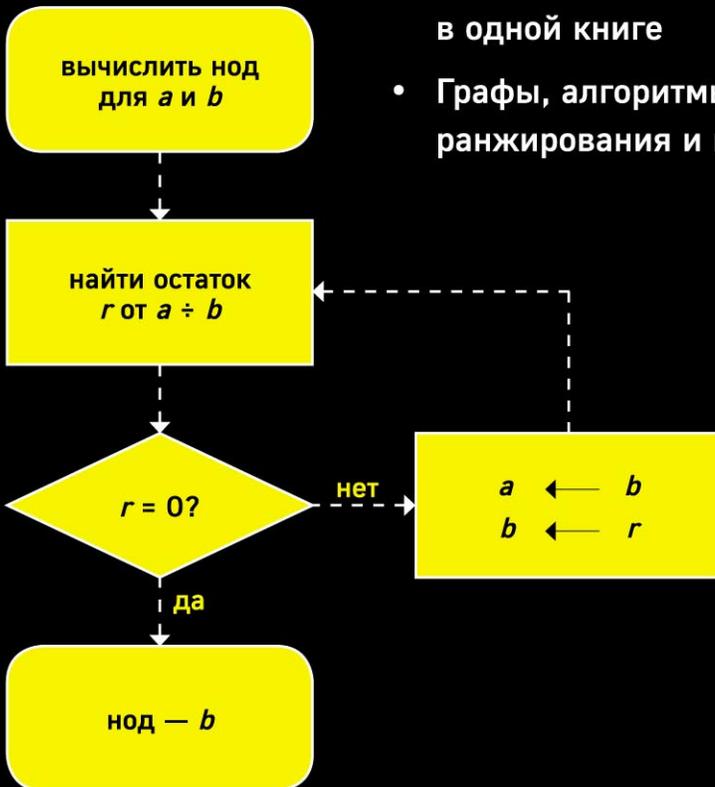


ПАНОС ЛУРИДАС

АЛГОРИТМЫ

Самый краткий и понятный курс

- Доступное изложение
- Вся основная информация в одной книге
- Графы, алгоритмы поиска, ранжирования и многое другое



БИБЛИОТЕКА

MIT

PANOS LOURIDAS

ALGORITHMS

ПАНОС ЛУРИДАС

АЛГОРИТМЫ

Самый краткий и понятный курс

- Доступное изложение
- Вся основная информация в одной книге
- Графы, алгоритмы поиска, ранжирования и многое другое

УДК 510.5
ББК 22.12
Л86

Algorithms
Panos Louridas

© 2020 Massachusetts Institute of Technology
The rights to the Russian-language edition obtained
through Alexander Korzhenevski Agency (Moscow)

Луридас, Панос.
Л86 Алгоритмы. Самый краткий и понятный курс / Панос Луридас ;
[перевод с английского М. А. Райтмана]. — Москва : Эксмо, 2022. —
192 с. — (Библиотека MIT).

ISBN 978-5-04-115765-4

Если вам нужно разобраться в том, что из себя представляют алгоритмы и графы, как они работают и какими бывают, эта книга для вас. Ее автор, Панос Луридас, уже много лет использует алгоритмы при проектировании программного обеспечения, криптографии, машинном обучении и является научным сотрудником Афинского университета экономики и бизнеса. Очень доступным даже для новичков языком он знакомит читателей с концепцией алгоритмов и принципами их работы — для чтения книги достаточно базового школьного образования.

УДК 510.5
ББК 22.12

ISBN 978-5-04-115765-4

© Райтман М.А., перевод на русский язык, 2022
© Оформление. ООО «Издательство «Эксмо», 2022

Мир нестабилен, но его нельзя назвать непостижимым. Главное, не забывать простое правило: все, что он выражает в виде бесчисленных жизней и существ, всегда заканчивается знаками восклицания и не носит вопросительного характера.

