leq \frac{b}{\gcd(a,b)} \text{ и } |y| \leq \frac{a}{\gcd(a,b)},$$

где равенство бывает в том случае, когда a кратно b, или b кратно a. Если a и b взаимно простые, то мы имеем $gcd(a, b) = 1$. Это означает, что с помощью расширенного алгоритма Евклида мы можем найти две целых числа x и y, таким образом:

$$1 = xa + yb \text{ при } |x| < b \text{ and } |y| < a.$$

Это, в свою очередь, значит, что, если $0 < x < b$, то x — мультипликативная инверсия a по модулю b. Если $x < 0$, то путем добавления и отбрасывания ab в вышеприведенной формуле мы получаем $1 = xa + ab + yb - ab = (x + b)a + (y - a)b$, при $0 < x + b < b$. Поэтому, если $x < 0$, мультипликативная инверсия a по модулю b — это $x + b$.

Алгоритм 5.3 показывает процедуру нахождения модульной мультипликативной инверсии в алгоритмической форме. Вы можете заметить, что, если расширенный алгоритм Евклида возвращает НОД, отличный от 1, то модульной мультипликативной

инверсии нет, поэтому мы возвращаем ноль, что является недопустимым значением.

Алгоритм 5.2. Расширенный алгоритм Евклида

```
ExtendedEuclid(a, b) → (r, x, y)
    Вводные данные: a, b, положительные целые числа.
    Выводимые данные: такие r, x, y, чтобы r = gcd(a, b) = xa + yb.)
1   if b = 0 then
2       return (a, 1, 0)
3   else
4       (r, x, y) = ExtendedEuclid(b, a mod b)
5       return (r, y, x − ⌊a/b⌋ · y)
```

a	b	$a \bmod b$	$(r, y, x - \lfloor a/b \rfloor \cdot y)$
3	352	3	$(1, -117, 1 - \lfloor 3/352 \rfloor \times (-117)) = (1, -117, 1)$
352	3	1	$(1, 1, 0 - \lfloor 352/3 \rfloor \times 1) = (1, 1, -117)$
3	1	0	$(1, 0, 1 - \lfloor 3/1 \rfloor \times 0) = (1, 0, 1)$
1	0		$(1, 1, 0)$

Рисунок 5.2. Пример работы расширенного алгоритма Евклида

На рисунке 5.2 можно увидеть пример работы расширенного алгоритма Евклида для взаимно простых чисел 3 и 352. Чтобы прочесть данную таблицу, вам нужно сверху вниз просмотреть столбцы a, b и $a \bmod b$, а затем последний столбец — снизу вверх, так как именно таким способом составляется триплет (r, x, y) при возвращении каждого рекурсивного вызова. К концу алгоритма мы имеем:

$$1 = 3 \times (-117) + 352 \times 1,$$

что, как можно убедиться, верно. Следовательно, мультипликативная инверсия 3 по модулю 352 есть $-117 + 352 = 235$. Перед нами значение d, которое мы использовали выше, в примере RSA.

В общем, чтобы найти число d закрытого ключа RSA, нам надо найти мультипликативную инверсию e по модулю $(p - 1)(q - 1)$. Для этого мы используем расширенный алгоритм Евклида со

входными данными e и $(p - 1)(q - 1)$. В результате мы получаем триплет $(1, x, y)$. Если $x > 0$, x — число, которое мы ищем. Если $x < 0$, то число, которое мы ищем, $x + (p - 1)(q - 1)$.

Алгоритм 5.3. Алгоритм модульной инверсии

`ModularInverse(`a, n`)` $\rightarrow r$

Вводные данные: a, n, положительные целые числа; n является модулем.

Выводимые данные: такое r, что $r = a^{-1}$, если оно существует, а иначе — 0

```
1  (r, x, y) = ExtendedEuclid(a, n)
2  if r ≠ 1 then
3      return 0
4  else if x < 0 then
5      return x + n
6  else
7      return x
```

Мы объяснили, как реализуются различные шаги RSA; осталось ответить на вопрос, почему же RSA вообще работает. Тут нам снова поможет теория чисел. Для любого целого положительного числа n количество целых чисел k при $1 \le k \le n$ и таких, чтобы k и n были взаимно простыми, является значением функции, называемой фи-функцией Эйлера и представленной как $\varphi(n)$. Если n — простое число, тогда оно взаимно простое со всеми положительными натуральными числами, которые меньше него, поэтому для простого n мы имеем $\varphi(n) = n - 1$.

В RSA мы используем $p - 1 = \varphi(p)$ и $q - 1 = \varphi(q)$. Когда два числа, p и q взаимно простые, $\varphi(pq) = \varphi(p)\,\varphi(q) = (p - 1)(q - 1)$. И p, и q — простые числа, следовательно, и взаимно простые, это означает, что в этапе 3 мы выбираем целое число e, которое является взаимно простым с $\varphi(pq) = \varphi(n)$. На самом деле в нашем доказательстве мы могли бы заменить $\varphi(n)$ на более громоздкое $(p - 1)(q - 1)$.

Фи-функция Эйлера появляется в теореме Эйлера, которая гласит, что для любого n, являющегося целым положительным числом, и любого $0 < a < n$, $a^{\varphi(n)} \bmod n = 1$. Особый случай в теореме Эйлера, когда n — простое число. Если мы возьмем p для обозначения такого n, у нас получится $a^{p-1} \bmod p = 1$. Вот мы и пришли к малой теореме Ферма, которая называется так для отличия от

другой теоремы Ферма, известной как великая теорема Ферма, которая веками не давала покоя математикам и ставила их в тупик.

Вернемся к RSA. Чтобы доказать, что она работает, мы должны удостовериться, что результатом ее дешифрования является исходное сообщение. Так как у подписания и проверки подлинности одинаковое шифрование и расшифровка, нам всего лишь надо проверить шифрование и расшифровку. Так как мы имеем:

$$D_{S(A)}(C) = C^d \bmod n = (M^e \bmod n)^d \bmod n = M^{ed} \bmod n$$

мы должны показать, что:

$$M^{ed} \bmod n = M \bmod n = M.$$

Первым делом рассмотрим случай, когда $M = 0$. Тогда у нас:

$$M^{ed} \bmod n = 0^{ed} \bmod n = 0 \bmod p = 0 = M,$$

и дело сделано; но нам все еще надо показать, что будет, если $M \neq 0$. Если $M \neq 0$, а p — простое число, то согласно малой теореме Ферма $M^{p-1} \bmod p = 1$. Вспомним, что для целого числа k

$$1 = ed + k(p - 1)(q - 1)$$

или эквивалентно

$$ed = 1 + k'(p - 1)(q - 1)$$

с учетом того, что $k' = -k$. Таким образом мы получаем:

$$\begin{aligned} Med \bmod p &= M^{1+k'}(p - 1)(q - 1) \bmod p \\ &= MM^{k'(n-1)(p-1)} \bmod p \\ &= [(M \bmod p)((M^{p-1})^{k'(q-1)} \bmod p)] \bmod p \\ &= [(M \bmod p]((M^{p-1} \bmod p)^{k'(q-1)} \bmod p)] \bmod p \\ &= [(M \bmod p)((1 \bmod p)^{k'(q-1)} \bmod p)] \bmod p \\ &= [(M \bmod p)(1^{k'(q-1)} \bmod p)] \bmod p \\ &= M \bmod p. \end{aligned}$$

Точно так же, пользуясь малой теоремой Ферма, где $M^{p-1} \bmod p = 1$, мы имеем:

$$M^{ed} \bmod q = M \bmod q.$$

Вот почему мы работали с произведением $(p - 1)(q - 1)$: мы применяем малую теорему Ферма к $p - 1$ и $q - 1$. Теперь обращаемся к другой теореме из теории чисел, Китайской теореме об

остатках, открытой китайским математиком Сунь Цзы в III–V веке нашей эры, и с ее помощью находим

$$M^{ed} \bmod p = M \bmod p$$

и

$$M^{ed} \bmod q = M \bmod q,$$

получаем

$$M^{ed} \bmod pq = M \bmod pq.$$

Но $pq = n$, так что

$$M^{ed} \bmod n = M \bmod n = M,$$

что мы и стремились показать.

RSA решает проблему передачи ключей, она анализировалась многие годы. Ее надежность базируется на сложности разложения на простые множители. Множитель числа — это целое число, которое делит заданное число без остатка. Простой множитель — это множитель, который, ко всему прочему, является простым. Разложение на простые множители — это процесс нахождения простых множителей числа и выражения этого числа в виде их произведения. Известные на сегодняшний день алгоритмы, решающие данную проблему, очень емкие в вычислительном плане, поэтому требуется куча времени, чтобы разложить на множители большое целое число. В нашем случае, с заданным n, очень сложно отыскать числа p и q. Если бы в вашем распоряжении был эффективный метод разложения на простые множители, вы бы с его помощью запросто нашли числа p и q; так как e может видеть кто угодно, вы бы самостоятельно вычислили d на этапе 4 RSA и получили бы закрытый ключ.

Конечно, для нахождения множителей можно начать подставлять подряд все целые положительные числа, которые меньше, чем n; именно поэтому просто необходимо, чтобы p и q были огромными. Как и в методе Диффи — Хеллмана, мы не ищем простые числа вслепую; мы можем использовать для нахождения нужных нам простых чисел алгоритм. В разделе 16.4 рассказывается об эффективном методе нахождения простых чисел.

RSA беззащитна лишь перед квантовыми вычислениями, потому что при таких расчетах возможно совершить разложение за

полиноминальное время. На данный же момент RSA обеспечивает надежную защиту.

Покуда RSA надежна и решает проблему распространения ключей, у вас может возникнуть вопрос, зачем же нам тогда вообще симметричная криптография. Ведь можно зашифровать сообщение с помощью RSA. Ответ прост: RSA намного медленней симметричной криптографии, подобной AES. Так что чаще всего RSA используется в гибридных методах. Две стороны прибегают к RSA для согласования ключа, которым они будут пользоваться с помощью AES, вместо совершения обмена ключами по методу Диффи — Хеллмана.

1. Алиса хочет послать Бобу длинное сообщение *M* (в данном случае RSA слишком медленная) и выбирает случайный ключ *K*, который она потом использует с симметричным шифром, таким как AES.
2. Алиса шифрует K с помощью открытого ключа Боба:

$$C_K = E_{P(B)}(K).$$

3. Алиса шифрует *M*, используя *K* с AES:

$$CM = E_{AES(K)}(M).$$

4. Алиса посылает (C_K, C_M) Бобу.
5. Боб с помощью своего закрытого ключа расшифровывает сообщение C_K, в котором содержится AES-ключ:

$$K = D_{S(B)}(CK).$$

6. Боб использует ключ *K* для расшифровки сообщения *M*:

$$M = D_{AES(K)}(CM).$$

Таким образом мы совмещаем лучшее из двух миров: мы добываем с помощью RSA надежные ключи для коммуникации, а затем с помощью AES шифруем кипу данных. Точно так же вместо RSA мы могли бы использовать обмен ключами Диффи — Хеллмана. Оба метода востребованы. Однако же, если нам нужно, чтобы и подпись, и шифрование были выполнены одним и тем же способом, мы обращаемся к RSA.

5.3. Хеширование сообщения

Вопрос скорости не обходит стороной и цифровую подпись. Подписание длинного сообщения M может занять много времени. В таком случае Алиса может оставить на сообщении цифровой отпечаток пальца или просто отпечаток. Отпечаток сообщения представляет собой крохотный кусочек удостоверяющих данных, мы получаем его, применяя к сообщению особую быструю функцию $h(M)$; ее еще называют профилем сообщения. Функция такова, что из любого сообщения она формирует последовательность битов небольшого фиксированного размера, скажем, 256 бит. Функция называется функцией хеширования; поэтому отпечаток можно также назвать хешом сообщения. Нам также нужно, чтобы было почти невозможно найти два сообщения с одинаковыми отпечатками, то есть такими M и M', чтобы $h(M) = h(M')$. Функции хеширования с таким свойством мы называем функциями хеширования со стойкостью к коллизиям. При таком подходе отпечаток идентифицирует сообщение, так как очень сложно создать еще одно сообщение с точно таким же отпечатком. Алиса подпишет M, пройдя поочередно все этапы, описанные ниже.

1. Алиса вычисляет отпечаток для M:

$$H = h(M).$$

2. Алиса подписывает отпечаток для M:

$$C = \mathrm{E}_{S(A)}(H).$$

3. Алиса посылает (M, $C = \mathrm{E}_{S(A)}(H)$) Бобу.
4. Боб самостоятельно вычислят $H = h(M)$.
5. Боб проверяет, что полученная подпись принадлежит Алисе:

$$H \stackrel{?}{=} D_{P(A)}(C).$$

6. Боб проверяет, что отпечаток, который он вычислил в пункте 4, идентичен подписи, которую он опознал в пункте 5.

Конечно, не бывает так, что нет коллизий, то есть разных сообщений с одинаковым отпечатком. Например, если длина сообщения 10 килобайт, то существует 2^{80000} возможных сообщений, так

как 10 килобайтов содержат 10 000 байтов или 80 000 битов. Функция хеширования связывает с этими 2^{80000} сообщениями куда меньшее количество возможных отпечатков. Если она создает сообщение длиной 256 бит, то количество возможных отпечатков — 2^{256}. Поэтому теоретически существует огромное количество коллизий: $2^{80000}/2^{256} = 2^{79744}$. Тем не менее на практике мы встретим лишь наименьшую часть из всех теоретически возможных 2^{80000} сообщений; нам нужно будет подписать сообщения только из этого самого меньшинства. Нам нужно, чтобы у $h(M)$ было как можно меньше шансов отметить два сообщения одним и тем же отпечатком, и оказывается, что существуют такие функции, но нам нужно с осторожностью подходить в выбору, останавливаясь на той, которую научное сообщество сочло достаточно устойчивой к коллизиям.

Следует отметить, что термин «хеш» почти всеобъемлющ и им обозначают схожие, но в то же время совершенно различные понятия; хеширование является основным методом сортировки и получения данных. Мы возьмемся исследовать хеширование в главе 13. В целом функция хеширования — это такая функция, которая преобразовывает данные произвольной величины в данные с фиксированным, куда меньшим объемом. Хорошая функция хеширования — та, что устойчива к коллизиям. У функций хеширования, которые мы используем для электронной подписи и отпечатков, есть дополнительное свойство — они все односторонние: получив результат функции хеширования невозможно добраться до вводных данных, которые были заданы функции хеширования. Такая хеш-функция называется криптографической функцией хеширования. Криптографической функцией хеширования с хорошей системой защиты и широким спектром применения является SHA-2*.

5.4. Анонимизация интернет-трафика

Объединение симметричной криптографии и шифрования с открытым ключом может также разрешить и другие задачи. Например совмещение RSA, AES и обмен ключами Диффи — Хеллмана

* SHA — от Secure Hash Algorithm, надежный алгоритм хеширования.

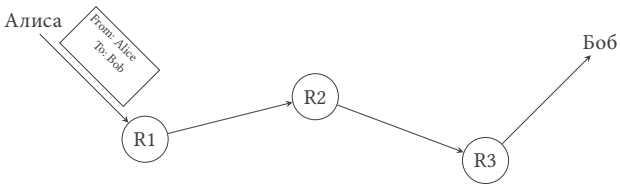
касаются анонимизации интернет-трафика. Данный случай затрагивает куда более обширную проблему, включающую в себя шифрование не только данных, но еще и метаданных. Это не просто данные сами по себе, а данные, описывающие свойства данных. В телефонном разговоре метаданными являются личности собеседников, дата и время звонка. В Интернете метаданные содержат не информацию, которой обмениваются люди, а данные о самих участниках коммуникации. Если Алиса посылает Бобу электронное письмо, то данными называется содержание этого письма; метаданные — это дата и время, отосланного письма, а также сам факт того, что Алиса послала Бобу сообщение. Если Алиса заходит на какой-то сайт, то контент сайта — это данные; само же посещение сайта Алисой в конкретный день и час — метаданные.

С помощью Диффи — Хеллмана или RSA мы можем получить надежные закрытые ключи, которыми мы шифруем содержание наших бесед. Ни одному шпиону не узнать, о чем говорят Алиса и Боб. В то же время шпион будет знать, что Алиса и Боб общаются, что само по себе может быть важной информацией. А если и не важной, то все равно, общение Боба и Алисы — их личное дело и ничье больше.

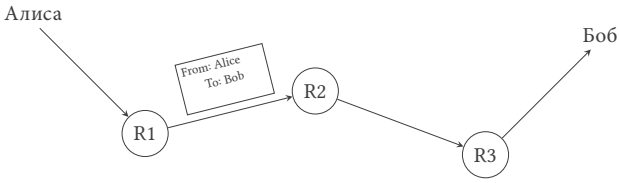
Когда Алиса заходит на сайт Боба, информация между Алисой и Бобом циркулирует в виде пакетов, перескакивающих с компьютера на компьютер, с компьютера Алисы на компьютер Боба и обратно. Каждый попадающийся на пути компьютер, перенаправляющий интернет-трафик, называется маршрутизатором и знает, что он передает данные от Алисы Бобу. Даже если общение зашифровано, шифрование охватывает содержание беседы, то есть данные. Адреса Алисы и Боба, которые содержатся в пакетах, передающих зашифрованные данные, все еще являются открытым текстом и могут быть перехвачены любым шпионом; на рисунке 5.3 изображен пример того, как путешествует сообщение от Алисы к Бобу, проходящее через три маршрутизатора.

Общение в Интернете можно анонимизировать с помощью криптографии, а точнее задумки, прозванной луковой маршрутизацией. Она работает путем заключения исходного пакета в не-

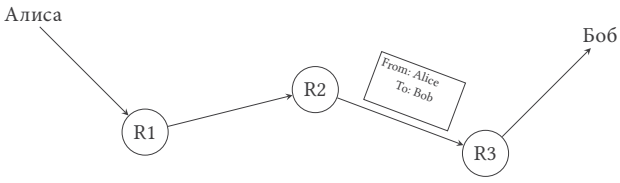
сколько наслоенных друг на друга, словно на луковице, пакетов; взгляните на рисунок 5.4.



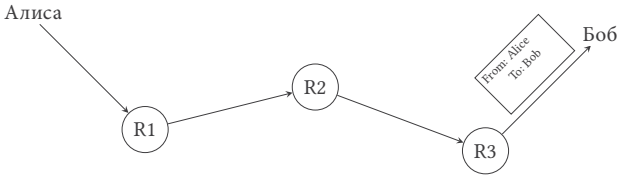
(a) Прыжок от Алисы к R1



(b) Прыжок от R1 к R2



(c) Прыжок от R2 к R3



(d) Прыжок от R3 к Бобу

Рисунок 5.3. Путь сообщения от Алисы к Бобу

Первый маршрутизатор, получивший пакет от Алисы, может считать только внешний слой, на котором указан адрес следующего маршрутизатора, куда дальше отправится пакет. Второй маршрутизатор получит пакет уже без прежнего внешнего слоя и сможет лишь считать новый внешний слой (иными словами,

второй слой из всех, укрывающих исходный пакет). Из него второй маршрутизатор узнает адрес третьего маршрутизатора, к которому нужно направить пакет. Третий маршрутизатор повторит тот же процесс, сняв еще один слой, и так будет продолжаться до тех пор, пока исходный пакет не дойдет до Боба.

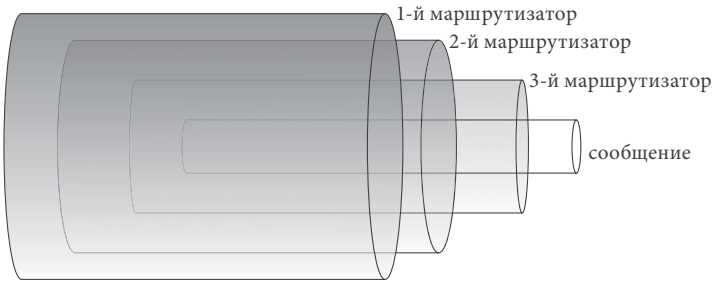


Рисунок 5.4. Укрытый слоями пакет луковой маршрутизации

Обратите внимание, что первому маршрутизатору известны только адреса Алисы и второго маршрутизатора. Второй маршрутизатор знает лишь то, что пакет получен от первого маршрутизатора и надо отправить его на адрес третьего маршрутизатора. Боб же будет знать, что получил пакет с какого-то маршрутизатора. Никто, кроме Алисы, не знает, что изначально пакет шел от Алисы к Бобу.

Как Алиса создает укрытый слоями пакет и как прокладывает путь до Боба, привлекая промежуточные маршрутизаторы, которые не знают конечного пункта назначения? Все дело в «луковой» маршрутизации, Tope*, который работает примерно так, как описано ниже.

Тор включает в себя некое количество маршрутизаторов-посредников, которые доступны Алисе. Первым делом Алиса выбирает цепочку таких маршрутизаторов, с которыми она будет работать. Их называют луковыми маршрутизаторами, или ORs**. Допустим, Алиса берет три луковых маршрутизатора: OR_1,

* Tor — The onion router, луковый маршрутизатор.

**ORs — Onion Routers.

OR_2, OR_3. Она хочет, чтобы ее сообщение проделало путь Алиса → OR_1 → OR_2 → OR_3 → Боб. Данный маршрут известен только Алисе. Сообщение будет идти от маршрутизатора к маршрутизатору, теряя на каждом из них по слою шифрования. Взгляните на рисунок 5.5, где мы полагаем, что Tor содержит ORn_3 луковых маршрутизатора, упорядоченных в виде матрицы; реальная структура Tora, конечно же, не такая, и она меняется со временем в зависимости от добавленных или убранных узлов. Во время следующей переписки с Бобом Алиса может выбрать иную цепочку маршрутизаторов, поэтому постоянно отслеживать ее общение невозможно.

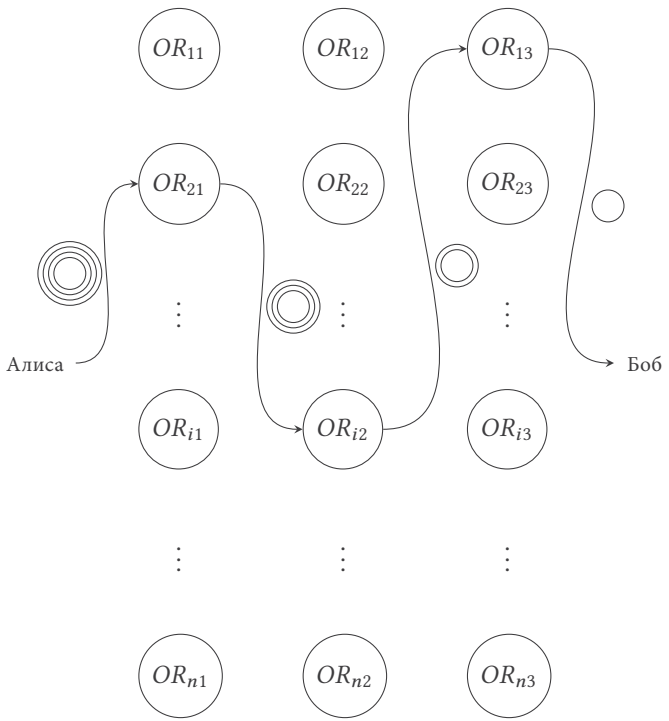


Рисунок 5.5. Алиса посылает Бобу сообщение через луковые маршрутизаторы

Сперва Алиса, используя RSA, связывается с OR_1 и отсылает инструкции по созданию маршрута коммуникации. Так как речь

идет о пакете, с помощью которого Алиса приказывает OR_1 совершить определенные действия, мы можем обозначить его как командный пакет. В нем содержится ее фрагмент обмена ключами по Диффи — Хеллману с OR_1. Вдобавок в нем содержится команда, которая сообщает OR_1, что Алиса будет помечать пакеты специальным выбранным ею идентификатором, который называется кодом цепи, мы обозначим его C_1. Назовем наш командный пакет пакетом `CreateHop`(C_1, g^{x1}), сократив в нашем описании часть с Диффи — Хеллманом. OR_1 отзывается с помощью своей фрагмента обмана ключами по методу Диффи — Хеллмана. Все сообщения, которые Алиса пошлет на OR_1, будут зашифрованы заданным ключом, скажем, DH_1.

Далее Алиса снова связывается с OR_1 и сообщает ему, что теперь он должен передать все полученные от нее сообщения на OR_2. Для этого она посылает OR_1 командный пакет, в котором содержится команда продлить маршрут и Алисин фрагмент нового обмена ключами по Диффи — Хеллману. Фрагмент Диффи — Хеллмана зашифрован с помощью открытого RSA-ключа OR_2. Весь пакет зашифрован с помощью DH_1. Давайте назовем этот командный пакет `ExtendRoute`(OR_2, g^{x2}). Когда OR_1 получает пакет, он его шифрует. Затем он создает новый пакет `CreateHop`(C_1, g^{x1}), который посылается на OR_2. Командный пакет содержит фрагмент Диффи — Хеллмана, полученного от Алисы, для OR_2 и сообщает OR_2, что он будет помечать пакеты другим кодом цепи, скажем, C_2. Он сообщает OR_2 только это, не упоминая, что сообщения от Алисы. OR_1 фиксирует факт, что пакеты, отмеченные C_1, отправятся на OR_2, а пакеты, полученные с OR_2 и отмеченные C_2, отправятся обратно к Алисе. OR_1 передает обратно Алисе ответ Диффи — Хеллмана, полученный от OR_2, таким образом, у Алисы и OR_2 есть ключ Диффи — Хеллмана, DH_2.

Чтобы проложить путь к OR_3, Алиса создает командный пакет `ExtendRoute`(OR_3, g^{x3}), который продлевает маршрут от OR_2 до OR_3. В пакете содержится фрагмент ключа Диффи — Хеллмана, которую она хочет разделить с OR_3. Фрагмент зашифрован с помощью открытого RSA-ключа OR_3. Весь пакет зашифрован с помощью DH_2 и сверх того зашифрован с помощью DH_1. Алиса посыла-

ет пакет на OR_1. Когда OR_1 получает пакет, он может расшифровать только первый слой. OR_1 знает, что ячейки данных, отмеченные C_1, должны отправиться дальше, в пункт назначения, связанный с C_2, OR_2, но не знает ничего об их содержимом. Он делает на пакете отметку C_2 и, удалив один слой, отправляет к OR_2.

OR_2 получает пакет от OR_1 и расшифровывает его с помощью DH_2, извлекая `ExtendRoute`(OR_3, g^{x3}). Как только OR_2 получает командный пакет, который продлевает маршрут и в котором содержится Алисин фрагмент обмена ключами Диффи — Хеллмана с OR_2, он выполняет все те же самые действия, что и OR_1 до него. Он создает и отправляет к OR_3 новый командный пакет `CreateHop`(C_3, g^{x3}). Пакет содержит Алисин фрагмент Диффи — Хеллмана для OR_3 и сообщает, что тот будет помечать пакеты новым кодом цепочки, скажем, C_3. OR_3 фиксирует факт, что пакеты, отмеченные C_2, отправятся к OR_3, а пакеты, полученные с OR_3 и отмеченные C_3, отправятся обратно к OR_1. OR_2 передает обратно Алисе, через OR_1, ответ Диффи — Хеллмана, полученный от OR_3, таким образом, у Алисы и OR_3 есть ключ Диффи — Хеллмана, DH_3.

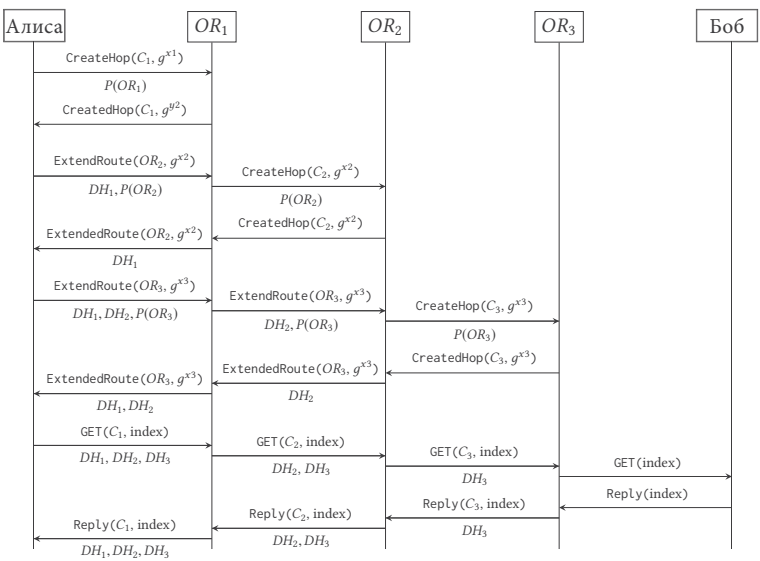


Рисунок 5.6. Обмен с помощью Тора

Теперь Алиса может послать сообщение Бобу, зная, что от посторонних глаз укрыто не только содержимое послания, но и весь маршрут. Чтобы послать Бобу сообщение, Алиса создает пакет, в котором содержится ее сообщение, адресованное Бобу и зашифрованное с помощью DH_3, который, в свою очередь, зашифрован с помощью DH_2, зашифрованного с помощью DH_1 и с отметкой C_1. Пакет сначала отправляется на OR_1. Так как на пакете отметка C_1, OR_1 знает, что его надо переправить к OR_2. OR_1 удаляет первый слой с помощью DH_1 и отправляет пакет на OR_2, оставив отметку C_2. OR_2 счищает второй слой с помощью DH_2. Он знает, что пакеты с отметкой C_2 нужно переправить к OR_3, поэтому он ставит отметку C_3 и отправляет пакет на OR_3.

OR_3 получает пакет от OR_2 и расшифровывает его с помощью DH_3. Он видит, что сообщение адресовано Бобу, поэтому он просто пересылает его получателю. Ответ Боба проделает тот же путь в точности наоборот, Боб → OR_3 → OR_2 → OR_1 → Алиса, зашифрованное с помощью DH_1, потом DH_2 и наконец DH_3 и с использованием отметок C_3, C_2, C_1. Вы можете посмотреть весь цикл на рисунке 5.6, изучая его сверху вниз и слева направо. Стрелками обозначены сообщения. Подпись сверху стрелок обозначает содержимое, а подпись снизу — примененное шифрование, Диффи — Хеллмана или RSA с открытым ключом получателя. Алиса просит веб-сервер Боба прислать ей главную страницу, и в ответ Боб выполняет запрос. Чтобы запросить главную страницу, Алисе нужно обратиться к серверу Боба с запросом GET, используя протокол передачи гипертекста (HTTP*), однако мы не будем сейчас вдаваться в подробности.

Тор также позволяет создавать ключи Диффи — Хеллмана с помощью нового протокола, не использующего RSA; новый протокол быстрее и лучше, но намного сложнее в описании. К тому же при общении двух сторон в Торе (Алиса, луковые маршрутизаторы и Боб) применяются протоколы аутентификации и шифрования. В общем, Тор представляет собой мощную базу для поддержания анонимности в Интернете.

---

* HTTP — Hypertext Transfer Protocol, протокол передачи гипертекста.

Примечания

Все материалы по криптографии смотрите в примечаниях предыдущей главы. Алгоритм RSA был описан Роном Ривестом, Ади Шамиром и Леонардом Адлеманом в 1977 году и зарегистрирован как патент США 4 405 829. В августе 1977 года Мартин Гарднер опубликовал их алгоритм в журнале «В мире науке» (Scientific American), в своей статье «Математические игры» [75]. Лишь год спустя RSA была опубликована как отдельный научный труд [165]. RSA беззащитна перед квантовыми вычислениями, а точнее алгоритмом Шора [185].

Концепция цифровых подписей описана Уитфилдом Диффи и Мартином Хеллманом в их научной работе, в которой они презентовали свою систему обмена ключами [50], однако они не касались метода подписания сообщения. Данный метод появился с алгоритмом RSA. С тех пор были представлены и другие способы подписания. Подпись Меркла [138], названная в честь Ральфа Меркла, способна противостоять квантовым вычислениям и используется в файловых системах для гарантии надежности и в одноранговых протоколах, например BitTorrent.

Надежный алгоритм хеширования SHA-2 представляет собой семейство функций хеширования, разработанных Агентством национальной безопасности (АНБ) и отмеченных как стандарт Национальным институтом стандартов и технологий (НИСТ) [207]. Семейство SHA-2 включает в себя функции хеширования, которые создают отпечатки, то есть значения хеша из 224, 256, 384 или 512 бит, и называются SHA-224, SHA-256, SHA-384 и SHA-512 соответственно; самая популярная из них — SHA-256. В августе 2015 года НИСТ обнародовал новую криптографическую функцию хеширования, SHA-3 [208], не в качестве замены SHA-2 (так как работает она прекрасно), а как альтернативу. Упомянутый алгоритм стал победителем на конкурсе функций хеширования, организованном НИСТ. Для более детального рассмотрения криптографических функций хеширования обратитесь к [167].

Ознакомиться с техническими данными Tora [52]. Недавнее обсуждение его основных принципов — здесь [81]. Изначально луко-

вая маршрутизация разрабатывалась Научно-исследовательской лабораторией ВМС США (DARPA) в середине 1990-х годов, а затем — Агентством оборонных перспективных исследовательских разработок. Научно-исследовательская лаборатория ВМС выпустила Tor с бесплатной лицензией; в 2006 году был учрежден проект Tor, который стал благотворительной организацией, занимающейся дальнейшей поддержкой и развитием Tora.

Упражнения

1. Шифрование с открытым ключом не лучше симметричного шифрования, потому что оно медленнее. Найдите библиотеку программ, реализующую RSA, и еще одну, реализующую AES, и сравните их работу с сообщениями разной длины. Убедитесь, что выбранные вами библиотеки программ реализованы на одном и том же программном языке. В частности, у скриптовых языков есть чистые реализации, написанные на конкретном скриптовом языке, а реализации, которые взаимодействуют с библиотеками, написаны на более быстром, компилируемом языке. Будет нечестным сравнивать их между собой.
2. Применение криптографических функций хеширования, таких как SHA-2, заключается в том, чтобы проверять, что раздробленные данные уже где-то сохранены. Например, вместо того, чтобы хранить файл одним цельным куском, его дробят на блоки фиксированных размеров. Блоки сохраняются и могут быть извлечены по значению хеша SHA-2. Если нужно разместить еще один файл, точно такой же или похожий на первый, то у всех или многих блоков будут те же значения хеша SHA-2, поэтому следует добавить только те блоки, значения хеша которых не соответствуют тем, что имеются. Данная основа техники складирования получила название дедубликация, так как она удаляет повторяющиеся блоки. Напишите программу, которая берет файл и размер блока, дробит файл на блоки и рассчитывает их хеши SHA-2.

6 Все по порядку

Как правило, задания, которые вам нужно выполнить, содержат какие-то условия. Задачи могут быть связаны одна с другой таким образом, что выполнение одной задачи может начаться не раньше, чем закончится выполнение другой или других. Благодаря подобным условиям в нашей жизни, мы с ранних лет узнаем, что, например, для того, чтобы чайник закипел, надо сперва набрать в него воду и воткнуть шнур в розетку. Мы можем греть воду (точнее, ее будет греть чайник) и параллельно заниматься другим делом: вставлять хлеб в тостер и ждать, пока он его поджарит. Однако мы не можем взяться готовить кофе, потому что нам надо сперва дождаться кипятка из чайника.

Такие задачи касаются не только бытовых вещей; они могут быть частью проекта, в котором требуется особая последовательность завершения дел и все дела взаимосвязаны между собой. Может, это какие-то учебно-научные задачи. Чтобы защитить степень, вам нужно сдать определенное количество дисциплин, которые зачастую требуют знания дополнительных предметов. В общем, вы можете представить задачи в виде списка дел, связанных определенной последовательностью, поэтому какие-то дела должны делаться в первую очередь, а какие-то — только после их завершения.

Классический пример порядка таких задач — одевание. Большинство людей овладевает данным умением еще в раннем возрасте (хотя «оделся» в детском возрасте не всегда значит «правильно надел все вещи»), поэтому мы считаем данное умение чем-то само собой разумеющимся. Однако если вдуматься, то, чтобы одеться, нам нужно выполнить множество операций. Зимой нужно надевать нижнее белье, носки, несколько слоев одежды, пальто, шапку, перчатки и ботинки. Мы тут можем видеть ряд особых требований, например нельзя надеть ботинки прежде носков. В то

же время имеется некая свобода действий, например вы можете сперва надеть левый ботинок, потом правый, а после них — пальто; или можно надеть правый ботинок, затем пальто, а потом уже заняться левым ботинком. Когда мы учимся одеваться, мы, помимо прочего, учимся порядку действий.

Еще один пример — мероприятие. Мы хотим связаться с неким количеством людей и что-то предложить или же о чем-то спросить; мы можем предлагать скидочный купон или пробную версию какого-нибудь продукта или же собирать пожертвования. Давно известно, что лучший способ повлиять на человеческие решения — это показать людям то, чем занимаются другие люди, особенно если этих других людей знают и уважают. Поэтому, когда мы знаем, что Боб посетит наше мероприятие, если там будет Алиса, имеет смысл сперва пригласить Алису, а потом уже Боба. Представьте, что о нашем мероприятии мы хотим известить несколько человек, отношения которых мы решили упорядочить и схематично изобразить. В результате у нас получился граф, который можно видеть на рисунке 6.1.

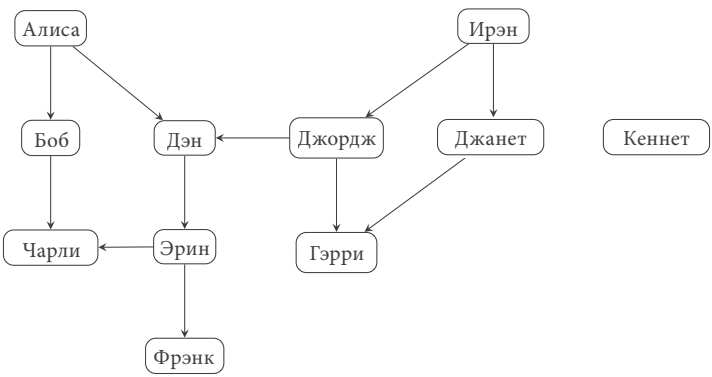


Рисунок 6.1. Граф с потенциальными участниками мероприятия

Теперь нам следует решить задачу, в каком порядке всех их известить? Нам не следует говорить с Чарли прежде, чем мы свяжемся с Бобом и Эрин. Точно так же не стоит уведомлять Боба прежде Алисы. Кеннет живет сам по себе, так что мы можем информировать его когда хотим. У задачи есть несколько решений.

На рисунке 6.2 мы изобразили два разных, но одинаково годных упорядочивания, выполненные путем перестановки людей в графе потенциальных участников мероприятия. В обоих случаях порядок отношений между людьми не нарушен, так как ни одна стрелка не уходит назад.

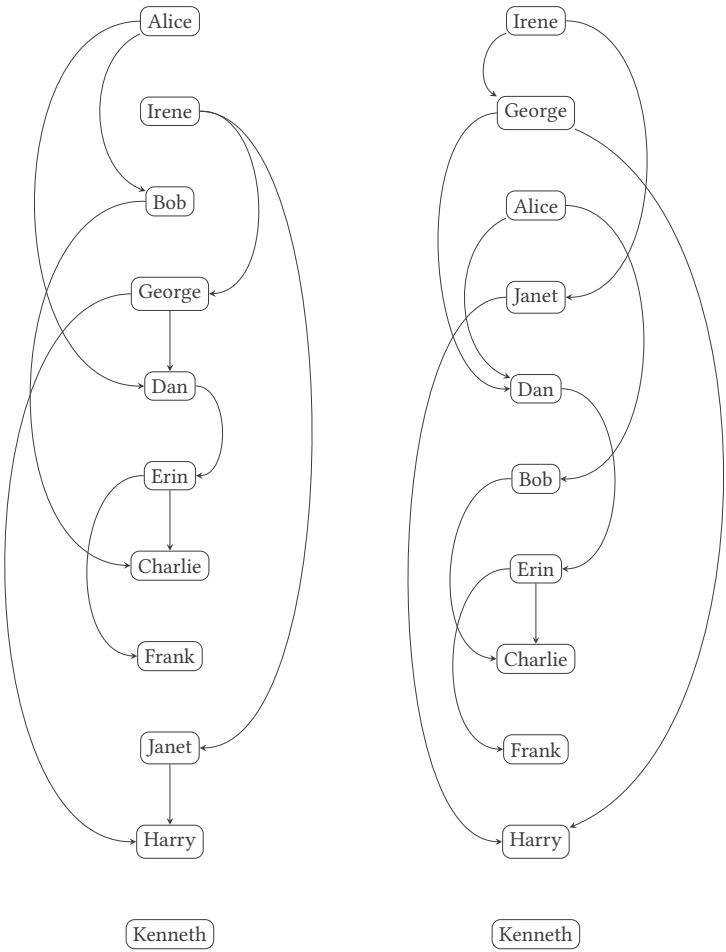
6.1. Топологическая сортировка

Мы хотим найти универсальный способ построения таких вот распределений. Выражаясь терминами графов, нам нужен особый метод сортировки ориентированных ациклических графов, или ОАГов. Топологическое упорядочивание или топологическая сортировка ориентированного ациклического графа $G = (V, E)$ представляет собой упорядочивание вершин V в графе таким образом, что для каждого ребра (u, v), которое возникает в E, вершина u появляется в упорядочивании прежде, чем вершина v.

На рисунке 6.2 показано, что у графа может быть не одна топологическая сортировка. Верно и обратное: у графа вовсе может не быть топологической сортировки. В частности, подумайте, что происходит с ориентированным циклическим графом. В нашем примере нам надо связаться сначала с Алисой и только потом с Бобом, сперва с Бобом, а потом с Чарли, и значит, с Алисой раньше, чем с Чарли. Если бы у нас было ребро от Чарли к Алисе, то получилось бы, что мы прежде Алисы хотим связаться с Чарли, и мы уперлись бы в тупик. В ориентированных циклических графах топологические сортировки не существуют, потому что от них никакого толку. Так что прежде, чем приступить к созданию топологической сортировки графа, нужно убедиться, что перед нами ациклический граф, то есть ОАГ.

Вернемся к нашему примеру, мы имеем дело с ОАГом, а значит, с ним можно работать. Вам нужна топологическая сортировка данного графа, которая будет представлять из себя упорядочивание людей, с которыми вы можете связаться только после того, как связались с людьми перед ними. Гораздо проще разрешить задачку, если следовать обратной логике: с кем вы свяжетесь в последнюю очередь? Ясное дело, с человеком, который никак не свя-

зан с другими людьми; иными словами, с человеком, у которого нет исходящих ребер, тянущихся к кому-либо.



(a) Упорядочивание (b) Еще одно упорядочивание

Рисунок 6.2. Упорядочивание контактов по порядку сверху вниз

В нашем примере такими людьми могут быть Фрэнк, Гэрри, Кеннет или Чарли. Мы нашли того, с кем свяжемся в последнюю очередь, теперь подумаем, с кем мы побеседуем до них. Вы можете снова использовать тот же подход: связаться с человеком, у которого нет связей. Если вы выбрали вашим последним контактом

Фрэнка, то вам нужно выбирать между Чарли, Гэрри и Кеннетом. Предположим, вы выбрали Чарли. В последнюю очередь вы свяжетесь с Фрэнком, до него — с Чарли. А с кем до Чарли? Мы можете выбрать Гэрри или Кеннета, а еще Боба, так как он единственный, кто связан с Чарли, который уже присутствует в топологической сортировке как предпоследний контакт.

Выбор очередного контакта, от последнего к первому, то же самое, что исследование графа в глубину: вы опускаетесь к самому нижнему узлу и проделываете от него обратный путь до места старта. Исследование графа насколько можно глубже — это то, что происходит при поиске в глубину; действительно, применение поиска в глубину к ОАГу даст нам в итоге его топологическую сортировку.

Алгоритм 6.1. Поиск в глубину для топологической сортировки

`DFSTopologicalSort(`*G, node*`)`

Вводные данные: $G = (V, E)$, ОАГ; *node*, узел в G.

Данные: *visited*, массив размером $|V|$; *sorted*, список.

Результат: *visited*[*i*] будет `true`, если до узла *i* можно дотянуться от *node*; *sorted* в самом начале содержит, в обратном порядке, тупики, в которые мы пришли при использовании поиска в глубину, начиная путь от *node*

```
1  visited[node] ← TRUE
2  foreach v in AdjacencyList(G, node) do
3      if not visited[v] then
4          DFSTopologicalSort(G, v)
5  InsertInList(sorted, NULL, node)
```

В подтверждение сказанному, в алгоритме 6.1 показан переделанный поиск в глубину; он работает так же, как обычный поиск в глубину, но вдобавок использует некоторые новые данные: *sorted* — список, в чью голову мы добавляем каждый элемент, когда при поиске в глубину натыкаемся на тупик и нам нужно вернуться обратно. Точнее, основная идея алгоритма 6.1 в том, что, когда мы заходим в тупик при поиске в глубину и начинаем обратный путь, мы находимся в строчке 5. Путем добавления

нынешнего узла — тупика в соответствующем рекурсивном вызове — во главу списка, мы заполняем список от конца к началу. Это означает, что мы заполняем наш список от конца до начала каждым встреченным нами тупиком.

Имея под рукой алгоритм 6.1, мы теперь можем подступиться к алгоритму 6.2, который реализует топологическую сортировку. По сути, этот алгоритм создает окружение для `DFSTopologicalSort`. Он инициализирует *visited* и *sorted*. Затем ему нужно только вызывать `DFSTopologicalSort` — в цикле строчек 5–7 — до тех пор, пока все узлы не окажутся посещенными. Если мы можем добраться до всех узлов из узла 0, цикл выполнится только один раз. Если же не можем, то при первом непосещенном узле цикл перезапустится, и так далее.

Алгоритм 6.2. Топологическая сортировка для ОАГ

`TopologicalSort`(G) $\rightarrow$ *sorted*

Вводные данные: $G = (V, E)$, ОАГ.

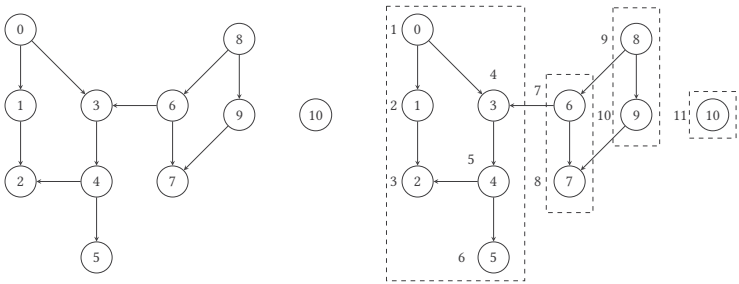
Выводимые данные: *sorted*, список размером $|V|$ с узлами графа, расположенными в топологическом порядке.

```
1  visited ← CreateArray(|V|)
2  sorted ← CreateList()
3  for i ← 0 to |V| do
4      visited[i] ← FALSE
5  for i ← 0 to |V| do
6      if not visited[i] then
7          DFSTopologicalSort(G, i)
8  return sorted
```

Что касается выполнения, то, так как алгоритм действительно совершает обход графа в глубину и обход графа требует $\Theta(|V|+|E|)$ времени, топологической сортировке тоже потребуется $\Theta(|V|+|E|)$ времени.

Давайте посмотрим, как работает алгоритм. Прежде чем запустить алгоритм, мы раздаем узлам уникальные индексы. Если мы раздаем индексы в алфавитном порядке, то Алиса станет 0, Боб станет 1 и так далее. Таким образом, граф на рисунке 6.1 станет

эквивалентным графу на рисунке 6.3a. На рисунке 6.3(b) изображен проход графа в глубину. Числа рядом с узлами указывают порядок посещения. Пунктирными прямоугольниками выделены посещенные за каждый вызов `DFSTopoloticalSort` узлы.



(a) Людям присвоены численные индексы

(b) Исследование в глубину графа, представляющего список людей

Рисунок 6.3. Граф нумерованного списка людей и его исследование в глубину

Так как граф не сильно связный, при его обходе первыми посещаются узлы, до которых можно добраться из узла 0. Затем мы посещаем узлы, доступные из узла 6, за исключением узла 3, в нем мы уже побывали, поэтому посещаем только узел 7. Затем идем к узлу 8 и проходим к единственному непосещенному соседу, узлу 9. Последним мы посещаем узел 10. Таким образом, цикл алгоритма 6.2 выполнен четыре раза. Мы проверили непосещенные узлы и все соседние узлы, посещая их в порядке возрастания, как показано в строчке 5, но это совсем не обязательно. Алгоритм работает корректно с любым заданным для исследования узлов порядком.

Давайте рассмотрим алгоритм 6.2 поближе, чтобы лучше увидеть, что происходит во время каждой итерации цикла. В первой итерации мы вызываем `DFSTopoloticalSort` для узла 0. Помните, что после посещения всех соседей любого узла, мы знаем, что нынешний узел — последний из оставшихся узлов до которых мы хотим добраться. Итак, мы начинаем с узла 0, затем через узел 1 попадаем в узел 2, видим, что узел 2 — тупик, и значит, добавляем его в начало (пустого) списка *sorted*. Потом возвращаемся в узел 1,

видим, что у него больше нет непосещенных соседей, поэтому мы добавляем узел один в начало *sorted*. У узла 0 есть непосещенный узел 3, так что мы идет в узел 3, а из него — в узлы 4 и 5. Узел 5 тупиковый, поэтому мы добавляем его в начало *sorted*. Затем мы добавляем в начало *sorted* узел 4, узел 3 и, наконец, узел 0. К концу первого цикла наш список *sorted* = [0, 3, 4, 5, 1, 2]. Теперь мы вызываем `DFSTopoloticalSort` для узла 6 и получаем *sorted* = [6, 7, 0, 3, 4, 5, 1, 2]. Затем вызываем `DFSTopoloticalSort` для узла 8 и получаем *sorted* = [8, 9, 6, 7, 0, 3, 4, 5, 1, 2]. В последний раз мы вызываем `DFSTopoloticalSort` для узла 10 и получаем *sorted* = [10, 8, 9, 6, 7, 0, 3, 4, 5, 1, 2].

На рисунке 6.4 изображен порядок посещения каждого узла и их топологический порядок. Топологическая сортировка для нашего графа 10 → 8 → 9 → 6 → 7 → 0 → 3 → 4 → 5 → 1 → 2. Заменим индексы именами и получим Кеннет → Ирен → Джанет → Джордж → Гэрри → Алиса → Дэн → Эрин → Фрэнк → Боб → Чарли. Можете посмотреть на рисунок 6.5 и убедиться, что ни одна стрелка не идет вверх, а значит, наше решение верно. К тому же данное решение отличается от решений, изображенных на рисунке 6.2.

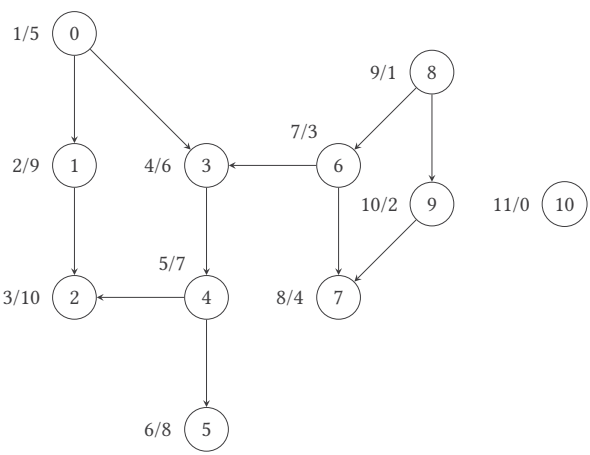


Рисунок 6.4. Обход в глубину графа, отображающего список людей, с топологической сортировкой; отметка *i*/*j* рядом с узлом означает, что во время обхода в глубину узел посетили в *i* порядке, а *j* указывает место узла в топологической сортировке

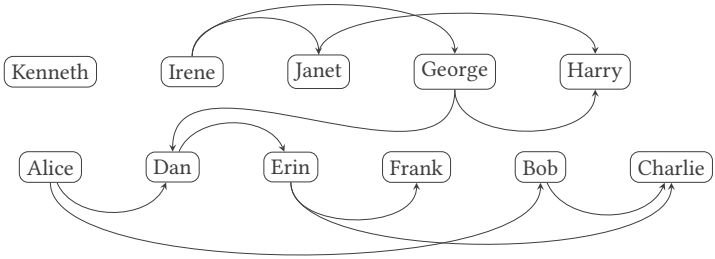


Рисунок 6.5. Топологическая сортировка для графа со списком людей; порядок просмотра слева направо и сверху вниз

Мы можем применять топологическую сортировку везде, где нужно выстроить задачи в особом порядке. В нашем примере мы хотели связаться с людьми, но задача может быть любой. Например, это могут быть процессы, на которые влияют определенные ограничения, поэтому какие-то процессы должны быть выполнены раньше остальных. В книгах в самом начале часто встречаются указатели, своего рода навигационные карты, которые подсказывают читателю, в каком порядке лучше всего знакомиться с материалом и в каком порядке читать главы. Термины в глоссариях могут идти в алфавитном порядке, а могут располагаться так, что сперва будут встречаться только те из них, с помощью которых определяются другие термины.

6.2. Взвешенные графы

До этого мы пользовались графами, которые отображали элементы в виде узлов, а связи между ними — в виде ребер. Мы можем расширить графы и назначить для каждого ребра число, называемое весом. Такие графы называются взвешенными графами. Они представляют собой общий вид невзвешенных графов, так как невзвешенный граф можно представить в виде взвешенного графа, в котором все ребра имеют одинаковый вес, например тот, который мы рассмотрели. Мы отметим вес ребра (u, v) с помощью $w(u, v)$. Взвешенные графы полезны, так как позволяют нам отобразить больше информации. Если представить граф в виде до-

рожной сети, веса будут отображать расстояния или время, потраченное на путь из одного пункта в другой.

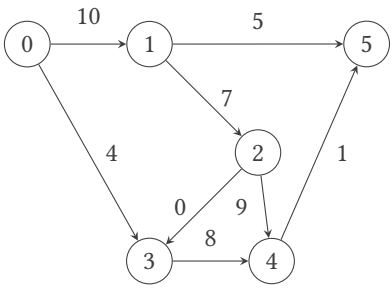


Рисунок 6.6. Взвешенный граф

На рисунке 6.6 изображен взвешенный ориентированный граф. Он может показаться похожим на невзвешенный неориентированный, который мы видели прежде, в главе 2 на рисунке 2.3. Все веса на рисунке являются неотрицательными целыми числами, но это вовсе не обязательно. Веса могут быть как положительные, так и отрицательные: узлы могут обозначать достижения, а веса — награды или штрафы. Веса также могут быть действительными числами; в зависимости от требований к их применению. Во взвешенном графе, когда мы говорим о расстоянии между двумя узлами, мы подразумеваем не количество ребер между узлами, а сумму всех весов этих ребер. Поэтому на рисунке 6.6 расстояние от 0 до 2 равно не 2, а 17.

Мы можем представить взвешенный граф точно так же, как и невзвешенный. В матрице смежности для взвешенного графа в качестве элементов выступают веса ребер или особое значение для несвязанных элементов. Таблица 6.1 показывает матрицу смежности для графа с рисунка 6.6. При отсутствии связей мы используем в качестве особого значения ∞, хотя точно так же можно прибегнуть к любому другому значению, которое не может являться весом. Если наши веса неотрицательные, мы можем использовать –1, или `null`, или что-нибудь еще. Как говорилось выше, это зависит от требований области применения.

Таблица 6.1. Матрица смежности для графа с рисунка 6.6

	0	1	2	3	4	5
0	∞	10	∞	4	∞	∞
1	∞	∞	7	∞	∞	5
2	∞	∞	∞	0	9	∞
3	∞	∞	∞	∞	8	∞
4	∞	∞	∞	∞	∞	1
5	∞	∞	∞	∞	∞	∞

Как вариант мы можем использовать представление списка смежности. В каждом узле списка мы храним не только название его вершины, но также вес соответствующего ребра. Представление списка смежности для взвешенного графа рисунка 6.6 можно увидеть на рисунке 6.7.

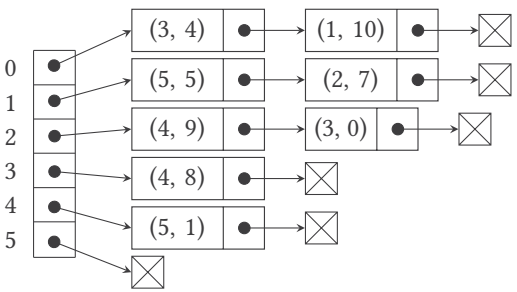


Рисунок 6.7. Представление списка смежности для взвешенного графа с рисунка 6.6

6.3. Критические пути

Если мы занимаемся организацией дел, то сталкиваемся с еще одной проблемой, похожей на топологическую сортировку, — с нахождением критического пути в ОАГе, который представляет поэтапное описание процесса. В данной проблеме процесс, который требуется завершить, представлен в виде графа, где узлы — это задачи, а связи — отображение порядка между задачами процесса. Каждому ребру (u, v) мы приписываем вес $w(u, v)$, являющийся временем, за которое выполняется задача u, после чего может начаться выполнение задачи v. На рисунке 6.8 изображен

такой вот граф планирования. Узлам соответствуют отдельные задачи (числом от нуля и выше), а весам соответствуют единицы времени, например недели, требуемые для завершения одного задания и начала другого.

Взгляните на рисунок 6.8, нам потребуется 17 недель, чтобы перейти от задачи 0 к задаче 1. И тут возникает вопрос: какое наименьшее время потребуется на выполнение всех задач?

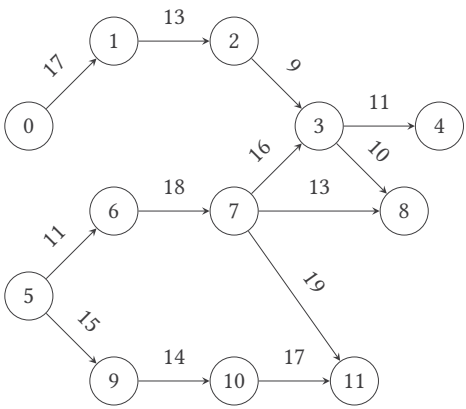


Рисунок 6.8. Граф распределения задач

Некоторые задачи могут выполняться параллельно, например мы можем приступить к задачам 0 и 5. Точно так же с задачами 1, 6 и 9, они могут выполняться, не ожидая завершения друг друга, после того, как будут готовы задачи 0 и 5. Однако не все задачи могут выполняться параллельно: к задаче 3 можно приступить только после задач 2 и 7. Принимая во внимание возможность параллельного выполнения задач, на какое минимальное время для завершения всего процесса мы можем рассчитывать?

Чтобы узнать ответ, мы сделаем следующее. Сперва мы добавим в граф два дополнительных узла. Стартовый узел, *s*, с которого, мы полагаем, начнется весь процесс. Назовем узел *s* исходным узлом или истоком. Мы соединяем *s* напрямую с остальными узлами графа, чтобы обозначить задачи, с которых мы начнем выполнение процесса. Добавленные нами ребра имеют нулевой вес. Мы добавляем еще один узел, *t*, который, как мы полагаем, будет

концом всего процесса. Назовем узел *t* узлом стока или стоком. Мы напрямую соединяем все узлы с узлом *t*. Вес ребер опять-таки равен нулю. В итоге у нас получился рисунок 6.9.

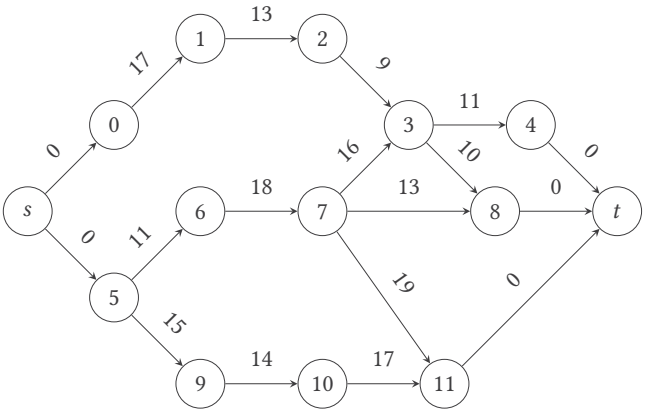


Рисунок 6.9. Граф распределения задач с исходным узлом и узлом стока

Мы добавили исходный узел и узел стока, и теперь наша проблема в следующем: какой путь от *s* до *t* самый длинный? Так как нам нужно посетить все узлы графа, нам придется проследовать по каждому пути от *s* до *t*. Какие-то пути мы можем пройти одновременно, параллельно. Как бы то ни было, мы никак не можем завершить процесс до тех пор, пока мы не пройдем путь, требующий наибольшего количества времени. Путь, требующий наибольшего количества времени, называется критическим путем. Небольшой пример такого пути изображен на рисунке 6.10.

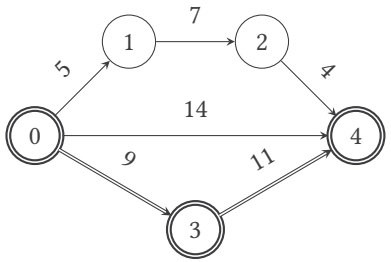


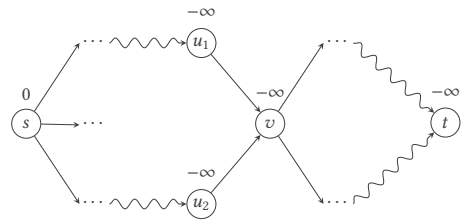
Рисунок 6.10. Пример критического пути

Путь 0 → 1 → 2 → 4 имеет длину 16; путь 0 → 4 имеет длину 14, а путь 0 → 3 → 4 имеет длину 20. Если длина — это недели, то мы можем приступить к задаче 4 не раньше, чем через 20 недель, поэтому путь 0 → 3 → 4 является критическим путем; мы обозначаем узлы данного пути удвоенными ребрами и границами. Точно так же в графе на рисунке 6.9, мы не можем приступить к задаче *t*, которая является всего лишь отметкой для завершения процесса, раньше, чем истечет время, требуемое на самый длинный путь от *s* до *t*.

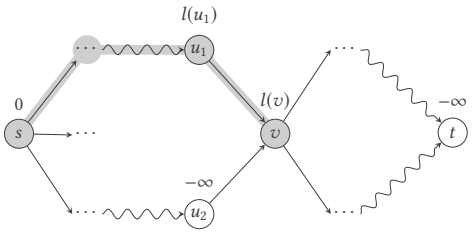
Как нам найти наидлиннейший путь? Мы должны исследовать граф по порядку. А точнее, нам надо начать с *s*, посетить каждый узел графа и в каждом узле рассчитать расстояние от *s* до данного узла. Изначально нам известно только одно расстояние, до *s*, и оно равно нулю. Мы будем посещать узлы и по мере продвижения будем обновлять информацию о пройденном расстоянии. Предположим, мы пришли в узел *u*, мы нашли наибольшее расстояние, от *s* до *u*, а *u* связан с узлом *v*. Если наибольшее расстояние, которое мы нашли от *s* до *v* больше расстояния от *s* до *u* плюс вес ребра (*u*, *v*), то нам больше нечего делать. Однако если наибольшее расстояние, которое мы нашли от *s* до *v*, меньше или равно расстоянию от *s* до *u* плюс вес ребра (*u*, *v*), то нам надо зафиксировать информацию, что наибольшее расстояние, которое мы нашли до *v*, идет через *u*, и, соответственно, обновить посчитанное расстояние. Так как мы собрались обновлять расстояние всякий раз, когда ищем самое большое, то в самом начале мы присваиваем всем неизвестным нам расстояниям, то есть всем расстояниям, за исключением расстояния до *s*, наименьшее из всех возможных значений, которое равно $-\infty$. Чтобы понять, что мы имеем в виду, взгляните на рисунок 6.11(a), где изображен граф с двумя путями, идущими от *s* к *v*; с помощью $l(u)$ мы обозначаем расстояние от *s* до узла *u*.

Мы обходим граф и посещаем узел u_1. Это первый попавшийся нам узел, который связан с *v*, поэтому мы фиксируем факт, что длиннейший путь до *v*, который нам довелось измерить на данный момент, идет через u_1, и равно $l(v) = l(u_1) + w(u_1, v)$; взгляните на рисунок 6.11(b). После, во время нашего прохода по графу, мы посещаем узел u_2. Мы замеряем длину пути от *s* до *v*, который проходит через u_2, $l'(v) = l(u_2) + w(u_2, v)$, и сравниваем с расстоянием,

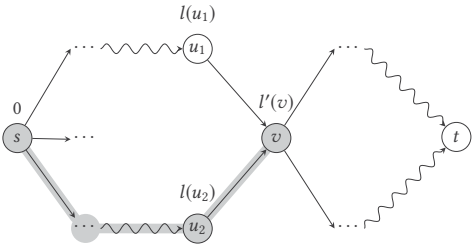
которое мы замерили ранее. Если оно больше, мы фиксируем, что самый длинный путь тот, что идет через u_2, как изображено на рисунке 6.11(c). Если есть еще один узел, u_3, который связан с v, то мы делаем еще одно сравнение и, если нужно, обновляем при посещении узла u_3 данные о расстоянии. Процесс проверки и, при необходимости, обновления расчетов в системе показателей графа путем внесения новых более точных данных, — типичная вещь во многих графовых алгоритмах, у этого процесса даже есть название: релаксация. Наша предварительная оценка длины пути, –∞,утрирована и релаксирует, то есть уменьшается, до более адекватного значения каждый раз, когда мы обновляем длину пути.



(a) Граф с двумя путями до *v*



(b) Первый проход до *v*



(c) Второй проход до *v*

Рисунок 6.11. Обновление данных о наибольшем расстоянии

При первой проверке пути до v нам обязательно нужно узнать наименьшее возможное значение для расстояния $-\infty$, чтобы мы потом смогли корректно обновить данные о длине пути, проходящего через u_1. Именно поэтому мы изначально присваиваем общему расстоянию значение $-\infty$, исключением является только путь до s, который равен 0, для уверенности, что все будет работать, как показано на рисунке 6.11(а). К тому же необходимо, чтобы мы посетили узел v прежде, чем обойдем все узлы, ведущие к v. Для этого мы обойдем граф в топологическом порядке. Алгоритм 6.3 объединяет в себе все указанные условия. Мы полагаем, что, как и прежде, мы приписываем узлам уникальные индексы, поэтому, когда вы видите s, его стоит воспринимать его как целое число, отличное от прочих узловых индексов.

Алгоритм 6.3. Критический путь

`CriticalPath`$(G) \rightarrow (pred, dist)$

Вводные данные: $G = (V, E, s)$ взвешенный ОАГ со стартовым узлом s.

Выводимые данные: *pred*, такой массив размером $|V|$, что $pred[i]$ является предшественником узла i с критическим путем от s до i; *dist*, такой массив размером $|V|$, что $dist[i]$ содержит длину критического пути от s до i.

```
1  pred ← CreateArray(|V|)
2  dist ← CreateArray(|V|)
3  for i ← 0 to |V| do
4      pred[i] ← −1
5      dist[i] ← −∞
6  dist[s] ← 0
7  sorted ← TopologicalSort(V, E)
8  foreach u in sorted do
9      foreach v in AdjacencyList(G, u) do
10         if dist[v] < dist[u] + Weight(G, u, v) then
11             dist[v] ← dist[u] + Weight(G, u, v)
12             pred[v] ← u
13 return (pred, dist)
```

Мы используем две структуры данных: массив *pred* и массив *dist*. Элемент i в *pred*, то есть $pred[i]$, отмечает узел, предшествующий i в критическом пути, который мы нашли до узла, соответ-

ствующего *i*. Элемент *i* в *dist*, то есть *dist*[*i*], содержит длину критического пути, который мы нашли до узла, соответствующего *i*. Мы также используем функцию `Weight`(*G*, *u*, *v*), которая возвращает вес между узлами *u* и *v* графе *G*. Обратите внимание, что алгоритму 6.3 не нужно *t* в качестве входных данных. У нас достаточно данных, чтобы знать, что это за узел; длина критичного пути задана с помощью *dist*[*t*], и, чтобы найти путь, мы отслеживаем *pred*, начиная с *pred*[*t*].

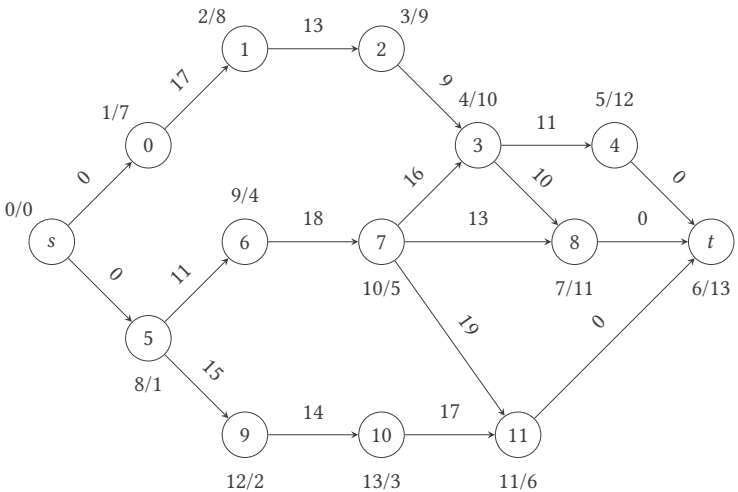


Рисунок 6.12. Граф распределения задач с топографическим порядком. Как и на рисунке 6.4, отметка *i*/*j* рядом с узлом означает, что во время обхода в глубину узел посетили в *i* порядке, а *j* указывает место узла в топологической сортировке

В строчках 1–6 мы создаем и инициализируем структуры данных *pred* и *dist*. Мы обозначаем каждый элемент из *pred* как ошибочный, несуществующий узел, названный –1, а расстоянию до каждого узла, то есть каждому элементу из *dist*, присваиваем значение –∞, за исключением исходного узла *s*, значение которого равно 0. Затем мы выполняем топологическую сортировку графа. В строчках 8–12 мы сортируем каждый узел в топологическом порядке. Поочередно для каждого узла мы проходимся по его соседям путем проверки списка смежности.

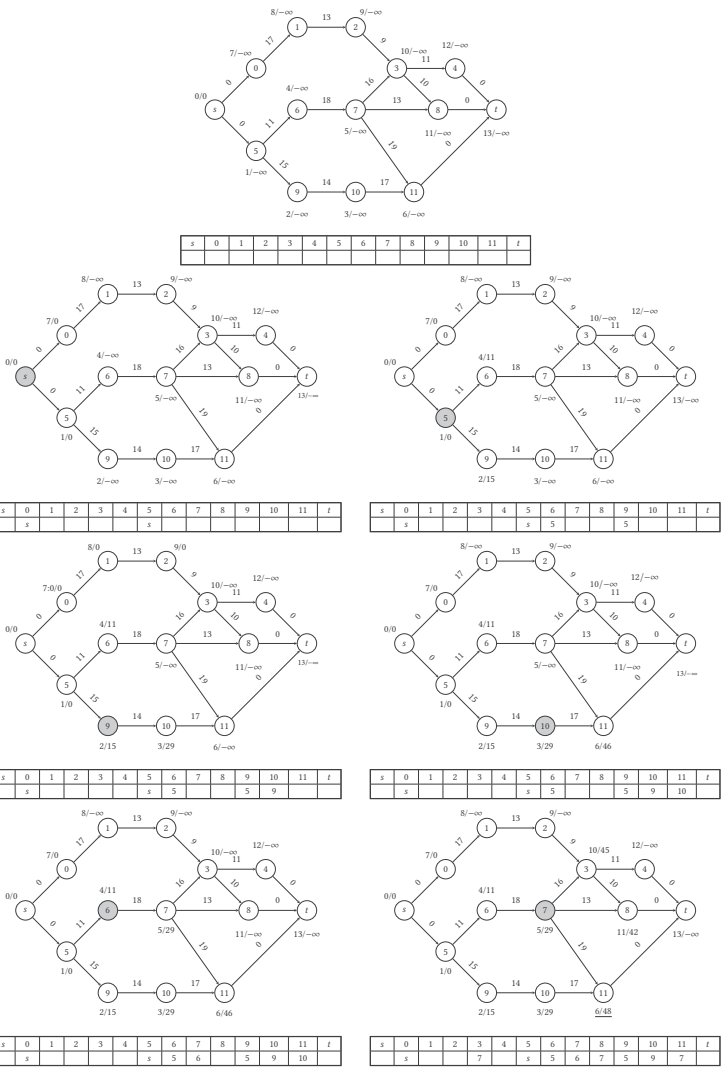


Рисунок 6.13. Нахождение критического пути. Отметка *i*/*j* рядом с узлом означает, что узел имеет топологический порядок *i* и расстояние *j*. Матрица под каждым изображением показывает *pred* с пустыми элементами вместо –1, чтобы уменьшить лишние скопления

В строчках 10–12 мы смотрим, длиннее ли путь до каждого соседнего узла, проходящий через нынешний узел, чем первичный

путь, который мы рассчитали до этого. Если он длиннее, мы обновляем данные о наибольшем расстоянии, убираем из них предыдущий узел в самом длинном пути и заменяем его на нынешний узел. Так работает релаксация на практике. В алгоритме 6.3 мы сперва релаксируем с $-\infty$ до длины, полученной в первом узле, а затем, как только находим бóльшую длину, мы релаксируем до нее.

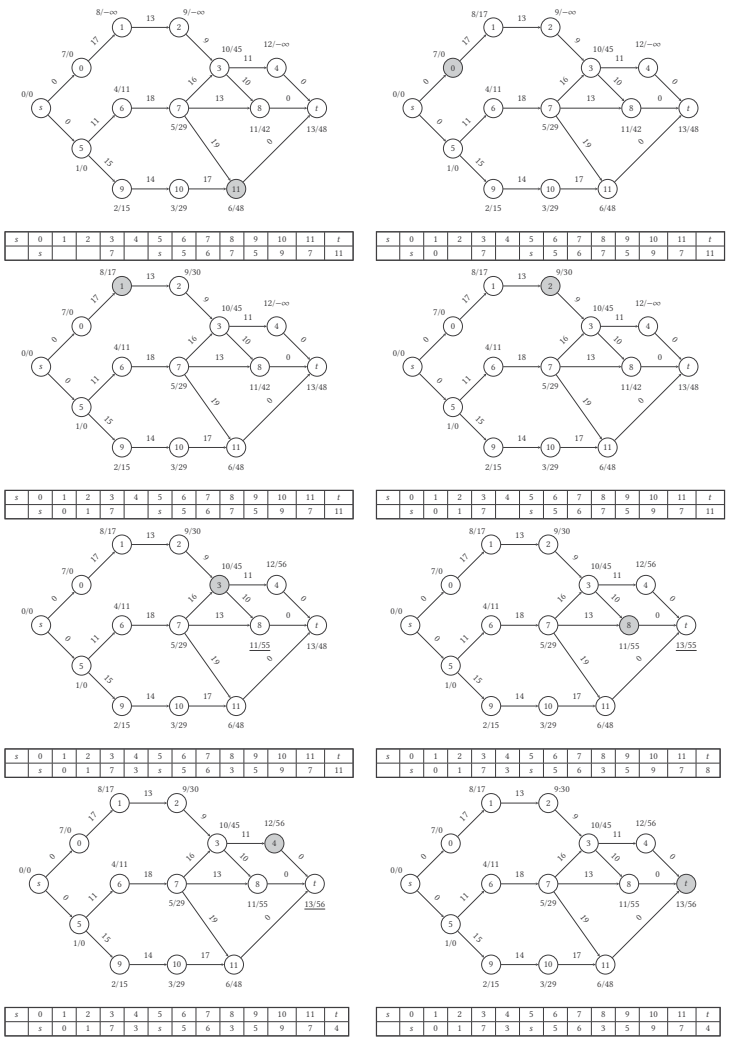


Рисунок 6.14. Нахождение критического пути (продолжение)

Данный алгоритм работает на ура. Топологическая сортировка занимает $\Theta(|V|+|E|)$ времени. Цикл в строчках 3–5 занимает $\Theta(|V|)$ времени. Затем цикл в строчках 8–12 проходит через строчки 11–12 по разу на каждое ребро: каждое ребро релаксируется по одному разу. Это значит, что весь цикл требует $\Theta(|E|)$ времени. В целом, весь алгоритм выполняется за $\Theta(|V|+|E|+|V|+|E|) = \Theta(|V|+|E|)$ время.

Примером работы алгоритма для графа на рисунке 6.9 служат рисунки 6.13 и 6.14. Серым цветом мы помечаем наш нынешний узел в топологическом порядке. Посмотрите, что происходит, когда мы посещаем узел 7, в котором обновляем расстояние до узла 11, вычисленное до этого. То же самое повторяется, когда мы посещаем узлы 3, 8 и 4, где результаты релаксации сказываются на рассчитанной ранее длине пути. Матрица под каждым графом на приведенных двух рисунках отображает содержание массива *pred*. Как мы говорили, *s* и *t* — на самом деле индикаторы узла, поэтому у нас может быть $s = 12$ и $t = 13$, но куда лучше нарисовать матрицу, где слева *s*, а справа *t*. В конце выполнения алгоритма мы получаем критический путь, следуя к *pred*[*t*], который равен 4; затем мы идем к *pred*[4], который равен 3, и так далее. Таким образом, критический путь представляет собой

$$s \to 5 \to 6 \to 7 \to 3 \to 4 \to t$$

Примечания

Дональд Кнут [112] дает алгоритм для топологической сортировки — не тот, что мы рассматривали здесь. Рассмотренный нами алгоритм принадлежит Кормену, Лейзерсону, Ривесту и Штайну [42]; его предшественник был описан Робертом Тарьяном [198].

В области проектной техники моделирования использование критических путей называется методом критического пути, а также лежит в основе метода оценки и пересмотра программы. В терминах графов нахождение критического пути — это то же самое, что нахождение наибольшего расстояния.

Упражнения

1. Поиск в глубину и топологическая сортировка не требуют от нас исследования узлов графа в особом порядке. Реализуйте алгоритм топологической сортировки таким образом, чтобы исследование узлов происходило в порядке возрастания или же убывания, и затем проверьте результаты. Убедитесь, что они верны, хотя могут отличаться друг от друга.
2. Топологическая сортировка не применима к графам с циклами, так как топологический порядок просто невозможен в подобных графах. Когда мы не уверены, ацикличный ли у нас граф, хорошо бы уметь отлавливать циклы в ходе выполнения алгоритма топологической сортировки и выдавать результаты. Оказывается, это довольно легко сделать. При поиске в глубину мы следуем по ребрам как можно дальше. Если мы сохраняем узлы, которые присутствуют в нашей рекурсивной очереди вызовов, то есть в стеке рекурсивных вызовов, то, если узел, который мы собираемся посетить, находится среди узлов нынешней рекурсивной очереди вызовов, мы знаем, что наткнулись на цикл. Например, на рисунке 6.4, когда мы находимся в узле 2, стек рекурсивных вызовов содержит узлы 0, 1 и 2 (в таком порядке). Если бы существовало ребро от 2 до 0 или от 2 до 1, тогда мы получили бы цикл. Мы бы тут же нашли его, проверив, что 0 или 1 содержатся в нашем стеке рекурсивных вызовов. Переделайте `DFSTopologicalSort` так, чтобы она обнаруживала циклы и сообщала о своих находках.
3. Чтобы найти критический путь в графе G, нам надо добавить исходный узел s и узел стока t. Мы не описали способ, как это делается, но все довольно просто. Если в нашем графе, пронумерованном от 0 до $|V|-1$, исходно $|V|$ узлов, мы добавляем в граф еще два узла: s будет пронумерован как $|V|$, а t — как $|V|+1$. Затем мы ищем узлы, на которые не указывают другие узлы: они будут соединены с s. Далее мы находим узлы, которые не указывают на другие узлы: они будут соединены с t. Реализуйте данную систему так, чтобы у вас получилась программа, которая выполняет всю процедуру нахождения критических путей.

7 Строки, абзацы, пути

Текст, который вы сейчас читаете, состоит из набора букв, объединенных в слова, сгруппированных в строки и разбитых на абзацы. Когда вы пишете обычное, бумажное письмо, вы начинаете выводить строку и, когда доходите до края листа, стараетесь либо уместить слово на той же строке, либо, если возможно, переносите его на следующую.

То же самое и с текстовыми редакторами. Когда вы печатаете текст, программа будет выстраивать слова в строку, пока та не закончится. Затем, когда вы набираете слово, которое не влезает в строку, текстовый редактор должен решить, что с ним делать. Он может оставить слово на той же строке, сократив расстояние между словами, и тогда вся строка немного сожмется, чтобы вместить последнее слово. Можно перенести все слово целиком на следующую строчку и увеличить расстояние между предыдущими словами. Если ни то, ни другое вас не устраивает, потому что текст выглядит слишком сжатым или слишком растянутым, текстовый редактор попробует переносить слова по частям, оставляя на нынешней строчке начало слова и убирая его конец на следующую.

Разбивка текста на строки называется разрывом строки; мы без труда проделываем ее, когда пишем письма от руки, а вот текстовому редактору каждый раз приходится решать, как поступить с той или иной строкой. На рисунке 7.1(a) можно увидеть применение данного метода разрыва строки на примере первых абзацев сказки братьев Гримм «Король-лягушонок». Данный прием разрывает строки, как только их объем достигает тридцати символов (включая пробелы и знаки препинания). Абзацы выглядят читабельным, но теперь взгляните на рисунок 7.1(b). Те же самые абзацы, но теперь мы применили иной метод разрыва строки. Абзацы стали на две строки короче, все лишние пустоты убрались, и текст выглядит куда аккуратней.

Если задуматься, описанный нами метод видит каждую строку как самостоятельную единицу. В то же время мы видим, что можно достичь лучших результатов, если при разрыве строк рассматривать все строчки абзаца как единое целое. При таком подходе возможно создать удобные для чтения абзацы, потому что во главу угла ставится содержимое абзацев, а не разрыв строки при первом удобном случае.

«В старые годы, когда стоило лишь пожелать чего-нибудь и желание исполнялось, жил-был на свете король; все дочери его были одна краше другой, а уж младшая королевна была так прекрасна, что даже само солнышко, так много видавшее всяких чудес, и то дивилось, озаряя ее личико. Близ королевского замка был большой темный лес, а в том лесу под старой липой вырыт был колодец. В жаркие дни заходила королевна в темный лес и садилась у прохладного колодца; а когда ей скучно становилось, брала она золотой мячик, подбрасывала его и ловила: это была ее любимая забава».

(a) Абзац поделен на строки

«В старые годы, когда стоило лишь пожелать чего-нибудь и желание исполнялось, жил-был на свете король; все дочери его были одна краше другой, а уж младшая королевна была так прекрасна, что даже само солнышко, так много видавшее всяких чудес, и то дивилось, озаряя ее личико. Близ королевского замка был большой темный лес, а в том лесу под старой липой вырыт был колодец. В жаркие дни заходила королевна в темный лес и садилась у прохладного колодца; а когда ей скучно становилось, брала она золотой мячик, подбрасывала его и ловила: это была ее любимая забава».

(b) Абзац поделен на строки, более читабельный вариант

Рисунок 7.1. Разрыв строк в абзацах

Добиться подобного результата поможет осмысление различных способов разбивки абзаца на строки и присвоение числового значения для каждой строки. Числовое значение должно соотно-

ситься с актуальностью разрыва строки. Чем меньше значение, тем желаннее данный разрыв строки; чем больше значение, тем больше нам хочется избежать подобного разрыва строки и мы хотим поискать другие варианты.

> «В старые годы, когда стоило лишь пожелать чего-нибудь и желание исполнялось, жил-был на свете король; все дочери его были одна краше другой, а уж младшая королевна была так прекрасна, что даже само солнышко, так много видавшее всяких чудес, и то дивилось, озаряя ее личико. Близ королевского замка был большой темный лес, а в том лесу под старой липой вырыт был колодец. В жаркие дни заходила королевна в темный лес и садилась у прохладного колодца; а когда ей скучно становилось, брала она золотой мячик, подбрасывала его и ловила: это была ее любимая забава».

Рисунок 7.2. Разбитый на строки абзац, который описан на рисунке 7.3

На рисунке 7.3, который соотносится с рисунком 7.2, вы можете видеть данный процесс в действии. Слева указаны номера строк. Для каждой строки мы обдумываем возможные места разрыва, чтобы абзац оставался определенной ширины. Первая строка может быть оборвана в двух местах, «lived» и «a». Числа возле ребер рисунка отражают то, насколько уместно каждое место разрыва. Вторая точка разрыва лучше первой. В третьей строке только одна возможная точка разрыва. В восьмой и девятой строках четыре возможных точки разрыва. В последней же строчке, конечно, только один вариант.

На рисунке 7.3 на самом деле изображен граф, чьи узлы отображают возможные точки разрыва. У ребер есть веса, которые оценивают, насколько уместен в данном случае разрыв строки; они рассчитаны с помощью особого алгоритма, описывать который здесь ни к чему. Нам нужно лишь знать, что для каждой строки алгоритм производит число, которое показывает, насколько хорошо она смотрится; чем кривее смотрится строка, тем

выше числовое значение. Нам нужно, чтобы в нашем абзаце было поменьше «неприглядностей». Таким образом, проблема верстки нашего абзаца, который должен быть приятен глазу, превращается в задачу на прохождение графа сверху донизу и нахождение в нем кратчайшего пути, при том что путь здесь является суммой весов всех пройденных ребер. Чтобы понять на примере, что подразумевается под неприглядностью, взгляните на рисунок 7.4. Первая строка получилась в результате выбора левой ветки узла «much», расположенного в третьей строчке рисунка 7.3; вторая же строка приведенного текста получилась в результате выбора правой, выделенной ветки.

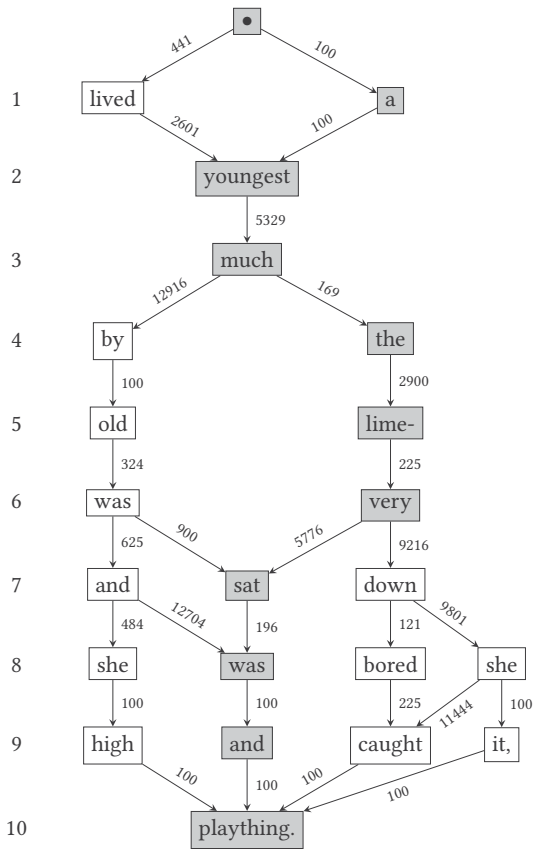


Рисунок 7.3. Оптимизация разрывов строк

Может быть, вас так и подмывает сказать, что разница невелика, однако она очень значима, когда речь идет о качестве публикации. Метод, описанный нами только что, разработан Дональдом Кнутом и Майклом Плассом и реализован в виде программы набора, широко используемой в научных публикациях; эта программа называется TEX. Название имеет греческий корень τεχ, который означает искусство и технологии; поэтому X в слове TEX — это не «икс», а греческая «хи» и читается соответственно. Система набора документов называется LATEX и является ответвлением TEX; с его помощью набран данный текст. Напомним, что X в LATEX произносится как «х». Поисковые службы, кажется, давно осознали, что запросы, связанные с LATEX, не имеют никакого отношения к каучуку.

том лесу под старой липой вырыт был колодец. Близ королевского замка был большой темный лес, а в том лесу под старой липой вырыт был колодец. Близ королевского замка

Рисунок 7.4. Примеры неприглядности текста

7.1. Кратчайшие пути

Перенос слова, или разбивание абзаца на строки, — лишь одно из множества применений задачи о кратчайшем пути, в которой мы пытаемся отыскать в графе наименьшее расстояние от стартового узла (истока) до узла назначения (стока). Задача о кратчайшем пути — одна из самых распространенных в алгоритмах; множество задач окружающего мира по сути является задачами о кратчайшем пути, и в зависимости контекста существуют самые разные алгоритмы ее решения. Построение дорожного маршрута — самая явная из всех задач о кратчайшем пути, когда вам нужно как можно быстрее добраться из пункта отправления в пункт назначения, проезжая через различные города. Кратчайший путь может обозначаться как общее расстояние, которое вам нужно покрыть, или же — с точки зрения времени — как минимально

требуемое время, за которое вы посетите все города на пути к вашему пункту назначения.

Чтобы рассмотреть задачу более подробно, представьте, что вам нужно проложить маршрут на карте, представленной в виде сетки, где точки пересечения — это перекрестки, а оценочные данные показывают время, нужное на преодоление пути между перекрестками. Сетка на карте может быть обычной, ровной, если в городе оказывается геометрически ровная дорожная сеть; случается и по-другому, однако на ровной сетке проще объяснить, что мы имеем в виду. Взгляните на рисунок 7.5, на нем как раз изображена такая сеть. Мы решили не давать названия узлам, чтобы не загромождать рисунок; веса ребер обозначены числами, им могут соответствовать минуты, требуемые на прохождение пути от одного узла к другому. Если нам нужно указать на какой-то конкретный узел, мы назовем его координаты (i, j), где i — это ряд, а j — столбец; и ряды, и столбцы отсчитываются с нуля.

Чтобы разобраться с задачей о построении маршрута из точки отправления в точку назначения, мы обратимся к более общей задаче, касающейся нахождения в нашей сетке всех кратчайших путей от исходного до конечного пунктов. Речь идет о задаче кратчайшего пути с единственным источником. Для ее решения нам понадобится метод релаксации. Изначально нам известно лишь расстояние до стартовой точки: оно равно нулю. Мы вводим наши оценочные данные для всех расстояний и задаем им наибольшее из возможных значений, ∞. Затем мы выбираем узел с наименьшими данными. Вначале у нас есть только стартовый узел с оценочными данными ноль. Мы исследуем все смежные с ним узлы. Находим вес ребра, соединяющего нынешний узел с каждым из соседних узлов, и если он меньше нынешних оценочных данных, то мы корректируем информацию и отмечаем, что предыдущий узел в кратчайшем пути является нынешним узлом, в котором мы находимся. Когда мы все сделали, берем следующий узел с наименьшими оценочными данными и релаксируем его смежные узлы. Работа закончена, когда больше не осталось узлов для исследования. Вы можете проследить процесс на ри-

сунках 7.6 и 7.7, на которых мы находим кратчайшие пути из верхнего левого угла до каждого другого узла графа.

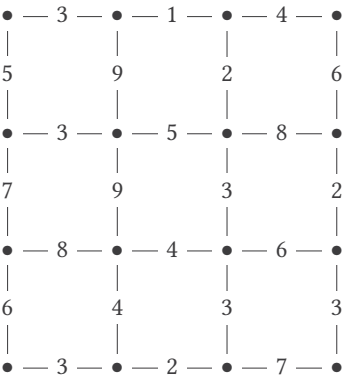


Рисунок 7.5. Дорожная сетка

Мы начинаем с определения затрат на путь до узла в верхнем левом углу и получаем ноль. Затем мы выбираем узел с наименьшими оценочными данными пути; им становится узел в верхнем левом углу, узел (0, 0). Мы посещаем его, закрашиваем черным цветом и определяем затраты на путь до его соседей. Для узла (0, 1) они равны трем, а для узла (1, 0), расположенного под стартовым, — пять. Мы отмечаем кратчайший путь до этих узлов жирной линией. На данный момент самые низкие данные у узла (0, 1), поэтому мы сперва посетим его; отмечаем данный узел серым цветом. Мы посещаем узел (0, 1) и вычисляем оценочные данные для его соседей. Мы продолжаем процедуру до тех пор, пока не останется узлов, которые тянутся к соседям и чьи оценочные данные нужно проверить.

Обратите внимание, что происходит, когда мы посещаем узел (1, 1), на пятом изображении в рисунке 7.6. У нас уже есть данные для узла (1, 1), 12, полученные при посещении узла (0, 1). Однако теперь, если мы следуем к узлу (1, 1) из узла (1, 0), у нас есть кратчайший путь с длиной 8; мы подчеркиваем узел (1, 2), отмечая его, и обновляем наш маршрут. То же самое случается на рисунке 7.7, когда мы обновляем маршрут для узлов (3, 0) и (3, 3). В конце алгоритма путь от (0, 0) до (3, 3) имеет длину 18 и проходит следующим образом: (0, 0) → (1, 0) → (2, 0) → (2, 1) → (2, 2) → (3, 2) → (3, 3).

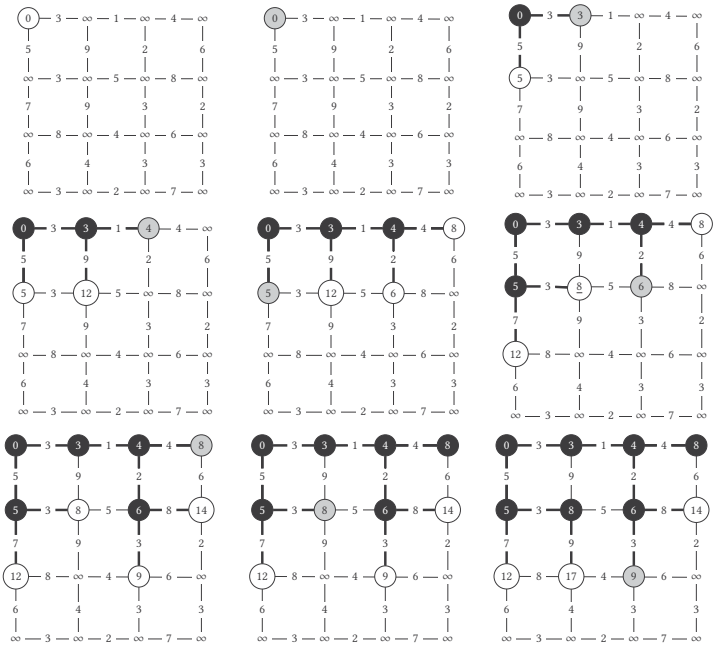


Рисунок 7.6. Кратчайшие пути из верхнего левого угла

Кратчайшие пути от нашего стартового узла к каждому узлу графа формируют дерево, которое изображено на рисунке 7.8. Дерево, чьи узлы являются узлами графа и чьи ребра являются подмножеством ребер графа, называется связующим деревом. Связующие деревья много где применяются, особо востребованы связующие деревья с минимальным весом, то есть сумма весов их ребер минимальна по сравнению с другими способами выбора ребер, формирующими связующее дерево. Такие связующие деревья называются минимальными связующими деревьями. Обратите внимание, что метод кратчайшего пути, который мы описали, выдает связующее дерево, хотя не обязательно минимальное.

7.2. Алгоритм Дейкстры

Способ, которым мы вычисляли кратчайшие пути, похож на алгоритм Дейкстры, названный в честь голландского программиста

Эдсгера Дейкстры, который открыл его в 1956 году и опубликовал в 1959. Алгоритм 7.1 описывает данный метод в алгоритмической форме. Он берет в качестве вводных граф и стартовый узел графа и возвращает два массива.

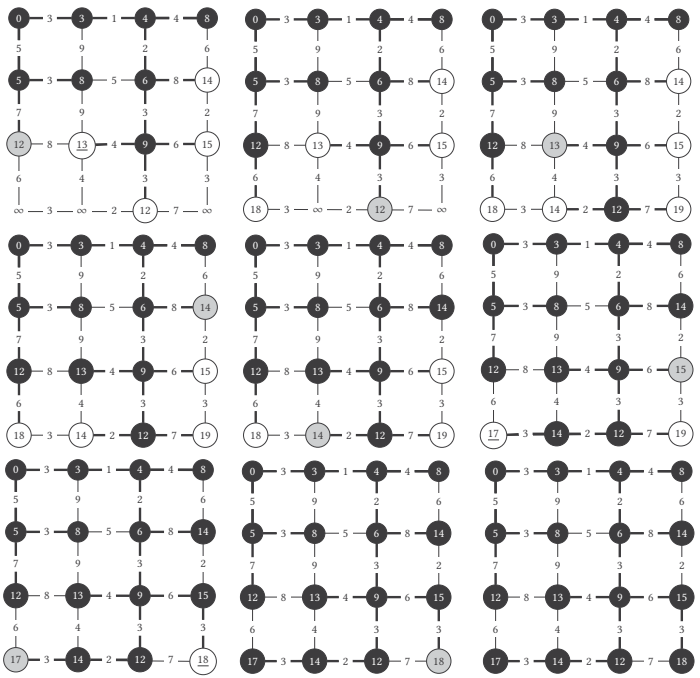


Рисунок 7.7. Кратчайший путь из верхнего левого угла (продолжение)

Алгоритм использует минимальную очередь с приоритетом, чтобы отслеживать, какой узел мы посетим следующим. Очередь с приоритетом должна поддерживать операции, описанные в разделе 3.2, но с некоторыми изменениями, соответствующими требованиям алгоритма Дейкстры. В очередь с приоритетом мы будем добавлять узлы и расстояния. Также нам нужно на месте обновлять расстояние до узла, находящегося в очереди с приоритетом. Это означает, что требуются следующие операции.

- `CreatePQ`() создает новую, пустую очередь с приоритетом.
- `InsertInPQ`(*pq*, *n*, *d*) добавляет узел *n* и расстояние *d* в очередь с приоритетом.

- `ExtractMinFromPQ`(*pq*) вынимает из очереди и возвращает узел с минимальным расстоянием.
- `izePQ`(*pq*) возвращает число элементов в очередь с приоритетом *pq*.
- `UpdatePQ`(*pq*, *n*, *d*) обновляет очередь с приоритетом путем изменения хранящегося в ней расстояния, связанного с *n*, на расстояние *d*.

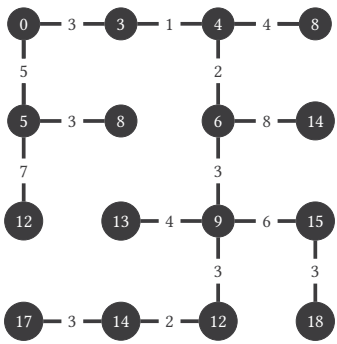


Рисунок 7.8. Связующее дерево дорожной сети

Алгоритм использует две дополнительных структуры данных: массив *pred* и массив *dist*. Элемент *i* в *pred*, то есть *pred*[*i*], отмечает узел, который идет до *i* в кратчайшем пути, который мы нашли от узла *s* до узла, отвечающего *i*. Элемент *i* в *dist*, то есть *dist*[*i*], содержит длину кратчайшего пути, который мы нашли до узла, отвечающего *i*. Строчки 1–10 алгоритма инициализируют эти структуры данных, поэтому ни у какого узла нет предшественника (узла –1 не существует). К тому же единственный известный нам кратчайший путь — путь до стартового узла, равный нулю; поэтому в строчке 7 всем прочим узлам изначально присваивается значение ∞, за исключением стартового узла, чье расстояние в строчке 9 устанавливается на 0. Затем в строчке 10 мы добавляем в очередь с приоритетом нынешний узел цикла, *v*. Цикл в строчках 11–17 повторяется до тех пор, пока не опустеет очередь с приоритетом (строчка 11), он вынимает из нее наименьший элемент (строчка 12) и расслабляет оценочные данные

пути для каждого смежного узла, который получает лучшее значение пути во внутреннем цикле строчек 13–17. Точнее, он проверяет в строчке 14, больше ли расстояние от нынешнего узла u до соседа v, чем расстояние, которое мы получили бы, выбрав путь, что кончается связкой (u, v). Если больше, расстояние до v заменяется расстоянием данного пути (строчка 15), и предыдущий узел v в нашем пути заменяется узлом u (строчка 16). Если значение пути релаксируется, нужно убедиться, чтобы очередь с приоритетом обновилась, что и происходит в строчке 17. В завершение алгоритм возвращает два массива, *pred* и *dist* (строчка 18).

Алгоритм 7.1. Алгоритм Дейкстры

`Dijkstra`$(G, s) \rightarrow (pred, dist)$

Вводные данные: $G = (V, E)$, граф; s, исходный узел.

Выводимые данные: *pred*, такой массив размером $|V|$, что $pred[i]$ является предшественником узла i в кратчайшем пути от s; *dist*, такой массив размером $|V|$, что $dist[i]$ является длиной кратчайшего пути, вычисленной от узла s до узла i.)

```
1  pred ← CreateArray(|V|)
2  dist ← CreateArray(|V|)
3  pq ← CreatePQ()
4  foreach v in V do
5      pred[v] ← −1
6      if v ≠ s then
7          dist[v] ← ∞
8      else
9          dist[v] ← 0
10     InsertInPQ(pq, v, dist[v])
11 while SizePQ(pq) ≠ 0 do
12     u ← ExtractMinFromPQ(pq)
13     foreach v in AdjacencyList(G, u) do
14         if dist[v] > dist[u] + Weight(G, u, v) then
15             dist[v] ← dist[u] + Weight(G, u, v)
16             pred[v] ← u
17             UpdatePQ(pq, v, dist[v])
18 return (pred, dist)
```

Обратите внимание, что каждый узел вынимается из очереди только раз. Так происходит, потому что, когда мы извлекаем узел из очереди с приоритетом, все смежные с ним узлы получат данные кратчайшего пути, длина которых будет равна длине пути до извлекаемого нами узла. На самом деле с начала работы алгоритма и до самого ее конца все узлы будут получать кратчайший путь, равный длине рассматриваемого нами пути. Поэтому узел никак не сможет вернуться в очередь с приоритетом до тех пор, пока не выполнится условие в строчке 14.

В алгоритме 7.1 есть один небольшой момент. В строчке 14 мы добавляем вес ребра к оценочным данным пути. Что происходит, если оценочные данные пути равны ∞? Такое случится, если до узла ниоткуда нельзя дотянуться, поэтому, когда мы достаем его из очереди с приоритетом, его оценочные данные пути остаются все же, начальным значением. Тогда мы добавим что-нибудь к бесконечности. В математике так можно; добавляя что-то к бесконечности, мы получаем бесконечность. Однако во многих программных языках бесконечности нет. Обычно вместо бесконечности берется самое огромное число. Если мы прибавим к нему число, мы не получим бесконечность; в большинстве случаев мы просто перенесемся в начало и получим результат добавления к наименьшему отрицательному числу. Тест в строчке 14 проверит $dist[v]$ на отрицательность, найдет ее, и алгоритм станет выдавать абракадабру. Поэтому при перенесении строчки 14 из теории на практику — из алгоритмического образца в рабочую реализацию — может потребоваться проверка $dist[u]$, что он не ∞, только после этого к нему можно будет что-либо добавлять. Подробнее о таких аспектах компьютерной арифметики мы расскажем в разделе 11.7.

Возможно ли, что мы пропустили кратчайший путь до узла? Представьте, что мы вынимаем из очереди с приоритетом узел v со оценочными данными кратчайшего пути $dist[v]$. Если данные не корректны, значит, существует еще один путь, который мы не нашли и который короче уже найденного. Как говорилось выше, во всех расчетах путей, в которых фигурируют узлы, еще не изъятые из очереди с приоритетом, не будет кратчайших путей. Поэто-

му должен существовать путь до узла *v*, который использует уже изъятые узлы и который короче того, что мы уже нашли. Этот путь должен заканчиваться в узле *v*. Если в этом пути *u* был узлом, предшествующим *v*, то при извлечении *u* мы бы нашли данный путь, так как у нас было бы $dist[u] + \texttt{Weight}(g, u, v) < dist[v]$. Так что пропустить кратчайший путь нельзя; следовательно, каждый раз, когда мы вынимаем узел из очереди с приоритетом, кратчайший путь до него уже найден. В ходе выполнения алгоритма узлы, извлеченные из очереди, — это узлы, для которых мы нашли соответствующие кратчайшие пути.

Так что алгоритм Дейкстры можно использовать для нахождения кратчайшего пути до одиночного узла. Пока узел вынут из очереди с приоритетом, кратчайший путь до него найден, поэтому, если нам нужно знать только кратчайший путь, мы можем остановить алгоритм и получить результат.