Карл Уве Кнаусгор. «Лето»

ОГЛАВЛЕНИЕ

Предисловие	7
Благодарности	12
Глава 1. Что такое алгоритм?	13
Глава 2. Графы	38
Глава 3. Поиск	59
Глава 4. Сортировка	74
Глава 5. PageRank	97
Глава 6. Глубокое обучение	120
Послесловие	150
Глоссарий	158
Примечания	177
Предметный указатель	183
Библиография	185
Для дальнейшего чтения	190
Об авторе	191

ПРЕДИСЛОВИЕ

Я знаком с двумя подростками, которые обладают знаниями, невообразимыми для любых ученых, философов и исследователей прошлых веков. Это мои сыновья. Нет, я не из тех, кто любит похвастаться экстраординарными талантами своих детей. Но у этих двух парней есть карманные устройства, дающие им доступ к самому обширному хранилищу информации из когда-либо созданных. Теперь, когда они овладели искусством поиска по Интернету, нет такого фактологического вопроса, на который они не в состоянии ответить. Они переводят тексты с любого иностранного языка, не листая огромные словари (к слову, мы их не выбрасываем, чтобы наши дети знали, как все обстояло лишь несколько лет назад). Они мгновенно получают новости со всех уголков мира. Они общаются со своими сверстниками так, что вы даже не замечаете, независимо от того, какое расстояние их разделяет. Они детально планируют свои прогулки. С другой стороны, они могут тратить время на видеоигры или следовать модным тенденциям, которые меняются настолько быстро, что я даже не знаю, почему они так важны.

Все перечисленное выше стало возможным благодаря огромному прогрессу цифровых технологий. Устройства, которые мы сегодня носим в наших карманах, обладают большей вычислительной мощностью, чем те, с чьей помощью люди добрались до Луны. Как показывает пример с этими двумя подростками, наша жизнь претерпела огромные изменения; предсказания о будущем варьируются от утопий, в которых людям больше не придется работать, до антиутопий, где горстка счастливиц сможет наслаждаться полноценной жизнью, тогда как остальные окажутся обреченными на безысходность. К счастью, мы сами творцы своего будущего, и важную роль в этом играет то, насколько хорошо мы владеем технологиями, лежащими в основе потенциальных свершений и изменений. Мы живем в лучший период человеческой истории, хотя в суете повседневной жизни об этом можно забыть. Мы здоровее, чем когда-либо, и средняя продолжительность нашей жизни больше, чем у любого предыдущего поколения. Несмотря на несправедливость вопиющего неравенства, огромная часть человечества освободилась из оков нищеты. Мы еще никогда не были так близко друг к другу — как в буквальном, так и в виртуальном смысле. Можно сетовать на коммерциализацию массового глобального туризма, но дешевые путешествия позволяют нам знакомиться с другими культурами и посещать места, которыми раньше

мы любовались только в иллюстрированных журналах. Весь этот прогресс может и должен продолжаться.

Но, чтобы сделать свой вклад в этот прогресс, недостаточно просто использовать цифровые технологии. В них нужно разбираться. Во-первых, по сугубо практической причине: это открывает отличные карьерные возможности. Во-вторых, даже если вы не планируете работать в сфере технологий, вам нужно понимать их основополагающие принципы; это позволит как следует оценить их потенциал и поучаствовать в их развитии. Наряду с оборудованием (компонентами, из которых состоят компьютеры и цифровые устройства) не менее важной частью цифровых технологий является программное обеспечение — приложения, позволяющие выполнять на этом оборудовании ту или иную задачу. В основе приложений лежат реализуемые ими алгоритмы — наборы инструкций, которые описывают способы решения этих задач (не самое точное определение алгоритма, но не волнуйтесь: у нас есть целая книга, чтобы его уточнить). Без алгоритмов компьютеры были бы бесполезными, а все современные технологии попросту не существовали бы.

Необходимые знания меняются со временем. На протяжении значительной части истории человечества образование считалось необязательным. Большинство людей были неграмотными, но даже если их чему-то обучали, то только практическим навыкам и религиозным текстам. В начале девятнадцатого века более 80% мирового населения никогда не посещало школу; теперь же большинство людей обучались в ней хотя бы несколько лет. Ожидается, что к концу этого века число необразованных людей в мире упадет до нуля. Время, которое мы тратим на обучение, тоже выросло. Если в 1940 году менее 5% американцев имели степень магистра, то в 2015-м этот показатель почти достиг трети.