Давайте представим, что мы реализуем очередь с приоритетом в виде простого массива. Тогда `InsertInPQ`(pq, n, d) равно заданному $pq[n] \leftarrow d$, которому требуется постоянное время, то есть $O(1)$. Так как массив содержит фиксированное количество элементов со времени его создания, нам нужно отслеживать количество элементов, добавляемое в очередь с приоритетом путем приращения индикатора каждый раз, когда мы вызываем `InsertInPQ`. `UpdatePQ`(pq, n, d) тоже равно заданному $pq[n] \leftarrow d$ и так же требует $O(1)$ времени. `ExtractMinFromPQ`(pq) требует поиска по всему массиву, как если бы он не был отсортирован, и нам надо пройтись по нему до самого конца, чтобы найти наименьший элемент. Для n элементов в очереди с приоритетом требуется $O(n)$ времени. Так как мы не можем на самом деле достать элемент из массива, вполне достаточно установить значение на ∞ и затем уменьшать значение индикатора, оставляя число элементов в очереди. Даже если изъятие не настоящее, ведет оно себя как вполне настоящее и не влияет на точность алгоритма. В разделе 3.3 мы видели, что очереди с приоритетами могут быть реализованы как кучи. Это вовсе не обязательно; если нас не волнует скорость выполнения, то мы всегда можем реализовать ее в виде простого массива, помня, что с таким подходом для нахождения

наименьшего элемента нам потребуется несколько больше времени, чем с реализацией в виде кучи.

Вся инициализация алгоритма в строчках 1–10 выполняется $|V|$ раз; так как все операции в ней занимают постоянное время, ей требуется $O(|V|)$ времени. В строчке 12 алгоритм извлекает каждый узел как наименьший элемент только раз, поэтому у нас есть $|V|$`ExtractMinFromPQ`(pq) операций, каждая занимает $O(|V|)$ времени, и всего $O(|V|^2)$ времени. Последовательность релаксаций в строчках 14–17 выполняется самое большое $|E|$ раз, по разу на каждое ребро в графе, так что у нас максимально $|E|$`UpdatePQ`(pq, n, d) операций, которым требуется $O(|E|)$ времени. Итого алгоритм Дейкстры выполняется за $O(|V|+|V|^2 + |E|) = O(|V|^2)$. Можно использовать более удобные очереди с приоритетами, сокращая время до $O((|V| + |E|)\lg|V|)$. Если до всех узлов графа можно добраться от исходного узла, тогда количество узлов не может превышать количества ребер, поэтому в подобных графах алгоритм Дейкстры выполняется за до $O(|E|\lg|V|)$ времени.

Алгоритм Дейкстры работает и с ориентированными, и с неориентированными графами. Цикличность не имеет значения, так как цикл лишь прибавит длину пути, поэтому кратчайшего пути не получится. В то же время от отрицательных весов ребер так просто не отмахнешься; у вас могут быть отрицательные веса, когда веса отображают не реальные расстояния, а какие-то другие величины, которые могут иметь как положительные, так и отрицательные значения. Корректное выполнение алгоритма зависит от нахождения путей с увеличивающейся длиной. Если у нас будут отрицательные веса, мы не сможем уверенно сказать, что все пути, которые мы рассчитаем в будущем, не окажутся короче того, который мы уже вычислили, когда вынимали узел из очереди с приоритетом. В двух словах: если у вас в графе отрицательные веса, алгоритм Дейкстры вам не подходит.

Посмотрим на граф рисунка 7.9. Кратчайший путь от узла 0 до узла 3 — это путь 0 → 1 → 2 → 3, общей длиной 5 – 4 + 1 = 2. Тем не менее взгляните на работу алгоритма Дейкстры на рисунке 7.10. На нем видно, что, так как мы уже вынимали узел 2 из очереди с приоритетом, при обновлении данных кратчайшего пути от

узла 1 узел 2 нельзя достать повторно. Следовательно, кратчайший путь к узлу 3 никогда не изменится или не найдется.

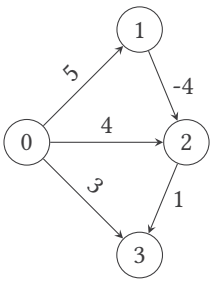


Рисунок 7.9. Граф с отрицательными весами

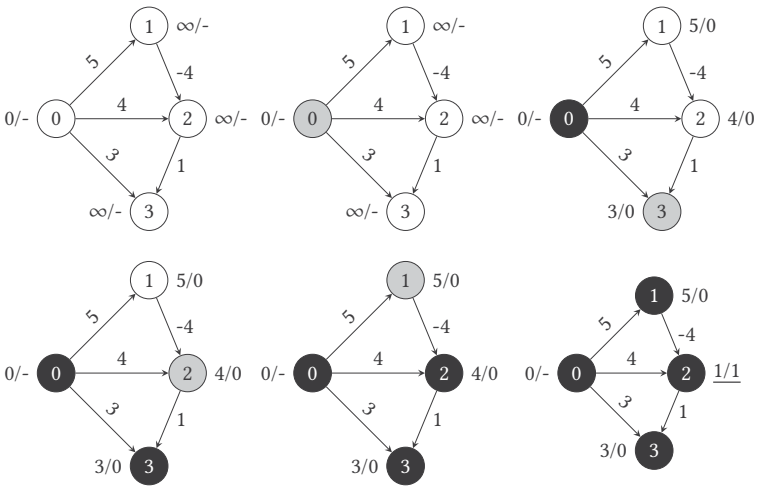


Рисунок 7.10. Работа алгоритма Дейкстры в графе с отрицательными весами. Отметка *d*/*p* рядом с узлом означает, что *d* — это значение пути, а *p* — предыдущий узел

Может показаться, что все это очень просто решается путем добавления константы к весу каждого ребра, чтобы у всех ребер в итоге был положительный вес. Не сработает. Рисунок 7.11 демонстрирует результаты подобного перераспределения весов графа. Тут мы видим, что кратчайший путь от узла 0 до узла 3 изменился, и теперь он 0 → 3, так что мы не можем прибегнуть к подобной

трансформации, чтобы получить же результаты. Причина в том, что мы добавили четыре не только одному ребру на пути от узла 0 до узла 3, но и трем ребрам на пути от узла 0 до узла 3, проходящем через узлы 1 и 2, что спутало все взаимосвязи длин и путей.

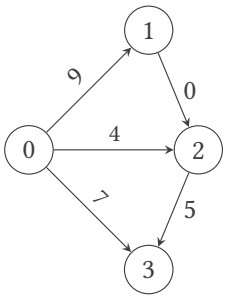


Рисунок 7.11. Граф с перераспределенными весами

Если не брать в расчет данное ограничение, алгоритм Дейкстры можно использовать для получения всякой другой дополнительной информации о графе. Прежде всего для расчета всех кратчайших путей от одного выбранного узла в любой другой узел графа; это называется задачей о кратчайшем пути между всеми парами вершин. Вам всего лишь надо выполнить алгоритм Дейкстры $|V|$ раз, начиная каждый раз с разных узлов, как показано в алгоритме 7.2. Для хранения результатов мы используем два массива с $|V|$ элементами, *preds* и *dists*, чьи элементы указывают на массивы с результатами, которые выдает `Dijkstra`; поэтому каждый раз, когда мы вызываем `Dijkstra` для узла *u*, мы присваиваем *preds*[*u*] и *dists*[*u*] массивы *pred* и *dist*, которые возвращает `Dijkstra`.

При расчете кратчайших путей между всеми парами узлов в графе вы можете вычислить метрику в топологии всего графа. Диаметр графа — это длина самого длинного из кратчайших путей между двумя узлами. Если в графе есть два узла, *v* и *u*, с кратчайшим путем между ними длиною *d* при том, что кратчайший путь, соединяющий любую другую пару узлов меньше, чем *d*, то самый длинный из кратчайших путей, по которому мы можем пройти в графе, имеет ту же длину, что и путь от *v* до *u*.

В то время как под диаметром фигуры подразумевается наибольшее расстояние между двумя точками этой фигуры, в графе, если его представить как фигуру, а его узлы — как точки, путь от v до u является его диаметром, как у фигуры, отсюда и название. Чтобы вычислить диаметр графа вы сперва запускаете алгоритм кратчайших путей между всеми парами, затем просматриваете результаты в поисках длиннейшего из выданных путей — и вот он, диаметр графа.

Алгоритм 7.2. Кратчайшие пути между всеми парами вершин

`AllPairsShortestPaths`(G) $\rightarrow$ (*preds*, *dists*)

Вводные данные: $G = (V, E)$, граф.

Выводимые данные: *preds*, такой массив размером $|V|$, что *pred*[u] является массивом с предшественниками, получившимся в результате вызова `Dijkstra` для узла u; *dists*, такой массив размером $|V|$, что *dist*[i] является массивом с расстояниями, получившимся в результате вызова `Dijkstra` для узла u.

```
1  preds ← CreateArray(|V|)
2  dists ← CreateArray(|V|)
3  foreach u in V do
4      (preds[u], dists[u]) ← Dijkstra(G, u)
5  return (preds, dists)
```

Диаметр графа дает нам наиогромнейшее количество связок, которыми мы должны исчертить весь граф, чтобы связать в нем все пары узлов. Такой подход схож с так называемой теорией шести рукопожатий, согласно которой любые два человека на Земле разделены шестью или менее людьми, знающими друг друга. Именно про это и говорят «мир тесен». Как ни странно, но выяснилось, что узлы в огромных сетях в среднем куда ближе друг к другу, чем нам кажется.

Примечания

Алгоритм разрыва строки, описанный в данной главе и применяемый в TEX и LATEX, принадлежит Кнуту и Плассу [117]. TEX бес-

подобно расписан в TEXbook [111]. LATEX был создан Лесли Лэмпортом и описан в [120].

Эдсгер Дейкстра открыл алгоритм кратчайшего пути, названный в его честь в 1956 году и опубликованный в 1959 [51]. В интервью Дейкстра рассказал об обстоятельствах, сопутствовавших его разработкам: «Как быстрее всего добраться из Роттердама в Гронинген? Для ответа на этот вопрос и нужен алгоритм кратчайшего пути, который я придумал всего-то за двадцать минут. Как-то раз мы с моей невестой все утро ходили по амстердамским магазинам, устали, а потому решили отдохнуть в кафе. Устроились на террасе, заказали кофе, и я подумал, а смогу ли я вычислить кратчайшую дорогу. Тогда я взял и придумал алгоритм, определяющий кратчайший путь. Все так, я разработал его за двадцать минут» [141].

Алгоритм Дейкстры может работать более эффективно, если мы целесообразно воспользуемся реализациями очереди с приоритетом [71]. Есть и другие способы улучшения работы алгоритма Дейкстры, в их основе лежит использование других структур данных вместо очередей с приоритетом, реализованных в виде куч: посмотрите алгоритмы, предложенные Ахуджей, Мельхорном и Тарьяном [1], Торупом [201] и Раманом [163].

Алгоритм Дейкстры можно рассматривать как частный случай более общего алгоритма, называемого поиском по первому наилучшему совпадению, который к тому же может оказаться более эффективным [60]. Еще одним обобщением алгоритма Дейкстры является алгоритм A*, разработанный Хартом, Нильсоном и Рафаэлем в 1968 году [87] (оснащенный более весомыми доказательствами в 1972 году [88]); A* широко применим, и его чаще всего используют для нахождения пути в компьютерных играх.

Так как алгоритм Дейкстры можно использовать для нахождения кратчайшего маршрута в дорожной сети между городами, его стали применять в сетевых протоколах маршрутизации, чья задача состоит в том, чтобы перемещать пакеты данных между узлами сети. К таким протоколам относятся открытый протокол предпочтения кратчайшего пути (OSPF) [147] и протокол связи между промежуточными системами (IS-IS) [34].

Самый известный научный прорыв, связанный с теорией шести рукопожатий, совершил Стэнли Милгрэм, в 1960-х годах [139], [204]; пьеса Джона Гуэйра «Шесть степеней отчуждения» и одноименный фильм, снятый по ней, изложили данную теорию простым языком.

Упражнения

1. В алгоритме 7.1 мы добавляем в очередь с приоритетом все узлы графа в ходе этапов инициализации. Если очередь с приоритетом осуществляет операцию, проверяющую есть ли в ней элементы, тогда это не обязательно. Когда мы берем узел *v*, соседствующий с узлом *u*, и обнаруживаем (во время релаксации), что наилучший путь проходит через *u*, мы можем проверить, содержится ли *u* в очереди с приоритетом. Если да, мы можем обновить данные, как в алгоритме 7.1; если же нет, мы можем тут же добавить узел *u* в очередь. Переделайте `Dijkstra` так, чтобы она работала с данным изменением.
2. Напишите две реализации алгоритма Дейкстры; пусть первая будет использовать массив для очереди с приоритетом так же, как мы описывали выше, а вторая реализует очередь с приоритетом в виде кучи. Сравните выполнение этих двух реализаций.
3. Сегодня довольно легко проверить диаметры окружающих нас графов, так как вокруг целая куча разных общедоступных сетей; выберите одну, определите ее диаметр и посмотрите, насколько в данном случае верна теория шести рукопожатий. Если вы взяли большой граф, вам нужно будет убедиться в эффективности ваших реализаций.

8 Маршрутизация, арбитраж

Когда вы заходите на сайт, совершается множество невидимых вашему глазу операций. Ваш браузер посылает команду серверу, на который вы зашли, и команда просит сервер отослать вашему браузеру содержимое страницы, которую вы желаете посмотреть. Вот что происходит, если объяснять человеческим языком; но машины воспринимают происходящие процессы по-другому. Компьютеры не просто посылают команды или говорят друг с другом. Они общаются с помощью тщательно прописанного протокола, согласно которому каждая сторона коммуникации может пользоваться только особым набором высказываний, изложенных в определенной форме.

Организованный подобным образом способ коммуникации так и называется — протокол. Его можно представить в виде ряда инструкций, которые распределяют, как в ролевой игре, кто что и когда говорит. Протоколы передачи данных для компьютеров бесчисленны. Представим, например, будничный телефонный разговор. Когда Алиса звонит Бобу и Боб берет трубку, Алиса ждет его ответа, чтобы опознать его по произнесенному «алло», «Боб слушает» или иной реплике. Если Боб не отвечает, Алиса может попробовать добиться ответа, спросив: «Алло? С кем я говорю?» — хотя обычно телефонные беседы так не начинаются. Затем, когда обе стороны разговаривают, они, скорее всего, бессознательно подтверждают свое присутствие в беседе короткими высказываниями. Если Боб все говорит и говорит, то Алиса время от времени будет вставлять что-то вроде «ага», «ясно» или «ну да». На самом деле ей не с чем соглашаться, просто, если Боб будет долго что-то рассказывать, не получая от нее отклика, он, со своей стороны, задаст вопрос, проверяющий связь: «Понятно говорю?» или «Меня слышно?»

Мы вставляем подобные реплики уже на автомате, потому что ежедневно сталкиваемся с сотнями таких ситуаций; но это не пустяк, ведь таким образом вы проверяете слышимость собеседника, когда связь плохая или вовсе обрывается, особенно в мобильных телефонах.

Когда ваш браузер обменивается данными с веб-сервером и получает страницу сайта, он также следует четко прописанному протоколу, называемому HTTP*, — протоколу передачи гипертекста. Протокол неимоверно прост: он содержит несколько команд, которые браузер может послать серверу, и ряд ответов, которые сервер присылает обратно браузеру. Во время HTTP-соединения браузер и сервер могут обмениваться только этими заданными командами и ответами.

Простейшая из таких команд называется `GET`, команда получения. Браузер посылает серверу команду `GET`, направляя вместе с ней адрес запрашиваемой страницы. Вот пример команды:

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1
```

Команда просит сервер с именем www.w3.org прислать браузеру веб-страницу, обозначенную как /pub/WWW/TheProject.html, используя версию 1.1 HTTP-протокола. Сервер найдет данную страницу и отошлет обратно браузеру.

Когда мы говорим, что браузер просит сервер о чем-либо, мы опускаем львиную долю технических подробностей. Браузер посылает запрос серверу по каналу связи с сервером. Браузер не имеет никакого отношения к тому, каким образом установлено соединение или как оно работает. Он полагает, что уже есть стабильный канал связи между ним и сервером и что он спокойно может посылать запросы на сервер. Точно так же через то же соединение сервер может посылать данные обратно браузеру, он тоже не имеет ни малейшего понятия об устройстве и работе установленного соединения. Соединение можно представить как трубу между браузером и сервером, по которой туда и обратно течет поток данных.

---

* HTTP — от HyperText Transfer Protocol, протокол передачи гипертекста.

Так как ни браузер, ни сервер не связаны с устройством соединения и способом переправки данных, значит, нечто иное отвечает за работу канала связи. Этим иным является еще один протокол — протокол управления передачей данных, Transmission Control Protocol (TCP). Его задача — предоставлять своим пользователям, в данном случае HTTP-протоколу, соединение, с помощью которого можно передавать данные туда и обратно. TCP-протокол ничего не знает о командах, которые он передает во время соединения, и совершенно без понятия, что такое HTTP и `GET`. Он знает лишь, что надо взять данные, которые могут быть той же командой `GET`, доставить их на другой конец соединения, дождаться там ответа и переслать его отправителю команды `GET`. Он берет HTTP-команды и ответы частями, называемыми сегментами, которые потом посылает по установленному каналу связи. Если данных слишком много, он может разбить их на куски определенного размера. HTTP-протокола ничего это не касается; TCP соединит все части и воссоздаст полный запрос или ответ, который получит HTTP.

TCP отвечает за установку соединения и обмен данными между двумя соединенными компьютерами, однако он не отвечает за само перемещение данных между компьютерами. Он лишь создает иллюзию, что существует подобное надежное соединение, хотя на самом деле не происходит никакого физического оборота, как в трубах с водой или, что ближе к теме, в проводах. Более того, TCP-протокол даже не знает, каким образом данные должны добраться до места назначения. Как если бы он просто управлял соединением, но сам не знал, как это соединение устроено. Представьте трубу, состоящую из множества частей. TCP знает, как поместить данные в трубу, как отправить их в путь, как забрать их на другом конце и проверить, все ли дошло до места назначения, однако он совершенно не имеет представления, как устроена сама труба, из каких частей состоит и сколько этих самых частей. TCP может выполнять свою работу, а труба тем временем изменится: если она забьется, какие-то ее части исчезнут, какие-то новые добавятся. Труба может даже протечь и потерять данные. TCP заметит утечку и потребует повторной передачи потерянных данных, но он не может знать, что

происходит с данными, когда они попадают в трубу. Это работа уже другого протокола — интернет-протокола, IP.

IP принимает данные в виде сегментов из TCP и формирует из них пакеты, в каждом из которых содержатся адреса пунктов отправления и назначения. Пакеты могут иметь определенный размер, в зависимости от сети, по которой их будут пересылать, поэтому каждый сегмент может быть разбит на несколько пакетов. Отправитель принимает каждый IP-пакет и передает его дальше, к месту назначения через сетевой интерфейс, который кладет данные в исходную сеть и затем забывает о них. Порой местом назначения может оказаться компьютер, напрямую соединенный с источником, но такое случается редко. Компьютеры соединены друг с другом, и маршрут от пункта отправления до пункта назначения может пролегать через несколько соединений, установленных между несколькими компьютерами; точно так же с дорожной сетью: порой находится прямая дорога от одного города к другому, но чаще всего приходится проезжать через несколько городов, один за другим. IP-протокол не гарантирует доставку и не отслеживает пакеты. Он не гарантирует и того, что пакеты прибудут в нужном порядке. Не гарантирует, что все пакеты прибудут в пункт назначения одним и тем же путем. Он всего лишь передает пакеты вперед, к месту доставки. Он знает только, куда ему отправить полученный пакет или куда доставить его, если пункт назначения в его зоне досягаемости. Когда пакеты прибыли на место, он передает их TCP в виде сегментов, собирая в единое фрагменты, если необходимо. Дальше уже задача TCP принять сегменты, разложить их по порядку, обнаруживая среди них пропажи, и послать запрос своему двойнику на другом конце соединения, чтобы тот повторно отослал недостающие сегменты. Таким образом, TCP сделает все нужное для создания иллюзии надежного соединения, укрывающей собой ненадежный транспортный уровень, который всего лишь передает пакеты вперед и тут же забывает о них.

Хорошей метафорой, которая может помочь вам понять взаимоотношения между TCP и IP, может служить город с вымышленной водопроводной магистралью. Представьте, что в этом городе водопроводные краны работают как обычно: вода течет из них по-

током. Однако, если вы посмотрите, как вода поступает в ваш кран, то обнаружите, что нет никаких труб, тянущихся к нему от резервуара с водой. Есть только водовозы у резервуара, которые наполняют ведра водой. Когда вы открываете кран, эти водовозы доставляют ведра с водой к цистерне возле вашего дома. Они наполняют ее, и у вас в кране появляется вода, поэтому создается впечатление, что вода течет из резервуара непосредственно в ваш кран, хотя на самом деле все происходит благодаря усилиям водовозов, переправляющих ведра с водой. Более того, каждый водовоз волен сам выбирать удобнейший путь до вашего дома; им всем вовсе не обязательно ехать к вам одной и той же дорогой. При этом, если они привозят воду достаточно быстро, создается иллюзия, что вы живете в городе, оснащенном водопроводными трубами. Водовозы — это IP-протокол; поток воды, который вы видите, — это TCP-протокол, который заставляет вас думать, что вода берется из труб, и тем самым закрывает собой всю подноготную доставки воды.

Весь процесс показан на рисунке 8.1. На каждом уровне протоколы обмениваются данными, создавая впечатление, что они общаются со своим соратником на другой стороне коммуникации. На самом же деле данные спускаются от верхних протоколов к материальной сети, разбиваясь, если нужно, на части, а уже оттуда переходят на другую сторону соединения и поднимаются вверх по уровням протокола, собираясь, если нужно, воедино.

Другой способ осмыслить происходящий процесс — это рассмотреть его снизу вверх. На самом нижнем уровне рисунка 8.1 изображена материальная сеть, которая перемещает биты с помощью, например, световых волн, если в основе материальной сети оптические волокна. Компьютер обменивается данными с материальной сетью с помощью сетевого интерфейса. Сетевой интерфейс отправляет данные в сеть и забирает их оттуда. Данные должны знать, куда они следуют. За это отвечает IP-протокол, занимающийся маршрутизацией данных между компьютерами, объединенными Сетью. IP-протокол не гарантирует доставку данных на место, он лишь направляет их вперед и тут же забывает о них. За гарантированную доставку и установление надежного соединения, прикрывающего ненадежный механизм

переправки данных, отвечает TCP-протокол. Он заметит, что некоторые данные, которые по запросу должен был прислать ему IP, не дошли до пункта назначения, и попросит TCP на другой стороне послать их заново. В таком случает он позволяет HTTP или другому протоколу уровня приложений вызывать свои команды и получить их ответ, не задумываясь о самих командах.

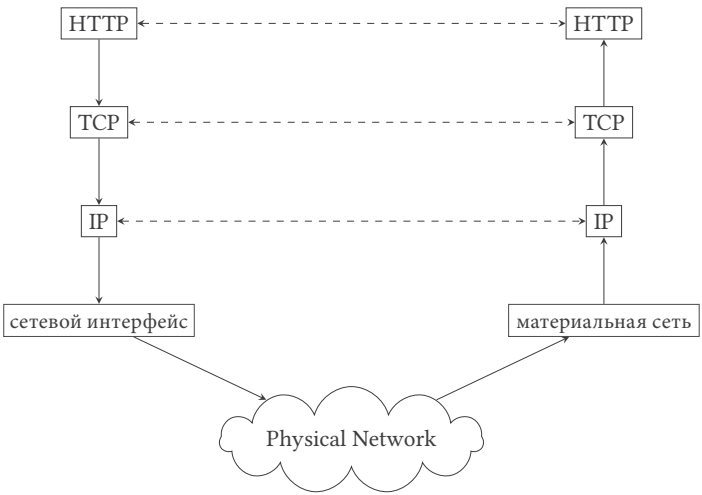


Рисунок 8.1. Стек протокола TCP/IP с HTTP

Просто удивительно, что весь Интернет базируется на ненадежном способе транспортировки, однако совместная работа TCP и IP протоколов заставляют все работать. Они, как правило, всегда упоминаются вместе и называются набором протоколов TCP/IP. Интернетом же называется группа компьютеров по всему свету, связанная между собой с помощью набора протоколов TCP/IP.

8.1. Маршрутизация Интернета

Как мы уже говорили, если два компьютера не соединены друг с другом, то есть не связаны одной общей физической сетью, IP не может переправлять пакеты из исходной точки в место назначения, но обязан отсылать их посредникам. Такие посредники

называются маршрутизаторами, так как они прокладывают путь данным внутри сети. Принцип работы такой, что IP видит адрес на пакете, знает, к какому маршрутизатору его отправить, и надеется, что маршрутизатор находится ближе к месту доставки и разберется, что делать с посылкой дальше. Чтобы лучше понять, представьте группы курьеров в различных городах. Если один курьер получает пакет от другого курьера, он первым делом проверяет адрес: в его ли городе находится получатель. Если да, то курьер тут же доставляет пакет адресату. Если нет и пакет направляется в соседний город, курьер знает, что нужно отправить его туда. Если ни то, ни другое, он проверяет записи и рассуждает так: «Все пакеты с адресами подобного вида должны отправляться вот в этот город, который, насколько мне известно, лучше знает, что с ними делать». Например, курьер знает, что все с пометкой «север» должно переправляться в соседний город на севере, где знают, что делать с полученными пакетами.

Как IP узнает, куда передавать пакеты? В Интернете нет никакой единой справочной службы или карты. Интернет состоит из больших подсетей, называемых автономными системами. Каждой автономной сети соответствует сеть передачи данных какого-либо университета, или сеть крупного поставщика интернет-услуг (ISP). Комплексный протокол BGP — протокол пограничной маршрутизации — отвечает за организацию путей между маршрутизаторами разных автономных систем. Другой протокол, называемый RIP — протокол маршрутной информации, — отвечает за организацию путей между маршрутизаторами внутри каждой автономной сети.

Чтобы разобраться в работе RIP, вернемся к метафоре с курьерами. Представьте, что каждый курьер — истинный бюрократ. Когда курьеры занимают свой пост, они не знают ничего, кроме одного факта: их город напрямую связан с несколькими другими городами. Все, что они могут делать, — это посылать пакеты в соседние города и принимать пакеты от соседей. У них также есть учетный журнал, в который записываются инструкции по передаче пакетов. Изначально журналы пустуют, в них лишь инструкции по доставке пакетов в соседние города.

Курьеры посылают друг другу особые сообщения, в которых записано, с какими городами связан их город и насколько далеко они расположены. Эти сообщения крайне важны. Кода курьер получает из соседнего города такое сообщение, он видит, что его город связан с городами, которые совсем далеко. Теперь он знает, что сообщения, в которых местом назначения указан соседний город или же соседи соседних городов, будут направляться в этот соседний город, потому что соседский курьер лучше знает, что делать с подобными сообщениям.

Соседний город, в свою очередь, тоже будет получать такие сообщения, поэтому будет знать, как передавать сообщения дальше. А также периодически будет делиться своими знаниями с первым курьером. Таким образом через какое-то время у всех курьеров сформируется общее представление о том, куда переправлять полученные сообщения. Более того, иногда курьер может получить сообщение, в котором говорится, что путь через соседний город короче, чем путь, который использовался до сих пор, и в таком случае курьер вносит в свой учетный журнал коррективу.

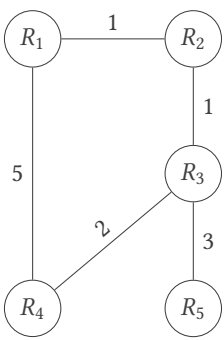


Рисунок 8.2. Автономная система с пятью маршрутизаторами

Если процесс все еще остается загадкой, объясним на другом примере. На рисунке 8.2 изображена автономная система с пятью маршрутизаторами, а на рисунке 8.3 показана работа RIP.

Хотя на рисунке 8.3 изображена общая таблица со всеми путями между маршрутизаторами, на самом деле такой таблицы не

существует. У каждого маршрутизатора есть своя таблица, которая похожа на ту, что изображена на рисунке, вот только в ней нет пустых ячеек. Так как каждый маршрутизатор не знает, сколько всего других маршрутизаторов, его индивидуальная таблица содержит столько маршрутизаторов, о скольких ему известно в данный конкретный момент. Поэтому в начале в таблице для R_1 содержится только три элемента, затем четыре, а позже — пять. На рисунке в каждой ячейке (R_i, R_j) мы указываем расстояние от R_i до R_j и следующий маршрутизатор, к которому нужно направить пакеты. Так что, если в ячейке (R_5, R_2) заключено значение $4/R_3$, это означает, что расстояние от R_5 до R_2 — 4 и пакеты до R_2 должны отправиться к R_3. Мы добавляем *D*, которым обозначаем прямое соединение.

На рисунке 8.3a все элементы имеют вид *x*/*D*, потому что нам известны только прямые соединения. Затем на рисунке 8.3(b) мы показываем, что происходит после того, как R_1 получил сообщение от R_2. R_1 теперь знает, что может передавать сообщения в R_3 через R_2; потому что R_2 может передавать сообщения R_3, проходя путь длиной в единицу, и общее расстояние от R_1 до R_3 через R_2 равно двум; мы подчеркиваем обновленную информацию. Потом, на рисунке 8.3(c), R_2 получает сообщение от R_3. R_2 теперь знает, что нужно передавать сообщения в R_4 и R_5 через R_3, поэтому мы обновляем элементы таблицы.

Когда мы прекращаем отслеживание работы RIP на рисунке 8.3, R_4 знает прямой путь до R_1, но не более короткий, который проходит через R_3. Так случилось потому что R_4 не получил сообщение от R_3 после того, как R_3 получил сообщение от R_2. Как только R_4 его получит, он будет знать удобнейший маршрут до R_1 и обновит элементы таблицы. В какой-то момент маршрутизаторы обменяются всеми необходимыми сообщениями, и им станут известны все самые оптимальные, кратчайшие пути.

Чтобы полностью понять, каким образом работает RIP, стоит рассмотреть то, что происходит в отдельно взятом маршрутизаторе. В самом начале он знает лишь пути до ближайших маршрутизаторов, напрямую связанных с ним. Затем он получает пакет от одного из них. Представим, что второй маршрутизатор сам не

получал никаких пакетов от другого маршрутизатора. Тогда пакет, отправленный им первому маршрутизатору, сообщает первому маршрутизатору только о соседях второго маршрутизатора, с которыми тот связан напрямую.

	R_1	R_2	R_3	R_4	R_5
R_1	0	1/D	-	5/D	-
R_2	1/D	0	1/D	-	-
R_3	-	1/D	0	2/D	3/D
R_4	5/D	-	2/D	0	-
R_5	-	-	3/D	-	0

(a) Начальное состояние

	R_1	R_2	R_3	R_4	R_5
R_1	0	1/D	$\underline{2/R_2}$	5/D	-
R_2	1/D	0	1/D	-	-
R_3	-	1/D	0	2/D	3/D
R_4	5/D	-	2/D	0	-
R_5	-	-	3/D	-	0

(b) $R_2 \rightarrow R_1$

	R_1	R_2	R_3	R_4	R_5
R_1	0	1/D	$2/R_2$	5/D	-
R_2	$1/R_1$	0	1/D	$\underline{3/R_3}$	$\underline{4/R_3}$
R_3	-	1/D	0	2/D	3/D
R_4	5/D	-	2/D	0	-
R_5	-	-	3/D	-	0

(c) $R_3 \rightarrow R_2$

	R_1	R_2	R_3	R_4	R_5
R_1	0	1/D	$2/R_2$	$\underline{4/R_2}$	$\underline{5/R_2}$
R_2	$1/R_1$	0	1/D	$3/R_3$	$4/R_3$
R_3	-	1/D	0	2/D	3/D
R_4	5/D	-	2/D	0	-
R_5	-	-	3/D	-	0

(d) $R_2 \rightarrow R_1$

	R_1	R_2	R_3	R_4	R_5
R_1	0	1/D	$2/R_2$	$4/R_2$	$5/R_2$
R_2	$1/R_1$	0	1/D	$3/R_3$	$4/R_3$
R_3	-	1/D	0	2/D	3/D
R_4	5/D	$\underline{3/R_3}$	2/D	0	$\underline{5/R_3}$
R_5	-	-	3/D	-	0

(e) $R_3 \rightarrow R_4$

	R_1	R_2	R_3	R_4	R_5
R_1	0	1/D	$2/R_2$	$4/R_2$	$5/R_2$
R_2	$1/R_1$	0	1/D	$3/R_3$	$4/R_3$
R_3	-	1/D	0	2/D	3/D
R_4	5/D	$3/R_3$	2/D	0	$5/R_3$
R_5	-	$\underline{4/R_3}$	3/D	$\underline{5/R_3}$	0

(f) $R_3 \rightarrow R_5$

	R_1	R_2	R_3	R_4	R_5
R_1	0	1/D	$2/R_2$	$4/R_2$	$5/R_2$
R_2	$1/R_1$	0	1/D	$3/R_3$	$4/R_3$
R_3	$\underline{2/R_2}$	1/D	0	2/D	3/D
R_4	5/D	$3/R_3$	2/D	0	$5/R_3$
R_5	-	$4/R_3$	3/D	$5/R_3$	0

(g) $R_2 \rightarrow R_3$

	R_1	R_2	R_3	R_4	R_5
R_1	0	1/D	$2/R_2$	$4/R_2$	$5/R_2$
R_2	$1/R_1$	0	1/D	$3/R_3$	$4/R_3$
R_3	$2/R_2$	1/D	0	2/D	3/D
R_4	$\underline{5/R_2}$	$3/R_3$	2/D	0	$5/R_3$
R_5	-	$4/R_3$	3/D	$5/R_3$	0

(h) $R_3 \rightarrow R_5$

Рисунок 8.3. Работа RIP-протокола для рисунка 8.2

Таким образом, первому маршрутизатору теперь известны еще и пути до соседей его соседа, второго маршрутизатора, то есть он осведомлен о путях длиной два соединения, тянущихся от него. Если второй маршрутизатор получает пакет от третьего маршрутизатора, который не получал пакетов от других маршрутизаторов, он помимо пути до третьего маршрутизатора будет знать еще и путь до его соседей, то есть ему тоже станут известны

пути длиной в два соединения. Когда же он пошлет новый пакет первому маршрутизатору, тот узнает пути длиной в три соединения, проходящие через второй маршрутизатор.

Все это напоминает процедуру релаксации, когда, каждый раз получая пакет, мы можем релаксировать пути с возрастающим числом соединений. Так и происходит: раз за разом мы применяем релаксацию к путям с возрастающим числом соединений. Чтобы убедиться в том, что RIP действительно работает, вам нужно посмотреть, как подобное применение релаксации находит кратчайшие пути в графе.

Вместо всех общающихся друг с другом маршрутизаторов, возьмем граф, в котором мы хотим найди кратчайший путь от стартового узла. Мы начинаем с того, что присваиваем каждому кратчайшему пути значение ∞, кроме значения пути нашего стартового узла, которое мы ставим на 0.

Далее мы берем каждое ребро графа и релаксируем значение пути до пункта назначения. Иными словами, мы берем каждое ребро графа и проверяем, возможно ли с его помощью добраться от нашего стартового узла до места назначения по пути более короткому чем тот, который мы уже нашли. Когда мы проделываем это первый раз, мы находим значения кратчайшего пути для узлов, которые напрямую связаны со стартовым узлом, поэтому мы получим кратчайшие пути, содержащие лишь одно ребро. Так начинается работа RIP в одном узле.

Мы повторяем описанную процедуру и релаксируем все ребра графа. В этот раз, второй по счету, мы найдем значения кратчайших путей для узлов, которые напрямую связаны со стартовым узлом или же с узлами, напрямую связанными со стартовым. То есть мы найдем значения кратчайшего пути для узлов, которые отделены от стартового двумя ребрами. Затем, если мы повторим процедуру $|V|-1$ раз, мы найдем значения кратчайшего пути для узлов, которые отделены от стартового $|V|-1$ ребрами. В графе не может быть пути от одного узла до другого, содержащего более $|V|-1$ узлов, если только в графе нет циклов. Если пути циклов имеют положительную длину, их не может быть в каком-либо кратчайшем пути. Если пути циклов имеют отрицательную дли-

ну, все равно их не может быть в кратчайшем пути графа, так как мы можем ходить по ним вечно, сокращая длину нашего пути до $-\infty$. В любом случае, после $|V|-1$ повторений процесса релаксации для всех ребер мы найдем кратчайшие пути от нашего стартового узла до каждого другого узла графа.

8.2. Алгоритм Беллмана — Форда (— Мура)

Данная процедура является высокоуровневым описанием алгоритма Беллмана — Форда, показанного в алгоритме 8.1. Он назван в честь Ричарда Беллмана и Лестера Форда-младшего, опубликовавшего алгоритм. Его еще называют алгоритмом Беллмана — Форда — Мура, так как Эдвард Ф. Мур опубликовал данный алгоритм в одно с ними время. Таким образом, RIP-протокол распределенную версию алгоритма Беллмана — Форда для нахождения не только кратчайших путей от стартового узла, но и всех кратчайших путей между любыми парами узлов. Иными словами, он решает задачу о кратчайшем пути между всеми парами вершин с помощью распределенного подхода.

Вернемся к базовой, то есть не распределенной, версии алгоритма Беллмана — Форда. В нее используется массив *pred*, в котором хранится предшественник каждого узла в кратчайшем пути, и массив *dist*, в котором содержится значение кратчайшего пути.

Алгоритм Беллмана — Форда начинается с инициализации структур данных в строчках 1–8, они отражают текущую ситуацию, в которой мы не знаем ни одного кратчайшего пути, кроме пути до нашего стартового узла. Следовательно, нет никаких предшествующих узлов, а длина всех кратчайших путей задана как ∞, кроме длины пути до стартового узла, которая, как мы знаем, равна нулю. После инициализации алгоритм просматривает кратчайшие пути вечно возрастающего числа ребер. Изначально пути могут состоять только из одного ребра, затем — из двух, и так, пока мы не достигнем максимального числа ребер в пути. Путь с максимальным числом ребер будет содержать все узлы в графе по одному, и, следовательно, будет иметь $|V|-1$ ребер. Все

это происходит в строчках 9–13. Цикл работает следующим образом:

- после первого прохода цикла мы находим кратчайшие пути, в которых не более одного соединения;
- после второго прохода цикла мы находим кратчайшие пути, в которых не более двух соединений;
- после k прохода цикла мы находим кратчайшие пути, в которых не более k соединений;
- после $|V|-1$ проходов цикла мы находим кратчайшие пути, в которых не более $|V|-1$ соединений.

Так как нет кратчайшего пути с $|V|-1$ соединениями (иначе нам придется ходить кругами), работа завершена.

Алгоритм 8.1. Беллман — Форд

`BellmanFord(`G, s`)` $\rightarrow (pred, dist)$

Вводные данные: $G = (V, E)$, граф; s, исходный узел.

Выводимые данные: $pred$, такой массив размером $|V|$, что $pred[i]$ является предшественником узла i в кратчайшем пути от s; $dist$, такой массив размером $|V|$, что $dist[i]$ является длиной кратчайшего пути, вычисленной от узла s до узла i.)

```
1  pred ← CreateArray(|V|)
2  dist ← CreateArray(|V|)
3  foreach v in V do
4      pred[v] ← −1
5      if v ≠ s then
6          dist[v] ← ∞
7      else
8          dist[v] ← 0
9  for i ← 0 to |V| do
10     foreach (u, v) in E do
11         if dist[v] > dist[u] + Weight(G, u, v) then
12             dist[v] ← dist[u] + Weight(G, u, v)
13             pred[v] ← u
14 return (pred, dist)
```

Если вы собираетесь реализовать алгоритм 8.1, то точно так же, как с алгоритмом Дейкстры, будьте внимательны в строч-

ке 11. Если программный язык, который вы используете, не знаком с бесконечностью и вы прибегаете к помощи большого числа, убедитесь, что вы не прибавляете к большому числу и получаете маленькое число, так как вышли за пределы.

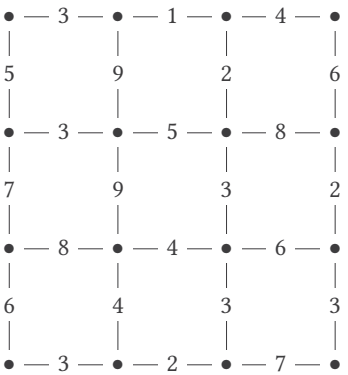


Рисунок 8.4. Дорожная сеть

Обратимся к рисунку 8.5 и проследим работу алгоритма Беллмана — Форда для графа дорожной сети, изображенной на рисунке 8.4. Для обозначения узлов сети мы будем использовать их координаты, начиная с нуля, так, что (0, 0) — это узел в верхнем левом углу, а (3, 3) — узел в нижнем правом углу, и так далее.

Каждому изображению на рисунке соответствует цикл процесса релаксации для всех ребер. На первом изображении представлены пути с нулями ребер, на втором — с одним ребром, и так до самого последнего изображения, где присутствуют семь ребер. Мы подчеркиваем ребра, в которых процесс релаксации включает в себя выбор между путями, то есть выбор между двумя ребрами, ведущими к конкретному узлу в конкретной итерации цикла. Поэтому во время второй итерации имеется два пути с двумя ребрами до узла в позиции (1, 1), и мы выбираем наилучший. В пятой итерации цикла узел в позиции (2, 1) получает улучшенный путь из-за выбора пути через узел (3, 2) вместо пути от узла (1, 1). В целом у нас нет группы узлов, про которые мы знаем, что их значения неизменны; у кратчайшего пути узла всегда есть шанс измениться, пока алгоритм не завершил работу. Более того, даже

после того, как мы вроде бы исследовали все узлы, как в шестой итерации, может оказаться так, что найдутся еще более удобные пути с бóльшим числом ребер. В восьмой итерации цикла так и случается, когда узел (3, 0) получает новый, лучший путь через узел (3, 1), в котором содержится семь ребер и который идет через (0, 0) → (0, 1) → (0, 2) → (1, 2) → (2, 2) → (3, 2) → (3, 1) → (3, 0), вместо старого пути, в котором всего три ребра и который проходит через (0, 0) → (1, 0) → (2, 0) → (3, 0).

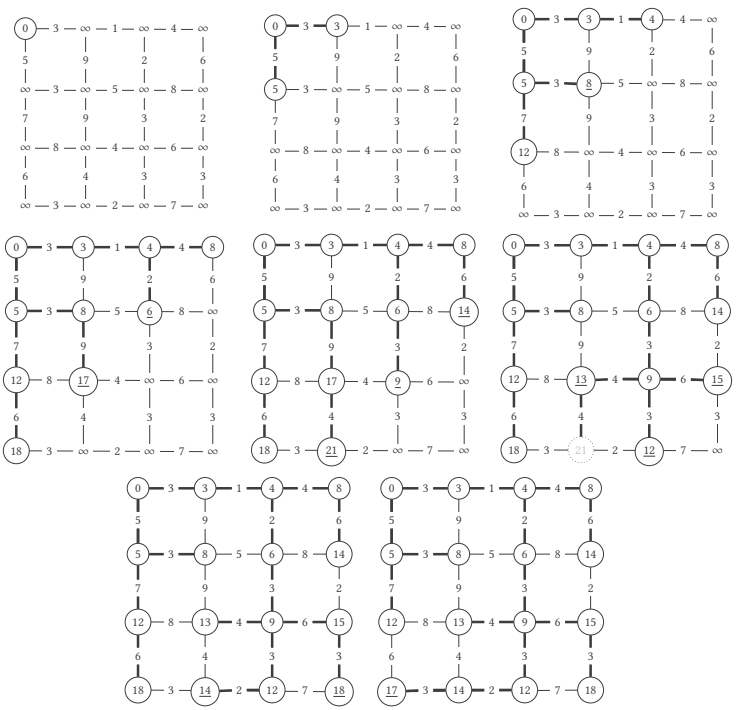


Рисунок 8.5. Кратчайший путь с алгоритмом Беллмана — Форда

В пятой итерации цикла мы отмечаем (3, 1) серым текстом и заключаем в пунктирный кружок, чтобы показать, что в данный момент он находится несуществующем состоянии. Путь до узла (2, 1) обновлен, и вместо пути с тремя ребрами, полученного в четвертой итерации цикла, у него теперь путь с пятью ребрами. В итерации *i* алгоритм находит кратчайшие пути, содержащие до

i ребер, не больше. Поэтому путь до узла (3, 1) от обновленного пути до узла (2, 1) на самом деле найден не в пятой итерации цикла, так как он имеет шесть ребер. На него просто наткнулись. Его можно найти случайно, если мы релаксируем ребро (2, 1) → (3, 1) после того, как мы релаксируем (2, 2) → (2, 1), но в алгоритме нет особых указаний на то, в каком порядке мы релаксируем ребра. Но не стоит переживать, так как узел (3,1) получит верный путь при последующей итерации цикла, и этот путь идет вовсе не из (2, 1).

С точки зрения времени алгоритм Беллмана — Форда выполняет $|V|$ итераций при инициализации. Инициализация включает в себя установку значений в массивах, которая занимает постоянное, $O(1)$ время, поэтому весь процесс занимает $O(|V|)$ времени. Цикл в строчках 9–13 выполняется $|V|-1$ раз, и каждый раз он просматривает все ребра графа, поэтому ему требуется $O((|V|-1)|E|) = O(|V||E|)$ времени. Весь алгоритм занимает $O(|V|+|V||E|)$ времени, то есть $O(|V||E|)$ времени.

Обратите внимание, что алгоритм, как мы его описали, не останавливается на седьмой итерации цикла. Он будет пытаться найти кратчайший путь с бóльшим числом ребер, пока не дойдет до пятнадцатой итерации. Как бы то ни было, в данном конкретном графе более короткого пути уже не найдется, поэтому нам не обязательно отслеживать оставшиеся итерации цикла и мы останавливаемся. Однако в целом это показывает нам возможность оптимизации в отношении алгоритма 8.1. Представьте, что мы находимся в итерации i алгоритма, мы расслабляем и в то же время обновляем кратчайшие пути, скажем, до m узлов. Назовем эти узлы $i_1, i_2,\ldots, i_m$. Кратчайшие пути до них будут иметь не более i ребер. При итерации $(i + 1)$ цикла алгоритма единственные ребра, которые нам надо проверить, — это ребра, смежные с узлами $i_1, i_2,\ldots, i_m$. Почему? Потому что $(i + 1)$ цикл алгоритма будет искать кратчайшие пути с не более чем $i + 1$ ребрами, так как мы уже нашли пути с не более чем i ребрами. Однако данные пути с i ребрами являются путями, чье последнее ребро заканчивается в одном из узлов $i_1, i_2,\ldots, i_m$. Поэтому в каждой итерации алгоритма вместо проверки всех ребер нам нужно просмотреть лишь ре-

бра узлов, значения которых мы обновили в прошлой итерации цикла.

Прежде чем мы воспользуемся преимуществами данной оптимизации, вам, наверно, интересно узнать, а почему этот алгоритм вообще работает. Объяснение чем-то похоже на то, которое мы приводили выше, и работает благодаря индукции. В самом начале, еще до прохождения первой итерации, для каждого узла массив *dist* содержит длину кратчайшего пути от стартового узла *s* до тех самых узлов, длина имеет не более нуля ребер. Это действительно так, потому что $dist[s] = 0$ и $dist[u] = \infty$ для всех $u \neq s$. Представим, что то же самое у нас при итерации *i*. Тогда для каждого узла *u* $dist[u]$ содержит длину кратчайшего пути от *s* до *u* с не более чем *i* ребер. А теперь подумайте, что происходит во время $(i + 1)$ итерации. Путь от *s* до любого узла *u* во время данной итерации может иметь не более $i + 1$ ребер. Если все такие пути содержат не более *i* ребер, мы уже нашли их во время предыдущих итераций, и поэтому $dist[u]$ ничего не меняет в нынешней итерации. Если имеются пути с $i + 1$ ребрами, то для каждого такого пути мы рассуждаем следующим образом. Путь будет состоять из двух частей. Он начнется от узла *s* и через *i* ребер прибудет в некий узел *w*; от *w* потянется еще одно ребро к узлу *v*: $s \overset{i}{\rightsquigarrow} w \to v$. Первая часть, $s \overset{i}{\rightsquigarrow} w$, содержит *i* ребер, и поэтому мы их уже нашли в *i* итерации. Это значит, что она не может быть длиннее кратчайшего пути от *s* до *w*. При итерации $(i + 1)$ цикла мы нашли ребро $w \to v$, которое имеет наименее возможный вес, добавляемый к весу пути $s \overset{i}{\rightsquigarrow} w$, поэтому путь $s \overset{i}{\rightsquigarrow} w \to v$ — это кратчайший путь с не более чем $i + 1$ соединениями.

Возвращаясь к оптимизации алгоритма, мы можем сохранять узлы, чьи оценочные данные мы обновляем при каждой итерации цикла путем их добавления в очередь вида «первым пришел — первым ушел». Нам также надо узнать, находится ли в очереди какой-либо элемент. Проще всего для этого использовать булев массив *inqueue*, в котором $inqueue[i]$ будет истинным, если *i* находится в *q*, в противном же случае он будет ложным. Таким образом, мы можем переделать Беллмана — Форда в алгоритм 8.2. Давайте вернемся к графу на рисунке 8.5 и проследим ход выпол-

нения алгоритма 8.2; отслеживание алгоритма показано на рисунке 8.6. Под каждой сеткой изображено содержимое очереди. Его видно и на самой сетке: узлы, находящиеся в данный момент в очереди, закрашены серым цветом.

Алгоритм 8.2. Основанный на очереди алгоритм Беллмана — Форда

`BellmanFordQueue(`G, s`)` $\rightarrow$ ($pred$, $dist$)

(Вводные данные: $G = (V, E)$, граф; s, исходный узел.
Выводимые данные: $pred$, такой массив размером $|V|$, что $pred[i]$ является предшественником узла i в кратчайшем пути от s; $dist$, такой массив размером $|V|$, что $dist[i]$ является длиной кратчайшего пути, вычисленной от узла s до узла i.)

```
1  inqueue ← CreateArray(|V|)
2  Q ← CreateQueue()
3  foreach v in V do
4      pred[v] ← −1
5      if v ≠ s then
6          dist[v] ← ∞
7          inqueue[v] ← FALSE
8      else
9          dist[v] ← 0
10 Enqueue(Q, s)
11 inqueue[s] ← TRUE
12 while Size(Q) ≠ 0 do
13     u ← Dequeue(Q)
14     inqueue[u] ← FALSE
15     foreach v in AdjacencyList(G, u) do
16         if dist[v] > dist[u] + Weight(G, u, v) then
17             dist[v] ← dist[u] + Weight(G, u, v)
18             pred[v] ← u
19             if not inqueue[v] then
20                 Enqueue(Q, v)
21                 inqueue[v] ← TRUE
22 return (pred, dist)
```

В этот раз алгоритм действительно останавливается на девятой итерации, и вы можете заметить выгоду, взглянув на предыдущую реализацию. Обратите внимание на то, что происходит

в шестой, седьмой и восьмой итерациях: узлы, которые уже были в очереди, повторно добавляются в нее, когда мы ищем кратчайшие пути с бóльшим числом ребер. В четвертой итерации мы нашли путь до узла (2, 1), затем в шестой нашли более короткий путь до него. В четвертой итерации цикла мы также нашли путь до узла (3, 0), затем в восьмой итерации мы нашли более короткий путь. В пятой же итерации нашли путь до узла (3, 1), который в седьмой итерации заменили более удобным.

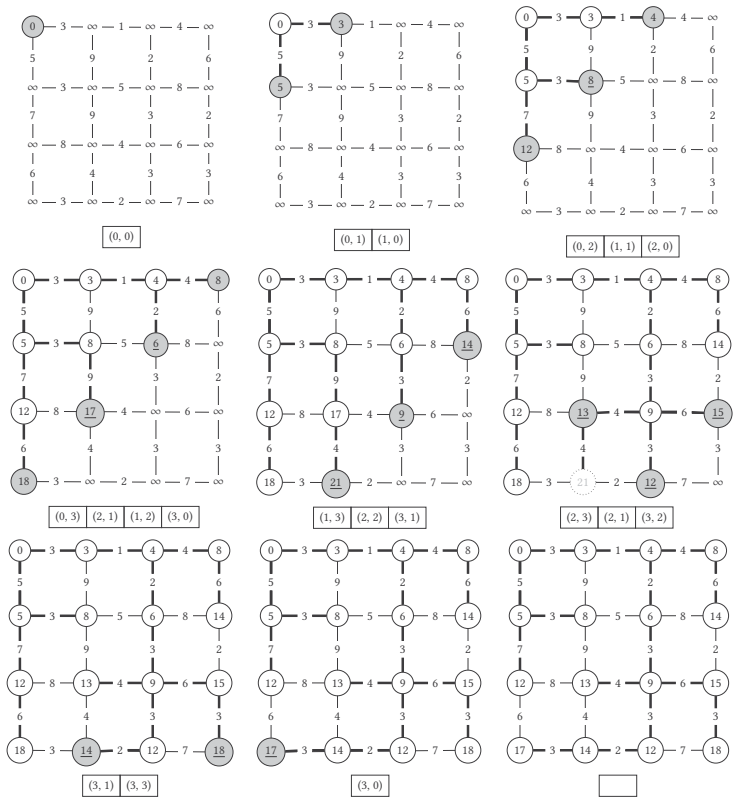


Рисунок 8.6. Кратчайшие пути с алгоритмом Беллмана — Форда, основанном на очереди

Время выполнения алгоритма Беллмана — Форда, $O(|V||E|)$, в целом намного хуже времени выполнения алгоритма Дейкстры, нижняя граница которого $O(|V|\lg|E|)$. Рисунки 8.5 и 8.6 слегка

вводят в заблуждение, так как они освещают лишь один момент из всей работы итераций, а ведь в них еще много чего происходит. В основанной на очереди версии проверка всех ребер смежных с узлами в очереди происходит в промежуток между двумя изображениями. В упрощенной версии все ребра графа проверяются заново при каждом перемещении от изображения к изображению. Основанный на очереди алгоритм во многих случаях улучшает время выполнения из алгоритма 8.1, но не всегда. Может случиться так, что при каждой итерации цикла графа мы обновляем оценочные данные пути всех узлов, поэтому при каждой итерации цикла нам нужно перепроверять все ребра. На деле же алгоритм Беллмана — Форда довольно эффективен.

8.3. Отрицательные веса и циклы

Не всегда у нас есть возможность выбирать между алгоритмами Дейкстры и Беллмана — Форда. В отличие от первого, алгоритм Беллмана — Форда умеет справляться с отрицательными весами и выдает корректные результаты. И правда, если вы возьмете граф, изображенный на рисунке 8.7, и примените к нему алгоритм Беллмана — Форда, вы получите верные результаты, изображенные на рисунке 8.8.

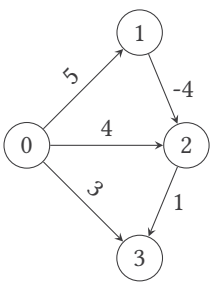


Рисунок 8.7. Граф с отрицательными весами

Теперь пойдем дальше и зададимся следующим вопросом: что случится, если в графе есть отрицательные веса, а заодно и отрицательные циклы, как на графе рисунка 8.9?

Алгоритм 8.1 остановится после $|V|$ проходов цикла и выдаст результаты, в которых не будет смысла, потому что, как мы говорили, кратчайший путь в цикле с отрицательными весами — это бесконечный путь с весом $-\infty$. С алгоритмом 8.2 все еще хуже: он вовсе не остановится. Когда он наткнется на цикл, он навечно уйдет в повтор одних и тех же операций, добавляя и вынимая узлы цикла в используемую очередь, как показано на рисунке 8.10.

Все это значит, что для завершения основанного на очереди алгоритма, имеющего дело с отрицательными циклами, нам нужно проделать дополнительную работу. Мы можем найти цикл, если обнаружим путь с более чем $|V| - 1$, потому что длиннейший путь в графе с $|V|$ узлами проходит через них все и имеет $|V| - 1$ ребер. Пути с бóльшим числом ребер возвращаются к узлам, в которых уже побывали.

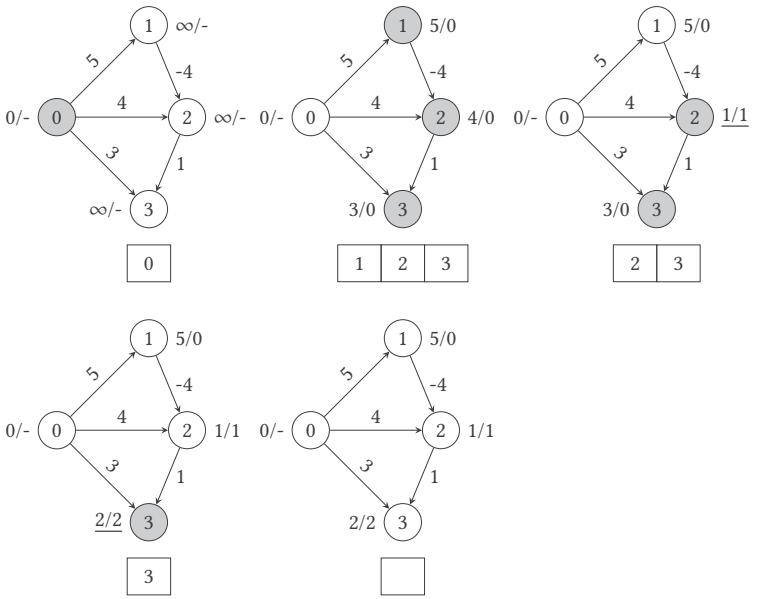


Рисунок 8.8. Алгоритм Беллмана — Форда над графом с отрицательными ребрами

Вспомним, что основная идея Беллмана — Форда заключается в исследовании путей с возрастающим количеством ребер.

Изначально, когда мы еще находимся в исходном узле *s*, мы имеем пути с нулями ребер. Когда мы добавляем в очередь соседние с *s* узлы, мы получаем пути с одним ребром. Когда мы добавляем в очередь узлы, смежные с соседними, мы получаем пути с двумя ребрами, и так далее. Вопрос в том, как узнать, когда мы прекращаем обработку одной группы соседей и переходим к другой. Когда такое случается $|V| - 1$ раз, мы знаем, что алгоритм должен завершиться.

Мы может разрешить данную задачку, используя сигнальную метку в очереди. В целом, сигнальные метки — это недопустимые значения, которые дают знать о нестандартных ситуациях. В нашем случае в качестве сигнальной метки мы используем число $|V|$, потому что такого узла у нас нет; узлы идут от 0 до $|V| - 1$. В частности, мы используем $|V|$, чтобы отделить в очереди группы соседних узлов, удаленных от *s* на то же самое число соединений. Как это работает, можно увидеть на рисунке 8.11.

Сначала мы помещаем наш исходный узел, 0, и сигнальную метку, 5, в очередь. Также устанавливаем индикатор, *i*, для отображения числа ребер в пути. Изначально он равен нулю. Каждый раз, когда 5 достигает головы очереди, мы увеличиваем *i* и отправляем 5 в конец очереди. Взгляните, что происходит, когда узел 3 находится во главе очереди. Мы убираем узел 3 из головы и добавляем в конец очереди двух его соседей, узлы 1 и 4. Теперь 5 становится во главу очереди, поэтому мы повышаем значение *i*. Мы убираем 5 в конец очереди, чтобы показать, что узлы 1 и 4 удалены от узла 0 одинаковым количеством соединений. Когда сигнальная метка снова достигает головы очереди и *i* увеличивается до 5, мы знаем, что начали ходить кругами, и можем остановиться. Измененный Беллман — Форд проделывает все это в алгоритме 8.3. Алгоритм 8.3 возвращает, помимо массивов *pred* и *dist*, булево значение, которое будет истинно при отсутствии отрицательного цикла и ложно, если таковой имеется. Алгоритм инициализирует сигнальную метку в строчке 10 и занимается индикатором *i* в строчках 16–17. Мы проверяем значение *i* в строчке 14; обратите внимание, что теперь у нас очередь никогда не пустует, потому что в ней всегда есть сигнальная метка.

Алгоритм 8.3. Основанный на очереди Беллман — Форд с отрицательными циклами

`BellmanFordQueueNC(`G, s`)` $\rightarrow$ ($pred$, $dist$, ncc)

Вводные данные: $G = (V, E)$, граф; s, исходный узел.

Выводимые данные: $pred$, такой массив размером $|V|$, что $pred[i]$ является предшественником узла i в кратчайшем пути от s; $dist$, такой массив размером $|V|$, что $dist[i]$ является длиной кратчайшего пути, вычисленной от узла s до узла i; nnc, будет `true`, если нет отрицательного цикла, а иначе — `false`.)

```
1  inqueue ← CreateArray(|V|)
2  Q ← CreateQueue()
3  foreach v in V do
4      pred[v] ← |V|
5      if v ≠ s then
6          dist[v] ← ∞
7          inqueue[v] ← FALSE
8      else
9          dist[v] ← 0
10 Enqueue(Q, s)
11 inqueue[s] ← TRUE
12 Enqueue(Q, |V|)
13 i ← 0
14 while Size(Q) ≠ 1 and i < |V| do
15     u ← Dequeue(Q)
16     if u = |V| then
17         i ← i + 1
18         Enqueue(Q, |V|)
19     else
20         inqueue[u] ← FALSE
21         foreach v in AdjacencyList(G, u) do
22             if dist[v] > dist[u] + Weight(G, u, v) then
23                 dist[v] ← dist[u] + Weight(G, u, v)
24                 pred[v] ← u
25                 if not inqueue[v] then
26                     Enqueue(Q, v)
27                     inqueue[v] ← TRUE
28 return (pred, dist, i < |V|)
```

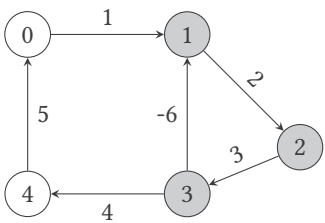


Рисунок 8.9. Граф с отрицательным циклом

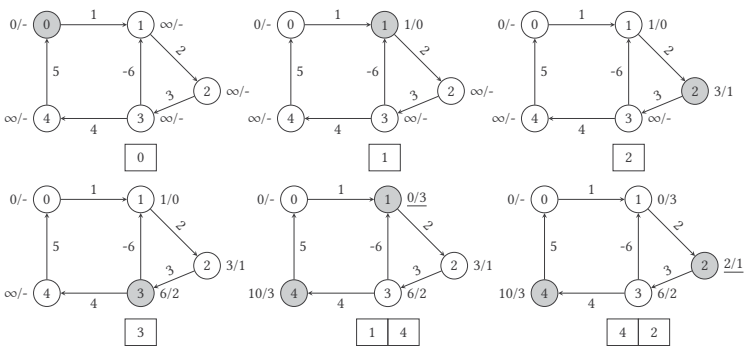


Рисунок 8.10. Ход выполнения алгоритма Беллмана — Форда с отрицательным циклом

8.4. Арбитраж

Не подумайте, что встреча с отрицательными циклами — это какое-то непонятное, надуманное явление. Циклы с отрицательными весами появляются и в повседневной жизни нашего мира. Выявление циклов с отрицательными весами в нашей жизни применяется для нахождения арбитражных возможностей. Арбитраж — это нечто сродни бесплатному сыру. Он включает в себя набор товаров, которыми можно обмениваться, покупать и продавать. Под такими товарами могут подразумеваться промышленные металлы, например медь, свинец и цинк, или валюта, как евро

и доллары, — вообще все, чем торгуют на рынке. Арбитраж — это использование разницы рыночных цен для получения выгоды. Приведем простой пример, представьте, что в Лондоне обменный курс евро и долларов €1 = $1.37, в то время как в Нью-Йорке обменный курс — $1 = €0.74. На €1,000,000 трейдер может купить в Лондоне $1,370,000, перевести их в Нью-Йорк и там снова обменять на евро. Он получит $1,370,000 × 0,74 = €1,013,800, сколотив из ниоткуда и без всякого риска €13,800 чистой прибыли.

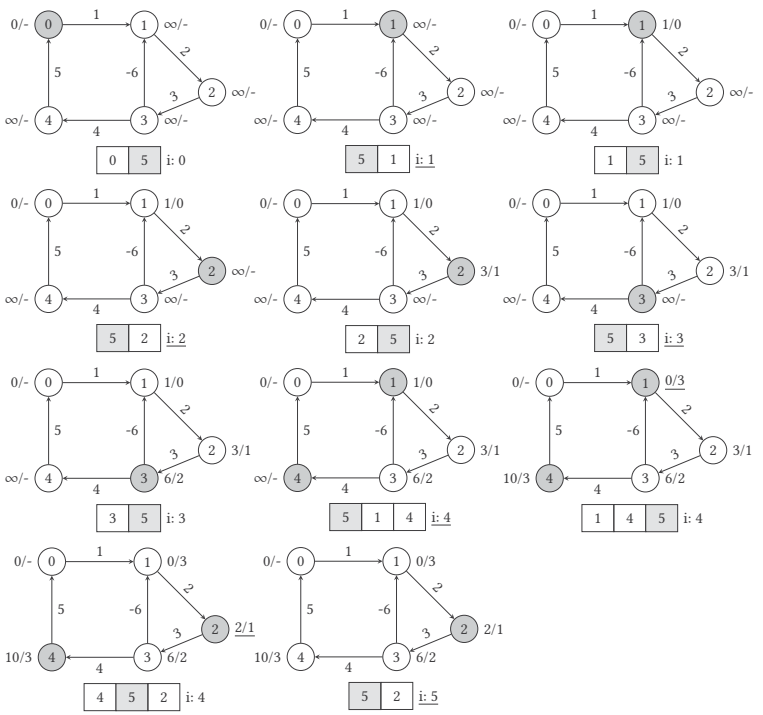


Рисунок 8.11. Ход выполнения измененного алгоритма Беллмана — Форда с отрицательным циклом

Так случается нечасто, потому что, как только появится арбитражная возможность, о ней тут же пронюхают трейдеры и рынки ее ликвидируют. Поэтому, возвращаясь к нашему примеру, после нескольких арбитражных сделок обменный курс либо снизится в Нью-Йорке, либо возрастет в Лондоне, так что все уравно-

весится: с курсом x в Лондоне мы очень быстро получим курс $1/x$ в Нью-Йорке.

Хотя арбитражная возможность появляется на короткий промежуток времени, тем не менее можно успеть наварить очень много денег, к тому же арбитражная возможность может оказаться вовсе не очевидной, чтобы увидеть ее как простую, прямую конверсию между двумя валютами. В таблице 8.1 в порядке приоритета размещены десять самых меняемых валют апреля 2013 года. Так как обмен валюты включает в себя пары, для всех валют процентная доля увеличена до 200% (включая те, что не вошли в десятку).

Таблица 8.1. Десять самых меняемых валют на рынке в апреле 2013 года

Rank	Currency	Code	% Daily Share
1	U.S. dollar	USD	87.0%
2	European Union euro	EUR	33.4%
3	Japanese yen	JPY	23.0%
4	United Kingdom pound sterling	GBP	11.8%
5	Australian dollar	AUD	8.6%
6	Swiss franc	CHF	5.2%
7	Canadian dollar	CAD	4.6%
8	Mexican peso	MXN	2.5%
9	Chinese yuan	CNY	2.2%
10	New Zealand dollar	NZD	2.0%

Если мы переводим деньги в таблице из одной валюты в другую, тогда может возникнуть арбитражная возможность, содержащая любой путь графа на рисунке 8.12. Таблица под графом показывает кросс-курс для валют, указанных в графе. Они представлены весами ребер, но их не так просто уместить в граф.

Представьте, что у вас есть американские доллары, и есть арбитражная возможность, возникающая при обмене ваших долларов на австралийские доллары, а те — на канадские доллары, а канадские — обратно на американские. Весь процесс обмена будет выглядеть следующим образом:

$$1 \times (\text{USD} \to \text{AUD}) \times (\text{AUD} \to \text{CAD}) \times (\text{CAD} \to \text{USD}).$$

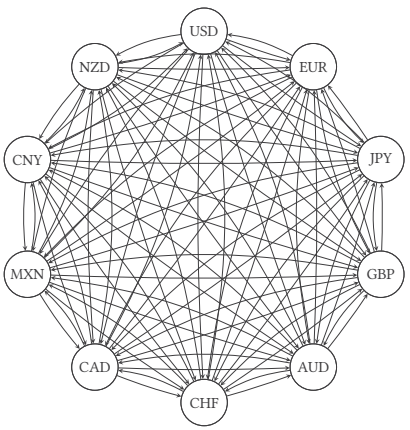
Вы получите изрядную выгоду, если полученное произведение больше единицы. Если представить валюту как c_1, c_2,…, c_n, тогда арбитраж *a* существует при

$$a = 1 \times (c_1 \rightarrow c_2) \times (c_2 \rightarrow c_3) \times \ldots \times (c_n \rightarrow c_1)$$

таким образом, что:

$$a > 1.$$

Обратимся к графу на рисунке 8.12; такая последовательность переводов отражает цикл графа, который проходит через узлы-валюты. Каждая арбитражная возможность будет находиться в цикле графа, и в этом цикле произведение весов должно быть больше единицы.



(a) Граф кросс-курсов

	USD	EUR	JPY	GBP	AUD	CHF	CAD	MXN	CNY	NZD
USD	1	1.3744	0.009766	1.6625	0.9262	1.1275	0.9066	0.07652	0.1623	0.8676
EUR	0.7276	1	0.007106	1.2097	0.6739	0.8204	0.6596	0.05568	0.1181	0.6313
JPY	102.405	140.743	1	170.248	94.8421	115.455	92.8369	7.836	16.6178	88.8463
GBP	0.6016	0.8268	0.005875	1	0.5572	0.6782	0.5454	0.04603	0.09762	0.5219
AUD	1.0799	1.4842	0.010546	1.7953	1	1.2176	0.979	0.08263	0.1752	0.9369
CHF	0.8871	1.2192	0.008663	1.4748	0.8216	1	0.8042	0.06788	0.144	0.7696
CAD	1.1033	1.5163	0.010775	1.8342	1.0218	1.2439	1	0.08442	0.179	0.9572
MXN	13.0763	17.9724	0.1277	21.7397	12.1111	14.7435	11.8545	1	2.122	11.345
CNY	6.167	8.4761	0.06023	10.2528	5.7118	6.9533	5.5908	0.4719	1	5.3505
NZD	1.153	1.5846	0.01126	1.9168	1.0678	1.2999	1.0452	0.08822	0.1871	1

(b) Матрица смежности для графа кросс-курсов

Рисунок 8.12. Граф кросс-курсов и матрица смежности

В нашем распоряжении нет алгоритма, позволяющего нам вычислить подобные циклы, однако один математический трюк позволит нам использовать имеющийся у нас арсенал. Вместо использования курсов обмена валют в качестве весов графа мы можем прибегнуть к отрицательному логарифму курсов обмена. То есть, если вес в ребре между узлами u и v — это $w(u, v)$, тогда мы подставляем $w'(u, v) = -\log w(u, v)$. Какие-то из данных логарифмов будут положительные, а какие-то — отрицательные; в частности, $w'(u, v) \geq 0$, если $w(u, v) > 1$. Мы применяем алгоритм Беллмана — Форда для циклов с отрицательными весами к нашему графу и смотрим, обнаружит ли он циклы с отрицательными весами. Если да, то мы ищем цикл. Предположим, что в цикле n ребер; тогда их сумма за один проход будет $w'_1 + w'_2 + \ldots + w'_n < 0$, где $w'_1 + w'_2 + \ldots + w'_n$ — это веса n ребер, образующие цикл между двумя узлами. Мы имеем

$$w'_1 + w'_2 + \ldots + w'_n = -\log w_1 - \log w_2 - \ldots - \log w_n,$$

так что для суммы путей в цикле с отрицательными весами мы получаем

$$-\log w_1 - \log w_2 - \ldots - \log w_n < 0.$$

Основное свойство логарифмов в том, для любого числа x и y произведение их логарифма $\log(xy)$ равно сумме их логарифмов $\log x + \log y$, точно так же $\log(1/x\ 1/y)$ равно $\log(1/x) + \log(1/y) = -\log x - \log y$. Отсюда следует, что последнее неравенство равнозначно:

$$\log\left(\frac{1}{w_1} \times \frac{1}{w_2} \times \cdots \times \frac{1}{w_n}\right) < 0.$$

Возводим в степень, чтобы убрать логарифм, и получаем:

$$\frac{1}{w_1} \times \frac{1}{w_2} \times \cdots \times \frac{1}{w_n} < 10^0 = 1.$$

Но это то же самое, что

$$w_1 w_2 \ldots w_n > 1,$$

именно это мы изначально и хотели найти; циклический путь с весами, чье произведение больше единицы. Каждый раз, когда вы обнаруживаете в графе цикл с отрицательными весами, перед вами арбитражная возможность. Не упустите!

Примечания

О работе TCP/IP и Интернета написано множество книг; классическое знакомство — книга Стивенса (с поправками Фолла) [58]; также посмотрите учебник Комера [38]. Чтобы увидеть, как общие идеи превращаются в сетевые протоколы, обратитесь к книге Перлмана [157]. Для знакомства с основами сетевой конфигурации посмотрите учебное пособие Куроза и Росса [119], а вместе с ним — учебник Таненбаума и Уэтеролла [196].

Лестер Форд опубликовал алгоритм в 1956 году [69]; затем Ричард Беллман — в 1958 [13], в то время как Эдвард Мур представил тот же алгоритм в 1957 году [145]. Если в графе нет циклов с отрицательными весами, то алгоритм можно улучшить, как показано Йеном [221]. Еще одна модификация, тоже для графов без циклов с отрицательными весами, была предложена совсем недавно Баннистером и Эпштейном [8].

9 Что важнее всего

Ежесекундно, не зная устали, в сети трудятся поисковые роботы — специальные программы, которые прыгают от страницы к странице, оставляя на каждой отметку. Они разбивают страницы на слова, предложения и фразы, затем передают другим страницам сети ссылки, которые найдут. Роботы берут полученное содержимое страницы и помещают его в большую структуру данных, называемую обращенный указатель, *inverted index*. Данный указатель похож на алфавитный указатель в конце книги, в котором приведены термины и страницы, где эти термины встречаются. Указатель называют обращенным, потому что его задача — позволять осуществлять выборку данных исходного контента, то есть добывать с нужной веб-страницы некий термин, появляющийся на ней. Обычно мы получаем содержимое страницы, когда читаем эту страницу; когда мы получаем страницу на основе ее содержимого, мы идем противоположным путем, то есть в обратном направлении.

Помимо указателя роботы используют ссылки, содержащиеся внутри веб-страниц, для поиска других веб-страниц, которые им нужно посетить и провести процесс индексирования; они также используют ссылки, чтобы создать большую карту сети, на которой отображено, какие страницы соединены между собой.

Все происходит таким образом, что, когда вы вбиваете запрос в поисковик, он выдает вам (хотелось бы верить) нужные результаты. Поисковая служба использует обращенный указатель для нахождения страниц, соответствующих вашему запросу. Но, как и с алфавитным указателем в книге, вашему запросу может соответствовать не одна страница. В книге это не столь важно, так как в ней найдется всего несколько нужных страниц и скорее всего искомый термин окажется на первой странице, где он появился. Но с сетью все иначе: здесь нет первой страницы, а число страниц,

отвечающих запросу, может превышать миллиарды. Естественно, вы не собираетесь просматривать их все. Вам нужно найти наиболее релевантные.

Например, вы ищете в сети Белый дом — вас ждет целая туча ссылок с пометкой «Белый дом». Обычный список всех страниц, соответствующих вашему запросу будет по большому счету бесполезен. Однако некоторые страницы наверняка окажутся более релевантными, чем другие; скорее всего, вам нужен сайт самого Белого дома, а не страничка неизвестного блогера, на которой излагаются специфические взгляды на политику Белого дома, — впрочем, если вы искали именно ее…

Один из способов решить задачу со страницами, выдаваемыми в результате поиска, — распределить их в порядке актуальности или значимости. Тогда встает следующий вопрос: как определить, что важно в мире виртуальных страниц?

9.1. Суть пейдж-ранка

Успешное и широко популярное решение было найдено основателями «Гугла», Сергеем Брином и Ларри Пейджем. Это решение, опубликованное в 1998 году, называется пейдж-ранком (PageRank), оно присваивает каждой веб-странице номер, который тоже называется пейдж-ранком. Чем выше пейдж-ранк, тем «важнее» страница. Главная суть пейдж-ранка заключается в том, что значимость страницы, ее пейдж-ранк, зависит от значимости страниц, связанных с ней.

Каждая страница может быть соединена с другими посредством исходящих ссылок, а другие страницы могут быть соединены с конкретной страницей посредством входящих ссылок. Мы помечаем страницу i с помощью P_i. Если у страницы P_j имеется m исходящих ссылок, тогда мы используем описание $|P_j|$ для числа m; иными словами, $|P_j|$ — это число исходящих ссылок страницы P_j. Мы полагаем, что значимость страницы $|P_j|$ одинаково зависит от соединенных с нею страниц. Например, если P_j связана с тремя страницами, то она обеспечит 1/3 своей значимости каждую из трех связанных с нею страниц. Мы

применяем описание $r(P_j)$ для пейдж-ранка страницы P_j. Следовательно, если P_j ссылается на P_i и имеет $|P_j|$ исходящих ссылок, то она передаст P_i часть своего пейдж-ранка равного $r(P_j)/|P_j|$. пейдж-ранк страницы является суммой пейдж-ранков, полученных ото всех ведущих к ней страниц. Если мы называем B_{Pi} группу страниц, связанных с P_i, то есть группу страниц с обратными ссылками к P_i, мы имеем:

$$r(P_i) = \sum_{P_j \in B_{P_i}} \frac{r(P_j)}{|P_j|}.$$

Если, например, у нас есть граф рисунка 9.1, тогда:

$$r(P_1) = \frac{r(P_2)}{3} + \frac{r(P_3)}{4} + \frac{r(P_4)}{2}.$$

Таким образом, чтобы найти пейдж-ранк страницы, нам нужно знать пейдж-ранк всех связанных с нею страниц; чтобы найти пейдж-ранк этих страниц, нам надо найти пейдж-ранк страниц, связанных уже с ними, и так далее. В то же время ссылки страниц друг на друга могут образовать циклы. Получается, мы приходим к задачке с курицей и яйцом, где для нахождения пейдж-ранка страницы нам надо вычислить нечто, для чего потребуется пейдж-ранк этой самой страницы.

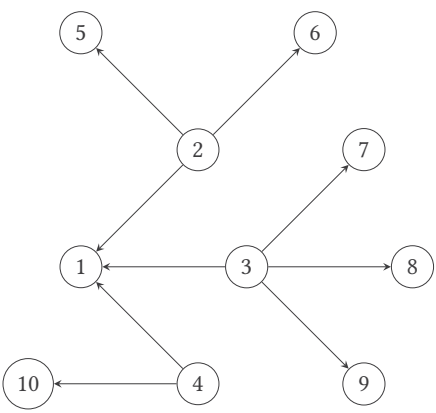


Рисунок 9.1. Минимальный веб-граф

Задача о курице и яйце решается путем применения повторяющейся процедуры. Сначала мы присваиваем каждой странице

некий пейдж-ранк. Если у нас n страниц, мы присваиваем каждой странице в качестве пейдж-ранка число $1/n$. Затем мы прибегаем к следующей итерации:

$$r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|P_j|}.$$

Нижние индексы $k + 1$ и k указывают на значение $r(P_i)$ и $r(P_j)$ в цикле $k + 1$ и k соответственно. Приведенная выше формула означает, что пейдж-ранк страницы рассчитывается путем использования пейдж-ранка страниц, соединенных с ней в предыдущей итерации входящими ссылками. Мы повторяем процедуру несколько раз в надежде, что через какое-то время расчеты пейдж-ранка придут к относительно устойчивым и точным значениям. Тут же возникает два вопроса, над которыми стоит поразмыслить.

- Приходит ли повторяющаяся процедура вычисления пейдж-ранка к общему значению после разумного количества итераций?
- Корректен ли результат, к которому приходит повторяющаяся процедура вычисления пейдж-ранка?

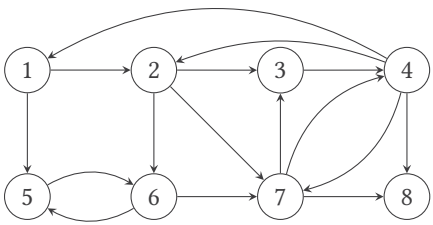


Рисунок 9.2. Еще один веб-граф

9.2. Матрица гиперссылок

Описанная только что процедура показывает, что происходит на каждой отдельной странице. Мы можем преобразовать ее в форму, которая обрисовывает происходящее на всех страницах через матрицы. В частности, мы начнем с определения матрицы, называемой матрицей гиперссылок. Она представляет собой квадрат-

ную матрицу с неким числом рядов или столбцов равным числу страниц. Каждому ряду и столбцу соответствует страница. Каждый элемент матрицы определяется как:

$$H[i,j] = \begin{cases} 1/|P_i|, & P_i \in B_{P_j} \\ 0, & \text{иначе} \end{cases}.$$

Иными словами, элемент $H[i, j]$ равен нулю, если нет ссылок от страницы P_i к странице P_j, или обратному количеству исходящих ссылок страницы P_i, если одна из ссылок идет к странице P_j.

Просмотрите граф на рисунке 9.2. Вот матрица H для данного графа, где для ясности добавлены указатели:

$$H = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \\ P_7 \\ P_8 \end{array} \begin{array}{c} \begin{array}{cccccccc} P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 \end{array} \\ \left[\begin{array}{cccccccc} 0 & 1/2 & 0 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 0 & 1/3 & 1/3 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1/4 & 1/4 & 0 & 0 & 0 & 0 & 1/4 & 1/4 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/3 & 1/3 & 0 & 0 & 0 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

Вы можете проверить пару особенностей приведенной матрицы. Сумма каждого ряда равна единице, так как делитель каждого элемента в ряду является количеством ненулевых элементов в этом ряду, кроме случая, когда у страницы нет исходящих ссылок: тогда весь ряд равен нулю. Что и происходит в восьмом ряду данной матрицы гиперссылок.

Мы также можем поместить в матрицу, или, точнее, в вектор, значения пейдж-ранка. Согласно условному обозначению векторы представляются в виде матриц, состоящих из одного столбца. Если вектор, содержащий пейдж-ранки, равен π, для группы из n страниц мы имеем:

$$\pi = \begin{bmatrix} r(P_1) \\ r(P_2) \\ \vdots \\ r(P_n) \end{bmatrix}.$$

Когда нам нужно преобразовать вектор из столбца в ряд, мы используем следующую запись:

$$\pi^T = \Big[r(P_1)\ r(P_2) \cdots r(P_n)\Big].$$

Возможно, выглядит не очень, но такой вид получил широкое распространение, поэтому вам следует привыкнуть к нему; T обозначает транспозицию. Также не забывайте, что мы используем массивы для представления матриц в компьютерах и что в большинстве программных языков массивы отсчитываются от нуля, поэтому мы имеем:

$$\pi^T = \Big[r(P_1)\ r(P_2) \cdots r(P_n)\Big] = \Big[\pi[0]\ \pi[1] \cdots \pi[n-1]\Big].$$

Имея под рукой данные определения, давайте получим произведение матриц π^T и H и используем его в качестве основания для повторяющейся процедуры:

$$\pi_{k+1}^T = \pi_k^T H\,.$$

Быстрая проверка показывает, что в самом деле умножения данных матриц полностью равноценны повторяющейся процедуре, с которой мы начали. Так как умножение двух матриц C и D, где в C содержится n столбцов, а в D — n рядов, представлено матрицей E, такой, как

$$E[i,j] = \sum_{t=0}^{n-1} C[i,t]D[t,j];$$

элемент i из $\pi_{k+1}^T = \pi_k^T H$ на самом деле:

$$\pi_{k+1}[i] = \sum_{t=0}^{n-1} \pi_k[t]H[t,i]$$

$$= \pi_k[0]H[0,i] + \pi_k[1]H[1,i] + \cdots + \pi_k[n-1]H[n-1,i]$$

В нашем примере каждый цикл вычисляет:

$$
\begin{aligned}
r_{k+1}(P_1) &= \frac{r_k(P_4)}{4} \\
r_{k+1}(P_2) &= \frac{r_k(P_1)}{2} + \frac{r_k(P_4)}{4} \\
r_{k+1}(P_3) &= \frac{r_k(P_2)}{3} + \frac{r_k(P_7)}{3} \\
r_{k+1}(P_4) &= r_k(P_3) + \frac{r_k(P_7)}{3} \\
r_{k+1}(P_5) &= \frac{r_k(P_1)}{2} + \frac{r_k(P_6)}{2} \\
r_{k+1}(P_6) &= \frac{r_k(P_2)}{3} + r_k(P_5) \\
r_{k+1}(P_7) &= \frac{r_k(P_2)}{3} + \frac{r_k(P_4)}{4} + \frac{r_k(P_6)}{2} \\
r_{k+1}(P_8) &= \frac{r_k(P_4)}{4} + \frac{r_k(P_7)}{3},
\end{aligned}
$$

что он и должен вычислять.

9.3. Степенной метод

Последовательное умножение матриц образует степенной метод, так как оно включает в себя увеличение вектора — в нашем случае вектора значений пейдж-ранка — до последовательных степеней. Два вопроса, возникших в описании исходной итерационной процедуры, теперь появились и в царстве степенного метода. По сути, мы хотим знать, придет ли ряд операций умножения к точному логичному π^T после нескольких циклов. Если да, то мы зовем вектор, к которому приводят все эти операции умножения, стационарным вектором, потому что дальнейшие циклы расчетов степенного метода не изменят его значения. Можно ли степенным методом найти верный стационарный вектор?

Простого контрпримера достаточно, чтобы показать, что результат не всегда верен. Взгляните на рисунок 9.3. На нем изображены всего три узла и одна ссылка от одного узла к другому.

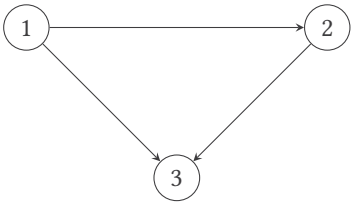


Рисунок 9.3. Проседание пейдж-ранка

Все тем же способом, которым мы описали пейдж-ранк, страница при каждой итерации дает часть своей значимости страницам, к которым она ведет. В то же время она берет значимость от страниц, которые ведут к ней. В нашем контрпримере страница P_1 будет давать свою значимость странице P_2, но сама не будет ниоткуда получать значимость. В результате получится, что все пейдж-ранки исчезнут через три цикла:

$$\begin{bmatrix} 1/3 & 1/3 & 1/3 \end{bmatrix} \begin{bmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1/6 & 1/2 \end{bmatrix},$$

затем

$$\begin{bmatrix} 0 & 1/6 & 1/2 \end{bmatrix} \begin{bmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1/6 \end{bmatrix},$$

и наконец

$$\begin{bmatrix} 0 & 0 & 1/6 \end{bmatrix} \begin{bmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}.$$

Наш контрпример включает только три узла, но он отображает основную проблему. Страницы, у которых нет исходящих ссылок, ведут себя таким вот образом, они берут значимость от страниц в веб-графе, не давая ничего взамен. Их называют повисшими узлами.

Прежде чем иметь дело с повисшими узлами, нам нужно добавить некоторые замечания к вероятности в нашей модели. Пред-

ставьте случайного пользователя, который переходит со страницы на страницу. Матрица гиперссылок, H, дает вероятность, что, если пользователь остановится на странице P_i, следующая страница, на которую он зайдет, будет страницей P_j с вероятностью равной $H[i, j]$, где j является любым ненулевым элементом в ряду страницы P_i. Следовательно, для матрицы H, соответствующей рисунку 9.2, если пользователь останавливается на странице 6, существует 1/2 вероятности, что следующая страница, которую посетит пользователь, будет страница 5, и 1/2 вероятности, что следующей страницей будет страница 7. Вопрос в том, что происходит, когда пользователь останавливается на странице 8. На данный момент от нее не исходит ссылок. Чтобы выбраться из тупика, мы решаем, что, когда пользователь прибывает на страницу без исходящих ссылок, он может перейти на любую другую страницу графа с вероятностью $1/n$. Схожим образом можно предположить, что у случайного пользователя есть телепортирующее устройство, которое может переместить своего владельца в любую случайную точку графа, когда тот оказывается в беспросветном тупике. Данное телепортирующее устройство — то же самое, что изменение любого ряда, состоящего из одних нулей, на ряд, состоящий из одних $1/n$.

Чтобы прийти к этому математическим путем, нам надо добавить к H матрицу, чьи ряды полны нулей, за исключением тех рядов, которые соответствуют рядам нулей в H; им мы присваиваем значение $1/n$. Мы называем эту матрицу A, а полученную матрицу — S, и для рисунка 9.3 у нас получается:

$$S = H + A = \begin{bmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 \end{bmatrix} = \begin{bmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}.$$

Матрицу A можно определить, используя вектор столбцов w с элементами:

$$w[i] = \begin{cases} 1, & |P_i| = 0 \\ 0, & \text{иначе} \end{cases},$$

то есть w — это такой вектор столбцов, в котором элемент i равняется нулю, если у страницы P_i есть исходящие ссылки, или же рав-

няется единице, если у страницы P_i их нет. Точно так же элемент i из w равен единице, если все элементы ряда i из H равны нулю, и равен нулю в противном случае. Вместе с w A становится

$$A = \frac{1}{n} w \mathbf{e}^T,$$

где $\mathbf{e}$ — это вектор столбцов из единиц, а e^T — вектор рядов из единиц, поэтому

$$S = H + A = H + \frac{1}{n} w \mathbf{e}^T.$$

Матрица S соответствует новому графу, показанному на рисунке 9.4. Ребра исходного графа нарисованы сплошными четкими линиями, а ребра нового, добавленного — пунктирными.

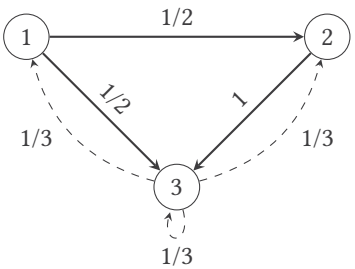


Рисунок 9.4. Повисший узел пейдж-ранка «подключен»

Матрица S является стохастической матрицей; стохастические матрицы — это такие матрицы, в которых все элементы неотрицательные, и сумма элементов в каждом ряду равняется единице. Еще точнее, такие матрицы называются стохастическими матрицами справа; если же сумма элементов в каждом столбце равняется единице, то их называют стохастическими матрицами слева.

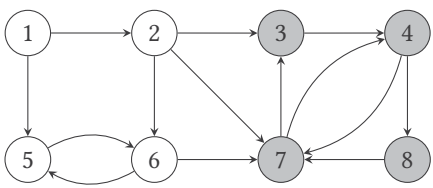


Рисунок 9.5. Веб-граф с разрывным циклом

Их называют так, потому что они дают вероятность перемещению из одной ячейки в другую путем вероятностного, или стохастического, процесса.

Сумма всех вероятностей в данном состоянии, представленных случайным пользователем, находящимся в узле, то есть рядом матрицы S, равна единице, и отрицательных вероятностей не существует.

Мы можем проверить, что тупика, находящегося в узле 3 графа на рисунке 9.3, больше не существует на рисунке 9.4, потому что степенной метод приходит к значениям:

$$\pi^T = \begin{bmatrix} 0.18 & 0.27 & 0.55 \end{bmatrix}.$$

Матрица S решает задачу выхода из тупика, поэтому степенной метод действует корректно, однако случайный пользователь может столкнуться с более хитрой задачкой. Граф на рисунке 9.5 схож с графом на рисунке 9.2 с изъятыми ссылками, идущими от узла 4 к узлам 3 и 2, и ссылкой от узла 7 к узлу 8, замененной ссылкой из узла 8 в узел 7. Теперь, если случайный пользователь попадет в один из узлов 3, 4, 7, 8, он никак не сможет выбраться из данного цикла.

Перед нами снова случай общего характера: как справляться с тупиками, которые появляются в не сильно связном графе. В подобных графах, когда случайный пользователь останавливается в той части графа, которая не ведет к другим частям графа, такая ловушка неизбежна.

Давайте посмотрим, как с ней справится степенной метод. Матрица H для графа с рисунка 9.5 такова:

$$H = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \\ P_7 \\ P_8 \end{array} \begin{array}{c} \begin{array}{cccccccc} P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 \end{array} \\ \begin{bmatrix} 0 & 1/2 & 0 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 0 & 1/3 & 1/3 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}.$$

Здесь нет рядов с нулями, поэтому $S = H$. Тем не менее, если мы применим степенной метод, мы увидим, что он определяет следующие значения:

$$\pi^T = \begin{bmatrix} 0 & 0 & 0.17 & 0.33 & 0 & 0 & 0.33 & 0.17 \end{bmatrix}.$$

Вот что происходит: тупиковый цикл, содержащий узлы 3, 4, 7 и 8, забирает пейдж-ранки других страниц графа и повторяется снова. Возьмем решение, которое мы использовали во время «застревания» в отдельном узле. Мы улучшаем возможности нашего телепортирующего прибора, и случайный пользователь не всегда перескакивает от узла к узлу с помощью матрицы S; пользователь прибегнет к S с вероятностью a между нулем и единицей или же не станет использовать S и просто перепрыгнет куда угодно с вероятностью $1 - a$. Иными словами, пользователь берет случайное число между нулем и единицей. Если число меньше или равно α, пользователь отправится в место, указанное матрицей S. В противном случае телепортирующий прибор перенесет случайного пользователя в другую, случайную страницу графа.

Такая способность телепортирующего прибора в некотором роде схожа, что проделывает каждый человек, когда ищет информацию в сети. На протяжении какого-то времени пользователь переходит по ссылкам от одной странице к другой. Однако в какой-то момент он останавливается и идет на совершенно другую страницу, с помощью закладок, либо вводит в строке новый адрес или же проходит по ссылке, кинутой его друзьями.

9.4. Матрица «Гугла»

Мы можем объяснить все это, прибегнув к новой матрице, G, вместо S. У матрицы G будут такие же вероятности, как мы описывали выше, а значит, ее можно определить как

$$G = \alpha S + (1 - \alpha)\frac{1}{n} J_n,$$

где J_n представляет собой квадратную матрицу $n \times n$, все элементы которой равны 1. Так как $\mathbf{e}$ является вектором столбцов со всеми единицами, а $\mathbf{e}^T$ — это вектор рядов с единицами, мы имеем:

$$J_n = \mathbf{e}\mathbf{e}^T.$$

Таким образом, нам удобнее записать:

$$G = \alpha S + (1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T.$$

Мы делаем так потому, что J_n будучи матрицей $n \times n$ занимает n^2 места, тогда как $\mathbf{e}\mathbf{e}^T$ является произведением двух векторов, каждый из которых занимает n места, которое может оказаться полезным, если нам на протяжении всех вычислений не нужно сохранять всю нашу матрицу $n \times n$. Вы увидите дальше, что все так и происходит.

Матрица G является стохастической, что вытекает из определения G. Возьмите ряд i из матрицы S. В ряду содержится сколько-то, скажем, k, положительных элементов, в то время как остальные равны нулю. Сумма ряда i матрицы S:

$$\sum_{S_{i,j}>0} S_{i,j} + \sum_{S_{i,j}=0} S_{i,j} = k\frac{1}{k} + (n-k)0 = 1.$$

Сумма того же ряда i в матрице G:

$$\sum_{S_{i,j}>0} G_{i,j} + \sum_{S_{i,j}=0} G_{i,j}.$$

Однако первое слагаемое суммы

$$\sum_{S_{i,j}>0} G_{i,j} = \alpha k\frac{1}{k} + (1-\alpha)k\frac{1}{n} = \alpha + (1-\alpha)k\frac{1}{n},$$

а второе слагаемое

$$\sum_{S_{i,j}=0} G_{i,j} = (1-a)(n-k)\frac{1}{n}.$$

Так что сумма целого ряда:

$$\begin{aligned}\sum_{S_{i,j}>0} S_{i,j} + \sum_{S_{i,j}=0} S_{i,j} &= \alpha + (1-\alpha)k\frac{1}{n} + (1-a)(n-k)\frac{1}{n} \\ &= \frac{\alpha n + (1-\alpha)k + (1-\alpha)(n-k)}{n} \\ &= \frac{\alpha n + k - \alpha k + n - k - \alpha n + \alpha k}{n} \\ &= 1.\end{aligned}$$

У G есть еще одно важное свойство — примитивная матрица. Матрица M примитивна, если для некой степени p все элементы матрицы M^p являются положительными числами. Это очевидно. Все нулевые элементы S были превращены в положительные числа со значением $(1 - \alpha)1/n$, поэтому g примитивна для $p = 1$.

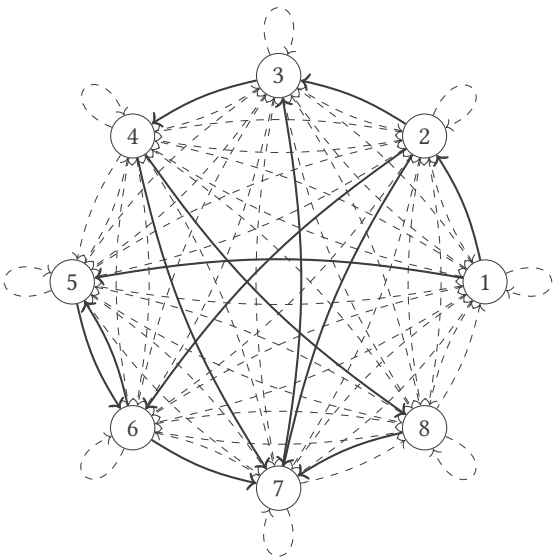


Рисунок 9.6. Граф, соответствующий G, для рисунка 9.5

Граф, соответствующий G, показан на рисунке 9.6, он отличается от исходного графа на рисунке 9.5: мы снова обозначаем ребра исходного графа четкими линиями, а все добавленные ребра — пунктирными; мы видим, что G является на самом деле полным графом.

Теперь мы подходим к ключевому моменту: линейная алгебра показывает, что, когда матрица является стохастической и примитивной, степенной метод определяет уникальный вектор положительных значений; более того, данный вектор содержит компоненты, сумма которых равна единице. К тому же совершенно не важно, с какого вектора мы начинаем. Поэтому, несмотря на то, что мы стартовали с начального вектора с $1/n$ для всех значений пейдж-ранка, мы с тем же успехом могли бы взяться за любой

другой вектор и получили бы точно такой же результат. Следовательно, если степенной метод применяется после k циклов для матрицы g мы имеем:

$$\pi^T G = 1\pi^T$$

с

$$\pi_1 > 0, \pi_2 > 0, \ldots, \pi_n > 0$$

и

$$\pi_1 + \pi_2 + \cdots + \pi_n = 1.$$

Эти условия соответствуют ряду разумных значений для наших пейдж-ранков, так как все они положительные.

Вернемся к нашему примеру, матрица G на рисунке 9.6 для $\alpha = 0{,}85$ будет:

$$G = \begin{bmatrix} \frac{3}{160} & \frac{71}{160} & \frac{3}{160} & \frac{3}{160} & \frac{71}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} \\ \frac{3}{160} & \frac{3}{160} & \frac{29}{96} & \frac{3}{160} & \frac{3}{160} & \frac{29}{96} & \frac{29}{96} & \frac{3}{160} \\ \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{139}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} \\ \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{71}{160} & \frac{71}{160} \\ \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{139}{160} & \frac{3}{160} & \frac{3}{160} \\ \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{71}{160} & \frac{3}{160} & \frac{71}{160} & \frac{3}{160} \\ \frac{3}{160} & \frac{3}{160} & \frac{71}{160} & \frac{71}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} \\ \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{3}{160} & \frac{139}{160} & \frac{3}{160} \end{bmatrix}$$

Если мы запустим несколько итераций, то увидим, что степенной метод теперь находит вектор пейдж-ранка:

$$\pi^T = \begin{bmatrix} 0.02^+ & 0.03^+ & 0.15^+ & 0.26^- & 0.06^+ & 0.08^+ & 0.28^- & 0.13^- \end{bmatrix}.$$

Плюсы и минусы рядом с числами показывают, получаем ли мы при округлении результатов до сотых долей число больше или меньше, чем реальное значение. Обратите внимание, что, если мы

складываем числа с учетом такого округления, их сумма будет 1,01, а не 1; это куда удобней, чем иметь дело с кучами цифр тут и там.

Матрица G называется матрицей «Гугла». Матрица «Гугла» огромна; для всей всемирной компьютерной сети она насчитывает миллиарды и миллиарды рядов и столбцов. Стоит посмотреть, можно ли что-то сохранить в памяти. Вспомните, что

$$G = \alpha S + (1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T,$$

поэтому мы имеем

$$\pi_{k+1}^T = \pi_k^T\left(\alpha S + (1-\alpha)\frac{1}{n}\mathbf{e}_{n\times n}\right).$$

Помните также, что

$$S = H + A = H + \frac{1}{n}w\mathbf{e}^T,$$

поэтому мы получаем:

$$\begin{aligned}
\pi_{k+1}^T &= \pi_k^T\left(\alpha H + \alpha\frac{1}{n}w\mathbf{e}^T + (1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T\right)\\
&= \alpha\pi_k^T H + \pi_k^T\left(\alpha w\mathbf{e}^T\frac{1}{n} + (1-\alpha)\mathbf{e}\mathbf{e}^T\frac{1}{n}\right)\\
&= \alpha\pi_k^T H + \pi_k^T\left(\alpha w + (1-\alpha)\mathbf{e}\right)\mathbf{e}^T\frac{1}{n}\\
&= \alpha\pi_k^T H + \left(\pi_k^T\alpha w + (1-\alpha)\pi_k^T\mathbf{e}\right)\mathbf{e}^T\frac{1}{n}\\
&= \alpha\pi_k^T H + \left(\pi_k^T\alpha w + (1-\alpha)\right)\mathbf{e}T\frac{1}{n}\\
&= \alpha\pi_k^T H + \pi_k^T\alpha w\mathbf{e}^T\frac{1}{n} + (1-\alpha)\mathbf{e}^T\frac{1}{n}.
\end{aligned}$$

В предпоследней строке мы использовали факт, что $\pi_k^T\mathbf{e} = 1$, потому что это произведение равно сумме всех пейдж-ранков, которая равна единице. В последней строке обратите внимание на то, что $\alpha w\mathbf{e}^T(1/n)$ и $(1 - \alpha)\mathbf{e}^T$ являются на самом деле постоянными значениями, и их нужно вычислить всего один раз. Далее при проходе каждой итерации нам нужно лишь умножить π_k на

постоянное значение и добавить к результату другое постоянное значение. Более того, матрица H чрезвычайно разрежена, так как каждая веб-страница имеет около десяти ссылок на другие страницы вместо матрицы G, которая чрезвычайно плотная (вся заполненная). Поэтому нам на самом деле вовсе не нужно заполнять G, и общее число операций намного меньше, чем если бы мы использовали определение G.

Примечания

Пейдж-ранк, изобретенный Ларри Пейджем и Сергеем Брином, был опубликован в 1998 году и назван в честь Ларри Пейджа [29]. Он стал первым алгоритмом, который использовала поисковая система «Гугл»; сегодня «Гугл» применяет в своей работе большее количество алгоритмов, но какие именно — не афишируется. Тем не менее пейдж-ранк все еще играет ведущую роль из-за своей простоты и потому что он сформировал часть основы одной из громаднейших корпораций в истории человечества.

Если вас интересует математическая составляющая, лежащая в основе пейдж-ранка, советуем обратиться к труду Брайана и Лейзе [31]. Доступным и понятным введением в тему пейдж-ранка и поисковых систем в целом, является книга Лангвиля и Мейера [121]; посмотрите также небольшую книжку Берри и Брауни [18]. Книга Мэннинга, Рагхавана и Хинриха Шюце в более общих чертах освещает тему нахождения информации, включающую в себя порядок индексирования. Бюттхен, Кларк и Кормак рассказывают о нахождении информации, их объяснения качественны и точны [33].

Пейдж-ранк — не единственный алгоритм, который использует ссылки для ранжирования. В данной области есть еще один значимый алгоритм — HITS*, изобретенный Джоном Клейнбергом [108], [109].

* HITS — Hyperlink Induced Topic Distillation, осуществляемая посредством гиперссылок тематическая дистилляция

10 Прочность голосования

Компания собирается нанять повара, который каждый день готовил бы сотрудникам обеды. Проведя небольшое исследование рынка услуг, компания нашла трех кандидатов, которые оптимально соответствуют параметрам качества и цены. Но есть одна загвоздка. Первый кандидат — назовем его *M* — готовит преимущественно мясные блюда, такова его специализация. Второй повар, *P*, предлагает разные блюда, как мясные, так и вегетарианские. Третий же соискатель, *B*, специализируется исключительно на вегетарианском меню.

Отдел кадров решает провести среди сотрудников компании опрос о гастрономических предпочтениях и организует голосование. Результаты следующие: 40% проголосовали за *M*, 30% — за *P* и еще 30% — за *B*. Поэтому решили нанять повара *M*.

Заметили проблему? Вегетарианцы, работающие здесь, вряд ли одобрят подобное решение, но им придется смириться, или же придется ходить на работу с собственным обедом.

Если вы приглядитесь, то увидите, что проблема появилась не сама по себе: ее породило голосование. Хотя 30% сотрудников выбрали *B*, некоторые из них предпочли бы *P* вместо *M*. А сотрудники, проголосовавшие в пользу *M*, скорее всего, выбрали бы *P* вместо *B*. И наконец те, кто отдал свой голос в пользу *P*, на самом деле согласился бы на *B* вместо *M*. Давайте составим таблицу всех предпочтений:

40%: [M, P, B]
30%: [B, P, M]
30%: [P, B, M]

А теперь давайте подсчитаем попарные предпочтения, то есть каждую пару выбранных предпочтений, сколько голосующих предпочитают одно другому? Мы начинаем с исследования *M* и *P* для групп голосующих: 40% сотрудников предпочитают *M*

вместо *P* и ставят *M* в первую позицию. В то же время 30% голосующих предпочитают *P* вместо *M*, хотя их первый выбор — *B*, и оставшиеся 30% сотрудников предпочитают *P* вместо *M* и ставят *P* своим первым выбором. В итоге получается, что 60% проголосовавших отдают предпочтение *P*, а не *M*, так что *P* выигрывает у *M* со счетом 60% против 40%.

Сравним *M* и *B*. Логика все та же: 40% сотрудников предпочитают *M*, а не *B*, и своим первым выбором указывают *M*. В то же время 30% голосующих предпочитают *B* и ставят его в первую позицию, и другие 30% голосующих, указавшие первым пунктом *P*, так же предпочитают *B*, а не *M*. Так что *B* выигрывает у *M* со счетом 60% против 40%.

И наконец, сравним *P* и *B*. 40% сотрудников выбрали *M* и предпочитают *P* вместо *B*, и 30% сотрудников выбрали *B* и предпочитают *B* вместо *P*. 30% голосующих, кто указал первым пунктом *P*, предпочитают *P* вместо *B*, что и так ясно. Таким образом, *P* выигрывает у *B* со счетом 70% против 30%.

Подытоживая наши расчеты, мы получаем, что в попарном сравнении *P* побеждает своих конкурентов: *M* — 60% к 40% и B 70% к 30%, — в то время как *B* обходит лишь одного конкурента, а *M* остается в хвосте предпочтений. Так что победителем голосования объявляется *P*.

10.1. Система голосования

Данный пример обозначает проблему, которая существует в хорошо известной и популярной избирательной системе, основанной на получении большинства голосов. В данной системе голосующие отмечают в бюллетене единственного кандидата. Кандидат, набравший большинство голосов, побеждает в выборах. Проблема с таким видом голосования в том, что избирателям не дают возможности составить список предпочитаемых кандидатов, они могут выбрать лишь одного. Поэтому может случиться так, что один кандидат будет желаннее другого, но не победит в выборах, потому что большинство голосующих не указали его как самого предпочтительного из всех.

Условие, по которому победителем выборов становится кандидат, чью популярность сравнили с популярностью каждого другого кандидата, называется критерием Кондорсе. Победителя называют кандидатом Кондорсе, или победителем Кондорсе. Имя принадлежит Мари Жану Антуану Николя де Кондорсе, французскому математику и философу XVIII века, именно он в 1785 году описал данную проблему.

Чтобы критерий Кондорсе не представлялся вам неким мистическим мерилом, относящимся к Франции XVIII века, вспомните о двух примерах, которые мы только что обсуждали.

В 2000 году в США прошли президентские выборы, на которых избирателям предлагалось выбрать одного из трех кандидатов: Джорджа У. Буша, Альберта Гора или Ральфа Нейдера. В США президента избирает Коллегия выборщиков. Она основывается на результатах выборов по отдельным штатам США. Опуская все сопутствующие страсти и драмы, скажем, что результаты выборов 2000 года определились голосами избирателей штата Флорида. Итоговые результаты выглядели так:

- Джордж У. Буш получил 2 912 790 голосов, что равно 48,847% всех избирателей.
- Альберт Гор получил 2 912 253 голоса, что равно 48,838% всех избирателей.
- Ральф Нейдер получил 97 421 голос, что равно 1,634% от всех избирателей.

Джордж У. Буш победил с крохотным перевесом в 537 голосов, или 0,009% избирателей, живущих во Флориде. Однако есть основания полагать, что большинство проголосовавших за Ральфа Нейдера предпочли бы Джорджу У. Бушу Альберта Гора. Если это так, то при использовании на президентских выборах метода, позволяющего составлять список из двух желаемых кандидатов, победил бы Альберт Гор.

Сменим континент и отправимся во Францию, на президентские выборы 2002 года, состоявшиеся 21 апреля. Гражданам предлагалось сделать выбор между Жаком Шираком, Лионелем Жоспеном и четырнадцатью другими кандидатами. Чтобы стать

президентом, кандидат должен набрать более 50% голосов. Если такого не случается, тогда устраивается второй тур выборов, в котором участвуют два кандидата, набравшие в первом тур большие всего голосов. Весь мир оказался потрясен, узнав, что во второй тур вместе с Жаком Шираком вместо Лионеля Жоспена прошел ультраправый Жан-Мари Ле Пен.

Вовсе не означает, что большинство французов поддержали ультраправого кандидата. Результаты первого тура, прошедшего 21 апреля 2002 года, были таковы:

- Жак Ширак получил 5 666 440 голосов, что равно 19,88% всех избирателей;
- Жан-Мари Ле Пен получил 4 805 307 голосов, что равно 16,86% всех избирателей;
- Лионель Жоспен получил 4 610 749 голосов, что равно 16,18% всех избирателей.

Во втором туре, который прошел 5 мая 2002 года, Жак Ширак получил более 82% голосов, в то время как Жан-Мари Ле Пен получил менее 18%. Мы видим, что Ширак получил поддержку от избирателей, ранее проголосовавших за других четырнадцать кандидатов, а результаты Ле Пена едва отличны от результатов первого тура.

Такая проблема возникла, потому что изначально выбирали из шестнадцати кандидатов, а затем — из двух популярнейших, прошедших во второй тур. Когда имеется шестнадцать кандидатов, то несложно наткнуться на такую ситуацию, при которой голоса между ними разделятся таким образом, что в лидеры, обойдя умеренных конкурентов, выйдет экстремистский кандидат, которого поддерживают только его ярые сторонники. В целом получается так, что кандидат, который нравится большинству людей, но не является первым в списке их предпочтений, проиграет кандидату, которого ненавидят, но который стоит первым в списке верного ему меньшинства.

Избирательная система, основанная на большинстве голосов, не единственная система, которая не соответствует критерию Кондорсе, однако именно на нее набрасываются критики: ведь она применятся почти повсеместно.

В одобрительном голосовании избиратели могут отметить в бюллетени не одного, а нескольких кандидатов. Победителем становится тот, что получил большинство голосов. Представим, что у нас на выборах три кандидата, *A*, *B* и *C*, и мы получили следующие результаты (мы не применяем к данным бюллетеням квадратные скобки, дабы подчеркнуть, что порядок выбора не имеет значения):

60%: (A, B)

40%: (C, B)

Так как за *B* проголосовали все 100% избирателей, *B* побеждает. Предположим, что 60% голосовавших предпочитают *A* вместо *B* и *B* вместо *C*, а 40% — *C* вместо *B* и *B* вместо *A*, но они не могут отобразить приоритет в бюллетени. Иными словами, будь у них такая возможность, бюллетень выглядела бы так:

60%: [A, B, C]

40%: [C, B, A]

Мы видим, что *A* выигрывает у *B* со счетом 60% против 40%; таким образом, хотя большинство избирателей предпочитает *A*, а не *B*, *A* остается не у дел.

Есть еще один способ голосования — правило Борда, названное так в честь другого француза, жившего в XVIII веке: математика и политолога Жана-Шарля де Борда, который описал данный метод в 1770 году. Согласно системе Борда, избиратели оценивают кандидатов определенным количеством баллов. Если имеется n кандидатов, первый, самый желанный кандидат списка получает $n - 1$ баллов, второй — $n - 2$ и так до самого последнего, ненавистного кандидата, который получает ноль баллов. Победителем становится тот, кто наберет больше всех баллов. Представим, что у нас три кандидата, *A*, *B* и *C*, и результаты голосования следующие:

60%: [A, B, C]

40%: [B, C, A]

Если у нас $100m$ избирателей, тогда кандидат *A* получает $(60 \times 2)m = 120m$ баллов, кандидат *B* получает $(60 + 2 \times 40)m = 140m$ баллов, а кандидат *C* получает $40m$ баллов. Побеждает кандидат *B*. Однако большинство избирателей предпочли кандидата *A* кандидату *B*.

Вернемся к критерию Кондорсе, задача здесь в том, чтобы найти способ, который вычислит победителя Кондорсе, если такой вообще есть. Победители Кондорсе может и не существовать. Например, у нас три кандидата, *A*, *B* и *C*, для которых мы получили следующие бюллетени:

30: [A, B, C]

30: [B, C, A]

30: [C, A, B]

Если мы проведем попарное сравнение, мы увидим, что *A* выигрывает у *B* со счетом 60 против 40, *B* выигрывает у *C* со счетом 60 против 40 и *C* выигрывает у *A* со счетом 60 против 40. Хотя каждый кандидат смог обойти одного соперника, никто из них не смог победить всех конкурентов, поэтому победителя нет.

Теперь рассмотрим другие выборы с тремя кандидатами, где бюллетени распределились в ином порядке:

10 × [A, B, C]

5 × [B, C, A]

5 × [C, A, B]

A обходит *B* с результатом 15 против 5, B обходит *C* со результатом 15 против 5, а результаты *C* и *A* равны, у каждого по 10. Так как и *A*, и *B* выигрывают в попарном сравнении, у нас опять нет победителя.

Это немного странно, потому что в отличие от предыдущего примера здесь бюллетени распределились не поровну. Ясно видно, что большинство избирателей высказались за вариант, указанный в первом бюллетене, однако это никак не повлияло на результаты голосования. Так что нас бы заинтересовал метод, который учитывает критерий Кондорсе, но где равный счет встречается не так часто, как в простом методе, который мы использовали прежде.

10.2. Метод Шульце

Метод, с помощью которого можно определить на выборах победителя Кондорсе, если таковой имеется, существует — это метод

Шульце, разработанный в 1997 году Маркусом Шульце. Данный способ используется в технологических организациях и избегает равных результатов. Его суть в том, что на основе попарных предпочтений избирателей мы строим граф. Предпочтения между кандидатами выявляются с помощью отслеживания путей данного графа.

Первый этап в методе Шульце — нахождение попарных предпочтений для кандидатов. Представим, что у нас n бюллетеней с m кандидатами. Бюллетени — это $B = B_1, B_2,\ldots, B_n$, а кандидаты — это $C = c_1, c_2,\ldots, c_m$. Мы поочередно берем каждый бюллетень B_i, где $i = 1, 2,\ldots n$. В каждом бюллетене указаны имена кандидатов, в порядке убывания предпочтения, таким образом, кандидат, отмеченный в бюллетене прежде других кандидатов и стоящий впереди них, более желаем избирателю, который ранжирует в бюллетени свои предпочтения. Иными словами, для каждой пары кандидатов c_j и c_k в бюллетене B_i избиратель предпочитает c_j вместо c_k, если c_j стоит перед c_k. Для подведения итогов мы используем массив P размером $m \times m$. Чтобы подсчитать содержимое массива, мы сперва устанавливаем все значения на ноль. Затем, когда мы рассматриваем каждый бюллетень B_i для каждой пары кандидатов c_j и c_k, где c_j идет прежде, чем c_k, мы добавляем единицу к элементу $P[c_j, c_k]$ массива P. Когда мы перебрали все бюллетени, каждый элемент из P, $P[c_j, c_k]$ покажет нам, сколько всего избирателей предпочитают кандидата c_j кандидату c_k.

Например, у нас бюллетень $[c_1, c_3, c_4, c_2, c_5]$, что значит, избиратель предпочитает кандидата c_1 всем остальным кандидатам, кандидат c_3 более желаем, чем кандидаты c_4, c_2, c_5, кандидат c_4 лучше кандидатов c_2, c_5, а кандидат c_2 предпочтительней кандидата c_5. В данном бюллетене кандидат c_1 идет прежде кандидатов c_3, c_4, c_2 и c_5, поэтому мы добавляем единицу к элементам $P[c_1, c_3]$, $P[c_1, c_4]$, $P[c_1, c_2]$ и $P[c_1, c_5]$. Кандидат c_3 предшествует кандидатам c_4, c_2 и c_5, поэтому мы добавляем единицу к элементам $P[c_3, c_4]$, $P[c_3, c_2]$ и $P[c_3, c_5]$, и так далее, вплоть до кандидата c_2, когда мы добавляем единицу к элементу $P[c_2, c_5]$:

$$\begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{array} \begin{array}{c} \begin{array}{ccccc} c_1 & c_2 & c_3 & c_4 & c_5 \end{array} \\ \begin{bmatrix} - & +1 & +1 & +1 & +1 \\ - & - & - & - & +1 \\ - & +1 & - & +1 & +1 \\ - & +1 & - & - & +1 \\ - & - & - & - & - \end{bmatrix} \end{array}.$$

Именно это проделывает алгоритм 10.1.

Алгоритм 10.1. Подсчет попарных предпочтений

`CalcPairwisePreferences(`*ballots*, *m*`)` $\rightarrow P$

Вводные данные: *ballots*, массив с бюллетенями, в котором каждый бюллетень — это массив с кандидатами; *m*, число кандидатов.

Выводимые данные: *P*, массив размером $m \times m$ с попарными предпочтениями кандидатов; $P[i, j]$ — число избирателей, которые предпочитают кандидата *i* кандидату *j*.

```
1  P ← CreateArray(m · m)
2  for i ← 0 to m do
3      for j ← 0 to m do
4          P[i,j] ← 0
5  for i ← 0 to |ballots| do
6      ballot ← ballots[i]
7      for j ← 0 to |ballot| do
8          c_j ← ballot[j]
9          for k ← j + 1 to |ballot| do
10             c_k ← ballot[k]
11             P[c_j, c_k] ← P[c_j, c_k] + 1
12 return P
```

В строчке 1 алгоритм создает массив *P*, в котором будут содержаться попарные предпочтения, а в строчках 2–4 ставит значения для всех попарных предпочтений на 0. Это занимает $\Theta(|B|^2)$ времени. Затем, в цикле в строчках 5–11, алгоритм берет каждую бюллетеня $|B|$. Во вложенном цикле в строчках 7–11 для каждого бюллетеня он поочередно перебирает каждого кандидата. Так как кандидаты в бюллетенях указаны в порядке предпочтения, то

каждый раз, когда мы доходим до вложенного цикла, выбранный кандидат оказывается предпочтительней всех следующих за ним в бюллетене кандидатов. Поэтому, если попадается кандидат c_j, то для каждого другого кандидата c_k, который идет за c_j в бюллетене, мы добавляем единицу к элементу $P[c_j, c_k]$. Алгоритм по порядку проделывает то же самое со всеми прочими кандидатами, стоящими во главе списка бюллетене. Если в бюллетене содержатся все $|C|$ кандидатов, он обновит данные в массиве предпочтений $(|C| - 1) + (|C| - 2) + \ldots + 1 = |C|(|C| - 1)/2$ раз. В наихудшем случае, когда во всех бюллетенях содержится $|C|$ кандидатов, требуемое на каждую бюллетень время равно $O(|C|(|C| - 1)/2) = O(|C|^2)$, а время для всех бюллетеней — $O(|B||C|^2)$. В итоге алгоритм 10.1 выполняется за $O(|C|^2 + |B|^2)$ времени.

Для примера возьмем выборы с четырьмя кандидатами, *A*, *B*, *C* и *D*.

В выборах принимают участие 21 избирателей. При подсчете результатов мы увидели, что бюллетени распределились следующим образом:

6 × [A,C,D,B]
4 × [B,A,D,C]
3 × [C,D,B,A]
4 × [D,B,A,C]
4 × [D,C,B,A]

То есть шесть бюллетеней [*A*,*C*,*D*,*B*], четыре бюллетеня [*B*,*A*,*D*,*C*] и так далее. В первых шести бюллетенях избиратели предпочли кандидата *A* кандидату *C*, кандидата *C* кандидату *D* и кандидата *D* кандидату *B*.

Чтобы подсчитать массив предпочтений, мы находим, что кандидат *A* более желаем, нежели кандидат *B*, только в первой подборке бюллетеней, поэтому элемент в массиве для попарных предпочтений между *A* и *B* будет равен 6. Точно так же мы видим, что кандидат *A* более желаем, чем кандидат *C*, в первой, второй и четвертой подборках бюллетеней, поэтому попарные предпочтения между *A* и *C* будут 14. Продолжая в том же духе мы получаем следующий массив предпочтений для кандидатов наших выборов:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 6 & (6+4+4) & (6+4) \\ (4+3+4+4) & 0 & (4+4) & 4 \\ (3+4) & (6+3+4) & 0 & (6+3) \\ (3+4+4) & (6+3+4+5) & (4+4+4) & 0 \end{bmatrix} \end{array},$$

то есть

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 6 & 14 & 10 \\ 15 & 0 & 8 & 4 \\ 7 & 13 & 0 & 9 \\ 11 & 17 & 12 & 0 \end{bmatrix} \end{array}.$$

Второй этап метода Шульце — построение графа, в котором узлами являются кандидаты, а весами ребер — интервалы предпочтений одного кандидата другому. Если для двух кандидатов c_i и c_j число избирателей $P[c_i, c_j]$, которые предпочитают c_i вместо c_j, больше числа избирателей $P[c_j, c_i]$, кто предпочитает c_j вместо c_i, мы добавляем ребро $c_i \to c_j$ и назначаем число $P[c_i, c_j] - P[c_j, c_i]$ весом ребра $c_i \to c_j$. Мы используем для других пар $-\infty$, дабы показать, что соответствующего ребра не существует. Для этого, когда нужно, мы проводим сравнения и операции:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & (6 < 15) & (14 - 7) & (10 < 11) \\ (15 - 6) = 9 & 0 & (8 < 13) & (4 < 17) \\ (7 < 14) & (13 - 8) & 0 & (9 < 12) \\ (11 - 10) = 1 & (17 - 4) & (12 - 9) & 0 \end{bmatrix} \end{array},$$

а затем заменяем отрицательные и нулевые элементы на $-\infty$:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} -\infty & -\infty & 7 & -\infty \\ 9 & -\infty & -\infty & -\infty \\ -\infty & 5 & -\infty & -\infty \\ 1 & 13 & 3 & -\infty \end{bmatrix} \end{array}.$$

Вы можете увидеть соответствующий граф на рисунке 10.1. Как и говорилось ранее, узлами графа представлены кандидаты, а ребрами — положительные интервалы разницы предпочтений между каждой парой кандидатов.

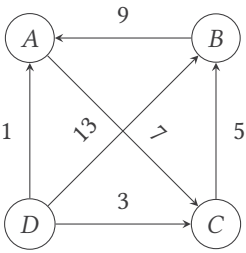


Рисунок 10.1. Граф выборов

Построив граф, мы идем дальше и вычисляем прочнейшие пути между всеми узлами графа предпочтений. Мы определяем прочность пути, или, по-другому, ширину пути, как минимальный вес в ребрах, составляющих этот путь. Если вы представите себе путь в виде ряда мостов, соединяющих узлы, то путь будет настолько прочен, насколько прочно слабейшее ребро, то есть мост. Между двумя узлами графа предпочтений может существовать несколько путей, каждый со своей прочностью. Путь с наибольшей прочностью является прочнейшим из всех путей графа. Вернемся к метафоре с мостами, прочнейшим путем является путь, который позволяет нам перевезти тяжелейший груз между двумя узлами. На рисунке 10.2 изображен граф с двумя прочнейшими путями. Прочнейший путь между узлом 0 и узлом 4 проходит через узел 2 и имеет прочность 5; точно так же прочнейший путь между узлом 4 и узлом 1 проходит через узел 3 и имеет прочность 7.

Задача нахождения прочнейшего пути возникает и в других областях. В компьютерных сетях это то же самое, что нахождение пропускной способности между двумя компьютерами в Интернете, когда соединение между двумя любыми машинами или маршрутизаторами в сети имеет ограниченную пропускную способность. Ее еще называют задачей о широчайшем пути, так как прочность здесь является синонимом ширины, и задачей о максимальной пропускной способности пути, потому что она упрощает нахождение максимальной пропускной способности пути в графе. Максимальная пропускная способность пути ограничена его слабейшим ребром; максимальная пропускная способ-

ность пути между двумя узлами — это максимальная пропускная способность из всех пропускных способностей путей между двумя узлами.

Для нахождения прочнейшего пути между всеми парами узлов в графе мы руководствуемся следующим. Мы берем все узлы графа по очереди, c_1, c_2,…, c_n. Находим прочнейший путь между всеми парами узлов, c_i и c_j в графе с помощью нулевых промежуточных узлов из очереди c_1, c_2,…, c_n. Прочнейший путь без промежуточных узлов между двумя узлами существует, если есть ребро, напрямую соединяющее c_i и c_j; в противном случае такого пути нет.

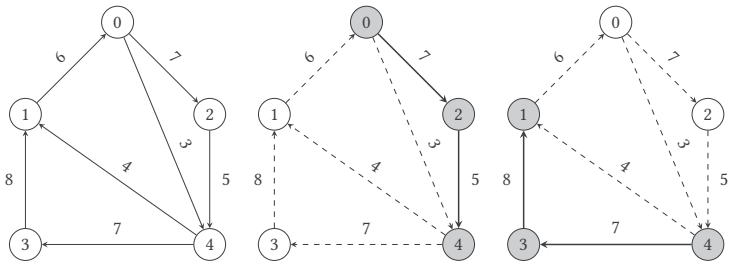


Рисунок 10.2. Примеры прочнейшего пути

Мы продолжаем поиск прочнейшего пути между всеми парами узлов c_i и c_j, используя первый из узлов в очереди, узел c_1, в качестве промежуточного узла. Если мы уже нашли прочнейший путь между c_i и c_j на предыдущем этапе, то, если существует путь $c_i \to c_1$ и $c_1 \to c_j$, мы сравниваем прочность пути $c_i \to c_j$ с прочностью пути $c_i \to c_1 \to c_j$ и берем прочнейший из них, утверждая его новым прочнейшим путем между c_i и c_j.

Мы исполняем все ту же процедуру до тех пор, пока не переберем все n узлов в очереди. Представим, что мы нашли прочнейший путь между всеми парами узлов графа, c_i и c_j, используя первые k узлов очереди в качестве промежуточных узлов. Теперь мы пробуем найти прочнейший путь между двумя узлами c_i и c_j, используя первые $k + 1$ узлов в очереди. Если мы находим путь от c_i до c_j с помощью $(k + 1)$ узла, в ней будет две части. Первая часть будет путем от c_i до c_{k+1} с использованием первых k узлов очереди

в качестве промежуточных, а вторая часть будет от c_{k+1} до c_j с использованием первых k узлов очереди в качестве промежуточных. Мы уже нашли прочность этих двух путей на предыдущем этапе процедуры. Если мы называем $s_{i,j}(k)$ прочностью пути от c_i до c_j с использованием первых k узлов, два пути, использующие узел $k + 1$ будут $s_{i,k+1}(k)$ и $s_{k+1,j}(k)$. Прочность пути, идущая от c_i к c_j через c_{k+1}, по определению является наименьшей прочностью путей $s_{i,k+1}(k)$ и $s_{k+1,j}(k)$, и мы уже нашли их ранее. Поэтому:

$$s_{i,j}(k+1) = max\Big(s_{i,j}(k), min\big(s_{i,k+1}(k), s_{k+1,j}(k)\big)\Big).$$

В конце, перебрав все n узлов в очереди, мы найдем все прочнейшие пути между всеми парами узлов в графе. Алгоритм 10.2 детально описывает данную процедуру.

В строчках 1–2 мы создаем два массива выводимых данных. В строчках 2–10 задаем начальную прочность пути между двумя узлами, она равна весам ребер, напрямую соединяющих данные узлы, если такие ребра существуют. Они отвечают за нахождение прочнейших путей без использования промежуточных узлов. Затем в строчках 11–18 мы вычисляем прочнейший путь с помощью все большего и большего количества промежуточных узлов. Внешний цикл, над переменной k, отвечает за промежуточный узел, который мы добавляем к нашему набору промежуточных узлов. Каждый раз, когда мы увеличиваем k, мы проверяем все пары узлов, обозначенных i и j, и в строчках 14–16 корректируем, если нужно, данные прочнейшего пути. Мы также отслеживаем наши пути с помощью массива *pred*, который в каждой позиции (i, j) выдает предыдущий узел вместе с прочнейшим путем от узла i до узла j.

Алгоритм работает успешно. Первый цикл в строчках 2–10 выполняется n^2 раз, а второй цикл в строчках 11–18 выполняется n^3 раз, где n — количество вершин графа. Так как граф представлен в виде матрицы смежности, все операции графа выполняются за постоянное время, поэтому в общей сложности требуется $\Theta(n^3)$ времени. Если мы вернемся к выборам, то n будет количеством кандидатов. В нотации, которую мы позаимствовали, требуемое время равно $\Theta(|C|^3)$.

Вы можете увидеть ход работы алгоритма в нашем примере на рисунке 10.3. На каждом изображении мы отмечаем серым цветом промежуточный узел, который используется для создания нового пути на каждом этапе. Обратите внимание, что на последнем этапе, когда мы добавляем узел D к нашему набору промежуточных узлов, мы не находим пути более прочного, чем уже имеем. Так может случиться, но заранее никогда не предугадаешь. Такое могло бы произойти и на более ранних этапах, хотя в нашем случае не произошло.

Алгоритм 10.2. Вычисление прочнейших путей

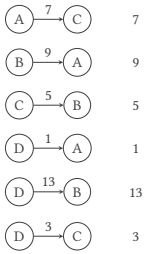
`CalcStrongestPaths`$(W, n) \rightarrow (S, pred)$

Вводные данные: W, массив размером $n \times n$, представляющий матрицу смежности графа, $W[i, j]$ является весом ребра между узлами i и j; n, размер каждой величины W.

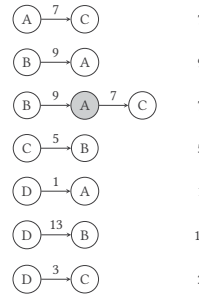
Выводимые данные: S, такой массив размером $n \times n$, что $S[i, j]$ является прочнейшим путем между узлами i и j; $pred$, такой массив размером $n \times n$, что $[i, j]$ является предшественником узла i в прочнейшем пути к узлу j.

```
1  S ← CreateArray(n · n)
2  pred ← CreateArray(n · n)
3  for i ← 0 to n do
4      for j ← 0 to n do
5          if W[i, j] > W[j, i] then
6              S[i, j]  ← W[i, j] − W[j, i]
7              pred[i, j] ← i
8          else
9              S[i],[j] ← −∞
10             pred[i, j] ← −1
11 for k ← 0 to n do
12     for i ← 0 to n do
13         if i ≠ k then
14             for j ← 0 to n do
15                 if j ≠ i then
16                     if S[i, j] < Min(S[i, k], S[k, j]) then
17                         S[i, j] ← Min(S[i, k], S[k, j])
18                         pred[i, j] ← pred[k, j]
19 return (S, pred)
```

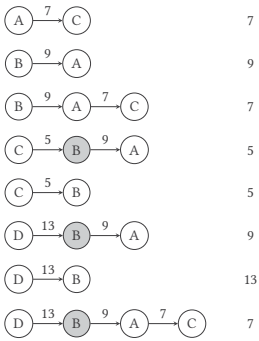
Как бы то ни было, мы выполняем алгоритм до конца, поочередно исследуя все узлы графа в качестве промежуточных.



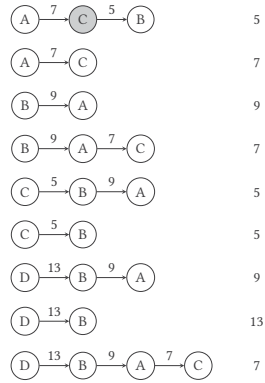
(a) Промежуточных узлов нет



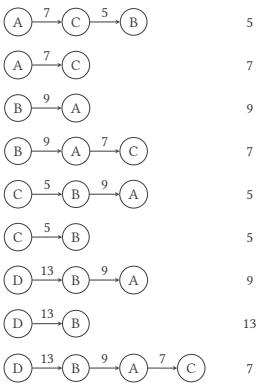
(b) Промежуточный узел A



(c) Промежуточные узлы A, B



(d) Промежуточные узлы A, B, C



(e) Промежуточные узлы A, B, C, D

Рисунок 10.3. Вычисление прочнейших путей

Полученные из алгоритма 10.2 пути возвращаются через массив *pred*. Если хотите увидеть пути на графе, посмотрите на рисунок 10.4. Убедитесь, что прочнейший путь от узла *D* к узлу *C* проходит через узлы *B* и *A*, а не по прямому пути от *D* к *C*; на третьем изображении ранние данные обновляются на более актуальные, когда мы используем узел *B* в качестве промежуточного. В то же время, когда мы добавляем *B* в качестве промежуточного узла, мы можем получить путь $D \rightarrow A \rightarrow B$ с прочностью 1; но у нас уже есть путь $D \rightarrow A$ с прочностью 1, поэтому нет нужды обновлять данные о прочнейшем пути между *A* и *D*. Аналогичным образом мы можем получить путь $D \rightarrow A \rightarrow C$ с прочностью 1; но у нас уже есть путь $D \rightarrow C$ с прочностью 3, так что прочнейший путь между *D* и *C* остается неизменным.

Алгоритм вычисляет части для решения нашей задачки и шаг за шагом складывает их, чтобы получить полное решение. Он находит короткие прочнейшие пути и выстраивает из них, если возможно, длинные прочнейшие пути. Такой подход, когда сперва решаются кусочки общей задачи, а затем, путем преобразования результатов, находится полное решение, называется динамическим программированием, он лежит в основе многих примечательных алгоритмов.

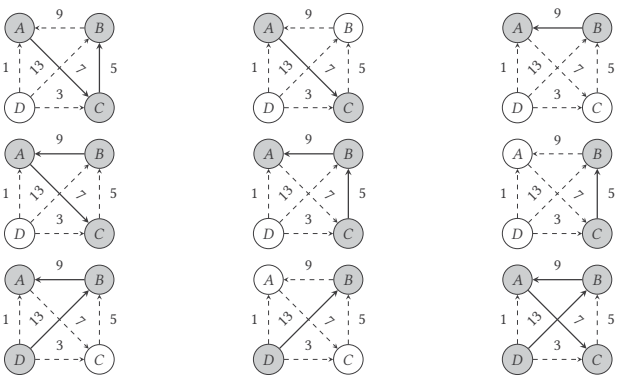


Рисунок 10.4. Прочнейшие пути для графа выборов

В нашем примере алгоритм 10.2 создает массив *S* с прочностью прочнейших путей между всеми парами узлов:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array}\begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} -\infty & 5 & 7 & -\infty \\ 9 & -\infty & 7 & -\infty \\ 5 & 5 & -\infty & -\infty \\ 9 & 13 & 7 & -\infty \end{bmatrix} \end{array}.$$

Имея под рукой такие данные, мы можем перейти к третьему этапу метода Шульце. Мы узнали для каждых двух кандидатов, c_i и c_j, каков уровень поддержки кандидата c_i относительно кандидата c_j и наоборот, каков уровень поддержки кандидата c_j относительно кандидата c_i. Поддержка выражена в виде прочнейших путей от c_i до c_j и от c_j до c_i, соответственно. Если прочность пути от c_i до c_j больше прочности пути от c_j до c_i, тогда мы говорим, что кандидат c_i одержал победу над кандидатом c_j. После чего для каждого кандидата c_i нам нужно узнать, скольких других кандидатов обошел c_i. Мы с легкостью узнаем это, пройдясь по массиву с прочностями путей, которые мы вычислили с помощью алгоритма 10.2, и добавив количество раз, когда избиратели предпочли c_i вместо c_j. Мы используем алгоритм 10.3, который возвращает нам список побед, где элемент i списка побед содержит список кандидатов, которых обошел кандидат i.

Алгоритм 10.3. Подсчет результатов

```
CalcResults(S, n) → wins
```

Вводные данные: S, массив размером $n \times n$ с прочнейшими путями между узлами, $s[i, j]$ является прочнейшим путем между узлами i и j; n, размер каждой величины S.

Выводимые данные: *wins*, список размером n, элемент i в *wins* представляет собой список, содержащий m целочисленных элементов j_1, j_2, …, j_m, для которых $S[i, j_k] > S[j_k, i]$.

```
1  wins ← CreateList()
2  for i ← 0 to n do
3      list ← CreateList()
4      InsertInList(wins, NULL, list)
5      for j ← 0 to n do
6          if i ≠ j then
7              if S[i, j] > S[j, i] then
8                  InsertInList(list, NULL, j)
9  return wins
```

В нашем примере мы видим, что *A* выигрывает у *C*, *B* выигрывает у *A*, *C* проигрывает всем, а *D* выигрывает у *A*, *B* и *C*. Если точнее, победы = [[2], [2, 0], [], [2, 1, 0]]. Так как количество раз, когда кандидат побеждает всех остальных кандидатов, $A = 1$, $B = 2$, $C = 0$ и $D = 3$, кандидат *D* оказывается самым предпочтительным. Алгоритму 10.3 требуется $O(n^2)$ времени, где n — это количество кандидатов или же $|C|$, согласно обозначению, которое мы использовали ранее, поэтому требуемое время равно $O(|C|^2)$. Так как алгоритм 10.1 выполняется за $O(|C|^2 + |B|^2)$ времени, алгоритм 10.2 выполняется за $\Theta(|C|^3)$ времени, а алгоритм 10.3 выполняется за $O(|C|^2)$ времени, весь метод Шульце требует полиномиального времени и отлично справляется с работой.

Обратите внимание, что мы не просто нашли победителя выборов: мы получили ранжированный список кандидатов. Как следствие, к методу Шульце можно прибегать, когда мы хотим найти лидирующие k среди n кандидатов: нам нужно лишь выделить первые k в полученном списке.

В нашем примере не случилось равного счета, и места между кандидатами распределились в строгом порядке. Так бывает не всегда. Метод Шульце укажет на победителя Кондорсе, когда таковой имеется, но, конечно же, не даст результатов, когда победителя нет. К тому же может возникнуть равный счет внизу списка, когда за победителем следуют два кандидата, борющихся за второе место. Например на выборах может случиться так, что *D* обойдет *B*, *C* и *A*, в то время как *A* обойдет *C*, *B* обойдет *D*, а *C* останется позади всех. *D* станет победителем, а *A* и *B*, оба займут второе место.

Теперь вернемся к примеру, который привел нас к методу Шульце. Вспомним, что бюллетени с кандидатами *A*, *B* и *C* у нас тогда расположились следующим образом:

10 × [A, B, C]

5 × [B, C, A]

5 × [C, A, B]

С помощью простого сравнения, еще не прибегая к методу Шульце, мы тогда обнаружили, что *A* выигрывает у *B* со счетом

15 против 5, B выигрывает у C со счетом 15 против 5, а C и A имеют равный счет, у каждого по 10. Так как в попарном сравнении победили сразу и A, и B, мы получили ничью. Как же справляется метод Шульце? Массив предпочтений для кандидатов получается следующий:

$$\begin{array}{c} \\ A \\ B \\ C \end{array}\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 0 & 15 & 10 \\ 5 & 0 & 15 \\ 10 & 5 & 0 \end{bmatrix} \end{array},$$

из чего мы получаем матрицу смежности для графа выборов:

$$\begin{array}{c} \\ A \\ B \\ C \end{array}\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} -\infty & 10 & -\infty \\ -\infty & -\infty & 10 \\ -\infty & -\infty & -\infty \end{bmatrix} \end{array}.$$

Сам граф изображен на рисунке 10.5. На нем ясно видно, что в графе два (прочнейших) пути идут из A: путь $A \to B$ и путь $A \to B \to C$, — тогда как из B идет только один прочнейший путь, $B \to C$. Поскольку обратных путей для сравнения нет, метод Шульце выдаст результат, согласно которому A выигрывает у B и C, B выигрывает у C, а C проигрывает всем. Предыдущая ничья оборачивается победой A. В целом метод Шульце выдает куда меньше «ничьих», чем простое попарное сравнение. Его надежность подтверждается тем, что он сталкивается с критерием разрешимости, который означает низкую вероятность ничейного результата.

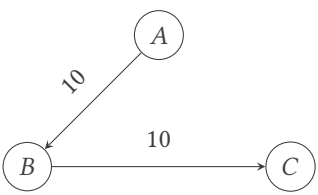


Рисунок 10.5. Еще один граф выборов

Используйте на свой страх и риск и будьте осмотрительны. Мы рассказали о критерии Кондорсе и объяснили, почему он является неотъемлемой частью выборов.

Мы также рассмотрели метод Шульце, представляющий собой осмысленную процедуру голосования, которая учитывает критерий Кондорсе. Однако это вовсе не означает, что метод Шульце — наилучший из всех способов подсчета голосов или что критерий Кондорсе — единственный верный критерий, на который следует опираться в принятии политических решений и организации проведения выборов. Как доказал лауреат Нобелевской премии Кеннет Джозеф Эрроу в своей докторской диссертации, не существует идеальной системы голосования, в которой избиратели могли бы выразить в бюллетене свои предпочтения. Его доказательство называется парадоксом Эрроу. Самое главное, чтобы выбор способа голосования был осмысленным, а не брался с потолка или фабриковался на заказ. Избиратели должны с умом решать, каким образом организовать голосование. С учетом вышесказанного, метод Шульце приводит действительно хороший пример с графовым алгоритмом, именно поэтому мы описали его в данной книге.

10.3. Алгоритм Флойда — Уоршелла

Алгоритм 10.2, используемый для расчета прочнейших путей, является разновидностью известного старого алгоритма для расчета кратчайших путей между всеми парами в графе, алгоритма Флойда — Уоршелла. Алгоритм был опубликован Робертом Флойдом, однако схожие алгоритмы были опубликованы Бернардом Роем и Стивеном Уоршеллом.

Взгляните на алгоритм 10.4. Как и алгоритм 10.2, он выполняется за $\Theta(n^3)$ времени, что, в целом, делает его куда медленнее алгоритма Дейкстры, однако он хорошо работает с плотными графами. К тому же он прост в реализации, ему не требуются особые структуры данных, плюс он работает в графах с отрицательными весами. Следует обратить внимание на строчку 16, чтобы избежать переполнения в программных языках, в которых нет обозначения для бесконечности и вместо нее нужно использовать большие числа. При таких настройках вам нужно проверить, не равны ли $dist[i, k]$ и $dist[k, j]$ тому, чем вы заменили ∞. Если равны,

нет смысла их складывать, кратчайший путь от i до j все равно не пройдет через них.

Алгоритм 10.4. Кратчайший путь Флойда — Уоршелла через все пары вершин

`FloydWarshall`$(W, n) \to (dist, pred)$

Вводные данные: W, массив размером $n \times n$, представляющий матрицу смежности графа, $W[i, j]$ является весом ребра между узлами i и j; n, размер каждой величины W.

Выводимые данные: $dist$, такой массив размером $n \times n$, что $dist[i, j]$ является кратчайшим путем между узлами i и j; $pred$, такой массив размером $n \times n$, что $[i, j]$ является предшественником узла i в прочнейшем пути к узлу j.

```
1  dist ← CreateArray(n · n)
2  pred ← CreateArray(n · n)
3  for i ← 0 to n do
4      for j ← 0 to n do
5          if W[i, j] ≠ 0 then
6              dist[i, j] ← W[i, j]
7              pred[i, j] ← i
8          else
9              dist[i, j] ← +∞
10             pred[i, j] ← −1
11 for k ← 0 to n do
12     for i ← 0 to n do
13         if i ≠ k then
14             for j ← 0 to n do
15                 if j ≠ i then
16                     if dist[i, j] > dist[i, k] + dist[k, j] then
17                         dist[i, j] ← dist[i, k] + dist[k, j]
18                         pred[i, j] ← pred[k, j]
19 return (dist, pred)
```

Примечания

Кондорсе описал приведенный критерий в своей книге «Эссе о применении анализа к оценке выборов большинством голосов» [39]. Метод Шульце описан в [175].

Теория голосования — захватывающая тема, и просвещенным гражданам следует знать способы получения результатов. Прочитайте книги Саари [169], Брамса [27], Шпиро [195] и Тейлора и Пачелли [200].

Роберт Флойд опубликовал свою версию алгоритма Флойда — Уоршелла в [67]; Стивен Уоршелл также опубликовал свою версию в 1962 году [213], в то время как Бернард Рой опубликовал свой вариант несколько раньше, в 1959 году [168]. Версия, содержащая три вложенных цикла, принадлежит Ингерману [101].

11 Методы перебора, невесты и дихотомии

Нравится ли вам искать свои ключи, носки, очки? Кружение по дому и рытье в вещах — далеко не самое приятное занятие, однако вам повезло: вы ищете физические объекты. Вы избавлены от труда искать телефон друга, песню в сборнике, платежную квитанцию для банковской операции, которая затерялась в забитой до отказа папке, — теперь всем этим занимаются компьютеры. Они помогают нам следить за вещами, отыскивают их по нашему запросу, и работа их настолько эффективна, что мы порой даже не замечаем ее. Компьютеры то и дело занимаются поиском; сложно представить какую-нибудь полезную программу, которая в том или ином виде не занималась бы поиском.

По своей сути компьютеры — прекрасные поисковики. Они куда лучше людей справляются с монотонной работой, они никогда не устают, никогда не жалуются. Они не отвлекаются, не витают в облаках и даже после проверки миллионов файлов они все так же внимательно обрабатывают следующий файл: его они ищут или нет.

Мы используем компьютеры для поиска того, что на них хранится. Поисковым алгоритмам посвящено несметное количество книг, а тема поиска до сих пор активно исследуется. Мы сохраняем на дисках все больше и больше данных. Но данные становятся бесполезными, если нельзя добраться до них за определенное время, и порой счет идет на секунды. Есть множество способов поиска: эффективность того или иного алгоритма зависит от того, что вы хотите найти и согласно каким параметрам. Принципы поиска легко понятны и даже знакомы, так как мы постоянно имеем с ними дело в нашей повседневной жизни. Еще одно прелестное свойство заключается в том, что поисковые алгоритмы создают окно, сквозь которое мы можем ясно наблюдать взаимосвязь

между умозрительным алгоритмом и его пошаговой реализацией на практике. Однако скоро вы увидите, что все на самом деле намного хитрее, чем может показаться на первый взгляд.

11.1. Последовательный поиск

Когда мы говорим о поиске, нужно сразу определиться, идет ли речь об отсортированных или смешанных в кучу данных. Поиск среди неупорядоченных данных схож с поиском карты в перетасованной колоде. Поиск же среди упорядоченных данных похож на поиск слова в словаре.

Начнем с неотсортированных данных. Как бы вы искали карту в колоде? Самый простой способ — перевернуть верхнюю карту и посмотреть, ее вы ищете или нет. Если карта не та, смотрите следующую. Можно начать и с конца, взяв сначала последнюю карту, затем предпоследнюю и так далее. Еще один способ — начать перебирать карты в случайном порядке. Если первая попавшаяся карта не та, что надо, взять наугад следующую, потом еще и еще… Или вы можете просматривать каждую последующую карту или каждые пять последующих, убирая их из колоды, пока не наткнетесь на нужную вам.

Каждый из этих способов работает, и один не лучше другого. Общее у них то, что мы перебираем карты — или наши данные — тем или иным манером, убеждаясь, что по разу просмотрели каждую карту. Так как карты расположены в неизвестном порядке, то есть мы не может вычислить никакой закономерности, нам остается лишь сесть и монотонно, одну за другой, перебрать все наши карты, или данные. Никаких хитростей и премудростей, мы решаем проблему «в лоб», то есть обстоятельно прочесываем весь материал, и главным залогом успеха здесь является наша скорость. Никакой изобретательной тонкости.

Простейший поиск методом перебора — это незамысловатый последовательный поиск, описанный выше. Идешь в самое начало и начинаешь просматривать каждый кусок информации, пока не найдешь нужный. Тут возможны два итога. Либо вы найдете искомое, либо отложите данные с уверенностью, что искомое на-

ходится где-то еще. Алгоритм 11.1 описывает поиск путем методом перебора. Он полагает, что мы ищем элемент в массиве. Мы просматриваем каждый элемент массива (строчка 1) и проверяем, является ли он нужным нам элементом (строчка 2), для этого мы используем функцию `Matches`(x, y), о которой расскажем далее чуть подробней. Если нужный элемент нашелся, алгоритм возвращает индекс этого элемента в массиве. Если же элемент не найден, алгоритм проходит весь массив до конца и не возвращает никакого индекса; в таком случае мы получим некорректное значение –1, которое сигнализирует нам, что искомый элемент в массиве не найден. На рисунке 11.1 изображен пример удачного и неудачного поиска методом перебора.

Алгоритм 11.1. Поиск методом перебора

```
SequentialSearch(A, s) → i
    Input: A, a array of items
           s, a element we are searching for
    Output: i, the position of s in A if A contains s, or −1 otherwise
1   for i ← 0 to |A| do
2       if Matches(A[i], s) then
3           return i
4   return −1
```

(a) Удачный поиск методом перебора для числа 437

(b) Неудачный поиск методом перебора для числа 583

Рисунок 11.1. Поиск методом перебора

Если в массиве A содержится n элементов, то есть $|A| = n$, то какой работоспособности ждать от поиска методом перебора? Если элементы в A расположены в совершенно случайном порядке, s с равной вероятностью может находиться в любой позиции A: s с вероятностью $1/n$ может стоять в первой позиции, с вероятностью $1/n$ может быть и во второй позиции и с такой же вероятно-

стью, $1/n$, может находиться и в последней позиции. Если s является первым элементом A, цикл в строчках 1–3 выполнится только один раз. Если s будет вторым элементом A, цикл выполнится дважды. Если же s будет стоять в самом конце A или же его не будет в массиве, цикл выполнится n раз. Так как вероятность для каждого случая равна $1/n$, среднее количество выполнений цикла будет:

$$\frac{1}{n} \times 1 + \frac{1}{n} \times 2 + \cdots + \frac{1}{n} \times n = \frac{1 + 2 + \cdots + n}{n} = \frac{n+1}{2}.$$

Последняя часть уравнения следует из утверждения:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

Следовательно, успешный поиск методом перебора занимает в среднем $O((n + 1)/2) = O(n)$ времени. Поиск перебором среди n элементов займет у вас в среднем время пропорциональное n. Если искомого элемента в массиве нет, вы пройдете через все элементы массива и обнаружите, что элемент не найден, поэтому время выполнения для неудачного поиска равно $\Theta(n)$.

11.2. Соответствие, сравнение, записи, ключи

В алгоритме мы прибегаем к функции `Matches`(x, y), которая проверяет одинаковы ли элементы, и возвращает `true`, если одинаковы, и `false`, если различны. Чуть позже мы обратимся к другой функции, `Compare`(x,y), которая сравнивает два элемента и возвращает +1, если x лучше y, –1, если y лучше x, и 0, если сравнение показало, что они равны. Подобные трехзначные функции сравнения часто встречаются в программировании, так как результат одного вызова охватывает все три возможных исхода. Как работают эти две функции и почему бы вместо `Matches` просто не проверить равны ли x и y? А потому, что, скорее всего, нам нужно совсем другое. Чтобы понять это, надо немного поразмыслить, что же мы хотим получить на самом деле.

Как правило, данные, среди которых ведется поиск, называются записями данных. Запись может содержать несколько областей, или атрибутов, схожие с человеческими отличительными

чертами, приметами. Каждый атрибут имеет присвоенное ему значение. Атрибут или группа атрибутов, которые распознают запись, называются ключами. Обычно, когда мы ищем запись, мы ищем ее с конкретным ключом. Например, если нам надо найти конкретного человека, мы найдем его по номеру его паспорта, потому что сочетание цифр в номере уникально. На рисунке 11.2(a) изображен пример записи с личными данными.

Конечно, мы можем искать человека и по другим приметам, не по ключевым, например по фамилии. В таком случае может найтись куда больше соответствий: поиск методом перебора найдет первого попавшегося человека с заданной фамилией, однако однофамильцев может оказаться еще целая куча.

В ключе может содержаться не один атрибут. В таком случае мы говорим о компоновщике, или составном ключе. На рисунке 11.2(b) вы можете видеть запись для комплексного числа. Ключом этой записи может быть комбинация обеих частей, действительной и мнимой.

Имя: Иван
Фамилия: Иванов
Номер паспорта: AI892495
Возраст: 36
Должность: преподаватель

(a) Запись личных данных

Действительная: 3,14
Мнимая: 1,62

(b) Запись комплексного числа

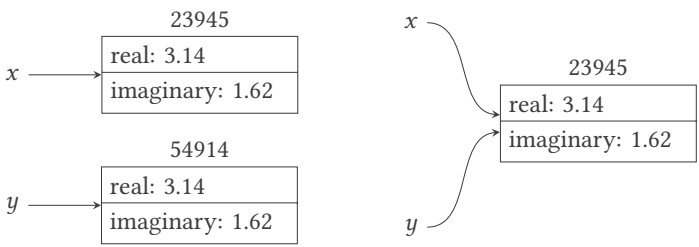
Рисунок 11.2. Примеры записи данных

У записи может быть более одного ключа. К примеру, номер паспорта и номер страхового свидетельства. Она также может содержать одновременно и простой, и составной ключи. Например, регистрационный номер может использоваться как простой ключ для данных студента, а имя, фамилия, год рождения, факультет, год поступления — как составной ключ. Мы полагаем, что комбинация всех этих атрибутов неповторима. Когда у нас более одного ключа, мы обычно назначаем один из них главным: тот, который будем использовать чаще всего. Его мы называем первичным ключом, а все остальные ключи — вторичными.

Чтобы упростить объяснение, представим, что нам нужно что-то найти: элемент или запись или еще что; мы не будем утверждать, что ищем что-то конкретное с определенным ключом. И мы будем использовать `Matches`(*x*, *y*). Памятуя о сказанном, на рисунке 11.1 мы показали только ключ записи, которую хотим найти, ведь только он нам и нужен. Мы будем придерживаться той же логики и на последующих рисунках.

Что же с $x = y$? Тут все по-другому. Все записи, все элементы, хранятся в памяти компьютера в определенном месте с определенным адресом. Сравнение $x = y$ для двух записей означает «сообщи мне, если x хранится в той же области памяти, если у него тот же адрес, что и у y». Так же и с $x \neq y$, оно означает «проверь хранятся ли x и y в разных областях памяти и разные ли у них адреса».

На рисунке 11.3 представлены две противоположные ситуации. На рисунке 11.3(a) у нас две переменные, x и y, которые указывают на две разные области памяти, где хранятся записи. Две части памяти содержат одни и те же данные, но имеют разные адреса, подписанные сверху. Записи, на которые указывают x и y, равносильны, но не одинаковы. На рисунке 11.3(b) у нас две переменные, x и y, которые указывают на одну и ту же область памяти. Две переменные идентичны друг другу.



(a) Одинаковое содержимое, но $x \neq y$ (b) Одна и та же запись, $x = y$

Рисунок 11.3. Два вида равенства

Проверить $x = y$ или $x \neq y$ для двух записей быстрее, чем заниматься сравнением их ключей, потому что так мы сравниваем только их адреса, которые являются всего-навсего целыми числами. Сравнение $x = y$ называется строгим сравнением. Так что мы

выигрываем в скорости, если строгое сравнение — то, что нужно в нашей ситуации. Следовательно, в алгоритме 11.1 мы могли бы использовать $A[i] = s$ в строчке 2, если бы мы не искали запись, основанную на содержании, или ключах, однако вместо этого мы пытаемся узнать, идентичен ли элемент s конкретной записи в $A[i]$. В большинстве случаев, когда мы что-то ищем, мы не прибегаем к строгому сравнению.

11.3. Эффект Матфея и степенные законы

Если вам повезет и искомые элементы окажутся в где-то в начале массива $|A|$, тогда поиск методом перебора — то, что вам надо. К нему выгодно прибегать в случаях, когда перед вами ряд элементов, упорядоченных не по единственному атрибуту (вроде имени или фамилии), а по частоте повторения их внешних признаков. Если самый заурядный объект находится в самом начале массива, а за ним идет второй по заурядности объект, и чем дальше, тем все меньше в объектах ординарных признаков, поиск методом перебора проявит себя во всей красе.

Данное предложение делается интересней, если принять во внимание факт, что львиная доля всех вещей распределяется не просто неравномерно, а колоссально неравномерно. Одна из таких вещей — благосостояние; лишь малый процент людей обладает большим процентом мирового богатства. Еще один яркий пример — частота повторения слов в языке: одни слова употребляются гораздо чаще других. Ту же закономерность можно заметить в размерах городов: большинство людских населенных пунктов не насчитывают миллиона жителей, но в то же время существует несколько городов-мультимиллионеров. В виртуальной реальности большинство сайтов ограничиваются несколькими посетителями, и лишь несколько сайтов пользуются бешеной популярностью и имеют огромную аудиторию. В литературной области большинство книг не находят своих читателей, и лишь несколько выбиваются в бестселлеры. Все это схоже с явлением, когда «богатые богатеют, а бедные беднеют», его еще называют эффектом Матфея, в честь стиха 25:29 из Евангелия от Матфея,

где говорится: «…ибо всякому имеющему дастся и приумножится, а у неимеющего отнимется и то, что имеет…»

В лингвистике данное явление именуют законом Ципфа, в честь гарвардского лингвиста Джорджа Кингсли Ципфа, который заметил, что *i*-тое самое употребительное слово в языке появляется с частотой пропорциональной 1/*i*. Закон Ципфа гласит, что вероятность встретить *i*-тое самое употребительное слово в корпусе текстов из *n* слов такова:

$$P(i) = \frac{1}{i}\frac{1}{H_n},$$

где:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Число H_n встречается в математике достаточно часто, чтобы получить особое имя: оно называется *n*-ное гармоническое число. Почему такое название? Из-за обертонов, или гармонических призвуков в музыке. Струны звучат в длине волны основного тона, а также в обертонах, которые составляют 1/2, 1/3, 1/4… длины волны основного тона: это соответствует бесконечной сумме, которую мы получаем при $n = \infty$, что называется гармоническим рядом.

Помимо того, что закон Ципфа выдает вероятности для случая, он так же касается одноименного распределения вероятностей. В таблице 11.1 представлены двадцать самых употребляемых в корпусе английского языка слов (использовался брауновский языковой корпус текстов, содержащий 981 716 слов, 40 234 из которых уникальны), их эмпирические вероятности вычислены путем подсчета их наличия в корпусе, а их теоретические вероятности — по закону Ципфа, или распределения. То есть мы указываем позицию, слово, эмпирические и теоретические распределения.

На рисунке 11.4 отображен график для таблицы 11.1. Обратите внимание, что распределение указано только для целых значений; мы добавили линию, чтобы показать общую тенденцию. Также заметьте, что эмпирические и теоретические вероятности

совпадают не полностью. Так случается всегда, когда математическая идея сталкивается с воплощением в реальном мире.

Таблица 11.1. Двадцать самых распространенных слов в брауновском корпусе и их вероятности: эмпирические и заданные законом Ципфа

Rank	Word	Empirical Probability	Zipf Law
1	THE	0.0712741770532	0.0894478722533
2	OF	0.03709015642	0.0447239361267
3	AND	0.0293903735907	0.0298159574178
4	TO	0.0266451804799	0.0223619680633
5	A	0.0236269959948	0.0178895744507
6	IN	0.0217343916163	0.0149079787089
7	THAT	0.0107913082806	0.0127782674648
8	IS	0.0102972753831	0.0111809840317
9	WAS	0.00999779977101	0.00993865247259
10	HE	0.00972582702126	0.00894478722533
11	FOR	0.00966572817393	0.0081316247503
12	IT	0.00892315089089	0.00745398935445
13	WITH	0.0074247542059	0.00688060555795
14	AS	0.00738808372279	0.00638913373238
15	HIS	0.00712629721834	0.00596319148356
16	ON	0.00686654796295	0.00559049201583
17	BE	0.0064957686337	0.00526163954431
18	AT	0.00547205098012	0.0049693262363
19	BY	0.00540482176108	0.00470778275018
20	I	0.00526017707769	0.00447239361267

Когда мы видим стремительное падение тенденции, как на рисунке 11.4, следует посмотреть, что же происходит, если вместо привычных осей x и y использовать логарифмические оси. С ними мы превращаем все значения в их логарифмы и строим соответствующий график. Логарифмический вариант графика с рисунка 11.4 показан на рисунке 11.5: для каждого y мы используем $\log y$, а для каждого x — $\log x$. Как можно заметить, тенденция для теоретического распределения вытянулась прямой линией; эмпирическое же распределение слегка поднимается вверх над тем, что предписала теория. В большинстве случаев наблюдается разница между теоретическим распределением и тем, что получается на практике. К тому же два наших рисунка показывают лишь выборку из первых двадцати самых употребляемых слов, так что, опира-

ясь только на них, мы не можем прийти к точному результату. Чтобы увидеть, что происходит, посмотрите на рисунки 11.6 и 11.7, где представлены подробные распределения для всех 40 234 уникальных слов брауновского корпуса текстов. Сразу бросаются в глаза два момента. Во-первых, без логарифмических координат построение графика бесполезно. Перед нами убедительное доказательство того, как неравномерно распределение вероятностей. Чтобы увидеть хоть какую-то тенденцию, нам нужны логарифмические значения. Во-вторых, когда мы переходим в логарифмическую систему координат, соответствие между теоретическими значениями и эмпирическими наблюдениями становится более точным.

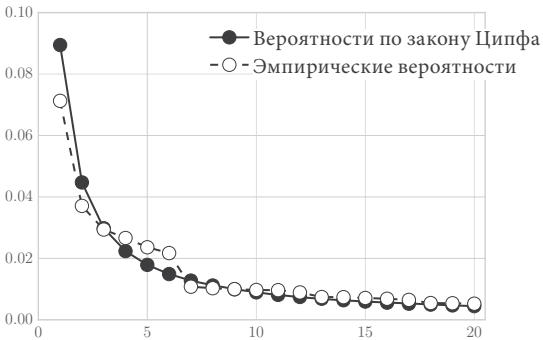


Рисунок 11.4. Распределение по Ципфу для двадцати самых употребляемых слов брауновского корпуса

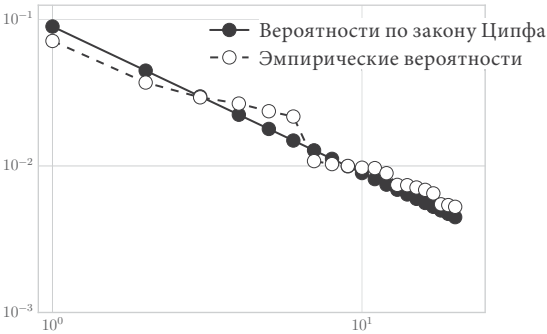


Рисунок 11.5. Распределение по Ципфу для двадцати самых употребляемых слов брауновского корпуса в логарифмической системе координат

Нам яснее видна разница в логарифмической системе координат, так как закон Ципфа является частным случаем степенного закона. Степенной закон вступает в силу, когда вероятность значения пропорциональна отрицательной степени значения, или, говоря математическим языком:

$$P(X = x) \propto cx^{-k} \qquad \text{где } c > 0,\ k > 0.$$

В данной формуле символ ∝ означает «пропорционально». Теперь мы можем объяснить, почему логарифмический график представляет собой прямую линию. Если у нас есть $y = cx^{-k}$, мы получаем $\log y = \log(cx^{-k}) = \log c - k \log x$. Последняя часть — это линия с отсекаемым отрезком y равным $\log c$ и наклоном равным $-k$. Поэтому, когда мы встречаем данные, чей график в логарифмических координатах выглядит как прямая линия, перед нами явный признак того, что их теоретические распределения могут соответствовать степенному закону.

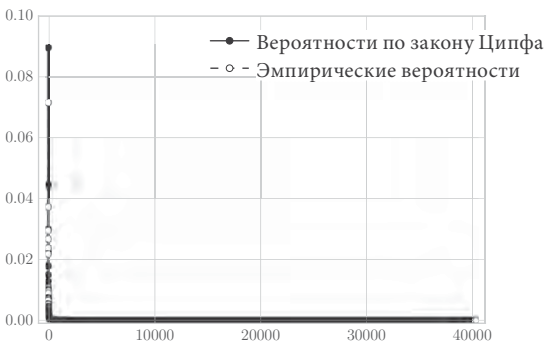


Рисунок 11.6. Эмпирические и теоретические распределения брауновского корпуса

В экономике примером степенного закона может служить принцип Парето, согласно которому около 20% действий отвечает за примерно 80% результатов. Как говорят в народе и менеджменте: 20% работников выполняют 80% работы. В случае принципа Парето можно доказать, что $P(X = x) = c/x^{1-\theta}$, где $\theta = \log 0{,}80/\log 0{,}20$.

Степенные зависимости встречаются то тут, то там, за последние двадцать лет возникла целая область изучения сопряженного

явления; кажется, куда ни глянь — везде найдешь степенную закономерность. Помимо примера, который мы привели, когда рассказывали о эффекте Матфея, степенные законы можно встретить в совершенно различных областях: ссылках на научные работы, интенсивности землетрясений и диаметре лунных кратеров. Мы также можем заметить их в увеличении количества биологических видов на протяжении времени, во фракталах, в способах добывания пищи у хищников и в силе гамма-лучевых вспышек на Солнце. Список можно продолжать бесконечно: ежедневное количество междугородних телефонных звонков, число людей, потерявших сознание, имена, встречающиеся в одной семье, и так далее.

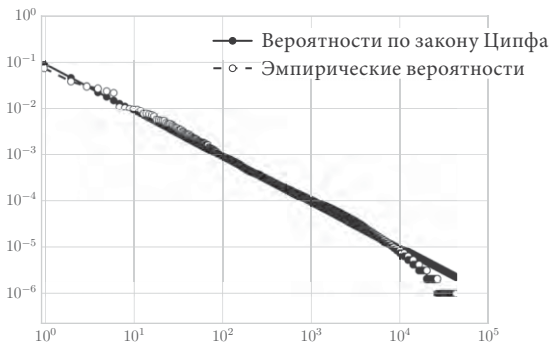


Рисунок 11.7. Эмпирические и теоретические распределения брауновского корпуса в логарифмических координатах

Такие закономерности, кажется, возникают на ровном месте. Например, схожий закон — закон Бенфорда, названный в честь физика Фрэнка Бенфорда, или также известный как закон первой цифры. Он касается частоты распределения цифр в разного рода данных, а точнее, в нем говорится, что в 30% случаев первой цифрой в числе идет единица. Каждая цифра от 2 до 9, стоящая в начальной позиции, встречается все реже. Говоря математическим языком, данный закон утверждает, что вероятность того, что первой цифрой числа будет $d = 1, 2, \ldots, 9$, такова:

$$P(d) = \log\left(1 + \frac{1}{d}\right)$$

Если мы просчитаем вероятности для каждой цифры, мы получим результаты таблицы 11.2. Числа в таблице говорят нам, что, если у нас имеется набор чисел в ряде данных, первой цифрой в 30% случаев будет 1: почти 30% всех чисел будут начинаться с 1, около 17% чисел будут начинаться с 2, около 12% — с 3 и так далее.

На рисунке 11.8 вы можете увидеть график для закона Бенфорда. Он не сильно отличается от распределения по Ципфу, поэтому нам может стать любопытно, что случится, если мы начертим его в логарифмической системе координат. Результат вы можете видеть на рисунке 11.9. Почти что прямая линия, учитывая, что закон Бенфорда связан со степенными законами.

Таблица 11.2. Закон Бенфорда, касающийся вероятности встретить ту или иную цифру в первой позиции

Leading Digit	Probability
1	0.301029995664
2	0.176091259056
3	0.124938736608
4	0.0969100130081
5	0.0791812460476
6	0.0669467896306
7	0.0579919469777
8	0.0511525224474
9	0.0457574905607

Охват закона Бенфорда просто потрясает. Он применим к таким разнообразным рядам данных, как физические константы, высота самых высоких зданий в мире, число населения, биржевой курс, адреса домов и прочее-прочее. В самом деле, он кажется настолько всеобъемлющим, что для проверки данных на подлинность нужно лишь проверить подчиняются ли содержащиеся в них числа закону Бенфорда. Мошенники меняют подлинные значения или подставляют на их место случайные, не обращая внимания на то, подчиняются ли новые значения закону Бенфорда. Поэтому, как только нам попадается подозрительный ряд данных, первым делом лучше всего проверить, соответствует ли он вероятностям Бенфорда.

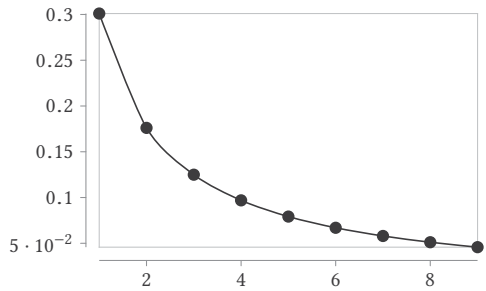


Рисунок 11.8. Закон Бенфорда

Закон Бенфорда может повлиять на наш поиск, если структура нашего поиска отражает структуру распределения, то есть, если ключи записей и искомые нами ключи следуют закону Бенфорда. При таком раскладе у нас будет больше записей с ключами, начинающимися с 1, и больше поисков для данных записей, уже меньше будет для ключей, начинающихся с 2, и так далее.

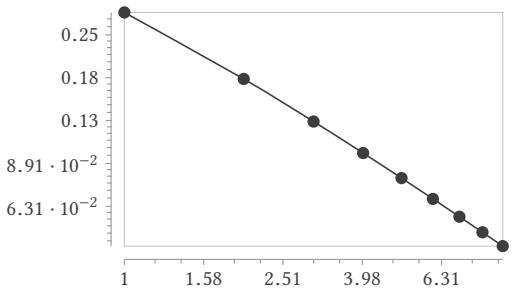


Рисунок 11.9. Закон Бенфорда в логарифмической системе координат

11.4. Самоорганизующийся поиск

Вернемся к поиску. Если мы заметили, что наши поисковые записи ищутся по схемам, согласно которым одни запросы более популярны, чем другие, то не следует ли нам обратить данный факт себе на пользу? То есть если мы знаем, что большинство наших запросов будут касаться одного объекта, меньшее их количество — другого объекта и совсем немного запросов — еще

какого-то объекта, то мы видим, что популярность запросов следует степенному закону, и значит, можно что-то из этого извлечь. В частности, если мы сможем преобразовать данные в такой вид, чтобы самые часто повторяющиеся элементы шли впереди, тогда мы существенно упростим работу. Описанную нами задумку называют самоорганизующимся поиском, потому что мы используем поисковый шаблон, чтобы организовать наши данные и добиться оптимального порядка.

Мы реализуем эту идею в алгоритме 11.2, где нам нужно просмотреть списки элементов с помощью метода движения к началу. В цикле строчек 1–9 мы проходим по элементам списка. В ходе цикла мы отслеживаем предыдущий элемент списка, а не тот, в котором находимся на данный момент. Предыдущий элемент мы обозначаем переменной *p*. Изначально в строчке 1 мы присваиваем *p* значение `null` и обновляем его каждый раз, когда проходим цикл, что описано в строчке 9. Если мы находим соответствие, мы проверяем в строчке 4, находится ли соответствие во главе списка. Так случится, если $p \neq$ `null`, потому что $p =$ `null` только при первом прохождении цикла. Обратите внимание, что сравнение с `null` является строгим сравнением, потому что мы всего-навсего проверяем, не указывает ли переменная на пустое место. Если мы не во главе списка, мы помещаем нынешний элемент в начало списка, а затем возвращаем его. Если мы находим соответствие, но мы во главе списка, мы лишь возвращаем соответствие. Если мы не находим соответствия, мы не можем ничего вернуть, поэтому возвращаем `null`.

Как это работает изображено рисунке 11.10. Мы ищем элемент 4. Находим его, берем и переносим в начало списка. Чтобы перенести элемент во главу списка, нам нужна функция `RemoveListNode`, за которой следует функция `InsertListNode`. Это значит, что изъятие элемента из списка и его добавление в начало — нечто, что можно сделать без перемещения остальных элементов; мы оперируем только связями между элементами. С массивами все было бы по-другому. Если убрать элемент из массива, останется дыра или же все последующие элементы сдвинутся на одну позицию вперед. Добавление элемента в начало массива привело бы к смещению

всех элементов на одну позицию. Это проблематично и неудобно, именно поэтому в алгоритме 11.2 используются списки.

Алгоритм 11.2. Самоорганизующийся поиск с движением к началу

`MoveToFrontSearch(`L, s`)` $\longrightarrow$ s or NULL

Вводные данные: L, список элементов; s, искомый элемент.
Выводимые данные: нужный нам элемент, или же `null`, если его нет.

```
1  p ← NULL
2  foreach r in L do
3      if Matches(r, s) then
4          if p ≠ NULL then
5              m ← RemoveListNode(L, p, r)
6              InsertListNode(L, NULL, m)
7              return m
8          return r
9      p ← r
10 return NULL
```

Еще один схожий метод для самоорганизующегося поиска не перемещает соответствующий элемент в начало, а меняет его местами с предыдущим элементом. Например, если мы ищем элемент и находим его в пятой позиции наших данных, мы меняем его с предшествующим, четвертым, элементом. Таким образом самые популярные элементы будут перемещаться в начало, хотя не за один присест, а по одной позиции за каждый раз, когда их ищут. Алгоритм 11.3 показывает, как это работает; данный подход называется методом перестановки.

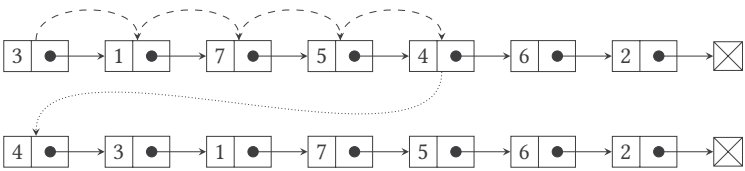


Рисунок 11.10. Самоорганизующийся поиск: метод движения к началу

Тут нам нужно отслеживать два списка элементов: предыдущий элемент p и тот, что стоит перед p, назовем его q. Изначально,

в строчках 1–2, оба элемента ставятся в положение `null`. Когда мы просматриваем элементы в списке, мы в строчке 10 заставляем q указывать на p, прежде чем обновить p, чтобы он указывал на элемент, в котором мы только что побывали (строчка 11).

Алгоритм 11.3. Самоорганизующийся поиск с перестановкой

```
TranspositionSearch(L, s) → s or null
    Вводные данные: L, список элементов; s, искомый элемент.
    Выводимые данные: нужный нам элемент, или же null, если его
    не существует.
1   p ← NULL
2   q ← NULL
3   foreach r in L do
4       if Matches(r, s) then
5           if p ≠ NULL then
6               m ← RemoveListNode(L, p, r)
7               InsertListNode(L, q, m)
8               return m
9           return r
10      q ← p
11      p ← r
12  return NULL
```

Прибегая к методу перестановки, мы используем слегка измененную версию `Insert`. Нам нужно добавить элемент в произвольную область списка, поэтому у нас `Insert`(L, p, i), которая означает, что мы добавляем элемент i сразу после элемента p. Если p = `null`, — что случится, если r является вторым элементом списка, а p является первым элементом списка, — `Insert` вставит m в начало списка. На рисунке 11.11 мы иллюстрируем, что произойдет с тем же поиском, какой мы отображали на рисунке 11.10, если мы будем переставлять элементы, а не двигать их в начало.

Перестановка местами двух элементов эффективна как в списках, так и массивах, поэтому метод перестановки подходит тем и другим. В алгоритме 11.4 мы показываем, как самоорганизующийся поиск с перестановкой работает в массиве. Нам лишь понадобится функция `Swap`(x, y), которая меняет x на y и y на x. В массиве `Swap` меняет один элемент на другой.

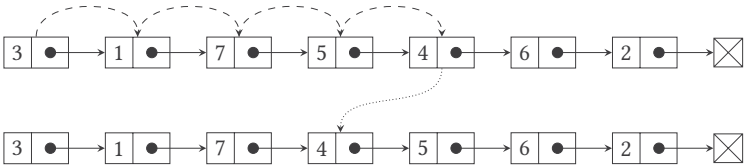


Рисунок 11.11. Самоорганизующийся поиск: метод перестановки

Логика алгоритма 11.4 та же, что в алгоритме 11.3, вот только вместо самого элемента мы возвращаем его индекс. Так как мы имеем дело с индексом, в строчке 3 мы без проблем можем узнать, где находимся: просто-напросто сравниваем индекс с нулем, чтобы узнать, находимся ли мы в первом элементе.

Алгоритм 11.4. Самоорганизующийся поиск с методом перестановки в массиве

```
TranspositionArraySearch(A, s) → i
```

Вводные данные: A, массив элементов; s, искомый элемент.

Выводимые данные: i, позиция s в A, если в A содержится s, или иначе -1.

```
1  for i ← 0 to |A| do
2      if Matches(A[i], s) then
3          if i > 0 then
4              Swap(A[i − 1], A[i])
5              return i − 1
6          else
7              return i
8  return −1
```

Самоорганизующийся поиск может значительно сэкономить время. Если вероятности наших ключей следуют закону Ципфа, можно доказать, что среднее число сравнений, требуемых при методе движения к началу, равно $O(n/\lg n)$, что значительно лучше, чем $O(n)$, которое мы получаем при обычном методе перебора. Методу перестановки требуется еще меньше сравнений, чем методу движения к началу, но выполняется он слишком долго. Так что, если речь не идет о долговременном поиске, движение к началу оказывается наиболее предпочтительным методом.

11.5. Задача о разборчивой невесте

Давайте обратимся к иной формулировке нашей поисковой задачи: представим, что в ряде объектов нам надо найти лучший. Конечно же, «лучший» диктуется контекстом. Мы можем искать по параметру дешевизны, высшего качества или чего угодно. Нас лишь интересует, что мы можем сопоставить один элемент с другим и определить, лучше он или нет. Все просто. Если нам нужно найти, скажем, самый дешевый объект, мы всего лишь просматриваем весь ряд объектов — на их стоимость ничто не виляет — и выбираем тот, у которого цена ниже всех.

А теперь добавим заковырку. Представьте, что каждый раз, когда мы просматриваем объект, нам нужно сразу же решить, берем мы его или нет. Если мы отказываемся от него, потом мы уже не сможем передумать. Если же мы берем его, то тем самым отвергаем все последующие объекты.

Такое часто случается, когда мы получаем данные из внешнего источника; мы не отвечаем за их порядок, нам нужно выбрать наилучший из них, и мы не можем принять отвергнутый ранее выбор. Такую ситуацию называют задачей о разборчивой невесте. Представим, что к невесте по одному приходят женихи. Хитрость в том, что после разговора с женихом невеста не может взять время на раздумье и должна дать ответ сразу. Как же ей выбрать лучшего жениха?

Получается, что если к ней пришли n женихов, разумней всего отвергнуть предложения первых n/e из них, где $e \approx 2{,}7182$, то есть эйлерово число. Соотношение n/e равно примерно 37% от всех женихов. После того как невеста отвергла 37% процентов женихов, ей стоит отметить среди них лучшего, а затем из последующих кандидатов выбрать первого, кто окажется лучше отмеченного среди первых n/e. Если ей не попадется никто лучше, то нужно принять предложение последнего из оставшихся. Вероятность выйти замуж за лучшего кандидата равна $1/e$, почти 37%, и это лучший выбор, который может сделать невеста.

Алгоритм 11.5 описывает данный механизм, когда решения должны приниматься сразу же и без возможности вернуться

к предыдущим объектам. Мы либо принимаем, либо отвергаем, и в каждом случае решение необратимо.

Алгоритм 11.5. Невестин поиск

SecretarySearch(A) → i

Вводные данные: A, массив элементов.

Выводимые данные: i, позиция элемента в A, которую мы должны выбрать, или же –1, если нам не удалось найти лучший элемент.

```
1  m ← ⌈|A|/e⌉
2  c ← 0
3  for i ← 1 to m do
4      if Compare(A[i], A[c]) > 0 then
5          c ← i
6  for i ← m to |A| do
7      if Compare(A[i], A[c]) > 0 then
8          return i
9  return −1
```

В строчке 1 мы вычисляем количество элементов, которые мы отвергнем во время поиска лучшего из них. Так как $|A|/e$ — не целое число, мы берем ближайшее целое число m, которое больше частного $|A|/e$. Мы используем переменную c как индекс лучшего элемента среди первых m элементов. Изначально мы присваиваем ему нулевое значение, первого элемента. В строчках 3–5 мы выискиваем лучший элемент из первых m элементов. Мы применяем функцию `Compare`(x, y), которая сравнивает два элемента и возвращает +1, если x лучше y, –1, если y лучше x, и 0, если они одинаковы. Сравнение выполняется с помощью ключей баз данных; когда мы говорим, что сравниваем элементы или записи данных, мы на самом деле имеем в виду, что мы сравниваем их ключи. Найдя лучший среди первых m элементов, $A[c]$, мы начинаем искать первый из последующих элементов, который окажется лучше $A[c]$. Если мы находим такой элемент, мы тут же его возвращаем; в противном случае мы возвращаем –1, что обозначает неудачу, так как лучший элемент находился среди первых m элементов.

Не забывайте, что алгоритм 11.5 не обязательно выдаст вам лучший элемент в массиве A. Если лучший находится среди пер-

вых m элементов, то он, увы, упущен, и вы останетесь ни с чем. Но лучшего способа нет. Так что если вы отсылаете резюме работодателю, не лишним будет удостовериться, что его не просмотрят одним из первых, или же вовсе не стоит отправлять его в компании с подобным методом отбора.

Алгоритм 11.5 разрешает любую задачку, где тут же требуется решить взять объект или оставить. Схожая по описанию задачка появляется, когда мы ожидаем ряд вводных данных или событий и нам надо решить, с каким из них мы будем иметь дело. Нам известно, сколько событий нас ждет, но у нас нет никакого указателя, который заранее подсказал бы нам, какое из них лучше всего выбрать. Представим, что мы получаем приглашения. Как только мы прочитываем каждое, нам надо либо принять его, либо отвергнуть. Принять можно только одно приглашение. Когда мы отказываемся от приглашения, оно исчезает. Все подобные ситуации относятся к теории оптимальной остановки, которая связана с выбором верного времени для свершения действия, чтобы получить максимальное вознаграждение или минимизировать затраты. В некотором роде мы ищем не просто элемент, а способ выяснить, когда лучше всего прекратить поиск. Прочие ситуации, похожие на задачу о разборчивой невесте, называют онлайновыми алгоритмами, где вводные данные представлены и обрабатываются рядом функций, по частям: вводные данные как единое целое недоступны нам с самого начала. Поточные алгоритмы, где вводные данные представляют собой последовательность элементов, которые можно исследовать поштучно по ходу поступления, являются примерами онлайновых алгоритмов.

Если говорить о сложности алгоритма 11.5, в строчках 1–5 мы просматриваем первые m элементов. Далее идут строчки 6–8 схожие по структуре со строчками 1–3 алгоритма 11.1, однако вместо n элементов у нас $n - m$ элементов. Следовательно, общая сложность $O(m + (n - m + 1)/2) = O(m/2 + (n + 1)/2) = O(n/2e + (n - 1)/2) = O(n)$.

В задаче о разборчивой невесте предполагается, что невеста хочет выбрать наилучшего из всех женихов и ее интересует именно он. Почти с 37% вероятностью она выберет того самого, однако может случиться, что она окажется ни с чем. Идеальный

жених для нее — весь мир, об остальных она не хочет даже думать. Теперь же представим, что невеста снизила планку. Вместо того, чтобы желать одного-единственного неповторимого — или, обобщая тему, искать единственный наилучший элемент среди прочих — и воротить нос от всех остальных, она решила оценивать каждого согласно его достоинствам.

1 1.0079	2 4.0025	3 6.941	4 9.0122	5 10.811	6 12.011	7 14.007	8 15.999	9 18.998	10 20.180
H	He	Li	Be	B	C	N	O	F	Ne
Hydrogen	Helium	Lithium	Beryllium	Boron	Carbon	Nitrogen	Oxygen	Fluorine	Neon

(a) Первые десять химических элементов, распределенные по порядковым номерам

89 227	13 26.982	95 243	51 121.76	18 39.948	33 74.922	85 210	56 137.33	97 247	4 9.0122
Ac	Al	Am	Sb	Ar	As	At	Ba	Bk	Be
Actinium	Aluminium	Americium	Antimony	Argon	Arsenic	Astatine	Barium	Berkelium	Beryllium

(b) Первые десять химических элементов, распределенные в алфавитном порядке

89 227	47 107.87	13 26.982	95 243	18 39.948	33 74.922	85 210	79 196.97	5 10.811	56 137.33
Ac	Ag	Al	Am	Ar	As	At	Au	B	Ba
Actinium	Silver	Aluminium	Americium	Argon	Arsenic	Astatine	Gold	Boron	Barium

(c) Первые десять химических элементов, распределенные согласно химическим знакам

Рисунок 11.12. Химические элементы в разных порядках. Каждый элемент изображен с соответствующим ему химическим знаком, порядковым номером и атомной массой

Таким образом, она решила, что ей подойдут те элементы, достоинства которых соответствуют заявленному для сравнения параметру. Например, если вы собираетесь купить машину и главное для вас, чтобы она была быстрой, тогда обязательным параметром станет скорость: вас будут интересовать только быстрейшие машины, а все остальные отметутся в сторону как автохлам. Компромисс означает, что более всего вы цените быстрые машины, однако не пренебрегаете и остальными. Вторая по скорости машина будет менее желанна, чем первая, но она лучше, чем ничего, то же самое и с третьей по скорости машиной и так далее. Вы не приглядываетесь к одним только быстрым машинам, заклеймив все остальные как никчемные. Если бы вы так поступили, тогда получилось бы, что вам нужно рассматривать и отвергать не первые n/e автомобилей, а первые $\sqrt{n}$. Более того,

с увеличением числа n ваша вероятность обзавестись наилучшей машиной становится 1, вместо $1/e$, что кажется довольно странным. Вы всего лишь немного изменили формулировку задачи — и пришли к совершенно иному решению и иным шансам добраться до оптимального решения. Такое часто случается в программировании и математике, где, на первый взгляд, крохотное изменение переворачивает с ног на голову всю задачу.

11.6. Бинарный поиск

До сего момента мы полагали, что не можем контролировать порядок поступающих к нам элементов. Теперь же давайте посмотрим, какую пользу мы можем извлечь, если заранее знаем порядок.

Важно, чтобы в основе порядка элементов находился ключ, с помощью которого вы будете их искать. На рисунке 11.12 изображены десять первых химических элементов, расположенные в трех разных порядках: по порядковым номерам, в алфавитном порядке согласно их названиям и в алфавитном порядке согласно их химическим знакам.

Как вы видите, все порядки различны. Даже алфавитные распределения по названиям и химическим знакам отличаются друг от друга: в основе химических знаков лежат не английские, а латинские называния. Если вы собираетесь найти химический элемент по его порядковому номеру, вам нужно отсортировать элементы согласно этому ключу, в то время как если вы хотите найти элемент по химическому знаку, то вам понадобится порядок, изображенный на рисунке 11.12(c). Искать элементы, отсортированные в одном порядке, среди элементов, распределенных в ином порядке, совершенно бесполезно, же успехом можно применить простой метод перебора.

То же самое можно сказать и про обратную ситуацию: когда нам известен верный порядок, совершенно неуместно применять метод перебора. Если перед вами на столе стопка резюме, отсортированная по фамилиям кандидатов, то, когда вы просматриваете ее в поисках фамилии, начинающейся на «Д», вы ожидаете, что нужная фамилия найдется где-то в начале стопки. И наобо-

рот, если вы ищите фамилию на букву «Ф», вы подозреваете, что наткнетесь на нее ближе к концу стопки.

Итак, у нас есть хорошее обоснованное предположение, и оно может нам помочь, если мы столкнемся с простой игрой-угадайкой, когда один игрок говорит: «Я задумал число от одного до ста. Угадай, какое число я задумал. Если называешь неверное число, я говорю, больше оно или меньше того, что я задумал». Если только вы не телепат, вам выгодней всего начать высказывать предположения с числа 50. Если вы угадали, то настает конец игры. Если 50 больше задуманного числа, то вы знаете, что вам надо искать среди чисел от 0 до 50. Вы снова делите промежуток пополам и высказываете предположение — 25. Если вы будете продолжать в том же духе, вам понадобится, самое большое, семь предположений, чтобы угадать задуманное число; мы увидим, почему именно столько, когда вернемся к нашей беседе о поиске среди упорядоченных элементов.

У описанного нами метода есть название — бинарный поиск. Слово «бинарный» взято потому, что при принятии каждого решения мы разбиваем область поиска напополам. Сначала мы знали, что число находится в промежутке от 0 до 100. Затем мы узнали, что оно находится либо между 0 и 50, либо между 50 и 100; а потом — между 0 и 25 или между 25 и 50. Алгоритм 11.6 приводит бинарный поиск в алгоритмический вид.

В алгоритме используются две переменные, l и h, чтобы отобразить, соответственно, нижнюю и верхнюю границы области поиска. Изначально l является первым элементом массива, в котором производится поиск, а h является последним элементом того же массива. В строчке 4 мы вычисляем середину, m, массива путем нахождения срединного числа между l и h и его округления до ближайшего целого числа. Если срединное число меньше искомого элемента, мы понимаем, что нужно искать к началу, в промежутке между срединным числом и l в строчке 7. Если срединное число больше искомого элемента, мы понимаем, что нужно искать к концу, в промежутке между срединным числом и h в строчке 8. Если срединное число и есть искомый элемент, мы просто возвращаем его.

Процедура повторяется до тех пор, пока не находится искомый элемент. Каждый раз мы разбиваем область нашего поиска напополам. Или, еще лучше, почти напополам: если нам надо выполнить поиск среди семи элементов, то получится, что половина — не целое число, поэтому мы берем в качестве срединного числа 3 и разделяем нашу область поиска межу первыми двумя и последними четырьмя элементами, если третий элемент не тот, что мы ищем. Это никак не влияет на положение дел. В какой-то момент нам будет негде искать. Если такое случается, мы возвращаем –1 и тем показываем, что нам не удалось найти нужный элемент.

Алгоритм 11.6. Бинарный поиск

`BinarySearch`$(A, s) \to i$

Вводные данные: A, отсортированный массив с элементами; s, искомый элемент.

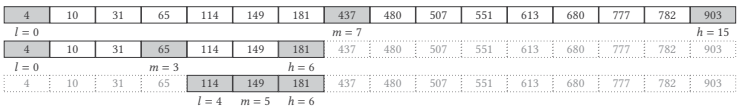
Выводимые данные: i, позиция s в A, если в A содержится s, или иначе –1.

```
1  l ← 0
2  h ← |A|
3  while l ≤ h do
4      m ← ⌊(l + h)/2⌋
5      c ←Compare(A[m], s)
6      if c < 0 then
7          l ← m + 1
8      else if c > 0 then
9          h ← m − 1
10     else
11         return m
12 return −1
```

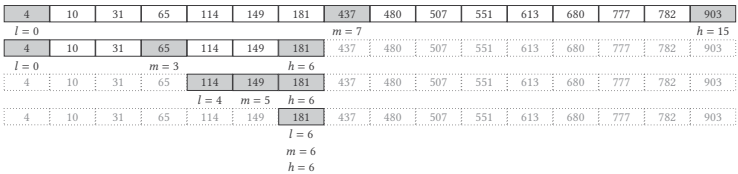
Можно обрисовать это по-другому. В каждом цикле строчек 3–11 либо увеличивается l, либо уменьшается h. Таким образом, в определенный момент условие $l \le h$ в строчке 3 перестанет соблюдаться. Тогда нам пора остановиться.

Вы можете видеть ход работы бинарного поиска на рисунках 11.13 и 11.4. Наш массив содержит числа 4, 10, 31, 65, 114, 149, 181, 437, 480, 507, 551, 613, 680, 777, 782, 903. На рисунке 11.13(а) видим

успешный поиск для числа 149. На нем отмечены переменные *l*, *h* и *m* алгоритма 11.6. Мы отмечаем блекло-серыми числами отброшенную область поиска, поэтому можно ясно увидеть, как она сокращается. При поиске числа 149 мы находим его в третьей итерации, когда срединное число указывает прямиком на него. На рисунке 11.13(b) мы, к счастью, ищем число 181. В этот раз мы находим его, когда область поиска сокращается до размеров одного элемента — того, который нам нужен.



(a) Удачный поиск для числа 149



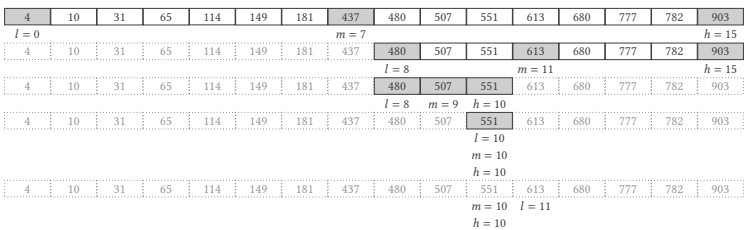
(b) Удачный поиск для числа 181

Рисунок 11.13. Бинарный поиск, удачные примеры поиска

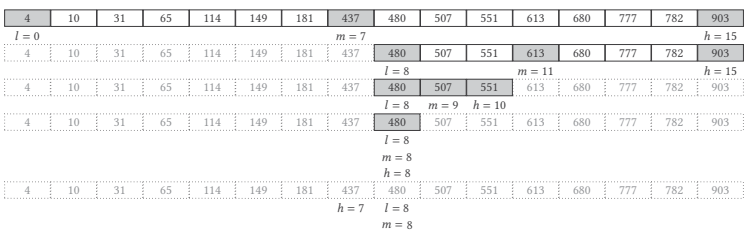
При неудачном поиске элемента, как на рисунках 11.14(a) и 11.14(b), мы сокращаем область поиска до тех пор, пока не остается места, и не находим того, что искали. На рисунке 11.14(a) левая граница нашей области поиска перемещается прямо в правую границу, что не имеет смысла. Это нарушает условие, заданное в строчке 3 алгоритма 11.6, поэтому алгоритм возвращает –1. То же самое с рисунком 11.14(b), правая граница нашей области поиска перемещается прямо в левую границу, что приводит нас к аналогичному результату.

Бинарный поиск был разработан еще на заре компьютерной эпохи. Тем не менее детальная реализация для его корректной работы доставляла программистам массу хлопот. Выдающийся исследователь и программист Джон Бентли обнаружил в 1980 году, что примерно 90% профессиональных программистов после семи часов труда не могли реализовать бинарный по-

иск так, чтобы он работал без ошибок. В 1988 году еще один исследователь обнаружил, что только 5 из 20 учебников содержат точное описание нужного алгоритма (хочется верить, что ситуация с тех пор изменилась в лучшую сторону). По прихоти судьбы оказалось так, что программа Джона Бентли была нерабочей: в нее закралась ошибка, которая оставалась незамеченной почти что двадцать лет. Ее нашел Джошуа Блох, известный инженер по программному обеспечению, который реализовал множество основных свойств программного языка Java. Что интересно, ошибка, найденная Блохом, перекочевала в его собственную реализацию бинарного поиска на Java и оставалась незамеченной многие годы.



(a) Неудачный поиск числа 583



(b) Неудачный поиск числа 450

Рисунок 11.14. Бинарный поиск, неудачные примеры поиска

Что за ошибка? Вернемся к алгоритму 11.6 и взглянем на строчку 4. В ней простое математическое выражение, с помощью него вычисляется середина между двумя числами, которая затем округляется до меньшего целого числа. Мы вычисляем середину путем сложения l и h и деления их суммы пополам. С математической точки зрения тут все верно, но как только мы отходим от

теории и приступаем к реализации, может возникнуть путаница. Два числа, *l* и *h*, являются положительными числами. Складывая их, мы всегда получаем целое число, которое больше каждого из них. Пока не приступим к сложению на компьютере.

11.7. Представление целых чисел на компьютере

У компьютеров есть свой предел возможностей: их память не бесконечна. Это значит, что не стоит складывать слишком огромные числа. В огромном числе цифр будет больше, чем может вместить память компьютера. Если *l* и *h* помещаются в память, то *l* + *h* уже не влезут, и тогда операция сложения невозможна.

Чаще всего ограничение куда меньше доступной памяти. Да и большинство программных языков не дают нам использовать произвольно заданные числа. Они не дают нам перегрузить память, так как из соображений работоспособности у них установлен лимит на размер чисел, с которыми они могут справиться. Они используют последовательность из *n* битов для выражение целого числа в виде бинарного, где *n* является предварительно заданным числом степени двух, таким как 32 или 64.

Предположим, нам надо выразить только беззнаковые числа, то есть положительные. Если бинарное число состоит из *n* битов $B_{n-1}\ldots B_1 B_0$, его значение равно $B_{n-1} \times 2^{n-1} + B_{n-2} \times 2^{n-2} + .. + B_1 \times 2^1 + B_0 \times 2^0$, где каждое B_{n-1}, B_{n-2}, …, B_0 равно либо 1, либо 0. Первый бит (B_{n-1}) имеет наибольшее значение и называется наиболее значимым битом. Последний же бит называется наименее значимым битом.

Какие числа мы можем выразить с помощью *n* битов? Возьмем, к примеру, четырехбитные числа. С четырьмя битами, каждый из которых равен нулю, мы получаем бинарное число 0000 = 0, в то же время, когда все биты равны единицам, мы получаем бинарное число 1111, которое представляет собой $2^3 + 2^2 + 2^1 + 2^0$. Оно на один меньше, чем бинарное число 10 000, которое представляет собой $2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 2^4$. Теперь же познакомимся с небольшой нотацией: чтобы не перепутать число *b* с бинарным, будем записывать $(b)_2$, поэтому для ясности, что

10 — это 2 в бинарной системе, а не 10 в десятичной, мы будем писать $(10)_2$. Применяя данный метод обозначения, мы получаем $(10000)_2 = (1111)_2 + 1$. Это означает, что у нас $(1111)_2 = (10000)_2 - 1 = 2^4 - 1$. С помощью четырех битов мы можем выразить все числа от 0 до $2^4 - 1$, в общей сложности 2^4 чисел.

Можно все свести к общим понятиям. Еще одна полезная нотация — применение $d\{k\}$ для пояснения, что цифра d повторяется k раз. С помощью данного способа записи мы можем написать $1\{4\}$ вместо 1111. С n битами мы можем выразить числа от нуля до $(1\{n\})_2$, которое равно $2^{n-1} + 2^{n-2} + \ldots + 2^1 + 2^0$. Оно на один меньше, чем $(10\{n\})_2$: $(10\{n\})_2 = (1\{n\})_2 + 1$. Следовательно, $(1\{n\})_2 = (10\{n\})_2 - 1$, и так как $(10\{n\})_2 = 2^n$, мы получаем $(1\{n\})_2 = 2^n - 1$. Отсюда получается, что с помощью n битов мы можем выразить все числа от 0 до $2^n - 1$, в общей сложности 2^n.

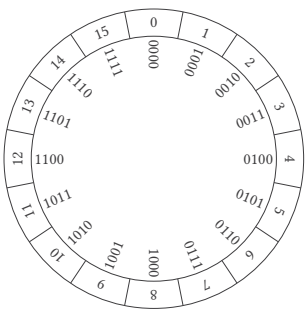


Рисунок 11.15. Числовое колесо

Чтобы понять, как складываются такие числа, вам надо представить, что они расположены на колесе, как на рисунке 11.15, где изображено числовое колесо с выраженными с помощью четырех битов числами. Сложение осуществляется путем передвижения по часовой стрелке. Чтобы вычислить 4 + 7, начните с нуля, пройдите вперед на четыре шага, остановитесь на 4, а затем пройдите еще 7 шагов. Вы получите 11, результат сложения, что и показано на рисунке 11.16.

Пока все идет прекрасно, однако если вы пробуете вычислить 14 + 4, как на рисунке 11.17, вы сперва идете 0 к 14, а затем добавляете четыре шага, что приводит вас к числу 2, которое яв-

ляется результатом сложения. Такая ситуация называется переполнением, потому что вычисление выходит за рамки доступного числового лимита. Если мы проводим арифметические расчеты, мы получаем $14 + 4 = (1110)_2 + (0100)_2 = (10010)_2$. Последнее число содержит пять битов, а не четыре. Компьютеры в такой ситуации просто отсекают слева лишний бит, наиболее значимый бит; по сути происходит то же самое, что с прогулкой по колесу и получением в итоге $(0010)_2 = 2$.

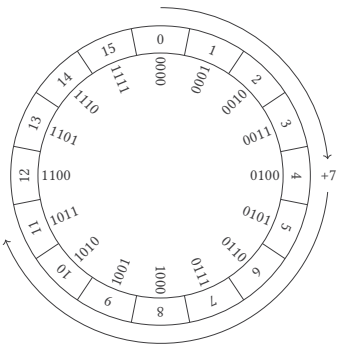


Рисунок 11.16. Вычисление 4 + 7

Вернемся к строчке 4 алгоритма 11.6. Если сумма двух чисел l и h больше, чем наибольшее из возможных целых чисел, верного результата мы не получим, так как произойдет переполнение: число $l + h$ будет неверным, и алгоритм не будет работать.

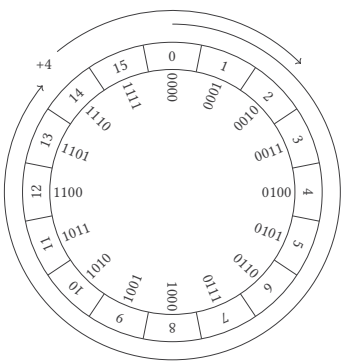


Рисунок 11.17. Вычисление 14 + 4

Если у нас в программном языке 32 бита для выражения беззнаковых целых чисел, тогда множество целых чисел, которые мы можем выразить, будет располагаться в промежутке от 0 до $2^{32} - 1$, или от нуля до 4 294 967 295. Если вы будете складывать такие числа, сумма которых превысит 429 миллиарда, вы столкнетесь с переполнением. Четыре миллиарда когда-то казались заоблачным числом, однако в наши дни ряды данных подобного размера — дело обычное.

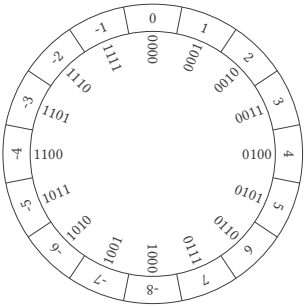


Рисунок 11.18. Числовое колесо дополнительного кода

До сего момента мы полагали, что выражаем только положительные числа, но что происходит, когда наш язык программирования выражает числа со знаком, то есть не только положительные, но и отрицательные целые числа. Чаще всего для выражения отрицательных целых чисел используется последовательности битов, которые начинаются с 1 для отрицательных чисел, в то время как последовательности битов, начинающиеся с 0, соответствуют положительным числам. В данном случае мы называем наиболее значимый бит знаковым битом, так как он указывает на знак числа. Это означает, что у нас осталось $n - 1$ цифр для выражения наших чисел, поэтому мы можем представить 2^{n-1} различных положительных чисел и 2^{n-1} отрицательных. Мы причисляем ноль к положительным числам, так как его знаковый бит — ноль. Если при сложении наши числа слишком велики, мы можем получить в результате отрицательное число: одна половина нашего числового колеса (правая часть) будет заполнена положительными числами, а другая половина (левая часть) — отрицательны-

ми числами. Переполнение будет случаться всякий раз, когда мы складываем одно положительное число с другим, которое уносит нас из мира положительных чисел в мир отрицательных.

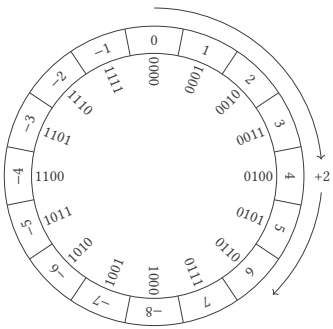


Рисунок 11.19. Вычисление 4 + 2 в дополнительном коде

Например, взгляните на рисунок 11.18, на котором изображены знаковые четырехбитные целые числа. На нем показаны целые числа, представленные с помощью системы, называемой дополнительным кодом, или кодом с дополнением до двух. Знаковый бит для положительных чисел равен нулю, и они располагаются в диапазоне от 0 до 7, или от $(0000)_2$ до $(0111)_2$. Отрицательные числа следуют этому же правилу: берется положительное число x с n битами. Отражать их отрицательность в бинарном виде будет число $c = 2^n - x$, то есть число, которое добавляется к x и дает 2^n. Мы дополняем до 2^n, отсюда и название. Так что минус числа $5 = (0101)_2$ является бинарным числом, которое получается в результате $2^4 - 5 = 16 - 5 = 11 = (1011)_2$. Вы можете убедиться, что с данной системой мы получим отрицательные переполнения: 4 + 2 дает верный результат, как показано на рисунке 11.19, в то время как 4 + 7 из-за переполнения выдает отрицательное число, –5, как можно видеть на рисунке 11.20.

В целом, если у нас n цифр — потому что у нас 2^{n-1} цифр для положительных чисел плюс еще ноль и 2^{n-1} цифр для отрицательных чисел, — мы можем выразить числа от 0 до $2^{n-1} - 1$ и от –1 до -2^{n-1}. Для 32 битов наибольшее положительное число составляет 2 147 483 647. Число просто огромное, но, по меркам на-

ших дней, не заоблачное. Если вы превысите его, то вместо получения большого числа вы получите $-2^{32} = -2\ 147\ 483\ 648$. Что случилось в 2014 году, когда по Интернету разошлась популярная песня «Gangnam Style». Первого декабря мы увидели, что количество просмотров на YouTube перевалило за 2 147 483 647. Очевидно, когда разрабатывался YouTube, никто не думал, что такое возможно. Сервис обновили, и счетчик просмотров стал использовать 64-битные целые числа. Теперь переполнение ему грозит лишь после $2^{63} = 9\ 223\ 372\ 036\ 854$ просмотров.

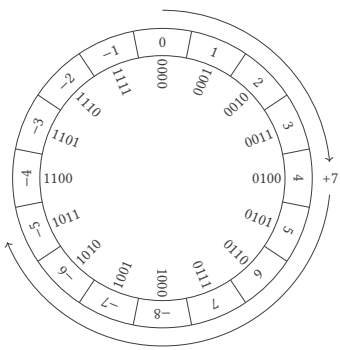


Рисунок 11.20. Вычисление 4 + 7 с помощью дополнительного кода

Дополнительный код может показаться довольно странным устройством, так как, похоже, есть простые решения. Один из альтернативных способов — создать противоположное число путем зеркального отражения всех его битов. Так, например, противоположностью $(0010)_2 = 2$ будет $(1101)_2$. Такой метод называют обратным кодом, или кодом с дополнением до единицы. Мы дополняем каждую отдельную цифру в соответствии с единицей, иными словами, формируем противоположное число путем добавления числа, нужного для получения единицы. Еще один вариант — метод, называемый представлением в прямом коде со знаком: мы создаем противоположное число путем смены его знакового бита. Таким образом, противоположностью $(0010)_2 = 2$ будет $(1010)_2$.

Код с дополнением до двух является наиболее распространенным способом представления целых чисел со знаком по двум причинам.

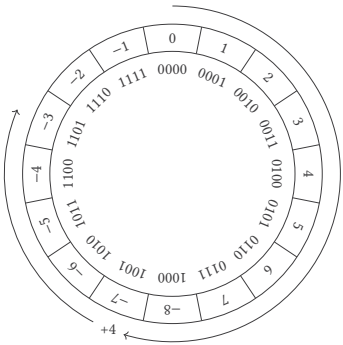


Рисунок 11.21. Вычисление 4 – 7 с помощью дополнительного кода

Во-первых, в коде с дополнением до единицы и представлении в прямом коде со знаком вы получаете два нуля, $(0000)_2$ и $(1111)_2$, что порождает путаницу. Во-вторых, и сложение, и вычитание проще выполнять в коде с дополнением до двух. Вычитание выполняется как сложение чисел с противоположными знаками. Чтобы найти x и y, вам надо взять числовое колесо, найти $-y$ и пройти x шагов по часовой стрелке, как показано на рисунке 11.21.

11.8. И снова к бинарному поиску

Вернемся к бинарному поиску. К счастью, ситуация легко разрешается. Вместо расчетов $\lfloor (l + h)/2 \rfloor$ мы можем вычислить эквивалентное $l + \lfloor (h - l)/2 \rfloor$, что даст нам верное значение для m и не приведет к переполнению. Вместо нахождения середины между двумя числами путем их сложения и деления суммы пополам, мы находим разницу этих двух чисел, делим ее пополам и прибавляем к полученному частному наименьшее число — вот и искомая середина. Действительно, так как l — целое число, мы имеем $l + \lfloor (h - l)/2 \rfloor = \lfloor l + (h - l)/2 \rfloor = \lfloor (l + h)/2 \rfloor$. На рисунке 11.22 изображены два равнозначных вычисления срединного числа. Из-за расчетов $(h - l)$, которое в любом случае меньше h, не случится переполнения; сумма $\lfloor (h - l)/2 \rfloor$ и l не выйдет за пределы h, поэтому и здесь бояться переполнения не надо. Решение описано в алгоритме 11.7, в строчке 4.

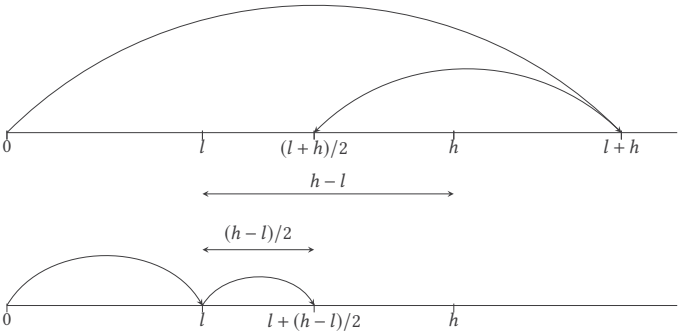


Рисунок 11.22. Избежание переполнения при вычислении срединного числа

Все это может показаться вам очевидным, и, если обернуться в прошлое, так оно и есть. История с досадной ошибкой произошла не с нами, поэтому не нам судить поистине выдающихся людей тех времен. Мы можем извлечь урок, что даже люди великого ума порой спотыкаются на мелочах, а потому стоит помнить о главнейшем человеческом качестве, которое следует иметь всем программистам: о скромности. Неважно, насколько вы умны и насколько хороши в своем деле — промахов и ошибок не избежать. Люди, которые хвастаются тем, что их код безупречен и в нем нет прорех, как правило, не написали ни единого стоящего кода для промышленного пользования. Морис Уилкс, еще один первопроходец в области компьютерных наук, вспоминая о временах зарождения программирования, когда он в 1949 году работал с ЭДСАКом* в Кембриджском университете, писал:

«Однажды, когда я в очередной раз шел из комнаты, где располагался ЭДСАК, к перфорационному оборудованию, на меня со всей мощью снизошло озарение, сродни словам «ибо кроток и смирен я»: я вдруг осознал, что остаток жизни, лучшую его часть, проведу за поиском ошибок в собственных программах».

Данная мысль должна занять почетное место среди самых точных и провидческих высказываний в истории программирования.

---

* EDSAC — электронный автоматический вычислитель с памятью на линиях задержки.

Кстати, под перфорационным оборудованием подразумевается бумажное перфорационное оборудование, которое использовалось для ввода компьютерных данных, а вовсе не перфораторы.

Алгоритм 11.7. Надежный бинарный поиск без переполнения