В девятнадцатом веке ни в одной школе не преподавали молекулярную биологию, так как о ней еще никто не знал; ДНК открыли лишь ближе к середине двадцатого века. И теперь это часть любой учебной программы. Нечто похожее можно сказать об алгоритмах: их придумали еще в древние времена, но до изобретения современных компьютеров ими интересовались лишь немногие. Автор этой книги убежден, что мы достигли уровня, на котором алгоритмы являются одним из ключевых аспектов того, что мы считаем базовыми знаниями. Тот, кто не знает, что они представляют собой и как работают, не может оценить их потенциал и понять, как они влияют на нашу жизнь, чего от них следует ожидать, какие у них ограничения и что требуется для их работы. Поскольку наше общество все больше полагается на алгоритмы, нам, как осведомленным гражданам, следует в них ориентироваться.

Наряду с оборудованием (компонентами, из которых состоят компьютеры и цифровые устройства) не менее важной частью цифровых технологий является программное обеспечение — приложения, позволяющие выполнять на этом оборудовании ту или иную задачу. В основе приложений лежат реализуемые ими алгоритмы.

Изучение алгоритмов может иметь и другие преимущества. Если математика знакомит нас с формальным видом мышления, знание алгоритмов позволяет рассуждать о вещах новым, алгоритмическим образом, направленным на практическое решение задач. Благодаря этому эффективные реализации алгоритмов в виде программ могут быстро выполняться на компьютерах. Подход, в котором акцент делается на практичных и эффективных процессах проектирования, может пригодиться не только профессиональным программистам.

Эта книга призвана познакомить с алгоритмами читателя, далекого от компьютерных наук, и объяснить ему, как они работают. Мы не стремимся показать, как алгоритмы влияют на наши жизни, — с этим отлично справляются другие книги, в которых речь идет о том, как прогресс в обработке больших данных, искусственный интеллект и повсеместное внедрение вычислительных устройств могут изменить человеческую природу. Нас в первую очередь интересует не то, что может произойти в будущем, а, скорее, как это может случиться. Для этого мы представим здесь настоящие алгоритмы и покажем не только то, что они делают, но и принцип их работы. Вместо общих фраз вас ждут подробные объяснения.

Знание алгоритмов позволяет рассуждать о вещах новым, алгоритмическим образом, направленным на практическое решение задач. Благодаря этому эффективные реализации алгоритмов в виде программ могут быстро выполняться на компьютерах.

На вопрос «что такое алгоритм?» есть удивительно простой ответ. Это определенный способ решения задачи, который можно описать в виде простых шагов и затем выполнить на компьютере с поразительной скоростью и эффективностью. Однако в таких решениях нет ничего волшебного. Тот факт, что они состоят из элементарных шагов, означает, что в них способны разобраться большинство людей.

На самом деле для чтения этой книги достаточно знаний, полученных в средней школе. Конечно, в некоторых главах есть немного математики, так как невозможно рассуждать об алгоритмах вообще без формул и формальных обозначений. В тексте объясняются любые концепции, распространенные в мире алгоритмов, но малоизвестные за пределами компьютерных наук.

Вот что написал недавно скончавшийся физик во введении в свою самую популярную книгу, «Краткая история времени», изданную в 1988 году: «Мне сказали, что каждая включенная в книгу формула уменьшит число покупателей вдвое». Это звучит довольно угрожающе для настоящей книги, так как математика в ней встречается не раз. Но я решил не обращать на это внимание по двум причинам. Во-первых, для понимания физики Хокинга требуется знание математики университетского уровня; здесь все намного доступнее. Во-вторых, эта книга призвана показать не только назначение алгоритмов, но и то, как они работают внутри, поэтому читателю должны быть знакомы некоторые термины, используемые здесь, в том числе математические. Система обозначений не является прерогативой технической интеллигенции, и владение ею поможет развеять ореол загадочности, окружающий данную тему; в конце вы увидите, что это в основном сводится к умению рассуждать о вещах четко и поддающимся измерению способом.