```
SafeBinarySearch(A, s) → i
    Вводные данные: A, отсортированный массив с элементами;
    s, искомый элемент.
    Выводимые данные: i, позиция s в A, если в A содержится s,
    или иначе –1.
1   l ← 0
2   h ← |A|
3   while l ≤ h do
4       m ← l + ⌊(h − l)/2⌋
5       c ←Compare(A[m], s)
6       if c < 0 then
7           l ← m + 1
8       else if c > 0 then
9           h ← m − 1
10      else
11          return m
12  return −1
```

Бинарный поиск — полезный алгоритм. Более того, бинарный поиск в среднем быстрее любого другого. Каждый раз, когда мы выполняем сравнение в строчке 5 алгоритма 11.7, мы либо ищем нужный нам элемент, либо разбиваем область поиска надвое, продолжая поиск либо в одной, либо в другой половине. В свою очередь, каждый раз, когда мы делим пополам, число элементов, среди которых ведем поиск, уменьшается. Мы не можем вечно делить на два: количество раз, когда мы можем поделить число пополам, является логарифмическим основанием двух этих чисел: для n оно равно $\lg n$.

11.9. Деревья сравнений

Один из способов визуально изобразить то, что происходит, — прибегнуть к дереву сравнений, как на рисунке 11.23. Данное дере-

во показывает различные сравнения, которые могут быть проведены в массиве из 16 элементов, $A[0]$, $A[1]$,… $A[15]$. В начале работы алгоритма мы сравниваем элемент, который ищем, s, с элементом $A[\lfloor(0 + 15)\rfloor/2] = A[7]$. Если $s = A[7]$, мы останавливаемся, если же $s < A[7]$, мы спускаемся на левое поддерево, а если $s > A[7]$, мы спускаемся на правое поддерево. Левое поддерево содержится элементы $A[0]$, $A[1]$… $A[6]$, а правое поддерево — элементы $A[8]$, $A[9]$… $A[15]$. В левом поддереве мы сравниваем s с $A[\lfloor(0 + 6)\rfloor/2] = A[3]$. В правом поддереве мы сравниваем s с $A[\lfloor(8 + 15)\rfloor/2] = A[11]$. Дерево прорастает вниз путем расщепления поисковой области напополам до тех пор, пока мы не окажемся в поисковой области, состоящей из одного элемента. Они являются листьями дерева. Спустившись по ним ниже, вы увидите прямоугольные узлы: они соответствуют значениям s, которое находится не в массиве A. Если мы приходим в точку, где, сравнивая s с $A[8]$, мы получаем, что $s > A[8]$, то наш поиск провалился, потому что s находится между $A[8]$ и $A[9]$, то есть $s \in (A[8], A[9])$. Точно так же, если мы прибываем в точку, где, сравнивая s с $A[0]$, мы получаем, что $s < A[0]$, наш поиск не удался, так как $s \in (-\infty, A[0])$.

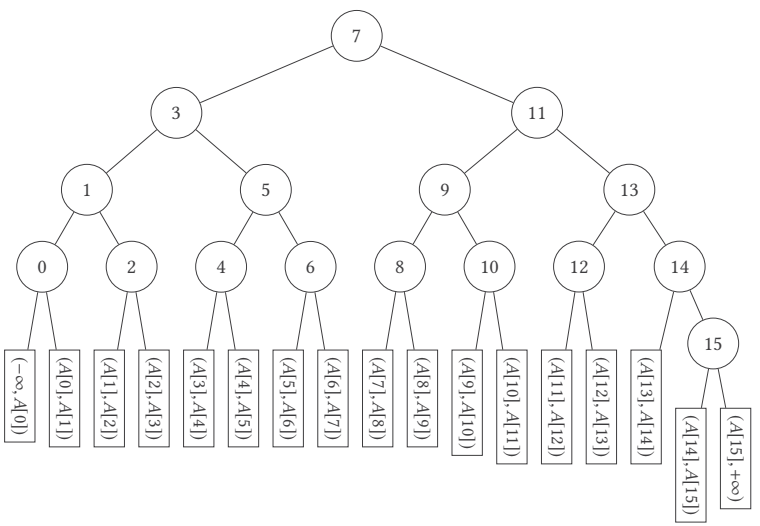


Рисунок 11.23. Дерево сравнений, представляющее бинарный поиск для 16 элементов

Основной принцип построения дерева сравнений базируется на том, что если n — это число элементов массива A, то мы берем элемент $A[\lfloor n/2 \rfloor]$ в качестве корня дерева. Первые $\lfloor n/2 \rfloor - 1$ элементов мы помещаем в левое поддерево, оставшиеся $n - \lfloor n/2 \rfloor$ элементов — в правое. Мы проделываем то же самое с каждым поддеревом до тех пор, пока не получим поддеревья, состоящие из одного элемента, в таком случае у нас будет только одиночный, корневой узел.

Деревья сравнений позволяют нам изучить ход работы бинарного поиска. Число сравнений зависит от числа круглых узлов дерева — мы не считаем прямоугольные узлы корректными узлами дерева. Способ построения дерева, когда мы рекурсивным путем создаем поддеревья, означает, что все листья, кроме самого последнего, заполнены. В самом деле, в каждом узле мы разделяем поисковую область надвое и переходим с этого узла вниз. На рисунке 11.24 изображены деревья, соответствующие значениям n от 1 до 7. Если в области поиска содержится более одного узла, мы получим два поддерева, и число элементов в двух поддеревьях не различаются более, чем на один, что случится, если элементы нельзя нацело разделить пополам. Невозможно перейти на новый уровень, пока предыдущий заполнен. Не может быть кривобоких деревьев, как на рисунке 11.25.

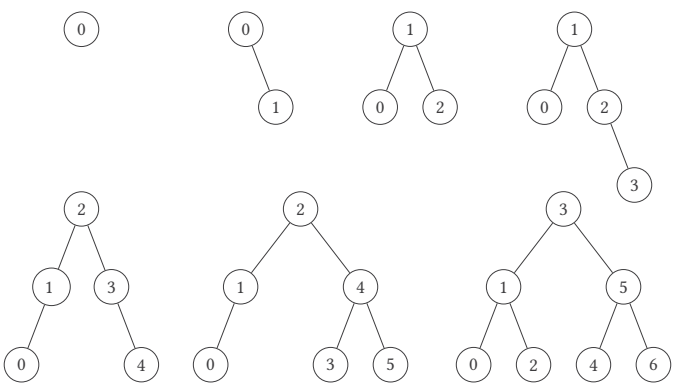


Рисунок 11.24. Деревья сравнений для поиска от 1 до 7 элементов

Корневой, нулевой уровень дерева сравнений содержит один узел. Первый уровень дерева может содержать до двух узлов, но

вовсе не обязательно, чтобы их было именно два. У дерева сравнений может быть два уровня со всего лишь одним узлом на первом уровне; у корня лишь одно дитя, как у второго дерева на рисунке 11.24. На втором уровне может находиться до $2 \times 2 = 2^2$ узлов. Напоминаем, что их может быть и меньше; данное число — возможный максимум при полностью заполненном уровне. Продолжая в том же духе, мы доходим до уровня k, который может содержать до 2^k узлов. Следовательно, дерево сравнений с последним уровнем k может содержать в общей сложности до $1 + 2 + \ldots + 2^k$ узлов. Обратите внимание, что $1 = 2^0$ и $2 = 2^1$, поэтому дерево сравнений, где последним уровнем будет уровень k, может иметь до $2^0 + 2^1 + \ldots + 2^k$ узлов включительно. Но мы уже встречались с данным числом: $(1\{k + 1\})_2$, которое, как мы знаем, равно $2^{k+1} - 1$. Следовательно, мы имеем $n \leq 2^{k+1} - 1$, или эквивалентное $n < 2^{k+1}$.

Так как уровень k — последний уровень, все уровни до и включая $k - 1$ будут целиком заполнены. Применяя всю ту же, изложенную выше, логику, мы получаем, что у дерева будет $2^k - 1$ узлов на уровнях до и включая уровень $k - 1$. Так что у нас $n > 2^k - 1$ или равноценное $n \geq 2^k$. Объединяем наши результаты и получаем $2^k \leq n < 2^{k+1}$.

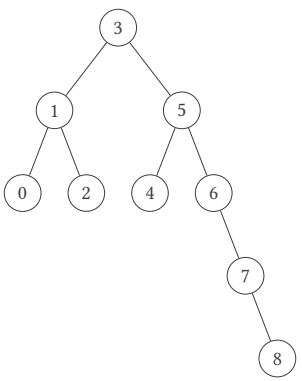


Рисунок 11.25. Такое дерево сравнений нельзя получить

Удачный поиск может прекратиться на любом из уровней дерева, включая уровень k, где требуется все сравнения от одного до $k + 1$, поэтому, если $2^k \leq n < 2^{k+1}$, минимальное количество срав-

нений равно единице, а максимальное — $k + 1$. Если уж мы заговорили об успешных сравнениях, то, если $n = 2^{k+1} - 1$, последний уровень будет заполнен и нам нужно ровно $k + 1$ сравнений. Если $2^k \le n < 2^{k+1} - 1$, то последний уровень заполнен не до конца и нам нужно либо k, либо $k + 1$ сравнений.

Если перевести все вышеизложенное в область сложности, то успешный поиск равен $O(\lfloor \lg n \rfloor) = O(\lg n)$, а неудачный — $\Theta(\lg n)$. Помните, мы говорили, что в игре-угадайке верное число находится максимум за семь попыток? Теперь мы понимаем почему. При бинарном поиске среди 100 элементов вам понадобится $\lg 100 \approx 6,65 < 7$ попыток, чтобы отгадать задуманное число.

Вспомните, что $2^{32} - 1 = 4\,294\,967\,295$. Это значит, что любой поиск в отсортированном массиве из 4 294 967 295 элементов потребует не более 32 сравнений. Сопоставьте это с миллиардами сравнений, которые требуются методу перебора при работе с тем же массивом.

Можно восхищаться мощью повторяющегося раз за разом деления — что является сутью логарифма, — глядя и на обратный процесс, повторяющееся умножение, или возведение в степень. Хорошей иллюстрацией можно считать миф, согласно которому появились шахматы. Говорят, что, когда изобрели шахматы, правителю страны, где их придумали, они понравились настолько, что он призвал к себе изобретателя и спросил, какую тот хочет награду. Изобретатель сказал, что желает в награду столько риса (или пшеницы, в зависимости от версии мифа), сколько получится, если одну крупицу положить на первую клетку, две крупицы — на вторую, четыре — на третью и так далее. Правитель повелел исполнить желание, но вскоре казначей сообщил ему, что всех запасов королевства не хватит, чтобы выдать назначенную награду.

На рисунке 11.26 вы можете увидеть, почему так произошло. Показатель степени повышается каждый раз, когда мы переходим на следующую клетку; отсюда следует, что размер награды растет в геометрической прогрессии. На последней клетке получается 2^{63} крупиц, что равно где-то 9 с 18 нулями. Задумайтесь на минутку: мы можем найти из 9×10^{18} элементов единственный

нужный всего за 63 сравнения. Сила логарифма потрясающа, как и обратная его операция — экспоненциальный рост.

2^{56}	2^{57}	2^{58}	2^{59}	2^{60}	2^{61}	2^{62}	2^{63}
2^{48}	2^{49}	2^{50}	2^{51}	2^{52}	2^{53}	2^{54}	2^{55}
2^{40}	2^{41}	2^{42}	2^{43}	2^{44}	2^{45}	2^{46}	2^{47}
2^{32}	2^{33}	2^{34}	2^{35}	2^{36}	2^{37}	2^{38}	2^{39}
2^{24}	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}
2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
2^{8}	2^{9}	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
2^{0}	2^{1}	2^{2}	2^{3}	2^{4}	2^{5}	2^{6}	2^{7}

Рисунок 11.26. Экспоненциальный рост на шахматной доске

Идея разрешать проблему путем ее дробления на части — полезный инструмент. Подход «разделяй и властвуй» применяется в широком кругу задач, которые нам помогают решить компьютеры. Он также хорошо подходит для решения головоломок. Представьте, что вас просят решить следующую задачу: у вас девять монет, одна из которых фальшивая и легче всех остальных. У вас есть весы. Как найти фальшивую монету, если весами можно воспользоваться только два раза?

Ключевым моментом для нахождения ответа является тонкое наблюдение, что мы можем использовать весы для разделения нашей поисковой области на три части (а не на две, как при бинарном поиске). Представим, что мы пометили монеты как c_1, c_2… c_9. На одну чашу весов мы кладем c_1, c_2, c_3, а на другую — c_7, c_8, c_9. Если они равны по весу, то мы знаем, что фальшивка находится среди монет c_4, c_5, c_6. Теперь мы берем c_4, c_5. Если их вес равен, подделкой является c_6. Если же не равен, то поддельная монета та, чей вес на весах меньше. Если же c_1, c_2, c_3 и c_7, c_8, c_9 не равны по

весу, тогда, опять же, мы сразу понимаем, в каком трио находится фальшивка, и проделываем с данными тремя монетами все то же, что мы описали для монет c_4, c_5, c_6. Вы можете посмотреть весь ход взвешиваний на рисунке 11.27, где вес монет c_i, c_j, c_k мы обозначили как $w(c_i, c_j, c_k)$. Каждому внутреннему узлу соответствует процесс взвешивания, а каждому листу — фальшивая монетка.

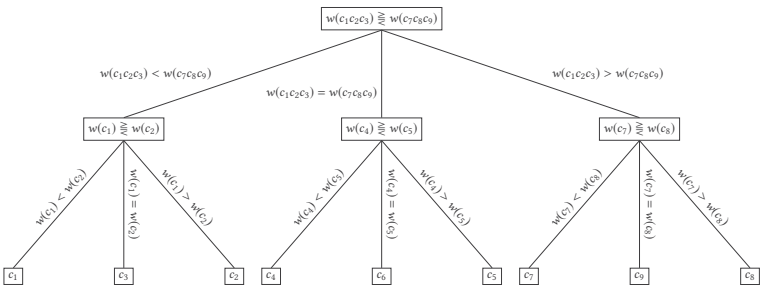


Рисунок 11.27. Дерево взвешивания монет

Так как на каждом уровне мы разделяем область поиска на три части, общее количество шагов, требуемых для завершения поиска, высчитывается с помощью логарифмического основания 3, а не 2, как мы делали прежде. Для девяти монет у нас $\log_3(9) = 2$, что и является высотой дерева на рисунке 11.27.

Примечания

Джордж Кингсли Ципф изложил свои наблюдения в двух книгах в 1935 году [226] и 1949 [227]. Брауновский корпус был составлен из 500 английских текстовых отрывков в 1960-х годах Генри Кучерой и У. Нельсоном Франсисом, работавшими в Брауновском университете. Вильфредо Парето описал распределение степенных законов, хотя в XIX веке никто их так еще не называл [154]. В начале XX века Джордж Удни Юл в своих исследованиях происхождения биологических видов тоже открыл степенные законы [223]. Общее внимание на степенные законы обратил Барабаси [10], [11]. Если вам интересна история, посмотрите доклад Ми-

ценмахера [143]. Если интересен критический взгляд на степенные законы, то обратитесь к размышлениям Стампфа и Портера [194].

Фрэнк Бенфорд открыл закон, названный его именем, в 1938 году [14]; тем не менее не он первым опубликовал его. Данный закон был опубликован ранее, в 1881 году [149] математиком и астрономом Саймоном Ньюкомом. Чтобы подробней узнать о получении закона Бенфорда, смотрите работу Хилла [92], а за пояснениями обращайтесь к работе Фьюстера [65].

Помимо методов перестановки и движения к началу существует еще и метод подсчета. В данном подходе мы считаем количество раз, когда элемент был посчитан, и заносим наши подсчеты в список так, чтобы элементы, посчитанные наибольшее количество раз, стояли во главе списка. Самоорганизующийся поиск был придуман Джоном Маккейбом [135]. Доказательство, что метод перестановки при долгосрочной перспективе требует меньше сравнений, чем метод движения к началу, было представлено Рональдом Ривестом [166]. Слетор и Тарьян показали, что общие затраты метода движения к началу связаны с постоянным множителем четырех из оптимума; с методами перестановки и подсчета все по-другому [189]. Их результаты помогли экспериментам Бентли и Макгиоха, которые показали, что на практике в большинстве случаев выгодней применять метод движения к началу [17]. Чтобы узнать о более новых исследованиях и способах реализации, посмотрите работы Бахраха и Эль-Янива [5] и Бахраха, Эль-Янива и Рейнштедтлера [4].

У задачи о разборчивой невесте* интересная история; впервые ее напечатали в феврале 1960 года в журнале Scientific American, в рубрике Мартина Гарднера, решение же напечатали в марте 1960 года; посмотрите доклад Фергюсона [64], где он указывает, что подобные задачи восходят к Кеплеру. Решение варианта, где все кандидаты оцениваются согласно их достоинствам, было дано Бирденом [12].

---

* В англоязычной литературе она известна как «Задача о секретаре», в которой секретарь просматривает резюме в поисках нового сотрудника.

Согласно Кнуту [114], бинарный поиск уходит корнями к началу компьютерной эпохи. Джон Мокли, один из разработчиков ЭДСАКа, первой универсальной электронно-вычислительной машины, описал его в 1946 году. Тем не менее не ясно, что делали, когда число элементов в массиве, в котором производился поиск, не являлось степенью двух. Первое описание бинарного поиска, который смог работать с массивом без данного ограничения, появилось в 1960 году, его дал математик Деррик Генри Лемер, который также внес большой вклад в развитие программирования. Проблемы, с которыми сталкиваются профессиональные программисты при реализации бинарного поиска, описаны Джоном Бентли в его книге [16]. Исследование, которое показало, что три четверти учебников содержат ошибки в описании бинарного поиска, было проведено Ричардом Паттисом [155]. Джошуа Блох объявил о своем промахе в исследовательском блоге Гугла [21]. Цитата Мориса Уилкса взята из его мемуаров [217, с. 145].

12 Сортировочный компот

Что общего в упорядоченном списке имен, в библиотеке с песнями, в размещении объектов в хронологическом порядке, в сортировке наших писем в алфавитном порядке по имени отправителя или названию темы или же дате получения? Все они требуют сортировки элементов. То, что упомянутые приложения раскладывают все по порядку, мы и так знаем, но сортировка не ограничивается одними только очевидными примерами. Подумайте о компьютерной графике. Когда рисуешь картину на компьютере, важно рисовать разные ее части по порядку. В частности, дальний план нужно рисовать в первую очередь, а ближний план рисуется уже потом. Таким образом мы можем быть уверены, что объекты переднего плана закрывают собой объекты дальнего плана. Поэтому нам надо обозначить разные части, рассортировать их и нарисовать в правильном порядке, от дальнего к ближнему — это называется алгоритмом художника.

Перейдем от компьютерной графики к биологии, сортировка является основным компонентом в алгоритмах вычислительной биологии; переходим от биологии к сжатию данных, сортировка лексикографических вариантов данных используется в основном методе сжатия, так называемом преобразовании Барроуза — Уилера, которое также называется сжатием блочной сортировки.

Мы проводим в Сети не по одному часу в день и наверняка сталкиваемся с рекомендациями обратить внимание на тот или иной товар. Системы рекомендаций делают две вещи: они фильтруют наши запросы, отбрасывая то, что нас не интересует, и сортируют то, что может представлять для нас интерес, поэтому мы видим наиболее подходящие варианты.

Редко встретишь компьютерное приложение, в котором не использовалась бы сортировка; по сути дела, нужда в сортировке предшествовала появлению цифровых компьютеров. «Сортиро-

вочный ящик» был разработан для совместного использования со счетно-аналитической машиной Германа Холлерита в 1880-х годах. Холлерит изобрел счетно-аналитическую машину для помощи в вычислениях во время переписи в 1890 году, когда население США выросло до таких размеров, что проведение переписи вручную заняло бы тринадцать лет — к следующей переписи населения, в 1900 году, может и уложились бы.

Так как сортировка встречается почти что везде, не удивительно, что люди с давних пор выискивали методы сортировки; первые сортировочные методы, использованные в компьютерах, были разработаны одновременно с этими компьютерами, во второй половине XX века. Исследования методов сортировки ведутся до сих пор, они касаются как улучшения уже известных методов и их реализаций, так и разработки новых алгоритмов, которые оптимальны для той или иной специфической области. К счастью, хоть и невозможно полностью охватить всю сферу сортировки, можно проследить ее основные принципы и базовые алгоритмы. Особо радует то, что алгоритмы, с которыми мы, сами того не зная, сталкиваемся каждый день, не сложно найти и исследовать. Как мы увидим, основные сортировочные алгоритмы занимают не более нескольких строчек псевдокода, чтобы описать их.

Сортировка связана с поиском, так как поиск более эффективен, если мы ищем среди упорядоченных элементов. В сортировке применяется та же терминология, что и в поиске: данные, содержащиеся в записях, или, по-другому, базах данных. У записей может быть несколько атрибутов, но мы проводим сортировку, используя один из них, который называется ключом. Если ключ содержит более одного атрибута, тогда мы имеем дело со составным ключом.

12.1. Сортировка методом выбора

Наверное, самый простой сортировочный алгоритм состоит из нахождения минимального элемента среди наших объектов, его извлечения, нахождения минимального элемента среди остав-

шихся и добавления его к тому, что мы изъяли ранее; данный процесс повторяется до тех пор, пока мы не переберем все элементы. Если у вас на столе куча бумаг, каждая из которых обозначена номером, тогда вы ищете наименьший, достаете его, затем приступаете к поиску следующего наименьшего числа, а когда находите, помещаете рядом с уже найденным. Так вы продолжаете разбирать бумаги до тех пор, пока не останется кучи на столе.

Такая простая процедура называется сортировкой методом выбора, потому что каждый раз мы ищем наименьший элемент среди оставшихся. Алгоритм 12.1 реализует сортировку методом выбора в массиве A, который и является его вводными данными; он сортирует A, поэтому ему не нужно ничего возвращать, так как он проводит сортировку на месте; тот же самый, но уже отсортированный A является результатом работы алгоритма.

Алгоритм 12.1. Сортировка методом выбора

```
SelectionSort(A)
    Вводные данные: A, массив элементов, которые надо отсортировать.
    Результат: A отсортирован.
1   for i ← 0 to |A| − 1 do
2       m ← i
3       for j ← i + 1 to |A| do
4           if Compare(A[j], A[m]) < 0 then
5               m ← j
6       Swap(A[i], A[m])
```

Если нам надо отсортировать n элементов, сортировка выборкой выискивает наименьший из всех n элементов, затем наименьший из n – 1 элементов, затем наименьший из n – 2 элементов, пока у нас не останется всего один элемент. Строчка 1 алгоритма 12.1 задает цикл, проходящий через все элементы массива. В каждый проход цикла мы ищем минимальный из элементов, соответствующих нынешнему значению i, вплоть до конца массива. Как и в алгоритме бинарного поиска, мы используем функцию `Compare`(a, b), которая сравнивает два элемента по их ключам и возвращает +1, если a лучше b, –1, если b лучше a, и 0, если они равны.

Мы прибегаем к переменной *m* для поиска наименьшего из оставшихся элементов. Изначально *m* сохраняет индекс нашего стартового элемента в поиске минимума. В строчках 3–5 мы ищем минимум, переходят от элемента в *A*[*i* + 1] к концу массива. То есть для каждого *i* внешнего цикла мы ищем минимум из оставшихся *n* – *i* элементов. Как только мы находим минимум, то прибегаем к функции `Swap`(*a*, *b*), чтобы перенести наш минимум в нужную часть массива.

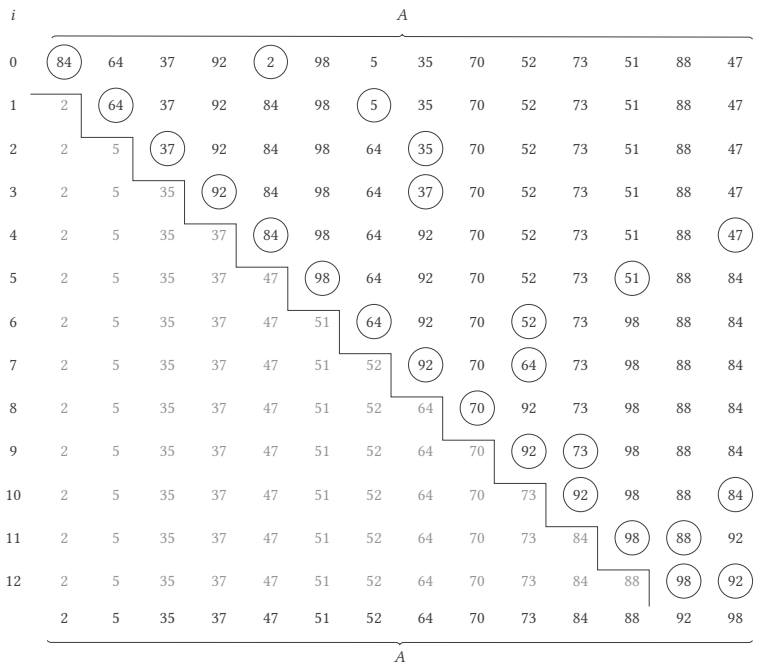


Рисунок 12.1. Пример сортировки методом выбора

На рисунке 12.1 изображена работа сортировки методом выбора в массиве из 14 элементов. В столбце слева указано значение *i*; в других столбцах содержится массив. При каждой итерации мы ищем наименьший элемент из оставшихся неотсортированных элементов массива и меняем его с первым в ряду оставшихся неотсортированных. В каждой итерации мы обводим эти два элемента кружочком. Закрашенная серым область в нижнем левом

углу отвечает за уже отсортированные элементы. В каждой итерации мы ищем минимум в правой части рисунка, которая, как вы видите, становится все меньше. Обратите внимание, что происходит, когда $i = 8$. Наименьший из неотсортированных элементов оказывается первым в их ряду, поэтому здесь у нас только один кружок.

Чтобы изучить ход работы сортировки методом выбора обратите внимание, что, если у нас n элементов, внешний цикл выполняется $n - 1$ раз, поэтому у нас $n - 1$ замены. В каждой итерации внешнего цикла мы сравниваем все элементы от $i + 1$ до конца массива A. В первый проход мы сравниваем все элементы от $A[1]$ до $A[n - 1]$, поэтому у нас $n - 1$ сравнений. Во второй проход мы сравниваем все элементы от $A[2]$ до $A[n - 1]$, поэтому у нас $n - 2$ сравнений. В последнем проходе цикла мы сравниваем оставшиеся два элемента, $A[n - 2]$ и $A[n - 1]$, и у нас всего одно сравнение. Получается, что всего сравнений у нас

$$\begin{aligned} 1 + 2 + \cdots + (n-1) &= 1 + 2 + \cdots + (n-1) + n - n \\ &= \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} . \end{aligned}$$

Следовательно, сложность сортировки методом выбора равна $\Theta(n - 1) = \Theta(n)$ заменам и $\Theta(n(n - 1)/2) = \Theta(n^2)$ сравнениям. Мы разделяем замены и сравнения, потому что у одной операции может быть совсем иная вычислительная стоимость, чем у другой. Обычно сравнения производятся быстрее, чем замены, так как операция замены включает в себя перемещение данных. Чем больше переставляемые данные, тем очевидней становится разница.

Вспомните теперь, что произошло с восьмеркой на рисунке 12.1: тогда элемент менялся местами сам с собой. Это кажется бессмысленным: зачем меняться с самим собой местами? Когда первый элемент в неотсортированном ряду является наименьшим из всех неотсортированных элементов, мы можем просто оставить его на месте, как есть. В этом нам поможет небольшое изменение, которое вы можете наблюдать в алгоритме 12.2. Здесь в строчке 6 мы проверяем, совпадает ли первый элемент неотсортирован-

ного ряда с наименьшим из всех неотсортированных элементов, и, если не совпадает, производим замену. Это значит, что число n является верхней границей для количества замен. Сложность измененного алгоритма составляет $\Theta(n)$ замен и $\Theta(n^2)$ сравнений.

Алгоритм 12.2. Сортировка методом выбора без ненужных замен

```
SelectionSortCheckExchanges(A)
    Вводные данные: A, массив элементов, которые надо отсортировать.
    Результат: A отсортирован.
1   for i ← 0 to |A| − 1 do
2       m ← i
3       for j ← i + 1 to |A| do
4           if Compare(A[j], A[m]) < 0 then
5               m ← j
6       if i ≠ m then
7           Swap(A[i], A[m])
```

Различие в поведении двух алгоритмов наиболее очевидно, когда части массива A уже находятся там, где им положено быть. Если A уже отсортирован, алгоритм 12.1 все равно заменит все n элементов (сами на себя). Алгоритм 12.2 всего лишь отметит, что массив уже отсортирован.

Но все же нельзя с уверенностью сказать, что алгоритм 12.2 на практике работает лучше алгоритма 12.1. Да, мы сделали замены не обязательными; элементы меняются друг с другом местами, только когда необходимо. Однако в то же время мы добавляем еще одно сравнение, в строчке 6. Так как строчка 6 выполняется n – 1 раз, мы добавляем n – 1 сравнений в ход работы алгоритма. Данное действие уравновешивает пользу, которую мы получаем от сокращения числа замен. Поэтому алгоритм 12.2 нельзя считать более оптимизированной версией сортировки методом выбора, которую можно рекомендовать всем, призывая отказаться от алгоритма 12.1. Если мы реализуем и сравним их, мы можем увидеть, что их работа едва различима и зависит от соотношения затрат на замены и затрат на сравнения, а также того, как сортируются наши данные в первую очередь. Прекрасно, когда

понимаешь, почему такое происходит. Также хорошо принять во внимание любые издержки улучшений, которые не всегда сразу же заметны в попытках оптимизировать работу.

У сортировки методом выбора есть несколько плюсов. Она проста для понимания и реализации. Ей требуется небольшое количество замен, что может быть важно, когда перемещение данных оказывается вычислительно затратным делом. Данная сортировка удобна для небольших массивов, в которых она выполняется довольно быстро, однако ее не используют в больших массивах, где более подходящие сортировочные алгоритмы выдают лучшие результаты.

12.2. Сортировка методом вставок

Еще один наглядный метод сортировки используется карточными игроками для сортировки руки. Представьте, что вам надо распределить приходящие в руку карты, одну за другой. Вы берете первую карту, затем берете вторую и размещаете относительно первой в зависимости от ее достоинства. После берете третью карту и помещаете ее в нужное место относительно первых двух. Вы проделываете все то же самое с остальными пришедшими картами. Вы можете посмотреть принцип работы на рисунке 12.2, где пять карт одной масти были рассортированы в восходящем порядке, считая туз главной картой. На самом деле карточные игроки чаще предпочитают помещать самые крупные карты слева, но не будем вдаваться в тонкости карточной игры.

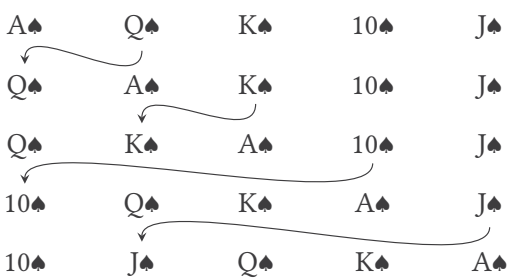


Рисунок 12.2. Сортировка карт

Такая сортировочная процедура называется сортировкой методом вставок, так как мы вставляем каждый элемент в положенное место, принимая во внимание уже отсортированные элементы. Чтобы смастерить алгоритм для описанной нами процедуры, нам нужен способ, который в точности бы описал перемещение карт, отмеченных стрелками, на рисунке 12.2. Первую карту, которую нам надо отсортировать, Q♠, мы попросту меняем местами с A♠. То же самое мы проделываем с K♠, меняя его местами с A♠. Перемещение 10♠ — более сложная задача, так как одной перестановкой здесь не обойдешься. Тем не менее перемещение 10♠ можно описать верной серией перестановок. Сначала мы меняем местами 10♠ и A♠. Затем меняем 10♠ и K♠. И наконец, меняем 10♠ и Q♠. То же самое, как если бы мы привели в порядок руку: Q♠K♠A♠10♠J♠ ⇝ Q♠K♠10♠A♠J♠ ⇝ Q♠10♠K♠A♠J♠ ⇝ 10♠Q♠K♠A♠J♠. Таким образом, для каждой карты, которую мы хотим поместить в надлежащее место, мы меняем ее местами с соседней левой картой до тех пор, ее значение не будет выше значения карты слева. Вся эта последовательность отображена в алгоритме 12.3 в соответствующих терминах.

Алгоритм 12.3. Сортировка методом вставок

```
InsertionSort(A)
    Вводные данные: A, массив элементов, которые надо отсортировать.
    Результат: A отсортирован.
1   for i ← 1 to |A| do
2       j ← i
3       while j > 0 and Compare(A[j − 1], A[j]) > 0 do
4           Swap(A[j], A[j − 1])
5           j ← j − 1
```

Алгоритм начинается с того, что в строчке 2 мы устанавливаем j в позицию второго элемента массива A, состоящего из элементов, требующих сортировки. Затем в строчках 3–5 происходит сравнение этого элемента с предыдущим и, если нужно, смена местами. Возвращаясь к строчке 2, мы проделываем то же самое с третьим элементом массива, присваивая j новое значение i. Каждый раз, проходя через строчку 2, переменная j занимает изначальную

позицию элемента, который мы передвигаем каждую итерацию внутреннего цикла. Значение j сокращается в строчке 5 внутреннего цикла, когда $A[j]$ выше, чем $A[j-1]$. Условие $j > 0$ во внутреннем цикла обязательно, чтобы удостовериться, что мы не выйдем за рамки начала массива, когда продвигаемся задом наперед, сравнивая и меняя местами. Оно нужно, когда мы перемещаем элемент в начало массива: тогда $j = 0$ и не существует $A[j-1]$, поэтому вторая проверка выдаст ошибку. Обратите внимание, что здесь мы используем упрощенное вычисление; мы не будем высчитывать $A[j-1] > A[j]$, когда не выполняется условие $j > 0$.

На рисунке 12.3 изображена сортировка методом вставок в массиве с элементами, которые мы использовали в 12.1. В целом ход работы сортировки методом вставок кое-чем напоминает сортировку методом выбора: уже отсортированной частью, в левом нижнем углу. Различие же в том, что в сортировке методом выбора элементы, выделенные серым цветом, упорядочены и останутся на своих позициях до конца, в то время как в сортировке методом вставок элементы, выделенные серым цветом, упорядочены между собой и не факт, что их порядок останется неизменным, так как между ними может быть вставлен элемент из правой части.

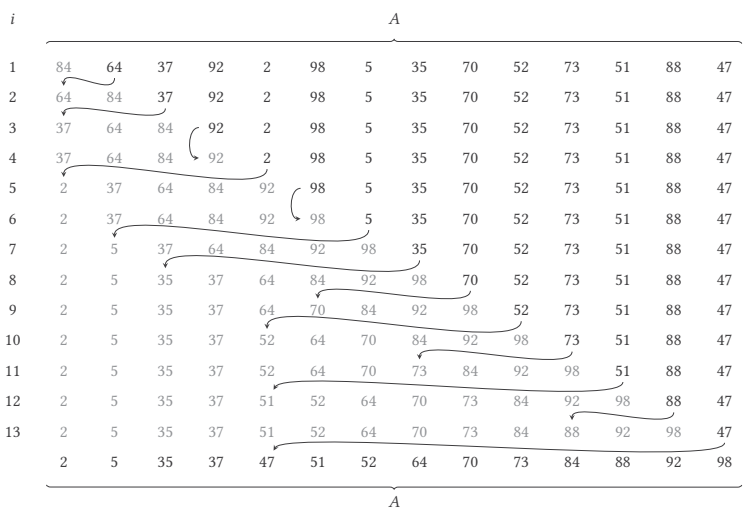


Рисунок 12.3. Пример сортировки методом вставок

Перемещение элемента при каждой итерации внешнего цикла может требовать ряд замен. Например, перескок из ряда с $i = 6$ в ряд с $i = 7$ происходит не за раз и требует выполнения нескольких операций: на рисунке 12.4 показано, что на самом деле происходит.

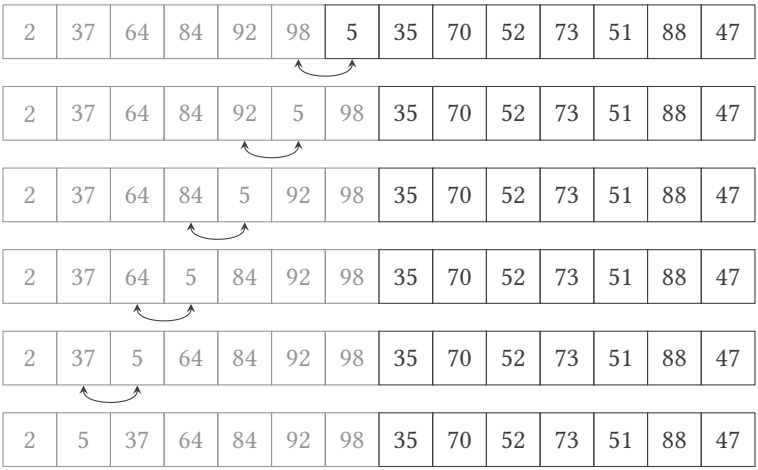


Рисунок 12.4. Перемена местами при $i = 6$ на рисунке 12.3

Чтобы изучить работу сортировки методом вставок, нам надо, как и в сортировке методом выбора, найти количество сравнений и замен. Проще начать с замен. В первом ряду рисунка 12.3 у нас только одна замена, так как $64 < 84$. Во втором ряду у нас две замены, так как $37 < 84$ и $37 < 64$. В третьем ряду нам ничего не надо менять, в то время как в четвертом ряду нужно сделать четыре замены. Можно заметить проступающую закономерность: число замен в каждом ряду равно числу элементов слева от элемента, который мы просматриваем и который меньше этих элементов. В самом деле, в восьмом ряду элемент 70 меньше элементов 98, 92 и 84, поэтому в данном ряду у нас три замены. Когда перед нами последовательность значений и одно значение больше другого, находящегося слева от него, мы, обращаясь к математическому языку, говорим, что мы имеем дело с инверсией двух элементов, так как они находятся по отношению друг к другу в обратном

порядке. Если коротко, инверсия возникает, когда два элемента расположены не по порядку. Тогда в каждом ряду число замен равно числу инверсий, содержащих перемещаемый элемент. Если объединить все ряды, то общее число всех замен равно общему числу инверсий для всех элементов исходного массива. Например, если мы имеем массив с числами 5, 3, 4, 2, 1, тогда у нас следующие инверсии: (5, 3), (5, 4), (5, 2), (5, 1), (3, 2), (3, 1), (4, 2), (4, 1), (2, 1). Это означает, что для сортировки с помощью метода вставок нам понадобится девять замен — можете удостовериться, что это действительно так.

Если массив уже отсортирован, то число инверсий равно нулю. Если массив с n элементами отсортирован в обратном порядке, тогда, если мы начнем с последнего элемента массива, у этого элемента будет $n - 1$ инверсий, потому он расположен в обратном порядке по отношению к другим элементам до него. Если мы берем ставший последним элемент, у этого элемента будет $n - 2$ инверсий по той же самой причине. Так мы продолжаем, пока не дойдем до второго элемента массива, у которого ровно одна инверсия, так как он расположен не по порядку лишь относительно первого элемента массива. В общем счете получается, что массив, элементы которого расположены в обратном порядке, имеет: $1 + 2 + \ldots + (n - 1) = n(n - 1)/2$ инверсий. Доказано, что массив, элементы которого располагаются в случайном порядке, имеет меньше инверсий, а точнее, $n(n - 1)/4$ инверсий. Мы видели, что число инверсий равно числу замен, хотя при самом благоприятном раскладе сортировке методом вставок не требуется ни единой замены, в наихудшем случае требуется $n(n - 1)/2$ замен, в среднем же нужно $n(n - 1)/4$ замен.

Теперь обратимся к количеству сравнений между элементами $A[j]$ и $A[j - 1]$ в строчке 3. Если массив отсортирован, то при каждом прохождении внешнего цикла, то есть для каждого значения i, у нас одно сравнение: следовательно, мы имеем $n - 1$ сравнений. Если массив отсортирован в обратном порядке, тогда, как мы видели, нам требуется $n(n - 1)/2$ замен. Для этого нам надо провести $n(n - 1)/2$ успешных сравнений в строчке 3: каждый элемент будет сравнен со всеми предыдущими вплоть

до самого первого, и мы снова получим уже знакомую нам формулу $1 + 2 + \ldots + (n - 1) = n(n - 1)/2$.

Если элементы массива расположены в случайном порядке, то, так как нам требуется провести $n(n - 1)/4$ замен, мы должны выполнить $n(n - 1)/4$ успешных сравнений. Вместе с ними также надо выполнить и неудачные сравнения. Например, представьте массив 1, 2, 5, 4, 3. Допустим, мы взяли элемент 4. Мы проводим сравнение $5 > 4$, которое будет удачным, а затем делаем сравнение $2 > 4$, которое окажется неудачным. Мы будем выполнять неудачные сравнения каждый раз, когда проходим через строчку 3, имея $j = 0$. Хотя нам неизвестно, сколько раз так случится, мы можем быть уверены, что не более $n - 1$ раз, потому что это максимальное количество раз, когда может случиться неудачное сравнение (если массив отсортирован в обратном порядке, такое случится ровно n раз, но мы уже рассмотрели данную ситуацию). Добавив две меры сравнения, мы получаем самое большое $n(n - 1)/4 + (n - 1)$ сравнений. Показатель $n - 1$ не влияет на общую сложность $n(n - 1)/4$, которую мы уже нашли.

Подытожим. Количество замен в сортировке методом вставок равно 0, если массив отсортирован, $\Theta(n(n - 1)/2) = \Theta(n^2)$ — если массив отсортирован в обратном порядке, и $\Theta(n(n - 1)/4) = \Theta(n^2)$ для массива со случайным порядком. Число сравнений для отсортированного массива равно $\Theta(n - 1) = \Theta(n)$, для отсортированного в обратном порядке — $\Theta(n(n - 1)/2) = \Theta(n^2)$, а для массива со случайным порядком — $\Theta(n(n - 1)/4) + O(n) = \Theta(n(n - 1)/4) = \Theta(n^2)$.

Сортировка методом вставок — как и сортировка методом выбора — проста в реализации. На практике реализации данного вида сортировки работают даже быстрее, чем реализации сортировки методом выбора, что делает сортировку методом вставок удобной для небольших баз данных. Ее также можно использовать в качестве онлайн-алгоритма, то есть алгоритма, который сортирует последовательность элементов сразу же, как только получает их: вовсе не обязательно, чтобы во время начала работы алгоритма ему были доступны все элементы.

Сложность порядка $\Theta(n^2)$ может подходить для маленьких баз данных, но она не практична для работы с большими базами

данных. Если у нас есть один миллион элементов, n^2 возрастает до одного триллиона: числа с двенадцатью нулями. Получается слишком медленно. Если один миллион элементов кажется вам слишком большим числом — а оно не такое уж и большое, — то для 100 тысяч элементов n^2 увеличится до 10 миллиардов, что тоже является большим числом. Если нам хочется управиться с огромными базами данных, тогда надо поискать что-то получше, чем сортировки методом выбора и методом вставок.

12.3. Сортировка кучей

Мы можем найти более выгодный способ, если вернемся к сортировке методом выбора и присмотримся к сопутствующим ей процессам. Каждый раз, когда мы ищем наименьший из неотсортированных элементов, мы осуществляем поиск путем перебора всех неотсортированных элементов. Но, возможно, мы может отыскать более разумный подход. Сперва давайте направим мысль в другое русло и представим сортировочную процедуру, похожую на сортировку методом выбора, в которой мы приступаем к нахождению наибольшего элемента, который затем помещаем в конец. Мы продолжаем данный процесс до тех пор, пока не закончатся элементы. Перед нами сортировка методом выбора наоборот: вместо поисков минимума, который должен занять первую позицию, мы занимаемся поисками максимума, который должен занять последнюю позицию, $|A| - 1$. Вместо нахождения второго наименьшего элемента для второй позиции мы ищем второй наибольший элемент для позиции $|A| - 2$. Очевидно, что это сработает. Теперь, если мы придумаем, как управляться с элементами так, чтобы нам было проще находить максимум из всех неотсортированных элементов, мы получим более удобный алгоритм, которому на выполнение подобной работы потребуется не так уж много времени.

Представим, что мы упорядочиваем элементы в массиве A таким образом, что у нас $A[0] \geq A[i]$ для всех $i < |A|$. Тогда $A[0]$ — это максимум, который мы должны поместить в конец A: мы можем поменять местами $A[0]$ и $A[n - 1]$. Теперь $A[n - 1]$ содержит наибольший из наших элементов, а $A[0]$ предыдущее значение $A[n - 1]$.

Если мы приводим в порядок элементы от $A[0]$ до $A[n-2]$ так, что у нас опять $A[0] \geq A[i]$ для всех $i < |A| - 1$, мы можем повторить ту же процедуру: поменять $A[0]$, который является наибольшим из элементов $A[0]$, $A[1]$, …, $A[n-2]$, на $A[n-2]$. Теперь $A[n-2]$ будет содержать второй наибольший элемент. Затем мы снова продолжаем преобразовать $A[0]$, $A[1]$, …, $A[n-3]$ так, что у нас $A[0] \geq A[i]$ для всех $i < |A| - 2$, и меняем местами $A[0]$ и $A[n-3]$. Если мы будет продолжать в том же духе: находить наибольший из оставшихся элементов и помещать его в конец A — мы в итоге разместим все наши элементы в верном порядке.

На данном этапе нам очень поможет представить массив A в виде дерева, где каждому элементу $A[i]$ соответствует узел с детьми $A[2i+1]$ и $A[2i+2]$. На рисунке 12.5 изображено соответствие между массивом, который мы использовали как пример сортировки, и его представлением в виде дерева. Вы должны понять, что дерево — это всего лишь муляж. Все на самом деле хранится в указанном ниже массиве; тем не менее взгляд на массив, как на дерево с описанным выше правилом, ясно проиллюстрирует происходящее.

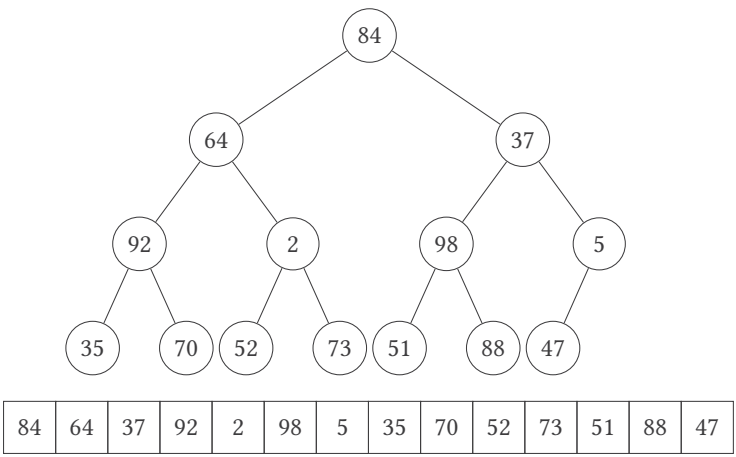


Рисунок 12.5. Проведение параллели между массивом и деревом

Если для каждого i мы имеем $A[i] \geq A[2i+1]$ и $A[i] \geq A[2i+2]$, тогда каждый узел дерева больше или равен своему дитяти. Тогда

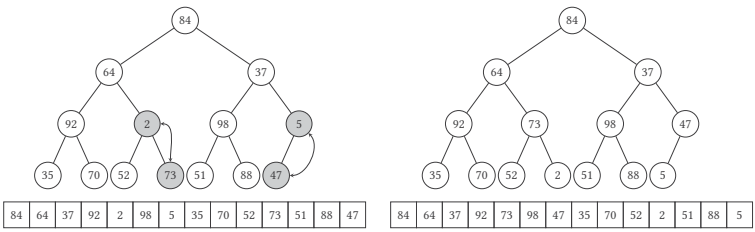
корень больше всех остальных элементов дерева, которое представляет собой наше исходное условие, что $A[0] \geq A[i]$ для всех $i < |A|$. Более того, данная структура, в которой узел как минимум равен детям, является кучей, а точнее максимальной кучей. Вопрос в том, как нам упорядочить A подобным образом? Иными словами, как нам преобразовать A в максимальную кучу? Мы уже встречались с кучами, в сжатии Хаффмана. Хотя здесь задействован несколько иной механизм, вы можете перечитать раздел 3.3 для других приложений и реализаций. Та же самая структура данных много где применяется — чем больше, тем лучше — и имеет множество различных вариантов реализации.

Если последний уровень дерева — это уровень h, мы начинаем с узлов уровня $h - 1$, перебирая их справа налево. Мы сравниваем каждый узел с его детьми. Если он больше их, тогда все в порядке. Если же нет, мы меняем его местами с наибольшим из его детей. Когда мы заканчиваем с уровнем $h - 1$, то знаем, что для всех узлов данного уровня у нас $A[i] \geq A[2i + 1]$ и $A[i] \geq A[2i + 2]$. Это можно видеть на рисунке 12.6(a), где мы проработали второй из трех уровней. Слева у нас исходное дерево, как на рисунке 12.5. Мы помечаем серым цветом узлы, которые поменяли местами. По причине, которая станет очевидной чуть позже, мы снова проходимся по узлам того же уровня, справа налево; сначала узел 5, затем узел 98, с которым мы ничего не делаем, потом узел 2 и, наконец, узел 92, который мы тоже не трогаем. Дерево, которое мы получили в итоге, изображено справа. Под каждым деревом находится массив A, в котором происходят настоящие перестановки. Вы можете убедиться, что на дереве отражаются все изменения, происходящие в массиве (так как на самом деле существует только массив, а дерево — лишь наглядное пособие).

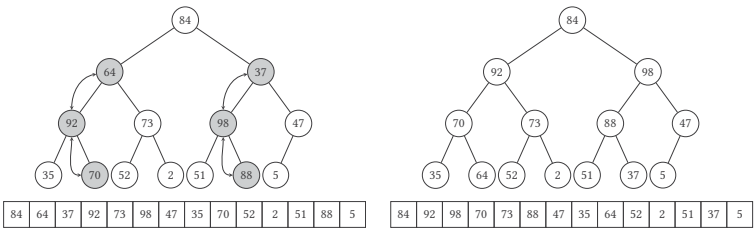
Мы проделываем то же самое с уровнем $h - 2$, однако в этот раз, если мы совершаем замену, нужно проверить, что происходит уровнем ниже: тот, по которому мы только что прошлись. Слева на рисунке 12.6(b) продублировано правое изображение рисунка 12.6(a), чтобы было ясно, откуда мы продолжаем работу. Нам нужно поменять местами узел 37 и узел 98. Если мы их меняем, то узнаем, что у 37 есть дочерние узлы, 51 и 88, которые не отвечают

нашему условию. Мы можем это поправить, поступив, как в прошлый раз: найдем наибольший из 51 и 88 и поменяем его местами с 37. Те же изменения происходят с узлом 64, который опускается к узлу 70. Когда мы покончили со вторым уровнем, наше дерево выглядит как то, что изображено справа на рисунке 12.6(b).

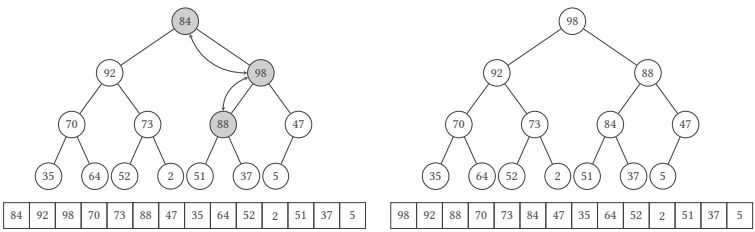
Наконец мы добрались до корневого уровня. Нам надо поменять местами 84 и 98. После этого нам надо поменять 84 и 88, и все. Изменения отображены на рисунке 12.6(c). Пожалуйста, проследите всю цепочку изменений (если еще не обращались к рисункам) и посмотрите, что происходит, когда мы берем рисунок 12.5 и прорабатываем его, как на рисунке 12.6.



(a) Создание максимальной кучи на втором уровне



(b) Создание максимальной кучи на уровне 1



(c) Создание максимальной кучи на корневом уровне

Рисунок 12.6. Создание максимальной кучи

Так как вы проследили, каким образом узлы перемещаются по конструкции максимальной кучи, мы можем сказать об узлах, что кажется, будто со своего прежнего места они погрузились на нужный уровень. Они опустились на уровень ниже, в то время как узлы, чье место они заняли, поднялись наверх и заняли полагающиеся им места. Алгоритм 12.4 как раз и является алгоритмом для данной процедуры погружения.

Алгоритм 12.4. Погружение

`Sink`(A, i, n)

Вводные данные: A, массив с элементами; i, индекс элемента, который нужно погрузить на место; n, число элементов, с которыми имеем дело.

Результат: A с элементом $A[i]$ погрузился в нужное место.

```
1  k = i
2  placed ← FALSE
3  j ← 2k + 1
4  while not placed and j < n do
5      if j < n − 1 and Compare(A[j], A[j + 1]) < 0 then
6          j ← j + 1
7      if Compare(A[k], A[j]) ≥ 0 then
8          placed = TRUE
9      else
10         Swap(A[k], A[j])
11     k ← j
12     j ← 2k + 1
```

Алгоритм берет в качестве вводных данных массив A и индекс i элемента, который мы хотим погрузить и разместить в верном месте. В качестве вводных данных алгоритм также берет число элементов, n, с которыми мы будем иметь дело в ходе алгоритма. У вас может возникнуть вопрос: зачем оно нам, ведь число этих элементов должно быть равно числу элементов в массиве, $|A|$? Затем, что в виде кучи мы используем только часть массива $|A|$. Вспомните, что изначально мы хотим иметь $A[0] \geq A[i]$ для всех $i < |A|$, поэтому весь массив A является кучей. После того, как мы находим максимум, $A[0]$, и меняем его местами с $A[n - 1]$, мы хотим найти максимум из $A[0]$, $A[1]$, …, $A[n - 2]$, так что мы ис-

пользуем первые n – 1 элементов массива A как кучу. Каждый раз, когда мы размещаем элемент в верной позиции, мы приступаем к работе с меньшей кучей.

В алгоритме мы используем k для обозначения индекса элемента. Изначально мы ему присваиваем значение i, но оно изменится, когда элемент опустится вниз дерева. Переменная *placed* будет отмечать, перемещен ли элемент в нужную позицию. Дети элемента с индексом k, если они есть, находятся в позициях $2k + 1$ и $2k + 2$. Мы назначаем j для указания на первое, левое дитя. Затем мы запускаем цикл, который будет повторяться до тех пор, пока узел не займет соответствующую позицию и пока у него есть дети ($j < n$). Сперва нам надо найти наибольшего из двух детей, если у узла их двое. Что мы и проделываем в строчках 5–6. Затем мы проверяем, больше ли узел, который мы погружаем, чем его дети. Если больше, мы закончили, что и прописано в строчках 7–8. Если же он меньше, то мы погружаем его еще ниже и поднимаем на его место наибольшего из его детей, как указано в строчках 9–10. После мы собираемся повторить тот же цикл на уровне ниже и обновляем k, чтобы тот указывал на новую область обитания элемента, который мы погружаем, и j, чтобы тот указывал на область, где мы ожидаем встретить его детей, если таковые имеются.

Самое изящное в алгоритме погружения, что он всего-навсего выстраивает максимальную кучу; к тому же он является краеугольным камнем, который лежит в основе создания нового сортировочного алгоритма, который нам нужен. Сперва мы используем его, как на рисунке 12.6, для превращения наших данных в кучу. На рисунке 12.6(a) показано, что происходит с `Sink`(A, i, $|A|$), для i = 6, 5, 4, 3; на рисунке 12.6(b) показано, что происходит с `Sink`(A, i, $|A|$) для i = 2, 1; и наконец на рисунке 12.6(c) показана ситуация для `Sink`(A, 0, $|A|$). Теперь, получив кучу, мы знаем, что максимальный из всех наших элементов является корнем дерева; он должен находиться в самом конце, когда все элементы будут отсортированы по порядку. Поэтому мы достаем его из корня кучи и меняем местами с последним элементом кучи, $A[n - 1]$: не забывайте, что дерево на самом деле — это замаскированный массив. Первые n – 1 элементов массива уже не куча, потому что

корень не больше своих детей. Но это легко исправить! Нам всего лишь надо запустить алгоритм погружения для первых $n - 1$ элементов, чтобы погрузить только что поменявшийся местами $A[0]$ в надлежащее ему место: `Sink`$(A, 0, |A| - 1)$. По сути мы работаем с кучей, которая уменьшилась на один элемент. Поэтому мы указываем число элементов как вводные данные для алгоритма, ведь нам нужно обозначить, что мы работаем с уменьшенной кучей. Когда алгоритм погружения завершит работу, у нас снова будет максимальная куча, наверху которой будет находиться второй по величине элемент. Мы можем тогда поменять местами $A[0]$ и последний элемент нынешней кучи, элемент $A[n - 2]$. Элементы $A[n - 2]$ и $A[n - 1]$ тогда будут находиться на соответствующих им позициях, позволяя нам продолжить процесс, чтобы переделать оставшиеся $n - 2$ элементов в максимальную кучу с `Sink`$(A, 0, |A| - 2)$. В конце все элементы окажутся отсортированы, начиная с конца массива A и до его начала. Такова сортировка кучей, она описана в алгоритме 12.5.

Алгоритм 12.5. Сортировка кучей

```
HeapSort(A)
    Input: A, a array of items
    Result: A is sorted

1   n ← |A|
2   for i ← ⌊(n − 1)/2⌋ to −1 do
3       Sink(A, i, n)
4   while n > 0 do
5       Swap(A[0], A[n − 1])
6       n ← n − 1
7       Sink(A, 0, n)
```

Сортировка кучей очень лаконична, если у нас есть `Sink`. Изначально, в строчке 1, мы заносим в переменную n общий размер A и приступаем к первой фазе сортировки кучей, в которой, в строчках 2–3, мы превращаем массив в максимальную кучу, чтобы сделать, что нужно для вызова `Sink`$(A, i, |A|)$ для всех i, которым соответствуют внутренние узлы дерева. Последний узел дерева — тот, чье дитя находится в позиции $n - 1$. Сам он

находится в позиции $\lfloor (n-1)/2 \rfloor$. Значения i, которым соответствуют позиции внутренних узлов, соответственно, равны $\lfloor (n-1)/2 \rfloor$, $\lfloor (n-1)/2 \rfloor - 1$, …, 0. Вспомните, что в циклах **for** («для») границы **to** («до») не включены, поэтому, чтобы получить 0, область цикла должна быть **to** («до») –1. Фаза построения кучи в сортировке кучей — именно ее ход отображен на рисунке 12.6. Мы берем значения i в нисходящем порядке, который объясняет, почему на рисунке 12.6 мы проходим по узлам справа налево.

Как только мы создали кучу, мы переходим ко второй фазе сортировки кучей, в которой мы запускаем цикл в строчках 4–7. Мы повторяем цикл количество раз, равное количеству элементов в массиве A. Берем первый элемент, $A[0]$, который является наибольшим из $A[0]$, $A[1]$,…, $A[n-1]$, меняем его местами с $A[n-1]$, уменьшаем n на один и перестраиваем кучу из первых $n-1$ элементов массива A.

В нашем примере с A вторая фаза сортировки кучей начинается так, как показано на рисунке 12.7. Каждый элемент берется с вершины кучи, то есть из первой позиции A, и помещается в позиции $n-1$, $n-2$, …, 1. Каждый раз, когда так происходит, замененный элемент условно кладется на верх кучи, а затем погружается на соответствующее ему в дереве место. После этого наибольший из оставшихся неотсортированных элементов снова находится в первой позиции A, и мы можем повторить всю предыдущую процедуру с уменьшенной кучей. На рисунке мы выделяем серым цветом элементы, которые уже заняли свои конечные позиции, и убираем их из дерева, чтобы подчеркнуть уменьшение кучи с каждым циклом.

Сортировка кучей — простой процесс, но что с его выполнением? Мы начали совсем не так, как с сортировкой выбором, когда мы помещали каждый элемент в соответствующую конечную позицию. Вместо поиска наибольшего из неотсортированных элементов мы прибегли к более хитрой процедуре, с помощью которой мы всегда получаем наибольший неотсортированный элемент наверху максимальной кучи. Оно того стоит?

Да. Сперва подсчитаем замены. Для создания максимальной кучи, когда мы добираемся до корня, нам может понадобиться до

h замен, где *h* — это последний уровень дерева. Для каждого узла первого уровня нам нужно *h* – 1 замен, до предпоследнего уровня, где на каждый узел нам надо по одной замене.

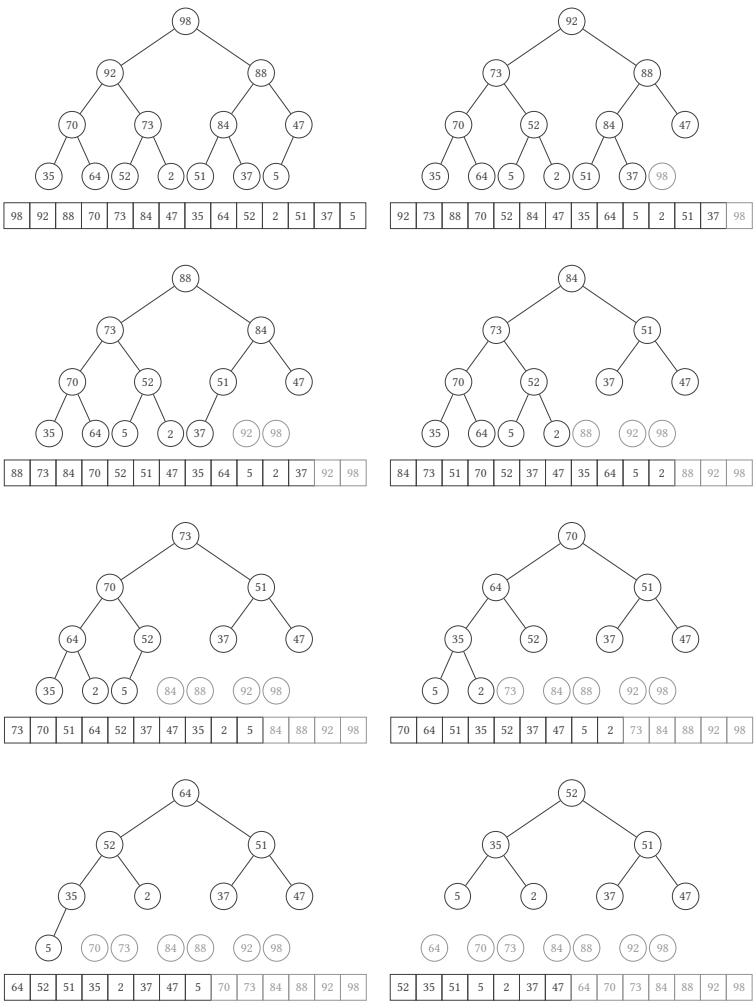


Рисунок 12.7. Вторая фаза сортировки кучей

Так как куча представляет собой полное бинарное дерево, на нулевом уровне (корне) имеется один узел, на первом уровне — $2 = 2^1$ узлов, на втором — $2 \times 2 = 2^2$ узлов и так далее; на пред-

последнем уровне, $(h - 1)$, 2^{h-1} узлов. Все замены складываются как $2^0 \times h + 2^1 \times (h - 1) + 2^2 \times (h - 2) + \ldots + 2^{h-1}$. Такова сумма всех членов $2^k(h - k)$ для $k = 0, 1,\ldots, h - 1$. Она равна $2^{h+1} - h - 2$ Так как h является последним уровнем, то, если n — это число элементов в дереве, мы имеем $n = 2^{h+1} - 1$, поэтому сумма становится $n - \lg n - 1$. Число замен для создания максимальной кучи тогда равно $O(n - \lg n - 1) = O(n)$.

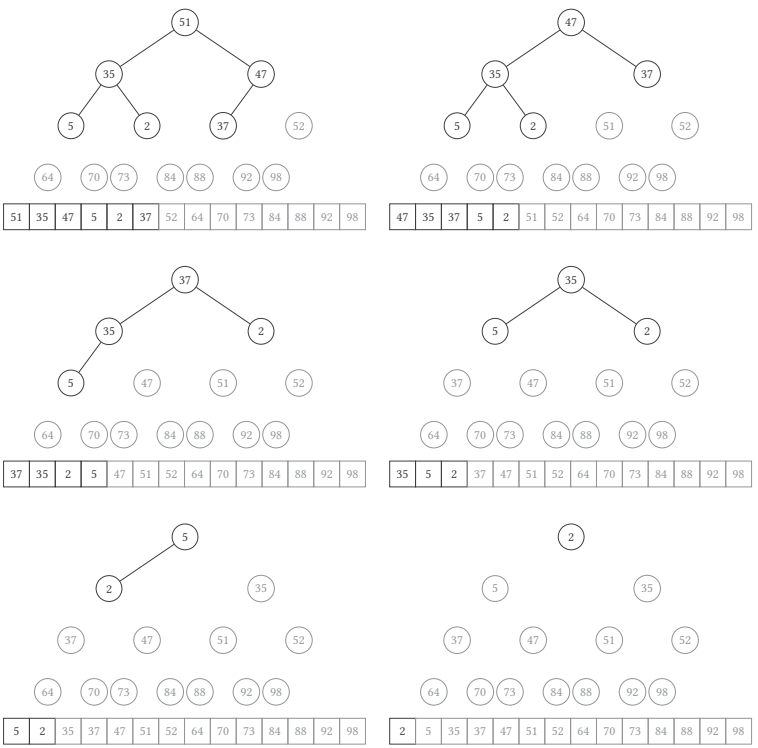


Рисунок 12.8. Вторая фаза сортировки кучей, продолжение

Количество сравнений — вдвое больше: на каждую замену нам надо одно сравнение между детьми и одно сравнение между родителем и наибольшим из детей, поэтому число сравнений для создания максимальной кучи равно $O(2n)$. Во второй фазе сортировки кучей при худшем раскладе приходится иметь дело с кучей с h уровнями, что требует h замен и $2h$ сравнений. Мы имеем

$h = \lfloor \lg n \rfloor$. Вторая фаза выполняется n раз; следовательно, у нас не может быть более $n\lfloor \lg n \rfloor$ замен и $2n\lfloor \lg n \rfloor$ сравнений. То есть $O(n \lg n)$ и $O(2n \lg n)$ соответственно.

Суммируя вышесказанное, мы получаем, что, если у нас n элементов, нам нужно менее $2n \lg n + 2n$ сравнений и $n \lg n + n$ замен, чтобы преобразовать их с помощью сортировки кучей. Что касается сложности, у нас $O(2n \lg n + 2n) = O(n \lg n)$ сравнений и $O(n \lg n + n) = O(n \lg n)$ замен.

Изменение от $\Theta(n^2)$ к $O(n \lg n)$ — нешуточное усовершенствование. Если у вас 1 000 000 объектов для сортировки, методам вставок и выбора потребуется $(10^6)^2 = 10^{12}$ сравнений; один триллион из них. Сортировке кучей же понадобится $10^6 \lg 10^6$, что меньше 20 миллионов. Мы смогли перешагнуть из мира астрономических чисел в мир разумных возможностей.

Сортировка кучей представляет собой алгоритм, который может работать — и прекрасно работает — с большими базами данных, особенно потому, что ей не требуется дополнительного места для работы: она оперирует только имеющимися данными. Сортировка кучей показывает, насколько много времени мы можем выгадать, если воспользуемся наиболее подходящей структурой данных.

12.4. Сортировка слиянием

Теперь представьте, что у вас два набора отсортированных элементов. Как получить единый набор отсортированных элементов? Можно связать их в один отсортированный набор и затем отсортировать. Если мы сортируем их методом вставок или методом выбора, то мы ограничены размерами набора данных. Более того, мы упустим преимущество, вытекающее из факта, что оба набора элементов уже отсортированы. А ведь наверняка есть способ воспользоваться им.

Конечно есть. Вернемся к карточным играм и представим, что у нас две стопки неотсортированных карт. Чтобы получить одну стопку отсортированных карт, нужно проделать следующее. Берете по первой карте из каждой стопки, изучаете, затем кладете

карту с наименьшим значением в третью, пока еще пустую стопку. После берете по еще одной карте из первых двух стопок; одна из них осталась неизменной, в то время как в другой стало на одну карту меньше, так как мы переложили карту в третью стопку. Карта с наименьшим значением опять отправляется в третью стопку, поверх уже лежащей там карты. Продолжайте в том же духе, пока в обеих стопках не закончатся карты. Если одна стопка опустела раньше другой, значит, значения всех карт из оставшейся стопки больше значений карт из третьей стопки, поэтому вы просто кладете их поверх карт третьей стопки. На рисунке 12.9 изображен пример объединения, или слияния, двух отсортированных карточных стопок с помощью данного метода.

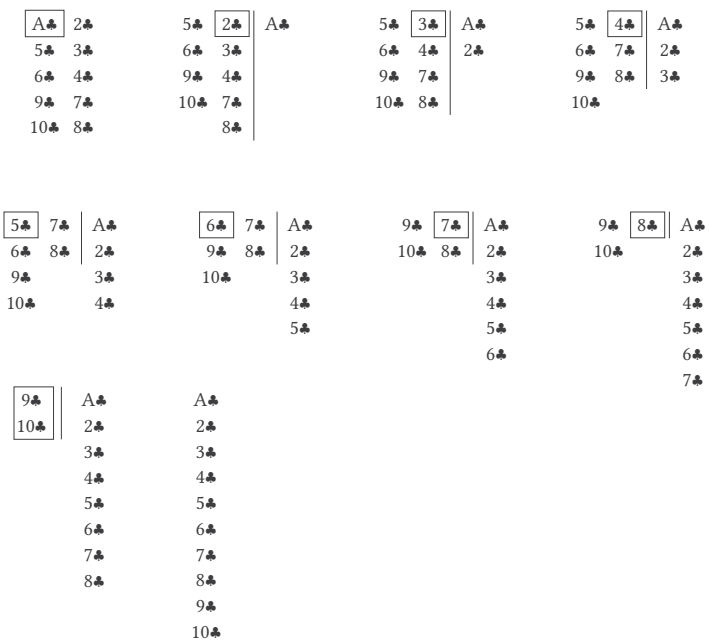


Рисунок 12.9. Слияние двух стопок отсортированных карт

Вернемся к массиву с элементами. Возьмите массив *A*, который мы использовали в наших примерах и разделите его на две части. Затем отсортируйте обе части с помощью какого-нибудь метода (сейчас совсем не важно, каким именно). У вас получится

два отсортированных массива, как нам и надо. Затем, чтобы получить единый массив, проделайте следующее. Берете первый элемент из первого отсортированного массива и первый элемент из второго отсортированного массива. Наименьший из них станет первым элементом нового отсортированного массива. Вы забираете наименьший элемент из каждого массива и перекладываете его в новый массив. Далее повторяете ту же процедуру до тех пор, пока не закончатся элементы в отсортированных массивах. Если один из массивов опустеет раньше другого, то просто добавляете оставшиеся элементы в конец нового массива. На рисунке 12.10 изображен ход процесса для первых четырех элементов двух массивов. Процесс слияния лежит в основе эффективного сортировочного метода, который называется сортировкой слиянием.

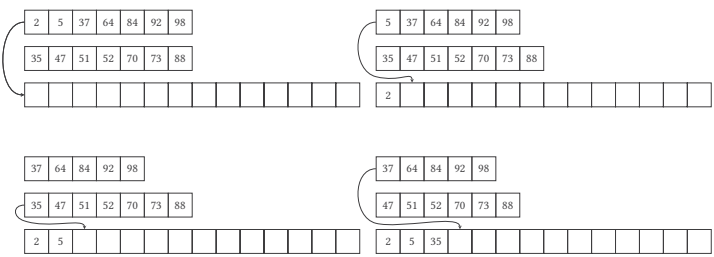


Рисунок 12.10. Слияние двух массивов

Прежде чем мы продолжим и рассмотрим саму сортировку слиянием, отметим пару моментов, касающихся слияния. Рисунок 12.10 показывает нам принцип работы данной сортировки, но на практике слияние происходит вовсе не так. Дело в том, что лучше избежать изъятия элементов из массива, так как это совсем неэффективно и довольно сложно. Лучше как есть оставить элементы в двух исходных массивах и просто отслеживать наше перемещение по массиву. Нам помогут два указателя, которые во время нашего передвижения будут отмечать начало оставшейся части каждого массива. Такое усовершенствование переносит нас от рисунка 12.10 к рисунку 12.11.

Теперь то, что происходит на рисунке 12.11, мы обратим в алгоритм 12.6. У нас в качестве вводных данных есть два массива, *A*

и B, а в качестве результата — массив C, чья длина $|C|$ равна сумме длин A и B: $|C| = |A| + |B|$. Мы используем указатель i, который проходит через массив A, указатель j, который проходит через массив B, и указатель k, который проходит через массив C.

В строчке 1 мы создаем итоговый массив C; в строчках 2 и 3 устанавливаем два указателя i и j на 0. Затем запускаем цикл строчек 4–16, который выполняется по разу для каждого элемента из A и B. Строчки 5–7 отвечают за то, что происходит, когда мы переложили все элементы из A в C. В данном случае все оставшиеся элементы B добавляются в конец C. Строчки 8–10 отвечают за обратный случай, когда мы переложили все элементы из B в C. Далее мы добавляем все оставшиеся элементы A в конец C. Если и в A, и в B еще есть элементы, мы проверяем, какой из нынешних двух элементов, $A[i]$ и $B[j]$, меньше или равен другому, и помещаем его в конец C. Если $A[i]$ меньше, то он отправляется в C, в строках 11–14; в противном же случае в $B[j]$ отправляется C, в строчках 14–16. Обратите внимание, что в каждой итерации, кроме k, передвигаем i или j, в зависимости от того, отправили ли мы A или B в C.

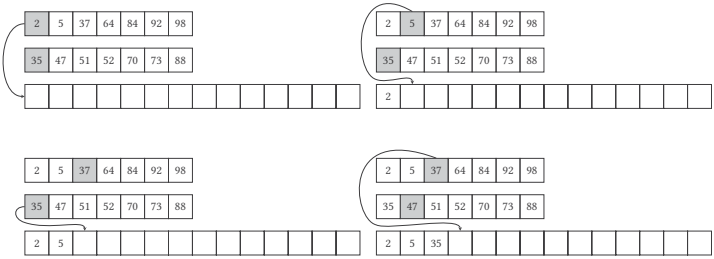


Рисунок 12.11. Слияние двух массивов с помощью указателей

До нынешнего момента мы полагали, что у нас есть два отсортированных массива и нам нужно сделать один общий. Давайте уберем условие, что у нас два отсортированных массива, и представим один массив, состоящий из двух отсортированных частей. Элементы массива от начала и до некой позиции m – 1 отсортированы; и элементы массива, начиная с позиции m и до самого конца, также отсортированы. Таким образом, вместо двух массивов на рисунках 12.10 и 12.11 мы получаем общий массив, состоящий

из двух массивов, которые идут друг за другом. Можем ли мы их соединить так же, как в алгоритме 12.6? Можем, но нам понадобится небольшое изменение.

Алгоритм 12.6. Слияние массивов

```
ArrayMerge(A, B) → C
    Вводные данные: A, отсортированный массив элементов; B, отсортированный массив элементов.
    Выводимые данные: C, отсортированный массив, содержащий элементы A и B.
1   C ← CreateArray(|A| + |B|)
2   i ← 0
3   j ← 0
4   for k ← 0 to |A| + |B| do
5       if i ≥ |A| then
6           C[k] ← B[j]
7           j ← j + 1
8       else if j ≥ |B| then
9           C[k] ← A[i]
10          i ← i + 1
11      else if Compare(A[i], B[j]) ≤ 0 then
12          C[k] ← A[i]
13          i ← i + 1
14      else
15          C[k] ← B[j]
16          j ← j + 1
17  return C
```

Во-первых, нам нужен временный массив, который мы будем использовать как вспомогательное место. Мы скопируем в него элементы наших двух отсортированных массивов. Затем мы выполним все так, как в алгоритме 12.6, только вместо копирования из двух различных массивов в новый, итоговый массив, мы будем копировать из временного массива в исходный, существующий массив. Так как мы собираемся вносить изменения непосредственно в наш исходный массив, то такой подход мы называем обменным слиянием массивов, вы можете видеть его в алгоритме 12.7.

Алгоритм 12.7 чуть более общий, он не требует, чтобы весь массив состоял из двух отсортированных частей. Ему нужно

лишь, чтобы в нем содержались две отсортированные части, одна за другой. Перед первой частью и после второй части могут находиться и другие элементы. Вскоре мы увидим, какая от этого польза. В качестве вводных данных алгоритм берет массив A, который надо отсортировать, и три индекса l, m и h. Индексы показывают, что элементы $A[l]$, …, $A[m]$ отсортированы и элементы $A[l + 1]$, …, $A[h]$ тоже отсортированы. Если две отсортированные части занимают весь массив, мы имеем $l = 0$ и $h = |A| - 1$.

Алгоритм 12.7. Обменное слияние массивов

```
ArrayMergeInPlace(A, l, m, h)
```

Вводные данные: A, массив элементов; l, m, h, такие индексы массива, что элементы $A[l]$, …, $A[m]$ и $A[m + 1]$, …, $A[h]$ отсортированы.

Результат: A с элементами $A[l]$, …, $A[h]$ отсортирован.

```
1  C ← CreateArray(h − l + 1)
2  for k ← l to h + 1 do
3      C[k − l] = A[k]
4  i ← 0
5  cm ← m − l + 1
6  ch ← h − l + 1
7  j ← cm
8  for k ← l to h + 1 do
9      if i ≥ cm then
10         A[k] ← C[j]
11         j ← j + 1
12     else if j ≥ ch then
13         A[k] ← C[i]
14         i ← i + 1
15     else if Compare(C[i], C[j]) ≤ 0 then
16         A[k] ← C[i]
17         i ← i + 1
18     else
19         A[k] ← C[j]
20         j ← j + 1
```

В обменном слиянии опять используются два указателя, только в этот раз они указывают не на два разных массива, а на две

разные области одного и того же массива. Нам нужен вспомогательный массив *C*, который мы создаем в строчке 1 и в строчках 2–3 заполняем элементами двух отсортированных частей *A*. Так как две отсортированные части находятся в областях *l*, *l* + 1, …, *h*, то, чтобы скопировать элементы, нам надо пройти цикл *h* – *l* раз: |*C*| = *h* – *l*. Мы снова используем два указателя, *i* и *j*, которые проходят через две отсортированные части, но в этот раз они проходят через скопированные элементы в массиве *C*. Первая отсортированная часть содержит *m* – *l* элементов, а вторая — *h* – *l* элементов. Следовательно, вторая часть начинается с позиции *cm* = *m* – *l* + 1 в *C* и заканчивается в *ch* = *h* – *l* + 1 в *C*. Указатель *i* начнет движение с позиции 0, а указатель *j* — с позиции *cm*. Мы указываем эти границы в строчках 4–7. После мы переходим к строчкам 8–10 алгоритма 12.7, которые по сути ничем не отличаются от строчек 4–16 алгоритма 12.6.

Во время выполнения алгоритма, после того, как мы скопировали элементы, которые сольются в массиве *C*, мы имеем дело с двумя отсортированными частями массива *C* и сливаем их, находя каждый раз наименьший элемент и записывая его в массив *A*. То же самое мы делали прежде, в алгоритме 12.6. По завершение алгоритма в массиве *A* будут содержаться объединенные элементы из двух отсортированных частей.

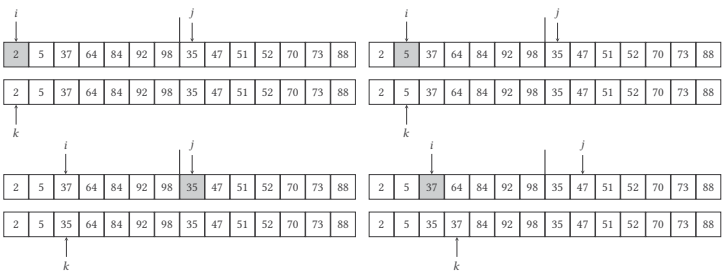


Рисунок 12.12. Обменное слияние

На рисунке 12.12 показано, что происходит в массиве, разделенном на две части, то есть почему *l* = 0 и *h* = |*A*| – 1. Мы отмечаем начало второй отсортированной части вертикальной чертой. Перед вами четыре пары массивов; верхний в каждой паре — это

массив C, а нижний — массив A. Изначально C является копией A — он бы копировал часть A, если бы $l > 0$ или $h < |A| - 1$. По ходу работы алгоритма элементы копируются из C в A, и массив A становится отсортированным.

Теперь чуть отстранимся и подсчитаем, чего же мы добились: если нам дать два отсортированных массива или две отсортированные части массива, мы можем слить их в один общий массив. В целом получается, что, если нам дадут отсортированный ряд, то мы можем разделить его на две части. Сортируем первую и вторую части, после чего получаем две отсортированные части, которые, как мы видели, объединяются путем слияния.

Похоже на фокус, ведь мы начали с условием, что сортируем одну последовательность, но увернулись от данной задачки и представили, что мы можем отсортировать последовательность двумя частями. Как же мы сортируем каждую половину? Мы проделываем все то же: разбиваем надвое, сортируем каждую половину, а затем сливаем их. Кажется, что мы забросили основную работу, так как вместо того, чтобы сортировать на месте, мы предполагаем, что каким-то образом — хотя мы даже не знаем, каким образом отсортировать все элементы, — половина из них будет отсортирована и объединена. Напоминает стишок о блохе:

> Под микроскопом он открыл, что на блохе
> Живет блоху кусающая блошка.
> На блошке той — блошинка-крошка.
> В блошинку же вонзает зуб сердито
> Блошиночка. И так ad infinitum.*

Нам же регрессия не грозит, так как мы не можем делить ряд надвое до бесконечности. В определенный момент, когда мы будем делить, мы получим последовательность из одного элемента, а такую последовательность можно назвать отсортированной: единичный элемент относительно самого себя всегда отсортирован. Это и есть ключ к решению нашей задачи. В ходе повторяющегося деления мы рано или поздно получаем последовательности из

* Стихотворение в переводе М. Лозинского.

одного элемента. Они уже отсортированы, и мы можем их тут же объединить: объединить — значит взять два элемента и расставить их по порядку. Заем мы можем взять последовательности побольше, которые образуются в результате слияния двух маленьких, и объединить их, и так далее, пока не получим две отсортированные последовательности, которые сложатся в одну общую.

Например, как нам отсортировать руку с картами A♡ 10♡ K♡ J♡ Q♡? Сперва мы разбиваем руку на две части: A♡ 10♡ K♡ и J♡ Q♡. Части не отсортированы, поэтому мы не можем их объединить. Берем первую часть и снова делим: A♡ 10♡ и K♡. В этот раз вторая часть, K♡, уже отсортирована, так как в ней одна карта, но первая часть все еще в случайном порядке, поэтому мы разбиваем ее на A♡ и 10♡. В каждой части по одной карте, значит, они отсортированы, поэтому мы сливаем их и получаем 10♡ A♡. Теперь мы сливаем их с K♡ и получаем 10♡ K♡ A♡. Возвращаемся к J♡ Q♡. Разделяем их надвое и получаем J♡ и Q♡. Каждая часть отсортирована, поэтому мы объединяем их и получаем J♡ Q♡. Теперь у нас две отсортированные части: 10♡ K♡ A♡ и J♡ Q♡. Мы сливаем их, чтобы получить конечную отсортированную последовательность, то есть руку: 10♡ J♡ Q♡ K♡ A♡. Весь процесс представлен на рисунке 12.13.

Как вы можете заметить, когда мы разбиваем J♡ Q♡, мы лишь тратим время, так как они уже отсортированы. Все верно; тем не менее мы соглашаемся с таким подходом, потому что проверка массива, отсортирован ли он, требует прохождения всего массива и проверки того, что каждый последующий элемент больше или равен предыдущему. Возможно, с J♡ Q♡ все очевидно, но если мы имеем дело с тысячами элементов, то лучше лишний раз перестраховаться.

Теперь коснемся описания алгоритма, который реализует данный процесс в алгоритме 12.8. Задумка деления последовательности до тех пор, пока не останется один элемент, воплощается с помощью рекурсии. `MergeSort`(A, l , h) берет массив A, чьи элементы $A[l]$, …, $A[h]$ мы хотим отсортировать. Если $l \geq h$, то A содержит только один элемент и нам не надо ничего делать. В противном случае в строчке 2 мы вычисляем середину и вызываем `MergeSort` для каждой из двух частей, то есть `MergeSort`(A, m, h)

и `MergeSort`(A, $m + 1$, h). В ходе вызова произойдут две одинаковые вещи: проверка, нужна ли сортировка; если нет, то случится разбивка на две части, которые подвергнутся тому же процессу. Так будет повторяться, пока мы не доберемся до сегмента с одним элементом. Когда это случится, мы начнем сливать их в сегменты побольше, пока не восстановим весь массив A. Чтобы все заработало, мы вызываем `MergeSort`(A, 0, $|A| - 1$).

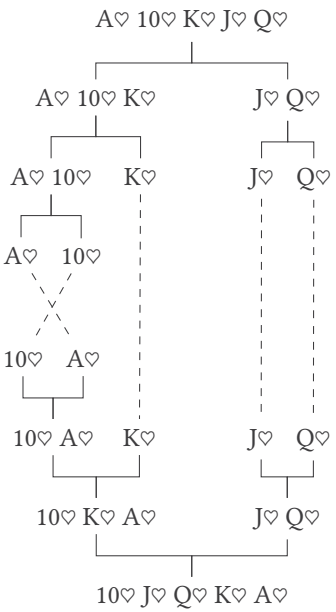


Рисунок 12.13. Сортировка руки с помощью метода слияния

Чтобы понять, что происходит, взгляните на рисунок 12.14. Рисунок представляет собой пример трассировки вызова, так как на нем отслеживаются вызовы функций, которые встречаются по ходу выполнения программы. Каждая функция, вызванная из другой функции, структурирована в соответствии с вызывающей функцией; мы также связываем их, чтобы было легче ориентироваться в рисунке. Как видите, исходный массив [84, 64, 37, 92, 2, 98, 5, 35, 70, 52, 73, 51, 88, 47] делится на все меньшие и меньшие куски до тех пор, пока мы не сможем приступить к слиянию, начиная с массивов, в которых один элемент, и переходя ко все

большим и большим массивам, пока в итоге не получим один общий массив.

Теперь вы можете понять, почему в вводные данные алгоритма 12.7 мы включили *l*, *m* и *h*: только на последнем этапе сортировки слиянием, когда мы объединяем две части в общий массив, при этом имея $l = 0$ и $h = 0$. До того мы сливаем меньшие части и нам надо точно знать, где они начинаются и где заканчиваются.

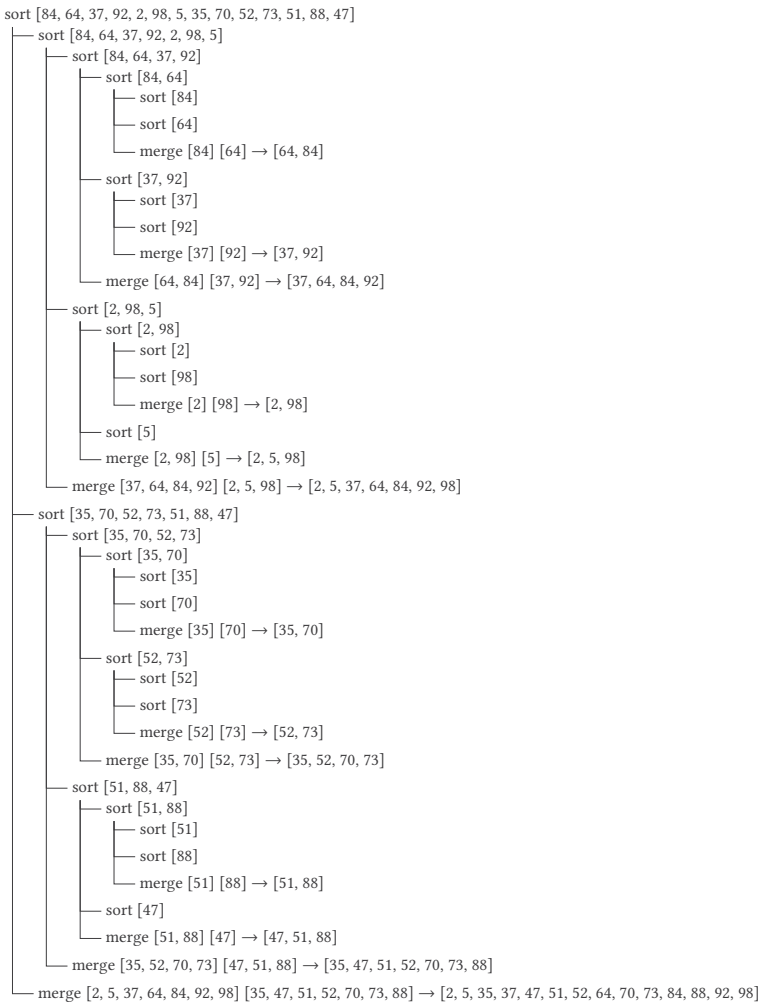


Рисунок 12.14. Трассировка вызова сортировки методом слияния

Сортировка методом слияния — это изящное использование принципа «разделяй и властвуй» в решении задач, так как она сортирует набор элементов путем деления на два и упорядочиванием двух получившихся половин. Помимо эстетики нам, конечно же, следует обратить внимание на исполнение, ведь нам нужно не только изящное, но и практичное решение. Так какова же сложность сортировки слиянием?

Алгоритм 12.8. Сортировка слиянием

`MergeSort`(A, l, h)

Вводные данные: A, массив элементов; l, h, индексы массива.

Результат: A с элементами $A[l]$, …, $A[h]$ отсортирован.

```
1  if l < h then
2      m = l + ⌊(h − l)/2⌋
3      MergeSort(A, l, m)
4      MergeSort(A, m + 1, h)
5      ArrayMergeInPlace(A, l, m, h)
```

Чтобы проанализировать сложность данного алгоритма, нам надо принять во внимание каждый рекурсивный этап, когда мы разделяем массив надвое. Когда мы делим массив пополам, мы по сути создаем бинарное дерево: неразделенный массив — это корень, а две его части — дети. Дерево можно увидеть на рисунке 12.14; оно состоит из сортировочных узлов и растет слева направо и сверху вниз. У каждого узла двое детей, кроме листьев. На рисунке 12.15 изображено то же дерево, но в более привычном формате: каждый узел отвечает за массив, который нужно отсортировать, то есть за вызов `MergeSort`(A, l , h).

На каждом уровне дерева нам нужно сравнить самое большее n элементов. В наихудшем случае при каждом слиянии, происходящем на уровне, придется сравнить — в строчке 15 алгоритма 12.7 — все элементы двух частей, которые нужно объединить. В целом, как мы уже говорили, одна из двух частей, которые надо объединить, может закончиться раньше другой в ходе слияния, в таком случае мы просто копируем оставшиеся элементы, не

сравнивая их. Впрочем, давайте разберем наихудший сценарий, так как анализ того, что может случиться в среднем, куда более сложен.

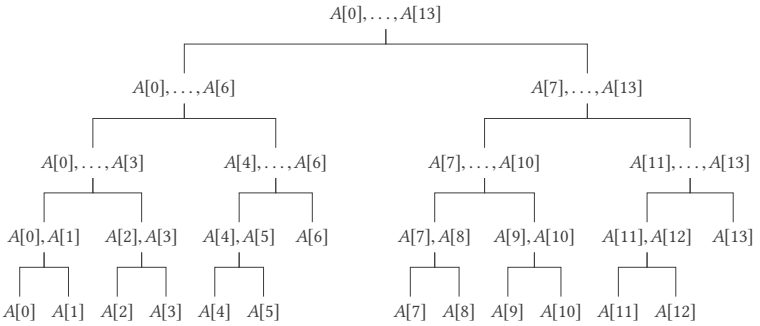


Рисунок 12.15. Дерево сортировки слиянием

В таком случае нам надо найти количество уровней в дереве. На самом верху располагается наш массив A с n элементами. На каждом уровне мы разбиваем массив, если возможно, на две части. Если n является степенью двух, то процесс не может повторяться более $\lg n$ раз, потому что мы получаем массивы с одним элементом и нам нужно $\lg n$ уровней. Если n не является степенью двух, то оно находится между двумя степенями числа 2: $2^k < n < 2^{k+1}$. Следовательно, количество уровней в таком дереве будет больше, чем k, и не более $k + 1$: то есть равно $\lceil \lg n \rceil$. В целом получается, что, когда n — это степень двух, у нашего дерева будет до $\lceil \lg n \rceil$ уровней. На каждом уровне все элементы A копируются дважды, из различных частей A в их буфера хранения, различные массивы C из строчек 1–3 алгоритма 12.7; так что у нас 2^n копий. При наихудшем раскладе у нас n сравнений, если нам надо сравнивать все элементы каждой пары на уровне дерева в строчках 8–20 алгоритма 12.7. Тогда, говоря о сложности, мы получаем $O(2n\lceil \lg n \rceil + 2n) = O(n \lg n)$ копий и $O(n\lceil \lg n \rceil + n) = O(n \lg n)$ сравнений. Чтобы уйти от $O(n\lceil \lg n \rceil + n)$ к $O(n \lg n)$, мы используем факт, что $n\lceil \lg n \rceil + n \leq n(\lg n + 1)$ $n = n \lg n + 2n$.

Чтобы еще улучшить работу сортировки, мы можем извлечь пользу из массивов или их частей, которые уже отсортированы между собой. Представьте более общий случай, когда мы разбиваем массив на две части, сортируем их независимо друг от дру-

га, а затем обнаруживаем, что все элементы первой части меньше или равны элементам второй части. Тогда нам не надо сливать две части; нам лишь нужно присоединить вторую часть к первой. Это легко осуществимо на практике. После вызовов `MergeSort` в алгоритме 12.8 мы проверяем $A[m]$, меньше ли он или равен $A[m + 1]$. Если меньше или равен, то слияние не нужно.

Сортировка слиянием — это сортировочный алгоритм, который применяется на практике и реализуется в стандартных библиотеках программных языков. Главный ее недостаток в том, что ей требуется много места: как мы видели, нам нужно копировать каждое слияние массива A со вспомогательным массивом C. Таким образом, сортировке методом слияния требуется дополнительное место размером n для сортировки массива с n элементами, что может оказаться критичным ограничением, если мы имеем дело с большими массивами.

12.5. Быстрая сортировка

Применение принципа «разделяй и властвуй» в сортировке подводит нас к одному значимому в информатике сортировочному алгоритму, который знаком каждому программисту, алгоритму, который мы подразумеваем, когда говорим о сортировке, без всяких мыслей об алгоритмах. Этот алгоритм называется быстрой сортировкой, ее разработал Ч. Э. Р. (Тони) Хоар в 1961 году.

Логика быстрой сортировки проста. Мы берем из набора элементов, которые хотим отсортировать, один элемент и собираемся поместить его в конечную позицию: позицию, которую он будет занимать, в уже отсортированном массиве. Как мы ее находим? Данный элемент будет находится в ней, когда все элементы, которые меньше него, будут располагаться перед ним, а все элементы, которые больше него, будут располагаться за ним. Так что мы перемещаем элементы, пока не выстраиваем их согласно этой задумке. Если у нас есть массив A, который нам надо отсортировать, представьте, что мы взялись за дело и ставим элемент $A[p]$, для некоего p, в конечную позицию. Затем элементы $A[0]$, $A[1]$, …,

$A[p-1]$ и $A[p+1], A[p+2], \ldots, A[n-1]$ где $n = |A|$, нужно поместить в их конечные позиции. Мы проделываем ту же процедуру для двух частей $A[0], A[1], \ldots, A[p-1]$ и $A[p+1], A[p+2], \ldots, A[n-1]$, что поставит еще два элемента в их конечные позиции и даст нам возможность продолжить процесс в двух частях, созданных для каждого из $A[0], A[1], \ldots, A[p-1]$ и $A[p+1], A[p+2], \ldots, A[n-1]$.

Представьте, что перед вами группа детей и вам надо построить их в шеренгу по росту. Быстрая сортировка предполагает, что вы выберете одного ребенка, скажем, Джейн, и скажете остальным ребятам: «Все, кто ниже Джейн, встают перед ней. Все, кто выше Джейн, встают позади нее». Затем вы идете к ребятам, стоящим перед Джейн, выбираете из них одного и ставите то же условие, что и с Джейн. Потом идете к ребятам, что стоят позади Джейн, и проделываете все то же самое. Вы повторяете процесс до тех пор, пока каждый в шеренге не займет соответствующую его росту позицию.

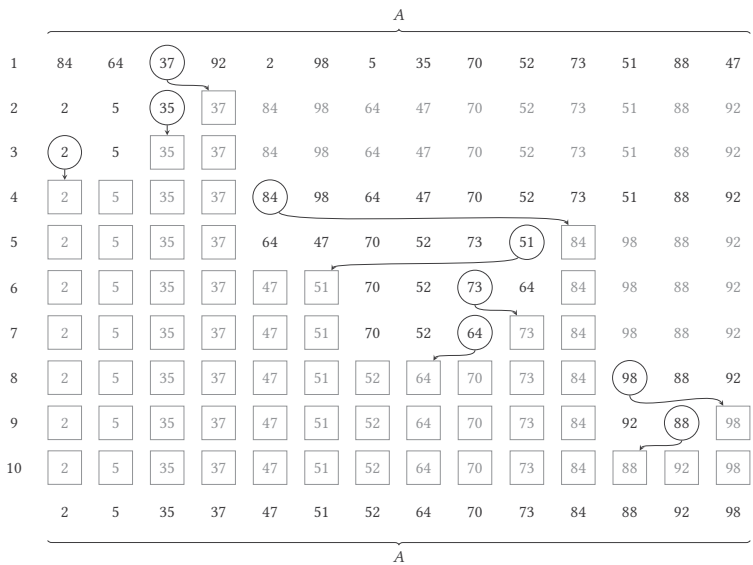


Рисунок 12.16. Пример быстрой сортировки

Процесс отображен на рисунке 12.6. Сначала мы берем число 37. Во втором ряду мы переставили все элементы меньше 37 слева

от него, а все элементы больше 37 — справа. Число 37 занимает конечную позицию, которую мы отмечаем квадратом. В каждом ряду мы переставляем элементы — те, что ниже или выше элемента, который находится в конечной позиции. Мы выделяем серым цветом оставшиеся элементы и закрашиваем черным те, которые переставляем в ряду: элементы, из которых, если вернуться к примеру с детьми, мы выбираем одного и просим остальных построиться относительно него.

Так что мы переходим к элементам 2, 5 и 35. Берем число 35; оно, на самом деле, и так уже в нужной позиции, потому что оно больше 3 и 5. Затем нам нужно разобраться с 2 и 5, которые тоже не нуждаются в перестановке. В четвертом ряду мы переходим к оставшимся элементам: тем, что больше 37. Здесь мы будем перемещать элементы от 82 и до конца, до 92. Берем 84, перемещаем элементы и продолжаем в том же духе начиная с пятого ряда, пока все элементы не займут нужные позиции.

Алгоритм 12.9. Быстрая сортировка

`Quicksort`(A, l, h)

Вводные данные: A, массив с элементами; l, h, индексы массива.

Результат: A с элементами $A[l]$, …, $A[h]$ отсортирован.