Книга вроде этой не может полностью охватить тему алгоритмов, но мы попытаемся провести краткий обзор и познакомить читателя с алгоритмическим образом мышления. Основа будет заложена в первой главе, из нее вы узнаете, что такое алгоритмы и как измерить их эффективность. Сразу отметим, что алгоритм — это конечная последовательность шагов, которые можно выполнить с помощью ручки и листа бумаги, и это простое определение не так уж далеко от истины. Дальше в главе 1 будет исследована взаимосвязь между алгоритмами и математикой. Отличаются же они практичностью: при написании алгоритмов нас интересуют прикладные способы решения задач. Это означает, что нам нужно как-то оценивать практичность и эффективность наших алгоритмов. Вы увидите, что эти аспекты можно рассматривать в строго очерченных рамках вычислительной сложности; именно в этом контексте будет проходить обсуждение алгоритмов на страницах данной книги.

Следующие три главы посвящены трем наиболее важным сферам применения алгоритмов. В главе 2 рассматриваются алгоритмы для решения задач, связанных с сетями, известными как графы. В число этих задач входит поиск маршрута на дороге или последовательности звеньев, соединяющих вас с другими людьми в социальной сети. Сюда же отнесем задачи из других областей с не такими очевидными связями: секвенирование ДНК и планирование соревнований; это проиллюстрирует тот факт, что разного рода задачи могут быть эффективно решены с помощью одних и тех же инструментов.

В главах 3 и 4 вы узнаете, как искать и упорядочивать элементы. Тема может показаться прозаической, но она одна из самых важных с точки зрения

практического применения компьютеров. Компьютеры тратят много времени на сортировку и поиск, однако мы редко обращаем на это внимание, поскольку указанные операции — неотъемлемая и невидимая часть большинства приложений. Сортировка и поиск даже дают некоторое представление о важном аспекте алгоритмов. Многие задачи можно решать разными способами. Выбор подходящего алгоритма зависит от его характеристик; некоторые алгоритмы больше подходят для определенных задач, чем другие. Поэтому вам полезно увидеть, как разные алгоритмы с разными характеристиками справляются с решением одной и той же задачи.

В следующих двух главах представлены важные способы применения алгоритмов в широких масштабах. В главе 5 мы снова вернемся к графам, чтобы объяснить алгоритм, который можно применять для ранжирования веб-страниц в порядке их значимости. PageRank использовался компанией Google в момент ее основания. Успех этого алгоритма в ранжировании результатов поиска сыграл ключевую роль в феноменальном успехе Google как компании. К счастью, разобраться в его работе не так уж сложно. Это позволит увидеть, как алгоритмы справляются с задачей, которую, на первый взгляд, нельзя решить с помощью компьютеров: как оценить важность тех или иных вещей?

В главе 6 будет представлена одна из самых динамично развивающихся областей компьютерных наук: нейронные сети и глубокое обучение. Об успешном применении нейронных сетей можно узнать даже из популярных средств массовой информации. Наибольший интерес проявляется к сюжетам о системах, которые выполняют такие задачи, как анализ изображений, автоматический перевод и медицинская диагностика. Мы начнем с отдельных нейронов и постепенно перейдем к построению все больших и больших нейронных сетей, способных выполнять все более сложные задачи. Вы увидите, что все они основаны на общих фундаментальных принципах. Их эффективность объясняется взаимосвязанностью множества простых компонентов и применением алгоритма, который позволяет нейронным сетям учиться.

Вслед за демонстрацией возможностей алгоритмов идет послесловие, посвященное тому, как ограничены компьютерные вычисления. Мы знаем, что компьютеры способны на многое, и ожидаем от них еще большего, но есть вещи, которые им не под силу. Обсуждение пределов их возможностей позволит более точно объяснить природу алгоритмов и вычислений. Мы уже говорили, что алгоритм можно описать в виде конечной последовательности шагов, которые можно выполнить с помощью ручки и листа бумаги, но какими могут быть эти шаги? И насколько близка аналогия с ручкой и бумагой к настоящим алгоритмам?

БЛАГОДАРНОСТИ

Прежде всего, благодарю Мари Лафкин Ли из издательства MIT Press — ей принадлежит идея создания этой книги; Стефани Коэн, которая мягко подгоняла меня в процессе написания; Синди Милстрейн за ее тщательную редактуру и Вирджинию Кроссмэн за превосходное внимание к деталям и заботу обо всех аспектах этой книги. Книга об алгоритмах должна войти в цикл *Essential Knowledge* (необходимые знания), и я горжусь тем, что являюсь ее автором.