```
1  if l < h then
2      p ← Partition(A, l, h)
3      Quicksort(A, l, p − 1)
4      Quicksort(A, p + 1, h)
```

Процесс может быть изложен в виде очень коротенького алгоритма 12.9. Как и в сортировке слиянием, алгоритм рекурсивный и вообще очень на него похож. В строчке 4 мы разделяем массив A на две части, $A[0]$, $A[1]$, …, $A[p-1]$ и $A[p+1]$, $A[p+2]$, …, $A[n-1]$. Элемент, который находится в позиции p, является разделяющим элементом, и мы зовем его центральным элементом, так как относительно него передвигаются все другие элементы. Элементы $A[0]$, $A[1]$, …, $A[p-1]$ меньше центрального, а элементы $A[p+1]$, $A[p+2]$, …, $A[n-1]$ больше или равны ему. Весь процесс имеет

смысл, пока в сегменте массива есть хотя бы один элемент, как указано в условии строчки 1. Чтобы приступить к сортировке, мы вызываем `Quicksort`(A, 0, $|A| - 1$), как и в сортировке слиянием.

В алгоритме 12.9 есть недостающая часть, определение функции `Partition`, которая и заведует всем волшебством. Существует несколько способов разделить массив на две части, с большими и меньшими, чем указанный, элементами. Один из них вот какой. Сперва мы выбираем центральный элемент. Мы хотим поместить его в конечную, отсортированную позицию, однако пока не знаем, где эта позиция находится, поэтому на время мы меняем центральный элемент местами с элементом, стоящим в самом конце, чтобы убрать его с пути.

Нам надо найти конечную позицию центрального элемента. Она разделит элементы после того, как мы их все переберем; она — граница, отделяющая два набора элементов: те, что меньше центрального, и те, что больше него. Изначально мы ничего не делим, поэтому можем установить значение нашей границы на ноль. Мы перебираем элементы, которые нужно разделить. Когда мы находим элемент, который меньше центрального, мы тут же узнаем две вещи: что этот элемент должен идти до конечной позиции центрального элемента и что конечная позиция центрального элемента будет больше той, что мы установили. Закончив перебор элементов, мы перемещаем центральный элемент в его конечную позицию, установленную в ходе просмотра элементов. Алгоритм 12.10 подробно описывает данную процедуру.

Как и в сортировке методом слияния, алгоритм 12.10 в качестве исходных данных берет массив A и два индекса l и h, разделяет массив на две части, для каждой свой индекс. Если $l = 0$ и $h = |A| - 1$, то он разделит целый массив; в противном случае он разделит элементы $A[l], \ldots, A[h]$. Он нам нужен по той же причине, что и в сортировке слиянием: в ходе рекурсивных вызовов быстрой сортировки нам надо разделить части A, не обязательно весь A. Быстрая сортировка сортирует A на месте, поэтому мы указываем в описании алгоритма, что у него есть и результат, и выводимые данные.

В строчке 1 мы выбираем центральный элемент. В строчке 2 меняем его местами с элементом в конце A и устанавливаем позицию, которая находится между разделенными элементами, в начало области, которую нам надо разделить. Мы обозначаем данную позицию переменной b, так как она отмечает границу между значениям, которые меньше центрального, и значениями, которые больше центрального. В цикле строчек 4–7 мы проходимся по области, которую нужно разделить. Каждый раз, когда мы находим элемент со значением ниже, чем у центрального, мы меняем его местами с элементом, находящимся в данный момент в позиции b, и передвигаем границу b на один элемент. После цикла мы помещаем центральный элемент, который мы ранее убрали в самый конец, в соответствующую, конечную позицию, которая является позицией b, и возвращаем его индекс.

Алгоритм 12.10. Разделение

`Partition`$(A, l, h) \rightarrow b$

Вводные данные: A, массив с элементами; l, h, индексы массива.

Результат: A разделен таким образом, что $A[0], \ldots, A[p-1] < A[p]$ и $A[p+1], \ldots, A[n-1] \geq A[p]$ при $n = |A|$.

Output: b, the index of the nal position of the pivot element

```
1  p ← PickElement(A)
2  Swap(A[p], A[h])
3  b ← l
4  for i ← l to h do
5      if Compare(A[i], A[h]) < 0 then
6          Swap(A[i], A[b])
7          b ← b + 1
8  Swap(A[h], A[b])
9  return b
```

Чтобы лучше понять, что происходит, взгляните на рисунок 12.17, на котором отображен процесс, происходящий между четвертым и пятым рядами рисунка 12.16. Мы имеем $l = 4$ и $h = 13$. Берем элемент 84 и меняем его местами с элементом $A[13] = 92$. Изначально $f = l = 4$; мы отмечаем позицию b — конечную цен-

тральную позицию — прямоугольником. Серым прямоугольником мы отмечаем нынешнее значение *i* в строчках 5–8 алгоритма 12.10. На каждом этапе алгоритма разбиения мы увеличиваем *i* и, следовательно, сдвигаем серый прямоугольник на одну позицию. Как только мы обнаруживаем, что $A[b] < 84$, мы меняем местами значения $A[i]$ и $A[b]$ и увеличиваем значение *b*. Обратите внимание, что конечное значение *i* равно $h - 1$, потому что $A[h]$ содержит центральный элемент, что соответствует значениям, которые может принимать *i* в строчке 5. После завершения цикла мы меняем местами центральный элемент $A[h]$ с элементом $A[b]$, чтобы поставить его в конечную, соответствующую ему позицию.

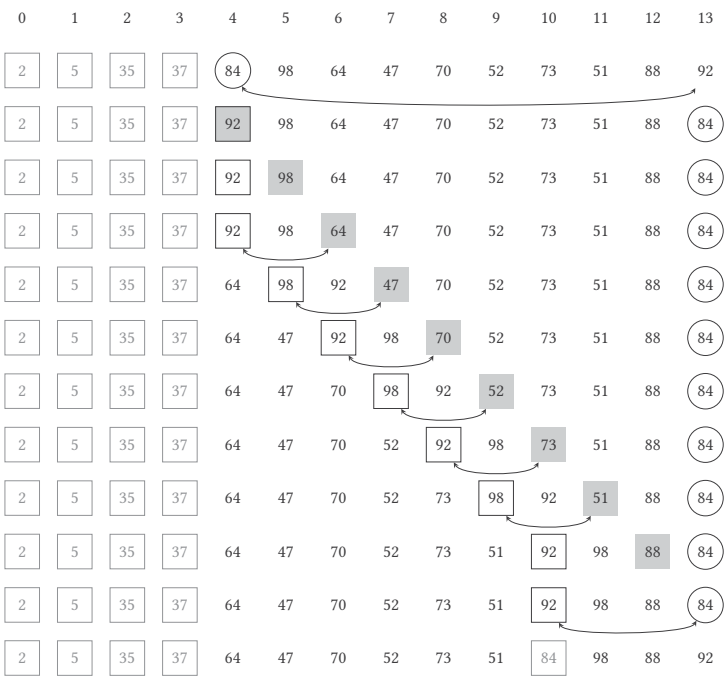


Рисунок 12.17. Пример разделения

Тут мы сталкиваемся с еще одной недостающей частью, которая поможет нам составить общую картину быстрой сортировки, — с функцией `PickElement`. Как видно на рисунке 12.16, в рядах 3 и 8 мы брали первый элемент (в третьем ряду их всего

два на выбор), в рядах 2, 5, 7, 9 брали последний элемент (в девятом ряду тоже выбор из двух), а в остальных рядах мы брали элемент из середины. По сути дела, мы каждый раз брали элементы в случайном порядке: когда из начала, когда из конца, когда еще откуда-то. У вас может возникнуть вопрос, почему же мы выбираем элементы случайно, а не руководствуемся каким-нибудь простым правилом, вроде взятия первого элемента ($A[l]$), или последнего элемента ($A[h]$), или среднего. На то есть своя причина, которая связана с рабочими характеристиками быстрой сортировки.

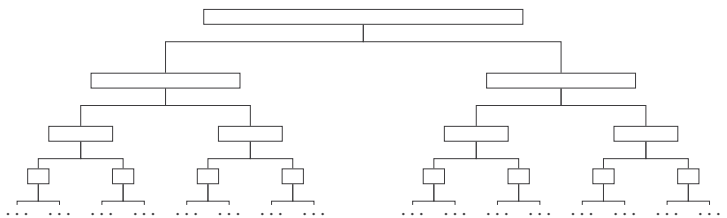


Рисунок 12.18. Оптимальное дерево быстрой сортировки

Быструю сортировку, как и сортировку слиянием, можно изобразить в виде дерева. Когда массив разбивается надвое, мы получаем две рекурсивных системы быстрой сортировки. Представим, что мы всегда берем средний элемент, то есть элемент, который разбивает массив на две части с равным количеством элементов. Так мы получаем дерево, похожее на дерево сортировки слиянием, его можно видеть на рисунке 12.18. Дерево уравновешено, потому что каждый раз, когда мы разделяем массив надвое, мы получаем две равные части (добавляем или забираем элемент, если число элементов, которые надо разбить, является четным).

На первом уровне дерева разбивка работает с n узлами в массиве A, так как каждый узел надо сравнить с центральным. На втором уровне дерева разбивка требует $n - 1$ сравнений, потому что мы разделяем два массива в общей сложности с $n - 1$ элементами; помните, что мы забрали центральный элемент на первом уровне. Так как наше дерево является бинарным, как и в сортировке слиянием, мы имеем $O(\lg n)$ уровней; на каждом уровне

у нас самое большее n сравнений и не более n перестановок, поэтому сложность алгоритма равна $O(n \lg n)$.

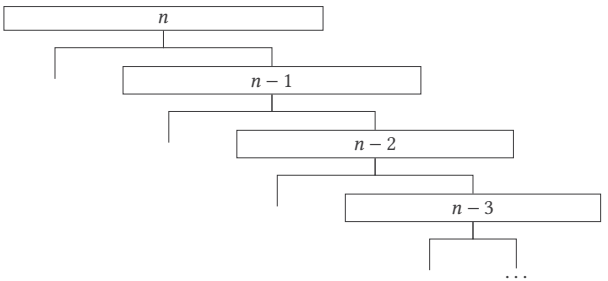


Рисунок 12.19. Худший случай для дерева быстрой сортировки

Теперь представим, что всякий раз центральным элементом мы берем наименьший из всех доступных. Такой подход работает, если только A уже отсортирован и мы перебираем на роль центрального элемента все элементы A от начала и до конца. Если мы поступим так, то полученные в ходе разбивки части окажутся совсем не равномерными. Каждый раз, указывая центральный элемент, мы будем получать две части со следующими характеристиками: одна будет редуцированной, полностью пустой, а вторая будет полна оставшимся элементами. Например, если нам надо разделить элементы [1, 2, 3, 4, 5] и в качестве центрального мы указываем число 1, то мы получаем две части: пустую, [ ], и [2, 3, 4, 5]. Если мы выберем центральным элементом число 2, то снова получим [ ], и [3, 4, 5].

Данную ситуацию можно увидеть на рисунке 12.19. Мы начинаем с разбивки n элементов, затем n – 1, n – 2 … 1 элементов. Таким образом, в этот раз в нашем дереве будет n уровней. На каждом из них — кроме последнего, на котором у нас массив с одним элементом, — с центральным элементом сравниваются все элементы на уровне, поэтому мы проводим $n + (n – 1) + … + 1 = n(n – 1)/2$ сравнительных операций. Следовательно, если массив уже отсортирован в восходящем порядке и мы каждый раз указываем центральным элементом самый первый, наименьший из всех, быстрая сортировка будет иметь сложность $O(n^2)$ вместо $O(n \lg n)$.

Короче говоря, в наилучшем случае сложность быстрой сортировки равна $O(n \lg n)$, а в наихудшем — $O(n^2)$. Лучший сце-

нарий предполагает, что нам всегда удается выбрать наилучший центральный элемент: тот, который разобьет массив ровно напополам. Увы, на деле такое не организуешь, потому что к общей сложности алгоритма тогда прибавятся затраты на поиск подобного элемента, что сделает алгоритм непрактичным.

Выбор случайного элемента уберегает нас от наихудшего сценария. Более того, доказано, что дерево быстрой сортировки, которое получается в ходе случайного выбора центрального элемента, не сильно хуже дерева, полученного при оптимально раскладе. Как ни парадоксально, но случайный выбор элементов гарантирует, что ожидаемая сложность быстрой сортировки останется $O(n \lg n)$.

Обратите внимание на слово «ожидаемая». Быстрая сортировка со случайным выбором центрального элемента приводит нас к элементу случайности в поведении алгоритма. Так как мы берем элементы в случайном порядке, мы не можем гарантировать, что не столкнемся с наихудшим сценарием. Когда же он случается, быстрая сортировка выполняется чудовищно медленно. Тем не менее шанс выпадения наихудшего случая чрезвычайно мал, поэтому мы ожидаем, — хотя и не можем стопроцентно гарантировать, — что быстрая сортировка сработает быстро.

Насколько малы шансы столкнуться с наихудшим раскладом? Если мы указываем центральные элементы в случайном порядке и предполагаем, что у всех элементов разные значения, вероятность выбора наименьшего элемента в первый раз равна $1/n$. Вероятность, что мы выберем наименьший элемент во второй раз равна $1/(n - 1)$. Мы продолжаем расчеты в том же духе, пока не получим массив с двумя элементами и вероятностью 1/2. Таким образом, вероятность, что мы каждый раз будем брать наихудший элемент, следующая:

$$\frac{1}{n} \times \frac{1}{n-1} \times \cdots \times \frac{1}{2} = \frac{1}{1 \times 2 \times \cdots \times n} = \frac{1}{n!}.$$

Значение $1/n!$ действительно меньше; только для десяти элементов мы получаем $1/10! = 1/3628800$, что меньше одного шанса на 3,5 миллиона. Быстрая сортировка является примером рандомизированного алгоритма, поведение которого зависит от

случайности. Такая сортировка очень хороша, потому что в среднем алгоритм работает быстро, а вероятность того, что он наткнется на худший сценарий, настолько мала, что на практике ею запросто можно пренебречь.

В основе быстрой сортировки лежит одно условие, которое есть у всех рандомизированных алгоритмов, и заключается оно в том, что мы должны выбирать числа в случайном порядке; в быстрой сортировке этим числам соответствуют индексы центрального элемента. Хоть это и не очевидно, в каждом стоящем языке в программных библиотеках будут хорошие функции, выполняющие данную роль. В главе 16 мы подробней расскажем о рандомизированных алгоритмах и уделим время генерированию случайных чисел.

У вас может возникнуть вопрос: если быстрая сортировка в среднем имеет ту же сложность, что и сортировка слиянием, плюс шанс — хоть и ничтожный, — что мы столкнемся с задержками, то зачем ее вообще использовать? Или почему у нее нет названия в честь кого-нибудь великого? Быстрая сортировка действительно заслуживает отдельного названия, она чаще остальных применяется в работе. В отличие от сортировки методом слияния, ей не требуется много дополнительного места. К тому же структура алгоритма удобна для быстрых реализаций на компьютерах. Цикл в строчках 4–7 алгоритма 12.10 включает в себя увеличение переменной и сравнение индексных значений с другими значениями — две быстрые операции, которые выполняются с большой скоростью.

12.6. Богатство выбора

Мы рассмотрели пять сортировочных методов, но на самом деле их куда больше. Однако даже взглянув на нашу пятерку, можем ли мы выбрать неоспоримого победителя, который во всем лучше остальных?

Такого волшебного метода не существует. Все методы имеют свои плюсы и минусы. Вспомните сортировку слиянием и быструю сортировку. Сортировка слиянием перебирает элементы, требующие сортировки, один за другим. Быстрой же сортиров-

ке требуется выбор в случайном порядке. Если мы имеем дело со стандартными массивами, то нам без разницы, какой метод применять, потому что ключевая особенность массивов в том, что все элементы достаются за одинаковое, постоянное время. Однако если мы работаем с данными, расположенными в другой структуре, где доступ к соседнему элементу быстрее, чем к случайному, то мы сразу заметим, что быстрая сортировка стала не так эффективна, в то время как сортировка слиянием ведет себя по-прежнему. Быстрая сортировка не подходит для сортировки списков, где переход к элементу возможен только через соседние, соединенные с ним элементы.

У сортировки кучей и сортировки слиянием есть гарантированная верхняя граница $O(n \lg n)$, в то время как у быстрой сортировки есть ожидаемое выполнение $O(n \lg n)$ и в редких случаях выполнение $O(n^2)$. В большинстве случаев разница спорна. Но бывают особые случаи, когда нам нужно точно знать, что сортировка займет не более $O(n \lg n)$, а значит, быстрая сортировка не подойдет.

Сортировка слиянием и сортировка кучей имеют одинаковую границу производительности, но их внутренняя работа отличается. Сортировка слиянием — это алгоритм, который можно распараллелить, то есть его части могут работать одновременно и независимо друг от друга на разных компьютерах, процессорах или процессорных ядрах. Например, если нам надо обработать 16 элементов, мы можем разделить и проработать наши данные четыре раза так, чтобы у нас было 16 сегментов для сортировки, а затем слияния. Мы можем выделить по сегменту каждому элементу, что в шестнадцать раз ускорит их сортировку.

Сортировка методом слияния также подходит для упорядочивания данных во внешней памяти. Мы описывали алгоритм, который работает с данными в основной памяти компьютера. Сортировка же во вторичной памяти — совсем другая песня, так как затраты на перемещение данных и чтение различны, когда мы работаем со внешней и основной памятями. Сортировка слиянием удобна при работе с внешним хранилищем, чего не скажешь о сортировке кучей.

Достигнув вычислительной сложности $O(n \lg n)$, естественно задаться вопросом, разумно ли воспользоваться алгоритмами, вроде сортировок методом выбора и методом вставок, которые достигают только $O(n^2)$. Ответ — разумно. В нашем анализе мы подсчитывали только сравнения и замены, так как они самые затратные при сортировке большого числа элементов. Однако мы не обрисовали всю картину. Всем кодам надо время на выполнение работы, что выражается в вычислительных затратах. У рекурсии, к примеру, своя стоимость. В каждый рекурсивный вызов компьютер должен зафиксировать полное состояние функции, которая выполняется в данный момент, он должен задать обстановку, позволяющую состояться рекурсивному вызову, и должен восстановить состояние функции, совершившей рекурсивный вызов, когда она возвращается. После сложения всех этих неочевидных затрат может оказаться, что в работе с небольшими базами данных выгодней, в плане скорости, прибегнуть к простому алгоритму, вроде сортировки методом выбора или сортировки методом вставок, так как преимущество $O(n \lg n)$ перед $O(n^2)$ никаким образом не может себя проявить. Один из способов воспользоваться этим — прибегнуть не к быстрой сортировке или сортировке слияния, а к более простым методам, когда мы имеем дело с небольшим количеством элементов, скажем, двадцатью.

Еще одно замечание касательно выбора метода сортировки, когда речь идет о доступной памяти. Мы видели, что сортировке слиянием требуется дополнительное место, по объему равное всем хранимым элементами, $O(n)$. Быстрой сортировке тоже нужно дополнительное место, хотя на первый взгляд и не скажешь. Все дело в ее рекурсивной природе. В среднем у дерева быстрой сортировки имеется глубина $O(\lg n)$; следовательно, нам нужно столько места для отслеживания рекурсии, сколько требуется на запись о состоянии между вызовами, когда мы переходим на уровень ниже. Всем остальным методам: выбору, вставкам и куче — требуется минимум дополнительного пространства: всего-то место под один элемент, чтобы произвести замену.

Помимо скорости и места нас также может интересовать, каким образом сортировочный алгоритм справляется с равен-

ством, то есть с записями, у которых одинаковые ключевые значения. Вернемся к картам и взглянем на рисунок 12.20. Нам нужно рассортировать карты соответственно их рангам. Если мы прибегнем к сортировке кучей, то получим ситуацию, изображенную на рисунке 12.20(a). Если мы прибегнем к сортировке методом вставок, то получим ситуацию с рисунка 12.20(b). В сортировке кучей относительный порядок 5♣ и 5♡ был изменен в отсортированном массиве на обратный, в то время как при сортировке методом вставок он сохранился. Поэтому мы называем сортировку методом вставок устойчивой сортировкой, а сортировку кучей — неустойчивой. Точное определение следующее: сортировочный алгоритм называется устойчивым, если он сохраняет относительный порядок расположения записей с одинаковыми ключами. Так что, если у нас есть две записи R_i и R_j с ключами K_i и K_j, соответственно, при этом `Compare`(K_i , K_j) = 0 и, более того, в вводных данных R_i идет прежде R_j, то такой алгоритм устойчивый, если в выводимых данных R_i идет прежде R_j. Среди всех рассмотренных нами алгоритмов устойчивыми являются только сортировка методом вставок и сортировка слиянием, в то время как остальные неустойчивые.

(a) Сортировка кучей: неустойчивая сортировка

(b) Сортировка методом вставок: устойчивая сортировка

Рисунок 12.20. Устойчивая и неустойчивая сортировки

Наш рассказ о сортировке затронул лишь крохотную часть сортировочных алгоритмов; данной теме спокойно можно посвятить целую книгу (и такие книги есть). Существует намного больше пяти сортировочных алгоритмов, которые мы рассмотрели в этой главе. Но в большинстве случаев хватет и их. На самом деле, в большинстве случаев достаточно и быстрой сортировки.

Однако всегда имеются нестандартные ситуации, когда иной алгоритм или иной вариант алгоритма может оказаться выгодней в плане скорости или места. Выбор и применение алгоритма в заданных условиях — творческий процесс, а не следование заученным формулам.

Примечания

Герман Холлерит изобрел первую счетно-аналитическую машину, названную в его честь, и проложил путь современным компьютерам [3]. Сортировки методом выбора и методом вставок уже существовали к 1956 году, когда их включили в библиографическое исследование методов сортировок [72]. Роберт Флойд представил исходную версию сортировки кучей в 1962 году, назвав ее древесной сортировкой [66]. Дж. У. Дж. Уильямс представил улучшенную версию, названную сортировкой кучей, в июне 1964 года, за которой, в декабре 1964 года [218], последовала еще одна версия Флойда, названная древесной сортировкой 3 [68]. Сортировка слиянием была предложена Джоном фон Нейманом в 1945 году [114, с. 158]. Быстрая сортировка была представлена Ч. Э. Р. Хоаром в 1961 году [93], [94], [95].

Упражнения

1. Реализуйте сортировку методом слияния, используя алгоритм 12.8 таким образом, чтобы на выходе она выдавала граф вызовов. Чтобы граф вызовов можно было прочесть, его надо структурировать: рекурсивные вызовы должны размещаться чуть правее относительно вызывающей функции. Соответственно, граф вызовов должен быть похож на граф, изображенный на рисунке 12.14, но без соединительных линий.
2. Реализуйте сортировку методом слияния, используя алгоритм 12.8 таким образом, чтобы на выходе получился граф

вызовов, как на рисунке 12.14, с соединительными линиями. Соединительные линии можно собрать из простых символов вроде |, _ и +.

3. При обменном слиянии массивов алгоритм 12.7 использует массив C длиной $h - l + 1$. Так как C может быть меньше A, нам надо быть аккуратней с различными индексами в C, для чего стоит прибегнуть к переменным *cm* и *ch* и индексированию $k - l$ в строчке 2. Реализуйте алгоритм, полагая, что C такой же длины, как и A, поэтому у обоих может быть общий диапазон индексов.
4. В небольших массивах сортировки методом вставок и методом выбора могут работать быстрее сортировки слиянием. Сравните свои реализации сортировки методом вставок, сортировки методом выбора и сортировки слиянием и найдите порог, где сортировка слиянием работает быстрее двух других.
5. Реализуйте сортировку слиянием, учитывая, что можно сэкономить на отсортированных массивах, которые не нужно объединять. Проверьте $A[m]$, меньше он или равен $A[m + 1]$, и объедините их, только если $A[m] > A[m + 1]$.
6. Если мы изменим строчку 15 алгоритма 12.7 так, что будет использоваться строгое сравнение (<), то будет ли сортировка слиянием по-прежнему устойчивым сортировочным алгоритмом?
7. Вместо того, чтобы выбирать случайный центральный элемент каждый раз, когда мы делаем разбивку при быстрой сортировке, можно до начала процесса просто перемешать все элементы массива A. Таким образом, элементы A будут стоять в случайном порядке, и мы сможем указывать в качестве центрального элемента самый первый элемент каждого массива, который мы хотим разделить. Используйте такой подход в реализации быстрой сортировки.
8. Еще один способ выбора центрального элемента заключается в том, чтобы вместо случайного элемента брать три элемента: первый, последний и один из середины массива, который надо разделить, — и в качестве центрального

элемента использовать медиану всех трех элементов, то есть элемент со средним значением. Такой подход также прекрасно работает на практике, и вероятность низкой скорости работы крайне мала. Реализуйте быструю сортировку, используя описанный метод.

9. Как и с сортировкой слиянием, бывает полезно сочетать быструю сортировку с другими, более простыми алгоритмами, например сортировкой методом вставок, при работе с небольшими массивами. Реализуйте быструю сортировку, сочетая ее с сортировкой методом вставок таким образом, чтобы, когда массив, требующий разделения, достигал определенного порога, быстрая сортировка сменялась методом вставок. Продолжайте экспериментировать, чтобы узнать, каков этот порог.

13 Гардероб, коллизия и слот

Когда вы сдаете в гардероб пальто или сумку, вам выдают номерок. Когда же вы собираетесь покинуть помещение, вы отдаете номерок, и вам возвращают ваше пальто или сумку.

Если вдуматься, какую задачку решает данный процесс, можно увидеть, что это задача о поиске и нахождении объекта. Ваше пальто — это объект, который вы передаете гардеробщику; пальто сначала привязывают к определенному номерку, а затем возвращают вам. Метод работает за счет того, что в гардеробе находятся ряды пронумерованных вешалок для одежды и пронумерованные отделения для сумок. Вашему номерку соответствует определенная вешалка или определенное отделение. Гардеробщик же отвечает за то, чтобы разместить ваш объект в нужном месте и чтобы забрать его с помощью отданного вами номерка.

Когда речь идет о поиске, люди чаще всего представляют себе, что нужно перебирать вещи до тех пор, пока не найдется нужная; процесс становится проще, если вещи отсортированы так, что мы можем искать среди них более слаженно. Гардероб же показывает, что есть и другой способ: присвоить вещи адрес, где она будет располагаться, и затем сразу достать ее из указанного места. В гардеробе адрес указан на номерках, и гардеробщик на самом деле ищет не ваши вещи: он смотрит на номерок, на адрес и идет к указанной вешалке или отделению с вашими вещами.

Весь метод в целом можно назвать определением местонахождения *без поиска*: мы не ищем объект, а получаем адрес места, где он будет храниться. Соотносим объект с адресом, а затем, когда хотим найти объект, мы идем прямиком к нужному месту.

Теперь рассмотрим данный метод с точки зрения компьютерного мира, где наши объекты представлены в виде записей, содержащих атрибуты. Мы хотим уметь переходить напрямую от

записи к адресу, по которому она размещена; адресом будет число, соответствующее области в компьютерной памяти. Главная загвоздка в такой задачке — то, как соотнести адрес с элементом. Нам нужен быстрый и надежный способ, и мы не можем положиться на какого-либо посредника. Когда мы ищем запись, мы ищем ее по одному или большему числу атрибутов, которые формируют ключ записи, поэтому нам нужен метод получения адреса из ключа записи. Так как адрес — это число, нам нужен способ, который перенесет нас от ключей к числам. Иными словами, мы нуждаемся в функции, скажем, $f(K)$, которая возьмет ключ K записи R и вернет значение $a = f(K)$. Значение a представляет собой адрес, где будет храниться запись, мы идем по нему и оставляем там нашу запись. Каждый раз, когда нам требуется достать нашу запись с помощью ключа, мы снова прибегаем к функции $f(K)$, которая вернет то же адресное значение a, откуда мы и заберем нашу запись. Функция должна быть быстрой и достаточно скоро находить нужный адрес: как минимум, не дольше, чем поиск по записям до тех пор, пока не найдется та, которую мы ищем. В противном случае толку от такой функции никакого. Если же мы в самом деле обзаведемся такой функцией, то разрешим нашу задачку с местоположением новым способом: не мы будем искать элемент, а сам элемент расскажет нам, где его найти: расскажет с помощью нашей функции.

13.1. Соотнесение ключей и значений

Представьте, что у нас есть n разных записей и все они нам известны. Тогда проблема заключается в том, чтобы найти такую функцию f, чтобы для каждого ключа из n записей f создавала различное значение от 0 до $n - 1$. Таким образом, мы можем положить каждую запись R в таблицу T, то есть массив, размером n так, чтобы, если $f(K) = a$, где K — это ключ от R, то $T[a] = R$. Такую установку вы можете видеть на рисунке 13.1, где слева расположены элементы, справа — таблица, а стрелочками обозначен эффект каждого вызова функции $f(K)$, то есть адрес, связанный с конкретной таблицей.

Подобрать такую функцию *f* для набора записей не просто. Мы можем сами написать *f*, но это нудно и сложно. Алгоритм 13.1 показывает функцию, которая соотносит каждое из 31 наиболее часто употребляемых английских слов с числом от –10 до 29 (мы можем добавить десять, чтобы все индексы таблицы были положительными). Предупреждение: вовсе не обязательно понимать работу алгоритма 13.1. Пока что примите его просто как есть.

Алгоритм использует функцию `Code`, которая берет символ и приписывает ему числовое значение. Символ забирается из строки вводных данных, *s*, которая, как мы полагаем, состоит как минимум из четырех символов, от *s*[0] до *s*[3]. Если в слове меньше четырех символов, мы полагаем, что у него в конце идут пробелы. `Code` сперва припишет нулевое значение пробелу, единицу — символу «A», двойку — «B» и так далее. Затем он добавит 1, если соотнесенное значение больше девяти, и 2 к значению результата, если оно больше 19. Коды, приписанные каждому из первых трех символов слова, преобразовываются вместе, и для каждого слова получается свое, уникальное число.

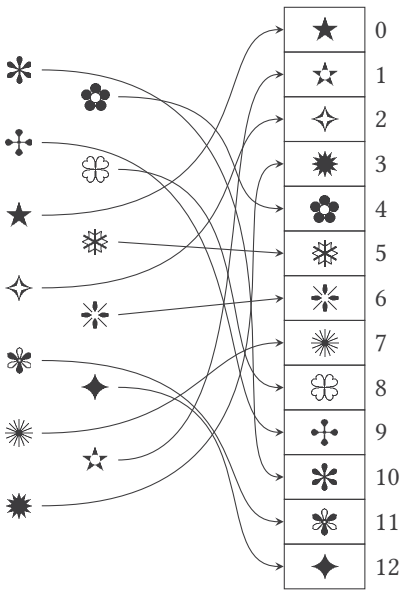


Рисунок 13.1. Соотнесение ряда элементов с позициями в таблице

Возможно, все происходящее вам кажется каким-то фокусом-покусом, но оно работает — в таблице 13.1 можно взглянуть на выводимые данные для введенного набора распространенных английских слов. Перед вами идеальное соотнесение, потому что всем словам присвоены различные значения.

Алгоритм 13.1. Идеальное числовое соотнесение для 31 наиболее распространенного слова английского языка

`PerfectMapping(`s`)` $\rightarrow r$

Вводные данные: s, строка символов в заданном списке из 31 наиболее распространенного слова в английском языке.

Выводимые данные: r, адрес в диапазоне –10, 1, …, 29.

```
1  r ← −Code(s[0])
2  s ← Code(s[1])
3  r ← r − 8 + s
4  if r ≤ 0 then
5      r ← r + 16 + s
6  s ←Code(s[2])
7  if s = 0 then
8      return r
9  r ← r − 28 + s
10 if r > 0 then
11     return r
12 r ← r + 11 + s
13 t ← Code(s[3])
14 if t = 0 then
15     return r
16 r ← r − (s − 5)
17 if r < 0 then
18     return r
19 r ← r + 10
20 return r
```

К тому же оно быстро работает, так как ему всего лишь требуется определенное количество тривиальных операций. Тем не менее такое соотнесение непрактично для любого другого набора слов. Оно может не сработать, если мы изменим хотя бы одно слово, так как может получиться одинаковое значение для двух разных слов и, следовательно, два одинаковых адреса для двух раз-

личных элементов. Мы просили вас принять данный алгоритм как есть не потому, что он полезен, а чтобы показать, что хорошую функцию соотнесения действительно сложно отыскать.

Несмотря на то, что алгоритм 13.1 и впрямь присваивает каждому слову уникальный адрес, он оставляет в распределении дыры. У нас 31 слово и значения функции –10, –9 … 39, так что есть девять значений функции, для которых не нашлось слова. Это не значит, что функция работает как-то не так, но нам бы лучше уйти от лишних затрат. Есть идея использовать первый символ каждого ключа, последний символ каждого ключа и определенное число символов в ключе. Мы можем присвоить числовой код первому и последнему символам и получить значение следующим образом: $h = \texttt{Code}(b) + \texttt{Code}(e) + |K|$, где K — это ключ, $|K|$ — длина ключа, b — символ в начале ключа, а e — символ в конце ключа. Итоговое значение будет индексом таблицы. Конечно же, тут проблема в нахождении подходящих числовых кодов, чтобы весь план заработал. Функция `Code`, которую мы применяли прежде, здесь не подойдет, потому что она не выдает уникальные значения. Например, слову «ARE» присвоено значение 9 (1 + 5 + 3), хотя у слова «BE» то же самое значение.

Таблица 13.1. Присвоение 31 наиболее распространенному слову английского языка числовых значений от –10 до 30

A	7	FOR	23	IN	29	THE	-6
AND	-3	FROM	19	IS	5	THIS	-2
ARE	3	HAD	-7	IT	6	TO	17
AS	13	HAVE	25	NOT	20	WAS	11
AT	14	HE	10	OF	4	WHICH	-5
BE	16	HER	1	ON	22	WITH	21
BUT	9	HIS	12	OR	30	YOU	8
BY	18	I	-1	THAT	-10		

Если мы ищем подходящее кодирование, то обнаружим, что такое действительно существует и мы используем его в алгоритме 13.2; он представлен в алгоритме массивом *C*. Каждой позиции в массиве соответствует числовой код, который мы приписываем символу. Позиция 0 в массиве содержит код для «A», позиция 1 —

код для «B» и так далее для всех остальных символов. Функция `Ordinal`(*c*) возвращает исходную позицию символа *c* в алфавите, начиная с нуля. Массив *C* содержит несколько элементов равных –1; это фиктивные элементы для символов, которые в нашем наборе данных не появляются ни первыми, ни последними. Например, «C» не является ни первой, ни последней буквой ни в одном из наших слов.

Алгоритм 13.2. Минимальное идеальное соотнесение

`MinimalPerfectMapping`(*s*) $\rightarrow r$

Вводные данные: *s*, строка символов в заданном списке из 31 наиболее распространенного слова в английском языке.

Выводимые данные: *r*, адрес в диапазоне 1, …, 32.

Data: *C*, an array of 26 integers

```
1  C ← [
2      3,  23,  -1,  17,   7,  11,  -1,   5,   0,
3     -1,  -1,  -1,  16,  17,   9,  -1,  -1,  13,
4      4,   0,  23,  -1,   8,  -1,   4,  -1
5  ]
6  l ← |s|
7  b ←Ordinal(s[0])
8  e ←Ordinal(s[l − 1])
9  r ← l + C[b] + C[e]
10 return r
```

Таблица 13.2. Минимальное идеальное соотнесение 31 наиболее распространенного английского слова числам от 1 до 31

A	7	FOR	27	IN	19	THE	10
AND	23	FROM	31	IS	6	THIS	8
ARE	13	HAD	25	IT	2	TO	11
AS	9	HAVE	16	NOT	20	WAS	15
AT	5	HE	14	OF	22	WHICH	18
BE	32	HER	21	ON	28	WITH	17
BUT	26	HIS	12	OR	24	YOU	30
BY	29	I	1	THAT	4		

Алгоритм 13.2 приписывает 31 слову 31 различный адрес, от 1 до 32, как видно в таблице 13.2. Это минимальное идеальное соотнесение, так как всем словам присвоены разные значения, и алго-

ритм приписывает *n* словам *n* различных значений, наименьшее из возможных чисел.

Можно ли считать, что мы улучшили предыдущий алгоритм? Можно. У нас есть верное соотнесение и не остается лишнего места. К тому же данный алгоритм намного проще. Является ли он общим, практическим решением нашей задачи? Нет, он является решением, только если нам известны все наши ключи и мы можем получить подходящее кодирование для букв. Такое не всегда возможно. Если в нашем наборе данных присутствует слово «ERA», алгоритм не будет работать, потому что у «ERA» будет точно такое же приписанное значение, как и у «ARE». Мы можем это исправить, внеся небольшие изменения в алгоритм 13.2, однако проблема никуда не денется: нет никаких гарантий, что в неизвестном наборе ключей каждому из них будет присвоен верный адрес. Более того, мы бы тогда пропустили интересную часть, раскрывающую то, каким образом мы вообще находим массив *C*. Нам бы пришлось прибегнуть к особому алгоритму. Так что все куда сложнее, чем кажется на первый взгляд.

13.2. Хеширование

Искомое решение заключается в том, чтобы найти функцию общего назначения, которая берет ключи, генерирует значения в заданном промежутки и работает, не зная ничего о предыдущих элементах и их ключах. Функция должна гарантировать, что, какой бы ни был ключ, они вернет значение, находящееся в заданном нами промежутке. Также функция должна избегать, насколько возможно, приписываний разным ключам одинаковых значений. Фраза «насколько возможно» употреблена здесь не зря: если число возможных ключей больше, чем промежуток приписанных значений, тогда просто невозможно избежать повторных соотнесений. Например, если у нас есть таблица размером *n* и есть 2*n* возможных ключа, то как минимум *n* ключам будут приписаны те же значения, что и у других ключей.

Коснемся терминологии. Метод, о котором мы ведем речь, называется хешированием. Название происходит от слова «хеш»

(hash), что переводится с английского как «резать и кромсать мясо на мелкие кусочки, перемешивая их и получая мясное крошево» (никакой связи с психогенными отравляющими веществами). Суть такова, что мы берем ключ, вырезаем его, кодируем и преобразуем так, чтобы получить из него адрес, который мы сможем использовать для поиска. Нам нужны, по возможности, неодинаковые адреса, поэтому мы в некотором смысле разбрасываем ключи по диапазону адресов; именно поэтому данный метод также называется рассеянной памятью. Функция, которую мы использовали, называется хешом, хеш-функцией или функцией хеширования. Массив, или таблица, в котором мы приписываем ключи, называется хеш-таблицей. Элементы таблицы называются слотами. Если функция может присвоить всем ключам разные значения, ее называют идеальной функцией хеширования. Если ей удается присвоить всем ключам разные значения, не требуя лишнего места, так, что промежуток сгенерированных адресов равен числу ключей, то ее называют минимальной идеальной функцией хеширования. Когда же такое невозможно и функция приписывает двум разным ключам одно и то же значение, мы говорим о коллизии. Как мы уже говорили, если диапазон адресов меньше числа возможных ключей, то коллизии неизбежны. Вопрос в том, как встретить минимальное количество коллизий и что делать, когда они все же попадаются.

Данная ситуация — яркий пример значимого в математике закона, принципа Дирихле, согласно которому, если у вас есть n элементов, m контейнеров при $n > m$ и вы хотите разместить элементы в контейнерах, то как минимум в одном контейнере будет находиться более одного элемента. Хотя это очевидно, все же взгляните на рисунок 13.2, принцип Дирихле применим не только к таким очевидным ситуациям.

Например, можно доказать, что в любом большом городе будут как минимум двое людей с одинаковым количеством волос на голове. Среднее количество волос на голове человека — примерно 150 000. Точное количество волос у людей варьируется, однако можно смело утверждать, что не существует человека, на голове которого, скажем, 1 000 000 волос. Следовательно, в любом городе

с населением больше миллиона человек будет как минимум два жителя с одинаковым количеством волос.

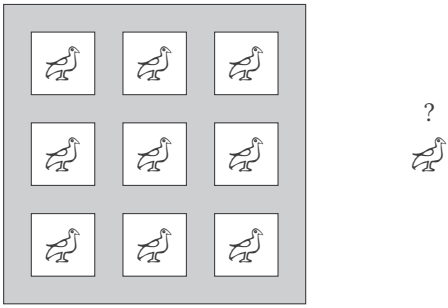


Рисунок 13.2. Принцип Дирихле

Еще один пример — так называемый парадокс дней рождения, когда вас спрашивают, сколько человек минимально должно находиться в комнате, чтобы у двух из них совпадал день рождения. Назовем вероятность такого совпадения $P(B)$. Вычислить $P(B)$ проще, если взяться за противоположную задачу, то есть узнать, какова вероятность, что среди людей, находящихся в комнате, у всех разные даты рождения. Согласно основным законам теории вероятности, если мы называем данную вероятность $P(\bar{B})$, то получаем $P(\bar{B}) = 1 - P(B)$. Теперь, чтобы такое случилось, все дни рождения должны выпадать на разные даты. Если мы будем по очереди перебирать людей, то день рождения первого человека в комнате может оказаться в любой день года, вероятность чего равна 365/365 (мы пренебрегаем високосными годами). День рождения второго человека может оказаться в любой день года, кроме того, в который родился первый человек. Таких дней 364, поэтому вероятность равна 364/365, а вероятность того, что у двоих первых опрошенных дни рождения в разные дни, равна $365/365 \times 364/365$. Мы продолжаем рассуждать в том же духе и получаем, что для n людей вероятность того, что у них окажутся разные даты рождения, составляет $365/365 \times 364/365 \times \ldots \times (365 - n + 1)/365$. Если мы произведем расчеты, мы увидим, что для $n = 23$ у нас $P(\bar{B}) = 365/365 \times 364/365 \times \ldots \times 343/365 \approx 0{,}49$. Это значит, что $P(B) \approx 0{,}51$, поэтому, если у вас в комнате 23 че-

ловека, то скорее всего окажется, что у двух из них общий день рождения. Конечно, если $n > 23$, вероятность возрастает; можете поспорить с кем-нибудь на большой шумной вечеринке и скорее всего выиграете спор. Если результаты вас удивляют, обратите внимание, что мы не ищем людей с особой датой рождения или кого-то, кто родился с вами в один день, мы ищем двух случайных людей с общим днем рождения.

13.3. Функции хеширования

Вернемся к функциям хеширования. Главная задача здесь — найти ту, которая минимизирует коллизии. Например, представим, что наши ключи состоят из 25 символов и содержат название улицы и номер дома. Число теоретически возможных ключей равно $25^{37} = 1{,}6 \times 10^{39}$: каждый символ может быть или одной из 26 букв, или пробелом, или одной из десяти цифр — в общей сложности 37 случаев, а у нас всего 25 На практике возможных ключей, конечно, гораздо меньше, потому что не все строки символов отображают реальные адреса; и в данном случае количество адресов, которые нам попадутся, будет меньше количества реальных, действующих в нашем мире адресов. Допустим, мы ожидаем порядка 100 000 ключей, или адресов с улицами и номерами домов, о которых мы ничего не знаем. Поэтому мы ищем хеш-функцию, которая соотнесет область размером $1{,}6 \times 10^{39}$ с набором из 100 000 различных значений. Согласно принципу Дирихле, коллизии неизбежны. Однако мы попробуем найти функцию, которая приписывает ключам как можно больше значений. Если функция хорошо сработает, нам может попасться совсем мало коллизий на 100 000 ключей.

Если наши ключи представлены не строками символов, а числами, то семейство функций, зарекомендовавших себя, является простым остатком, или модулем, от деления ключей на размер нашей таблицы адресов:

$$h(K) = K \textbf{ mod } m,$$

где m — это размер, а K — это ключ. Результат по определению будет находиться между 0 и m – 1 и поэтому вместится в таблицу.

Вычисления приводят нас к простому алгоритму 13.3. Скорее всего, единственное, что следует объяснить, — это то, что алгоритм работает для всех целых ключей, не только для неотрицательных, потому что деление с остатком работает как с положительными, так и с отрицательными числами.

Алгоритм 13.3. Функция хеширования для целых чисел

`IntegerHash`(k, m) $\rightarrow h$

Вводные данные: k, целое число; m, размер хеш-таблицы.

Выводимые данные: h, хеш-значение k.

```
1  h ← k mod m
2  return h
```

Например, представим, что мы имеем дело с записями, в которых содержится некая государственная информация и каждый ключ записи — это международный код для звонков. В таблице 13.3 содержатся первые 17 из самых населенных стран мира. Если мы начнем добавлять страны в хеш-таблицу размером 23, используя их международные коды в качестве ключей, то ситуация после добавления первых десяти стран будет выглядеть как на рисунке 13.3. С такой хеш-таблицей мы можем найти каждую страну, зная ее международный код, надо лишь вычислить $h(K) = K$ **mod** 23.

Таблица 13.3. Первые 17 самых населенных стран мира и их международные коды (2015)

China	86	Japan	81
India	91	Mexico	52
United States	1	Philippines	63
Indonesia	62	Vietnam	84
Brazil	55	Ethiopia	251
Pakistan	92	Egypt	20
Nigeria	234	Germany	49
Bangladesh	880	Iran	98
Russia	7		

Когда мы говорили, что функции вычисления остатка хороши, мы подразумевали, что данные функции хороши в плане

минимального количества коллизий: то есть коллизии будут встречаться редко. Однако это зависит от того, насколько рационально выбран размер таблицы. Если ключи представляют собой десятичные числа, то размер таблицы со степенью десяти, 10^x для некоего x, будет плохим выбором. Дело в том, что остаток от деления положительного целого числа на 10^x — это просто последние x цифр числа. Например, 12345 mod 100 = 45, 2345 mod 100 = 45 и так далее. В целом, когда каждое положительное целое число с n цифр $D_nD_{n-1}\ldots D_1D_0$ имеет значение $D_n \times 10^n + D_{n-1} \times 10^{n-1} + \ldots + D^1 \times 10^1 + D_0 \times 10^0$, для каждой степени десяти, 10^x, мы имеем:

$$\frac{D_nD_{n-1}\ldots D_1D_0}{10^x} = 10^x \times D_nD_{n-1}\ldots D_x + x_{x-1}D_{x-2}\ldots D_1D_0.$$

Если $x \leq n$, то первая часть суммы является частным, а правая часть — остатком. Если же $x > n$, то первой части, частного, вовсе не существует, поэтому всегда:

$$D_nD_{n-1}\ldots D_1D_0 \bmod 10^x = D_{x-1}D_{x-2}\ldots D_1D_0.$$

Это вытекает прямиком из факта, что значение цифры d в позиции x положительного целого числа является просто $d \times 10^x$; взгляните на рисунок 13.4, чтобы увидеть, как это сказывается на делении с остатком: у всех чисел, оканчивающихся на одинаковые x цифр, будет одинаковое значение хеша.

Если наше число является отрицательным целым, тогда возникает та же проблема и в делении с остатком играют роль только последние x цифр. Если мы используем определение модуля, мы увидим, что –12345 mod 100 = 55, –2345 mod 100 = 55 и так далее. Дело не только в урезании первых цифр числа, а в том, что проблема все еще не решена.

Мы следуем все той же логике, и еще один плохой выбор, если мы имеем дело с числами бинарной системы, — это таблица с размером степени 2. Нам вообще не нужна таблица, размером которой служит степень базовой системы счисления, которую используют наши ключи. Во всех подобных случаях принимается в расчет только подмножество цифр наших ключей: наименьшие x цифр, если размер таблицы равен b^x, где b — это база систе-

мы счисления. Нам нужен способ сбалансировать распределение ключей в хеш-таблице так, чтобы минимизировать вероятность коллизии. В идеальном варианте хеш-функция будет с равной вероятностью распределять все ключи между элементами таблицы, то есть значения хеша будут равнораспределенными. Если во время хеширования принимать во расчет только меньшие *x* цифр, то мы тут же столкнемся с коллизиями для всех чисел, которые заканчиваются на те же *x* цифр, вне зависимости от отличий предшествующих цифр. Похожая проблема подстерегает нас, если мы выбираем таблицу, размером которой является четное число. Все четные ключи будут соотнесены с четными хеш-значениями, а все нечетные ключи будут соотнесены с нечетными хеш-значениями. Лучше всего, чтобы все ключи с равной вероятностью были соотнесены с любым хеш-значением во избежание смещения, поэтому четные числа для деления с остатком не рассматриваются.

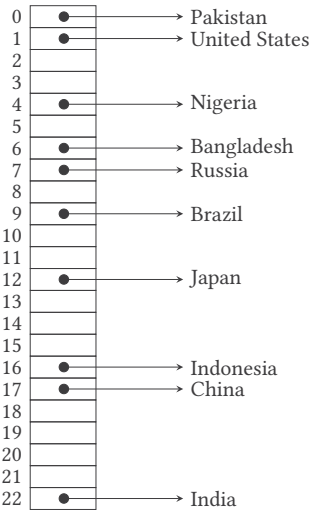


Рисунок 13.3. Хеш-таблица размером 23, содержащая первые десять самых населенных стран, их ключами являются международные коды

Принимая во внимание все вышесказанное, можно сделать заключение, что наилучшим размером таблицы является про-

стое число. На практике, если ваша таблица может вместить до 1000 ключей, ее размер будет не 1000, а простое число 997.

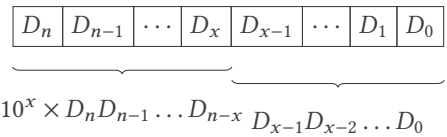


Рисунок 13.4. Разбивка положительного целого числа по модулю степени 10^x

До сего момента мы полагали, что ключи — это целые числа. Если ключи представляют собой строку символов, то мы можем обращаться с ними как с числами в системе счисления с подходящей основой, например 26, если у нас есть лишь буквы алфавита. Значение строки символов в таком случае вычисляется так же, как и любое значение в любой позиционной системе счисления. Если s — строка, состоящая из n символов, то ее значение v = `Ordinal`(s[0])b^{n-1} + `Ordinal`(s[1]) b^{n-2} + … + `Ordinal`(s[n – 1])b^0, а хеш-значение $h = v \bmod m$. Это можно записать в виде алгоритма 13.4. Если мы применим алгоритм к строке «HELLO» для b = 26 и m = 31, мы получим:

v_0 = `Ordinal`(«H») · 2^{64} = 7 · 456 976 = 3 198 832
v_1 = 3 198 832 + = `Ordinal`(«E») · 26^3 = 3 198 832 + 4 · 26^3 = 3 269 136
v_2 = 3 269 136 + `Ordinal`(«L») · 26^2 = 3 269 136 + 11 · 26^2 = 3 276 572
v_3 = 3 276 572 + `Ordinal`(«L») · 26^1 = 3 276 572 + 11 · 26 = 3 276 858
v_4 = 3 276 858 + `Ordinal`(«O») = 3 276 858 + 14 = 3 276 872
h = 3 276 872 mod 31 = 17,

где v_i является значением v в i-й итерации алгоритма.

Данная процедура равнозначна обращению со строкой, содержащей n символов, как с многочленом в степени n – 1:

$$p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0,$$

где каждый коэффициент a_i является порядковым значением i-го символа строки, если отсчитывать слева. Многочлен вычисляется при $x = b$, где b — это основание нашей придуманной системы,

число возможных символов. В заключении мы прибегаем к делению с остатком. В нашем примере со строкой «HELLO» многочленом является $p(x) = 7x^4 + 4x^3 + 11x^2 + 14$, и мы рассчитываем его для $x = 26$.

Алгоритм 13.4. Функция хеширования для строк

`StringHash`(s, b, m) $\rightarrow h$

Вводные данные: s, строка; b, основание числовой системы; m, размер хеш-таблицы.

Выводимые данные: h, хеш-значение s.

```
1  v ← 0
2  n ← |s|
3  for i ← 0 to n do
4      v ← v + Ordinal(s[i]) · b^(n−1−i)
5  h ← v mod m
6  return h
```

Нет ничего проще, чем вычислить многочлен. Для многочлена со степенью n, $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$, мы начинаем высчитывать все степени слева направо, что слишком расточительно. В данный момент для простоты объяснения мы полагаем, что все коэффициенты не являются нулями; довод, к которому мы прибегаем для общих случаев, когда у нас могут быть и нулевые коэффициенты. Если мы вычисляем x^n, то мы уже нашли x^{n-1}, но мы нигде его не используем. Для члена $a_n x^n$ алгоритму нужно $n - 1$ операций умножения (для расчета степени) плюс одно умножение степени с коэффициентом a_n; итого n операций умножения. То же самое с членом $a_{n-1}x^{n-1}$, ему нужно $n - 1$ умножений. В общей сложности алгоритму нужно $n + (n + 1) + \ldots + 1 = n(n - 1)/2$ операций умножения, а также n сложений, чтобы узнать сумму членов.

Лучше рассчитать многочлен, читая его слева направо и повторно используя степени для вычисления последующих множителей. Поэтому, если мы рассчитали x^2, нам не нужно заново вычислять x^3, потому что $x^3 = x \cdot x^2$. В таком случае нам нужно одно умножение для $a_1 x$, два — для $a_2 x^2$ (одно, чтобы получить x^2 из x, и одно, чтобы получить $a_2 x^2$), потом еще два умножения

для a_3x^3 (одно, чтобы получить x^3 из x^2, и одно, чтобы получить a_3x^3) и в целом по два умножения на каждый член, включая a_nx^n. В общей сложности потребуется $2n - 1$ операций умножения и n сложений, чтобы суммировать члены. Получилось лучше, чем в предыдущем подходе.

Можно добиться больших результатов, если мы прибегнем к схеме Горнера. Данный метод математическим языком описывает, как реорганизовать многочлен следующим образом:

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = \big(\ldots(a_nx + a_{n-1})x + \cdots\big)x + a_0.$$

Мы рассчитываем выражение, начиная с самой внутренней части, где мы берем a_n, умножаем его на x и добавляем предыдущий коэффициент a_{n-1}. Умножаем полученный результат на x и добавляем предыдущий коэффициент a_{n-2}. Мы повторяем процесс, пока в итоге у нас не останется промежуточный результат, который мы умножаем на x и добавляем a_0. На самом деле мы начинаем вычисление с самого внутреннего группированного фактора, $x(a_{n-1} + a_nx)$, и переходим ко внешней части, $x\big(a_{n-2} + x(a_{n-1} + a_nx)\big)$, подставляя внутреннее значение, которое мы нашли, затем снова передвигаемся ко внешней части и так, пока не пройдем все выражение. Данный процесс легко и понятно описан в алгоритме 13.5. Схема Горнера изображена на рисунке 13.5, где мы отметили результаты как r_n, что тривиально a_n, r_{n-1}, …,r_0 = r. Справа на рисунке изображено применение схемы Горнера к многочлену, соответствующему «HELLO», $p(x) = 7x^4 + 4x^3 + 11x^2 + 11x + 14$ для $x = 26$.

Заглянув в алгоритм 13.5, мы видим, что строчка 3 выполняется n раз. Расчет многочлена теперь требует всего n сложений и n умножений — то, что нам надо.

Теперь мы можем обратить внимание на кое-что другое. Мы вычисляем, значение которого может оказаться большим; как минимум, x^n для многочлена со степенью n. Тем не менее в конце операции хеширования мы получим результат для хеш-таблицы. Так что из чего-то большого, как x^n, мы получим что-то меньше, чем табличный размер, m. В самом деле, в нашем примере мы получили 3 276 872 только для того, чтобы опуститься до $h = 3\,276\,872 \bmod 31 = 17$.

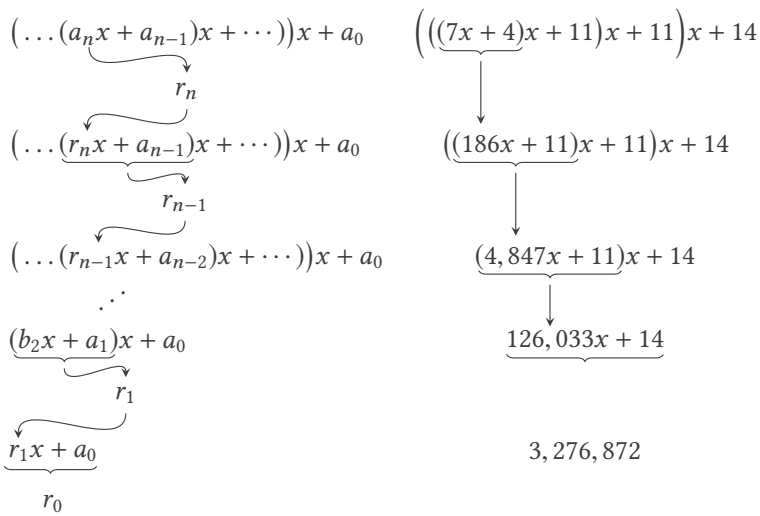


Рисунок 13.5. Схема Горнера

Особенность операции по модулю в том, что мы можем применить ее к компонентам большего выражения, опираясь на данные правила:

$$(a + b) \bmod m = \big((a \bmod m) + (b \bmod m)\big) \bmod m,$$
$$(ab) \bmod m = \big((a \bmod m) \cdot (b \bmod m)\big) \bmod m.$$

Объединяя свойства деления с остатком со схемой Горнера, мы получаем алгоритм 13.6; вместо того, чтобы напрямую повышать последовательные степени, возведение в степень происходит путем повторяющихся умножений предыдущего хеш-значения с основанием системы счисления в строчке 4 и взятия в каждой итерации остатка от операции.

В каждом этапе итерации непосредственно используется схема Горнера, мы подставляем b вместо x и `Ordinal`(c) (где c — это следующие друг за другом символы строки) для коэффициентов многочлена.

Если мы применим алгоритм 13.6 к строке «HELLO», мы получим, как ожидалось, те же, что и прежде, значения, однако возиться с расчетами нам придется куда меньше:

h_0 = `Ordinal`(“H”) mod 31 = 7 mod 31 = 7,
h_1 = (26 · 7 + `Ordinal`(“E”)) mod 31 = (182 + 4) mod 31 = 0,
h_2 = (26 · 0 + `Ordinal`(“L”)) mod 31 = 11 mod 31 = 11,
h_3 = (26 · 11 + `Ordinal`(“L”)) mod 31 = (286 + 11) mod 31,
h_4 = (26 · 18 + `Ordinal`(“O”)) mod 31 = 482 mod 31 = 17,

где h_i является значением h в i-том цикле алгоритма. Реализации алгоритма 13.6 с подходящими b и m используются в языках программирования, которые предоставляются хеш-функции для строк символов.

Алгоритм 13.5. Схема Горнера

`HornerRule`(A, x) → r

Вводные данные: A, массив, содержащий коэффициенты многочлена степени n; x, точка расчета многочлена.

Выводимые данные: r, значение многочлена в x.

```
1  r ← 0
2  foreach c in A do
3      r ← r · x + c
4  return r
```

Алгоритм 13.6. Оптимизированная хеш-функция для строк символов

`OptimizedStringHash`(s, b, m) → h

Вводные данные: s, строка; b, основание числовой системы; m, размер хеш-таблицы.

Выводимые данные: h, хеш-значение s.

```
1  h ← 0
2  foreach c in s do
3      h ← (b · h + Ordinal(c)) mod m
4  return h
```

Мы нашли функции хеширования для строк символов (алгоритм 13.6) или целых чисел (алгоритм 13.3), и теперь нас ожидает следующий кусок головоломки: как нам быть с действительными

числами, которые часто называют числами с плавающей запятой. Способ первый — превратить плавающую запятую в строку символов и хешировать ее с помощью алгоритма 13.6. Но преобразование чисел в строки — процесс небыстрый. Способ второй — превратить значения плавающей точки в целые числа путем извлечения запятой. Согласно этой задумке, 261,63 превратится в целое число 26 163, которое мы сможем хешировать с помощью алгоритма 13.3. Однако мы не можем просто взять и «извлечь запятую», поэтому что сделать что-то подобное можно, если только плавающее число представлено в расчетных данных в виде трех частей: целой части, запятой и части справа от запятой. А плавающие числа в компьютерах представляются вовсе не так.

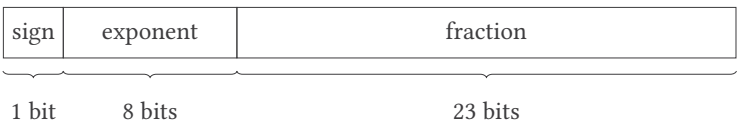


Рисунок 13.6. Представление числа с плавающей запятой

13.4. Представление чисел с плавающей запятой и хеширование

Компьютеры используют что-то схожее с экспоненциальной записью для представления действительных чисел, то есть для представления чисел с плавающей запятой. С помощью такой нотации числа представляются в виде произведения относительного числа с абсолютным значением между 1 и 10 и степенью десяти:

$$m \times 10^e$$

Число m называется по-разному: дробной частью, мантиссой или значащей частью числа. Число e является всего лишь показателем степени. Так что у нас:

$$0.00025 = 2.5 \times 10^{-5}$$

и

$$-6{,}510{,}000 = -6.51 \times 10^6.$$

Плавающие числа в компьютерах выглядят примерно так. Для их представления мы используем заранее установленное число битов. Как правило, у нас два выбора: 32 бита и 64. Давайте возьмем 32; с 64-битными числами все точно так же. Если мы используем для представления чисел с плавающей запятой на компьютере 32 бита, мы упорядочиваем их, как на рисунке 13.6.

Если перевести на человеческий язык, значение числа представляется таким вот образом, нам надо разобрать его на части и вычислить:

$$(-1)^s \times 1.f \times 2^{e-127},$$

где s — это знак, f — дробная часть, а e — показатель степени. Когда знак 0, $(-1)^0 = 1$ и число положительное. Когда знак 1, $(-1)^1 = -1$ и число отрицательное. Часть 1, f является двоичной дробью, которая не слишком отличается от десятичной. У десятичного числа, как $0{,}D_1D_2 \ldots D_n$, значение $D_1 \times 10^{-1} + D_2 \times 10^{-2} + \ldots + D_n \times 10^{-n}$. У двоичного числа, как $0{,}B_1B_2 \ldots B_n$, значение $B_1 \times 2^{-1} + B_2 \times 2^{-2} + \ldots + B_n \times 2^{-n}$. В целом дробное число $0{,}X_1X_2 \ldots X_n$ в позиционной системе счисления с основанием b имеет значение $X_1 \times b^{-1} + X_2 \times b^{-2} + \ldots + X_n \times b^{-n}$. Если у числа есть и целая, и дробная части, то его значение равно их сумме. Так что у числа $B_0{,}B_1B_2 \ldots B_n$ значение $B_0 + B_1 \times 2^{-1} + B_2 \times 2^{-2} + \ldots + B_n \times 2^{-n}$. Таким образом, число 1,01 в бинарной системе равно $1 + 0 \times 2^{-1} + 1 \times 2^{-2} = 1{,}25$ в десятичной. На рисунке 13.7 изображены наши примеры, представленные в компьютере в виде чисел с плавающей запятой.

Существует четыре случая, которые стоит рассмотреть, касающиеся расшифровки битов в плавающем числе, представленном как на рисунке 13.6. Первый случай — когда $f = 0$ и $e = 0$; тогда число равно 0. Точнее, существует два вида нулей: если $s = 1$, то у нас –0, а если $s = 1$, то у нас +0. Второй случай — когда у нас $e = 0$, а $f \neq 0$; тогда число расшифровывается немного иначе, чем в приведенной выше формуле, а именно:

$$(-1)^s \times 0.f \times 2^{e-127}.$$

Разница в том, что слева от дробной части у нас 0, а не 1. В таком случае мы говорим, что число не нормировано. Третий случай — когда $e = 255$, а $f = 0$; тогда число означает бесконечность.

Если $s = 1$, у нас $-\infty$, если же $s = 0$, то у нас $+\infty$. И последний случай — когда $e = 255$, а $f \neq 0$; тогда мы расшифровываем значение как «нечисло», или NaN*, которое обозначает неизвестное значение, или недопустимый результат вычислений.

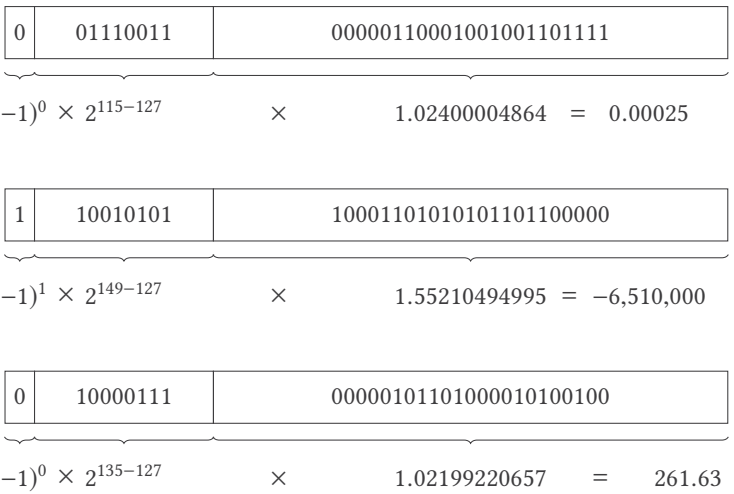


Рисунок 13.7. Примеры представления чисел с плавающей запятой

Таким образом, мы возвращаемся к арифметике с плавающей запятой. Все это кажется компьютерным чудом. Возможно, так и есть. На самом деле, полезно знать все случаи, если не можешь отыскать ошибки в программе, которая выдает совершенно непонятные числовые результаты. Также обратите внимание, что, хоть у нас и есть представление для бесконечности, оно актуально только для расчетов с плавающими числами. Для целых чисел подобного эквивалента не существует.

Теперь же пойдем обратно к хешированию. Число 261,63 представлено на компьютере как 01000011100000101101000010100100. Мы обходимся с данной последовательностью битов как с целым числом. На секунду мы забываем, что у нас там есть плавающая запятая, и ведем себя так, будто у нас целое число тоже набором битов.

* Not a Number.

Мы имеем:

$$(01000011100000101101000010100100)_2 = (1132646564)_{10}.$$

Мы просто берем 1132646564 в качестве нашего ключа для хеширования — мы используем нотацию $(X)_b$, чтобы показать, что число X находится системе счисления b. Вкратце, каждое число с плавающей запятой мы можем интерпретировать как целое число и как обычно использовать алгоритм 13.3. Это очень выгодно, так как нам не нужно делать никаких преобразований, мы просто обходимся тем же набором битов по-другому.

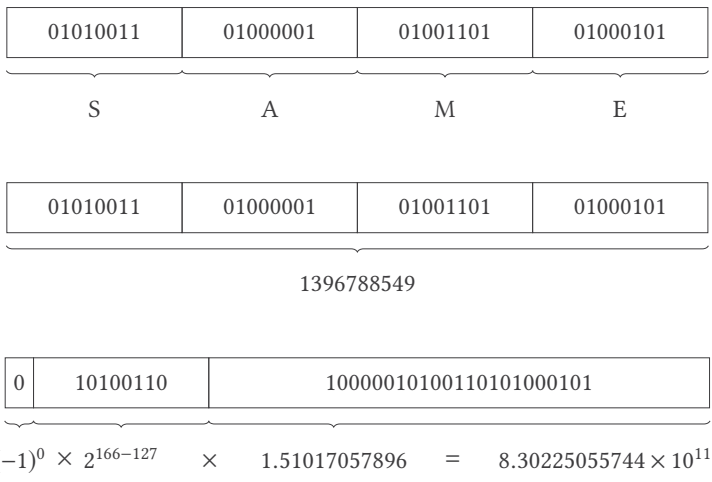


Рисунок 13.8. Расшифровка одной и той же части бита: строка ASCII, целое число и действительное число

Это приводит нас к важному моменту, касающемуся компьютеров в целом. Компьютерам известны только биты: нули и единицы, — и они не имеют ни малейшего понятия, что эти нули и единицы обозначают. Набор битов в памяти компьютера может обозначать что угодно. Понять, что же они значат, и найти им верное применение — это дело программ. На рисунке 13.8 показано, как один и тот же набор битов может обозначать три разные вещи: его можно интерпретировать как строку символов, как целое число или как число с плавающей запятой. Не стоит обращаться с наборами битов не по назначению.

Впрочем, бывают исключения, как, например, то, к которому мы прибегли совсем недавно; однако помните, что правило гласит иное.

13.5. Коллизии

Мы обзавелись надежным семейством хеш-функций для цифровых ключей и строк символов. Больше нам ничего не надо? Давайте снова посмотрим на таблицу 13.3 и представим, что мы добавили в нее еще одну из самых населенных стран мира — Мексику с международным кодом 52. Получается конфликт с Бангладешем: 52 mod 23 = 880 mod 23 = 6 — из-за чего ключ Мексики оказывается на том же месте в таблице, что и ключ Бангладеша, как изображено на рисунке 13.9.

Согласно принципу Дирихле, без коллизий не обойтись. Хорошая функция хеширования сократит количество коллизий до минимума, но никакая функция не сможет избежать их совсем. Поэтому, чтобы хеширование работало, нам нужен способ справиться со встречающимися коллизиями.

Самый распространенный метод — упорядочить слоты хеш-таблицы так, чтобы она не содержала простые записи, а указывала на списки записей. Каждый слот таблицы будет указывать на список, в котором содержатся все записи, чьи ключи в ходе хеширования получают одинаковые значения; то есть в нем будут находиться все записи, чьи ключи конфликтуют друг с другом. Если конфликтов нет, то список будет содержать один элемент. Если нет ключей, которые в ходе хеширования отправляются в слот, то слот будет пустым списком, указывающим на `null`. Мы часто прибегаем к термину «цепочка» для обозначения списка записей, которые в ходе хеширования получили одинаковые значения. Мы используем по цепочке для каждого хеш-значения, поэтому данная тактика зовется методом цепочек. На рисунке 13.10 показано, как он разрешает коллизию Бангладеша и Мексики. Чтобы обозначить указатель, ведущий к `null`, мы используем символ Ø. Так как мы хотим, чтобы операция выполнилась как можно быстрее, мы обычно прибегаем к простому односвязному

неупорядоченному списку. Элементы добавлены во главу списка, как и в случае с Мексикой и Бангладешем.

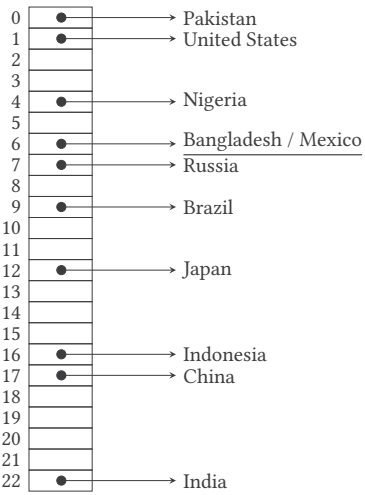


Рисунок 13.9. Хеш-таблица размером 23, содержащая первые 11 из самых населенных стран мира, их ключами являются международные коды

В таком случае мы можем добавить в нашу таблицу и другие страны. Часть из них отправятся к цепочкам с одним элементом, в то время как еще одна часть вступит в конфликт с другими странами в хеш-таблице, так что они отправятся в цепочку, в которой находятся все страны, чьи ключи получили в ходе хеширования одинаковые значения. Если у нас в общей сложности 17 стран, как на рисунке 13.11, у нас образуются цепочки с одним элементом, цепочки с двумя элементами и даже цепочка с тремя элементами, потому что международные коды Ирана, Мексики и Бангладеша получают в ходе хеширования одно и то же место в хеш-таблице.

Задача хеш-таблицы — обеспечить возможность быстро добираться до нужных элементов. Первый этап в извлечении элемента — подсчет хеш-значения его ключа. Затем мы смотрим в хеш-таблицу, на позицию, указанную хеш-значением. Если оно указывает на `null`, то мы знаем, что этого элемента в хеш-таблице нет. Если оно указывает на список, тогда мы просматриваем список,

пока не найдем нужный элемент или же не достигнем хвоста списка, так и не найдя нужного элемента.

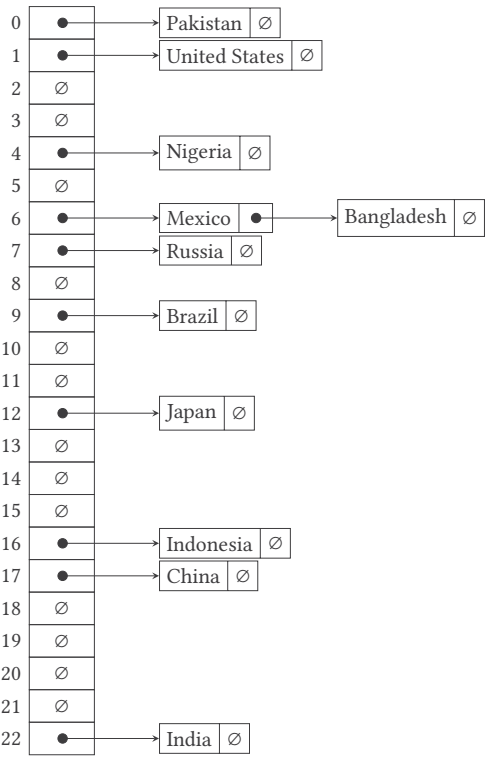


Рисунок 13.10. Хеш-таблица размером 23, содержащая первые 11 из самых населенных стран мира, их ключами являются международные коды, распределенные методом цепочек

Взгляните на рисунок 13.11: если нам дают ключ 880, мы вычисляем его хеш-значение, 6, а затем проходимся по списку, указанному в шестой позиции хеш-таблицы. Мы сравним ключ каждого элемента в списке с 880. Если нашли его, то мы знаем, что это Бангладеш. Такое случится после проверки трех элементов, потому что Бангладеш стоит в конце списка. Теперь представим, что мы ищем страну с кодом 213, то есть Алжир. Число 213 после хеширования также получает значение 6. Мы пройдемся по списку, но не найдем там ни одного элемента с ключом 213, так

что мы можем спокойно констатировать, что Алжира в нашей таблице нет.

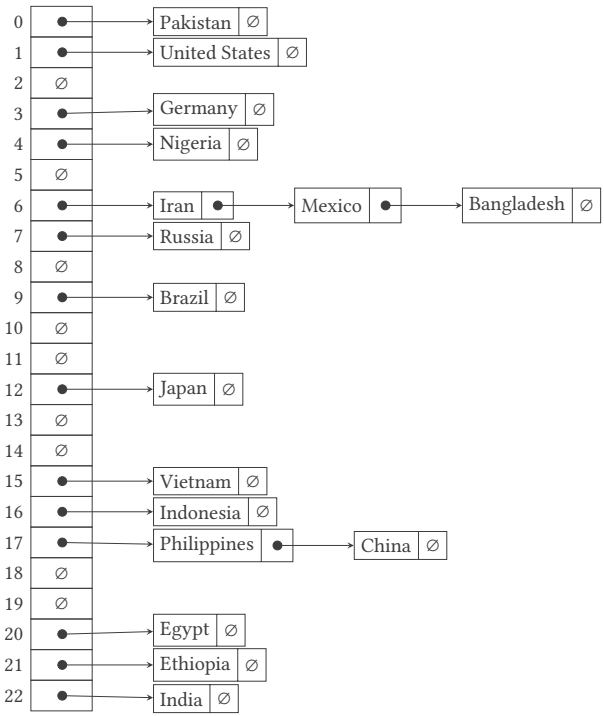


Рисунок 13.11. Хеш-таблица размером 23, содержащая первые 17 из самых населенных стран мира, их ключами являются международные коды, распределенные методом цепочек

Алгоритм 13.7 занимается добавлением в хеш-таблицу с цепными списками. Для записи x нам нужна функция `Key`(x), которая получает для x ключ. Функция `Hash`(k) вычисляет хеш-значение ключа k и может быть любой из функций хеширования, которые мы уже видели: в зависимости от типа ключа. В хеш-таблице находятся списки элементов, поэтому, если нашу хеш-таблицу назвать T, то $T[h]$ будет списком, в котором содержатся элементы с ключами, чье хеш-значение равно h.

Поиск элемента в хеш-таблице начинается так же, как и добавление; взгляните на алгоритм 13.8. Затем мы находим слот, в кото-

ром должен находиться список с искомым элементом, просто ищем список в слоте. `SearchInList`(L, i) ищет элемент i в списке L, поэтому нам на самом деле нужно вызвать `SearchInList`($T[h]$, x). Вспомните, что функция `SearchInList` возвращает искомый элемент или же `null`, если нужного элемента нет в списке. Если нам надо убрать элемент из хеш-таблицы, мы снова прибегаем к модификации, как в алгоритме 13.9, используя функцию `RemoveFromList`(L,d), которая убирает из списка первый узел, содержащий d, и возвращает его, если d находится в списке, в противном же случае она возвращает `null`.

Алгоритм 13.7. Добавление в хеш-таблицу с цепными списками

```
InsertInHash(T, x)
    Вводные данные: T, хеш-таблица; x, запись,
    которую нужно добавить в хеш-таблицу.
    Результат: x добавлена в T.
1  h ← Hash(Key(x))
2  InsertInList(T[h], NULL, x)
```

Алгоритм 13.8. Поиск по хеш-таблице с цепными списками

```
SearchInHash(T, x) → TRUE or FALSE
    Вводные данные: T, хеш-таблица; x, запись, которую нужно найти
    в хеш-таблице.
    Выводимые данные: true, если найдена, и false, если нет.
1  h ← Hash(Key(x))
2  if SearchInList(T[h], x) = NULL then
3      return FALSE
4  else
5      return TRUE
```

Из алгоритма 13.8 следует, что затраты на поиск элемента с помощью хеширования методом цепочек зависят от элемента. Сначала нам надо принять в расчет затраты на хеш-функцию. Если мы хешируем цифровые ключи, то затраты те же, что и у алгоритма 13.3, и равны времени, которое требуется на операцию деления. Мы принимаем их за константу, $O(1)$. Если мы хешируем строки

символов, то затраты те же, что у алгоритма 13.6, то есть $\Theta(n)$, где n — это длина строки. Так как алгоритм работает быстро и зависит только от ключа, а не от количества элементов, которые мы хешируем, мы также принимаем его затраты за константу. К ней мы добавляем затраты на операции, нужные после нахождения хеш-значения. Они могут быть равны $O(1)$, если слот указывает на `null`, или $O(|L|)$, где $|L|$ — это длина списка L, потому что поиск по списку занимает линейное время, так как мы поочередно переходим от одного элемента к другому. Возникает вопрос: насколько длинным может быть список?

Алгоритм 13.9. Извлечение из хеш-таблицы с цепными списками

`RemoveFromHash(`T`,` x`)` $\longrightarrow$ x or NULL

Вводные данные: T, хеш-таблица; x, запись, которую нужно извлечь из хеш-таблицы.

Выводимые данные: x, запись x, если она была извлечена, или `null`, если x не было в хеш-таблице.

```
1  h ← Hash(Key(x))
2  return RemoveFromList(T[h], x)
```

Ответ зависит от конкретной функции хеширования. Если хеш-функция хорошая и распределяет ключи по хеш-таблице равномерно, без каких-либо перекосов, тогда мы ожидаем, что длина каждого списка равно примерно n/m, где n — число ключей, а m — размер хеш-таблицы. Число n/m называется коэффициентом заполнения таблицы. При неудачном поиске нам нужно время на вычисление ключа и $\Theta(n/m)$, чтобы дойти до конца списка. При успешном поиске нам надо идти по элементам списка до тех пор, пока не найдем тот, что искали. Так как элементы добавляются в начало списка, мы должны перебрать элементы списка, которые были добавлены туда после элемента, который мы ищем. Получается, что время на поиск по списку равно $\Theta(1 + (n - 1)/2m)$.

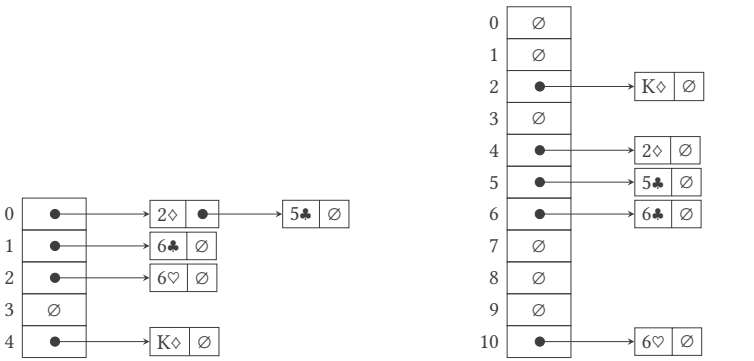
Мы видим, что время, требуемое на успешный и на неудачный поиски, зависит от n/m. Если количество ключей пропорционально размеру таблицы, то есть $n = cm$, тогда время на поиск по спи-

ску становится постоянным. В самом деле, при неудачном поиске у нас $\Theta(n/m) = \Theta(cm/m) = \Theta(c) = O(1)$. А успешный поиск не может длиться дольше неудачного, поэтому ему опять же требуется $O(1)$ времени. И в том и в другом случаях мы получаем постоянное время.

Постоянное время выполнения — значительное преимущество, поэтому хеш-таблицы являются очень популярным механизмом хранения данных. Впрочем, следует дать пару важных пояснений. Во-первых, элементы в хеш-таблицах неупорядочены. Если хеш-функция хороша, то записи добавляются в таблицу как придется, на случайные позиции. Так что, если мы хотим провести поиск в определенном порядке (скажем, числовом или алфавитном), когда мы находим элемент, а затем нам нужен следующий за ним по порядку, то хеш-таблицы нам не друзья. Во-вторых, работа хеш-таблицы зависит от коэффициента заполнения. Как мы видели, время на поиск, кода $n = cm$, связано с c; поэтому хорошо бы знать наверняка, что c не будет слишком большим. Например, если мы собираемся добавить n элементов, можно создать хеш-таблицу размером $2n$ и заранее знать, что поиску потребуется не более двух сравнений. Но что происходит, когда мы не знаем заранее количество элементов или же наши предположения оказываются ошибочны? Когда такое случается, мы меняем размер таблицы. Мы создаем новую, более вместительную хеш-таблицу, забираем все элементы из перегруженной хеш-таблицы и вставляем их в только что созданную, большую, затем удаляем старую таблицу и с этих пор пользуемся новой. Смена размера — дорогостоящая в вычислительном отношении операция, так что стоит заранее оценивать объем элементов, чтобы не пришлось к ней прибегать. С другой стороны, большинство реализаций хеш-таблиц отслеживают собственную загруженность и, достигая определенного порога, например 0,5, они меняют свой размер. В таком случае поиск по хеш-таблице останется постоянным, однако добавление будет медленным, когда дело коснется элемента, из-за которого сменился размер хеш-таблицы.

На рисунке 13.12 показан пример смены размера. Изначально у нас есть хеш-таблица размером 5, и мы добавляем в нее

записи, чьими ключами являются игральные карты. У каждой карты есть свое числовое значение: у туза треф — 0, а далее идем по возрастающей и получаем короля треф со значением 12; затем бубнового туза со значением 13, червового туза — 25 и пикового туза — 38; самое большое значение у пикового короля — 51. На рисунке 13.12(a) изображены таблица размером 5 и коэффициентом загрузки 1. Мы решаем увеличить таблицу. Чтобы у нас было простое число для деления по модулю, мы берем не 2 × 5, а 11. Мы переносим карты, как показано на рисунке 13.12(b); все новые карты будут добавлены в новую, увеличенную таблицу.



(a) Хеш-таблица размером 5 (b) Хеш-таблица размером 11

Рисунок 13.12. Смена размера хеш-таблицы

Важно не забывать, что работа хеш-таблиц вероятностная. При хорошей хеш-функции вероятность того, что поиску понадобится более n/m, крайне мала, но все же она существует. Мы не можем гарантировать, что каждая цепочка окажется короткой, но в среднем вероятность того, что она превысит средний размер, невелика. К тому же, когда мы прибегаем к смене размера, производительность во время смены размера падает.

Хеш-таблицы разнообразны и применяются во многих случаях. Их вероятностное поведение дает нам возможность соотносить время и скорость, позволяя совершенствовать приложения, которые используют хеш-таблицы. Если у нас нет проблем с местом, но есть проблемы со скоростью, мы просто создаем насколько можно большую хеш-таблицу. Если же нет лишнего ме-

ста, то приходится идти на компромисс со скоростью, запихивая больше ключей в таблицу с небольшим размером.

Так как содержимое хеш-таблиц неупорядочено, чаще всего к ней прибегают для реализации структур данных, в которых нам нужно лишь найти и отсортировать что-либо, но не по порядку. Одна из таких структур данных — множество. Оно содержит элементы, и элементы должны быть уникальными. Хеш-таблицы выполняют эти требования тут же: если мы задаем хеш-функцию для элементов, то мы просто вставляем элементы в хеш-таблицу, если их там еще нет. Если нам надо проверить, существует ли элемент во множестве, нам лишь надо проверить, есть ли он в хеш-таблице, для чего мы применяем к элементу хеш-функцию. Множество не упорядочено: элементы просто добавляются в хеш-таблицу, а как мы знаем, для нее нет никакой нотации, согласно которой определялся бы порядок их размещения. Списки, возникающие из каждого слота, упорядочены, однако не соответствуют ни одному правилу сортировки, скажем, алфавитному порядку или числовому, для самих элементов. Но это не важно, так как множества с математической точки зрения все равно неупорядочены.

Еще одна известная структура данных, которая хорошо уживается с хеш-таблицами, — словарь, ее еще называют картой или ассоциативной матрицей. Словарь работает с парами ключ-значение, как и обычный словарь, в котором содержатся термины (ключи) и определения (значения). С помощью ключа мы достаем соответствующее значение, как и в привычных словарях, когда находим определение для нужного нам слова. Однако в отличие от обычных словарей в этих нет никакого порядка, потому что хеш-таблицы, лежащие в их основе, ничего не упорядочивают. Привычные нам словари структурированы, что облегчает нам поиск; хеш-таблицам же никакой поиск вообще не нужен. Словарь похож на множество, все элементы которого разбиты по парам ключ-значение; мысленно мы можем представить их как массив с двумя элементами. Если нам надо добавить новую пару ключ-значение, мы хешируем ключ и добавляем пару ключ-значение в хеш-таблицу, как показано в алгорит-

ме 13.10. Обратите внимание, что сперва нам надо проверить, есть ли уже такая пара ключ-значение в словаре или нет. Функция `SearchInListByKey`($T[h]$, k) просматривает список в $T[h]$ в поисках элемента, который является парой ключ-значение с ключом k, и возвращает его, когда находит, или же возвращает `null`, если такого элемента нет. Она несколько отличается от функции `SearchInList`, которая ищет определенный элемент, сравнивая его с остальными. `SearchInListByKey` ищет определенный элемент, сравнивая его с ключами остальных элементов. Это общая закономерность: поисковые функции в структурах данных часто берут особый параметр, определяющий идентичность при сравнении, скажем, используют определенный атрибут элемента или комбинацию атрибутов, а не весь элемент целиком.

Алгоритм 13.10. Добавление в словарь (карту)

`InsertInMap`(T, k, v)

Вводные данные: T, хеш-таблица; k, ключ из пары ключ-значение; v, значение из пары ключ-значение.

Результат: значение v добавлено в словарь, соотнесенный с ключом k.

```
1  h ← Hash(k)
2  p ← SearchInListByKey(T[h], k)
3  if p = NULL then
4      p ← CreateArray(2)
5      p[0] ← k
6      p[1] ← v
7      InsertInList(T[h], NULL, p)
8  else
9      p[1] = v
```

Если элемент не обнаружен, тогда мы добавляем его в строчке 7 с помощью вызова функции `InsertInList`($T[h]$, `null`, p). Если элемент найден, мы пересматриваем словарь в поисках k, приписывая второму элементу существующей пары ключ-значение новое значение.

Если нам нужно достать значение, соотнесенное с ключом в хеш-таблице, мы хешируем ключ, смотрим пару ключ-значение,

хранящуюся в хеш-таблице, и если находим ее, то возвращаем значение; посмотрите алгоритм 13.11.

Алгоритм 13.11. Поиск по словарю (карте)

`Lookup(`T`, `k`)` $\longrightarrow v$ or NULL

Вводные данные: T, хеш-таблица; k, ключ.

Выводимые данные: v, соответствующее значение, если оно существует, или `null`, если его нет.

```
1  h ← Hash(k)
2  p ← SearchInHash(T[h], k)
3  if p = NULL then
4      return NULL;
5  else
6      return p[1]
```

Алгоритм 13.12. Удаление из словаря (карты)

`RemoveFromMap(`T`, `k`)` $\longrightarrow [k, v]$

Вводные данные: T, хеш-таблица; k, ключ для извлечения соответствующей пары ключ-значение из словаря.

Выводимые данные: $[k, v]$, пара ключ-значение, соответствующая k, если она была извлечена, или `null`, если соответствующей пары ключ-значение в словаре не нашлось.

```
1  h ← Hash(k)
2  return RemoveFromListByKey(T[h], k)
```

Наконец, если мы хотим удалить пару ключ-значение из словаря, то хешируем ключ и удаляем пару ключ-значение с соответствующим ключом из хеш-таблицы; посмотрите алгоритм 13.12. Мы прибегаем к функции `RemoveFromListByKey`(L, k), которая удаляет пару ключ-значение с ключом k из списка L и возвращает ее, если она существует, или же возвращает `null`, если ее нет. `RemoveFromListByKey` похожа на `RemoveFromList` точно так же, как `SearchInListByKey` похожа на `SearchInList`: она находит то, что надо удалить, путем поиска в списке элемента, сравнивая заданный ключ.

Описанное трио алгоритмов не сильно отличается от обычной работы с элементами хеш-таблиц, которую мы видели выше.

13.6. Цифровые отпечатки

Принцип хеширования лежит в основе приложений, которые имеют дело с идентификацией фрагментов данных. Если у вас есть запись о человеке и в ней содержится несколько атрибутов, то эту запись легко опознать: вам нужно лишь проверить, есть ли у вас запись с такими же ключами. А теперь представьте, что ваша «запись» не вписывается в знакомый шаблон атрибутов и ключей. Она может представлять собой изображение или музыкальную композицию, скажем, песню. Вам хочется узнать, попадалась ли вам эта песня раньше. Нет никаких атрибутов, связанных с песней. Название, исполнитель и композитор — всего этого в самой песне нет: когда перед вами данные с песней, у вас есть лишь последовательность звуков, из которых она складывается. Получается, что ваша песня — это просто набор звуков. Как их искать?

Нужно извлечь из аудиозаписи нечто уникальное, какую-то особенность, что можно было бы использовать как ключ к ее распознаванию. На секунду возьмем другой пример и представим, что нам надо установить личность человека. Вычислить неизвестного человека непросто, равно как и сравнить его с группой уже известных фигурантов. Вам нужно опираться на какую-то уникальную черту, сравнивая ее с уникальными чертами других людей. Уже двести лет в качестве такой неповторимой характеристики мы используем отпечатки пальцев. У нас есть отпечатки тех, кого мы знаем, и мы также получаем отпечатки человека, которого хотим опознать. Если мы можем сравнить отпечатки неизвестного и известных нам людей, то неопознанный человек перестает быть темной лошадкой: мы установили его личность.

В мире компьютеров такие уникальные следы называются цифровыми отпечатками. Существует несколько особенностей цифровых данных, по которым мы доподлинно можем распознать неизвестные данные. На деле это выглядит так: у вас есть аудио-

запись песни, и вам хочется узнать, как она называется и кто ее исполняет. У вас также есть огромная база данных с песнями, чьи цифровые отпечатки уже сняли. Вы вычисляете цифровой отпечаток вашей аудиозаписи и пробуете найти соответствие в базе данных — в итоге песня находится. Хеш играет тут немаловажную роль, потому что в базе могут быть миллионы песен и миллионы цифровых отпечатков, а вам нужно быстро найти нужное соответствие (или узнать, что такового нет). К тому же в цифровых отпечатках нет нотации распределяющей песни по порядку, так что большая хеш-таблица — идеальный помощник: ключами являются отпечатки, а записями — информация о песнях, то есть название, исполнитель и так далее.

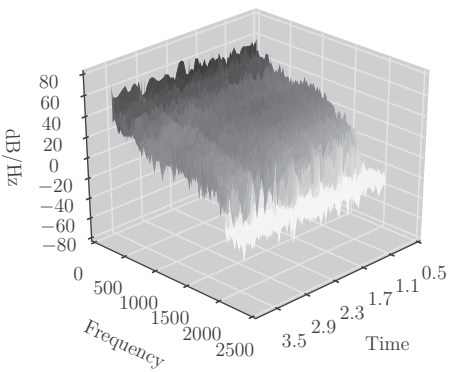


Рисунок 13.13. 3D-представление мощности, частот и времени

Чтобы сделать цифровые отпечатки для звука, нужно припомнить, что звук представляет собой вибрацию, которая распространяется с помощью посредника, чаще всего воздуха. Каждому звуку соответствует одна или больше частот. У чистой ноты одна частота, например частота ноты «ля» — 440 герц (Гц) в равномерно темперированном строе, а частота ноты «до» — 261,63 Гц. В аудиозаписи содержится множество частот, меняющихся со временем в зависимости от комбинации вокала и различных музыкальных инструментов. Более того, одни частоты звучат громче других, когда один инструмент играет громче остальных. Разница в глубине звука — это разница в мощности соответствующих частот.

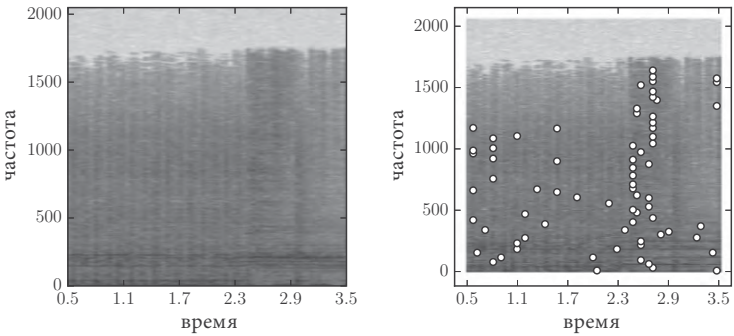
Разные частоты обладают разной мощностью в разный момент времени, на протяжении всей песни некоторые из них становятся громче, некоторые тише, а какие-то и вовсе замолкают.

Мы можем представить песню графически с помощью 3D-графика сил, частот и времени. На рисунке 13.13 представлен такой 3D-график мощности, частот и времени для трехсекундного отрывка песни. Обратите внимание, что стартовая точка (0, 0, 0) находится на самом дальнем ребре и что мы начинаем с 0,5 секунды песни (потому что сначала идет тишина). Рисунок сообщает нам, что у данного фрагмента песни преимущественно низкие частоты, так как в них больше мощности: в каждый момент времени низкие частоты имеют наивысшие значения z. На оси z мы измеряем силу частоты в соотношении децибел (дБ) к герцу. Минус появляется в обозначениях из-за логарифмического децибела. Он представляет соотношение двух значений: $10\log(v/b)$, где b — это основное значение. В нашем случае $b = 1$, поэтому при $v < 1$ мы получаем отрицательные значения.

Вместо использования 3D-представления, как на рисунке 13.13, мы обычно ограничиваемся двумя пространствами, помечая цветом мощность в каждый момент времени и частоты. Такое представление называется спектрограммой. Рисунок 13.14(a) является спектрограммой рисунка 13.13. Скорее всего, вы согласитесь, что данное представление воспринимается куда проще предыдущего, и в то же время вся важная информация сохранена. Фраза «чем меньше — тем лучше» почти всегда актуальна для визуализированных данных.

На спектрограмме видно, что мощность на протяжении времени и частот меняется рыками. Есть места, в которых точка с наивысшей мощностью (темная) граничит с точками низкой мощности (светлыми). То же самое на рисунке 13.13, когда значение достигает пика по сравнению с соседними значениями. Мы можем обнаружить такие пики математически. Если мы отобразим их поверх спектрограммы, то получим рисунок 13.14(b). Оказывается, что комбинация пиков мощности частоты в песне может служить цифровым отпечатком. Следовательно, мы можем взять пики звукового отрывка и посмотреть, соответствуют ли

они пикам уже известной нам песни. Если наш отрывок короче, чем песня, мы попробуем сопоставить пики в отрывке песни той же длины. Когда мы находим соответствие, у нас появляется хороший шанс, что мы отыскали нужную песню.



(a) Спектрограмма (b) Спектрограмма с пиками

Рисунок 13.14. Спектрограмма и пики мощности частоты

На деле иметь одни только частоты в качестве цифровых отпечатков — не лучшая идея. Более точно описать песню может пара пиков и промежуток времени между ними. Если у нас есть пик частоты f_1 в момент времени t_1 и еще один пик частоты f_2 в момент времени t_2, то вместо использования f_1 и f_2 в качестве ключей в хеш-таблице мы создаем новый ключ $k = f_1 : f_1 : t_2 - t_1$, как на рисунке 13.15a. Ключ k может быть представлением строки символов, объединяющей его части, таким как «1620.32:1828.78:350» с разница времени в миллисекунды.

Количество возможных пар (f_x, f_y) в песне может быть огромным, так как подразумевается соединение всех пиков частот между ними. Говоря математическим языком, это эквивалентно всем возможным способам выбора k элементов из n элементов при $k = 2$ и n — числе пиков частоты в песне, есть даже специальное описание:

$$\frac{n!}{k!(n-k)!} = \binom{n}{k}.$$

Таково число возможных комбинаций, то есть выборка без упорядочивания, k элементов из n элементов. Нотация $\binom{n}{k}$ читается как «n выбирает k».

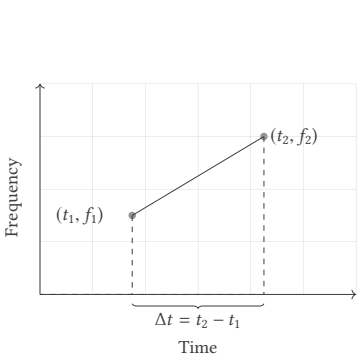
Если формула комбинаций кажется непонятной, подумайте, как мы находим число пермутаций, то есть распределенных выборов k элементов из набора n элементов. Существует n способов выбрать первый элемент, затем для каждого из них есть $n - 1$ способов выбрать второй элемент и так до элемента k, который остался последним, и поэтому есть только один путь выбрать его. В общем и целом у нас произведение $n \times (n - 1) \times \ldots \times (n - k + 1)$ способов выбрать в определенном порядке k элементов из n. Но:

$$n \times (n-1) \times \cdots (n-k+1) = \frac{n!}{(n-k)!}.$$

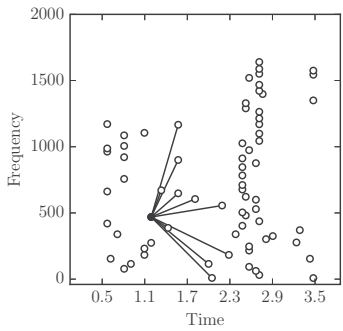
Так как пермутации упорядочены, нам надо разделить это число на количество возможных распределений k элементов. Получается k способов выбрать первый элемент; для каждого из них $k - 1$ способов выбрать второй элемент и так далее до последнего элемента. Так что получается $k \times (k - 1) \times \ldots \times 1 = k!$ возможных распределений k элементов. Так как количество комбинаций равно числу пермутаций, поделенных на количество возможных распределений, мы получаем число для $\binom{n}{k}$.

Вернемся к нашим частотным парам. Значение $\binom{n}{2}$ может привести к большому количеству ключей в хеш-таблице. Если у нас в песне 100 частотных пиков, это будет $\binom{100}{2} = 4950$, поэтому именно для данной песни нам понадобится именно такое количество ключей. Чтобы вписаться в размер хеш-таблицы и избежать перегрузки, мы идем на компромисс и используем только подмножество возможных пар для каждого частотного пика; к примеру, мы можем решить использовать не более десяти пар. Можно установить фиксированное значение, называемое коэффициентом нагрузочной способности, чтобы подобрать лучший вариант с меньшей вместимостью и повышенной скоростью. Большой коэффициент нагрузочной способности требует большей вместимости и оптимизирует точность поиска, хотя времени на поиск ему тоже требуется больше. На рисунке 13.15(b) изображены пары, которые берутся в расчет при вычислении хешей одного пика, когда мы устанавливаем нагрузочную способность

на десять и запрашиваем первые десять пиков, которые следуют за тем, что мы выбрали, по оси времени.



(a) Ключ хеша частот равен f_1: f_2: Δt



(b) Хеширование частот с нагрузочной способностью 10

Рисунок 13.15. Пики хеширования частот

Подытожим сказанное. Чтобы заполнить хеш-таблицу, мы разделяем спектрограмму каждой известной песни и используем в качестве ключа пары частот и разницу времени между ними, взяв за основу подходящий коэффициент нагрузочной способности. Записи, которым приписаны данные ключи, представляют собой сведения о возможных соответствиях, такие как называние песни и ее исполнитель. Когда мы хотим распознать песню, мы берем ее спектрограмму и пары частот, а затем ищем в хеш-таблице их ключ. Песню с наибольшим количеством совпадений мы объявляем максимально похожей на ту, что мы искали.

13.7. Фильтры Блума

Хеш-таблицы прекрасно подходят для хранения и извлечения данных. Порой, если снизить планку требований для них, можно добиться еще более эффективной работы, а также сбереженного места. А точнее, представьте, что нас не интересуют хранение и извлечение данных, нам надо лишь проверить существуют ли данные во множестве, при этом не извлекая их. Более того, представьте, что хоть изредка, но мы можем получить ответ, что

данные находятся во множестве, хотя на самом деле их там нет, но в то же время мы не получим ответа, что во множестве нет искомых данных, хотя они там есть. Мы соглашаемся на ложноположительный результат, то есть на провозглашение чего-либо верным, хотя оно ложно, однако мы не согласны на ложноотрицательный результат, когда что-либо объявляется ложным, хотя на самом деле это не так.

Подобные ситуации возникают при разных обстоятельствах. Когда у нас есть большая база данных, то извлечение из нее данных потребует некоторых затрат. Если можно мгновенно проверить, есть ли нужные нам данные в базе, то мы избавлены от ненужного поиска, который может оказаться напрасным, если искомых данных в базе нет. Ложноположительный результат приемлем, так как мы соглашаемся на лишние затраты, когда ищем что-то, чего даже не существует, однако в этом нет ничего страшного, если ложноположительные результаты случаются редко.

Механизмы кеширования работают схожим образом. Кеш — это временное хранилище данных, которые часто используются. Можно хранить данные в разных механизмах запоминания, однако некоторые данные используются чаще остальных. Мы помещаем эти группы данных в скоростное хранилище информации, например в оперативную память. Когда нам нужно получить какие-то данные, мы первым делом проверяем, есть ли они в кеше. Если есть, то мы возвращаем их из нашего хранилища. Если же их там нет, то нам придется доставать их из места, где хранится весь объем данных. Ложноположительный результат предполагает, что мы поверим в то, что данные находятся в кеше. Тут нет ничего криминального, так как мы сразу же узнаем, что ошиблись, и обращаемся к памяти большого объема. Такое случается редко, так что можно не беспокоиться.

Различные фильтры следуют тому же принципу. Используя фильтр, мы хотим заблокировать вредные элементы и пропустить надежные. При настройке антиспама в электронной почте мы отмечаем как спам все письма, приходящие с адресов, о которых нам точно известно — оттуда приходит только реклама и прочий мусор, и ложноотрицательный результат, когда хоть одно ре-

кламное письмо да проскочит, нас точно не устроит. Однако мы принимаем низкую вероятность ложноположительного результата, когда нужное письмо вдруг оказывается в папке со спамом. То же самое с сокращенными URL. Компания, предоставляющая услуги сокращения URL, должна гарантировать, что сокращенные URL не ведут на вредоносные сайты; следовательно, когда сокращенный URL готов, работникам компании нужно проверить, не отправляет ли адрес на дурной сайт. Ложноотрицательные результаты для компании неприемлемы, ни один зловредный сайт не должен проскочить. В то же время компания может отнестись с пониманием, если какой-то доброкачественный URL оказался отмечен как вредоносный, но только в том случае, если вероятность такой ошибки крайне мала. В цифровой криминалистике есть набор инструментов для вредоносных файлов. Мы разбиваем такие файлы на блоки и добавляем их в наш фильтр, затем, когда нам надо просканировать наш диск на предмет злонамеренной активности, мы берем блоки нашего хранилища данных и проверяем, есть ли они в нашем фильтре.

Алгоритм 13.13. Добавление в фильтр Блума

`InsertInBloomFilter(`T, x`)`

Вводные данные: T, битовый массив размером m; x, запись для добавления во множество, представленное фильтром Блума.

Результат: запись добавлена в фильтр Блума путем установки битов $h_0(x)$, $h_1(x)$, …, $h_{k-1}(x)$ на 1.

```
1  for i ← 0 to k do
2      h ← h_i(x)
3      T[h] ← 1
```

Продуктивный и популярный способ реализовать подобного рода схему, когда ложноположительные результаты нам не помеха, — прибегнуть к фильтрам Блума, названным в честь их изобретателя, Бертона Блума. Фльтр Блума представляет собой огромный массив T, состоящий из битов, размером m. Изначально все биты устанавливаются на ноль, так как фильтр пуст. Каждый раз, когда нам надо добавить элемент ко множеству, мы хешируем

элемент с помощью k независимых хеш-функций; под «независимыми» мы подразумеваем, что хеш-значения, которые они выдают для одного и того же ключа, никак не связаны с другими функциями. У каждой из этих функций хеширования должен быть диапазон от 0 до m – 1. Для каждого получаемого значения мы устанавливаем соответствующий бит из массива с битами на 1. Когда нам нужно проверить, находится ли элемент в фильтре, мы хешируем его с помощью k функций и смотрим, все ли k хеш-значений в массиве с битами установлены на 1. Если все, мы полагаем, что элемент попал в фильтр; существует вероятность, что он на самом деле не в фильтре, так как всем соответствующим битам назначены другие элементы, уже находящиеся в фильтре. Такая вероятность допустима до тех пор, пока она остается низкой. Если же не все k хеш-значений установлены на 1, тогда мы точно знаем, что элемент не в фильтре.

Алгоритм 13.14. Проверка наличия в фильтре Блума

`IsInBloomFilter(`T, x`)` → TRUE or FALSE

Вводные данные: T, битовый массив размером m; x, запись для проверки, находится ли она во множетсве, представленном фильтром Блума.

Выводимые данные: `true`, если x находится в фильтре Блума, или `false`, если нет.

```
1 for i ← 0 to k do
2     h ← h_i(x)
3     if T[h] = 0 then
4         return FALSE
5 return TRUE
```

Если говорить более конкретно, у нас есть k независимых хеш-функций h_0, h_1 … h_{k-1}, у каждой диапазон от 0 до m – 1. Если нам надо добавить элемент x в фильтр, мы вычисляем $h_0(x)$, $h_1(x)$, …, $h_{k-1}(x)$ и устанавливаем $T[h] = 1$ для всех $h = h_i(x)$, где $i = 0, 1, \ldots, k - 1$; взгляните на алгоритм 13.13. Если нам надо проверить, есть ли элемент x в фильтре Блума, мы снова рассчитываем $h_0(x)$, $h_1(x)$, …, $h_{k-1}(x)$. Если $T[h] = 1$ для всех $h = h_i(x)$, где $i = 0, 1, \ldots, k - 1$, то мы говорим, что элемент находится в фильтре. Если же нет, то мы говорим, что его там нет; смотрите алгоритм 13.14.

Когда мы начинаем работу с фильтром Блума, все его элементы обнулены. Затем по ходу заполнения множества некоторые элементы устанавливаются на 1. В самом начале на рисунке 13.16 у нас есть пустой фильтр с таблицей из 16 битов, в которую мы добавляем «в», «данной», «работе», «анализируется» («In» «this» «paper» «trade-offs»). Каждый из первых трех элементов хешируется в два пустых табличных элемента. Затем «анализируется» хешируется в один элемент, который уже установлен на 1, и один пустой. Такой частичный конфликт в фильтре Блума не берется в расчет, поэтому, если бы до того, как добавить «анализируется» в фильтр, мы проверяли, есть ли она там, то получили бы ложное значение, `false`. Именно поэтому мы используем более одной хеш-функции: мы подозреваем, что вряд ли случится так, что все хеш-значения элемента попадут на уже установленные элементы фильтра. Тем не менее вы можете спросить: насколько хватит такой стратегии? Ведь если мы применяем много функций хеширования, тогда фильтр Блума станет наполняться быстро и, когда установится большинство элементов, у нас пойдут конфликты. Хороший вопрос. Мы посмотрим, как найти оптимальное количество хеш-функций для нашего фильтра.

Если продолжим добавлять в фильтр элементы, то получим конфликт, ситуацию, когда сообщается, что элемент уже находится во множестве, хотя его там нет. Такое утверждение, что нечто верно, хотя оно на самом деле ложно, и есть ложноположительный результат. В частности, мы получим такой результат, когда алгоритм 13.14 вернет `true` для записи *x*, хотя на самом деле запись *x* не в фильтре. Что и происходит на рисунке 13.17, когда мы добавляем элементы в фильтр Блума. После добавления «взаимосвязи» («among») мы обнаруживаем, что фильтр обходится со словом «между» («certain»), как если бы оно уже находилось в нем.

Может показаться, что это сущий кошмар, однако все не так страшно. Если наша область применения подразумевает наличие отрицательно-положительных результатов и мы уже видели примеры подобных использований, то такое поведение не будет нас смущать. А вот что мы получим взамен — действительно непростой вопрос.

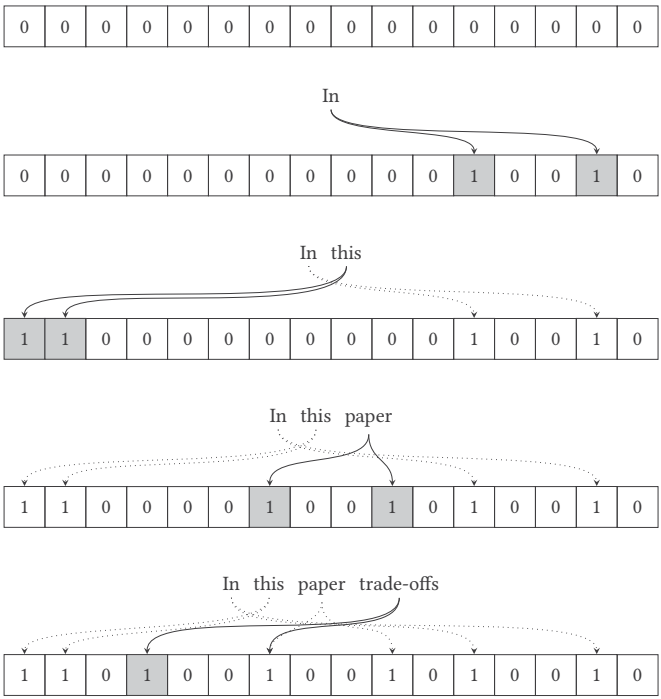


Рисунок 13.16. Добавление в 16-битную таблицу «В», «данной», «работе», «анализируется» с помощью фильтра Блума

В нашем примере мы успели столкнуться с ложноположительным результатом, но при этом успешно добавили во множество пять элементов. При использовании 16-битной таблицы у нас 1/6 ≈ 17% ложноположительных результатов. Теперь, внимание, ключевой момент: *на самом деле мы никуда не помещаем элементы нашего множества*. Мы всего лишь обозначаем элементы в нашей хеш-таблицы единицами. Фильтр Блума позволяет нам представить множество, не требуя перемещения элементов в это самое множество. Если бы мы прибегли к обычной хеш-таблице для представления установленных слов, то нам бы понадобилось место для таблицы плюс место для хранения строк символов, которые соответствуют словам. Слова в нашем примере* занимают

* В английском варианте «In this paper trade-offs among certain».

41 байт, что равно 328 битам, если на каждую букву приходится по одному биту. Мы получаем примерно 328/16 ≈ 20 места, когда мы соглашаемся на ложноположительные результаты.

Мы использовали простейший фильтр Блума из 16 бит для группы слов, прибегнув к двум хеш-функциям. Если бы дело коснулось реальной практики, мы бы воспользовались фильтром побольше, чтобы справиться с бóльшим объемом элементов, и, возможно, нам понадобилось бы более двух хеш-функций. Число m бит фильтра (его размер), число n элементов, с которыми мы имеем дело, и число k хеш-функций — все эти параметры нам нужно сбалансировать так, чтобы добиться желаемого поведения.

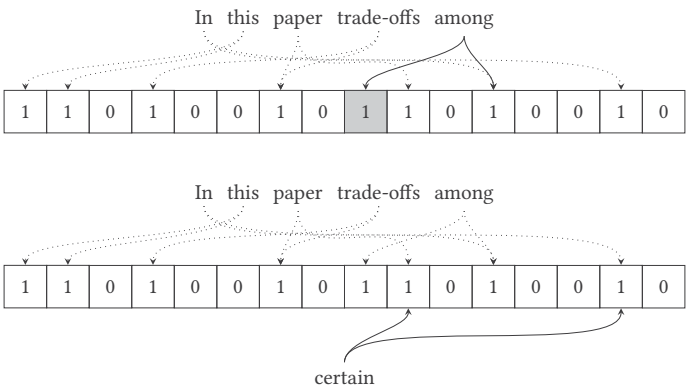


Рисунок 13.17. Добавление слова «взаимосвязи», ложноположительный результат для «анализируются»

Если у нас есть только одна хеш-функция, тогда вероятность того, что конкретный бит в таблице будет размещен, когда мы хешируем элемент, равна $(1/m)$, поэтому вероятность, что этот конкретный бит не будет размещен, равна $(1 - 1/m)$. Если мы используем k независимых хеш-функций, то вероятность, что конкретный бит не будет размещен, равна $(1 - 1/m)^k$. Мы может записать это как $(1 - 1/m)^{m(\frac{k}{m})}$. Когда m достаточно большое, из расчетов мы получаем, что $(1 - 1/m)^m = 1/e = e^{-1}$, поэтому вероятность, что конкретный бит не будет размещен, равна $e^{k/m}$. Если в нашем фильтре Блума содержится n элементов, тогда вероятность, что

бит не будет размещен, равна $e^{kn/m}$. И наоборот, вероятность, что бит будет размещен, равна $(1 - e^{-kn/m})$. Вероятность, что мы получим ложноположительный результат, равна вероятности, что после добавления n элементов мы проверяем элемент, который не в фильтре, а все k хеш-значения этого элемента находятся в занятых позициях. Она равна вероятности, что k конкретных битов размещены: $p = (1 - e^{-kn/m})^k$.

Из последнего выражения мы можем вычислить оптимальные значения для параметров фильтра Блума. Если у нас есть m (размер фильтра) и n (количество элементов), то снова с помощью расчетов мы можем получить наилучшее значение для k, как $k = (m/n)$ ln2. Например, если мы имеем дело с одним миллиардом элементов и используем фильтр Блума из 10 миллиардов бит, равных 1,25 гигабайта, нам нужно воспользоваться $k = (10^{10}/10^9)/\ln 2 \approx 7$ различными, независимыми хеш-функциями. С такими параметрами, так как $n/m = 10^9/10^{10} = 1/10$, вероятность ложноположительного результата равна $\left(1 - e^{-7/10}\right)^7 \approx 0.008$, или 8‰: обратите внимание, что тут обозначение не в процентах, а промилле, на тысячу. Мы можем графически изобразить функцию $\left(1 - e^{-k/10}\right)^k$ для различных значений k, как на рисунке 13.18. На графике показано, что даже с пятью хеш-функциями можно добиться менее 1% ложноположительных результатов.

Если мы берем $k = (m/n)\ln 2$ и подставляем в $p = (1 - e^{-kn/m})^k$, то, если говорить вкратце, мы получаем фильтр Блума размером m, вероятность ложноположительного результата у которого равна p при использовании k хеш-функций: $m = -n \ln p/(\ln 2)^2$. К примеру, если нам надо разобраться с одним миллиардом элементов при вероятности ложноположительного результата в 1%, мы получаем $m \approx 9\,585\,058\,378$, что примерно 10 миллиардов битов, или 1,25 гигабайтов, как и в нашем примере. Если средний размер элемента равен десяти байтам, тогда с 1,25 гигабайтами мы можем обработать 10 гигабайтов данных с вычислительной стоимостью нескольких хеш-функций!

Классические фильтры Блума, как мы их описываем, значительно экономят на затратах, однако помимо ложноположительных результатов имеется еще один минус, который ограничивает

область их применения. Элемент из фильтра Блума невозможно удалить. Если у вас два элемента и оба заняли один и тот же бит, то, если мы вернем в исходное положение биты, которые соответствуют одному из элементов, мы ненамеренно вернем в исходное положение и биты, которые соответствуют второму элементу. На рисунке 13.19 слова «работе» и «анализируются» хешируются на одну и ту же позицию фильтра Блума, $T[6]$. Если мы хотим удалить из фильтра одно из них, мы освобождаем позицию $T[6]$, которая в то же время должна быть назначена, так как нужна другому элементу.

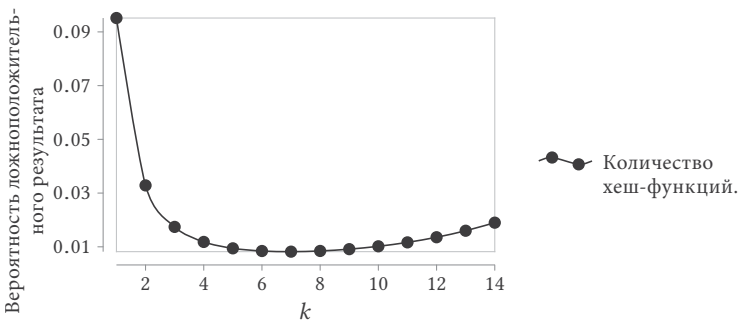


Рисунок 13.18. Вероятность ложноположительного результата в фильтре Блума в виде функции хеш-функций, k, при $n/m = 10$

Чтобы разрешить данную неприятность, нужно отказаться от использования битового массива, как хеш-таблица, и взглянуть в сторону счетного массива, например массива целых чисел, начинающегося с нуля. Каждый раз, когда мы добавляем элемент, увеличиваем на один индикаторы, соответствующие хеш-позициям. Чтобы убрать элемент, мы те же самые индикаторы уменьшаем. Проверка наличия элемента в массиве происходит так же, как и прежде: мы проверяем нули во всех хеш-значениях элемента. Мы называем такие фильтры счетными фильтрами Блума. В алгоритме 13.15 — как добавляются элементы, а в алгоритме 13.16 показано, как они удаляются. Проверка наличия в счетных фильтрах Блума происходит точно так же, как в обычных фильтрах Блума.

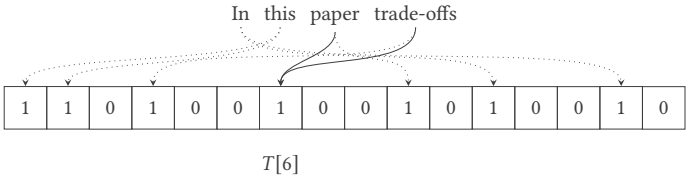


Рисунок 13.19. Проблема удаления фильтрах Блума

Алгоритм 13.15. Добавление в счетный фильтр Блума

`InsertInCntBloomFilter(`T, x`)`

Вводные данные: T, массив размером m, содержащий целые числа; x, запись для добавления во множество, представленное счетным фильтром Блума.

Результат: запись добавлена в счетный фильтр Блума путем увеличения на 1 индикаторов в $h_0(x)$, $h_1(x)$, …, $h_{k-1}(x)$.

```
1 for i ← 0 to k do
2     h ← h_i(x)
3     T[h] ← T[h] + 1
```

Если мы воспользуемся в нашем примере счетным фильтром Блума, то придем к ситуации, как на рисунке 13.20. Мы вставляем элементы путем добавления индикатора, соответствующего каждой хеш-позиции. Когда же придет пора убрать элемент, мы просто уменьшим индикаторы, соответствующие хеш-позиции элемента, который нужно удалить, как показано в последней части рисунка. Введение индикаторов решает нашу проблему, но требует определенных затрат. В обычных фильтрах Блума каждая позиция в таблице занимает место одного бита. В счетных фильтрах Блума каждая позиция в таблице занимает место одного индикатора; он может быть размером с байт или даже больше. Так что наша экономия с местом становится не так эффективна, как с обычными фильтрами Блума.

Остался последний момент, с которым нам надо как-то разобраться. Мы предполагали наличие группы независимых хеш-функций, которые для одинаковых вводных данных выдают набор различных хеш-значений, однако не привели в пример ни одной такой хеш-функции. Наши хеш-алгоритмы берут и выдают

особые данные. К счастью, существуют хеш-алгоритмы, которые могут менять свои выводимые данные в зависимости от параметра, который мы им прописываем, и разнообразие выводимых данных такое, что все они ведут себя как разные, независимые хеш-функции. Именно их мы можем использовать в фильтрах Блума. Алгоритм 13.17 является примером такого алгоритма, в основе которого лежит алгоритм Фаулера/Нола/Во, или, сокращенно, алгоритм FNV, названный в честь его изобретателей Глена Фаулера, Лендона Керта Нола и Фонга Во.

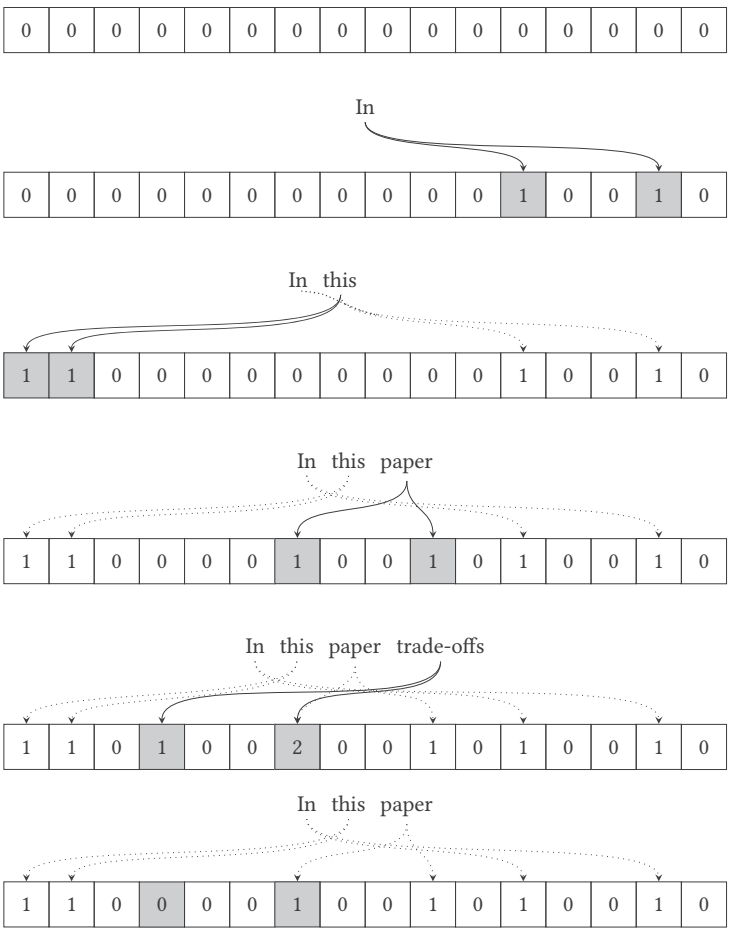


Рисунок 13.20. Добавление и удаление в счетном фильтре Блума

Алгоритм принимает строку *s*, еще один параметр *i* и возвращает 32-битное хеш-значение. Для разных значений *i* он генерирует разные хеш-значения для одной и той же строки *s*, поэтому мы можем взять его за основу фильтра Блума. Чтобы все так и получилось, хеш-значению в строчке 1 присваивается, на первый взгляд, волшебное число.

Алгоритм 13.6. Удаление из счетного фильтра Блума

`RemoveFromCntBloomFilter(`T, x`)`

Вводные данные: T, массив размером m, содержащий целые числа; x, запись для извлечения из множества, представленного счетным фильтром Блума.

Результат: запись извлечена из счетного фильтра Блума путем уменьшения на 1 индикаторов в $h_0(x)$, $h_1(x)$, …, $h_{k-1}(x)$.

```
1 for i ← 0 to k do
2     h ← h_i(x)
3     if T[h] ≠ 0 then
4         T[h] ← T[h] − 1
```

Алгоритм 13.17. FNV-1a, основанная на 32-битном хешировании

`FNV-1a(`s, i`)` $\rightarrow h$

Вводные данные: s, строка символов; i, целое число.

Выводимые данные: 32-битное хеш-значение s.

```
1 h ← 0x811C9DC5
2 p ← 0x01000193
3 h ← h ⊕ i
4 foreach c in s do
5     h ← h ⊕ Ordinal(c)
6     h ← (h × p) & 0xFFFFFFFF
7 return h
```

Над ним совершается операция `xor` с целым числом *i*, которое мы ввели в качестве параметра. Затем, в строчке 5, берется значение каждого символа из заданной строки, и «ксорится» с нынешним хеш-значением. Данное значение каждого символа присваивается функцией `Ordinal`(*c*), которая возвращает

код, использованный для символа: это может быть Юникод или ASCII, в зависимости от представления. В строчке 6 алгоритм умножает результат на другое волшебное число *p*, установленное в строчке 2, и берет последние 32 бита результата, так как нам нужно 32-битное хеш-значение. Чтобы взять последние 32 бита от результата, мы выполняем побитовую операцию «И» с символом & (амперсандом) между полученным произведением и числом 0×FFFFFFFF, которое равно 32 битам, установленным на 1. Побитовая операция «И» берет две последовательности битов и выводит 1 в каждую позицию, где у обеих последовательностей находится 1, в противном же случае она выводит 0; смотрите таблицу 13.4. В плане расчетов взятие последних 32 битов — это все равно что взятие остатка от деления, 2^{32}, однако побитовая операция «И» помогает нам избежать деления; мы просто исключаем всякий бит вне 32 битов вместо того, чтобы вычислять $(h \times p) \bmod 2^{32}$.

Таблица 13.4. Побитовая операция «И»

		x	
		0	1
y	0	0	0
	1	0	1

Между прочим, это простой и полезный трюк. Когда нам надо исключить определенные биты, мы прибегаем к двоичной комбинации, в нашем случае к 0×FFFFFFFF, содержащей лишь те биты, которые нам нужно оставить, если они уже и так есть. Такая двоичная комбинация называется битовой маской, или просто маской. На рисунке 13.21 показано, применение маски из четырех битов к числу с восемью битам для исключения всех битов кроме последних четырех. Битовая маска не обязательно состоит из одних только единиц, хотя чаще всего именно так.

Пока мы касаемся данной темы, стоит упомянуть противоположность побитовой операции «И» — побитовую операцию «ИЛИ»; взгляните на таблицу 13.5. Мы можем применить битовую маску с побитовой операцией «ИЛИ», когда хотим оставить биты, а не исключить, как на рисунке 13.22. Как правило, обозначается

побитовая «ИЛИ» вертикальной чертой (|). В нашем алгоритме нет нужды в побитовой операции «ИЛИ», однако знать о ней стоит, потому что когда-нибудь она может пригодиться. Знать о ней важно, потому что может получиться недопонимание, так как в определенных программных языках *a* && *b* означает, что выражение верно, если одновременно и *a*, и *b* верны, тогда как *a* || *b* означает, что выражение верно, если верно либо *a*, либо *b*, что соответствует «и» и «или» в наших алгоритмах. В то же время *a* & *b* значит, что надо вычислить побитовую «И» для *a* и *b*, тогда как *a* | *b* значит, что надо вычислить побитовую «ИЛИ» для *a* и *b*. Одна пара операторов сообщает о достоверности, а другая обрабатывает числа в двоичном представлении. Путать их не рекомендуется.

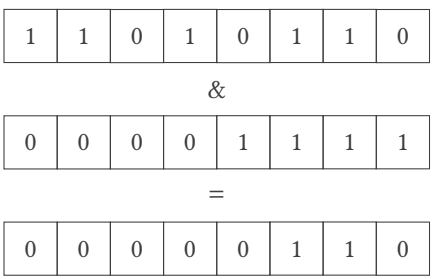


Рисунок 13.21. Применение битовой маски 00001111 = 0×F к 11010110 с побитовой операцией «И»

Таблица 13.5. Побитовая операция «ИЛИ»

		x	
		0	1
y	0	0	1
	1	1	1

Говоря об особенностях языка программирования, можно упомянуть еще одну, присутствующую во многих языках, она помогает упростить строчку 6 алгоритма 13.17 до $h \leftarrow h \times p$. Если программный язык хранит числа в 32 битах, то всякий раз, когда $h \times p$ превышает порог в 32 бита, то из-за переполнения только числа с 32 битами будут храниться в *h*. Поэтому нет нужды ни в битовой маске, ни в делении с остатком.

Кажется, мы сделали круг и вернулись к началу. Мы взялись за алгоритмы, которые казались похожи на трюк фокусника, и добрались до алгоритма тарабарскими числами. Числа в строчках 1 и 2 заставляют думать, что результаты хеш-функции получены случайным образом. Словно мы взяли в качестве вводных данных набор битов (нашу строку символов), встряхнули, смешали — и вдруг получился совершенно иной набор, который, кажется, никак не связан с вводными данными. Такое непросто сотворить. Здесь требуется применение нескольких способов рандомизации, и нам нужно удостовериться, что полученное поведение соответствует нашим условиям: в нашем случае, чтобы различные значения для *i* давали нам различные хеш-функции, которые вели бы себя как по-настоящему независимые.

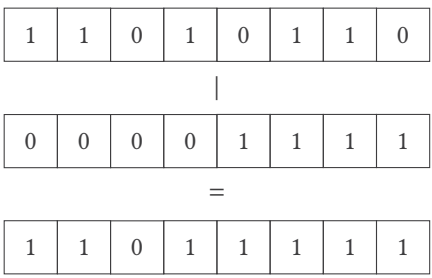


Рисунок 13.22. Применение битовой маски 00001111 = 0×F к 11010110 с побитовой операцией «ИЛИ»

Примечания

Принцип Дирихле, который также известен как «принцип ящиков», сформулирован в 1834 году математиком Петером Густавом Леженом Дирихле. Согласно истории хеширования [118, стр. xv], впервые данную идею упомянул Ганс Петер Лун во внутренней документации IBM в январе 1953 года. Арнольд Дами, соизобретатель почтовой сортировочной машины и криптоаналитик военной разведслужбы США, а после — Агентства национальной безопасности (АНБ), представил идею хеширования несколько

лет спустя, в 1956 году, в первой работе, посвященной хешированию [54], хотя сам термин тогда еще не употреблялся. Сам термин ввел в обиход в 1968 году программист и криптограф Роберт Моррис [146]. Моррис работал в «Лаборатории Белла» и осуществил значительный вклад в развитие операционной системы Unix, а затем в 1986 году перешел в АНБ, где проработал до самой пенсии. Простой пример семейства идеальных хеш-функций предложил Чичелли [37]. Кодирование отдельных символов в алгоритме 13.2 было найдено с помощью GPERF [173].

Уильям Джордж Горнер представил свой метод, названный в честь него, в 1819 году [97]. Однако метод был открыт еще до него. Ньютон использовал его еще за 150 лет до Горнера, также его использовал в Китае XIII века математик Цинь Цзю-Шао [113, с. 486]. Похожий процесс был выполнен Паоло Руффини в 1804 году, но, кажется, ни Горнер, ни Руффини ничего не знали об открытиях своих предшественников. Представление Горнера довольно сложное по сравнению с сегодняшним изложением математических текстов; в первом издании было упомянуто, что «Элементарность темы представляла собой явный вызов; его [Горнера] труднопостижимый способ анализа представлял собой явным залог к его признанию» [190, с. 232]. Алгоритм 13.6 похож на хеш-функцию, которая применяется в программном языке Java [22, с. 45–50].

Описанный нами процесс распознавания песен является упрощенной версией сервиса «Шазам» (Shazam) [212], [89]. Рисунки были изображены с помощью кода, полученного из проекта Dejavu (https://github.com/worldveil/dejavu). Бертон Блум описал фильтры, названные в честь него, в 1970 году [23], взяв в качестве примера их использования автоперенос, то есть разбивку по слогам. Исследование Блума начинается словами: «В данной работе анализируются взаимосвязи между определенными расчетными факторами в хеш-кодировании», — именно их мы и позаимствовали для наших примеров. Симсон Гарфинкель написал небольшое введение в компьютерную криминалистику, подходящее для знакомства ой [76]; чтобы узнать о применении фильтров Блума в компьютерной криминалистике, смотрите [222]. Хеш-алгоритм

FNV описан в [70]. Существуют и другие популярные алгоритмы, которые используются в фильтрах Блума, можно сказать, с большей продуктивностью, например MurmurHash, Jenkins и FarmHash. Вы с легкостью можете найти в Интернете их реализации в разных языках программирования. Для знакомства и получения математических формул, лежащих в основе фильтров Блума, загляните в [24]. Фильтры Блума есть в учебном пособии Миценмахера и Апфала, в разделе, посвященном рандомизированным алгоритмам [144, раздел 5.5.3]. Бродер и Миценмахер написали подробный обзор о применении фильтров Блума в Сети [30].

Упражнения

1. Опишите, как можно использовать хеш-таблицу для нахождения уникальных элементов неотсортированного массива или списка, и реализуйте соответствующую программу. Затем сделайте наоборот: найдите способ и напишите программу, которая будет находить в неотсортированном массиве или списке элементы, повторяющиеся несколько раз.
2. У вас есть неотсортированный массив, содержащий целые числа и целое число s. Как за время $O(n)$ можно отыскать все пары чисел (a, b), если a и b расположены в массиве так, что $a + b = s$? Напишите программу, которая отыскивает их с помощью хеш-таблицы.
3. Как с помощью хеш-таблицы можно найти наиболее употребляемое в тексте слово?
4. Как можно реализовать объединение, пересечение и разницу двух множеств с помощью хеш-таблицы?
5. Анаграмма — это отрывок текста, полученный путем перестановки букв в другом отрывке. Например, alert является анаграммой для alter*. Представьте, что у вас есть

* Alert — сигнал тревоги, alter — менять.

словарь, содержащий все английские слова. Можно найти все слова, которые являются анаграммами других слов, с помощью хеш-таблицы, следуя описанному далее способу. Просмотрите каждое слово в словаре, отсортируйте символы нужного слова; для “alter” вы получите “alert”. Затем используйте данную строку символов как ключ в хеш-таблице, чьими значениями являются все слова в словаре, состоящие из тех же букв. Напишите анаграммный поисковик, который будет отыскивать анаграммы по приведенной выше схеме.

14 Биты и деревья

«И-Цзин» (易經), или «Книга Перемен», является древним священным китайским текстом, который появился примерно между X и IV веками до нашей эры. Процесс гадания в «И-Цзин» включает в себя выбор случайных чисел, которые соответствуют особому символу, гексаграмме. Каждая гексаграмма состоит шести горизонтальных линий, расположенных друг над другом; линии могут прерываться или быть непрерывными. Так как для каждой линии существует два варианта, у нас есть $2 \times 2 \times 2 \times 2 \times 2 \times 2 = 64$ возможные гексаграммы. Традиционно гексаграммы располагаются в определенном порядке, как показано на рисунке 14.1. Их порядок обладает интересными свойствами. Гексаграммы образуют пары, 28 из них содержат гексаграммы, которые зеркально отражают друг друга по вертикали (как третья и четвертая гексаграммы); остальные восемь пар содержат гексаграммы, которые являются инверсиями друг друга так, что непрерывная линия в одной гексаграмме становится прерывной в другой (как в первой и второй гексаграммах).

Согласно «И-Цзин», у каждой гексаграммы есть определенное толкование. Изначально число получалось в ходе выкладывания стеблей тысячелистника. Гексаграмму, соответствующую определенному числу, затем толковали как божественный умысел.

В основе всех гаданий лежит идея, согласно которой божественная воля сообщается человеку через посредника и трактуется как знак, понятный только посвященным. Порой знаки и их интерпретации можно найти в самых неожиданных местах, например, чтение по чаинкам в чашке или гадание на кофейной гуще. Существует даже специальный термин, тассеология, который происходит от французского tasse — чашка и греческого λόγος, logos — слово. «И-Цзин» является примером клеромантии, образованной от греческих cleros — множество и manteia — пред-

сказание; следовательно, клеромантия означает предсказание по множеству.

Рисунок 14.1. Гексаграммы «И-Цзин»

Древним широко известным текстам всегда соответствует куча комментариев, и «И-Цзин» не исключение. Толкование полученной гексаграммы в каждом конкретном случае содержит огромную долю неясности и двусмысленности, что закономерно, потому что нельзя же иметь 64 универсальных толкования на все случаи жизни.

Если мы не будем брать в расчет вероятность, что божества каким-то образом и правда изъясняются с помощью гексаграмм «И-Цзин», тогда можно определить ее потенциальную правдивость, выяснив, как много может рассказать нам каждая гексаграмма. Конечно, у каждой гексаграммы множество толкований, однако, несмотря на всю их разнородность, каждая интерпретация начинается с одного и того же рисунка из прерывистых и непрерывных линий. Как много можем мы узнать из каждого рисунка? Иными словами, что на самом деле говорят оракулам божества? Хоть вопрос и звучит странновато, тем не менее у нас есть хороший научный ответ.

14.1. Предсказание как проблема коммуникации

Мы можем разбить предсказание на символы, как на рисунке 14.2, где дано общее схематичное представление коммуникационной системы. Обычно у нас есть источник информации, который передает сообщение. Сообщением может служить что угодно: текст, музыка, фильм и так далее. Прежде чем сообщение передать, его нужно превратить в сигнал, идущий по каналу коммуникации.

Во время прохождения по каналу коммуникации сигнал может столкнуться с помехами, которые, как ни печально, его исказят. В конце пути сигнал приходит к получателю, который умеет превратить его обратно в исходное сообщение и направить его дальше, в место назначения.

Если перед нами два разговаривающих друг с другом человека, Алиса и Боб, то сообщение формируется в мозгу одного из собеседников, скажем, Алисы. Мысль превращается в вербальную речь, в предложение, которое представляет собой аудиосигнал, передающийся в виде звука по воздуху, который, в свою очередь, является коммуникационным каналом. Ухо Боба трансформирует звуковые волны в электрические сигналы; его ухо является тем самым получателем. Электрические сигналы доходят до мозга Боба, конечного пункта назначения, куда и должно было прийти сообщение. Если Алиса и Боб разговаривают в шумной обстановке, то шум создаст помехи их звуковым волнам, и понять друг друга, скорее всего, станет сложно.

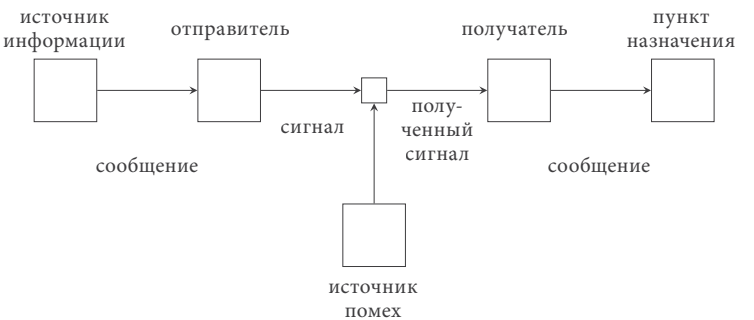


Рисунок 14.2. Система коммуникации

В современных коммуникациях сигналом, как правило, служит электромагнитная волна, которая идет по проводу или воздуху, или же световые импульсы, путешествующие по оптоволокну. В качестве сообщений может выступать что угодно, превращенное в подходящий сигнал и отправленное к месту назначения.

Если вернуться к предсказаниям, то божества точно так же посылают сообщения, трансформированные в подходящий сигнал.

В «И-Цзин» сигналом служит число, полученное из стеблей тысячелистника. Оракул получает сигнал и может превратить его в исходное сообщение. Возможно, свою роль сыграют помехи, и тогда толкование божественной воли окажется ошибочным. Именно поэтому не все предсказания сбываются.

Главное свойство системы коммуникации заключается в том, что она передает информацию из источника в пункт назначения. Но что такое информация? В бытовом значении информацией называют нечто, имеющее смысловое значение. Такая трактовка нас не интересует, нам нужно техническое определение информации. Мы можем начать с наблюдения, что мы передаем информацию, когда сообщаем кому-то нечто новое, ранее ему неизвестное. Информация передается, когда мы узнаем что-то, чего не знали раньше. Точно так же сообщение содержит информацию, которая непредсказуема. Если содержимое послания заранее известно, тогда зачем его посылать? Получатель все равно с легкостью предвидит содержание; ему не нужно дожидаться сообщения, если он хоть приблизительно знает, о чем в нем говорится. Поэтому для наличия информации нужна неопределенность с содержанием сообщения; мы должны рассказать нечто, о чем получатель не сможет заранее догадаться.

Простое сообщение, подходящее такому описанию, выглядит так: некто сообщает нам выборе, сделанном между двумя равнозначными вариантами. Выбор может быть между «да» и «нет», «ноль» и «один», «черное» и «белое», когда у нас нет никакой подсказки о том, какой вариант вероятней. В сообщении решается вопрос, следовательно, содержится и информация. Вся информация, представленная в выборе между двумя равновероятными вариантами, или, по-другому, двоичном выборе, называется битом, сокращенно от binary digit, двоичной цифры, потому что двоичная цифра может принимать одно из двух значений, ноль или единицу, которые представляют одно из двух возможных решений. Бит — это единица измерения информации.

Простейшее сообщение с «да» или «нет» содержит информацию объемом в один бит, поэтому сообщения могут передавать больше информации, если их содержимое представляет больше

вероятностей. Если в сообщении находятся ответы на два равновозможных вопроса «да» или «нет», то его объем информации равен двум битам. Точно так же, если сообщение содержит ответы на n схожих вопросов «да» или «нет», то его объем информации составляет n битов.

Если у нас имеется n различных битов и каждый из них отвечает за вероятность, независимую от всех других вероятностей, у нас может быть $2 \times 2 \times \ldots \times 2 = 2^n$ различных сообщений. Поэтому, чтобы понять информацию, содержащуюся в сообщении с n битами, стоит взглянуть на данное сообщение как на объем информации, который позволяет нам выбрать между 2^n различными вариантами, приводящих к n разным равновозможным решениям.

Каждая гексаграмма «И-Цзин» содержит ответ к шести двоичным вероятностям, что соответствует 64 различным решениям-толкованиям. Отсюда следует, что система предсказаний сводится к способу коммуникации путем сообщений вместимостью шесть бит. Вот столько много нам может сообщить предсказание. Просто поразительно, что ответы на важнейшие вопросы могут передаваться с помощью таких коротких сообщений.

Но такая система в «И-Цзин» не спроста: каждая гексаграмма и вправду сопоставима с шестибитным сообщением. Исторически сложилось, что божества изображаются довольно немногословными существами. Известную Пифию, оракула бога Аполлона в дельфийском храме времен древней Греции, почитали за ее великую силу. Когда Крез, царь Лидии, спросил ее, стоит ли ему переходить реку и атаковать персов, то получил следующий ответ: «Если ты перейдешь реку, то погубишь великое царство». Крез решил, что речь идет о персидском царстве и напал на врага, однако потерпел поражение, и его царство было разрушено. Ответ Пифии на вопрос: «Какое царство падет?» — был совершенно несодержательным сообщением с нулем бит информации.

14.2. Информация и энтропия

Когда мы сталкиваемся с равновозможным двоичным выбором, объем информации равен числу выборов; так как каждому выбо-

ру соответствует один бит, объем информации равен числу битов равному числу разных выборов. Как мы видели, для 2^n различных вариантов нам нужно n различных битов. Следовательно, объем информации в сообщении среди 2^n равновозможных сообщений равен $n = \lg(2^n)$ битов; говоря проще, он представляет собой логарифмическое основание двух от числа возможных исходов.

Это касается информационного содержания в сообщении с равновозможными исходами, однако мы обязаны просмотреть ситуацию, когда решения не равновозможны. В данном случае нам удобней будет изменить основу наших рассуждений и перейти от исходов к их вероятностям; вместо разговоров о принятых решениях мы поговорим о событиях и их вероятностях. Объем информации в сообщении m, дающий знать, что произошло событие с вероятностью p, можно в таком случае определить как $h(m) = -\lg p$. Чтобы получить положительный результат, мы должны отказаться от логарифма, потому что $p \leq 1$; вместо него мы можем использовать $h(m) = \lg(1/p)$, но проще не связываться с дробями. Когда вероятность события равна $p = 1/2$, что то же самое, что иметь два равновозможных исхода, мы получаем $h(m) = -\lg(1/2) = 1$ бит, как у нас и было прежде. Однако теперь в наших силах обобщить информацию, выходящую за рамки равновозможных событий. Например, допустим, нам известно (основываясь на прошлых и нынешних метеорологических данных), что вероятность дождя завтра составляет 80%. Сколько информации передает утверждение m «Завтра ожидается дождь»? Так как у нас $p = 8/10$, объем информации, содержащейся в сообщение, равен $h(m) = -\lg(8/10) \approx 0{,}32$ бита. Если мы выскажем обратное утверждение m', «Завтра дождя не ожидается», мы получим информацию равную $h(m') = -\lg(2/10) = 2{,}32$ бита. Мы получаем меньше информации, когда нам сообщают что-то, что мы и так уже знаем. Мы получаем больше информации, когда нам сообщают что-то неожиданное: мы полагаем, что дождь вряд ли пойдет, однако нас пытаются переубедить. Если вероятность дождя в завтрашний день составляет 50%, тогда можно только гадать, пойдет ли он завтра или нет. Если мы знаем, что с равной вероятностью может быть и так и этак, то прогноз, обещающий

дождь, сообщает столько же информации, сколько и прогноз, обещающий отсутствие осадков, то есть 1 бит информации. Наименее интригующие ситуации, когда мы точно знаем, что событие не произойдет, $p = 0$, или наоборот, произойдет, $p = 1$. В обоих случаях мы получаем ноль информации. В самом деле, если $p = 1$, у нас $-\lg 1 = 0$. Если $p = 0$, логарифм не определен, но, когда мы точно знаем и говорим, что событие не случится, это ведь то же самое, что провозгласить обратное, поэтому речь тут идет о дополнительной вероятности $1 - p$, которая, опять же, передает ноль информации.

Давайте перейдем от погоды к английским текстам, состоящим из символов. Появление или отсутствие особого символа в определенной позиции текста рассматривается как случай, поэтому мы можем исследовать объем информации, передаваемой каждой отдельной буквой. Помните, что мы говорим о техническом определении информации — о системе измерения битами, — а не о бытовом значении, когда под информацией текста подразумевается что-то совершенно иное. Если у нас есть частота появления букв, мы можем перейти от частоты к вероятностям и вычислить информацию, передаваемую ими, путем взятия логарифма вероятностей с основанием два. В таблице 14.1 показаны буквы, частоты, их вероятности и соответствующая информация. Наиболее употребляемая английская буква, E, обладает одной третью информационного содержания наименее употребляемой английской буквы Z. Так происходит, потому что в английских текстах E встречается намного чаще, чем Z, поэтому появление Z более неожиданно, чем появление E.

Вычислив информацию для каждой отдельной буквы, можно задаться вопросом: каков средний размер информации в английской букве? Так как нам известна вероятность встречи и информация каждой буквы, то среднее значение мы находим как обычно: умножаем каждое значение на вероятность, с которой оно появляется, а затем складываем все полученные произведения. Точно так же мы перемножаем каждое значение с числом раз, когда оно появляется, поделенным на общее число появлений всех значений (перед нами пропорция количества раз, когда

появляется конкретное значение) и складываем полученные произведения. Чаще всего, особенно в технических текстах, говорят об ожидаемом значении, а не о среднем.

Чтобы вычислить среднюю информацию английской буквы, нам всего-то надо умножить каждую вероятность на число битов информации и сложить все произведения. Если $p(A)$, $p(B)$, …, $p(Z)$ — это вероятности, то они равны:

$$p(A)h(A) + p(B)h(B) + \cdots + p(Z)h(Z)$$

Сумма дает нам 4,16 бита в качестве среднем объеме информации, содержащейся в английской букве. На самом деле данное значение слишком большое, потому что предполагает, что буквы никак не связаны друг с другом. Однако буквы в тексте связаны, и их появление можно предсказать, основываясь на уже имеющихся; более точно значение равно примерно 1,3 бита на букву.

Вычисление, которые мы только что выполнили, является определением среднего информационного объема исхода x_i из множества возможных исходов $X = \{x_1, x_{2, \ldots,} x_n\}$, вероятность каждого $p(x_i)$. Набор исходов x_i из множества исходов X с вероятностями $p(x_i)$ называется ансамблем. В нашем примере с английским текстом x_i является буквой, X — всеми буквами, а $p(x_i)$ — вероятностью конкретной буквы x_i. Говоря математическим языком, определение среднего количества информации для ансамбля с n различными исходами таково:

$$\begin{aligned} H(X) &= p(x_1)h(x_1) + p(x_2)h(x_2) + \cdots + p(x_n)h(x_n) \\ &= -p(x_1)\lg p(x_1) - p(x_2)\lg p(x_2) - \cdots - p(x_n)\lg p(x_n) \\ &= -\Big[p(x_1)\lg p(x_1) + p(x_2)\lg p(x_2) + \cdots + p(x_n)\lg p(x_n)\Big]. \end{aligned}$$

Значение $H(X)$ называется энтропией множества исходов X. Приведенная выше формула тремя разными способами говорит, что энтропия множества исходов является средним информационным объемом исхода, или ожидаемым информационным объемом исхода.

Данные определения информации и энтропии принадлежат Клоду Элвуду Шеннону, представивших их в 1948 году. Чаще всего техническое определение информации называется информацией

Шеннона, или шенноновским объемом информации; иногда используется слово *неожиданная*, чтобы подчеркнуть, насколько мы должны быть удивлены, ознакомившись с содержимым сообщения. Описание информации в математических терминах позволило разработать новую дисциплину, теорию передачи информации, которая лежит в основе современных коммуникаций, сжатия данных, а также касается таких областей, как лингвистика и космология.

Таблица 14.1. Информация английских букв

Буква	Частота	Вероятность	Информация
E	12.49	0.1249	3.0012
T	9.28	0.0928	3.4297
A	8.04	0.0804	3.6367
O	7.64	0.0764	3.7103
I	7.57	0.0757	3.7236
N	7.23	0.0723	3.7899
S	6.51	0.0651	3.9412
R	6.28	0.0628	3.9931
H	5.05	0.0505	4.3076
L	4.07	0.0407	4.6188
D	3.82	0.0382	4.7103
C	3.34	0.0334	4.904
U	2.73	0.0273	5.195
M	2.51	0.0251	5.3162
F	2.40	0.0240	5.3808
P	2.14	0.0214	5.5462
G	1.87	0.0187	5.7408
W	1.68	0.0168	5.8954
Y	1.66	0.0166	5.9127
B	1.48	0.0148	6.0783
V	1.05	0.0105	6.5735
K	0.54	0.0054	7.5328
X	0.23	0.0023	8.7642
J	0.16	0.0016	9.2877
Q	0.12	0.0012	9.7027
Z	0.09	0.0009	10.1178

Хотя объем информации и энтропия близки, тем не менее они отличаются друг от друга. Энтропия определяется информационным объемом. Информационный объем является множеством битов, соответствующих событию. Энтропия относится

к ансамблям и представляет собой средний объем информации всех событий в ансамбле.

Чтобы заметить разницу, поговорим об информационном объеме исхода «орла» или «решки» при подбрасывании монетки. Если монетка ровная, то вероятность решки p равна вероятности орла q, и тогда у нас $h(p) = h(q) = -\lg(1/2) = 1$ бит. Если монетка с изъяном и p более вероятна, чем орел, скажем, $p = 2/3$, а $q = 1/3$, то у нас $h(p) = -\lg(2/3) = 0{,}58$ битов и $h(q) = -\lg(1/3) = = 1{,}58$ бит. Решка более ожидаема, чем орел, следовательно, сообщение о том, что у нас выпала решка, менее информативно, чем сообщение о выпадении орла.

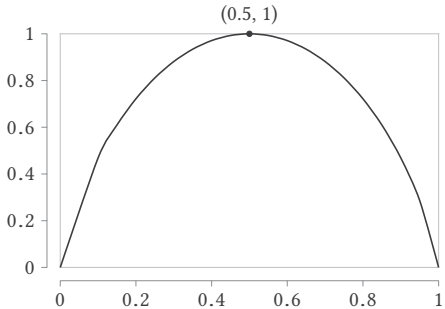


Рисунок 14.3. Энтропия ансамбля двух событий с вероятностями p и $q = 1 - p$. Вероятность p расположена на оси x, а энтропия — на оси y

Когда мы говорим о монетке, бессмысленно рассуждать об информационном объеме. Мы можем говорить лишь о среднем информационном объеме исходов, то есть среднем информационном объеме двух исходов, орла или решки, что является энтропией монеты. Если монета X ровная, то у нас $H(X) = -p \lg p - q \lg q = -(1/2) \lg(1/2) - (1/2)\lg(1/2) = 1$ бит. Если же монета со смещенным центром тяжести так, что $p = 2/3$, а $q = 1/3$, то мы имеем $H(X) = -p \lg p - q \lg q = -(2/3)\lg(2/3) - (1/3)\lg(1/3) \approx 0{,}92$ битов. Получается, что поведение монеты со смещенным центром тяжести более предсказуемо, и, следовательно, энтропия у нее меньше. Мы можем нарисовать график, показывающий изменение энтропии в зависимости от изменений p и q; взгляните на рисунок 14.3. Энтропия достигает своего апогея при $p = q = 0{,}5$, когда становится равна 1 биту.

Почему энтропия? Почему средний объем информации ансамбля так называется? Описание энтропии впервые было дано в физике австрийским физиком Людвигом Больцманом в 1872 году и американским ученым Дж. Уиллардом Гиббсом в 1878 году. Оба определения связаны с количеством хаоса в системе: чем меньше в системе порядка, тем выше энтропия.

Чтобы понять энтропию в терминах физики, мы можем начать с наблюдения, что у системы есть два состояния, внешнее и внутренне. Внешнее состояние касается того, как система выглядит снаружи; внутреннее состояние — того, что происходит внутри. Например, представим, что у нас есть прозрачная банка, наполненная двумя видами молекул, черными и белыми (возможно, молекулы черной и белой красок). Внутреннее состояние банки выражается в расположении каждой молекулы. Внешнее состояние банки выражается в цвете содержимого. Существует куда больше внутренних состояний, когда мы видим серый цвет, чем внутренних состояний, когда можно наблюдать резкое разделение цветов: одна половина банки белая, а другая — черная. Все из-за того, что во втором случае внешнему состоянию требуется, чтобы все молекулы одного цвета сгруппировались вместе, в то время как в первом случае у внешнего состояния подобного требования нет. Таким образом, вариантов случайного распределения молекул по банке куда больше, чем вариантов, требующих разделения молекул по цветам. У серого внешнего состояния больше внутренних состояний, чем у внешнего черно-белого, следовательно, и энтропия у него выше.

Внутренние состояния системы появляются из состояния составляющих ее частей и называются микросостояниями. Внешнее состояние системы является величиной, описывающей некий ее аспект, и называется макросостоянием. С помощью данных определений энтропия S в системе при определенном макросостоянии описывается Больцманном как

$$S = k_B \ln \Omega,$$

где Ω — это число микросостояний (все с равными вероятностями), которые соответствуют особому макросостоянию, а k_B является физической константой, называемой постоянной Больцмана.

Данная формула вычисляет случаи и берет натуральный логарифм, что несколько напоминает энтропию Шеннона. Схожесть между физической энтропией и информационной становится более очевидной, когда мы рассматриваем формулу Гибба для *S*:

$$S = -k_B\left[p(x_1)\ln p(x_1) + p(x_2)\ln p(x_2) + \cdots + p(x_n)\ln p(x_n)\right],$$

где x_1, $x_{2,\ldots,}x_n$ — различные микросостояния системы, а $p(x_i)$ — вероятность каждого микросостояния. Формула Гибба представляет собой общую версию формулы Больцмана, так как она не требует равных вероятностей. Если не считать k_B и использования натуральных логарифмов вместо логарифмов по основанию 2, то определение ничем не отличается от данного Шенноном.

Присмотревшись внимательней к синтаксической схожести энтропии в физике и информационной энтропии, так и тянет предположить, что, может быть, здесь скрывается куда большее значение. И в самом деле, вокруг не утихают дискуссии, касающиеся потаенного смысла. Один из способов подробней изучить данный вопрос — представить гиббову энтропию *S* как объем информации, требуемой для точного описания определенного микросостояния системы, например позиции и импульса каждой черной и белой молекулы в банке.

14.3. Классификация

Людям присуща удивительная способность: подразделять вещи на группы. Мы применяем ее ко всем вещам, живым и неживым; такая способность называется классифицированием. Она нужна для выживания, ведь нам не требуется разъяснений, что огромная рыжая кошка с черными полосками может представлять угрозу нашей жизни; мы знаем, что все похожее на тигра, возможно, и есть тигр, и следует соблюдать осторожность, даже если мы никогда прежде не встречали тигра. Точно так же, если мы впервые столкнемся с похожим на хищника животным, то мы, наверно, без всяких раздумий тут же решим, что оно опасно, и нам не захочется выяснять, так ли это на самом деле.

Умение классифицировать важно не только для выживания, но и для жизни в современном обществе. Банкам нужно классифицировать заемщиков по их возможностям выплатить долг. Розничные торговцы хотят классифицировать потенциальных покупателей, чтобы знать, на кого ориентировать рекламные компании. Операторы социальных медиа и работники соцопросов ищут способы классифицировать высказывания как «за» или «против» определенных заявлений и позиций.

Существует несколько способов классификации. Мы рассмотрим здесь самый очевидный: использование множества атрибутов, то есть характерных черт, свойственных каждому случаю, который мы хотим классифицировать. Атрибутами может выступать все что угодно, релевантное задаче классификации: например, для классификации заемщиков банк может рассматривать возраст, доход, пол, уровень образования и рабочую должность; в то же время такие черты, как рост или вес, релевантные в других классификациях, совершенно не подходят для распознавания добропорядочных заемщиков. Разделяя грибы на съедобные и несъедобные, мы можем использовать для нашей классификации такие черты, как форма шляпки, размер пластинки, форма ножки, запах и среда обитания, в то время как другие черты, например многочисленность или редкость, будут для нас бесполезны.

Атрибуты могут быть численные или категориальные. Значения численных атрибутов — числа, например, доходы. Значением категориального атрибута является одна из множества категорий. К уровню образования, например, применимы значения «начальное», «среднее», «высшее», «кандидат наук» и «доктор наук». У численных атрибутов, таких как рост, в качестве значений идут числа. У нас также могут быть булевы атрибуты со значениями «истинно» и «ложно». Если удобно, можно рассматривать их как категориальные атрибуты с двумя значениями или как численные — с нулем для «ложно» и единицей для «истинно». Класс элемента, зачастую определяемый значением одного из его атрибутов, называется классовым атрибутом, или отметкой. Заемщик может быть отмечен как «надежный» или «ненадежный».

Если у нас есть множество атрибутов, которые, как мы полагаем, подходят для классифицирования множества случаев, проблема классификации переходит к вопросу: как нам с помощью имеющихся атрибутов классифицировать разные случаи? Порой ответ очевиден, например, для безработного заемщика с нулевым доходом и неполным средним образованием. Однако такие однозначные ситуации попадаются редко. В прошлом хорошая классификация требовала работы опытных специалистов: они выносили решение, полагаясь на чутье и определенные правила, приобретенные за годы службы. В наши дни такой подход, как правило, больше не целесообразен. Сегодня нам приходится рассматривать и классифицировать слишком огромное количество случаев для самых разных целей: только представьте, на какие вопросы приходится ежедневно отвечать крупным розничным торговцам или сотрудникам службы поддержки и какое количество случаев им приходится классифицировать.

Поэтому хорошо бы научить компьютеры классификации. Такая задача приводит нас в мир машинного обучения, где все посвящено разработке методов, с помощью которых компьютеры смогут обучаться разным навыкам, например классификации, хотя она и считается некоей интеллектуальной особенностью, присущей человеку. Машинное обучение разделяется на три категории.

В контролируемом обучении ученики, то есть компьютеры, учатся выполнять задания с имеющимся набором данных, который называется обучающим набором, и правильными ответами к задаче. Суть контролируемого обучения состоит в том, чтобы использовать обучающий набор и ответы для выработки действий, которые будут потом применены в настоящей работе с реальными данными. Примером контролируемого обучения как раз и является классификация. Мы даем ученику набор элементов, их атрибуты и классы и хотим, чтобы он отыскал способ классифицировать другие элементы, которые еще не видел.

При неконтролируемом обучении ученик пытается найти в наборе данных некую скрытую структуру и у него нет правильных ответов, помогающих ему узнать эту структуру. Ученику пред-

лагается опираться лишь на характерные черты, не зная заранее о каких-либо результатах. Мы можем использовать неконтролируемое обучения для создания кластеров, то есть, имея заданный набор элементов с общими атрибутами, мы находим число кластеров, на которое лучше всего разделить элементы. Каждый кластер представляет собой такую группу элементов, в которой элементы одной группы более схожи друг с другом, чем с элементами других групп.

Что касается стимулированного обучения, здесь ученикам дается обучающий набор и их просят выполнить задание; ученики получают отзыв, однако в обратной связи нет оценок «верно» или «неверно», в ней содержатся награды или штрафы, которые ученик накапливает. Самые часто встречаемые представители стимулированного обучения — роботы: они взаимодействуют с окружающей средой и, основываясь на данных обратной связи, контролируют свои движения.

14.4. Деревья решений

Мы сосредоточимся на контролируемом методе обучения для классификации. Начнем с обучающего набора, в котором содержатся данные с известными классами. Мы хотим научить компьютер вычислять класс неизвестных данных. Метод использует принцип «разделяй и властвуй»: берется исходный обучающий набор, а также атрибуты, которые описывают тренировочные случаи, затем набор данных, основанных на значениях атрибутов, разбивается на все меньшие и меньшие подгруппы, пока дело не дойдет до подгрупп, соответствующих особым классам. К концу обучения ученик изучает заданный тренировочный набор и знает, как проводить классификацию с помощью выбранных атрибутов. Затем ученик переносит полученные знания на настоящие, рабочие данные. Разделение обучающего набора на подгруппы можно с легкостью представить в виде дерева; такое дерево называется деревом решений: деревом, внутренние узлы которого представляют собой проверку атрибута, соединения между родительскими и дочерними узлами — результаты теста

в родительском узле, а листьям соответствуют классы. Дерево решений — это пример предсказания класса неизвестных данных; такие примеры называются моделями прогнозирования.

Таблица 14.2. Простой обучающий набор для метеоданных

номер	атрибуты			класс
	примерный прогноз	влажность	ветер	
1	солнечно	высокая	нет	Н
2	облачно	высокая	нет	Г
3	дождь	высокая	нет	Г
4	солнечно	средняя	нет	Г
5	дождь	высокая	есть	Н
6	дождь	средняя	есть	Н

Пример обучающего набора приведен в таблице 14.2. В нем содержатся метеорологические данные для утра шести разных суббот. В данном дереве мы решаем, стоит ли гулять утром в каждую из приведенных суббот, опираясь на то, что нам сообщает обучающий набор. Мы классифицируем субботние утра следующим образом: «Г» (Гуляем) и «Н» (Не гуляем). Выделяем три атрибута, характеризующих день: примерный прогноз, влажность и наличие ветра. У примерного прогноза три значения: солнечно, облачно и дождь. Влажность может быть высокой или средней. Ветер либо есть, либо нет. Простое дерево решений, созданное на основе такого обучающего набора, изображено на рисунке 14.4.

Чтобы классифицировать элемент с помощью дерева решений, мы просто спускаемся по дереву согласно значениям атрибутов элемента. Например, погода у нас солнечная, влажность средняя, и дует ветер. Такого случая в обучающем наборе нет. Мы начинаем с корня, как на рисунке 14.5(a). Проверяем атрибут примерного прогноза — солнечно, поэтому опускаемся по левой ветке и оказываемся в узле со влажностью, где проверяем соответствующий атрибут, как на рисунке 14.5(b). Влажность средняя, поэтому опускаемся по правой ветке и прибываем в листовой узел Г, так что Г является классом, который соответствует нашему случаю, как

изображено на рисунке 14.5(c). Проверять атрибут ветра нам не нужно.

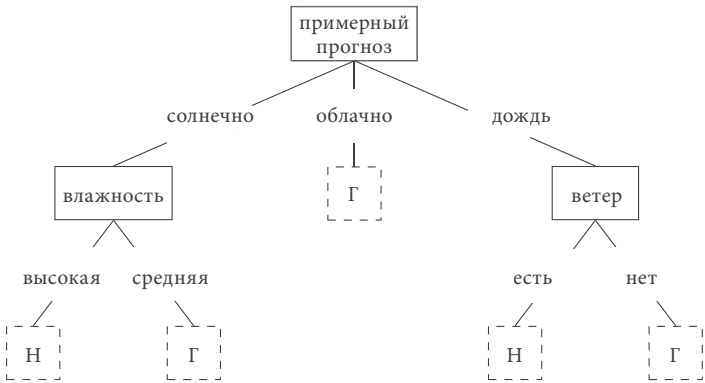


Рисунок 14.4. Простое дерево решений

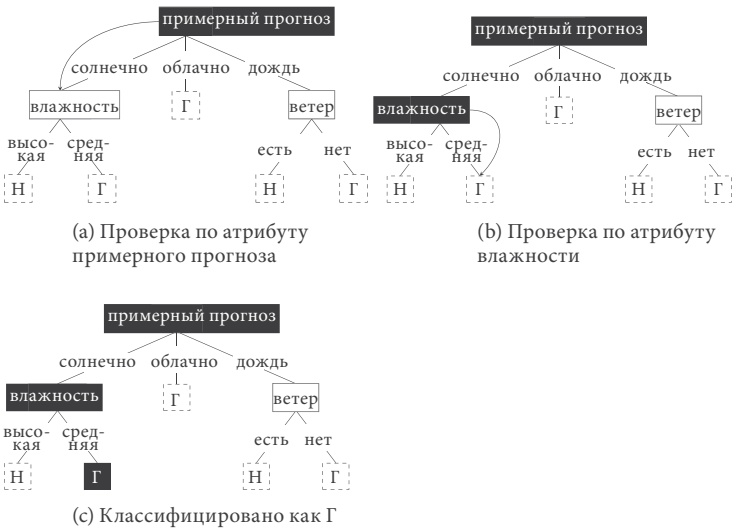
Дерево решений можно приравнять к перечню правил; каждый путь от корня к листу — это список правил, которые должны быть соблюдены, чтобы случай попал в класс, указанный в соответствующем листовом узле. Наш пример дерева решений дает нам пять таких правил:

- если примерный прогноз обещает солнечную погоду при высокой влажности, тогда Н;
- если примерный прогноз обещает солнечную погоду при средней влажности, тогда Г;
- если примерный прогноз обещает облачную погоду, тогда Г;
- если примерный прогноз обещает дождь и ветер, тогда Н;
- если примерный прогноз обещает дождь, но без ветра, тогда Г.

Обычно с деревом решений, в отличие от перечня правил, проще представить, что происходит. К тому же по полученному списку правил никак не понять, откуда они взялись, в то время как на дереве решений видны все атрибуты и порядок, в котором мы их проверяли при построении нашего дерева.

Мы можем отследить построение дерева решений на рисунке 14.6. Чтобы создать дерево решений с помощью обучающего

набора, мы начинаем с корня. Каждый внутренний узел, включая корневой, соответствует проверке атрибутов.



(a) Проверка по атрибуту примерного прогноза

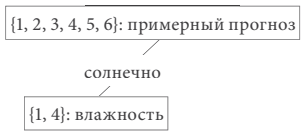
(b) Проверка по атрибуту влажности

(c) Классифицировано как Г

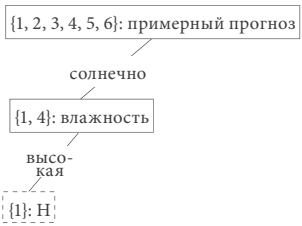
Рисунок 14.5. Классификация, использующая дерево решений для случая с солнечной ветреной погодой и нормальной влажностью

На рисунке 14.6(a) мы используем атрибут примерного прогноза как корневой, который можно прочесть как «проверьте атрибут примерного прогноза для наблюдений в наборе {1, 2, 3, 4, 5, 6}». Подгруппа {1, 4} содержит данные наблюдений для солнечной погоды. Мы создаем в дереве ветку для этой подгруппы и переходим к проверке значения другого атрибута. Берем атрибут влажности, как на рисунке 14.6(b), и видим, что тут только один случай с солнечной погодой и высокой влажностью, поэтому мы создаем ветку для высокой влажности с подгруппой {1}, как на рисунке 14.6(c). Теперь мы переходим к подгруппе, все элементы которой (только один) принадлежат к единственному классу, Н; мы прибыли в листовой узел, и не имеет смысла идти дальше. В дереве также представлен случай для солнечной погоды и средней влажности, так что мы создаем для средней влажности ветку, содержащую подгруппу {4}, как на рисунке 14.6(d), после чего мы снова оказываемся в листовом узле, но уже класса Г.

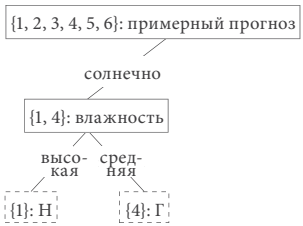
(a) Проверка примерного прогноза



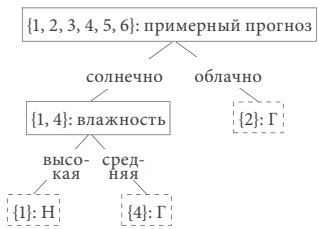
(b) Погода солнечная, проверяем влажность



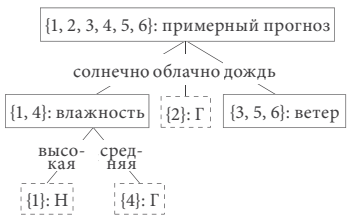
(c) Погода солнечная, влажность высокая



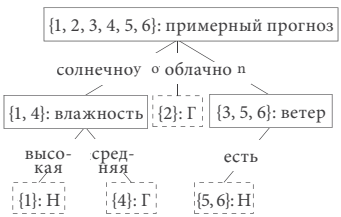
(d) Погода солнечная, влажность средняя



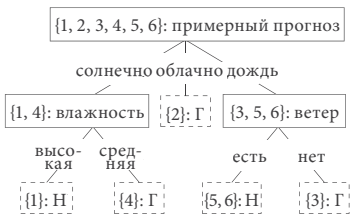
(e) Облачная погода



(f) Дождь, проверяем наличие ветра



(g) Дождь, ветрено



(h) Дождь, безветрено

Рисунок 14.6. Этапы построения дерева решений

Вернемся к проверке в корневом узле: второй возможный вариант для атрибута примерного прогноза — облачно. Мы видим, что тут только один случай для облачной погоды, поэтому мы создаем листовой узел с классом Г, как на рисунке 14.6(e). Третий вариант для атрибута примерного прогноза — дождь с соответ-

ствующей подгруппой {3, 5, 6}. Мы проверяем атрибут ветра в соответствующей подгруппе на рисунке 14.6(f). Если ветер есть, то все элементы подгруппы {5, 6} имеют класс Н, поэтому мы создаем листовой узел на рисунке 14.6(g); в безветрие мы получаем подгруппу {3} и помещаем ее в листовой узел с классом Г.

14.5. Выбор атрибутов

Есть один момент в построении нашего дерева, который мы приняли как данность, однако теперь нам нужно разобраться с ним, если мы хотим прийти к алгоритму построения настоящего дерева решений. Речь идет о выборе атрибутов для проверки в каждом узле. На рисунке 14.6 первым мы проверяли атрибут примерного прогноза, затем атрибут влажности, а за ним атрибут ветра. Но мы могли сделать по-другому; мы могли выбрать другой порядок атрибутов и тогда пришли бы к совершенно иному дереву решений.

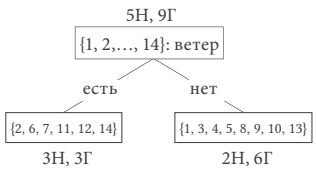
Таблица 14.3. Обучающий набор для метеоданных

номер	атрибуты				класс
	примерный прогноз	температура	влажность	ветер	
1	солнечно	жарко	высокая	нет	Н
2	солнечно	жарко	высокая	есть	Н
3	облачно	жарко	высокая	нет	Г
4	дождь	умеренно	высокая	нет	Г
5	дождь	прохладно	средняя	нет	Г
6	дождь	прохладно	средняя	есть	Н
7	облачно	прохладно	средняя	есть	Г
8	солнечно	умеренно	высокая	нет	Н
9	солнечно	прохладно	средняя	нет	Г
10	дождь	умеренно	средняя	нет	Г
11	солнечно	умеренно	средняя	есть	Г
12	облачно	умеренно	высокая	есть	Г
13	облачно	жарко	средняя	нет	Г
14	дождь	умеренно	высокая	есть	Н

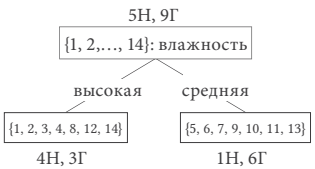
Чтобы исследовать выбор атрибутов, нам понадобится более сложный обучающий набор, показанный в таблице 14.3. В новом

наборе появился такой атрибут, как температура, которая может принимать значения «жарко», «умеренно» или «прохладно». Хоть она и сложнее таблицы 14.2, задачка все равно остается пустяковой. В настоящих классификационных задачах обучающие наборы могут состоять из десятков или даже сотен атрибутов и охватывать тысячи или миллионы случаев.

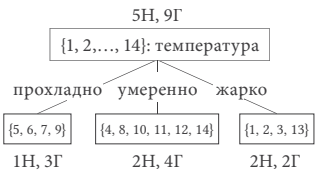
Так как каждый узел в дереве решений отвечает за решение, принятое в соответствии с атрибутом, для построения дерева решения, начиная с корня, нам надо решить, с какого атрибута мы начнем разделение обучающего набора на отдельные случаи. Перед нами четыре разных атрибута, поэтому у нас четыре варианта использования корневого узла. Какой же лучше?



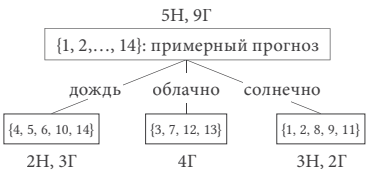
(a) В корневом узле атрибут ветра



(b) В корневом узле атрибут влажности



(c) В корневом узле атрибут температуры



(d) В корневом узле атрибут примерного прогноза

Рисунок 14.7. Вариации решающих атрибутов в корневом узле

На рисунке 14.7 изображен первый уровень деревьев решений, полученный путем помещения каждого из четырех атрибутов в корневой узел для последующей проверки. Опять же, какой из них лучше проверить в первую очередь?

Чтобы ответить на этот вопрос, нам нужно классифицировать обучающий набор так, чтобы мы оказались в узлах одного определенного класса. Если все случаи в узле относятся к одному определенному классу, то каждый узел, если говорить о данном

классе, полностью однороден. Если случаи в узле относятся к разным классам, то узел неоднороден. Чем больше разных классов в узле, тем более разнородный узел. И вот тут мы сталкиваемся с энтропией.

Если узел однородный, то угадать классовый атрибут для каждого случая очень легко, так как все случаи в узле одинаковы. Если же узел неоднородный, то тут не угадаешь; в то же время, чем больше случаев одного и того же класса в узле, тем больше вероятность угадать класс каждого случая.

На рисунке 14.7 показан первый уровень дерева решений для четырех разных атрибутов, помещенных в корневой узел. В каждом дочернем узле мы указываем число случаев с классами Н и Г. Во всем обучающем наборе у нас есть пять случаев класса Н и девять случаев класса Г. На рисунке 14.7(b) узел, к которому ведет ветка «средняя», имеет шесть случаев класса Г и один случай класса Н. Если мы остановим дерево решений здесь и примем решение, к какому классу относится случай, то логичней будет выбрать класс Г, так как соотношение между Г и Н равно шесть к одному, что видно по указанной внизу частоте.

Чем менее предсказуема ситуация, тем выше энтропия, потому что нам требуется больше битов для ее описания. Следовательно, с точки зрения энтропии, чем выше однородность узла, тем ниже его энтропия; чем выше неоднородность узла, тем выше его энтропия. Она определяется через вероятности событий; в деревьях решений событие — это наличие случая с определенным классовым атрибутом. Вероятность случая представляет собой частоту случаев с определенным классовым атрибутом в узле. Все это напрямую связано с определением энтропии, которое мы видели:

$$H(X) = -p(x_1)\lg p(x_1) - p(x_2)\lg p(x_2) - \cdots - p(x_n)\lg p(x_n).$$

В данной формуле X — это узел, в котором содержится группа случаев; у каждого случая есть значение для его классового атрибута; существует n таких возможных значений; и каждое x_i является событием наблюдения i-того значения классового атрибута в узле. В нашем примере с метеоданными есть только два x_i, одно

для Г и одно для Н. Энтропия для каждого узла дерева решений становится

$$H(X) = -p(x_1)\lg p(x_1) - p(x_2)\lg p(x_2)$$

для x_1 = Г и для x_2 = Н.

Если мы используем энтропию для обозначения однородности или неоднородности узла, то мы получаем значения, приведенные на рисунке 14.7, где отмечено значение H для каждого узла; скоро мы увидим, чем является G на рисунке.

Как и ожидалось, при вероятности каждого класса 50%, как для случая в левом дочернем узле рисунка 14.8(a), мы получаем $H = 1$. Точно так же, при вероятности класса 100%, как в среднем дочернем узле рисунка 14.8(d), мы получаем $H = 0$. Для промежуточных значений неоднородности мы получаем промежуточные значения H. Если бы мы графически изобразили значения H, мы бы получили рисунок 14.3, так как в данном конкретном примере существует только два значения для классового атрибута, поэтому ситуация соответствует энтропии ансамбля двух событий с вероятностями p и $1 - p$.

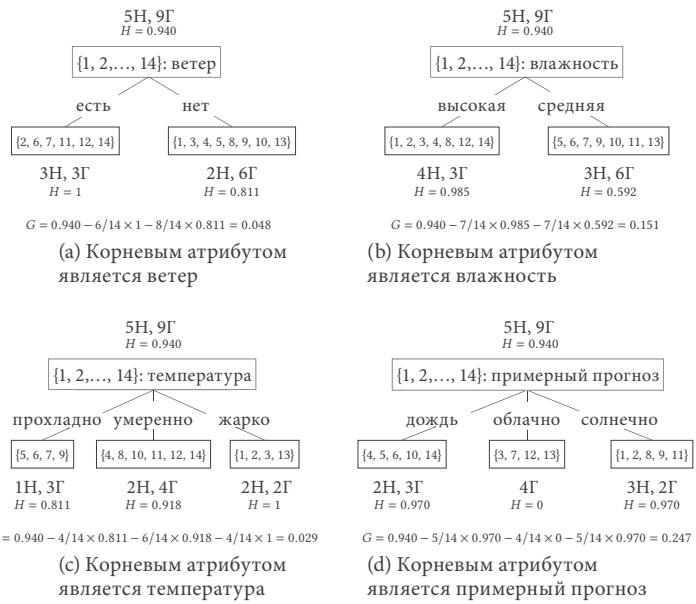


Рис. 14.8. Значения альтернативной энтропии и прироста информации

Мы подошли к ключевому моменту: как мы выбираем атрибут для решений. В каждом дереве рисунка 14.8 мы начинали с одной энтропией в корне и двумя или больше — уровнем ниже. Нам нужно разделить обучающий набор так, чтобы каждый из дочерних узлов был как можно более однородным. Мерилом однородности в отдельно взятом узле будет энтропия. Если у нас есть m дочерних узлов n_1, n_2… n_m, то для вычисления их однородности мы прибегнем к измерению

$$\frac{|n_1|}{n} \times H(n_1) + \frac{|n_2|}{n} \times H(n_2) + \cdots + \frac{|n_m|}{n} \times H(n_m),$$

где $|\ n_i\ |$ — число случаев в обучающем наборе, которые ведут к узлу n_i. Каждый дробный множитель в приведенной выше формуле является частотным отношением, пропорцией, или, по-другому, вероятностью; каждый сомножитель представляет собой значение энтропии в каждом узле. Таково среднее значение энтропии дочернего узла, или ожидаемого значения энтропии дочернего узла по отношению к классовому атрибуту:

$$p_1 H(n_1) + p_2 H(n_2) + \cdots + p_m H(n_m),$$

где $p_i = |\ n_i\ |/n$ — это вероятность, что случай находится в узле n_i.

Вычислив энтропию родительского узла и ожидаемую энтропию дочерних узлов, мы можем вычесть второе из первого и получить значение, которое будет показывать снижение энтропии от родительского к дочерним узлам. Разница между родительской энтропией и ожидаемой энтропией детей называется приростом информации. Мы получаем снижение энтропии, которая образуется из-за разбивки примеров согласно выбранному проверочному атрибуту. Если $H(X)$ — энтропия родительского узла, а a — проверочный атрибут, то прирост информации составит:

$$\begin{aligned} G(X, a) &= H(X) - \left[p_{1|a} H(n_{1|a}) + p_{2|a} H(n_{2|a}) + \cdots + p_{m|a} H(n_{m|a}) \right] \\ &= H(X) - p_{1|a} H(n_{1|a}) - p_{2|a} H(n_{2|a}) - \cdots - p_{m|a} H(n_{m|a}). \end{aligned}$$

На первый взгляд формула кажется сложной, однако расчеты в ней точно такие же, как в формуле выше. $n_{i|a}$ обозначает дочерний узел i, в который мы попадаем после выбора a в качестве

атрибута, проверяемого в родительском узле. Точно так же $p_{1|a}$ обозначает вероятность $p_i = |n_{i|a}|/n$, что случай находится в дочернем узле $n_{i|a}$, когда мы выбираем a в качестве атрибута, проверяемого в родительском узле. Мы вводим в формулу a, чтобы наглядно показать, что дерево растет в ходе выбора атрибута a в разбиваемом узле.

Мы можем записать более абстрактно:

$$G(X, a) = H(X) - H(X|a).$$

То есть прирост информации представляет собой разницу энтропии между узлом X и узлами, которые получаются при использовании a в качестве проверочного атрибута. Нотация $H(X|a)$ следует правилу выражения условных вероятностей: $p(x|y)$ — это вероятность, что мы обладаем x знаниями, что y истинно.

Вернемся к нашему примеру. Значение прироста информации, которое мы получили с помощью поочередного выбора проверочного атрибута, и есть то самое G, которое мы видели в нижних строчках изображений на рисунке 14.8.

Вы можете наблюдать, как энтропия, или информационное наполнение, уменьшается, когда мы в нашем дереве спускаемся на уровень ниже, поэтому термин «прирост информации» звучит парадоксально. Чтобы понять, почему термин все же уместен, нам нужно вернуться к восприятию информации как определенного количества битов, нужных для передачи сообщения. Сообщение, которое мы хотим передать сейчас, представляет собой простое информационное наполнение, состоящее из случая в узле. Если мы хотим представить узел в сообщении, то нам нужно $H(X)$ битов. Если мы хотим представить в сообщении его дочерние узлы, то нам понадобится уже меньше битов. Количество сэкономленных битов — это наше накопление, поэтому получается прирост информации. Мы получаем выгоду за счет того, что в дочерних узлах мы знаем больше, чем в корневом узле: значение проверяемого атрибута в каждом узле. Например, на рисунке 14.8(d) мы начинаем с 14 случаями, в которых могут содержаться все три значения атрибута примерного прогноза. В дочерних узлах — для каждого из них — мы знаем, что во всех случаях прогноз либо

солнечный, либо облачный, либо дождь, и ни в каком узле одно не смешивается с другим. Это знание позволяет нам освободиться от некоего объема информации дочерних узлов.

Прирост информации является основой выбора проверочного атрибута в каждом узле дерева решений. Мы рассчитываем прирост информации для каждого возможного выбора атрибута в узле и берем атрибут с наибольшим информационным приростом. Таким образом, на рисунке 14.8 мы видим, что наибольший прирост информации у атрибута примерного прогноза, поэтому мы выбираем его для разделения обучающего набора и помещаем в корень.

Давайте остановимся на минутку и подумаем о том, чего мы добились. Мы начали измерять информацию, а затем приступили к измерению хаоса, призвав на помощь энтропию. Мы увидели, что можно представить классификационные правила в виде дерева — дерева классификаций. Такие деревья прорастают путем разбивания нашего обучающего набора на все меньшие и меньшие части, выбирая на каждом этапе подходящий проверочный атрибут. Выбор атрибута определяется объемом получаемого информационного прироста. Теперь мы можем заложить данный механизм в основу разработки полномасштабного алгоритма для построения деревьев решений.

14.6. Алгоритм ID3

Энтропия и прирост информации являются базовыми составляющими алгоритма построения дерева решений ID3 (Итерационной Дихотомической процедуры 3), придуманного в 1970-х годах австралийским программистом Россом Квинланом. Алгоритм ID3 начинает работу в корневом узле дерева решений; в корне содержатся все случаи обучающего набора. Чтобы выбрать проверочный атрибут для разделения элементов обучающего набора, он вычисляет информационный прирост для каждого возможного атрибута и выбирает атрибут с наибольшим значением. Затем алгоритм разбивает обучающий набор, создавая дочерние узлы, и проделывает все то же самое рекурсивно с только что создан-

ными узлами: выбирает наилучший для разбивки атрибут, основываясь на информационном приросте, и разбивает подгруппу обучающего набора в нынешнем узле, и так далее.

Вот и все. Суть ID3 в том, чтобы выбирать проверочный атрибут, совершать разбивку, а затем проделывать то же самое с каждым дочерним узлом. Как и всякая рекурсивная процедура, ID3 не может выполняться вечно. В самом деле, есть три варианта, при которых рекурсия остановится на заданном узле; когда это происходит, мы создаем концевой узел с соответствующим классом.

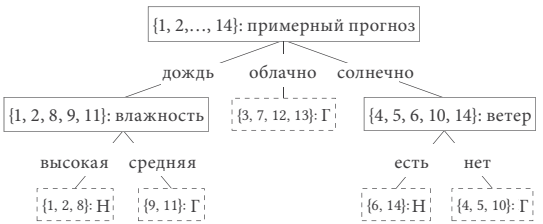


Рисунок 14.9. Концевые узлы содержат случаи одного класса

Вариант первый, мы можем прийти в узел, в котором у подгруппы обучающего набора будет то же значение для целевого классового атрибута (в нашем примере все классы Г или все классы Н). Тогда нам совершенно ясно, что продолжать спуск не имеет смысла, ведь мы добрались до узла, который точно классифицирует случаи, нисходящие от корня. Когда так происходит, мы превращаем узел в концевой с соответствующим классом, как на рисунке 14.9, где во всех концевых узлах содержатся случаи только одного класса. Например, если мы разделим множество {1, 2, 8, 9, 11}, используя атрибут влажности, два значения, высокая и средняя, разместятся в двух узлах: в одном все случаи Н, а в другом все случаи Г. Оба случая мы превращаем в концевые. То же самое происходит и с другими ветвями дерева.

Второй вариант, когда случится остановка работы, — прибытие в узел, в котором нет атрибутов для проверки, а оставшиеся примеры относятся к разным классам. Так как больше нет атрибутов, мы не можем разбить оставшиеся примеры. Приходится остановиться; мы создаем концевой узел, которому приписываем

класс, соответствующий большинству из оставшихся случаев. Если большинство не обнаружено из-за равенства 50:50, тогда мы прибегаем к любому правилу, разбивающему ничью. Чтобы понять, как это работает, представьте, что в наш обучающий набор состоит из случаев таблицы 14.3 плюс дополнительного случая, обозначенного как пятнадцатый, с солнечным прогнозом, умеренной температурой, высокой влажностью, безветренного и с классом Г. Алгоритм ID3 создал бы дерево с рисунка 14.10. Когда мы оказываемся в узле с подмножеством {8, 15}, мы используем последний доступный атрибут, ветер. Оба элемента из {8, 15} подтверждают наличие ветра, но у случая 8 класс Н, в то время как у случая 15 класс Г. Но у нас не осталось атрибутов, чтобы разделить их. Так что мы помещаем их в концевой узел, которому, в идеале, должен быть присвоен класс большинства, однако у нас нет большинства, поэтому мы прибегаем к классу Г — классу последнего исследованного случая (15).

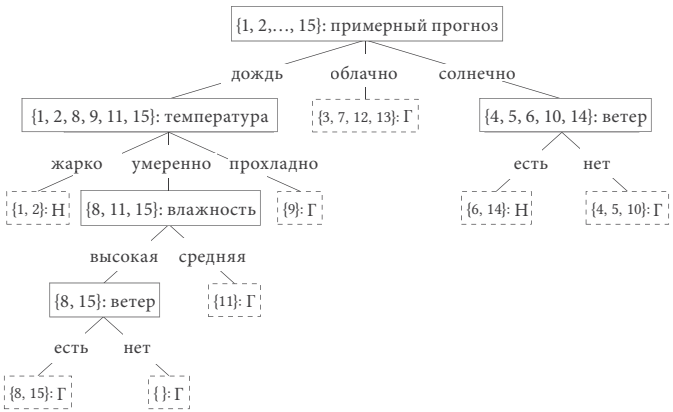


Рисунок 14.10. Концевые узлы со случаями более одного класса

Третий вариант остановить рекурсию — оказаться в ветви, в которой не осталось элементов для разбиения. Когда такое происходит, мы создаем лист с классом большинства случаев в родительском узле; опять же, если у нас нет большинства, мы вольны в выборе правила, разбивающего ничью. В расширенном примере обучающего набора такое случается в ветке «средняя», спускаю-

щейся к узлу с подмножеством {8, 15} на рисунке 14.10. Ни в восьмом, ни в пятнадцатом случаях ветра нет, поэтому мы создаем пустой лист, которому мы присваиваем класс Г. Мы пользуемся тем же, что и прежде, правилом, разрешающим ничью, так как нам нужно найти большинство (хоть и несуществующее) между классовыми атрибутами {8, 15}.

Описанный нами процесс — это еще не алгоритм. Чтобы получить алгоритм, нам надо четко описать его структуру. Процесс рекурсивный, в глубину и выполняется следующим образом.

- Создайте дерево с единственным, корневым узлом. Если все примеры относятся к одному и тому же классу, назначьте класс корневого узла и верните дерево (первый вариант закончить рекурсию). В противном случае, если больше нет атрибутов для проверки, припишите корневому узлу класс большинства оставшихся примеров и верните дерево (второй вариант остановить рекурсию). Иначе:
- Пусть *a* будет атрибутом с максимальным приростом информации. Он будет проверочным атрибутом корневого узла. Для каждого возможного значения v_i из *a*:
- Добавьте ветвь к корневому узлу для v_i. Найдите примеры со значением v_i для *a*. Пусть данный набор примеров будет *examples_*v_i.
- Если пуст, добавьте снизу новой ветви новый концевой узел с классом, соответствующем большинству примеров в родительском узле (третий вариант положить конец рекурсии, так как мы не спускаемся по дереву ниже).
- В противном случае добавьте снизу новой ветви новое дерево, построенное рекурсивно для *examples_*v_i и атрибутов, исключая атрибут *a*.
- После того, как разобрались со всеми атрибутными значениями v_i в данном узле, верните дерево, каким оно выросло за последующие рекурсивные вызовы.

Теперь мы имеем что-то более похожее на алгоритм, но еще не до конца. Лучше рассмотреть процесс в псевдокоде с применением управляющих конструкций и уже известных нам структур данных — в таком случае мы получим алгоритм 14.1. Параметры,

которые мы задаем алгоритму, — это примеры (*examples*), случаи из обучающего набора (те же метеоданные), целевой атрибут (*target_attribute*), классификационный атрибут (классовый атрибут в метеорологических данных), атрибуты (*attributes*), оставшиеся атрибуты случаев из обучающего набора, и значения атрибутов (*attribute_values*), значения, которые они могут принимать.

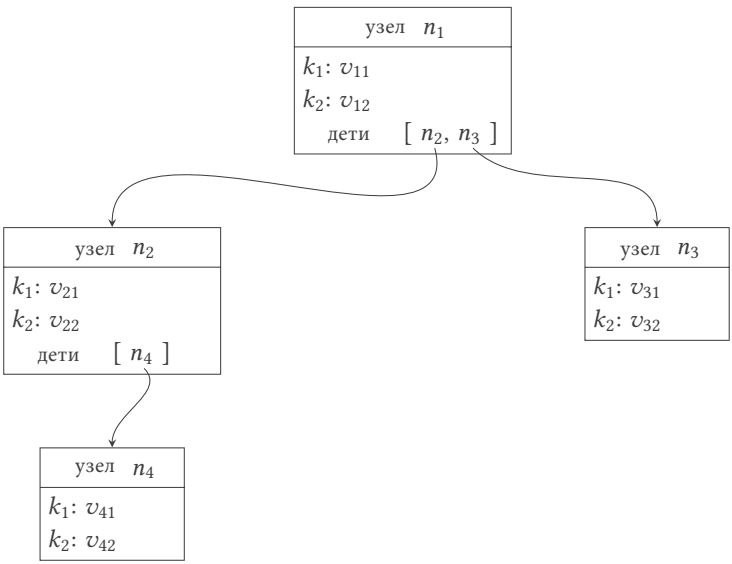


Рисунок 14.11. Дерево, представленное с помощью словарей

Examples представлены в виде списка, каждый из них является отдельным случаем обучающего набора. Отдельный тренировочный случай — это запись, в которой содержатся атрибуты, описывающие данный случай, и целевой атрибут, который представляет собой атрибут, приписывающий каждому случаю его класс. Каждая запись тренировочного случая представлена ассоциативным массивом, или словарем, который связывает названия атрибутов с их значениями. Например, первый случай в таблице 14.3 представляет собой словарь со следующей парой ключей-значений: (прогноз, солнечно), (температура, жарко), (влажность, высокая), (ветер, нет), (класс, Н). *Target_attribute* является всего лишь называнием классификационного атрибута (в нашем примере это

класс), а *attributes* — это группа, в которой содержатся названия остальных атрибутов случая; в нашем примере это примерный прогноз погоды, температура, влажность и ветер.

Алгоритм 14.1. ID3

`ID3(`*examples, target_attribute, attributes, attribute_values*`)` → *dt*

Вводные данные: *examples*, список, содержащий случаи из обучающего набора; *target_attribute*, классификационный атрибут; *attributes*, набор, содержащий оставшиеся атрибуты из обучающего набора; *attribute_values*, словарь, содержащий допустимые значения для каждого из *attributes*.

Выводимые данные: *dt*, дерево решений.

```
1  r ← CreateMap()
2  InsertInMap(dt, "instances", examples)
3  if CheckAllSame(examples, target_attribute) then
4      ex ← GetNextListNode(examples, NULL)
5      cv ← Lookup(ex, target_attribute)
6      InsertInMap(dt, target_attribute, cv)
7      return dt
8  if IsSetEmpty(attributes) then
9      mc ← FindMostCommon(examples, target_attribute)
10     InsertInMap(dt, target_attribute, mc)
11     return dt
12 a ← BestClassifier(examples, attributes, target_attribute)
13 InsertInMap(dt, "test_attribute", a)
14 foreach v in Lookup(attribute_values, a) do
15     examples_subset ← FilterExamples(examples, a, v)
16     if IsSetEmpty(examples_subset) then
17         mc ← FindMostCommon(examples, target_attribute)
18         c ← CreateMap()
19         InsertInMap(c, target_attribute, mc)
20         InsertInMap(c, "branch", v)
21         AddChild(dt, c)
22     else
23         offspring_attributes ← RemoveFromSet(attributes, a)
24         c ← ID3(examples_subset, target_attribute,
25                 offspring_attributes, attribute_values)
26         InsertInMap(c, "branch", v)
27         AddChild(dt, c)
28 return dt
```

Attribute_values представляет собой словарь, который связывает каждый атрибут из *attributes* с соответствующим списком доступных значений, которые этот атрибут может принять; в нашем случае *attribute_values* связывает примерный прогноз с [солнечно, облачно, дождь].

Каждый узел дерева представлен с помощью словаря, который создается пустым путем выполнения функции `CreateMap`. Чтобы управиться со словарями в алгоритме, мы используем одну функцию для добавления пар ключ-значение и одну функцию для извлечения. Функция `InsertInMap`(*m*, *k*, *v*) добавляет в словарь *m* значение *v* с ключом *k*; функция `Lookup`(*m*, *k*) извлекает из словаря *m* значение с ключом *k*.

Теперь немного остановимся и убедимся, что вы осознали — мы будем представлять дерево в с помощью структуры данных, именуемой словарем. Каждый узел — это словарь: информационное наполнение узла представлено с помощью пар ключ-значение. У другого ключа словаря в качестве значения имеется список его дочерних узлов. Каждый из дочерних узлов тем же образом представляет свое информационное наполнение и детей. На рисунке 14.11 показан пример дерева, представленного с помощью словарей. Каждое k_i и v_{ij} обозначают информационное наполнение пары ключ-значение, в то время как в «дочернем» ключе содержится список детей, если они есть.

Алгоритм начинает работу с создания в строчке 1 нового узла *dt*, в котором содержатся *examples*. Новый узел дерева, *dt*, изначально пустой; он будет представлять нынешние случаи обучающего набора, поэтому мы в строчке 2 добавляем в него *examples* с ключом в виде строки «случаи» (instances). Дабы не усложнять, мы полагаем, что здесь и в других частях алгоритма, где в качестве ключей мы используем особые строки (в строчках 13, 20, 26), ни одна из них не окажется названием любого другого атрибута.

В строчках 3–7 мы проверяем, не имеют ли все случаи примеров (*examples*) одинаковое значение в их целевом атрибуте (*target_attribute*). Если имеют, то здесь наш первый вариант остановки рекурсии. Для получения общего значения, мы получаем первый случай — голову списка примеров — и в строчке 5 ищем

его значения для *target_attribute*. Затем, в строчке 6, мы добавляем общее значение в *dt*, используя в качестве ключа *target_attribute*. Наличие *target_attribute* превращает его в листовой узел. Мы возвращаем *dt*.

В строчках 8–11 мы рассматриваем ситуацию, когда у нас больше нет атрибутов для проверки. Мы находим в строчке 9 самое общее значение *target_attribute* в и помещаем его в переменную *mc*. Для этого нам понадобится функция `FindMostCommon`. В строчке 10 мы добавляем *mc* в *dt*, используя в качестве ключа *target_attribute*; затем возвращаем *dt*.

Большая часть работы происходит там, где мы не прекращаем рекурсию. Для продолжения нам надо найти атрибут, который наилучшим образом разобьет случаи обучающего набора. От такого атрибута мы получим наибольший прирост информации. Чтобы найти его, мы пользуемся функцией `BestClassifier`. В строчке 12 мы помещаем называние атрибута в переменную *a* и в строчке 13 фиксируем данный факт в узле *dt* путем добавления ключа «проверочный атрибут» (test_attribute) со значением *a*.

Цикл в строчках 14–27 повторяется всякий раз для каждого возможного значения атрибута *a*, который мы определили как самый выгодный для разделения обучающего набора. Каждого возможного значения *v* мы в строчке 15 используем функцию `FilterExamples`, чтобы получить случаи со значением *v* для атрибута *a*; мы помещаем эти случаи в *examples_subset*.

Если таких случаев нет (что мы и проверяем в строчке 16), то мы переходим к строчке 17 и находим самое общее значение *target_attribute*, *mc*, которое помещаем в листовой узел. Для этого мы создаем новый пустой узел *c* в строчке 18, а в строчке 19 добавляем *mc* в *c*, применяя в качестве ключа *target_attribute*. Помечаем в строчке 20 значение проверочного атрибута, который доставляет нас к этому новому узлу, путем добавления ключа «ветвь» (branch) со значением *v* в новый узел. По сути ключ «ветвь» содержит информацию для метки ветви с рисунков деревьев решений, которые мы видели прежде. Покончив с этим, мы делаем *c* дочерним узлом *dt* в строчке 21, используя функцию `AddChild`.

Если в *examples* есть случаи со значением *v* для атрибута *a*, мы можем продолжить работу в строчках 22–27, пытаясь найти другой проверочный атрибут для дальнейшей разбивки этих случаев. Любое последующее разделение не может применить атрибут *a*, поэтому в строчке 23 мы назначаем *offspring_attributes* атрибуты, которые остались, когда мы убрали *a* из *offspring_attributes*. Теперь мы готовы совершить рекурсивный вызов в строчках 24–25, используя *examples_subset* в качестве наших случаев. Когда рекурсивный вызов возвращается, он выдает нам дерево (которое может состоять из одного узла), которое мы добавим как дочерний узел к узлу *dt*; мы проделываем это в строчках 26–27 е же этапами, что и в строчках 20–21.

Когда мы разберемся со всеми возможными значениями атрибута *a*, узел *dt* станет узлом дерева решений, чьи дочерние узлы также пройдут через алгоритм. Следовательно, мы можем вернуть *dt*.

Последняя страница или около того может представлять собой долгое прохождение простой процедуры, состоящей из выбора проверочного атрибута, разбивки множества и повторения той же процедуры для каждой последующей разбивки. Порой в программировании разница между интуитивным пониманием и строгим описанием (как в алгоритме) может быть просто огромной. Она неизбежна. Проще всего понять принцип работы ID3 (если он все еще ускользает от вас) — поэтапно пройтись по рисункам 14.9 и 14.10, помня о том, что алгоритм строит деревья решений в глубину, слева направо.

14.7. Основные принципы работы

Чтобы завершить объяснение, мы обязаны описать работу различных функций, примененных в алгоритме 14.1. Функция `CreateMap` вводит словарь. Она касается размещения массива, который будет использован в сопряжении хеш-функции. Добавление и поиск в словаре с помощью функций `InsertInMap` и `Lookup` работают как обычные словари, пристроенные сверху хеш-таблиц. Для добавления дочернего узла к узлу дерева мы

прибегаем к функции `AddChild`. Каждый узел — это словарь, поэтому мы можем использовать заданный атрибут вроде «дети», у которого в качестве значения будет список дочерних узлов, как мы уже обсуждали и видели выше на рисунке 14.11. Таким образом, функция `AddChild` — это функция, которая ищет данный атрибут. Если ей не удается ничего найти, она создает список с дочерним узлом как единственным элементом и добавляет этот список в качестве значения дочернего атрибута. Если же функция находит список, она добавляет в него дочерний узел.

`CheckAllSame` — самый простой способ получить значение *target_attribute* для первого случая в *examples* и затем пройтись по остальным *examples*, проверяя одинаковы ли значения *target_attribute* и первого случая. Если в каком-то случае мы увидим, что они различны, `CheckAllSame` вернет `false`; в противном случае она вернет `true`. Что и происходит в алгоритме 14.2. В строчке 1 мы получаем первый узел в списке *examples*, а в строчке 2 достаем примерный случай из этого узла; функция `getData` возвращает объем данных в узле списка. Затем, в строчке 3, мы получаем значение *v* для *target_attribute* в первом примере. В цикле строчек 4–7 просматриваем оставшиеся примеры. Обратите внимание на прием, который мы используем для прохождения по узлам списка. Каждый раз, когда мы берем узел, следующий за *n*, мы приписываем его к *n*: затем проверяем равен ли он `null`, который означает, что мы добрались до конца списка. В строчке 5 внутри цикла для каждого узла *n* мы достаем примерный случай, который он содержит, и в строчке 6 проверяем, такое же у него значение для *target_attribute* или нет. Если нет, мы возвращаем `false` в строчке 7. Если же у всех узлов одинаковое значение, мы возвращаем `true` в конце алгоритма, в строчке 8.

Функцию `FindMostCommon`, как видно в алгоритме 14.3, описать довольно просто. В качестве вводных данных алгоритм берет список тренировочных случаев и атрибут, *target_attribute*. Каждый случай имеет значение, соответствующее *target_attribute*, и нам надо найти самое часто встречающееся из данных значений. Мы используем систему подсчета словаря (*counts*), чтобы подсчитать частоту каждого значения *target_attribute*. Мы запускаем ее

в строчке 1; в строчке 2 мы устанавливаем *mc*, самое распространенное значение, которое мы ищем, на `null`; в строчке 3 устанавливаем *max*, самую большую частоту значения *target_attribute* на данный момент, на 0.

Алгоритм 14.2. Проверим, что все случаи обучающего набора имеют одинаковое значение для заданного атрибута

```
CheckAllSame(examples, target_attribute) → TRUE or FALSE

    Вводные данные: examples, список случаев из обучающего набора, представленный словарями; target_attribute, атрибут для проверки.

    Выводимые данные: булево значение, которое true, если у всех случаев в examples одинаковое значение для target_attribute, а иначе — false.

1   n ← GetNextListNode(examples, NULL)
2   example ← GetData(n)
3   v ← Lookup(example, target_attribute)
4   while (n ← GetNextListNode(examples, n)) ≠ NULL do
5       example ← GetData(n)
6       if Lookup(example, target_attribute) ≠ v then
7           return FALSE
8   return TRUE
```

Цикл в строчках 4–14 проходит по каждому примеру в списке примеров. В строчке 5 он получает значение *target_attribute* в нынешнем примере и помещает его в *v*. Затем он ищет *v* в *counts* и помещает результат в *count*. Если мы впервые столкнулись со значением *v*, *count* будет равен `null`, поэтому в строчках 7–8 мы меняем его на один. Если нам уже попадалось значение *v*, в строчке 10 мы увеличиваем *count* на один. В строчке 11 мы добавляем обновленный *count* в *counts*, затем, в строчке 12, сравниваем нынешний индекс, *count*, с наибольшим имеющимся у нас значением. Если мы получили новый максимум, то мы обновляем его в строчке 13, а в строчке 14 обновляем соответствующее самое частое значение. В строчке 15 мы завершаем цикл и возвращаем самое распространенное значение, которое нам удалось найти.

Чтобы отделить случаи с одинаковым значением для определенного атрибута, как в функции `FilterExamples`, нам нужно просмотреть случаи, которые мы хотим отфильтровать, прове-

рить каждый случай, соблюдает ли он нужные нам условия, и добавить в список отфильтрованных случаев те, что соблюдают. Именно это и делает алгоритм 14.4. В качестве вводных данных он берет список примеров, атрибут *a* и значение *v*; нам надо отфильтровать список примеров и вернуть только те, у которых значение *v* для атрибута *a*. Мы создаем пустой список, *filtered*, в строчке 1. В цикле строчке 2–4 мы проходимся по каждому примеру и проверяем, соблюдены ли заданные условия. Если да, мы добавляем пример в список *filtered*, который мы вернем в самом конце.

Алгоритм 14.3. Найдем самое распространенное значение заданного атрибута в списке случаев обучающего набора, представленных с помощью словарей

`FindMostCommon`(*examples, target_attribute*) → *mc*

Вводные данные: *examples*, список случаев из обучающего набора, представленный словарями; *target_attribute*, атрибут, чьи значения мы хотим исследовать.

Выводимые данные: *mc*, самое распространенное значение *target_attribute* в содержании *examples*.

```
1  counts ← CreateMap()
2  mc ← NULL
3  max ← 0
4  foreach example in examples do
5      v ← Lookup(example, target_attribute)
6      count ← Lookup(counts, v)
7      if count = NULL then
8          count ← 1
9      else
10         count ← count + 1
11     InsertInMap(counts, v, count)
12     if count ≥ max then
13         max ← count
14         mc ← v
15 return mc
```

Теперь, чтобы завершить описание алгоритма 14.1, нам осталось описать функцию `BestClassifier`. Как мы говорили, она выбирает из узла атрибут, который дает наибольший прирост

информации по сравнению со всеми остальными атрибутами. Шаг за шагом мы построим `BestClassifier`.

Так как прирост информации требует расчетов энтропии в узле, сначала нам надо задать алгоритм, вычисляющий энтропию. Алгоритм 14.5 реализует формулу энтропии, которую нам уже довелось видеть:

$$H(X) = -p(x_1)\lg p(x_1) - p(x_2)\lg p(x_2) - \cdots - p(x_n)\lg p(x_n).$$

Формула энтропии требует он нас нахождения пропорции каждого различного значения атрибута, который мы используем для расчета энтропии. Это значит, что нам надо подсчитать, как часто встречается каждое из различных значений, и разделить полученное число на общее количество случаев. В качестве вводных данных алгоритм 14.5 берет список примеров и атрибут *target_attribute*, который станет основой расчетов энтропии.

Алгоритм 14.4. Фильтруем примеры обучающего набора

`FilterExamples(`*examples, a, v*`)` → *filtered*

Вводные данные: *examples*, список случаев из обучающего набора, представленный словарями; *a*, атрибут для поиска в *examples*; *v*, значение *a*, которое будет использовано для фильтрации.

Выводимые данные: *filtered*, список, содержащий случаи в *examples* со значением *v* для атрибута *a*.

```
1  filtered ← CreateList()
2  foreach example in examples do
3      if Lookup(example, a) = v then
4          InsertInList(filtered, NULL, m)
5  return filtered
```

Чтобы подсчитать число встреч каждого различного значения ключа, мы в строчке 1 прибегнем к словарной системе подсчета, *counts*, которая изначально пуста. Мы будем следить за разными значениям, которые нам попадутся в списке *values*, который мы также инициализируем пустым в строчке 2. В цикле строчек 3–11 мы вычисляем итог. Для каждого отдельного примера мы получаем значение *target_attribute* и помещаем его в переменную *v* в строчке 4. В строчке 5 мы ищем *v* в *counts* и помещаем результат в *count*. Если его там нет, строчка 6, значит, мы

впервые сталкиваемся с данным значением, поэтому мы устанавливаем *count* на единицу в строчке 7. В строчке 8 мы добавляем *v* в список *values*. Если нам уже попадалась *v*, в строчках 9–10 мы увеличиваем *count* на единицу. Обновив *count*, в строчке 11 вставляем его в *counts*.

Алгоритм 14.5. Вычисляем энтропию списка случаев обучающего набора в соответствии с заданным атрибутом

`CalcEntropy(`*examples, target_attribute*`)` $\rightarrow h$

Вводные данные: *examples*, список случаев из обучающего набора, представленный словарями; *target_attribute*, атрибут, чьи значения будут использованы при вычислении энтропии.

Выводимые данные: h, энтропия *examples* по отношению к разным значениям *target_attribute*.

```
1  counts ← CreateMap()
2  values ← CreateList()
3  foreach example in examples do
4      v ← Lookup(example, target_attribute)
5      count ← Lookup(counts, v)
6      if count = NULL then
7          count ← 1
8          InsertInList(values, NULL, v)
9      else
10         count ← count + 1
11     InsertInMap(counts, v, count)
12 h ← 0
13 foreach v in values do
14     p ← Lookup(counts, v)/|examples|
15     h ← h − p · lg(p)
16 return h
```

Когда мы выходим из цикла, то можем рассчитать члены $p_i \lg(p_i)$ из формулы энтропии. В строчке 12 мы задаем нулевое значение энтропии, h. Проходимся по всем вероятным значениям, которые нам попались во втором цикле в строчках 13–15; ищем счетное значение для каждого значения *v* и делим его на общее число примеров; это и будет значение p_i при каждом проходе цикла. Мы вычитаем его из h и обновляем h в строчке 15. Когда все сделано, мы возвращаем значение энтропии.

Вы, наверно, обратили внимание, что алгоритм 14.5 похож на алгоритм 14.3. В самом деле, оба проходятся по списку элементов и рассчитывая те элементы, которые отвечают заданным условиям. Разновидность той же идеи разрешает проблему вычисления прироста информации для каждого проверочного атрибута в узле. Мы видели, что прирост информации задается формулой

$$G(X, a) = H(X) - p_{1|a}H(n_{1|a}) - p_{2|a}H(n_{2|a}) - \cdots - p_{m|a}H(n_{m|a}).$$

Ее можно превратить в алгоритм 14.6, который, в свою очередь, похож на алгоритм 14.5. Главное различие между двумя алгоритмами в том, что алгоритм 14.5 подсчитывает случаи, которые соответствуют условию, в то время как алгоритм 14.6 группирует все случаи, соответствующие условию.

`CalcInfogain` в строчке 1 создает словарь, *groups*, где каждая пара ключ-значение представляет собой *test_attribute* и список, содержащий случаи с одинаковым значением для *test_attribute*. Мы снова будем отслеживать разные значения, попадающиеся нам в списке *values*, который мы изначально делаем пустым в строчке 2. Мы заполняем *groups* в цикле строчек 3–10, который повторяется для каждого случая в *examples*. В строчке 4 ищем значение *test_attribute* случая и помещаем его в *v*; затем в строчке 5 ищем группу данного значения. Если такой группы нет (строчка 6), мы добавляем в *groups* новую пару ключ-значение с *v* в качестве ключа и список с единственным элементом, нынешний случай, в строчке 7; мы также добавляем *v* к *values* в строчке 8. Если такая группа есть (строчка 9), мы в строчке 10 добавляем случай в эту группу.

Выйдя из цикла, мы вычисляем энтропию узла и помещаем ее в g в строчке 11. Затем, в строчках 12–16 для каждого из различных значений *test_attribute* мы высчитываем отношение случаев соответствующих групп к общему числу случаев, энтропию группы с учетом *target_attribute* и вычитаем полученный результат из энтропии узла.

Имея в арсенале алгоритмом прироста информации, функция `BestClassifier` в алгоритме 14.1 являет собой простой повторяющийся выбор наибольшего из возможных приростов инфор-

мации, которые мы можем получить путем указания различных проверочных атрибутов в узле, как вы можете видеть в алгоритме 14.7.

Алгоритм 14.6. Вычисляем информационный прирост списка тренировочных случаев в соответствии с проверочным атрибутом и целевым атрибутом

`CalcInfoGain(`*examples, test_attribute, target_attribute*`)` → *g*

Вводные данные: *examples*, список словарей; *test_attribute*, проверочный атрибут; *target_attribute*, целевой атрибут.

Выводимые данные: *g*, информационный прирост *examples* по отношению к *test_attribute* и *target_attribute*.

```
1  groups ← CreateMap()
2  values ← CreateList()
3  foreach example in examples do
4      v ← Lookup(example, test_attribute)
5      group ← Lookup(groups, v)
6      if group = NULL then
7          InsertInMap(groups, v, [example])
8          InsertInList(values, NULL, v)
9      else
10         InsertInList(group, NULL, example)
11 g ← CalcEntropy(examples, target_attribute)
12 foreach v in values do
13     group ← Lookup(groups, v)
14     p ← |group|/|examples|
15     h ← CalcEntropy(group, target_attribute)
16     g ← g − p · h
17 return g
```

Мы назвали функцию `BestClassifier`, потому что каждый раз она выбирает наилучший для разбивки случаев на группы (классы) атрибут. Мы привели доводы в ее пользу с точки зрения энтропии и прироста информации. Но какова она на практике? Можно порассуждать, что применение этих двух систем измерения — лишь одни из возможных способов разделения случаев и есть разные другие, точно такие же или даже более эффективные. Возможно. Использование прироста информации в качестве

ключевого параметра основывается на предположении, что получаемые в итоге деревья решений несколько лучше других деревьев решений. Каким образом они лучше, увидим дальше.

14.8. Бритва Оккама

Давайте предположим, что мы применяем альтернативную реализацию `BestClassifier`. В ней вместо того, чтобы вычислять наибольший прирост информации для выбора проверочного атрибута в каждом узле, мы просто берем атрибуты в заранее заданном порядке: температура, влажность, ветер, примерный прогноз. Получившееся дерево решений изображено на рисунке 14.12.

Дерево на рисунке 14.12 создано с помощью того же обучающего набора, что и дерево с рисунка 14.9, однако оно явно отличается от собрата. Оно больше и глубже дерева, которые мы получили благодаря поиску информационного прироста. Так как дерево решений отражает набор правил, по которым происходит классификация случаев, обычное правило в дереве рисунка 14.9 короче обычного правила в дереве рисунка 14.12. Характерная особенность ID3 заключается в том, что, прибегая к измерению энтропии и прироста информации, алгоритм выбирает проверочные атрибуты, которые предпочитают не большие, а короткие деревья. Иными словами, ID3 выбирает проверочный атрибут так, чтобы правила принятие решений оказалось не длинными, а короткими.

Зачем же нам нужно придерживаться коротких правил принятия решений, а не длинных? Существует одно веское правило, или точнее принцип, решения задач, который называется бритвой Оккама; он гласит, что, если несколько конкурирующих теорий одинаково верны, нам следует выбрать теорию с наименьшим количеством условий. Назван принцип в честь средневекового францисканского монаха и богослова Уильяма Оккамского. Согласно данному правилу следует отсекать все лишние сущности, отсюда и взялся термин «бритва». Другая вариация бритвы Оккама звучит следующим образом: «Если все объяснения равны,

то самое простое объяснение и будет самым верным». Еще одна формулировка, приписываемая самому Оккаму, хотя и не сохранившаяся в его записях, гласит: «Не следует множить сущности без необходимости».

Алгоритм 14.7. Находим лучший классификационный атрибут путем нахождения максимального прироста информации

```
BestClassifier(examples, attributes, target_attribute) → bc
    Вводные данные: examples, список тренировочных случаев,
    представленный словарями; attributes, список атрибутов
    для проверки наибольшего информационного прироста;
    target_attribute, классовый атрибут случаев.
    Выводимые данные: bc, атрибут среди attributes, который
    дает наибольший информационный при рост по отношению
    к target_attribute.
1   maximum ← 0
2   bc ← NULL
3   foreach attribute in attributes do
4       g ← CalcInfoGain(examples, attribute, target_attribute)
5       if g ≥ maximum then
6           maximum ← g
7           bc ← attribute
8   return bc
```

Рисунок 14.12. Дерево решений, которое получается при взятии проверочных атрибутов по порядку: температура, влажность, ветер, примерный прогноз

Бритва Оккама широко используется в науке. Мы почти всегда предпочитаем простое научное объяснение сложному, и теория,

объясняющая природное явление нескольких утверждениях, нам ближе теории с кучей логических доводов и терминов. Тот же принцип лежит в основе лаконичности изложения идей: объяснение считается лаконичным, если оно объясняет многое с помощью немногих слов.

Простота в нашем понимании также идет рука об руку с изящностью, поэтому метод, согласно которому дерево решений строится наиболее простыми способами, представляется хорошим решением. К тому же он прекрасно работает на практике, ID3 лежит в основе большого числа других модернизированных алгоритмов классификации, которые применяются в классифицировании сложных данных окружающего мира.

14.9. Затраты, проблемы, улучшения

При некоторых допущениях вычислительные затраты ID3 получить довольно легко. Построение дерева решений зависит от количества уровне дерева. Представим, что у нас n случаев в обучающем наборе и у каждого из них m различных атрибутов. Мы можем предположить, что у дерева решений $O(\log n)$ уровней. В двоичных деревьях число уровней равно $O(\log n)$; здесь мы не можем предположить, что атрибуты двоичны, поэтому мы используем более общий логарифм с основанием 10. Однако разницы никакой, так как с точки зрения нотации большого О они одинаковы, потому что $\lg n = \log n / \log 2$ и $O(\lg n) = O((1/\log 2)\log n) = O(\log n)$.

При наихудшем раскладе все уровни заполнены так, что на каждом из них мы имеем дело со всеми n случаями. Поэтому, чтобы выстроить все уровни, проверяя по одному атрибуту на каждом уровне, потребуется $O(n \log n)$. Тем не менее мы проверяем более одного атрибута за уровень: на корневом уровне мы проверяем все m атрибутов и на уровне i проверяем $m - i + 1$ атрибутов (первый уровень — корневой). Чтобы лучше понять, представим, что мы проверяем m атрибутов на всех уровнях; мы ищем верхнюю границу, поэтому мы можем иметь дело не с самой устойчивой границей. Тогда общие вычислительные затраты на построение дерева решений будут $O(mn \log n)$, что является продуктивным.

Может, и продуктивно, однако ID3, каким мы его описали, не достаточно типичен, так как требует, чтобы все атрибуты были категориальными, то есть могли принимать только небольшое число заранее заданных значений. Это не касается числовых значений. К примеру, температура может быть дана в градусах Цельсия. Они — действительные числа; даже если мы округлим их до ближайшего целого числа, мы не сможем использовать температуру как проверочный атрибут, потому что нам понадобится количество ветвей, равное числу всех возможных значений температуры, что приведет нас к слишком большому дереву решений. Но проблема решается просто: превратим дерево решений в двоичное дерево. Берем среднее значение, то есть значение, которое разделяет случаи напополам: половина случаев до и половина после. Проверка становится $x \geq$ медианы, которая разбивает случаи.

Схожая проблема существует со ветвистыми атрибутами, атрибутами с большим количеством возможных значений. В данную категорию попадают как числовые атрибуты, так и нечисловые, например даты или атрибуты, которые однозначно описывают случай. Когда такое происходит, мы снова получаем узел с огромным количеством ветвей. К примеру, представьте, что мы добавили к потенциальным проверочным атрибутам таблицы 14.3 номер наблюдения, указанный в первом столбце. Мы начинаем со стартового узла и пытаемся найти атрибут, который даст нам наибольший прирост информации. Рассматриваем нумерацию: мы получим 14 дочерних узлов, идущих от корневого узла, по узлу на каждое отдельное значение номера. Энтропия в каждом дочернем узле равна $1 \lg 1 = 0$, потому что у каждого дитяти по одному классовому атрибуту. Следовательно, информационный прирост нумерации будет $G(root, serial\_number) = H(root) - 0 - 0 - \ldots - 0 = H(root)$. Это максимальный возможный прирост информации, поэтому мы можем взять нумерацию в качестве проверочного атрибута, что приведет нас к дереву на рисунке 14.13. Данное дерево идеально классифицирует обучающий набор, применяя одно очень просто правило: оно проверяет значение номера для каждого случая. В то же время такое дерево ничего больше не может классифицировать, потому

что в обучающем наборе больше нет случаев с каким-либо номером. Так что такой подход совершенно бесполезен. В самом деле, кто в здравом уме станет брать в качестве проверочного атрибута нумерацию или любой другой идентификатор, однако можно попробовать взять дату или другой атрибут, который хоть и не описывает случай, но обладает множеством различных значений, покрывающих случаи.

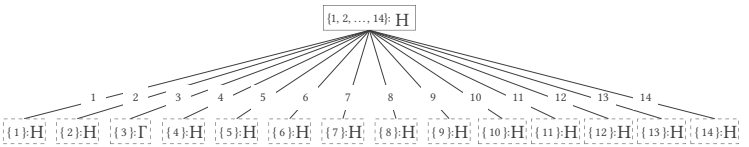


Рисунок 14.13. Ветвистый атрибут

Проблема возникает из факта, что ветвистые атрибуты приводят к большому информационному приросту. Чтобы разрешить данную проблему, нам надо предотвратить ее. В качестве мерила при выборе проверочного атрибута мы можем использовать вместо прироста информации его модификацию, называемую отношением прироста. Мы начинаем с введения нового параметра, называемого расщепленной информацией, заданного для узла X и потенциального проверочного атрибута a, который принимает c различных значений, как:

$$SI(X,a) = -p(y_{1|a})\lg p(y_{1|a}) - p(y_{2|a})\lg p(y_{2|a}) - \cdots - p(y_{c|a})\lg p(y_{c|a}),$$

где каждая $y_{i|a}$ — это пропорция случаев в нашем наборе с определенным значением для a. Расщепленная информация, повторимся, представляет собой энтропию. Но если до сего момента мы рассматривали энтропию с точки зрения целевого атрибута, то теперь мы рассматриваем ее с точки зрения потенциального проверочного атрибута. Отношение прироста тогда будет:

$$GR(X,A) = \frac{G(X,a)}{SI(X,a)}$$

Отношение прироста выбраковывает ветвистые атрибуты. У каждого атрибута, который полностью разбивает обучающий набор на подгруппы, состоящие из одного случая, при наличии

n случаев будет $SI(X, a) = -(1/n)\lg(1/n) - (1/n)\lg(1/n) - \ldots - (1/n)\lg(1/n) = -\lg(1/n) = \lg n$. В то же время, у атрибута, который разбивает обучающий набор на две равные подгруппы, будет $SI(X, a) = -(1/2)\lg(1/2) - (1/2)\lg(1/2) = \lg 2 = 1$.

Отношение прироста не полностью разрешает проблему, так как, если один из p_i будет слишком большим, отношение станет слишком большим, искажая результаты. Чтобы такого не случилось, мы можем сперва вычислить информационный прирост и воспользоваться отношением прироста на следующем этапе, для атрибутов с приростом информации выше среднего.

Еще одна проблема, возникающая в применении, связана с тем, что данным окружающего мира зачастую не хватает значений для атрибутов. Их называют отсутствующими значениями. Возьмем, к примеру, температуру: может оказаться, что у нас нет единиц измерения температуры для всех наших данных. Мы можем отмахнуться от проблемы, заявив, что отсутствующие данные — все равно данные, даже если выглядят они необычно. Мы можем приписать особое значение для недоступных данных (например NA) и представить, что это, к примеру, другие единицы измерения температуры. Такой подход сработает, но не всегда, и в подобных случаях нам придется воспользоваться другими, более изощренными путями.

Под конец затронем еще один важный момент, который возникает во всех методах классификации и представляет собой серьезную проблему в машинном обучении, — переобучение. Машинное обучение начинается с обучающего набора, с помощью которого мы учим компьютер выполнять определенное задание. При классифицировании у нас есть обучающий набор, в котором содержатся данные с известными классами, и мы получаем модель прогнозирования в виде дерева решений, которое мы используем для предсказания класса других данных. Переобучение происходит, когда наша модель прогнозирования, наше дерево, оказывается слишком однозначным, чтобы можно было извлечь из него какую-то пользу. На рисунке 14.13 изображено дерево являющее собой яркий пример переобучения; представлен настолько утрированный случай, что все его изъяны сразу же бросаются в глаза. Однако переобучение не всегда так очевидно, и мы можем

получить толковое с виду дерево решений, которое на деле не способно справиться с классификационным заданием.

Переобучение — коварная проблема и не имеет простого решения. Лучшее, что мы способны предпринять, — это принять факт, что оно может случиться, и предотвратить его. Можно, например, прибегнуть к контрольным данным, части нашего обучающего набора, которую мы используем не во время обучения, а после, когда проверяем, насколько хороша работа дерева решений. Панацеи не существует; переобучение все равно может случиться, и часто проявляется уже на практике. От него страдают и считают его настоящим проклятьем даже опытные специалисты в области машинного обучения.

Примечания

Клод Элвуд Шеннон заложил основу теории информации в работе, в которой представил введение в «математическую теорию коммуникации» [181]; этот фундаментальный труд доступен в книжном варианте, снабженном полезным вступлением [183]. Однако это не первое его достижение. В возрасте двадцати одного года, когда Шеннон был студентом Массачусетского технологического института, он показал, что булева алгебра может лечь в основу электрических цепей. Он также работал над оценкой энтропии английских букв, задавая значения между 1,3 и 1,5 битами для текстовых отрывков длиной в 16 букв [182]. Позже Томас Ковер и Роджер Кинг остановились на оценке в 1,3 бита на каждую букву [43]. Чтобы по достоинству оценить глубину и богатство информационной теории, стоит прочесть несколько книг, посвященных данной теме. Джеймс Стоун написал учебное введение в тему [193]; книгу Маккея [131] можно отнести к более продвинутой литературе, она плавно перетекает в другие темы, такие как вероятности, логические заключения и нейронные сети. Томас Ковер (тот самый, который остановился на оценке в 1,3 на каждую букву) и Джой Томас написали обширное введение в тему [44]. Если вас интересуют точные математические описания, загляни-

те в книгу Роберта Грэя [83]. В книге Сезара Идальго вы можете познакомиться с нетехническим обзором, посвященным роли информации в современных обществах и их экономиках.

Алгоритм ID3 является родственником более старого подхода, называемого идеей обучающей системы, разработанной в 1950-х годах [100]. Росс Квинлан после публикации ID3 [158], [159], [160] усовершенствовал его несколькими способами. Популярном дополнением ID3, которое получило широкое распространение, стал C4.5 [161]. Примеры метеорологических данных, которые мы использовали, взяты из первоначальных публикаций ID3. CART* — еще один распространенный алгоритм, который очень похож на ID3 [28].

Машинное обучение — обширнейшая область. Классическим введением в данную тему можно считать книгу Тома Митчела [142]. К популярным книгам, посвященным машинному обучению, относятся учебное пособие Хасти, Тибширани и Фридмана [90] и учебник для новичков Джеймса, Уиттена, Хасти и Тибширани [102]; книга Бишопа о распознавании образцов и машинном обучении [20]; введение, написанное Алпайдином [2]. Книга Мерфи рассматривает тему с точки зрения байесовской вероятности, что оказалось очень плодотворным [148]. Введение в поиск данных, подобласть машинного обучения, написали Уиттен, Фрэнк, Пэл и Холл [219]. Также можете посмотреть десять лучших алгоритмов по добыче данных, описанных на тематической конференции в 2006 году [220].

До нас не дошли записи Уильяма Оккамского (примерно 1287–1347 годы), в которых содержалось бы одноименное правило, и на самом деле похожие правила существовали еще задолго до Оккама. Например, Аристотель (384–322 до н.э.) высказывал подобный принцип во «Второй аналитике» (книга I, 25), что из нескольких конкурирующих теорий лучшей окажется теория с наименьшим количеством условий, допущений и утверждений. Позже Фрэнк Фильчек, лауреат Нобелевской премии по физике,

* От англ. Classification and Regression Trees — классификационные и регрессионные деревья.

сформулировал правило: «Мы говорим, что объяснение — или, если смотреть шире, теория — лаконично, если с малыми затратами объясняется очень многое… Мы интуитивно ощущаем, что так оно и есть: лаконичные объяснения лучше сложных — тех, которым требуется множество допущений, чтобы объяснить набор явлений или наблюдений» [216].

У «Гугла» есть поучительная история о моделях прогнозирования. Есть такой сервис, Google Flu Trends, который предсказывает распространение гриппа, основываясь на поисковых запросах в Интернете, связанных с гриппом [78]. Он был запущен в 2008 году, и его показания за год сравнялись с оценками Центра по контролю и профилактике заболеваний США; однако позже, в 2013 году предсказания стали неверными [49]. Виной тому стало переобучение и «надменность больших объемов данных»: допущение, что горы данных могут заменять, а не дополнять имеющуюся базу данных и анализы данных [122].

Так как алгоритмы, особенно те, что работают с большими объемами данных, плотнее входят в человеческую жизнь, охватывая все больший спектр ее областей, поднимаются вопросы, касающиеся моральности их применения. С помощью алгоритмов можно обнаружить потенциальных правонарушителей, отсортировать полученные резюме, подсчитать страховые взносы. Когда алгоритмы используются в подобных целях, важно понимать принцип их работы и каким образом они получают результаты, которые мы видим [152]. Алгоритмы могут облегчить нашу работу, но ответственность за ее исполнение всегда остается на наших плечах.

Упражнения

1. Латинская фраза Ibis redibis nunquam per bella peribis — «победа никогда поражение» — имеет два значения, в зависимости от того, где поставить запятую. Ibis, redibis, nunquam per bella peribis переводится как «победа, никогда поражение», в то время как Ibis, redibis nunquam, per bella

peribis означает «победа никогда, поражение». Эту фразу приписывают оракулам Додоны времен древней Греции (однако они говорили на греческом, а не на латыни, так что есть повод сомневаться в подлинности такого предположения). Какова энтропия данной фразы?

2. На рисунке 14.3 показана энтропия монетки для различных вероятностей орла и решки. Удастся ли вам построить график энтропии кубика, в котором первое событие — это «выпала единица», а второе — «выпало два, три, четыре, пять, шесть»? При какой вероятности первого события энтропия будет наибольшей? Каково максимальное значение энтропии?
3. Реализуйте алгоритм ID3, применяя в качестве классификатора более простой метод, когда вы берете по очереди каждый атрибут, как на рисунке 14.12.
4. Реализуйте алгоритм ID3, встроив в него внесение поправок в отношение прироста.
5. Индекс Джини — еще одно правило, которое можно применять при разбивке примеров в узле дерева классификации. Если $p(x_i)$ является вероятностью, что случай относится к классу x_i, то $p(1 - x_i)$ — это вероятность, что случай не относится к классу x_i. Если мы выбираем случай наугад, то вероятность, что он принадлежит классу x_i, равна $p(x_i)$. Если мы задаем случай в классе как попало, то вероятность, что мы неверно определим класс, равна $1 - p(x_i)$. Следовательно, вероятность, что мы нечаянно возьмем случай из класса x_i и неверно определим его класс, если мы просто бездумно классифицируем его, равна $p(x_i)(1 - p(x_i))$. Индекс Джини представляет собой сумму этих вероятностей для всех n элементов:
$$p(x_1)(1 - p(x_1)) + p(x_2)(1 - p(x_2)) + \cdots + p(x_n)(1 - p(x_n))$$
Индекс Джини равен нулю, когда все случаи в узле относятся к одной и той же категории. Теперь вместо прироста информации для разделения наших примеров мы можем использовать уменьшение индекса Джини, чтобы принимать решения. Переделайте алгоритм ID3 так, чтобы он применял индекс Джини.

15 Немного построчим

Когда вы просматриваете в браузере страницу с большим объемом текста, вы наверняка пользуетесь поиском по странице (или каким-то эквивалентом). Вы печатаете фрагмент интересующего вас текста, и браузер выделяет на странице места, где встречается данный фрагмент. Такая же функция есть во всех видах документов, например в PDF. Она является основой основ для текстовых процессоров и редакторов; исправление термина в документе состоит из нахождения и замены, и программа способна найти все места с неверным употреблением и заменить некорректную формулировку на правильную.

В основе всех этих случаев лежит одна и та же операция: сравнение строк, также известное как поиск строки. Все тексты в компьютерах представлены в виде строк, которые обычно оказываются массивами символов, зашифрованными определенным способом в числа. Как всегда, позиция в массивах представлена рядом символов, начинающихся с нуля. Абзац, страница или книга могут быть представлены в виде строк символов различной длины. По сути поиск чего-либо в строке — это поиск определенной строки, которая встречается внутри другой строки: представьте, как вы ищете в абзаце нужное вам слово. Если обе строки одинаковой длины, то мы на самом деле пытаемся не отыскать что-то, а понять, одинаково ли содержимое этих строк. Это тоже, в некотором роде, сравнение строк, и оно представляет собой более простую задачу по сравнению с общей задачей поиска одинаковых фрагментов в длинной строке, так как здесь нам всего лишь нужно посмотреть, одинаковы ли строки при поочередном сравнении каждого символа.

Сравнение строк применяется далеко не только в поиске текстовых отрывков. Тот же принцип используется, когда мы хотим найти что-то внутри чего-то и когда искомый элемент и область

поиска представляют собой последовательность символов из одного и того же алфавита языка. Алфавитом может быть алфавит человеческого языка или же что-то совсем иное.

В биологии генетический код представляет собой набор правил, согласно которым ДНК и РНК зашифровывают белки. ДНК состоит из последовательностей оснований. Оснований в ДНК всего четыре: аденин (A), гуанин (G), цитозин (C) и тимин (T). Триплет оснований, называемый кодоном, шифрует одну определенную аминокислоту. Последовательность кодонов шифрует определенный белок; такая последовательность кодонов является геном. ДНК находится в хромосомах; белки, соответствующие генам, образуются в других частях клетки. Код для каждого отдельного белка содержится в РНК, в определенной информационной РНК (иРНК), которая тоже использует четыре основания: A, G, C и, вместо тимина, урацил (U).

Таблица 15.1. Генетический код ДНК

①	T		C		A		G		③
				②					
T	TTT	фенилаланин	TCT	серин	TAT	тирозин	TGT	цистин	T
	TTC		TCC		TAC		TGC		C
	TTA	лейцин	TCA		TAA	стоп	TGA	стоп	A
	TTG		TCG		TAG	стоп	TGG	триптофан	G
C	CTT		CCT	пролин	CAT	гистидин	CGT	аргинин	T
	CTC		CCC		CAC		CGT		C
	CTA		CCA		CAA	глютамин	CGA		A
	CTG		CCG		CAG		CGG		G
A	ATT	изолейцин	ACT	треонин	AAT	аспарагин	AGT	серин	T
	ATC		ACC		AAC		AGC		C
	ATA		ACA		AAA	лизин	AGA	аргинин	A
	ATG	метионин/старт	AGC		AAG		AGG		G
G	GTT	валин	GCT	аланин	GAT	аспарагиновая кислота	GGT	глицин	T
	GTC		GCC		GAC		GGC		C
	GTA		GCA		GAA	глютаминовая кислота	GGA		A
	GTG		GCG		GAG		GGG		G

Открытие генетического кода стало огромным прорывом в молекулярной биологии. Существует 20 аминокислот, из которых состоят все белки, и эти 20 аминокислот зашифрованы в ДНК и РНК последовательностями кодонов. Также есть особые кодоны, которые обозначают начало и конец каждого шифрования белка, как пробел, который разделяет в тексте слова. Таким образом, генетический код представляет собой набор правил, написанных с помощью алфавита, состоящего из четырех символов. Все кодоны ДНК отображены в таблице 15.1. Обведенные в кружок числа наверху таблицы отмечают первую, вторую и третью

буквы каждого кодона. Пересечение первого столбца и столбца, отвечающего за вторую буквы, — это столбец со всем спектром вероятностей для третьей буквы; выбирая буквы из столбца с обведенной тройкой, мы получаем особую аминокислоту.

Сравнение строк в молекулярной биологии появляется на сцене, когда нам надо найти определенную последовательность ДНК или РНК. Соответственно, мы ищем нить ДНК или РНК для определенной последовательности кодонов. Последовательности могут быть длинными и содержать десятки тысяч оснований; содержимое всего человеческого генома, представляющего собой полный набор последовательностей аминокислот, оценивается примерно в 3,2 миллиарда оснований. В такой перспективе сравнение строк — задача нешуточная.

Наблюдение с использованием электронных средств — еще одна область с большими объемами данных для сравнения строк. Обычно разного рода организации стараются найти определенный шаблон, или сигнал, в куче перехваченных данных. Таким сигналом может быть слово, фраза или преступное действие. Организация-перехватчик просматривает данные, пытаясь отыскать соответствующие шаблоны. К сожалению, такие организации любят создавать широкие сети, собирая все виды данных подряд, хотя большинство данных совершенно бесполезны; поэтому, чтобы справиться со работой, нужен механизм быстрого сравнения строк.

В компьютерной криминалистике, где все свидетельства хранятся в компьютерах и на запоминающих носителях, сравнение строк используется для распознавания обрывков информации, получаемых в ходе наблюдения за поведением определенного пользователя. Например, власти могут искать доказательство, что конкретный пользователь посетил сайт с тем или иным адресом или что были применены определенные криптографические ключи.

Еще одна область применения — обнаружение вторжений, когда мы пытаемся определить, есть ли в компьютерной системе вредоносные программные средства. Если нам известна последовательность байтов, которая может распознать такие средства, то в обнаружении вторжений можно применять методы сравнения

строк для нахождения угроз, скрытых во внутренней или внешней памяти компьютера. Или еще лучше, мы можем использовать сравнение строк для предотвращения вторжений путем слежения за трафиком сети, то есть за потоком байтов, проходящим через границы системы. Опять-таки, если мы знаем, какую последовательность байтов нужно искать, мы можем перехватить вредоносные программы до того, как они доберутся до нашей системы. Нам нужны быстрые алгоритмы сравнения строк, которые бы не тормозили трафик.

Распознавание докучливых текстов — главная составляющая обнаружения спама; спам состоит из шаблонных текстов, поэтому почтовый фильтр может применять сравнение строк, чтобы отделить нужные письма от макулатуры. Повторяющиеся куски текста помогают выявить случаи плагиаторства; жуликов можно определить путем проверки фрагментов сочинений, домашних заданий или программ. С другой стороны, плагиаторы могут предварительно обработать скопированный текст. К счастью, существуют алгоритмы, которые принимают в расчет такое поведение, поэтому их сложно обхитрить.

Извлечение текста с веб-сайтов, так называемый анализ экранных данных, работает со сравнение строк. В сети существует множество информации, находящейся по ту сторону HTML-страниц. Такой текст полуструктурирован, в том плане, что разделен определенными HTML-тегами, такими как <*li*> и </*li*> для элементов списка, что позволяет нам извлечь из страницы текст, который отвечает определенному условию, например текст между определенными тегами.

У каждой сферы, где применяется сравнение строк, свои требования. К примеру, нас может интересовать точное соответствие, когда мы хотим найти точь-в-точь такую же строку, или приближенное соответствие, когда мы хотим найти разные ее вариации. Мы можем иметь дело как с большим алфавитом, так и с коротким (вспомните ДНК с четырьмя символами). Мы можем пожертвовать скоростью ради простых для понимания и реализации алгоритмов или искать самое быстрое решение. Мы начнем с простейшего подхода, когда скорость не стоит во

главе угла, а затем перейдем к более сложным и более действенным алгоритмам.

Сравнение строк предполагает поиск одной строки внутри другой. Чтобы не запутаться, что с чем и где мы сравниваем, назовем искомую строку — фрагментом, а строку, в которой мы будем искать фрагмент, — текстом. Хоть мы и определились с терминами, они все же не точны: текстом может быть любая строка, не только текст из людской книжки. Однако, если мы не будем об этом забывать, то ничего страшного в использовании такой терминологии нет. В конце концов, так часто делают.

15.1. Сравнение строк перебором

Самый простой, «лобовой» метод сравнения строк — обычный метод перебора, когда мы с самого начала буква за буквой проверяем соответствие. Его еще называют методом грубой силы, потому что никакой хитрости или премудрости в нем нет; он изложен в алгоритме 15.1.

В качестве вводных данных у нас две строки символов: искомый фрагмент p и текст, в котором мы ищем наш фрагмент, t. Мы поместим результаты, которые будут индексами из t, в котором мы ищем p, в очередь q так, чтобы иметь к ним доступ в том порядке, в котором мы их нашли. Если соответствий нет, q будет пустой. В строчке 1 мы создаем очередь, которую получим на выходе, затем в строчках 2 и 3 помещаем длину фрагмента в переменную m, а длину текста в ряд n. В строчках 4–9 проходим цикл, начиная с t и до $n - m$ позиции в ней; нынешний индекс в t задан с помощью i. Очевидно, никакого совпадения после него быть не может, потому что иначе наш фрагмент вывалится за пределы текста. Каждая итерация цикла передвигается на одну позицию в t. В каждой новой позиции мы готовимся и переходим к следующему циклу, в строчках 5–7. Внутренний цикл начинается в начале p и использует j в качестве индекса; он проверяет с помощью $j < m$, что p не исчерпался. Для каждого символа в p, заданного $p[j]$, мы проверяем одинаков ли он с j-ым символом из p, начиная с его нынешней, i-той, позиции. Если они одинаковы, увеличиваем j на

один. Таким образом, j показывает число символов, которые совпали в t, для каждой отдельной позиции, в которой мы пытаемся найти p. Обратите внимание, что проверки в строчке 6 должны идти именно в таком порядке; мы полагаем, что здесь действует упрощенное вычисление и нам не нужно идти с проверкой дальше конца p. Есть два способа выйти из внутреннего цикла, соответствующего двум проверкам в строчке 6. Если у нас $j = m$, то мы выходим из цикла с полным совпадением символов, и значит, мы нашли искомый фрагмент. После мы добавляем в очередь q индекс i позиции в t, где мы нашли совпадение. Если мы выходим из цикла, потому что наткнулись на несовпадение, это означает, что p не существует, начиная с i-ой позиции t; следовательно, нам надо испробовать следующую позицию t, поэтому мы совершаем новый проход внешнего цикла. В конце мы возвращаем q.

Алгоритм 15.1. Поиск строки методом перебора

`BruteForceStringSearch(`p, t`)` $\rightarrow q$

Вводные данные: p, фрагмент; t, текст.

Выводимые данные: q, очередь, содержащая индексы t, когда p найден; если p не найден, очередь пустует.

```
1  q ← CreateQueue()
2  m ← |p|
3  n ← |t|
4  for i ← 0 to n − m do
5      j ← 0
6      while j < m and p[j] = t[i + j] do
7          j ← j + 1
8      if j = m then
9          Enqueue(q, i)
10 return q
```

На рисунке 15.1 показано, что происходит, когда мы ищем BARD в BADBARBARD. Весь алгоритм можно представить в виде скольжения прозрачных санок с надписью BARD с вершины BADBARBARD. Каждое значение i отвечает за разные позиции санок; мы скользим на одну позицию вправо каждый раз, когда натыкаемся на несоответствие между BARD и буквами внизу.

На рисунке мы помечаем черно-белым символы, на которых выявилось несовпадение. Символы BARD, следующие за несовпадением, выделены серым, так как нам не нужно их проверять; данный рисунок иллюстрирует вторую проверку в строчке 6 нашего алгоритма. Первые два столбца на рисунке отображают значения *i* и *j* в конце каждого внутреннего цикла. Вы можете убедиться, что значение *j* показывает число символов, совпавших с каждым значением *i*.

i	*j*	B	A	D	B	A	R	B	A	R	D
0	2	B	A	R	D						
1	0		B	A	R	D					
2	0			B	A	R	D				
3	3				B	A	R	D			
4	0					B	A	R	D		
5	0						B	A	R	D	
6	4							B	A	R	D

Рисунок 15.1. Сравнение строк методом перебора

i	*j*	0	0	0	0	0	0	0	0	0	1
0	3	0	0	0	1						
1	3		0	0	0	1					
2	3			0	0	0	1				
3	3				0	0	0	1			
4	3					0	0	0	1		
5	3						0	0	0	1	
6	4							0	0	0	1

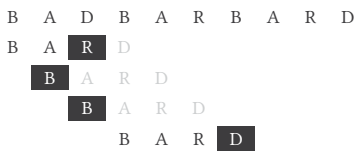
Рисунок 15.2. Худший случай для сравнения строк методом перебора

Метафора с санками дает нам способ обратиться к сложности поиска строк методом перебора. Внешний цикл будет выполняться $n - m$ раз. В худшем случае во внутреннем цикле каждый раз придется проверять все символы p и только в самом послед-

нем обнаруживать несовпадение. Например, такое случится, если p и t содержат только два символа, скажем, 0 и 1, и p совпадает в конце t, в то время как первые $m - 1$ символов p совпадают во всех предыдущих позициях; посмотрите на рисунок 15.2. Совершенно ясно, что подобная ситуация вряд ли встретится в человеческом тексте, однако она спокойно может произойти, когда мы ищем фрагмент в цифровых данных в целом. В таких патологических случаях нам понадобится m проходов внутреннего цикла для каждого прохода внешнего цикла. Так как у нас $n - m$ проходов внешнего цикла, конечный результат равен $m(n - m)$; соответственно, в худшем случае работа сравнения строк перебором будет $O(m(n - m))$. Обычно мы упрощаем до $O(mn)$, потому что n, как правило, намного длиннее, чем m, так, что $n - m \approx n$.

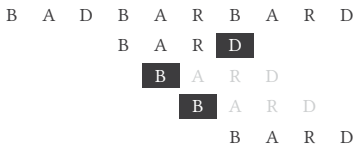
15.2. Алгоритм Кнута — Морриса — Пратта

Если вы еще раз взгляните на рисунок 15.1, то заметите, что мы потратили наше время на заведомо неудачные сравнения. Например, посмотрите, как мы начинаем:



Наша первая попытка провалилась, когда мы сравнили R и D. Затем мы пробуем сравнить B и A — не совпадают, затем беремся за B и D — и снова неудача. Но мы ведь уже знаем, что второй и третий символы нашего текста — это A и D соответственно. Как мы узнали? Мы еще в первый раз дошли до третьего символа нашего фрагмента, поэтому нам известно, что первые три символа текста — BAD. В то же время первые три символа нашего фрагмента — BAR, так что мы не можем проехаться BARом по BAD и получить совпадение; мы можем сразу перескочить к следующему символу, что равноценно смещению нашего фрагмента на три позиции вправо, и начать сравнение с оставшимся текстом:

Однако тут происходит то же самое.

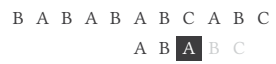


Нам известно, что в нынешней позиции текст читается BARB, потому что мы только что прочли его. Мы никак не сможем проскользить BARDом по BARB в одном или двух местах и получить полное совпадение; поэтому мы можем проехать по BARB на три позиции вправо и начать отсюда:

Давайте возьмем другой пример, с фрагментом ABABC и текстом BABABAABABC.

Мы тут же натыкаемся на несовпадение, поэтому перемещаем фрагмент на одну позицию вправо и пробуем заново:

В этот раз мы прошли минули четыре символа и столкнулись с несовпадением на пятом. Часть текста, которую мы пытались сопоставить, — ABABA, а фрагмент — ABABC. Возможно, нам захочется передвинуть наш фрагмент на четыре позиции, так как у нас в нем уже совпало четыре символа:



Однако это будет ошибкой, потому что в таком случае мы пропустим совпадение, которое мы получаем путем перемещения фрагмента на две позиции:

```
В А В А В А В С А В С
    А В А В С
```

Так что, кажется, нам стоит найти более вдумчивые подход, нежели простое перетаскивание фрагмента на возможное количество позиций вправо. Нам надо быть осторожней: слишком большой сдвиг — и пропустим совпадение. Каков же здесь основной принцип?

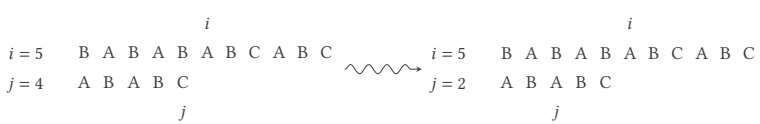
Мы использовали тот же метод, что и в алгоритме Кнута — Морриса — Пратта (названного в честь его изобретателей), он работает следующим образом. Мы проходимся по тексту буква за буквой. Скажем, в позиции i нашего текста у нас совпало j символов фрагмента. Мы увеличиваем i на один, до $i + 1$. Затем проверяем совпадает ли (j +1) символ фрагмента с (i + 1) символом текста. Если совпадает, мы увеличиваем i и j и продолжаем. Если же нет, пытаемся выяснить: при том, что у нас j совпавших символов в позиции i, но не совпадают $j + 1$ символов в позиции $i + 1$, сколько символов на самом деле мы можем сравнить в позиции $i + 1$? Мы обновляем j надлежащим образом и продолжаем.

На рисунке 15.3 изображен алгоритм Кнута — Морриса — Пратта (КМП) в действии на нашем примере. Вместо демонстрации фрагмента, скользящего по тексту, мы прибегли к двум указателям i и j, которые показывают, сколько символов совпало в тексте и фрагменте соответственно.

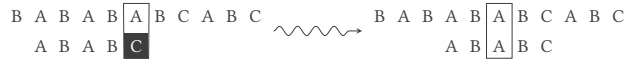
Мы начинаем с $i = 0$ и $j = 0$. Тут же несовпадение, но у нас еще не совпало ни одного символа во фрагменте, поэтому мы увеличиваем i и пробуем снова. Теперь символы совпали, мы увеличиваем i и j. Дальше идут совпадения, и мы продолжаем, повышая i и j до тех пор, пока не получаем $i = 5$ и $j = 4$. Затем у нас несовпадение и имеется число совпавших символов во фрагменте. Нам надо знать, откуда нам теперь начинать сравнение фрагмента; то есть какое значение мы должны присвоить j. Оказывается — мы позже увидим, каким образом, — что верное значение для j — это 2, так что мы делаем $j = 2$ и начинаем снова, увеличивая при каждом совпадении i и j. В конце у нас совпали все j, поэтому позиция совпадения равна $i – j + 1$ (мы добавляем единицу, потому строки отсчитываются от нуля).

Рисунок 15.3. Отслеживание алгоритма Кнута — Морриса — Пратта

Если хотите визуализировать работу алгоритма в виде смещений, а не как изменения i и j, то сдвиг на две позиции, который мы демонстрировали прежде, случается, когда мы увеличиваем i и нам нужно обновить j с меньшим значением. Именно поэтому фрагмент перемещается в право на $s = j_c - j$ позиций, где j_c — это нынешнее значение j. Вы можете удостовериться, что такое перемещение происходит, когда $i = 5$, $j = 2$. Так что это:



то же самое, что это:

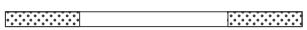


Алгоритм Кнута — Морриса — Пратта символ за символом сравнивает фрагмент с текстом. Когда такое перемещение встре-

чает несовпадение, оно пытается сохранить как можно больше уже совпавших символов вместо того, чтобы все обнулить и начать с самого начала фрагмента.

Для завершения алгоритма нам нужно точно знать, как мы решаем, какую часть фрагмента использовать еще раз, когда мы сталкиваемся с несовпадением. Чтобы понять это, давайте немного поговорим от терминологии. Часть строки называется подстрокой. Часть строки в начале строки — префикс; часть строки в конце строки — суффикс. У строки может быть много префиксов: все A, AB, ABA, … — это префиксы строки ABAXYZABA. Тупиковыми ситуациями являются ситуации с пустой строкой, которая рассматривается как префикс каждой строки, и полная строка, которая рассматривается как свой собственный префикс. Подходящий префикс — это такой который не является полной строкой. Аналогично у строки может быть множество суффиксов: все A, BA, ABA, … — это суффиксы ABAXYZABA. И снова мы вспоминаем о пустых строках и строках, которые сами себе суффиксы. Подходящий суффикс — это такой который не является полной строкой. В других книгах вам могут также попасться определения, в которых оговаривается, что подходящие префиксы и суффиксы не являются пустыми строками. Нам же предстоит иметь дело с непустыми префиксами и суффиксами.

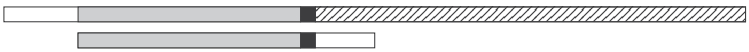
Граница — это подходящий префикс строки, который также является подходящим суффиксом той же строки; так что ABA представляет собой границу ABAXYZABA. Максимальная граница строки — строка с максимальной длинной: строки A и ABA — границы строки ABAYXABA. Максимальной границей является ABA. Мы определяем длину максимальной границы строки как ноль, если у строки вовсе нет границы. Как правило, строка с границей выглядит примерно так:



Мы обозначаем префикс и суффикс границы крапчатым фрагментом. Обратите внимание, что префикс и суффикс границы могут накладываться друг на друга, как в строке ABABABA

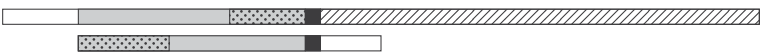
с границей ABABA, но сейчас нас данный случай не волнует и никак не касается наших рассуждений.

Теперь, имея под рукой данную терминологию, мы сумеем ответить на вопрос, какую же часть фрагмента можно повторно использовать при несовпадении. Представим, что у нас произошло следующее несовпадение:

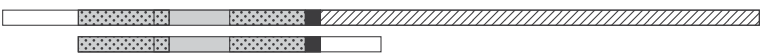


Сверху у нас текст, а снизу фрагмент. Штриховкой мы отмечаем часть текста, которую еще не прочли; серым закрашиваем одинаковые части, а черным — несовпадающие символы. В подобной ситуации давайте представим, что мы получим сравнение до несовпадающего символа включительно путем перемещения фрагмента вправо на определенное количество позиций:

В этот раз мы помечаем совпавшие части крапчатым фрагментом. Теперь представьте, что мы перемещаем фрагмент обратно в его исходную позицию. Получится так:



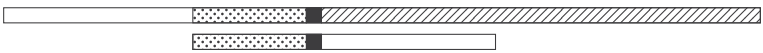
Так как мы знаем, что серые и крапчатые части совпадают, то единственный вариант для рисунка выше — это если крапчатая часть повторится в конце фрагмента; в противном случае получится несовпадение еще до того, как мы наткнулись на первое несовпадение. Поэтому у нас:



Получается, что у префикса совпавшего фрагмента должна быть граница! Наша проблема решена: когда мы натыкаемся на несовпадение, мы можем повторно использовать совпавшую часть фрагмента, если эта самая часть является границей. Вовсе не обязательно, что мы получим совпадение с границей, но нужно попробовать; более того, нужно попробовать получить

совпадение с максимальной границей, а затем с границами поменьше, в нисходящем порядке, чтобы не упустить возможное совпадение. Все из-за того, что, чем шире граница, тем меньше перемещение; а если наоборот, то, чем больше перемещение, тем уже границы. Сравните:

и



На втором примере граница шире и поэтому меньше сдвигается вправо, чем на первом примере. Это всего лишь следствие выражения $s = j_c - j$, которое выдает количество перемещений. Чем больше j, тем меньше перемещение вправо. Поэтому, как мы отмечали, начиная с максимальной границы и продолжая снижать границы, мы можем быть уверены, что не пропустим никакого совпадения. Скажем, нам надо найти фрагмент AABAAA в тексте AABAABAAAA. Мы натыкаемся на несовпадение через пять символов:

```
A A B A A B A A A A
A A B A A A
```

Совпавшим префиксом фрагмента является AABAA с максимальной границей длиною 2, AA, и еще одной границей, A, длиной 1. Нам нужно использовать границу с длиной 2, AA, приводящую к сдвигу на три позиции и совпадению:

```
A A B A A B A A A A
      A A B A A A
```

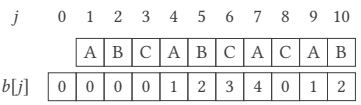
Если мы пропустим эту границу и вместо нее возьмем границу с длиной 1, A, то получим сдвиг на четыре позиции и пропустим совпадение, показанное выше:

```
A A B A A B A A A A
        A A B A A A
```

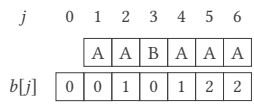
Так как во время поиска совпадений мы будем применять границы разных префиксов p, нам нужно их предварительно

вычислить таким образом, чтобы в случае чего, если у нас есть префикс из *p* длинной *j*, мы могли сразу же найти максимальную границу этого самого префикса из *p*. Если мы предварительно вычисляем границы, то удобней всего поместить их в массив *b* так, чтобы в *b*[*j*] находилась длина максимальной границы префикса из *p*, длина которого *j*.

На рисунке 15.4(a) изображен массив границ для фрагмента ABCABCACAB. Массив находится в самом низу рисунка, а вверху отображена длина каждого последующего префикса массива. Для префикса с нулевой длиной мы указываем, что его граница равна нулю. Если вы берете префикс длиной пять, то есть ABCAB, то можно видеть, что у него есть граница AB длиной в два. Точно так же, если вы берете префикс с длиной семь, то есть ABCABCA, можно видеть, что у него есть граница ABCA с длиной четыре.



(a) Массив границ для фрагмента ABCABCACAB



(b) Массив границ для фрагмента AABAAA

Рисунок 15.4. Массивы границ

Обратите внимание, что в массиве границ содержатся максимальные границы для каждого префикса. Так что на рисунке 15.4(b) вы можете видеть, что префикс с длиной пять, то есть AABAA, имеет две границы: AA и A. Максимальной границей является граница AA с длиной два, которая отображена в соответствующей клетке массива границ.

Если у нас есть функция `FindBorders`(*p*), которая создает массив *b* для фрагмента *p*, то мы имеем алгоритм 15.2, который приводит наши рассуждения в алгоритмическую форму.

Алгоритм без лишних хитростей реализует описанные нами идеи, и лучший способ понять его работу — прочесть его, параллельно просматривая пример, как на рисунке 15.3 или 15.5; обратите внимание, что мы не отслеживаем первые совпадения. В строчках 1–4 мы проводим подготовительную работу. Мы создаем очередь, которую вернем в конце, вычисляем длину фрагмен-

та и текста, а также массив границ. Затем в строчке 5 мы задаем j, которая будет подсчитывать символы, совпавшие в фрагменте. Еще одна переменная, i, будет подсчитывать, сколько символов в тексте мы уже прочли.

Алгоритм 15.2. Кнут — Моррис — Пратт

`KnuthMorrisPratt(`p, t`)` $\rightarrow q$

Вводные данные: p, фрагмент; t, текст.

Выводимые данные: q, очередь, содержащая индексы t, когда p найден; если p не найден, то очередь пустует.

```
1  q ← CreateQueue()
2  m ← |p|
3  n ← |t|
4  b ← FindBorders(p)
5  j ← 0
6  for i ← 0 to n do
7      while j > 0 and p[j] ≠ t[i] do
8          j ← b[j]
9      if p[j] = t[i] then
10         j ← j + 1
11     if j = m then
12         Enqueue(q, i − j + 1)
13         j ← b[j]
14 return q
```

В строчках 6–13 идет цикл, проходящий через каждый символ текста. Для каждого нового символа, если мы уже прочли и успешно сравнили часть фрагмента, а потом наткнулись на несовпадение, нам надо обновить j до длины максимальной границы совпавшего префикса фрагмента, что мы и делаем в строчках 7 и 8. Обратите внимание, что перед нами цикл, потому что мы можем найти несовпадение в границе и нам придется испробовать границу поменьше, и так далее. На самом деле, строчки 7–8 по нарастающей проверяют каждый последующий сдвиг, пока мы не наткнемся на совпадение. Это и происходит на рисунке 15.5, когда у нас $i = 5$ и $j = 5$, мы оставляем i неизменным и делаем $j = 2$, а затем $j = 1$. Так случается, потому что для $j = 5$ совпавшая часть

фрагмента — это AACAA с длиной границы 2, так что мы устанавливаем $j = 2$, но снова получаем несовпадение; следовательно, мы пробуем границу префикса с длиной 2, AA, которая представляет собой просто A с длиной 1. Поэтому мы ставим $j = 1$.

Когда мы находим совпавший символ, нам лишь нужно увеличить значение j, что и происходит в строчках 9–10. Если все символы фрагмента совпадут (что проверяется в строчке 11), то у нас будет полное совпадение; с строчке 12 мы добавляем позицию совпадения в очередь q. Затем, прежде, чем перейти к чтению следующего символа текста, нам надо обновить j до наибольшей границы в строчке 13, которая отвечает за наименьшее смещение, которое мы можем сделать без потери возможного совпадения.

Чтобы завершить описание алгоритма Кнута — Морриса — Пратта, расскажем о функции `FindBorders`. Работа происходит следующим образом. Если мы уже выяснили, что для префикса длиной i у нас есть граница длиной j, как на рисунке 15.6(a), мы легко можем проверить, есть ли у префикса длиной $i + 1$, который заканчивается в позиции i, граница длиной $j + 1$, которая заканчивается в позиции j. Такое может случиться, только если символ в позиции j фрагмента, $(j + 1)$-ый символ фрагмента, точно такой же, как символ в позиции i фрагмента, $(i + 1)$-ый символ фрагмента; взгляните на рисунок 15.6(b). Если такого не происходит, то лучшее, что мы можем сделать, проверить тут же границу поменьше, скажем, $j' < j$ длины, и посмотреть, совпадает ли последний символ границы со $(i + 1)$-м символом фрагмента; взгляните на рисунок 15.6(c). Если совпадения снова нет, то мы сразу же пробуем границу еще меньше, и так далее. Если вдруг мы обнаруживаем, что границы поменьше нет, то ясно, что граница для префикса с длиной $i + 1$ равна нулю.

Такое положение дел приводит нас к методу нахождения границ. Мы работаем с увеличивающимися префиксами строк, чьи границы мы хотим найти. Для префикса длиной ноль и один, длина границы равна нулю. Если у нас есть префикс с длиной i, мы делаем все, как описано выше: проверяем, можем ли мы расширить уже существующую границу на один символ; если нет, пробуем границу поуже, пока не вовсе не останется границ. Именно

это и проделывает алгоритм 15.3. Он берет в качестве вводных данных строку p и возвращает массив b так, что $b[i]$ является длиной границы префикса p с длиной i.

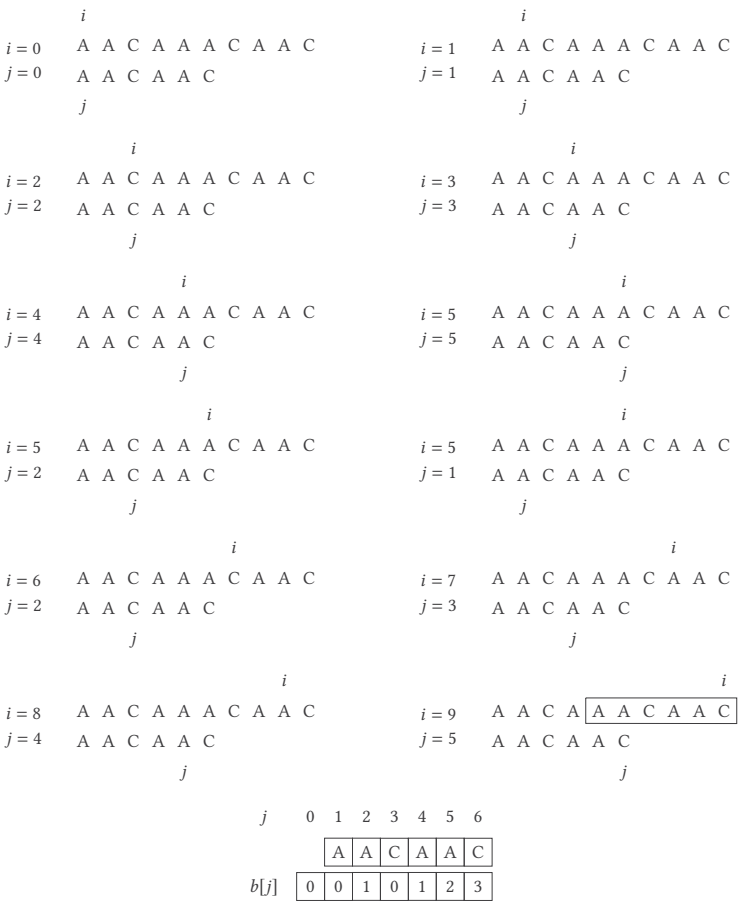
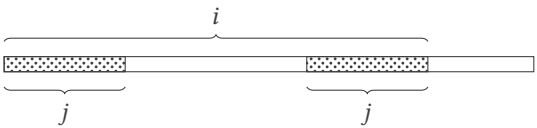


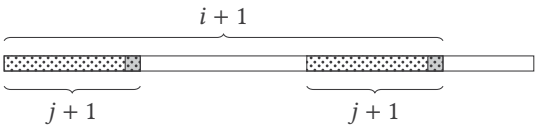
Рисунок 15.5. Еще один пример выполнения работы алгоритма Кнута — Морриса — Пратта; массив границ находится внизу

Алгоритм 15.3 начинает работу со вспомогательных операций: вычисление длины строки в строчке 1, создание итогового массива в строчке 2, установка длины нынешней границы, j, на ноль в строчке 3 и установка длины границ префикса p с длиной ноль и одни в строчках 4–5: как мы уже говорили, они равны нулю.

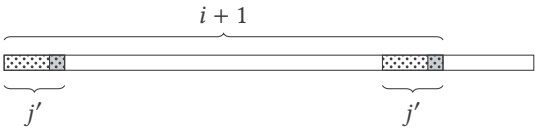
Затем в строчках 6–11 мы запускаем цикл для каждого символа *p*, начиная со *второго* символа и далее. Мы начинаем со второго символа, так как нам уже известно, что префикс длиной один имеет границу длиной ноль.



(a) Граница с длиной *j* для префикса с длиной *i*



(b) Расширение границы на один символ



(c) Проба более узкой границы

Рисунок 15.6. Нахождение границ строки

В строчках 7–8 мы пробуем отыскать границу поуже, которая могла бы подойти, если не подойдет нынешняя. В строчках 9–10 мы увеличиваем границу, какую бы границу мы ни нашли (даже нулевую) на один, если такое возможно. Затем в строчке 11 мы помещаем длину границы в подходящую позицию массива *b*. Так как первые два элемента *b* уже установлены, мы должны установить элемент в позицию *i* + 1. В конце мы возвращаем массив границ. Отслеживание работы алгоритма, находящего границы строки ACABABAB, показано на рисунке 15.7. Каждый ряд рисунка отображает значения *i*, *j* и содержимое *b* в начале и конце каждого внутреннего цикла.

Следует отметить, что алгоритм 15.3 очень похож на алгоритм 15.2. В самом деле, в алгоритме 15.2 мы сравниваем фрагмент *p* с текстом *t*; в алгоритме 15.3 мы сравниваем фрагмент *p* с самим собой. По сути мы прибегаем к одному и тому же процессу: сперва

находим границы нашего фрагмента, затем, найдя границы, выявляем сами соответствия. Возможно, алгоритм Кнута — Морриса — Пратта не так уж легко понять, но как только вы ухватите его суть, вы оцените его изящную простоту.

Алгоритм 15.3. Находим границы строки

```
FindBorders(p) → b

    Вводные данные: s, строка символов.
    Выводимые данные: b, массив длиной |p| + 1, содержащий длины
    границ p; b[i] содержит длину границы префикса p с длиной i.

1   m ← |p|
2   b ← CreateArray(m + 1)
3   j ← 0
4   b[0] ← j
5   b[1] ← j
6   for i ← 1 to m do
7       while j > 0 and p[j] ≠ p[i] do
8           j ← b[j]
9       if p[j] = p[i] then
10          j ← j + 1
11      b[i + 1] ← j
12  return b
```

Кнут — Моррис — Пратт не только прост, но и быстр. Так как алгоритмы 15.2 и 15.3 по сути одинаковы, нам нужно проанализировать только один из них. Внешний цикл алгоритма 15.2 выполняется n раз. При каждом его проходе у нас имеется несколько проходов внутреннего цикла строчек 7–8. Он повторяется, пока $j > 0$, и при каждом проходе j уменьшается. Более того, j может увеличиться только однажды, в строчке 10, во время прохождения внешнего цикла. Следовательно, во всех проходах внутреннего цикла j нельзя уменьшить более n раз, что означает, внутренний цикл не может повториться в общей сложности более n раз. Из этого следует, что вычислительная сложность алгоритма, не принимая в расчет `FindBorders`, составляет $O(2^n) = O(n)$. Применяя тот же логику, мы получаем, что вычислительная сложность `FindBorders` равна $O(m)$. Так что общее время выпол-

нения Кнута — Морриса — Пратта, включая предварительные этапы нахождения границ фрагмента, составляет $O(m + n)$. Плюс надо добавить небольшие затраты на пространство: нам нужно разместить массив b с границами. Впрочем, его длина $m + 1$, что, как правило, не является проблемой.

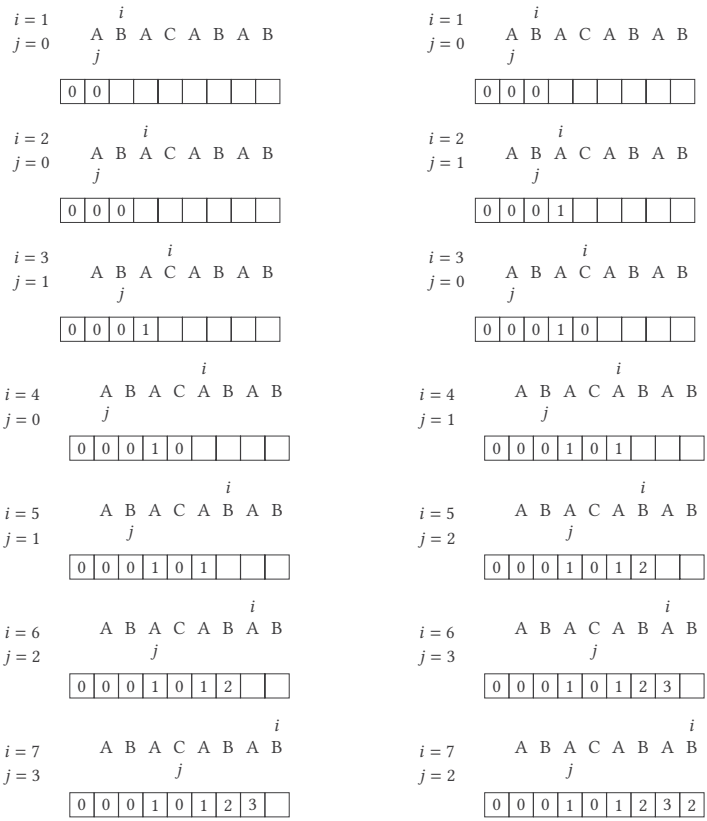


Рисунок 15.7. Ход работы по нахождению границ ABACABAB

15.3. Алгоритм Бойера — Мура — Хорспула

Прежде мы изучали наш текст слева направо. Если же мы изменим направление и станем рассматривать текст справа налево, то сможем работать с другим, простым, алгоритмом, хорошо зарекомендовавшим себя на практике. Алгоритм называется Бой-

ер — Мур — Хорспул, в честь его изобретателей. На рисунке 15.8 мы ищем фрагмент KETTLE в строке APESTLEINTHEKETTLE с помощью алгоритма Бойера — Мура — Хорспула. Мы помещаем KETTLE в начало текста. Затем, вместо проверки символов на соответствие слева направо, мы проверяем их справа налево.

Мы тут же сталкиваемся с несовпадением; последний символ KETTLE, E, не совпадает с последним символом текстового префикса APESTL, буквой L. Следовательно, мы можем начать перемещать наш фрагмент вправо. Так как самый правый символ, который мы прочли, — это символ L, наша следующая попытка сравнения должна передвинуть KETTLE таким образом, чтобы L во фрагменте совпала с L в APESTL. Иными словами, когда мы начинаем читать текст с позиции $i = 0$, то натыкаемся на несовпадение, которое мы пытаемся исправить путем передвижения фрагмента вправо на $r = 1$ позиций.

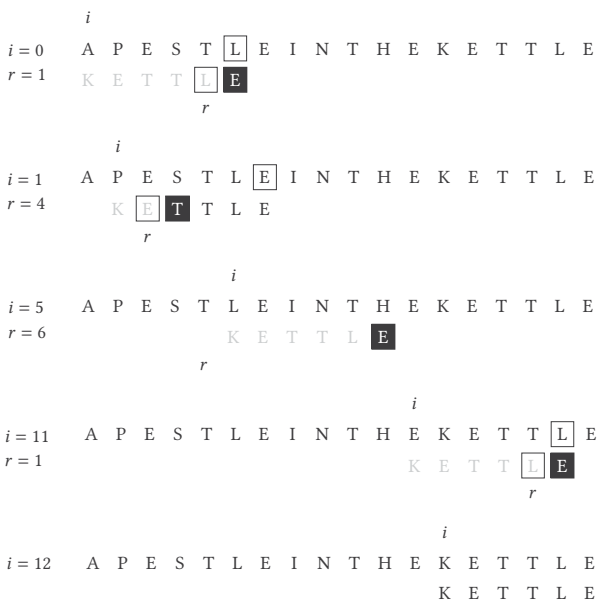


Рисунок 15.8. Алгоритм Бойера — Мура — Хорспула

Мы пробуем еще раз, следуя справа налево. В этот раз мы встречаем несовпадение на фрагментном символе T. Последний

символ, который мы прочли из текста, — символ E; следовательно, лучше всего будет переместить фрагмент вправо так, чтобы следующая встреча с E из фрагмента совпала с символом E, который является последним символом префикса APESTLE. Так что нам нужно сдвинуться вправо на $r = 4$ позиций.

Получаемая позиция тут же приводит нас к несовпадению. Более того, последнего символа текста, который мы прочли, символа H, вовсе нет во фрагменте. Так что мы можем до конца передвинуть фрагмент вправо от символа H и, значит, перемещаемся вправо на $r = 6$ позиций.

После этого мы оказываемся в позиции $i = 11$ и встречаем несовпадение L и E, похожее на то, что было в самом начале нашего поиска. Мы снова сдвигаемся на одну позицию вправо, к $i = 12$, где наконец находим полное совпадение фрагмента с текстом.

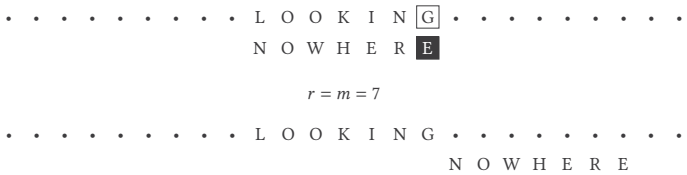
Как видно на рисунке 15.8, данный подход позволяет нам пропустить множество букв в тексте; однако, чтобы он сработал, нам надо придумать, как узнать, на сколько символов перемещать фрагмент вправо каждый раз. Как только это становится ясно, методом пользоваться очень просто.

Если нам встретилось несовпадение и неподходящего символа нет во фрагменте, мы передвигаем фрагмент на m позиций вправо, где m является длиной фрагмента, потому что нам нужно миновать символ, с которым точно не будет совпадений; взгляните на рисунок 15.9(a).

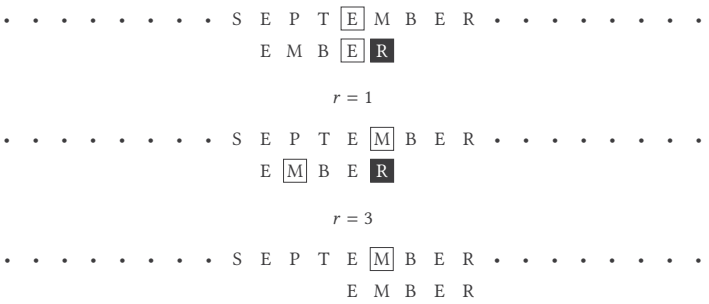
Если у нас есть несоответствие и несовпадающий символ в тексте присутствует во фрагменте в позиции $r \geq 1$, если считать от правого края фрагмента, то мы передвигаем фрагмент на r позиций вправо. Во фрагменте может случиться множество совпадений. Чтобы план сработал, t является индексом самого правого случая несовпадения в фрагменте, если считать от правого края; взгляните на рисунок 15.9(b).

Чтобы алгоритм заработал, нам нужен способ найти индекс самого правого случая каждого символа фрагмента. Нам поможет создание специальной таблицы. Таблица будет массивом с размером равным длине текста и строки. Например, если мы используем алфавит ASCII, в таблице будет 128 элементов. Исходя

из всего вышесказанного, если мы называем такую таблицу *rt*, содержимое *rt*[*i*] будет индексом самой правой позиции $r \geq 1$ для *i*-го символа ASCII в фрагменте, отсчитывая от конца фрагмента, если символ находится в фрагменте, или, в противном случае, длиной *m* фрагмента.



(a) Несовпадающего символа в фрагменте нет



(b) Несовпадающий символ в фрагменте есть

Рисунок 15.9. Правило для несовпадающего символа

Для фрагмента KETTLE в таблице *rt* все элементы будут равны шести, кроме элементов 69 (ASCII-код для E), 75 (ASCII-код для K), 76 (ASCII-код для L) и 84 (ASCII-код для T). Значения букв фрагмента и соответствующие им индексы, в десятичном и шестнадцатеричном видах, отображены слева на рисунке 15.10. На правой части рисунка показана та же информация для фрагмента EMBER; логика здесь та же, что и с KETTLE. Обратите внимание, что значение, соответствующее букве R, равно 5, все потому, что, если мы найдем несовпадение с ним, то нам нужно переместить фрагмент на всю длину, чтобы еще раз сравнить с текстом. Это соответствует определению правила несовпадающих символов, согласно которому $r \geq 1$. Если символ встречается только в конце фрагмента, его позиция справа равна нулю и условие $r \geq 1$ не вы-

полняется, поэтому мы обращаемся с ним, как с другими элементами таблицы, которых нет в фрагменте. Это дает нам гарантию, что несовпадение в том месте приведет к перемещению на всю длину фрагмента.

буква	K	E	T	T	L	E		E	M	B	E	R
ASCII (десятичный)	75	69	84	84	76	69		69	77	66	69	82
ASCII (шестнадцатеричный)	4B	45	54	54	4C	45		45	4D	42	45	52
встреча при отсчете справа	5	4	2	2	1	4		1	3	2	1	5

Рисунок 15.10. Самые правые случаи расположения букв во фрагменте.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
3	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	6	6	6	6	6	**4**	6	6	6	6	6	**5**	**1**	6	6	6
5	6	6	6	6	**2**	6	6	6	6	6	6	6	6	6	6	6
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
7	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

А. Таблица самых правых случаев для KETTLE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
2	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
3	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
4	5	5	**2**	5	5	**1**	5	5	5	5	5	5	5	**3**	5	5
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
7	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5

Б. Таблица самых правых случаев для EMBER

Рисунок 15.11. Таблицы самых правых случаев

На рисунке 15.11(а) изображена таблица самых правых случаев для KETTLE; мы всегда предполагаем, что алфавит состоит из 128 символов кодировки ASCII. На рисунке массив *rt* представлен в виде таблицы, где выделены позиции символов, которые

есть в KETTLE. В первом ряду и первом столбце содержатся шестнадцатеричные значения, по которым легко отыскать символы, работая с рисунком 15.10. Можете убедиться, что, к примеру, табличный элемент 0×4C, соответствующий символу L, имеет значение 1. На самом деле, *rt* является простым одномерным массивом с индексами от 0 до 127, однако на рисунке это не отображено из-за соображений компактности.

Алгоритм 15.4. Создаем таблицу самых правых случаев

`CreateRtOccurrencesTable`$(p, t, s) \to q$

Вводные данные: p, фрагмент; s, размер алфавита.

Выводимые данные: rt, массив размером s; для i-ой буквы алфавита $rt[i]$ будет индексом самой правой позиции $r \geq 1$, где внутри p встречается символ (отсчет идет с конца фрагмента), или, в противном случае, будет длиной фрагмента p.

```
1  rt ← CreateArray(s)
2  m ← |p|
3  for i ← 0 to s do
4      rt[i] ← m
5  for i ← 0 to m − 1 do
6      rt[Ord(p[i])] ← m − i − 1
7  return rt
```

В обеих частях рисунка 15.11 большинство табличных элементов равны длине фрагмента, именно поэтому алгоритм Бойера — Мура — Хорспула может быть эффективным: большинства символов алфавита не будет в таблице самых правых случаев, что позволит нам пропустить весь фрагмент всякий раз, когда мы сталкиваемся с одним из этих символов. В то же время, если в фрагменте содержится множество символов алфавита, то алгоритм может оказаться не так уж полезен, однако такое случается редко.

Нам нужен способ создать таблицу самых правых случаев, и тут нам помогает алгоритм 15.4. Хотя мы говорили в наших примерах о ASCII, данный алгоритм работает с любым алфавитом, если мы задаем размер алфавита в качестве аргумента. Функция

`Ord`(*c*) возвращает позицию символа *c* в алфавите, счет в котором идет от нуля. Сначала алгоритм создает массив *rt* (строчка 1) и вычисляет длину, *m*, фрагмента *p* (строчка 2). В строчках 3–4 он присваивает всему содержимому массива *rt* значение *m*. Затем в строчках 5–6 мы проходимся по каждому символу, *i*, фрагмента кроме последнего и вычисляем, как далеко он находится от правого края; полученный результат мы присваиваем содержимому *rt*. В конце мы возвращаем массив *rt*.

		E	M	B	E	R
i		0	1	2	3	
⋮	5	5	5	5	5	
B	5	5	5	**2**	2	
⋮	5	5	5	5	5	
E	5	**4**	4	4	**1**	
⋮	5	5	5	5	5	
M	5	5	**3**	3	3	
⋮	5	5	5	5	5	
R	5	5	5	5	5	
⋮	5	5	5	5	5	

Рисунок 15.12. Нахождение самых правых случаев в EMBER

Тут стоит обратить внимание на две вещи. Во-первых, мы не трогаем последний символ, потому что, как мы уже говорили, несовпадение с ним — если в фрагменте нет такого символа — в любом случае потребует передвижение всего фрагмента, так что верное значение для него — *m*. Во-вторых, в ходе выполнения алгоритма содержимое *rt*[*i*] может поменяться, если позже нам попадется такой же символ. На рисунке 15.12 показано, как алгоритм 15.4 изменяет значения таблицы *rt* для EMBER при выполнении цикла строчек 5–6. Таблица изображена вертикально; столбцы отвечают за состояние таблицы до цикла в строчках 5–6, а затем за каждое значение *i* в итерациях цикла. В самом левом столбце показаны буквы, которым присвоены места в таблице; в целях экономии пространства мы изображаем только важные нам буквы, а остальные помечаем вертикальным рядом точек. До начала цикла все значения таблицы устанавливаются на 5. При

первой итерации цикла значение для E меняется на 4; затем значение для M меняется на 3; после, значение для B меняется на 2; наконец снова меняется значение для E, становится 1.

Разработав алгоритм для самых правых случаев, можем взяться за алгоритм 15.5, алгоритм Бойера — Мура — Хорспула. В качестве вводных данных он берет искомый фрагмент *p*, текст *t*, в котором мы будем искать фрагмент, и размер, *s*, алфавита. Он возвращает очередь *q*, в которой находятся индексы из *t*, где найден *p*.

Алгоритм 15.5. Бойер — Мур — Хорспул

`BoyerMooreHorspool(`*p*, *t*, *s*`)` → *q*

Вводные данные: *p*, фрагмент; *t*, текст; *s*, размер алфавита.

Выводимые данные: *q*, очередь, содержащая индексы *t*, когда *p* найден; если *p* не найден, очередь пустует.

```
1  q ← CreateQueue()
2  m ← |p|
3  n ← |t|
4  rt ← CreateRtOccurrencesTable(p, s)
5  i ← 0
6  while i ≤ n − m do
7      j ← m − 1
8      while j ≥ 0 and t[i + j] = p[j] do
9          j ← j − 1
10     if j < 0 then
11         Enqueue(q, i)
12     c ← t[i + m − 1]
13     i ← i + rt[Ord(c)]
14 return q
```

Первые четыре строчки — подготовка к основной работе: создание итоговой очереди (строчка 1), получение длины фрагмента (строчка 2), получение длины текста (строчка 3) и получение таблицы самых правых случаев (строчка 4) с помощью алгоритма 15.4.

Настоящая работа идет в цикле строчек 6–13: здесь выполняется процесс, изображенный на рисунке 15.8. Установив в строчке 5 *i* на ноль, цикл будет повторяться, пока не сравнит *p* с *t*; для

этого требуется, чтобы $i \le n - m$, иначе p вывалится за правый край t. Переменная j возникает в конце, в строчке 7, и в цикле строчек 8–9 следует влево, к самому началу, пока не закончатся символы фрагмента, которые нужно проверить, и $p[j]$ совпадает с соответствующим символом в t, $t[i + j]$. Когда мы выходим из цикла, потому что у нас закончились символы фрагмента, у нас будет $j < 0$ и найденное совпадение, которое мы помещаем в очередь в строчках 10–11. Совпадение или нет, но нам нужно сдвигать фрагмент вправо. Количество символов, на которое нужно передвинуть фрагмент, зависит от содержимого таблицы *rt* для символа текста, который соответствует последнему символу фрагмента. Мы находим этот последний символ и сохраняем его в *c* в строчке 12. Затем мы получаем сдвиг из *rt*, просто взяв соответствующий элемент, и соответствующим образом обновляем *i* в строчке 13. Наконец мы возвращаем наши результаты, если таковые есть, в строчке 14.

(a) Худший вариант (b) Лучший вариант

Рисунок 15.13. Лучший и худший варианты для алгоритма Бойера — Мура —Хорспула

Алгоритм Бойера — Мура — Хорспула прост в реализации. Его работа, как правило, более чем хороша, для большинства p и t, хотя в вырожденных случаях может выполняться не быстрее сравнения методом перебора. На рисунке 15.13(a) показан пример

худшего варианта развития событий: у нас есть $n - m$ итераций внешнего цикла и $m - 1$ итераций внутреннего цикла для каждого из них кроме последнего, в котором у нас есть m внутренних итераций для оценки совпадения. В общей сложности мы получаем время выполнения $O(nm)$, равное времени, которое требуется сравнению методом перебора. На рисунке 15.13(b) показан лучший вариант развития событий, когда фрагмент снова и снова пропускает m символов, пока не находит совпадение в самом конце, что дает нам время выполнения $O(n/m)$. Может показаться, что такого не встретишь на практике — слишком уж хорошо и гладко все проходит, — однако на самом деле большинство поисков действительно такие, как на рисунке 15.13(b): искомая строка содержит лишь несколько символов алфавита, поэтому в среднем мы можем ожидать время выполнения работы равное $O(n/m)$. На создание таблицы rt у нас уходит $O(m)$ времени, что никак не сказывается на общей картине, потому что n, как правило, значительно длиннее, чем m. Большой помехой в работе Бойера — Мура — Хорспула может стать размер таблицы самых правых случаев. Алфавиту ASCII требуется таблица размером 128, что не сильно сказывается в применении, однако, если в нашем алфавите насчитывается не одна тысяча символов, то, возможно, придется задуматься о свободном месте.

Примечания

История сравнения строк представляет собой интересное взаимодействие теории и практики. Стивен Кук показал в 1971 году, что должен существовать алгоритм для сравнения строк, выполняющийся за $O(m + n)$ [40]. Еще раньше, в попытках уладить ошибки с реализацией текстового редактора Джеймс Х. Моррис придумал алгоритм со сложностью $O(m + n)$, хотя в то время это было еще не очевидно. Не зная ничего о его работе, Дональд Кнут принялся за разработку алгоритма согласно теоретической выкладке Кука. Кнут показал свою работу Вогану Р. Пратту, который подсказал несколько улучшений; Пратт, в свою очередь, показал алгоритм

Моррису, который понял, что перед ним по сути его же алгоритм. Все трое, вместе опубликовали алгоритм в 1977 году [116]. Еще одна версия алгоритма была предложена в 1997 году Рейнгольдом, Урбаном и Грисом [164].

Алгоритм Бойера — Мура — Хорспула был предложен Р. Найджелом Хорспулом в 1980 году [98]; мы использовали представление Лекрока [123]. Его сложность проанализировали Баеза-Ятес и Ренье [7]. На самом деле алгоритм представляет собой упрощенную версию более комплексного алгоритма, алгоритма Бойера — Мура, разработанного Робертом С. Бойером и Джеем Строзером Муром в 1977 году [26]. Первоначальный алгоритм Бойера — Мура имел в худшем случае сложность $O(m + n)$ при ненайденном в тексте фрагменте и $O(mn)$ при найденном фрагменте. Примерно в то же время вариант данного алгоритма быт независимо открыт Р. В. Госпером, основателем хакерского (не имеющего отношения ко взломщикам-кракерам) сообщества. Зви Галил показал в 1979 году, как можно усовершенствовать алгоритм, чтобы добиться сложности $O(m + n)$ даже при найденном в тексте фрагменте [73].

Тот небольшой объем материала, который мы предоставили здесь, позволяет по достоинству оценить богатство работы в сравнении строк. Если хотите знать больше, обратитесь к книгам Гасфилда [85] и Крошмора, Ханкарта и Лерока [45].

Упражнения

1. Все описанные в главе алгоритмы возвращают найденные совпадения. Если внести крохотное изменение, они станут возвращать только самое первое совпадение; напишите алгоритмы, которые будут выдавать только первое совпадение.
2. Алгоритм Бойера — Мура — Хорспула использует таблицу, в которой находятся самые правые случаи, так, что время поиска каждого символа минимально. Тем не менее результаты могут оказаться пустой тратой места, если

фрагмент представляет собой небольшой кусок алфавита. Попробуйте воспользоваться другой структурой данных, хеш-таблицей или множеством. Хотя время поиска должно по-прежнему оставаться неизменным, времени потребуется больше, чем на простую таблицу. Сравните работу реализаций исходного алгоритма Бойера — Мура — Хорспула и того, что сберегает место.

3. Причина, по которой мы используем очереди для полученных результатов в алгоритмах сравнения строк, заключается в том, что, если очереди реализованы так, что параллельно дают доступ к результатам, то пользователи алгоритма могут начать получать результаты сразу же, а не ждать обработку всего текста. Проверьте свои библиотеки и возможности программного языка и реализуйте алгоритмы таким образом, чтобы они давали параллельный, синхронизированный доступ к результатам.

16 Предоставим дело случаю

Случайность — вещь переменная, нестабильная, и чаще всего мы ее недолюбливаем: намного лучше иметь твердую почву под ногами и точно знать, куда ступаешь, а не рисковать с получением нежелательных последствий. Компьютеры — детерминированные машины, и мы не ожидаем в их работе сюрпризов. Ведь в самом деле, когда мы вдруг получаем несколько разных ответов на один и тот же вопрос, то начинаем подозревать, что где-то тут скрывается ошибка.

Точно так же мы не принимаем решения «от балды», как придется; мы вдумчиво взвешиваем все «за» и «против» и только потом приходим к определенным заключениям, пытаясь избежать предубеждений и оставаться объективными; наверняка замечание, что мы не оставили ошибке никаких шансов, прозвучит для наших ушей похвалой.

Однако случайность способна разрешить множество серьезных проблем. Помимо крохотной вероятности, что она поможет улучшить ваше материальное положение, подарив выигрыш в казино, она может оказаться полезной и в компьютерных процессах, решая важные задачи, для которых не существует более практичных, простых или подходящих методов.

Главное свойство удачи и шансов заключается в том, что их нельзя предсказать. У обычной монетки шанс выпадения орла или решки равен 50–50; если мы знаем, что центр тяжести в ней не смещен, мы понимаем, что шансы равны и не имеет смысла угадывать, какой стороной упадет монета.

Непредсказуемость лежит в области беспорядочности. Беспорядочность отвечает за отсутствие закономерности в ряде событий или данных. Что-либо беспорядочно или случайно, когда нет никакой системы, сколько ее ни ищи. Белый шум, то есть шум

помех на радио (невозможный на цифровом радио), беспорядочен. Последовательность чисел, выпадающих на игральной кости без дефектов, тоже случайна. Броуновское движение — это хаотичное движение частиц, взвешенных в текучей среде. Частицы сталкиваются с атомами или молекулами текучей среды (газа или жидкости), и эти столкновения заставляют частицы изменять маршрут; и столкновения, и маршрут — хаотичны. Если отследить частицы в броуновском движении, то получится изображение случайной траектории.

Отсутствие закономерности может стать хорошим подспорьем. Вспомните соцопросы. Так как в большинстве случаев невозможно опросить все население, опросчикам приходится работать с выборочной группой жителей. В идеале им хотелось бы получить представительную выборку, то есть выборку, обладающую всеми теми же характеристиками, что и все население: в таком случае результаты будут максимально достоверны. Население в данном случае не обязательно относится к человеческим существам, здесь это скорее некая совокупность, объединяющая любой набор элементов, которые мы хотим изучить.

Примером ошибки отбора может служить ошибка выжившего, когда мы отбираем только тех субъектов, которые присутствуют во время опроса или взятия пробы, в то время как для исследования также важно и то, что происходило ранее. Возьмем, к примеру, опрос фирм, переживших финансовый кризис: такая выборка ошибочка, так как в нее не включены фирмы, которым кризис пережить не удалось.

Случайная выборка, то есть образец, взятый из совокупности наугад, устраняет необъективность. Пока нет никаких предпочтений или закономерностей касательно кандидатов выборки, можно спокойно предполагать, что полученная выборка отражает общую совокупность.

Чтобы получить случайную выборку, нам нужно прибегнуть к процессу, который задействует хаотичность, потому что в противном случае мы получим предсказуемую выборку. Таким процессом, если говорить алгоритмическим языком, будет рандоми-

зированный* алгоритм: алгоритм, который в своих операциях применяет беспорядочность.

Тут нас поджидает значительный смысловой скачок. Обычно мы предполагаем, что алгоритмы полностью обусловлены и какие данные мы введем на входе, такие и получим на выходе. Если мы согласимся, что выводимые данные могут зависеть не только от вводных данных, но еще и от степени хаотичности, перед нами откроются новые возможности.

Существуют задачи, для решения которых у нас нет подходящего алгоритма; более того, есть такие задачи, про которые мы заведомо знаем, что подходящих алгоритмов для них не бывает. Под «подходящим» мы имеем в виду оптимальное соотношение вычислительных ресурсов, ресурсов места и ресурсов времени. Также существуют задачи, для которых у нас есть подходящие алгоритмы решения, но есть и рандомизированные алгоритмы, которые намного проще любого не рандомизированного, известного нам на данным момент.

Конечно же, у хаотичности есть своя цена. Алгоритм не будет совершенно предсказуемым. Непредсказуемость может проявить себя разными способами: порой алгоритму не удастся выдать верные результаты или же ему потребуется много времени на выполение. Залог к успешной работе с рандомизированным алгоритмом заключается в подсчете этих самых «порой». Нам нужно узнать, насколько редко алгоритм не сможет выдать нам верный ответ и каково ожидаемое и наихудшее время выполнения. К тому же следует взять в расчет степень точности; алгоритм может дать ответ, который будет корректным в пределах определенного интервала значений, и чем дольше работает алгоритм, тем меньше становится интервал и, следовательно, тем выше достоверность ответа.

Польза рандомизированных алгоритмов — одно из важнейших исследовательских достижений в информатике за последние десятки лет. О ней можно писать до бесконечности, однако у нас сейчас несколько другая цель. Мы рассмотрим лишь небольшой

* От англ. random — случайный, беспорядочный.

пример рандомизированных алгоритмов, касаясь различных областей их применения; хоть на краткий миг, но вы увидите, на что способен шанс и как он может вам помочь.

16.1. Случайные числа

Прежде чем мы возьмемся за рандомизированные алгоритмы, нам придется разрешить фундаментальную проблему: как нам вообще добиться нужной хаотичности. Обычно речь идет о случайном числе или ряде случайных чисел, которые мы скармливаем нашему алгоритму. Но где нам взять случайные числа? Где найти так называемый генератор случайных чисел, его компьютерное воплощение, которое снабдит нас случайными числами?

Одна из самых известных цитат в программировании прозвучала в 1951 году из уст одного из первопроходцев данной области, Джона фон Неймана, который заметил:

> «Всякий, кто питает слабость к арифметическим методам получения случайных чисел, грешен вне всяких сомнений».

Даже если во всех девяти кругах ада нам не сыскать случайных чисел, факт останется фактом: попробуй получить случайные числа закономерным путем — и ты обречен на провал. Детерминированные машины, выполняющие заданные им алгоритмы, никаким образом не смогут породить хаотичность: одно просто противоречит другому. Если вам дадут алгоритм, который генерирует случайные числа, и он станет выдавать вам результаты, вы с легкостью сможете предугадать, каким будет следующее число. Просто обратите внимание на нынешнее состояние алгоритма и проделайте его следующий этап самостоятельно — вы со стопроцентной вероятностью предскажете следующее число. Никакой случайности.

Настоящие генераторы случайных чисел основываются на физических процессах, которые мы полагаем случайными. Такие генераторы называются истинными генераторами случайных чисел (ИГСЧ). Существует несколько подобных генераторов. Мож-

но использовать счетчик Гейгера в источнике радиации. Можно обнаружить фотоны с помощью полупрозрачного зеркала. Согласно квантовым эффектам, фотоны с одинаковой вероятностью будут либо проходить сквозь зеркало, либо отражаться от него, так что результат случаен. Можно поймать атмосферные радиопомехи, то есть шум, порожденный атмосферными процессами, например ударами молнии. Существуют аппаратные генераторы случайных чисел, которые можно вделать в компьютеры и работать с по-настоящему хаотичными источниками; однако такие генераторы мало где найдешь, и, возможно, они не в состоянии выдать нужные нам объемы случайных чисел.

Если по какой-то причине у нас нет под рукой ИГСЧ или он не способен сгенерировать нужное нам количество чисел, нам придется обратиться к генератору псевдослучайных чисел (ГПСЧ). Он представляет собой явно греховную задумку: детерминированный алгоритм, который будет создавать числа, похожие на случайные, но по сути не являющиеся таковыми, то есть всегонавсего псевдослучайные числа. С другой стороны, не так просто определить, что значит «похожие на случайные». Основное условие заключается в том, чтобы ГПСЧ генерировал числа, следуя равномерному распределению. Это такое распределение, в котором каждое число равновозможно. Таким образом, равномерное распределение ряда чисел от 1 до 10 будет содержать число 1 одну десятую времени, число 2 одну десятую времени и так далее до числа 10 включительно.

Не забывая, что числа, генерируемые ГПСЧ, псевдослучайны, мы станем называть выводимые данные таких алгоритмов случайными числами, без уточнений, что они на самом деле псевдослучайны.

Равномерное распределение хорошо нам подходит, но не в достаточной мере. Возвращаясь к предыдущему примеру, мы можем увидеть, что последовательность чисел:

1, 2,… , 10, 1, 2,…, 10, 1, 2,…, 10,…

то есть повторяющаяся по порядку последовательность чисел от 1 до 10, является равномерным распределением, но она совсем не

похожа на случайную. ГПСЧ создаст равномерное распределение, которое будет выглядеть как случайное. Для гарантии, что так и будет, существуют статистические тесты, которые проверяют последовательность чисел на хаотичность. Такие статистические тесты при получении последовательности чисел отмечают, насколько сильно они отклонились от того, что можно считать последовательностью по-настоящему случайных чисел.

Алгоритм 16.1. Линейный конгруэнтный генератор случайных чисел

```
LinearCongruential(x) → r

    Вводные данные: x, число 0 ≤ x < m.
    Данные: модуль m, m > 0 ; множитель a,
    0 < a < m; особое дополнение c, 0 < c < m.
    Выводимые данные: r, число 0 ≤ r < m.)

1   r ← (a × x + c) mod m
2   return r
```

Простой генератор псевдослучайных чисел, которым пользуются уже долгое время, приведен в алгоритме 16.1. Алгоритм представляет собой реализацию следующего вычислительного метода, называемого линейным конгруэнтным методом:

$$X_{n+1} = \left(aX_n + c\right) \bmod m, \quad n \geq 0.$$

Данный метод создает новое случайное число X_{n+1} на основе предыдущего, X_n. Он умножает X_n на особый множитель a, прибавляет к произведению особое дополнение c, а затем берет остаток деления по, опять же, специально выбранному модулю m. Чтобы метод заработал, нам нужно скормить ему начальное значение, X_0, которое называется исходником, *seed*.

Алгоритм работает в точности как описано. Он берет значение, X_n, названное в алгоритме как x, и создает новое значение, X_{n+1}, названное в алгоритме как r. Сперва мы вызываем его для начального значения s, для исходника; затем, скормив ему вводные данные, вызываем для выведения данных из предыдущего вызова, то есть у нас получается ряд вызовов:

$x \leftarrow$ `LinearCongruential`(s),

$x \leftarrow$ `LinearCongruential`(x),

$x \leftarrow$ `LinearCongruential`(x)

и так далее. При каждом вызове мы получаем новое значение, x, которое будем использовать как случайное число и как вводные данные для следующего вызова `LinearCongruential`.

В реализации линейного конгруэнтного метода, как и с другими ГПСЧ, мы не вводим x при каждом вызове; эти вызовы обращаются в вызовы более высокого уровня. Обычно для установки начального значения используется отдельный вызов, а затем создаются случайные значения путем вызова функции, которой не требуется переменных, потому что x является скрытой переменной. Сначала вы делаете вызов `Seed`(s), а потом для каждого случайного значения вызываете `Random`(). В итоге происходит все то, что мы описали.

Очевидно, что последовательность чисел, создаваемая алгоритмом 16.1, будет зависеть от начального значения. Один и тот же исходник всегда будет выдавать одну и ту же последовательность чисел. На самом деле это полезная вещь, потому что, когда мы пишем программы, применяя ГПСЧ, и хотим проверить, насколько корректно работает та или иная программа, хаотичность нам не нужна, а нужна возможность получить предсказуемые результаты.

Мы отметили, что значения a, m и c особые. Выбирать их нужно очень осторожно. ГПСЧ никогда не выдаст более m значений; когда он столкнется на уже созданное значение, он станет создавать те же значения, что и прежде. Фактически у метода есть интервал повторяющихся чисел. Мы должны выбирать a, m и c таким образом, чтобы интервал оказался насколько можно больше, в идеале m, а числа в интервале следовали равномерному распределению. При неудачном выборе мы получим короткие интервалы и, следовательно, предсказуемые числа, потому что интервал закончится очень быстро. Например, если мы назначим $s = 0$, $m = 10$, $a = 3$ и $c = 3$, то получим:

3, 2, 9, 0, 3, 2, 9, 0,…

Чтобы получить максимальный интервал, равный m, для каждого исходного значения, a, m и c должны соответствовать трем критериям. Во-первых, m и c должны быть взаимно простыми, то есть одно нельзя разделить на другое. Во-вторых, $a - 1$ должно делиться на все простые множители m. В-третьих, $a - 1$ должно делиться на 4, если m делится на 4. Нам не нужно углубляться в математику, чтобы выяснить, почему так; более того, нам не нужно ходить и выискивать числа, отвечающие данным требованиям. Как правило, мы довольствуемся параметрами, которые рекомендуют исследователи в данной области. Например, набор параметров $s = 2^{32}$, $a = 32310901$, c — нечетное число и m — степень двух.

Линейный конгруэнтный метод генерирует числа от 0 до $m - 1$, включительно, то есть в промежутке, обозначенном $[0, m - 1]$. Если нам нужны числа из другого промежутка, например $[0, k]$, мы можем умножить результат на $k/(m - 1)$. Еще можно добавить дополнительное x, чтобы получить промежуток в виде $[x, k + x]$. Особый случай с числами от 0 до 1, обозначенный $[0, 1]$, очевидно получается путем деления на $m - 1$. Часто там нужен интервал между 0 и 1, не включая 1, обозначенный как $[0, 1)$, его мы получаем путем деления на m.

Последние несколько лет ведутся качественные исследования, в которых пытаются выяснить, пригодны ли числа, сгенерированные линейным конгруэнтным методом, для работы (то есть появляются ли они в достаточно случайном порядке). В том же ключе идут рассуждения на счет других методов, претендующих на роль лучшего, так как они прошли большее количество тестов на хаотичность. Многообещающей альтернативой, преимущество которой заключается в быстрой работе, можно считать генератор xorshift64* (читается как «ксор шифт 64 звездочка»), который изображен в алгоритме 16.2. Генератор xorshift64 создает числа из 64 битов.

Алгоритм прост, если вы знакомы с операндами << и >>. Символ << является побитовым оператором сдвига влево; $x << a$ означает перемещение битов числа x на a позиций. Например, 1110 << 2 = 1000; пример сдвига всего байта на три бита влево можно

видеть на рисунке 16.1(a). Напротив, символ >> является побитовым оператором сдвига вправо; $x >> a$ означает перемещение битов числа x вправо на a позиций. Например, 1101 >> 2 = 0011; взгляните на рисунок 16.1(b).

Алгоритм 16.2. xorshift64*

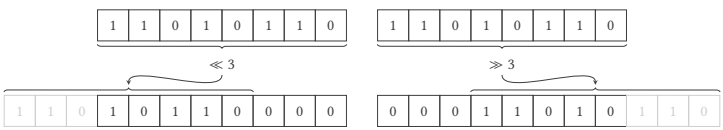
`XORShift64Star(`x`)` $\longrightarrow r$

Вводные данные: x, 64-битное целое число отличное от 0.

Выводимые данные: r, 64-битное число.

1 $x \leftarrow x \oplus (x \gg 12)$
2 $x \leftarrow x \oplus (x \ll 25)$
3 $x \leftarrow x \oplus (x \gg 27)$
4 $r \leftarrow x \times 2685821657736338717$
5 **return** r

Алгоритм xorshift64* используется точно так же, как линейный конгруэнтный метод. Он берет значение и создает новое значение, которое мы используем как случайное число и применяем в качестве вводных данных при следующем вызове алгоритма. Самый первый раз мы задаем ему начальное значение, которое не должно равняться нулю.



(a) Побитовый сдвиг влево (b) Побитовый сдвиг вправо

Рисунок 16.1. Побитовые операторы сдвига

Выводимые данные получаются за счет сдвига битов x вправо и применяем `xor` к смещенным битам с самим x в строчке 1. Мы повторяем ту же операцию, сдвигая биты влево и выполняя `xor` в строчке 2, а затем сдвигая биты вправо и выполняя `xor` в строчке 3. После этого мы умножаем x на волшебное с виду число и возвращаем его. Волшебное число в алгоритме является числом без явного значения; в нашем случае число особенное, так как именно оно гарантирует, что выводимые данные будут выглядеть хаотично.

Алгоритм xorshift64* работает быстро и качественно генерирует случайные значения; более того, у него интервал $2^{64} - 1$. Если вам нужно что-то получше, то вы можете поднять планку и обратиться к брату к алгоритма xorshift64* — алгоритму xorshift1024*, который можно увидеть в алгоритме 16.3. Генератор xorshift1024* тоже создает 64-битные числа, хотя по его названию так не скажешь. Он генерирует намного, много больше чисел.

Алгоритм 16.3. xorshift1024*

`XORShift1024Star(`S`)` $\rightarrow r$

Вводные данные: S, массив из 16 беззнаковых 64-битных целых чисел.

Данные: p, число, изначально установленное на 0.

Выводимые данные: r, 64-битное случайное число.

1 $s_0 \leftarrow S[p]$
2 $p \leftarrow (p + 1)\ \&\ 15$
3 $s_1 \leftarrow S[p]$
4 $s_1 \leftarrow s_1 \oplus (s_1 \ll 31)$
5 $s_1 \leftarrow s1 \oplus (s_1 \gg 11)$
6 $s_0 \leftarrow s_0 \oplus (s_0 \gg 30)$
7 $S[p] \leftarrow s_0 \oplus s1$
8 $r \leftarrow S[p] \times 1181783497276652981$
9 **return** r

В отличие от xorshift64* у алгоритма xorshift1024* огромный интервал, $2^{1024} - 1$, и он генерирует случайные числа, которые проходят даже большее количество статистических тестов на хаотичность. Алгоритм использует в качестве вводных данных массив S из 16 беззнаковых 64-битных целых чисел. В первый вызов алгоритма данный массив представляет собой начальное значение, и лучше всего, чтобы в нем было 16 чисел, сгенерированных алгоритмом xorshift64*. С каждым последующим вызовом содержимое S меняется, и S с новыми значениями используется в качестве вводных данных при следующем вызове алгоритма xorshift1024*. Именно массив S подарил алгоритму его называние, так как $16 \times 16 = 1024$.

В алгоритме также применяется индекс p, который применяется к целым числам массива S; при первом вызове p устанав-

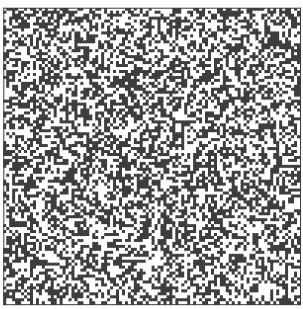
ливается на ноль и обновляется с каждым следующим вызовом. Алгоритм возвращает случайные 64-битные целые числа путем действий над частями массива *S*. В частности, в каждом вызове он работает с двумя элементами из *S*. Один обозначается с помощью *p* и помещается в s_0 в строчке 1. Затем, в строчке 2, *p* принимает следующее значение, а в строчке 3 мы получаем второй элемент *S* для данного вызова `XORShift1024*` и помещаем его в s_1. Строчка 2 является эффективным способом заставить *p* вести себя как индикатор от 0 до 15, а затем обратно до 0, и так далее. Именно это и проделывает побитовое «И» с 15. Если вы сомневаетесь, вспомните, что 15 — это 1111 в двоичной системе, поэтому побитовое «И» с 15 оставляет четыре последних бита как они есть, а остальные устанавливает на ноль; например, 1101 & 1111 = 1101. Если $p + 1 < 16$, «И» ничего не делает с суммой, потому что у $p + 1$ есть только последние четыре бита, которые остаются неизменными. Если $p + 1 = 16$, «И» превращает $p + 1$ в ноль, потому что 16 & 15 = 10000 & 1111 = 0.

Остальная часть алгоритма крутится вокруг сложных действий с битами из *S*. С помощью этих операторов мы в строчке 4 сдвигаем s_1 влево на 31 бит и проделываем операцию `xor` над полученными результатами и самим s_1. В строчках 5–6 мы проделываем то же действие с s_1, применяя сдвиг вправо, а затем еще одно действие с s_0, применяя сдвиг влево. «Ксорим» получившиеся s_0 и s_1, помещаем их обратно в $S[p]$ и возвращаем содержимое $S[p]$ с другим волшебным числом.

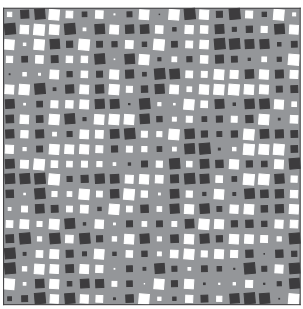
Алгоритм xorshift1024* — это сложный высокоточный механизм с продуманно подобранными константами, которые применяются как операнды в ряде установленных операций. Он появился в результате серьезной работы и многих усилий. Он создает хаотичность, но в ней все же нет места слепому случаю. И здесь самое время привести вторую цитату о хаотичности и компьютерах, принадлежащую Дональду Кнуту, одному из светил в области информатики: «Не стоит генерировать случайные числа случайным способом».

На рисунке 16.2(а) показана сетка с 10 000 случайными битами; нулевые биты черные, а единичные белые. Скорее всего, вам не

удастся увидеть в них какую-либо закономерность, что и к лучшему; если бы вы нашли закономерность, данные числа можно было бы смело назвать неслучайными (или вас — жертвой парейдолии, такого явления, когда вы видите образы и закономерности там, где их нет). Впрочем, даже без каких-либо закономерностей хаотичность может порождать интересные художественные образы, как на рисунке 16.2(b), в котором содержатся 400 случайных чисел, изображенные в черных или белых квадратов, приведенных к размерам и перевернутых согласно их значениям.



(a) 10 000 случайных битов



(b) Случайно сгенерированное изображение

Рисунок 16.2. Изображения хаотичности

Генераторы случайных чисел, которые мы рассматривали до сих пор, хорошо справляются со своей работой, однако они не подходят для генерации случайных чисел, которые можно применять в криптографических задачах. Существует множество областей криптографии, в которых нужны случайные числа: одноразовые блокноты, генерация ключей шифрования и одноразовые номера, произвольные числа, которые используются только единожды в особых протоколах.

Когда нам нужны случайные числа для криптографии, одних статистических тестов на хаотичность недостаточно, числа должны еще быть стойкими к особым атакам. В частности, не должно существовать алгоритма с полиномиальным временем, который мог бы рассчитать получаемые числа; к тому же не должно быть способа, учитывая состояние алгоритма, запустить обратную работу и узнать сгенерированные ранее числа.

В криптографии мы прибегаем к особым генераторам, которые называются криптографически стойкими генераторами псевдослучайных чисел (КСГПСЧ). Их построение намного сложнее, чем все рассмотренные нами методы, они подвергаются множеству исследований и тестов, проводимых криптографическим сообществом. КСГПСЧ можно считать рабочим, если он способен пережить все атаки и устоять.

Чтобы обрисовать общую концепцию работы КСГПСЧ, мы рассмотрим популярный алгоритм «Фортуна», названный так в честь римской богини удачи. Если описать схематически, то «Фортуна» работает, как показано на рисунке 16.3.

Чтобы гарантировать генерацию действительно непредсказуемых чисел, Фортуна применяет аккумулятор энтропии, задача которого — собирать непредсказуемые данные. Непредсказуемые данные имеют нечто общее с энтропией, отсюда и название. Энтропия должна получаться не из алгоритма, а из внешних источников. Такими источниками служат действия пользователя, например, нажатие клавиш или движения мышкой; события сети, такие как принятие или отправка данных; а также события диска, например, запись на диск. Информация о происходящих событиях записывается и сохраняется в накопителях энтропии, которые служат источниками настоящей беспорядочности для алгоритма.

Рисунок 16.3. Криптографически стойкий генератор псевдослучайных чисел Фортуна

Модуль генератора отвечает за создание случайных чисел, которые нам нужны. Он работает с помощью внутреннего индикатора C и ключа K. Берет ключ и шифрует значение индикатора блочным шифром, используя K в качестве ключа к блочному шифру. Полученный результат, $E_K(C)$, является выводимыми дан-

ными. Ключ можно считать эквивалентом начального значения, которое мы использовали в других методах. Чтобы убедиться, что выводимые данные нельзя предугадать, нам нужно время от времени менять ключ, в чем и состоит суть смены исходника, то есть начального значения. При смене исходника берутся вводные данные из случайных данных, которые были аккумулированы накопителями энтропии, и создается новый ключ с определенной периодичностью. В результате модуль генератора меняет состояние обоих, потому что индикатор *C* после шифрования изменяет новый блок и потому что получает новый ключ *K* между небольшими интервалами; так что, если даже кто-то угадает значение индикатора, нужно будет еще узнать значение *K*, а данное значение базируется на действительно непредсказуемых данных.

В Фортуне есть еще множество примечательных нюансов, которые делают ее максимально надежной и устойчивой к атакам. КСГПСЧ радикально отличается от старого простенького ГПСЧ, и мы лишь едва коснулись его общих черт. Главное, никогда не забывайте, что, если вам вдруг понадобились случайные числа для криптографии, нужно не полениться и воспользоваться Фортуной или еще каким-нибудь надежным КСГПСЧ.

16.2. Случайная выборка

Мы начали наше обсуждение с рандомизированной, или случайной, выборки, так что имеет смысл продолжить наши исследования и узнать, каким же образом осуществляется случайная выборка из совокупности. Если у нас есть совокупность размером n и нам нужно выбрать наугад m элементов из данной совокупности, то самое простое — это просмотреть все элементы и включить в нашу выборку элементы с вероятностью m/n. К сожалению, такой подход обречен на провал, потому что выбор m элементов совокупности будет осуществляться не каждый раз, в каждом случае, а на основе среднего значения. Но нам ведь нужна случайная выборка размером m для каждого случая.

Чтобы разрешить назревшую проблему, представьте, что мы уже просмотрели t элементов и выбрали k элементов для нашей

выборки. В нашей совокупности n элементов, и нам нужна случайная выборка размером m. Так как мы просмотрели t элементов, у нас все еще $n - t$ неизученных элементов, к тому же мы уже добавили $m - k$ элементов в нашу выборку. Сколько путей развития у данного процесса? Столько же, сколько способов, с помощью которых можно выбрать из оставшихся элементов совокупности пропущенный элемент для нашей выборки; если точнее, то столько же, сколько способов — обозначим их количество как w_1, — когда мы можем выбрать $m - k$ элементов из $n - t$ элементов. Это число возможных комбинаций, то есть беспорядочных выборок $m - k$ элементов из $n - t$, так что мы имеем $w_1 = \binom{n-t}{m-k}$. Если подобное описание вам незнакомо, загляните в главу 13.6. Рассуждая таким образом, если мы просмотрели $t + 1$ элементов и включили $k + 1$ элементов в выборку, то число способов развития процессе, назовем его w_2, будет $w_2 = \binom{n-t-1}{m-k-1}$. Подытоживая вышесказанное, вероятность того, что мы пройдем от t и k к $(t + 1)$ и $(k + 1)$ равна w_2/w_1:

$$\binom{n-t-1}{m-k-1} \Big/ \binom{n-t}{m-k} = \frac{m-k}{n-t}.$$

Правая часть уравнения получается путем подстановки в левую $\binom{a}{b} = \frac{(a)!}{(b)!(a-b)!}$.

Нам известно, что $(k + 1)$ элемент избирается с вероятностью $(m - k)/(n - t)$, если t из k элементов уже содержатся в выборке, что позволяет нам получить рандомизированную выборку, алгоритм 16.4, который реализует процесс отбора, соответствующий нашим условиям.

В качестве вводных данных алгоритм 16.4 берет массив P содержащий элементы совокупности и размер случайной выборки, m. Мы помещаем элементы случайной выборки в массив S размером m, который создаем в строчке 1. Переменная k в ходе выполнения алгоритма представляет число уже выбранных элементов; изначально, в строчке 2, ей присваивается нулевое значение. Точно так же переменная t представляет число просмотренных элементов, мы присваиваем ей нулевое значение в строчке 3. В строчке 4 задаем размер совокупности.

Цикл строчек 5–10 повторяется до тех пор, пока мы не выберем достаточное количество элементов. При каждой итерации цикла мы генерируем в строчке 6 равнораспределенное случайное число, u, находящееся в промежутке между 0 и 1, не включая 1. Применяем функцию `Random`(0, 1), которая отвечает за это и возвращает число в промежутке [0, 1). Опираясь на вышесказанное, нам нужно включить рассматриваемый в данный момент элемент, $P[t]$, с вероятностью $(m - k)/(n - t)$; такое случится, если $u < (m - k)/(n - t)$. Полевое испытание проходит согласно $u \times (n - t) < (m - k)$, потому что умножать проще, чем делить. Если прописанное в строчке 7 условие выполняется, тогда мы добавляем $P[t]$ в s (строчка 8) и увеличиваем число уже выбранных элементов (строчка 9). При любых условиях мы увеличиваем в строчке 10 число уже просмотренных элементов на один; завершаем алгоритм, возвращая S.

Чтобы доказать работоспособность алгоритма, нам нужно продемонстрировать, что он возвращает m случайно выбранных элементов — ни больше, ни меньше — с нужной вероятностью, ровно такой, какую мы хотим. Если после просмотра t элементов мы выбрали k элементов, вероятность, что мы выберем элемент $(k + 1)$, равна $(m - k)/(n - t)$. Можно показать, что общая вероятность того, что каждый элемент будет выбран, равна m/n. Под «общей вероятностью» мы подразумеваем вероятность, что элемент будет выбран независимо от того, выбраны ли предыдущие k из t. Для читателей, которым интересна математическая сторона дела, отметим, что переход от $(m - k)/(n - t)$ к m/n основывается на разнице между условными и безусловными вероятностями. Условная вероятность события — это вероятность, что событие произойдет, потому что произошло другое событие; в нашем случае речь идет о параметре $(m - k)/(n - t)$. Безусловная вероятность — это просто вероятность, что что-то случится, и все. Параметр m/n дан для нас; элемент выбирается вне зависимости от того, что там случилось ранее.

Теперь о количестве выбранных элементов. Представьте, мы столкнулись с ситуацией, когда нам осталось просмотреть $n - t$ элементов и еще нужно выбрать $n - t$ элементов так, чтобы $m - k = n - t$. Тогда $u \times (n - t) < m - k$ превратится в $u < 1$, и мы

совершенно точно выберем следующий элемент. Вспомните, что `Random`(0, 1) возвращает числа в промежутке [0, 1), поэтому неравенство будет выполняться. Такая же ситуация возникнет при $k + 1$ и $t + 1$, затем при $k + 2$ и $t + 3$ и так далее, до последнего элемента массива P.

Алгоритм 16.4. Рандомизированная выборка

`SelectionSampling`(P, m) $\rightarrow S$

Вводные данные: P, массив, содержащий элементы совокупности; m, количество элементов, которые нужно выбрать.

Выводимые данные: S, массив с m случайно выбранными из P элементами.

```
1  S ← CreateArray(m)
2  k ← 0
3  t ← 0
4  n ← |P|
5  while k < m do
6      u ← Random(0, 1)
7      if u × (n − t) < (m − k) then
8          S[k] ← P[t]
9          k ← k + 1
10     t ← t + 1
11 return S
```

Иными словами, проверка вероятности в строчке 7 будет работать таким образом, что все оставшиеся $n - t$ элементов в P окажутся выбраны. Следовательно, алгоритм не может завершить работу, выбрав менее m элементов. С другой стороны, если мы уже выбрали m элементов прежде, чем добраться до конца P, цикл завершится и не будет больше выбирать элементы, что на самом деле экономит время, но не является обязательным условием. Если бы цикл продолжал работу, $u \times (n - t) < m - k$ превратилось бы в $u \times (n - t) < 0$, что невозможно, поэтому алгоритм не стал бы больше выбирать элементы и просто дошел бы до конца P. Таким образом, мы совершенно точно выбираем не более m элементов. Так как мы можем выбрать минимум m элементов и максимум m элементов, у нас не остается другого варианта, кроме как выбрать ровно m элементов, и значит, алгоритм будет работать как надо.

Цикл в алгоритме будет выполняться, самое большое, n раз, если последний элемент P попадет в случайную выборку. Впрочем, довольно часто цикл выполняется меньшее число раз. Вероятность для каждого элемента, что его выберут, равна m/n; следовательно, вероятность выбора последнего элемента тоже равна m/n, а вероятность, что алгоритм завершит работу прежде, чем дойдет до последнего элемента, равна $1 - (m/n)$. Видно, что среднее число элементов, просмотренных до завершения алгоритма, составляет $(n + 1)m/(m + 1)$.

Для рандомизированной выборки требуется знать, размер совокупности n, однако мы не всегда можем обладать такой информацией. Совокупность может содержать записи в файле, и нам будет неизвестно, сколько их там. Мы можем сперва пересчитать записи в файле, а потом уже приступить к рандомизированной выборке, но тогда нам придется просматривать файл дважды: для подсчета и для выборки. Также совокупность может поступать к нам в виде потока данных, и мы не знаем, когда он прекратится, когда мы должны завершить выборку элементов, которые мы уже получили, не имея возможности заново пересмотреть их. Было бы полезно обзавестись алгоритмом, который работал бы с подобными случаями.

Такой алгоритм существует и называется накопительной выборкой. Его суть в том, что, если нам нужна выборка из m случайных элементов, мы наполняем резервуар из m элементов, как только находим их; то есть мы помещаем первые m элементов прямиком в накопительный резервуар. После при получении каждого нового элемента мы меняем содержимое резервуара так, чтобы каждый элемент резервуара имел m/t вероятность находиться в нем, где t — это число встреченных элементов. Когда мы просматриваем всю совокупность, каждый элемент в резервуаре окажется там с вероятностью m/n, если n — это размер совокупности. Накопительная выборка представляет собой онлайновый алгоритм, потому что ей не нужно скармливать все вводные данные.

Чтобы понять, как нам этого добиться, представьте, что у нас m элементов в резервуаре, куда каждый из них попал с вероят-

ностью m/t. Сначала, когда мы добавляем первые m элементов, вероятность составляет $m/t = m/m = 1$, так что условие с легкостью выполняется. Зная, что условие выполняется для t, мы хотим показать, что оно выполняется для $t + 1$.

Когда мы получаем элемент $(t + 1)$, то добавляем его в резервуар с вероятностью $m/(t + 1)$, заменяя один из уже находившихся там элементов. Извлекаемый элемент мы выбираем случайным образом, так что каждый элемент в резервуаре может быть вынут оттуда с вероятностью $1/m$.

Так как мы выбрали вероятность надлежащим образом, элемент, помещаемый в резервуар, переходит туда с требуемой вероятностью $m/(t + 1)$. Мы должны выяснить, каковы вероятности для элементов, оставшихся в резервуаре. Вероятность, что определенный элемент в резервуаре будет замещен, равна вероятности, что в резервуаре появится новый элемент и этот определенный элемент отправится оттуда прочь: $m/(t + 1) \times (1/m) = 1/(t + 1)$. И наоборот, вероятность, что элемент останется в резервуаре, равна $1 - 1/(t + 1) = t/(t + 1)$. Так как элемент уже находился в резервуаре с вероятностью m/t, вероятность, что уже находящийся в резервуаре элемент останется там, равна $(m/t) \times t/(t + 1) = m/(t + 1)$. Следовательно, и новые, и ранее добавленные элементы находятся в резервуаре с в надлежащей вероятностью после просмотра и обработки элемента $(t + 1)$.

В итоге, если мы на элементе $t \leq m$, то мы просто помещаем его в резервуар. Если мы на элементе $t > m$, то мы добавляем его в резервуар с вероятностью m/t и вынимает из резервуара случайный элемент, помещенный туда ранее. Реализация накопительной выборки показана в алгоритме 16.5.

Алгоритм просматривает элементы совокупности из *scr*, который может быть чем угодно, что имеет функцию `GetItem`, которая возвращает новый элемент из *scr* или `null`, если элементов больше нет. Также он берет в качестве параметра размер выборки; он возвращает массив из m случайно выбранных элементов из *scr*.

Алгоритм 16.5 начинает работу с создания резервуара S в строчке 1 и перемещения в строчках 2–3 первых m элементов *scr*

непосредственно в S. Затем в строчке 4 он назначает для t число уже просмотренных элементов. Цикл в строчках 5–9 выполняется до тех пор, пока больше не останется элементов. Возвратное значение `getItem`(*scr*) сохраняется в переменной a; если a равно `null`, тогда цикл прекращается; в противном случае мы выполняем строчки 6–9.

Первым делом внутри цикла мы увеличиваем значение t до $t + 1$ в строчке 5. Затем в строчке 7 — ключевой момент всего алгоритма — мы вызываем функцию `RandomInt`(1, t), которая возвращает случайное целое число u от 1 до t включительно, то есть в промежутке $[1, t]$. Условие $u \le m$ (строчка 8) точно такое же, как $u/t \le m/t$ поэтому, если условие выполняется, мы перемещаем элемент в резервуар. Чтобы взять наугад элемент, добавленный ранее в резервуар, мы снова используем u: оно ведь, в конце концов, случайное число между 1 и m, поэтому нам всего-то надо поместить a в позицию $u - 1$ массива S, отсчитываемого от нуля, (строчка 9). Когда в *scr* закончились элементы, мы выходим из цикла и возвращаем S в строчке 10.

Алгоритм 16.5. Накопительная выборка

```
ReservoirSampling(scr, m) → S
    Вводные данные: scr, источник элементов совокупности; m, количество элементов, которые нужно выбрать.
    Выводимые данные: S, массив с m случайно выбранными из scr элементами.
1   S ← CreateArray(m)
2   for i ← 0 to m do
3       S[i] ← GetItem(scr)
4   t ← m
5   while (a ← GetItem(scr)) ≠ NULL do
6       t ← t + 1
7       u ←RandomInt(1, t)
8       if u ≤ m then
9           S[u − 1] ← a
10  return S
```

Пример накопительной сортировки изображен на рисунке 16.4. Мы выбираем четыре элемента из шестнадцати. Резервуар

находится слева. В верхней части рисунка мы заполняем резервуар первыми четырьмя элементами. Затем, в зависимости от значения *u* при каждой итерации цикла, помещаем нынешний элемент, выделенный жирной линией, в резервуар. Обратите внимание, что во время выполнения алгоритма вполне может случиться так, что одна и та же позиция в резервуаре не принимает более одного нового значения; в нашем примере такое происходит с позициями один и два. В данном примере резервуар постоянно меняется, до самого конца, однако вполне может случиться и по-другому, потому что наполнение резервуара зависит от случайного значения *u*.

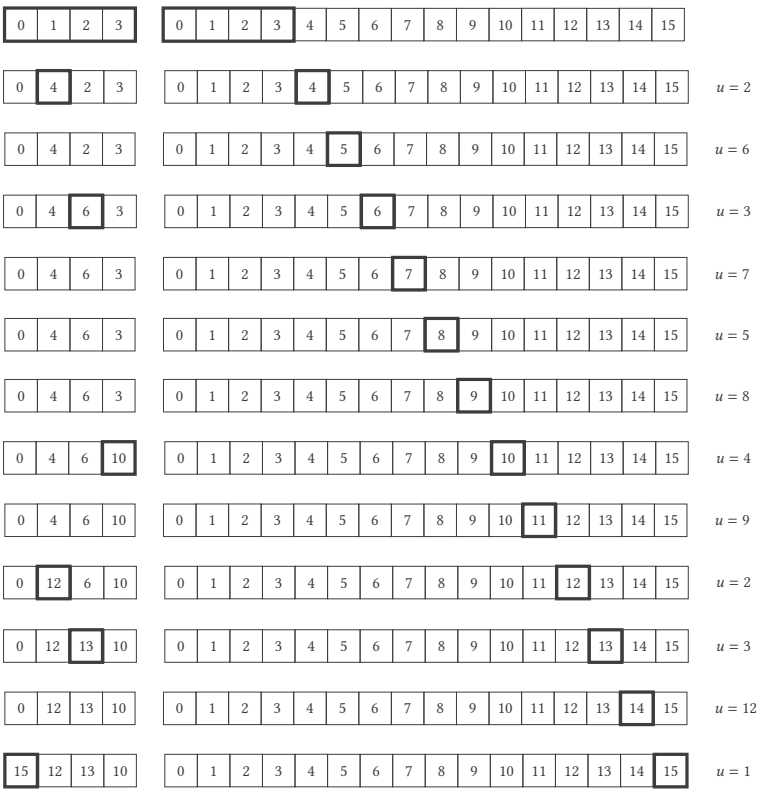


Рисунок 16.4. Накопительная выборка четырех элементов из шестнадцати

16.3. Борьба за власть

Можно ли как-то взвесить голос? Каков вес голоса на выборах? Такие значимые вопросы всегда витают вокруг голосований, и не всегда на них находятся однозначные ответы. Естественно, когда избиратель заполняет бюллетень, он ожидает, что его голос что-то значит, а иначе зачем вообще ходить на выборы или поддерживать политическую систему, которая дает голосовать, но не считается с голосами избирателей?

Если брать самую распространенную практику выборов, построенных по принципу один человек — один голос, то ответ не так просто найти. При миллионе бюллетеней шанс того, что один-единственный голос на что-то влияет, очень уж мал. Как правило, исход выборов определяется перевесом в более чем один голос. Таким образом мы натыкаемся на парадокс: если вероятность того, что ваш голос повлияет на результаты, очень мал, то голосовать бессмысленно, если только вас не тащат на выборы силой. Голосование занимает время, так что, если единичное мнение ничего не значит, избиратель попусту тратит это самое время. Но все же огромное количество людей ходят на выборы и участвуют в голосованиях. Такое положение дел называется парадоксом голосования, и проводится неимоверное количество исследований, которые пытаются разрешить его.

Каков вес голоса — данный вопрос актуален не только в обычных выборах, но и в других сферах. Существуют ситуации, в которых мы можем осмысленно изучить силу каждого голоса, чтобы получить любопытные и порой даже парадоксальные результаты.

В качестве примера приведем историю из жизни, в 1958 году был назначен предшественник нынешнего Евросоюза — Европейское экономическое сообщество (ЕЭС). Его основали шесть стран: Франция (FR), ФРГ (DE; Германия тогда еще была разделена), Италия (IT), Бельгия (BE), Нидерланды (NL) и Люксембург (LU). Одним из административных органов ЕЭС являлся совет министров. Министры стран-участниц созывались на собрания и голосовали в отношении тех или иных решений, касающихся политики ЕЭС.

Одним из первых возник вопрос, каким образом будут проходить голосования. Система «одна страна — один голос» тут же уравнивает голоса всех стран, однако при таком подходе не берется в расчет, что страны сильно различаются в плане ресурсов и численности населения. Разве честно, что голос маленького Люксембурга равен голосу куда большей по размерам Франции? Тогда в совете министров решили, что у голоса каждой страны будет определенный вес: у Франции, Германии и Италии вес голоса в четыре единицы, у Бельгии и Нидерландов — две единицы, а у Люксембурга — одна. Чтобы голосование прошло, нужно было собрать как минимум 12 единиц.

Сразу же становится очевидно, что у больших государств вес больше, чем у средних, и у всех них вес больше, чем у Люксембурга. Кажется, что все вполне приемлемо. Мы подозреваем, что Люксембург сам по себе не сможет управлять большинством и что крупным странам проще, чем более мелким, задавать свой курс. К сожалению, у системы имелся существенный недостаток.

Таблица 16.1. Голосование на первом заседании совета министров ЕЭС

FR (4)	DE (4)	IT (4)	NL (2)	BE (2)	LU (1)	итог
✓	✓	✓				12
✓	✓		✓	✓		12
✓		✓	✓	✓		12
	✓	✓	✓	✓		12

В таблице 16.1 мы привели список всех вероятностей получить 12 голосов. Мы доберемся до нужной границы, если все три крупных государства проголосуют одинаково или же если два крупных и два средних государства также сойдутся во взглядах. Конечно, если их поддержат другие страны, то решение будет принято всеми единодушно, но в этом нет необходимости. Проблема в том, что голос Люксембурга ни на что не влияет: в любом вопросе большинство голосов преспокойно набирается и без участия Люксембурга. Так что хоть Люксембургу и дали право

высказаться, его голос не имеет никакого веса. С тем же успехом можно было вовсе не давать ему голоса — в принятых решениях ничего бы не поменялось.

Здесь мы можем видеть острую проблему, когда избирателя по сути лишают голоса. В других ситуациях такие проблемы могут быть менее заметны: даже если избирателю дают право выбирать, насколько значим его голос по сравнению с голосами других избирателей? Возвращаясь к примеру с ЕЭС, зададимся вопросом, насколько Германия значимей Бельгии? Единицы голоса говорят нам, что Германия вдвое значимей, но разве это правда?

Чтобы разобраться в данном вопросе, нам нужно прибегнуть к помощи систематического подхода. У нас есть множество избирателей, $V = \{v_1, v_2, \ldots, v_n\}$, и множество весов, $W = \{w_1, w_2, \ldots, w_m\}$. У избирателя v_i вес w_i, заданный соотнесением f: $V \to W$. Для принятия решения нужно набрать квоту Q. В нашем примере с ЕЭС $Q = 12$. Набор V, W, f и Q называется игрой голосования.

Каждая группа избирателей называется коалицией. Коалиция, которая набирает квоту, называется победившей. Коалиция, которая не смогла набрать квоту, называется проигравшей. Если мы добавляем еще избирателей в коалицию победителей, мы получаем другую коалицию-победителя. В примере с ЕЭС коалицией победителей является {DE, FR, IT}; еще одна — {DE, FR, IT, BE}. Соответственно, коалиция победителей может расти все больше, однако это ни на что особо не повлияет. Но мы тут же увидим последствия, если сократим коалицию-победителя. Если мы начнем с коалиции победителей {DE, FR, IT, BE} и заберем оттуда BE, то получим коалицию победителей {DE, FR, IT}. Но если мы возьмем коалицию {DE, FR, IT} и заберем из нее одну страну, то эта коалиция перестанет быть победителем. Минимальная коалиция-победитель — это такая коалиция, что, если из нее убрать одного участника, она тут же станет проигравшей. Избиратель здесь является ключевой фигурой, опорным элементом, убери его — и получишь коалицию проигравших. В минимальной коалиции-победителе важен каждый ее участник, однако участники могут играть ведущую роль не только в минималь-

ной коалиции-победителе: представьте, если мы уберем DE из {DE, FR, IT, BE}.

Ключевым избирателем является избиратель, голос которого влияет на результат выборов. Избиратель оказывается важен, потому что как минимум одна коалиция победителей потеряет свою силу, если он ее покинет, так что результат выборов может зависеть от поведения конкретного участника. Избиратель, чье наличие в коалиции необязательно, — это тот избиратель, который никак не может повлиять на итоги выборов. Такого избирателя можно назвать липовым. В нашем примере с ЕЭС таким липовым участником оказался Люксембург. Если бы Люксембург был частью коалиции победителей, его уход из нее тут же превратил бы коалицию в проигравшую, однако мы видим, что действия Люксембурга никак бы не повлияли на результат выборов.

Теперь мы можем оценить вес голоса, который также называется индексом Банцафа, в честь его открывателя Джона Ф. Банцафа III. Начнем с определения банцафовой оценки избирателя v_i, которая является числом коалиций, где v_i играет ключевую роль. Мы определяем банцафову оценку избирателя v_i с помощью $\eta(v_i)$. Сама по себе оценка Банцафа не дает нам никакой информации, потому что мы не получаем показаний важности тех коалиций, в которых избиратель v_i играет ключевую роль. Избиратель может быть важен в целому ряду коалиций, но при этом еще большему числу коалиций он будет безразличен. Чтобы оценка Банцафа заработала, мы используем индекс Банцафа для общего числа голосов, то есть количество коалиций, в которых v_i важен, поделенное на общее число значимых коалиций для всех избирателей, что представляет собой соотношение значимых коалиций, приписанных определенному избирателю v_i. Если мы представим общее влияние голосов в виде пирога, то индекс Банцафа будет частью этого пирога, соотношением влияний, относящихся к каждому избирателю. Мы определяем индекс Банцафа с помощью $\beta(v_i)$, поэтому у нас есть:

$$\beta(v_i) = \frac{\eta(v_i)}{\eta(v_1) + \eta(v_2) + \cdots + \eta(v_n)}.$$

К примеру, возьмём четырёх избирателей A, B, C, D с соответствующими весами, равными 4, 2, 1, 3, и квотой $Q = 6$. Значимыми коалициями, в которых мы подчёркиваем ключевых избирателей, получаются {A, B}, {A, D}, {A, B, C}, {A, B, D}, {A, C, D}, {B, C, D}. Обратите внимание, что {A, B, C, D} побеждает, но она не значима. Подсчитав значимые коалиции для каждого избирателя, мы получаем $\eta(v_A) = 5$, $\eta(v_B) = 3$, $\eta(v_C) = 1$ и $\eta(v_D) = 3$. С ними мы получаем индексы Банцафа $\beta(v_A) = 5/12$, $\beta(v_B) = 3/12$, $\beta(v_C) = 1/12$, $\beta(v_D) = 3/12$. Результаты могут вас удивить. Хотя у всех избирателей был разный вес голоса, у избирателей *B* и *D* оказались одинаковые соотношения общего влияния голосов, приписанного им. У избирателя *D* вес голоса больше, чем у избирателя *B*, однако он не дает ему большего влияния. Избиратель *D* может радоваться иллюзорному первенству, в то время как *B* будет втайне посмеиваться над ним, зная, что на самом деле обладает куда большей властью, чем кажется на первый взгляд.

	x	y	z
$\varnothing$	0	0	0
$\{z\}$	0	0	1
$\{y\}$	0	1	0
$\{y, z\}$	0	1	1
$\{x\}$	1	0	0
$\{x, z\}$	1	0	1
$\{x, y\}$	1	1	0
$\{x, y, z\}$	1	1	1

Рисунок 16.5. Соответствие между подмножествами множества и двоичными числами

Мы рассчитали индекс Банцафа вручную; для этого нам понадобилось найти значимые коалиции и опорный элемент в каждой из них. Все легко просчитывается с помощью бумаги и ручки,

если число избирателей и возможных коалиций мало, однако совсем другое дело, когда речь идет о крупных масштабах.

Индекс Банцафа — это относительная мера, схожая с определением величины пропорции общего дохода группы на каждого отдельного участника; вместо дохода представьте общее влияние голосов. Еще нас интересует абсолютная мера влияния голоса так же, как и размер дохода отдельного человека, не связанный с распределением внутри группы.

Чтобы получить такую меру, мы начинаем с наблюдения, что коалиция представляет собой подмножество из *V*, общего множества избирателей, поэтому общее число всех возможных коалиций равно числу всех возможных подмножеств из *V*. Количество всех возможных подмножеств множества *S* с *n* элементами равно 2^n. Чтобы увидеть это, представьте двоичное число с *n* цифрами. Цифре *i* из нашего числа соответствует элемент *i* из множества *S*. Любое подмножество из *S*, следовательно, может быть представлено как число из *n* цифр с соответствующими битами, установленными на 1; посмотрите для примера на рисунок 16.5, включающий множество $S = \{x, y, z\}$. В результате число всех возможных подмножеств *S* — это число разных двоичных чисел с *m* цифрами, которое равно 2^n. Множество, в котором содержатся все подмножества множества *S*, называется степенным множеством *S*, и его обозначение — 2^S. Так что число всех возможных подмножеств *S* является числом элементов в его степенном множестве, 2^S, и оно, как мы уже видели, равно 2^n.

Если каждая коалиция равновозможна, то вероятность встречи определенной коалиции равна $1/2^n$. Если мы берем избирателя v_i из *V*, то получаем множество из $n - 1$ избирателей, поэтому общее число коалиций в множестве $V - v_i$ равняется 2^{n-1}. Опорная возможность для избирателя v_i, то есть возможность, что избиратель окажется опорным элементом, равна вероятности, при которой одна из коалиций $V - v_i$ становится значимой, если в нее добавить v_i. Вероятность равна числу значимых коалиций с v_i, поделенному на число всех коалиций без v_i, что является оценкой Банцафа $\eta(v_i)$, поделенной на 2^{n-1}, и она определяет банцафову меру влияния голосов, или просто меру Банцафа, обозначенную $\beta'(v_i)$:

$$\beta'(v_i) = \frac{\eta(v_i)}{2^{n-1}}.$$

Это и есть абсолютная мера, которую мы искали. Она дает нам вероятность, что если при уже подсчитанных голосах мы не знаем, как проголосует v_i, то отданный в ту или иную пользу голос v_i окажет влияние на результат всех выборов. Можно сказать и по-другому: абсолютная мера дает нам вероятность, что, если мы знаем, как проголосует v_i, то результат выборов изменится, если v_i переменит свое решение.

Давайте вернемся к нашей небольшой простенькой игре голосования. У нас четыре избирателя, $V = = \{A, B, C, D\}$, им приписаны веса, $W = \{4, 2, 1, 3\}$, и квота $Q = 6$. Мы хотим найти меру Банцафа избирателя A. У нас есть $2^3 = 8$ коалиций, в которых нет A, из них пять становятся значимыми, если в них добавить A; данная ситуация описана в таблице 16.2. Мера Банцафа для A в нашей игре голосования равна 5/8. Если мы проделаем ту же процедуру для трех других избирателей, то получим меры Банцафа $B = 3/8$, $C = 1/8$, $D = 3/8$.

Обратите внимание, что полученные нами числа ненормализированные (то есть в сумме они не дают единицу). Так происходит, потому что мера Банцафа не относительная мера, которая бы показывала, как общее влияние голосов распределено между избирателями. Мера Банцафа абсолютная, как мы и хотели, она показывает, каким влиянием обладает каждый избиратель. Следовательно, мы можем применить меру Банцафа и сравнить влияния избирателей в разных выборах, что было бы совершенно бессмысленно, вооружись мы индексом Банцафа. Если нужно, из меры Банцафа можно получить индекс Банцафа путем изменения масштаба $\beta'(v_i)$ таким образом, чтобы сумма всех $\beta(v_i)$ давала единицу. У нас есть:

$$\beta(v_i) = \beta'(v_i) \times \frac{2^{n-1}}{\eta(v_1) + \eta(v_2) + \cdots + \eta(v_n)}.$$

Поэтому мы можем рассматривать меру Банцафа как исходное понятие, а индекс Банцафа — как в производное.

Данная расчетная процедура, которую мы использовали в нашем примере для вычисления $\beta'(v_i) = \eta(v_i)/2^{n-1}$, хороша, пока количество избирателей невелико. Расчет значения $\beta'(v_i)$ для огром-

ного числа избирателей — дело посерьезней. Знаменатель $\beta'(v_i)$, 2^{n-1}, очень большой, но это не проблема, так как мы можем вычислить его самостоятельно. Будучи степенью двух, он является двоичным числом из n цифр, с единицей в качестве первой цифры и всеми последующими нулями. Заминка возникает с числителем, $\eta(v_i)$. Пока еще не нашелся эффективный способ рассчитать $\eta(v_i)$. Можно посмотреть в сторону более продуманных подходов, чем простой пересчет (например, когда нам известно, что коалиция достигает квоты, не имеет смысла расширять данную коалицию), однако они никак не меняют общую сложность задачи, которая не укладывается в алгоритм полиномиального времени. Нам нужен иной подход к работе с существенным количеством избирателей.

Таблица 16.2. Вычисление влияния голоса A в простой игре голосования с голосами $A = 4$, $B = 2$, $C = 1$, $D = 3$ и квотой $Q = 6$

коалиции без A	коалиции с A	голоса	побеждающая	значимая
$\varnothing$	$\{A\}$	4		
$\{B\}$	$\{A, B\}$	6	✓	✓
$\{C\}$	$\{A, C\}$	5		
$\{D\}$	$\{A, D\}$	7	✓	✓
$\{B, C\}$	$\{A, B, C\}$	7	✓	✓
$\{B, D\}$	$\{A, B, D\}$	9	✓	✓
$\{C, D\}$	$\{A, C, D\}$	8	✓	✓
$\{B, C, D\}$	$\{A, B, C, D\}$	10	✓	

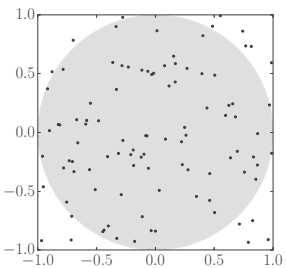
Иной подход прибегает к случаю. Вместо того, чтобы подсчитывать все возможные коалиции и проверять значимы ли они, мы можем просто брать коалиции наугад и через какое-то время проверим случайную выборку всех возможных коалиций. Если говорить в терминах выборки, наша совокупность содержит все возможные коалиции с определенным избирателем, и мы делаем из данной совокупности выборку. Если наша выборка достаточно велика, то соотношение значимых коалиций со всеми возможными коалициями должно быть почти таким же, как соотношение значимых случаев в нашей выборке со всеми коалициями в нашей выборке.

Тут нам поможет метод Монте-Карло, вычислительный подход, который для получения результатов использует случайную выборку. Метод получил свое название в честь известного казино. Методы Монте-Карло имеют выдающееся происхождение; их разработали люди, занимавшиеся разработкой первых цифровых компьютеров. Данные методы применяются в различных сферах, от естественных наук и техники до финансов и бизнеса.

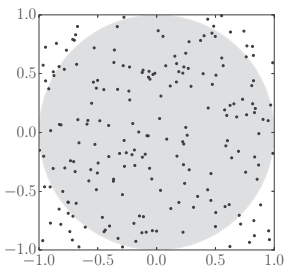
Прежде чем узнать, как метод Монте-Карло может помочь в наших расчетах меры Банцафа, полезно рассмотреть более простые случаи его применения. Лобовой метод Монте-Карло — похожий по сути на тот, что мы будем применять в расчетах меры Банцафа — можно использовать для вычисления значения π. Если у нас есть квадрат со сторонами длиною в две единицы, его площадь будет равна четырем единицам. Если мы впишем в этот квадрат круг, его диаметр будет равен двум единицам, а радиус — одной единице. Следовательно, площадь круга будет равна π. Если мы разбросаем по квадрату мелкие элементы, скажем, рис, некоторые зернышки окажутся внутри круга, а некоторые за его пределами. Если мы разбросаем достаточно элементов, соотношение между теми, что упали внутрь, и общим количеством разбросанных элементов будет равно π/4. На рисунке 16.6 показаны состояния данного процесса после разбрасывания 100, 200, 500 и 1000 случайных точек. Вы можете видеть вычисленное значение π и среднеквадратическую погрешность (s_e) каждого случая. Обратите внимание, что значение π остается неизменным, когда мы переходим от 200 к 500 точкам, однако мы видим изменение в точности, так как среднеквадратическая погрешность уменьшается. Когда же мы доходим до 1000 случайных точек, то получаем $\pi \approx 3{,}14$ с погрешностью 0,005.

Возможно, вы задаетесь вопросом, как мы получаем значения погрешности. Так как процедура у нас рандомизированная, не стоит ждать совершенно точного значения: такое вряд ли случится. В наших результатах всегда будет какая-то погрешность. Мы получаем объем погрешности, основываясь на статистике. Каждая точка находится либо внутри, либо снаружи круга. Мы определяем переменную X, которая равна единице, если точка на-

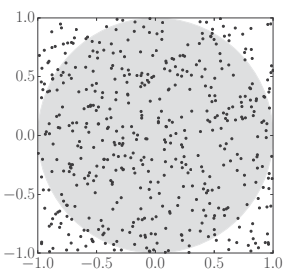
ходится внутри круга, и равна нулю, если точка снаружи. Ожидаемое значение X — это $E[X] = \pi/4 \times 1 + (1 - \pi/4) \times 0 = \pi/4$. Вариантность X — это $\sigma^2 = E[X^2] - (E[X])^2$. Мы имеем $E[X^2] = (\pi/4) \times 1^2 + [1 - (\pi/4)] \times 0^2 = \pi/4$. Поэтому $\sigma^2 = \pi/4 - (\pi/4)^2 = (\pi/4)(1 - \pi/4)$. Среднеквадратическая погрешность тогда $s_e = \sigma/\sqrt{n}$, где n является числом точек, а σ, квадратный корень вариативности, представляет собой среднеквадратическое отклонение. Именно по этой формуле мы рассчитали среднеквадратическую погрешность на рисунке 16.6. Согласно статистике, с помощью среднеквадратической погрешности s_e мы с вероятностью 95% ожидаем, что истинное значение π будет находиться в пределах $\pm 1{,}96 s_e$ вычисленного значения.



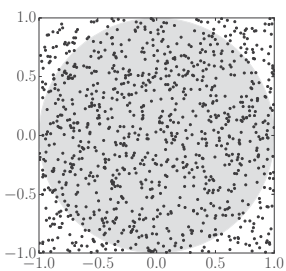
(a) 100 точек, π = 3,2, se = 0,017



(b) 200 точек, π = 3,12, se = 0,012



(c) 500 точек, π = 3,12, se = 0,008



(d) 1000 точек, π = 3,14, se = 0,005

Рисунок 16.6. Вычисление π методом Монте-Карло

Хотя данный метод нахождения π может служить прекрасной иллюстрацией при знакомстве с методами Монте-Карло в целом, на деле он не так хорош в расчетах π. Существуют более целесо-

образные способы, однако здесь он выигрывает с педагогической точки зрения.

Вернувшись к голосованию, мы сгенерируем случайные коалиции и проверим, значимы ли они. Из соотношения тех, которые значимы, ко всем коалициям, которые мы генерируем, мы получим приблизительное значение меры Банцафа. Случайная коалиция является случайным подмножеством, поэтому первым делом нам нужно найти способ генерировать случайные подмножества.

Алгоритм 16.6. Генерирование случайного подмножества

```
RandomSubset(S) → RS
    Вводные данные: S, множество.
    Выводимые данные: RS, случайное подмножество S.
1   RS ← CreateList()
2   foreach m in S do
3       r ← Random(0, 1)
4       if r < 0.5 then
5           InsertInList(RS, NULL, m)
6   return RS
```

Тут нам помогает алгоритм 16.6. Он берет в качестве вводных данных множество *S* и возвращает случайное подмножество *RS* из элементов *S*. Сначала, в строчке 1, *RS* назначается пустым списком. Затем в цикле строчек 2–5 мы перебираем элементы *S*. На каждом элементе мы по сути подбрасываем монетку, решая, включить его в случайное множество или нет. Для этого в строчке 3 мы берем случайное число из промежутка [0, 1) и в строчке 4 сравниваем его с 0,5. Если оно меньше 0,5, в строчке 5 включаем его в *RS*. Мы возвращаем *RS*, как только он заполнится, в строчке 6.

Алгоритм 16.6 является общим алгоритмом для генерации случайных подмножеств; он не ограничен одними только коалициями, и мы можем применять его в любых задачах, когда нам нужна рандомизированная выборка (случайного размера) из множества. В особом случае, когда нам надо вычислить меру Банцафа с помощью метода Монте-Карло, он позволяет прийти к простому алгоритму, удобному для наших расчетов, алгоритму 16.7.

Данный алгоритм берет в качестве вводных данных избирателя *v*, чью меру Банцафа мы хотим узнать, список *ov*, содержащий остальных избирателей, квоту *q*, которую необходимо набрать, ассоциативный массив *w*, содержащий вес голоса каждого избирателя, и число попыток, *t*, сколько раз мы будем искать значимую коалицию.

Алгоритм 16.7. Вычисление меры Банцафа с помощью метода Монте-Карло

`BanzhafMeasure(`v, ov, q, w, t`)` $\rightarrow b$

Вводные данные: *v*, избиратель; *ov*, список, содержащий других избирателей; *q*, требуемая квота; *w*, ассоциативный массив, содержащий вес каждого избирателя; *t*, число попыток.

Выводимые данные: *b*, мера Банцафа для избирателя *v*.

```
1  k ← 0
2  nc ← 0
3  while k < t do
4      coalition ← RandomSubset(ov)
5      votes ← 0
6      foreach m in coalition do
7          votes ← votes + Lookup(w, m)
8      if votes < q and votes + Lookup(w, v) ≥ q then
9          nc ← nc + 1
10     k ← k + 1
11 b ← nc/k
12 return b
```

Мы подсчитываем количество попыток в переменной *k*, которую в строчке 0 мы устанавливаем на ноль. Мы ведем счет найденным значимым коалициям в переменной *nc*, которую мы изначально также устанавливаем на ноль (строчка 1). Цикл в строчках 3–10 повторяется *t* раз. При каждом проходе цикла мы получаем случайную коалицию с помощью алгоритма 16.6. Нам нужно найти число избирателей в каждой случайной коалиции; изначально их ноль, так мы прописываем в строчке 5. Затем в промежуточную сумму к каждому участнику коалиции (цикл

в строчках 6–9) мы добавляем голоса участников в строчке 7. Если промежуточная сумма меньше квоты, однако набирает ее с помощью голосов избирателя v (строчка 8), мы получаем значимую коалицию, поэтому мы увеличиваем счет (строчка 9). Мы увеличиваем индекс в конце цикла (строчка 10). В завершении алгоритма мы получаем соотношение значимых коалиций и всех коалиций (строчка 11) и затем возвращаем его.

Точно так же, как с вычислением π методом Монте-Карло, нам нужно знать, сколько итераций цикла нам понадобится, чтобы добиться точного результата. Тут нам еще немного поможет математика; оказывается, что, если мы хотим получить результаты в районе $\pm\in$ с вероятностью δ, то требуемое число выборок таково:

$$k \geq \frac{\ln \frac{2}{1-\delta}}{2\epsilon^2}.$$

Мы можем использовать алгоритм 16.7 для вычисления меры Банцафа в примере из реальной жизни, таком как президентские выборы США. Президент США выбирается народом не напрямую, а через коллегию выборщиков. Она состоит из числа выборщиков от каждого штата и округа Колумбия. После местных выборов партия, которая победила в каждом отдельном штате и округе Колумбия, назначает там выборщиков (почти: Мэн и Небраска могут разделить своих выборщиков, но мы закрываем глаза на такую возможность). Затем эти выборщики голосуют непосредственно за президента, которому нужно набрать большинство голосов. Всего 538 выборщиков, квота для избрания в президенты составляет 270 голосов.

Число выборщиков на каждый штат меняется в зависимости от последней переписи населения. В 2015 году в Калифорнии было 55 выборщиков, в то время как в Вермонте всего 3. С помощью этих данных мы получаем меру Банцафа для всех штатов и округа Колумбия в таблице 16.3. Штаты и округ Колумбия идут в порядке убывания меры Банцафа.

У Калифорнии наибольшее влияние, но вместе с тем и крохотный Вермонт не декоративная пустышка. Банцафова мера Калифорнии примерно в 20,65 раз больше меры Вермонта, коллегия

выборщиков Калифорнии примерно в 18,33 раз больше коллегии Вермонта, поэтому Калифорния получает несколько больше влияния, чем можно было бы ожидать, достаточно просто взглянуть на соотношение выборщиков.

Таблица 16.3. Коллегия выборщиков США, число выборщиков и мера Банцафа

CA	55	0.475	MD	10	0.076	UT	5	0.038
TX	34	0.266	MN	10	0.076	WV	5	0.038
NY	31	0.241	WI	10	0.076	HI	4	0.030
FL	27	0.209	AL	9	0.069	ID	4	0.030
IL	21	0.161	CO	9	0.068	ME	4	0.030
PA	21	0.161	LA	9	0.068	NH	4	0.030
OH	20	0.153	KY	8	0.061	RI	4	0.030
MI	17	0.130	SC	8	0.060	AK	3	0.023
GA	15	0.114	CT	7	0.053	DC	3	0.023
NC	15	0.114	IA	7	0.053	DE	3	0.023
NJ	15	0.114	OK	7	0.053	MT	3	0.023
VA	13	0.099	OR	7	0.053	ND	3	0.023
MA	12	0.091	KS	6	0.046	SD	3	0.023
IN	11	0.084	MS	6	0.046	VT	3	0.023
MO	11	0.084	AR	6	0.045	WY	3	0.023
TN	11	0.084	NE	5	0.038			
WA	11	0.083	NM	5	0.038			
AZ	10	0.076	NV	5	0.038			

Любопытный читатель может сопоставить и другие пары, чтобы найти какое-либо противоречие. Если говорить о точности результатов, метод Монте-Карло выполнен с $\in = 0{,}001$ и $\delta = 0{,}95$, что требует 1 844 440 выборок для каждой меры Банцафа. Число немаленькое, но все же пустячное по сравнению с числом возможных подмножеств множества из 51 участника.

16.4. В поисках простых чисел

Во многих областях криптографии очень часто ключевым моментом является нахождение больших простых чисел. Под большими простыми числами там часто подразумеваются простые числа из m битов, где m — это огромная степень двух (1024, 2048, 4096, …). Надежность таких криптографических алгоритмов, как RSA и об-

мен ключами по Диффи — Хеллману, зависит от них, равно как зависят и криптографические протоколы, встроенные в программы, и устройства, которыми мы пользуемся каждый день.

Отыскать большие простые числа не так-то просто. Мы знаем, что существует бесконечное множество простых чисел. Также нам примерно известно, сколько существует простых числе меньше или равных числу n. Согласно теореме о распределении простых чисел, если n велико, то количество простых чисел менее или равных n будет приблизительно $n/\ln n$. Загвоздка в том, как найти одно из них.

Можно найти все простые числа с m битами, которые являются всеми простыми числами меньше или равными $n = 2^m$, и выбрать из них одно. Существует несколько способов нахождения всех простых чисел меньше или равных n. Самый лучший — решето Эратосфена, древний алгоритм, названный в честь греческого математика Эратосфена Киренского, жившего в 276–194 годах до н.э. Алгоритм находит простые числа путем вычеркивания чисел, которые не являются простыми; те, что остаются, — простые, отсюда и название «решето». Мы начинаем с простого числа 2. Вычеркиваем все числа, которые меньше или равны n, кратные двум; конечно же, это составные числа. Затем мы переходим от последнего найденного простого числа, 2, к первому числу, не отмеченному как составное. Перед нами число 3, и оно простое. Мы снова вычеркиваем все числа меньше или равные n, которые кратны трем. Затем мы снова переходим от последнего простого числа, 3, к первому не зачеркнутому. Перед нами число 5, и оно простое. Мы продолжаем в том же духе, пока у нас не останется чисел в пределах n. Суть такова, что каждое число p, которое мы находим и которое не отмечено как составное, является простым, потому что мы вычеркнули все кратные для всех чисел меньше p; следовательно, p не кратно никакому числу, а значит, оно простое. В конце процедуры мы получаем незачеркнутые простые числа.

На рисунке 16.7 показана операция при $n = 31$. Мы проверяем числа кратные двум и трем; число 4 закрашено как составное, поэтому мы переходим к числу 5 и обнаруживаем, что составных чисел больше не осталось. Это не случайность. Для любого n все

составные числа, которые больше или равны $\sqrt{n}$ оказываются закрашены, когда мы проверяем кратные числа из чисел $p \le \sqrt{n}$. В самом деле, любое составное число c, которое $\sqrt{n} \le c \le n$, может быть записано в виде произведения двух множителей $c = f_1 \times f_2$, где выполняется как минимум одно условие, $f_1 \le \sqrt{n}$ или $f_2 \le \sqrt{n}$ (мы получаем равенство, когда $c = n$); но тогда оно уже будет отмечено как составное, так как является числом, кратным f_1 и f_2.

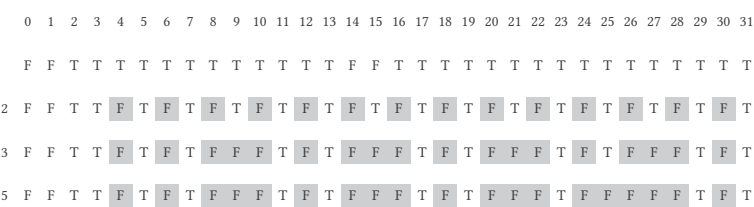


Рисунок 16.7. Решето Эратосфена для $n = 31$

Можно обратить внимание на еще один момент. Когда мы начинаем вычеркивать числа кратные p, мы сразу же можем вычеркнуть p-ое кратное число, p^2. Такое возможно потому, что все $p \times 2$, $p \times 3$, …, $p \times (p - 1)$ уже просмотрены во время проверки чисел 2, 3, …, $p - 1$.

Решето Эратосфена представлено в алгоритме 16.8. Он берет в качестве вводных данных натуральное число $n > 1$ и возвращает массив *isprime*, в котором, если $p \le n$ — это простое число, *isprime*[p] будет `true`, а если иначе, то будет `false`. Сначала, в строчках 1–5, мы создаем и запускаем массив *isprime* так, чтобы все его элементы кроме первых двух, были `true`, то есть, мы предварительно помечаем их как простые; числа 0 и 1 не рассматриваются как простые. Обратите внимание, что размер массива $n + 1$, чтобы представить все числа от 0 до n включительно. Потом, в строчке 6, мы присваиваем p значение 2, так как 2 является простым числом. Цикл в строчках 7–13 помечает составные числа. В строчке 7 мы прибегаем к условию $p^2 \le n$ вместо $p \le \sqrt{n}$, потому что на вычисление квадратного корня, как правило, нужно больше времени, чем на возведение в квадрат. Если p не отмечено как составное число (строчка 8), тогда в строчках 9–12 мы указываем составными числа $p \times p$, $p \times (p + 1)$, … $p \times [n/p]$. Мы назначаем j равное p в строчке 9,

а затем проходим цикл до тех пор, пока j не больше $[n/p]$ (строчка 10). В цикле мы формируем кратное число $j \times p$ и отмечаем соответствующий элемент в *isprime* (строчка 11), потом переходим к следующему j (строчка 12). Если p уже отмечено как составное, оно является кратным числу $p' < p$, которое мы рассматривали до него, следовательно все кратные ему числа кратны p' и нам не нужно проходить строчки 9–12. Увеличиваем p в строчке 13 так, чтобы мы могли начать другой внешний цикл. Когда со всеми циклами покончено, мы в строчке 14 возвращаем *isprime*.

Алгоритм 16.8. Решето Эратосфена

`SieveEratosthenes(`n`)` $\longrightarrow$ *isprime*

Вводные данные: n, натуральное число больше 1.

Выводимые данные: *isprime*, такой булев массив размером $n + 1$, что, если $p \le n$ является простым, *isprime*$[p]$ будет `true`, а иначе — `false`.

```
1  isprime ← CreateArray(n + 1)
2  isprime[0] ← FALSE
3  isprime[1] ← FALSE
4  for i ← 2 to n + 1 do
5      isprime[i] ← TRUE
6  p ← 2
7  while p^2 ≤ n do
8      if isprime[p] = TRUE then
9          j ← p
10         while j ≤ ⌊n/p⌋ do
11             isprime[j × p] ← FALSE
12             j ← j + 1
13     p ← p + 1
14 return isprime
```

Внешний цикл алгоритма выполняется $\sqrt{n}$ раз. Внутри цикла мы избавляемся от всех чисел кратных 2, которые равны $[n/2]$; затем от всех чисел кратных 3, которые равны $[n/3]$; затем от всех чисел кратных 5, которые равны $[n/5]$; и так далее до наибольшего простого числа k при $k \le \sqrt{n}$. Таким образом, мы избавились от всех $n/2 + n/3 + n/5 + \ldots + n/k$ составных чисел, которые равны $n(1/2 + 1/3 + 1/5 + \ldots + 1/k)$. Сумма $(1/2 + 1/3 + 1/5 + \ldots + 1/k)$ является суммой обратных дробей простых чисел, которые не больше $\sqrt{n}$.

В целом можно сказать, что сумма обратных дробей простых чисел, которые не больше числа m, равна $O(\log \log m)$. Следовательно, общее время, которое мы тратим на вычеркивание, равно $O(n \log \log \sqrt{n}) = O(n \log \log n)$, что составляет сложность алгоритма.

Существуют более эффективные алгоритмы нахождения всех простых чисел меньше или равных какому-либо числу. Они выполняются за $O(n)$ и даже $O(n/(\log \log n))$ время. Наверняка вам кажется это очень выгодным, ведь мы тогда с легкостью можем находить простые числа.

К сожалению, все не так просто. $O(n)$ или $O(n/(\log \log n))$ показывает сложность с точки зрения размера числа, но тоже укрывает размер вне n. Как мы говорили, $n = 2^m$, при большом m, поэтому подлинная сложность, которая требуется $O(n)$, равна $O(2^m)$ этапов: она экспоненциальная, если говорить с точки зрения битов вводимых чисел, а не линейная, как мы сперва подумали. Для 4096 битов мы получаем $O(2^{4096})$, чрезмерное количество. $O(n/(\log \log n))$ лишь в некоторой степени делит данное число, не меняя общей картины.

Так как нахождение всех простых чисел нецелесообразно плюс мы знаем, что существует примерно $n/\ln n$ простых чисел вне заданных ограничений, мы можем попытать удачу и взглянуть, что случится, если мы возьмем одно число, меньше или равное n и проверим, простое ли оно. Если мы берем все n чисел, мы ожидаем найти $n/\ln n$ простых чисел; если мы берем одно число, то вероятность, что мы наткнемся на простое, составляет $1/\ln n$. И наоборот, теория вероятности говорит нам, что для нахождения простого числа нам нужно перепробовать n чисел. Скажем, мы ищем простое число из 4096 битов. Нам надо испробовать $\ln(2^{4096})$ чисел, чтобы встретить простое, и нам обязательно надо проверять каждое из них, действительно ли оно простое.

Простейший способ узнать, является ли число простым, — посмотреть, делится ли оно на какое-либо число кроме единицы. Когда речь о числе n, нам достаточно проверить делится ли оно на любое число до $\lfloor \sqrt{n} \rfloor$. $\sqrt{n}$ нам вполне хватает, потому что число больше $\lceil \sqrt{n} \rceil$ может дать n, только если его умножить на число не больше, чем $\lfloor \sqrt{n} \rfloor$, которое мы уже проверили. Так-

же можно разбить это число пополам, потому как можно пропустить все четные числа больше 2: если n делится на любое четное число больше 2, то оно делится и на 2. Если мы полагаем, что деление занимает один этап, то алгоритму потребуется $O\left((1/2)\sqrt{n}\right) = O(\sqrt{n})$ этапов, что выглядит не так уж плохо. Еще раз: это иллюзия. Размер наших вводных данных равен размеру числа n, что в двоичной системе будет 2^m, и значит, алгоритму потребуется $O\left(\sqrt{2^m}\right) = O\left((2^m)^{1/2}\right) = O(2^{m/2})$. Даже если нужно наугад выбрать одно небольшое число из множества чисел, нам понадобится приличное количество времени на выяснение, является ли случайно выбранное число простым.

К счастью, есть эффективный способ узнать, является ли число простым. Метод, который мы рассмотрим, всегда будет сообщать нам, что простое число в самом деле простое — таким образом, не будет ложноотрицательных результатов, говорящих, что простое число на самом деле составное. Он также будет сообщать, что составное число является составным в большинстве случаев. Останется вероятность, что метод не сможет опознать составное как таковое и сообщит нам, что оно простое, — мы получим ложноположительный результат. Тем не менее мы увидим, что вероятность достаточно невелика и никак не сказывается на практической пользе метода. Мы имеем дело с проверкой вероятных простых чисел, в которой все шансы играют нам на руку.

В основе данной проверки лежат некоторые утверждения из теории чисел. Если случайное число, о котором мы хотим узнать, простое ли оно, — это p, то p должно быть нечетным; если оно четное, мы тут же отбрасываем его как составное. Отсюда получается, что $p - 1$ должно быть четным. Если мы снова и снова делим любое четное число на 2, то мы дойдем либо до 1, либо до другого нечетного числа. Например, если неоднократно делим 12 на 2, мы получаем сначала 6, потом 3, поэтому $12 = 2^2 \times 3$. Если мы неоднократно делим 16 на 2, мы получаем 8, 4, 2, 1, поэтому $16 = 24 \times 1$. Если говорить в целом, то мы имеем $p - 1 = 2^r q$, где $r \geq 1$, а q — нечетное число.

Теперь давайте возьмем еще одно случайное число x, чтобы $1 < x < p$, и вычислим $y = x^q \bmod p$. Если $y = 1$, то есть $x^q \bmod p = 1$, мы

имеем $(x^q)^t \bmod x = 1$ при $t \geq 0$. Бесспорно, это следует из утверждения, что для любых целых чисел a и b у нас есть $[(a \bmod p)(b \bmod p)] \bmod p = (a \cdot b) \bmod p$. Берем $a = b$, неоднократно применяем — и получаем $(a \bmod p)^t \bmod t = a^t \bmod p$, поэтому, если $a \bmod p = 1$, у нас $a^t \bmod p = 1$. Затем, заменив t на 2^r в $(x^q)^t$, мы получаем $(x^q)^{2^r}$ **mod** $p = 1$, $x^{2^r q}$ **mod** $p = 1$ или x^{p-1} **mod** $p = 1$. Согласно малой теореме Ферма, если p является простым числом, то данная зависимость должна обязательно сохраняться. И наоборот, если зависимость сохраняется, то p вероятно является простым числом, однако может им и не быть. Мы уже встречались прежде с малой теоремой Ферма, когда говорили о криптосистеме RSA в разделе 5.2.

Давайте подытожим, что же мы успели проделать. Мы выбрали случайное число p, желая определить, простое оно или нет. Записываем его в виде $p = 1 + 2^r q$. Затем берем случайное число x при $1 < x < p$ и вычисляем $y = x^q \bmod p$. Если $y = 1$, мы можем сказать, что p возможно является простым числом. Назовем данную ситуацию «Вещдоком А» и чуть позже увидим, почему.

Если $y = x^q \bmod p \neq 1$, мы можем начать процесс возведения в квадрат, создавая значения:

$$(x^q)^2 \bmod p = x^{2q} \bmod p$$
$$(x^{2q})^2 \bmod p = x^{4q} \bmod p$$
$$\ldots$$
$$(x^{2^{r-1}q})^2 \bmod p = x^{2^r q} \bmod p = x^{p-1} \bmod p.$$

Если p простое, то благодаря малой теореме Ферма последовательность значений закончится на единице. На самом деле мы можем получить 1 еще прежде, чем доберемся до r-го возведения в квадрат: все последующие возведения в квадрат будут давать единицу, как мы видели выше. Более того, значение, которое мы получаем до первой единицы, обязательно должно быть $p - 1$.

Если говорить, почему так происходит (если вы и так знаете ответ, переходите к следующему абзацу), обратимся к теории чисел. Для любого числа y, если у нас $y^2 \bmod p = 1$, где p — простое число, это означает, что $y^2 = kp + 1$ при целом числе k или $y^2 - 1 = kp$, или $(y - 1)(y + 1) = kp$. Чтобы так случилось, если $k \neq 0$, число p должно делиться на $(y - 1)/k$ или $(y + 1)/k$, чего не может быть, потому

что p простое. Следовательно, у нас должно быть $k = 0$; тогда будет $y - 1 = 0$ или $y + 1 = 0$, что значит, у нас может быть только $y = 1$ или $y = -1$. Поэтому при последовательном возведении в квадрат прежде, чем получить в первый раз $y = 1$, у нас должно быть $y = -1$. Мы должны выразить это в терминах арифметики по модулю, где $0 < y < p$. Вспомните, что согласно математическому определению модуль, $a \bmod b$, представляет собой число $c \geq 0$, такой остаток, что $a = qb + c$, где q является нижней границей деления a/b, $\lfloor a/b \rfloor$. Мы сталкивались с данным определением в разделе 4.2, когда знакомились с оператором деления по модулю. Таким образом, мы получаем $c = a - b\lfloor a/b \rfloor$. Остаток от деления -1 на p будет тогда равняться $-1 \bmod p = -1 - p\lfloor -1/p \rfloor = -1 - p(-1) = p - 1$. Так что в самом деле, ровно перед получением $y = 1$ мы должны получить $y = p - 1$.

Итак, снова подведем итог. Если мы начинаем неоднократное возведение в квадрат $y = x^q \bmod p$ и p у нас простое число, в какой-то момент мы получим $y = p - 1$; тогда следующее возведение в квадрат (которое вообще-то нам не нужно делать) даст нам $y = 1$. И наоборот, если мы добираемся до $y = p - 1$, ничего не зная о p, то вероятно, что p является простым числом, хотя и не обязательно. Назовем такую ситуацию, когда $y = p - 1$, «Вещдоком Б».

Если мы добрались до $y = 1$, не получив в предыдущем возведении в квадрат $y = -1$, то нам точно известно, что p не простое число, потому что все доводы, приведенные в предыдущих абзацах, показывают, что, если бы оно было простым, мы бы ровно перед ним получили $y = -1$. Мы назовем такую ситуацию, когда $y = 1$ без получения $y = p - 1$ до, «Вещдоком В».

Наконец, если мы добрались до $y = x^{2^r q} \bmod p = x^{p-1} \bmod p$ и $y \neq 1$, тогда, согласно малой теореме Ферма, мы с уверенностью можем сказать, что наш число не простое. Назовем данный случай «Вещдоком Г».

Как мы видим, у нас получилось найти надежные, точные, признаки того, что число p является составным: вещдок В и вещдок Г; в то же время у нас есть вероятные признаки того, что p является простым числом: вещдок А и вещдок Б. Мы называем функцию, которая подтверждает определенный интересующий нас признак, доказательством: таким образом, мы можем исполь-

зовать результаты наших исследований для построения доказательства для p, являющегося составным; доказательство показано в алгоритме 16.9.

Алгоритм 16.9. Доказательство для составных чисел

`WitnessComposite(`p`)` → TRUE or FALSE

Вводные данные: p, целое нечетное число.

Выводимые данные: булево значение, которое `true`, если число совершенно точно составное, а иначе — `false` .

```
1  (r, q) ← FactorTwo(p − 1)
2  x ← RandomInt(2, p − 1)
3  y ← x^q mod p
4  if y = 1 then
5      return FALSE
6  for j ← 0 to r do
7      if y = p − 1 then
8          return FALSE
9      y ← y^2 mod p
10     if y = 1 then
11         return TRUE
12 return TRUE
```

Доказательный алгоритм поразительно небольшой, если учесть все доводы, которые привели нас к нему. По сути он очень прост, хотя в основе его лежит существенный фундамент.

Мы начинаем с вызова в строчке 1 `FactorTwo`(p – 1), которая возвращает r и q такими, что $p - 1 = 2^r q$; очень скоро мы вернемся к `FactorTwo`. Затем, в строчке 2, `RandomInt`(2, p – 1) создает целое число в интервале от 2 до p – 1 включительно. В строчке 3 мы вычисляем $x^q \bmod p$ и помещаем его в y. Из вещдока А мы знаем, что, если $y = 1$ (строчка 4), число p вероятно является простым, поэтому мы возвращаем `false` в строчке 5. Если это не так, в цикле строчек 6–11 мы начинаем последовательно возводить в квадрат, самое большое r раз. Если $y = p - 1$ в какой-либо итерации цикла (строчка 7), то следующее возведение в квадрат даст $y = 1$, и из вещдока Б мы делаем заключение, что число вероятно простое, и возвращаем `false` в строчке 8. После возведения в квадрат y, по модулю p, в строчке 9 мы проверяем, полу-

чили мы $y = 1$ или нет. Если да, то получили мы его, не проходя $y = p - 1$, а значит, мы совершенно точно можем сказать (благодаря вещдоку В), что число p является составным, и возвращаем `true`. Если мы выходим из цикла, то перед нами вещдок Г и мы в строчке 12 возвращаем `true`, потому что p, вне сомнений, является составным числом.

Алгоритм 16.10. Тест Миллера — Рабина

`MillerRabinPrimalityTest(`p, t`)` → TRUE or FALSE

Вводные данные: p, целое нечетное число; t, количество раз, когда будет применяться функция доказательства простоты.

Выводимые данные: `true`, если число простое с вероятностью $(1/4)^t$, и `false`, если число совершенно точно составное.

```
1  for i ← 0 to t do
2      if WitnessComposite(p) then
3          return FALSE
4  return TRUE
```

Чтобы доказательный алгоритм оказался практичным, нам нужно знать, что вероятность его ошибки приемлемо мала. Выясняется, что алгоритм ошибается в 1/4 случаев, что и является ключом к его применению на практике. Если мы используем его всего раз, то с вероятностью 1/4 он выдаст нам составное число, назвав его простым. Если мы воспользуемся им дважды, тогда вероятность его ошибки в обоих случаях будет $(1/4)^2$. Если мы станем вызывать его снова и снова, мы сможем понизить вероятность до любого нужного нам уровня. Например, если мы вызываем алгоритм 50 раз, вероятность ошибочного результата будет равна $(1/4)^{50}$, что вполне приемлемо для его применения на практике.

Неоднократный прогон доказательного алгоритма называется тестом Миллера — Рабина на простоту, в честь Гари Л. Миллера и Михаэля О. Рабина, чьи идеи легли в его основу. Имея `WitnessComposite(`p`)`, алгоритм Миллера — Рабина очень прост; взгляните на алгоритм 16.10.

С точки зрения сложности, тест Миллера — Рабина довольно продуктивен. Возвращаясь к алгоритму 16.9, мы видим, что

возведение в степень по модулю в строчке 3 встречается только один раз и может быть эффективно выполнено за $O((\lg p)^3)$, так как $q < p$. Действительно, мы видели, как осуществлять быстрое возведение в степень в разделе 4.5, где мы также познакомились с этими сложными числами. Цикл строчке 6–11 выполняется $O(r)$ раз, где $r < \lg p$, поэтому у нас $O(\lg p)$ итераций цикла. При каждой итерации мы выполняем возведение в квадрат по модулю. Учитывая, что все $y < p$, возведению в квадрат по модулю требуется $O((\lg p)^2)$ времени; пройдя все итерации цикла, мы получаем $O((\lg p)^3)$. Мы можем допустить, что время, которое требуется `RandomInt`, будет меньше.

Алгоритм 16.11. Множитель n в качестве $2^r q$ при нечетном q

`FactorTwo`$(n) \to (r, q)$

Вводные данные: n, целое четное число.

Выводимые данные: (r, q), такое, что $n = 2^r q$, где q — нечетное число.

```
1  q ← n
2  r ← 0
3  while q mod 2 = 0 do
4      r ← r + 1
5      q ← q/2
6  return (r, q)
```

Теперь осталось только рассказать о функции `FactorTwo`, выдающей множитель $p - 1$, который умножается на наибольшую возможную степень двух. Мы можем записать данную функцию в виде ряда повторяющихся операций деления; взгляните на алгоритм 16.11. В строчке 1 его вводным данным, n, присваивается значение q; в конце алгоритма q будет нечетным числом, которое является остатком от n, поделенном на степень двух. Степень двух, которую мы ищем, — это r, которой мы изначально, в строчке 2, присваиваем нулевое значение. В цикле строчек 3–5 мы проверяем, является ли q четным (строчка 3); если да, то мы знаем, что оно делится на два, поэтому мы увеличиваем r (строчка 4) и производим деление (строчка 5). Если q нечетное, то мы ничего не делаем и возвращаем (r, q).

Неоднократное деление приводит к тому, что число этапов всего процесса является логарифмом с основанием два для вводных данных, которые представляют собой число $p - 1$, использованное нами в `WitnessComposite`, поэтому сложность `FactorTwo` равна $O(\lg p)$. Это никак не сказывается на общей сложности `WitnessComposite`, которой, следовательно, требуется $O((\lg p)^3)$ этапов, что очень хорошо. Для t итераций `WitnessComposite` мы получаем $O(t \cdot (\lg p)^3)$, так что у нас есть целесообразный метод нахождения больших простых чисел с размером до p. Мы продолжаем угадывать простое число; каждая попытка требует $O(t \cdot (\lg p)^3)$ этапов для проверки, а нам нужно p попыток.

Кстати, 50 итераций — скорее всего выше крыши. Куда с большей вероятностью, чем $(1/4)^{50}$, ваш компьютер выдаст ошибку по какой-то другой причине: технические неполадки, какое-нибудь электромагнитное вмешательство, можно даже подумать в сторону космических лучей, пронзивших атмосферу и добравшихся до микросхем.

Примечания

Попытки сгенерировать случайные числа на компьютерах стары, как и сами компьютеры. Компьютер «Ферранти Марк I», созданный в 1951 году, заключал в своей железной начинке генератор случайных чисел, принимавший установки Алана Тьюринга. Деррик Генри Лемер предложил линейный конгруэнтный генератор в 1949 году [127]. Цитата Джона фон Неймана о хаотичности и грешности появилась в раннем собрании работ, касавшихся метода Монте-Карло [211]. Предостережение о получении случайных чисел случайными способами пришло из вводных материалов Кнута, посвященных случайным числам [113, раздел 3.1].

Таблицы подходящих значений для a, m, c были даны Пьером Л'Экуаером [124]. Л'Экуаер и Ричард Симард написали внушительную библиотеку для тестирования генераторов случайных чисел [125]. Генераторы xorshift64* и xorshift1024* были изобретены Себастьяном Вигна [209], в их основу легли xorshift-

генераторы, представленные Джорджем Марсаглия [133]. Рисунок 16.2(b) получен на основе инструкции по созданию карт Хинтона, включенной в галерею примеров matplotlib*; изначальная задумка приписывается Дэвиду Уорду-Фарли.

КСГПСЧ Фортуна был разработан криптографами Нильсом Фергюсоном и Брюсом Шнайером [62, глава 10]; он также описан и в новой редакции оригинальной книги [63, глава 9]. Его предложили как преемника популярного генератора «Ярроу» [106]. Проверка надежности КСГПСЧ требует множества усилий и непрерывной работы; исследования показали, что в плане безопасности «Фортуне» есть еще куда развиваться [53].

Избирательная выборка наряду с другими методами была описана Ч.Т. Фаном, Мервином Э. Мюллером и Иваном Резухой в 1962 году [59]; в том же году, независимо от них, выборка была описана Т. Дж. Джонсом как метод получения «случайной выборки ровно *n* записей из фильма, содержащего *N* записей»; описание заняло не более 24 одностолбцовых строчек [103]. Книга Ива Тилле дает исчерпывающее описание работы разных выборочных алгоритмов [202]. Избирательная выборка также известна как алгоритм S, а накопительная выборка — как алгоритм R, обе они рассмотрены Кнутом [113, раздел 3.4.2]. Кнут приписывает накопительную выборку Алану Ватерману; также она была представлена Маклаудом и Беллхаусом [136] и Джеффри Скоттом Виттером [210]. Тилле отметил, что она являет собой особый случай метода, предложенного Чао [35]. Вы можете найти скрипт из одной строки Перла (Perl) для выбора случайной строки из файла в книге, посвященной Перлу [36, с. 314] (он поможет вам справиться с первым упражнением, приведенным ниже). Метод взвешенной выборки, приведенный во втором упражнении, принадлежит Ефремидису и Спиракису [55].

Первая работа, посвященная измерению влияния голоса, вышла в 1946 году и принадлежит Лайонелу Пенроузу [156]; Пенроуз по сути описал меру Банцафа, но его исследование осталось

---

* Matplotlib — библиотека на языке Python для визуализации данных с помощью двумерной и трехмерной графики.

незамеченным. В 1954 году появилась работа Ллойда С. Шепли и Мартина Шубика, которые описали иную систему измерения, индекс влияния Шепли — Шубика. Джон Ф. Банцаф опубликовал свой труд в 1955 году [9]. Количество итераций, требуемых для точного определения меры Банцафа выведено в [6]. Анализ алгоритмов для вычисления индексов влияния при взвешенной системе голосования можно найти в [134]. Детальное описание влияния голоса есть в книге Фелсенталя и Маковера [61]; также рекомендуем посмотреть книгу Тейлора и Пачелли [200]. Последнее время мера Банцафа подвергается критике за несоответствие реальным системам голосования, потому что в ней слишком много различных вероятностных допущений [77].

Чтобы узнать о сложности выборок простых чисел, обратитесь к отчету Соренсона [191]. Гари Л. Миллер впервые предложил тест на простоту в 1975 году [140]; данный тест не вероятностный, а обусловленный, однако базируется на бездоказательных математических гипотезах. Михаэль О. Рабин несколько лет спустя усовершенствовал его и получил вероятностный алгоритм, который не зависит от бездоказательных математических гипотез [162]. Кнут отметил, что компьютер скорее пострадает от космического излучения, чем от ошибочного результата Миллера — Рабина [113, раздел 4.5.4].

Упражнения

1. Как бы вы выбрали наугад одну строчку из файла, не читая его? То есть вы не знаете, сколько всего там строчек. Можно использовать накопительную выборку с резервуаром размером 1. Это значит, что при прочтении первой строки вы выбираете ее с вероятностью 1. Когда вы читаете вторую строку (если таковая имеется), вы выбираете ее с вероятностью 1/2, поэтому у строк 1 и 2 одинаковая вероятность того, что их выберут. Когда вы читаете третью строку (опять же, если она есть), то выбираете ее с вероятностью 1/3. Получается, что у строк 1 и 2 вероятность

того, что их выберут, равна 2/3, и они разделяют ее поровну, потому что мы прежде видели, что у каждой из них вероятность выбора равна 1/2; поэтому вероятность выбора каждой из трех страниц составляет 1/3. Мы продолжаем в том же ключе до конца файла. Реализуйте накопительную выборку, чтобы достать из файла случайную строку. Обратите внимание, что данная версия накопительной выборки может оказаться значительно компактней общей версии.

2. Есть случаи, когда нам нужно осуществить выборку согласно заранее заданным весам: то есть вероятность того, что элемент попадет в выборку, должна быть пропорциональна его весу, — это называется взвешенной выборкой. Мы можем осуществить ее с помощью одного из вариантов накопительной выборки. Начинаем с добавления в резервуар первого из m элементов и приписываем каждому элементу i ключ равный u^{1/w_i}, где w_i — это вес, а u — случайное число, равно выбранное из интервала от 0 до 1 (включительно). Затем для k последующих элементов мы снова получаем случайное число u в интервале [0, 1] и вычисляем его ключ u^{1/w_k}; если он больше наименьшего ключа в резервуаре, то мы добавляем в резервуар новый элемент, заменяя им тот, что с наименьшим ключом. Реализуйте данный подход, используя минимальную очередь с приоритетом для нахождения каждый раз элемента с наименьшим ключом в резервуаре.
3. Рассматривая решето Эратосфена, мы упомянули, что прибегаем к условию $p^2 \le n$ вместо $p \le \sqrt{n}$, потому что так, как правило, быстрее. Проверьте данное утверждение: реализуйте обе ситуации и сравните, сколько времени требуется каждой из них.
4. Проводится много выборов, в которых используются веса голосов; рассчитайте меру Банцафа для каких захотите выборов. Сделайте разное количество выборок для каждого выполнения программы, а затем проверьте точность и время выполнения программы.

Библиография

[1] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. Journal of the ACM, 37(2):213–223, April 1990.

[2] Ethem Alpaydın. Introduction to Machine Learning. The MIT Press, Cambridge, MA, 3rd edition, 2014.

[3] Geoffrey D. Austrian. Herman Hollerith: Forgotten Giant of Information Processing. Columbia University Press, New York, NY, 1982.

[4] Bachrach, El-Yaniv, and M. Reinstädtler. On the competitive theory and practice of online list accessing algorithms. Algorithmica, 32(2):201–245, 2002.

[5] Ran Bachrach and Ran El-Yaniv. Online list accessing algorithms and their applications: Recent empirical evidence. In Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '97, pages 53–62, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[6] Yoram Bachrach, Evangelos Markakis, Ezra Resnick, Ariel D. Procaccia, Jeffrey S. Rosenschein, and Amin Saberi. Approximating power indices: Theoretical and empirical analysis. Autonomous Agents and Multi-Agent Systems, 20(2):105–122, March 2010.

[7] Ricardo A. Baeza-Yates and Mireille Régnier. Average running time of the Boyer-Moore-Horspool algorithm. Theoretical Computer Science, 92(1):19–31, January 1992.

[8] Michael J. Bannister and David Eppstein. Randomized speedup of the Bellman-Ford algorithm. In Proceedings of the Meeting on Analytic Algorithmics and Combinatorics, ANALCO '12, pages 41–47, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[9] John F. Banzhaf, III.Weighted voting doesn't work: A mathematical analysis. Rutgers Law Review, 19:317–343, 1965.

[10] Albert-László Barabási. Linked: The New Science Of Networks. Basic Books, 2002.

[11] Albert-László Barabási and Eric Bonabeau. Scale-free networks. Scientific American, 288(5):50–59, May 2003.

[12] J. Neil Bearden. A new secretary problem with rank-based selection and cardinal payoffs. Journal of Mathematical Psychology, 50:58–59, 2006.

[13] Richard Bellman. On a routing problem. Quarterly of Applied Mathematics, 16(1):87–90, 1958.

[14] Frank Benford. The law of anomalous numbers. Proceedings of the American Philosophical Society, 78(4):551–572, 1938.

[15] Arthur Benjamin, Gary Chartrand, and Ping Zhang. The Fascinating World of Graph Theory. Princeton University Press, Princeton, NJ, USA, 2015.

[16] Jon Bentley. Programming Pearls. Addison-Wesley, 2nd edition, 2000.

[17] Jon L. Bentley and Catherine C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. Communications of the ACM, 28(4):404–411, April 1985.

[18] MichaelW. Berry and Murray Browne. Understanding Text Engines: Mathematical Modeling and Text Retrieval. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd edition, 2005.

[19] N. Biggs, E. K. Lloyd, and R. J. Wilson. Graph Theory, 1736–1936. Clarendon Press, Oxford, UK, 1986.

[20] Christopher M. Bishop. Pattern Recognition and Machine Learning. Springer, New York, NY, 2006.

[21] Joshua Bloch. Extra, extra—read all about it: Nearly all Binary Searches and Mergesorts are broken. http://googleresearch.blogspot.it/2006/06/extra-extra-read-all-about-it-nearly.html, June 2 2006.

[22] Joshua Bloch. Effective Java (2nd Edition) (The Java Series). Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2008.

[23] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, July 1970.

[24] James Blustein and Amal El-Maazawi. Bloom filters—a tutorial, analysis, and survey. Technical report, Dalhousie University, Faculty of Computer Science, 2002.

[25] J. A. Bondy and U. S. R. Murty. Graph Theory. Springer, New York, NY, 2008.

[26] Robert S. Boyer and J Strother Moore. A fast string searching algorithm. Communications of the ACM, 20(10):762–772, October 1977.

[27] Steven J. Brams. Mathematics and Democracy: Designing Better Votign and Fair-Division Processes. Princeton University Press, Princeton, NJ, 2008.

[28] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. Classification and Regression Trees. Wadsworth International Group, Belmont, CA, 1984.

[29] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems, 30(1–7):107–117, April 1998.

[30] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. Internet Mathematics, 1(4):485–509, 2003.
[31] Kurt Bryan and Tanya Leise. The $25,000,000,000 eigenvector: The linear algebra behind google. SIAM Review, 48(3):569–581, 2006.
[32] Russell Burns. Communications: An International History of the Formative Years. The Institution of Electrical Engineers, Stevenage, UK, 2004.
[33] Stefan Büttcher, Charles L. A. Clarke, and Gordon Cormack. Information Retrieval: Implementing and Evaluating Search Engines. The MIT Press, Cambridge, MA, 2010.
[34] R. Callon. Use of OSI IS-IS for routing in TCP/IP and dual environments. RFC 1195, December 1990.
[35] M. T. Chao. A general purpose unequal probability sampling plan. Biometrika, 69(3):653– 656, 1982.
[36] Tom Christiansen and Nathan Torkington. Perl Cookbook. O'Reilly, Sebastopol, CA, 2nd edition, 2003.
[37] Richard J. Cichelli. Minimal perfect hash functions made simple. Communications of the ACM, 23(1):17–19, January 1980.
[38] Douglas E. Comer. Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture. Pearson, 6th edition, 2013.
[39] Marquis de Condorcet. Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix. Imprimerie Royale, Paris, 1785.
[40] Stephen A. Cook. Linear time simulation of deterministic two-way pushdown automata. In IFIP Congress 1, pages 75–80, 1971.
[41] Thomas H. Cormen. Algorithms Unlocked. The MIT Press, Cambridge, MA, 2013.
[42] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliffort Stein. Introduction to Algorithms. The MIT Press, Cambridge, MA, 3rd edition, 2009.
[43] T. M. Cover and R. King. A convergent gambling estimate of the entropy of English. IEEE Transactions on Information Theory, 24(4):413–421, September 2006.
[44] Thomas M. Cover and Joy A. Thomas. Elements of Information Theory. Wiley-Interscience, Hoboken, NJ, 2nd edition, 2006.
[45] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. Algorithms on Strings. Cambridge University Press, Cambridge, UK, 2014.
[46] Joan Daemen and Vincent Rijmen. The Design of Rijndael: AES—The Advanced Encryption Standard. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
[47] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. Algorithms. McGraw-Hill, Inc., New York, NY, 2008.

[48] Easley David and Kleinberg Jon. Networks, Crowds, and Markets: Reasoning About a Highly Connected World. Cambridge University Press, New York, NY, USA, 2010.

[49] Butler Declan. When Google got flu wrong. Nature, 494(7436):155–156, 2013.

[50] W. Diffe and M. E. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, 22(6):644–654, November 1976.

[51] E.W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1(1):269–271, December 1959.

[52] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation Onion Router. In Proceedings of the 13th USENIX Security Symposium, Berkeley, CA, USA, 2004. USENIX Association.

[53] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. How to eat your entropy and have it too—optimal recovery strategies for compromised rngs. Cryptology ePrint Archive, Report 2014/167, 2014. http://eprint.iacr.org/.

[54] Arnold I. Dumey. Indexing for rapid random-access memory. Computers and Automation, 5(12):6–9, 1956.

[55] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. Information Processing Letters, 97(5):181–185, 2006.

[56] Leonhardo Eulerho. Solutio problematis ad geometrian situs pertinentis. Commetarii Academiae Scientiarum Imperialis Petropolitanae, 8:128–140, 1736.

[57] Shimon Even. Graph Algorithms. Cambridge University Press, Cambridge, UK, 2nd edition, 2012.

[58] Kevin R. Fall andW. Richard Stevens. TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley, Upper Saddle River, NJ, 2nd edition, 2012.

[59] C. T. Fan, Mervin E. Muller, and Ivan Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. Journal of the American Statistical Association, 57(298):387–402, 1962.

[60] Ariel Felner. Position paper: Dijkstra's algorithm versus Uniform Cost Search or a case against Dijkstra's algorithm. In Proceedings of the 4th Annual Symposium on Combinatorial Search (SoCS), pages 47–51, 2011.

[61] Dan S. Felsenthal and Moshé Machover. The Measurement of Voting Power: Theory and Practice, Problems and Paradoxes. Edward Elgar, Cheltenham, UK, 1998.

[62] Niels Ferguson and Bruce Schneier. Practical Cryptography. Wiley Publishing, Indianapolis, IN, 2003.

[63] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. Cryptography Engineering: Design Principles and Practical Applications. Wiley Publishing, Indianapolis, IN, 2010.

[64] Thomas S. Ferguson. Who solved the secretary problem? Statistical Science, 4(3):282–289, 08 1989.

[65] R. M. Fewster. A simple explanation of Benford's law. The American Statistician, 63(1):26–32, 2009.

[66] Robert W. Floyd. Algorithm 113: Treesort. Communications of the ACM, 5(8):434, August 1962.

[67] Robert W. Floyd. Algorithm 97: Shortest path. Communications of the ACM, 5(6):345, June 1962.

[68] Robert W. Floyd. Algorithm 245: Treesort 3. Communications of the ACM, 7(12):701, December 1964.

[69] L. R. Ford. Network flow theory, 1956. Paper P-923.

[70] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, and Donald Eastlake. The FNV noncryptographic hash algorithm. Internet-Draft draft-eastlake-fnv-09.txt, IETF Secretariat, April 2015.

[71] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM, 34(3):596–615, July 1987.

[72] Edward H. Friend. Sorting on electronic computer systems. Journal of the ACM, 3(3):134–168, July 1956.

[73] Zvi Galil. On improving the worst case running time of the boyer-moore string matching algorithm. Commun. ACM, 22(9):505–508, September 1979.

[74] Antonio Valverde Garcia and Jean-Pierre Seifert. On the implementation of the Advanced Encryption Standard on a public-key cryptocoprocessor. In Proceedings of the 5th Conference on Smart Card Research and Advanced Application Conference—Volume 5, CARDIS'02, Berkeley, CA, USA, 2002. USENIX Association.

[75] Martin Gardner. Mathematical games. Scientific American, 237(2):120–124, August 1977.

[76] Simson L. Garfinkel. Digital forensics. American Scientist, 101(5):370–377, September– October 2013.

[77] AndrewGelman, Jonathan N. Katz, and Francis Tuerlinckx. The mathematics and statistics of voting power. Statistical Science, 17(4):420–435, 11 2002.

[78] Jeremy Ginsberg, MatthewH. Mohebbi, Rajan S. Patel, Lynnette Brammer, Mark S. Smolinski, and Larry Brilliant. Detecting influenza epidemics using search engine query data. Nature, 457(7232):1012–1014, 2009.

[79] Oded Goldreich. Foundations of Cryptography: Basic Tools. Cambridge University Press, Cambridge, UK, 2004.

[80] Oded Goldreich. Foundations of Cryptography: II Basic Applications. Cambridge University Press, Cambridge, UK, 2009.

[81] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. Communications of the ACM, 42(2):39–41, February 1999.

[82] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. Data Structures & Algorithms in Python. John Wiley & Sons, Hoboken, NJ, 2013.

[83] Robert M. Gray. Entropy and Information Theory. Springer, New York, NY, 2nd edition, 2011.

[84] John Guare. Six Degrees of Separation: A Play. Random House, New York, NY, 1990.

[85] Dan Gusfield. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge, UK, 1997.

[86] David Harel and Yishai Feldman. Algorithmics: The Spirit of Computing. Pearson Education, Essex, UK, 3rd edition, 2004.

[87] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems, Science, and Cybernetics, 4(2):100–107, July 1968.

[88] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "A formal basis for the heuristic determination of minimum cost paths". SIGART Bulletin, 37:28–29, December 1972.

[89] Fiona Harvey. Name that tune. Scientific American, 288(6):84–86, June 2003.

[90] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer, New York, NY, 2nd edition, 2009.

[91] César Hidalgo. Why Information Grows: The Evolution of Order, from Atoms to Economies. Basic Books, New York, NY, 2015.

[92] Theodore P. Hill. A statistical derivation of the Significant-Digit law. Statistical Science, 10(4):354–363, 1995.

[93] C. A. R. Hoare. Algorithm 63: Partition. Communications of the ACM, 4(7):321, July 1961.

[94] C. A. R. Hoare. Algorithm 64: Quicksort. Communications of the ACM, 4(7):321, July 1961.

[95] C. A. R. Hoare. Algorithm 65: Find. Communications of the ACM, 4(7):321–322, July 1961.

[96] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. Communications of the ACM, 16(6):372–378, June 1973.

[97] W. G. Horner. A new method of solving numerical equations of all orders, by continuous approximation. Philosophical Transactions of the Royal Society of London, 109:308–335, 1819.

[98] R. Nigel Horspool. Practical fast searching in strings. Software: Practice and Experience, 10(6):501–506, 1980.

[99] D. A. Huffman. A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9):1098–1101, September 1952.

[100] Earl B. Hunt, Janet Marin, and Philip J. Stone. Experiments in Induction. Academic Press, New York, NY, 1966.

[101] P. Z. Ingerman. Algorithm 141: Path matrix. Communications of the ACM, 5(11):556, November 1962.

[102] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An Introduction to Statistical Learning: With Applications in R. Springer, New York, NY, 2013.

[103] T. G. Jones. A note on sampling a tape-file. Communications of the ACM, 5(6):343, June 1962.

[104] David Kahn. The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet. Scribner, New York, NY, revised edition, 1996.

[105] Jonathan Katz and Yehuda Lindell. Introduction to Modern Cryptography. CRC Press, Taylor & Francis Group, Boca Raton, FL, 2nd edition, 2015.

[106] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. In Howard Heys and Carlisle Adams, editors, Selected Areas in Cryptography, volume 1758 of Lecture Notes in Computer Science, pages 13–33. Springer, Berlin, 2000.

[107] Jon Kleinberg and Éva Tardos. Algorithm Design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2005.

[108] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '98, pages 668–677, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[109] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. Journal of the ACM, 46(5):604–632, September 1999.

[110] Donald E. Knuth. Ancient babylonian algorithms. Communications of the ACM, 15(7):671–677, July 1972.

[111] Donald E. Knuth. The TEXbook. Addison-Wesley Professional, Reading, MA, 1986.

[112] Donald E. Knuth. The Art of Computer Programming, Volume 1: Fundamental Algorithms. Addison-Wesley, Reading, MA, 3rd edition, 1997.

[113] Donald E. Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[114] Donald E. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, Reading, MA, 2nd edition, 1998.

[115] Donald E. Knuth. The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1. Addison-Wesley, Upper Saddle River, NJ, 2011.

[116] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. SIAM Journal on Computing, 6(2):323–349, 1977.

[117] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. Software: Practice and Experience, 11:1119–1194, 1981.

[118] Alan G. Konheim. Hashing in Computer Science: Fifty Years of Slicing and Dicing. John Wiley & Sons, Inc., Hoboken, NJ, 2010.

[119] James F. Kurose and KeithW. Ross. Computer Networking: A Top-Down Approach. Pearson, Boston, MA, 6th edition, 2013.

[120] Leslie Lamport. LATEX: A Document Preparation System.Addison-Wesley Professional, Reading, MA, 2nd edition, 1994.

[121] Amy N. Langville and Carl D. Meyer. Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press, Princeton, NJ, 2006.

[122] David Lazer, Ryan Kennedy, Gary King, and Alessandro Vespignani. The parable of Google flu: Traps in big data analysis. Science, 343(6176):1203–1205, 2014.

[123] Thierry Lecroq. Experimental results on string matching algorithms. Software: Practice and Experience, 25(7):727–765, 1995.

[124] Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. Mathematics of Computation, 68(225):249–260, January 1999.

[125] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. ACM Transactions on Mathematical Software, 33(4), August 2007.

[126] C. Y. Lee. An algorithm for path connections and its applications. IRE Transactions on Electronic Computers, EC-10(3):346–365, September 1961.

[127] D. H. Lehmer. Mathematical methods in large-scale computing units. In Proceedings of the Second Symposium on Large-Scale Digital Calculating Machinery, pages 141–146, Cambridge, MA, 1949. Harvard University Press.

[128] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. ACM Computing Surveys, 19(3):261–296, September 1987.

[129] Anany Levitin. Introduction to the Design & Analysis of Algorithms. Pearson, Boston, MA, 3rd edition, 2012.

[130] John MacCormick. Nine Algorithms That Changed the Future: The Ingenious Ideas that Drive Today's Computers. Princeton University Press, Princeton, NJ, 2012.

[131] David J. C. MacKay. Information Theory, Inference, and Learning Algorithms. Cambridge University Press, Cambridge, UK, 2003.

[132] Charles E. Mackenzie. Coded Character Sets, History and Development. Addison-Wesley, Reading, MA, 1980.

[133] George Marsaglia. Xorshift rngs. Journal of Statistical Software, 8(14):1–6, 2003.

[134] Tomomi Matsui and Yasuko Matsui. A survey of algorithms for calculating power indices of weighted majority games. Journal of the Operations Research Society of Japan, 43:71–86, 2000.

[135] John McCabe. On serial files with relocatable records. Operations Research, 13(4):609–618, 1965.

[136] A. I. McLeod and D. R. Bellhouse. A convenient algorithm for drawing a simple random sample. Applied Statistics, 32(2):182–184, 1983.

[137] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[138] Ralph C. Merkle. A certified digital signature. In Proceedings on Advances in Cryptology, CRYPTO '89, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[139] Stanley Milgram. The small world problem. Psychology Today, 1(1):60–67, 1967.

[140] Gary L. Miller. Riemann's hypothesis and tests for primality. In Proceedings of Seventh Annual ACM Symposium on Theory of Computing, STOC '75, pages 234–239, New York, NY, USA, 1975. ACM.

[141] Thomas J. Misa and Philip L. Frana. An interview with Edsger W. Dijkstra. Communications of the ACM, 53(8):41–47, August 2010.

[142] Thomas M. Mitchell. Machine Learning. McGraw-Hill, Inc., New York, NY, 1997.

[143] Michael Mitzenmacher. A brief history of generative models for power law and lognormal distributions. Internet Mathematics, 1(2):226–251, 2004.

[144] Michael Mitzenmacher and Eli Upfal. Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge, UK, 2005.

[145] E. F. Moore. The shortest path through a maze. In Proceedings of an International Symposium on the Theory of Switching, 2–5 April 1957, pages 285–292. Harvard University Press, 1959.

[146] Robert Morris. Scatter storage techniques. Communications of the ACM, 11(1):38–44, 1968.

[147] J. Moy. OSPF version 2. RFC 2328, April 1998.

[148] Kevin P. Murphy. Machine Learning: A Probabilistic Perspective. The MIT Press, Cambridge, MA, 2012.

[149] Simon Newcomb. Note on the frequency of use of the different digits in natural numbers. American Journal of Mathematics, 4(1):39–40, 1881.

[150] Mark Newman. Networks: An Introduction. Oxford University Press, Inc., New York, NY, USA, 2010.

[151] Michael A. Nielsen and Isaac L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, Cambridge, UK, 2000.

[152] Cathy O'Neil.Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy. Crown, New York, NY, 2016.

[153] Christof Paar and Jan Pelzl. Understanding Cryptography: A Textbook for Students and Practitioners. Springer-Verlag, Berlin, 2009.

[154] Vilfredo Pareto. Cours d' Économie Politique. Rouge, Lausanne, 1897.

[155] Richard E. Pattis. Textbook errors in binary searching. SIGCSE Bulletin, 20(1):190–194, February 1988.

[156] L. S. Penrose. The elementary statistics of majority voting. Journal of the Royal Statistical Society, 109(1):53–57, 1946.

[157] Radia Perlman. Interconnections: Bridges, Routers, Switches, and Internetworking Protocols. Addison-Wesley, 2nd edition, 1999.

[158] J. R. Quinlan. Discovering rules by induction from large collections of examples. In D. Michie, editor, Expert systems in the micro electronic age. Edinburgh University Press, Edinburgh, UK, 1979.

[159] J. R. Quinlan. Semi-autonomous acquisition of pattern-based knowledge. In J. E. Hayes, D. Michie, and Y.-H. Pao, editors, Machine Intelligence, volume 10. Ellis Horwood, Chichester, UK, 1982.

[160] J. R. Quinlan. Induction of decision trees. Machine Learning, 1(1):81–106, 1986.

[161] J. Ross Quinlan. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1993.

[162] Michael O. Rabin. Probabilistic algorithm for testing primality. Journal of Number Theory, 12(1):128–138, 1980.

[163] Rajeev Raman. Recent results on the single-source shortest paths problem. SIGACT News, 28(2):81–87, June 1997.

[164] Edward M. Reingold, Kenneth J. Urban, and David Gries. K-M-P string matching revisited. Information Processing Letters, 64(5):217–223, December 1997.

[165] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2):120–126, February 1978.

[166] Ronald Rivest. On self-organizing sequential search heuristics. Communications of the ACM, 19(2):63–67, February 1976.

[167] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resis-

tance, second-preimage resistance, and collision resistance. In Bimal Roy and Willi Meier, editors, Fast Software Encryption, volume 3017 of Lecture Notes in Computer Science, pages 371–388. Springer Berlin Heidelberg, 2004.

[168] Bernard Roy. Transitivé et connexité. Comptes rendus des séances de l' Académie des Sciences, 249(6):216–218, 1959.

[169] Donald G. Saari. Disposing Dictators, Demystifying Voting Paradoxes. Cambridge University Press, Cambridge, UK, 2008.

[170] David Salomon. A Concise Introduction to Data Compression. Springer, London, UK, 2008.

[171] David Salomon and Giovanni Motta. Handbook of Data Compression. Springer, London, UK, 5th edition, 2010.

[172] Khalid Sayood. Introduction to Data Compression. Morgan Kaufmann, Waltham, MA, 4th edition, 2012.

[173] Douglas C. Schmidt. GPERF: A perfect hash function generator. In Robert C. Martin, editor, More C++ Gems, pages 461–491. Cambridge University Press, New York, NY, USA, 2000.

[174] Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1995. /Брюс Шайнер, «Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си», Триумф, 2002г.

[175] Markus Schulze. A new monotonic, clone-independent, reversal symmetric, and Condorcet-consistent single-winner election method. Social Choice andWelfare, 36(2):267– 303, 2011.

[176] Robert Sedgewick. Algorithms in C—Parts 1–4: Fundamentals, Data Structures, Sorting, Searching. Addison-Wesley, Boston, MA, 3rd edition, 1998.

[177] Robert Sedgewick. Algorithms in C++—Parts 1–4: Fundamentals, Data Structures, Sorting, Searching. Addison-Wesley, Boston, MA, 3rd edition, 1998.

[178] Robert Sedgewick. Algorithms in C—Part 5: Graph Algorithms. Addison-Wesley, Boston, MA, 3rd edition, 2002.

[179] Robert Sedgewick. Algorithms in C—Part 5: Graph Algorithms. Addison-Wesley, Boston, MA, 3rd edition, 2002.

[180] Robert Sedgewick and KevinWayne. Algorithms. Addison-Wesley, Upper Saddle River, NJ, 4th edition, 2011.

[181] C. E. Shannon. A mathematical theory of communication. The Bell System Technical Journal, 27(3):379–423, July 1948.

[182] C. E. Shannon. Prediction and entropy of printed english. The Bell System Technical Journal, 30(1):50–64, January 1950.

[183] Claude E. Shannon andWarrenWeaver. The Mathematical Theory of Communication. University of Illinois Press, Urbana, IL, 1949.

[184] L. S. Shapley and Martin Shubik. A method for evaluating the distribution of power in a committee system. American Political Science Review, 48:787–792, September 1954.

[185] PeterW. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing, 26(5):1484–1509, October 1997.

[186] Joseph H. Silverman. A Friendly Introduction to Number Theory. Pearson, 4th edition, 2012.

[187] Simon Singh. The Code Book: The Secret History of Codes and Codebreaking. Fourth Estate, London, UK, 2002.

[188] Steven S. Skiena. The Algorithm Design Manual. Springer-Verlag, London, UK, 2nd edition, 2008.

[189] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. Communications of the ACM, 28(2):202–208, February 1985.

[190] David Eugene Smith, editor. A Source Book in Mathematics. McGraw-Hill Book Co., New York, NY, 1929. Reprinted by Dover Publications in 1959.

[191] Jonathan Sorenson. An introduction to prime number sieves. Computer Sciences Technical Report 909, Department of Computer Science, University of Wisconsin-Madison, January 1990.

[192] Gary Stix. Profile: David Huffman. Scientific American, 265(3):54–58, September 1991.

[193] James V Stone. Information Theory: A Tutorial Introduction. Sebtel Press, Sheffield, UK, 2015.

[194] Michael P. H. Stumpf and Mason A. Porter. Critical truths about power laws. Science, 335(6069):665–666, 2012.

[195] George G. Szpiro. Numbers Rule: The Vexing Mathematics of Democracy, from Plato to the Present. Princeton University Press, Princeton, NJ, 2010.

[196] Andrew S. Tanenbaum and David J.Wetherall. Computer Networks. Prentice Hall, Boston, MA, 5th edition, 2011.

[197] Robert Tarjan. Depth-first searcn and linear graph algorithms. SIAM Journal on Computing, 1(2):146–160, 1972.

[198] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. Acta Informatica, 6(2):171–185, 1976.

[199] Robert Endre Tarjan. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[200] Alan D. Taylor and Allison M. Pacelli. Mathematics and Politics: Strategy, Voting, Power and Proof. Springer, 2nd edition, 2008.

[201] Mikkel Thorup. On RAM priority queues. SIAM Journal on Computing, 30(1):86–109, April 2000.

[202] Yves Tillé. Sampling Algorithms. Springer, New York, NY, 2006.

[203] Thanassis Tiropanis, Wendy Hall, Jon Crowcroft, Noshir Contractor, and Leandros Tassiulas. Network science, web science, and internet science. Communications of the ACM,
58(8):76–82, July 2015.

[204] Jeffrey Travers and Stanley Milgram. An experimental study of the small world problem. Sociometry, 32(4):425–443, 1969.

[205] Alan Turing. Proposed electronic calculator. Technical report, National Physical Laboratory (NPL), UK, 1946. http://www.alanturing.net/ace/index.html.

[206] United States National Institute of Standards and Technology (NIST). Announcing the ADVANCED ENCRYPTION STANDARD (AES), November 26 2001. Federal Information Processing Standards Publication 197.

[207] United States National Institute of Standards and Technology (NIST). Secure hash standard (SHS), August 2015. Federal Information Processing Standards Publication 180-4.

[208] United States National Institute of Standards and Technology (NIST). SHA-3 standard: Permutation-based hash and extendable-output functions, August 2015. Federal Information Processing Standards Publication 202.

[209] Sebastiano Vigna. An experimental exploration of Marsaglia's xorshift generators, scrambled. CoRR, abs/1402.6246, 2014.

[210] Jeffrey S. Vitter. Random sampling with a reservoir. ACM Transactions on Mathematical Software, 11(1):37–57, March 1985.

[211] John von Neumann. Various techniques used in connection with random digit. In A.S. Householder, G. E. Forsythe, and H. H. Germond, editors, Monte Carlo Method, volume 12 of National Bureau of Standards Applied Mathematics Series, pages 36–38. U.S. Government Printing Office, Washington, D.C., 1951.

[212] Avery Li-ChunWang. An industrial-strength audio search algorithm. In Proceedings of the 4th International Conference on Music Information Retrieval (ISMIR 2003), Baltimore, MD, October 26–30 2003.

[213] StephenWarshall. A theorem on boolean matrices. Journal of the ACM, 9(1):11–12, January 1962.

[214] Duncan J. Watts. Six Degrees: The Science of a Connected Age. W. W. Norton & Company, New York, NY, 2004.

[215] T. A. Welch. A technique for high-performance data compression. Computer, 17(6):8–19, June 1984.

[216] Frank Wilczek. A Beautiful Question: Finding Nature's Deep Design. Penguin Press, New York, NY, 2015.

[217] Maurice V. Wilkes. Memoirs of a Computer Pioneer. The MIT Press, Cambridge, MA, 1985.

[218] J.W. J. Williams. Algorithm 232: Heapsort. Communications of the ACM, 7(6):347–348, June 1964.

[219] Ian H. Witten, Eibe Frank, and Mark A. Hall. Data Mining: Practical Machine Learning Tools and Techniques. Elsevier, Cambridge, 2016.

[220] XindongWu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. Knowledge and Information Systems, 14(1):1–37, January 2008.

[221] J. Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. Quarterly of Applied Mathematics, 27:526–530, 1970.

[222] Joel Young, Kristina Foster, Simson Garfinkel, and Kevin Fairbanks. Distinct sector hashes for target file detection. Computer, 45(12):28–35, December 2012.

[223] G. Udny Yule. A mathematical theory of evolution, based on the conclusions of Dr. J. C. Willis, F.R.S. Philosophical Transactions of the Royal Society of London: Series B, 213:21–87, April 1925.

[224] Philip Zimmermann. Why I wrote PGP. Part of the Original 1991 PGP User's Guide (updated), 1999. Available at https://www.philzimmermann.com/EN/essays/ WhyIWrotePGP.html.

[225] Philip Zimmermann. Phil Zimmermann on the importance of online privacy. The Guardian Tech Weekly Podcast, 2013. Available at http://www.theguardian.com/
technology/audio/2013/may/23/podcast-tech-weekly-phil-zimmerman.

[226] George Kingsley Zipf. The Psycho-Biology of Language: An Introduction to Dynamic Philology. Houghton Mifflin, Boston, MA, 1935.

[227] George Kingsley Zipf. Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. Addison-Wesley, Reading, MA, 1949.

[228] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. Information Theory, IEEE Transactions on, 23(3):337–343, May 1977.

[229] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. Information Theory, IEEE Transactions on, 24(5):530–536, September 1978.

Предметный указатель

Б

Е

З

И

К

Л

М

Н

О

П

Р

С

Ч

Ш

Э

Ю

ВВОДНЫЙ КУРС АЛГОРИТМОВ ДЛЯ НАЧИНАЮЩИХ И ПРОФЕССИОНАЛОВ

В ЭТОЙ КНИГЕ ВЫ НАЙДЕТЕ:

- **СТЕКИ И ГРАФЫ**
- **АЛГОРИТМЫ СЖАТИЯ**
- **АЛГОРИТМЫ ШИФРОВАНИЯ**
- **СОРТИРОВКУ И ПОИСК КРАТЧАЙШЕГО ПУТИ**
- **СИСТЕМУ ГОЛОСОВАНИЯ**

И МНОГОЕ ДРУГОЕ

ПАНОС ЛУРИДАС, ПРОФЕССОР УНИВЕРСИТЕТА ЭКОНОМИКИ И БИЗНЕСА, АВТОР ИЗДАТЕЛЬСТВА MIT PRESS. В ДОСТУПНОЙ ФОРМЕ АВТОР РАССКАЗЫВАЕТ ОБО ВСЕХ ФУНДАМЕНТАЛЬНЫХ АЛГОРИТМАХ И МЕТОДАХ, ПРИМЕНЯЕМЫХ ДЛЯ СЖАТИЯ, ШИФРОВАНИЯ, СОРТИРОВКИ И РЯДА ДРУГИХ ПОВСЕДНЕВНЫХ ПРОЦЕССОВ. ДАННОЕ ПОСОБИЕ ЯВЛЯЕТСЯ ЛУЧШИМ ВЫБОРОМ ДЛЯ ЧИТАТЕЛЕЙ, ЖЕЛАЮЩИХ ПОЛУЧИТЬ БАЗОВЫЕ ЗНАНИЯ ПО ТЕМЕ И НАЧАТЬ ПРИМЕНЯТЬ ИХ КАК В РАБОТЕ, ТАК И В ДАЛЬНЕЙШЕМ ОБУЧЕНИИ.

В ОСНОВЕ ХОРОШЕЙ ПРОГРАММЫ ВСЕГДА ЛЕЖИТ ЭФФЕКТИВНЫЙ И КРАСИВЫЙ АЛГОРИТМ. СТРОИТЕЛЬСТВО ДОМА НАЧИНАЕТСЯ С ЗАЛИВКИ ФУНДАМЕНТА, НАПИСАНИЕ ПРОГРАММЫ — С РАЗРАБОТКИ АЛГОРИТМА. АЛГОРИТМ — ВОТ АЛЬФА И ОМЕГА УСПЕШНОГО ПРОГРАММИРОВАНИЯ. ЭТА КНИГА — ПУТЕВОДИТЕЛЬ ПО СЛОЖНОМУ И ИНТЕРЕСНОМУ МИРУ АЛГОРИТМОВ, ОТЛИЧНЫЙ ВЫБОР ДЛЯ ТЕХ, КТО ХОЧЕТ ОТКРЫТЬ ДЛЯ СЕБЯ НОВЫЕ ПРОФЕССИОНАЛЬНЫЕ ГОРИЗОНТЫ.

Васильев А.Н.,
доктор физико-математических наук, профессор, автор книг по программированию и математическому моделированию

БОМБОРА

Бомбора — это новое название Эксмо Non-fiction, лидера на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bomborabooks
www.bombora.ru