Я признателен Диомидису Спинеллису за его комментарии к разным частям книги. Отдельное спасибо Константиносу Маринакосу, который прочитал мой черновик, нашел ошибки, за которые мне теперь стыдно, и подсказал, как их исправить.

Наконец, моя огромная благодарность двум подросткам, Адрианосу и Эктору, чьи жизни в какой-то степени будут определяться темой данной книги, и их матери, Элени; эта работа проделана благодаря им.

ЧТО ТАКОЕ АЛГОРИТМ?

Алгоритмическая эра

Мы любим давать названия временным периодам — наверное, потому, что это позволяет нам совладать со скоротечностью времени. Поэтому мы начали говорить о настоящем как о новой *алгоритмической эре*, в которой алгоритмам отводится все более важная роль. Интересно, что речь больше не идет об *эпохе компьютеров* или *Интернета*. Их мы воспринимаем как само собой разумеющееся. Но именно алгоритмы дают нам ощущение качественного сдвига. «Вот он, всемогущий алгоритм, отрывок компьютерного кода, претендующий на высшую власть в нашу светскую эпоху, своего рода бог», — пишет Кристофер Лайдон, бывший журналист в *New York Time*, ведущий *Radio Open Source*. И в самом деле, алгоритмы воспринимают как некую вышестоящую инстанцию, которая участвует в организации политических кампаний, отслеживает нашу активность онлайн, помогает нам с покупками, показывает рекламу, подсказывает, с кем пойти на свидание, следит за нашим здоровьем.

Вокруг всего этого существует аура таинственности, которая, наверное, льстит служителям алгоритмов. Должность «программиста» или «специалиста в области информационных технологий» делает вас достойным человеком, хоть и технарем. Приятно принадлежать к племени, стоящему в шаге от того, чтобы перевернуть все с ног на голову, не так ли?

Алгоритмам, несомненно, свойственно нечто божественное. Как и боги, они в основном выходят сухими из воды; события случаются не из-за человеческой деятельности, а потому, что так решил алгоритм и с алгоритма спроса нет. Компьютеры, которые выполняют алгоритмы, превосходят человека во все большем количестве отраслей, поэтому кажется, что наша роль

уменьшается день ото дня; кто-то верит, что не за горами день, когда компьютеры превзойдут нас во всех аспектах умственной деятельности.

Но, если посмотреть на вещи с другой стороны, алгоритмы совсем не похожи на богов, и мы часто об этом забываем. Результаты, которые производят алгоритмы, не являются откровением свыше. Мы в точности знаем, каким правилам они подчиняются и какие шаги они выполняют. Каким бы чудесным ни был результат, его всегда можно свести к элементарным операциям. Люди, для которых алгоритмы в новинку, могут удивиться тому, насколько тривиальными они бывают внутри. Это вовсе не умаляет их значимость; просто понимание того, что происходит за кулисами, снимает налет таинственности. Но в то же время это позволяет по достоинству оценить, насколько элегантно написан алгоритм.

Идея этой книги в том, что в алгоритмах и в самом деле нет ничего таинственного. Это инструменты, позволяющие нам легко справляться с определенными задачами; они заточены на решение проблем. В этом смысле их можно считать «умственными» инструментами, включающими числа и арифметику. У нас ушли тысячелетия на выработку системы счисления, с помощью которой дети в школе могут решать математические задачи; без нее эти задачи остались бы нерешенными. Мы считаем этот навык само собой разумеющимся, но несколько поколений назад им владела лишь небольшая часть людей.

Точно так же знание алгоритмов не должно быть прерогативой крошечного элитного меньшинства; это умственные инструменты, которыми может овладеть кто угодно, а не только специалисты по компьютерам. Более того, этот навык *должен* быть доступен большому числу людей, так как он позволяет взглянуть на алгоритмы в более широком контексте: понять, что они делают, как они это делают и чего от них можно ожидать.

Основная цель этой книги — дать базовое понимание алгоритмов, которое позволит принять полноценное участие в дискуссиях, происходящих в алгоритмическую эру. Начало этой эре мы положили сами, используя ранее разработанные инструменты. Изучение этих инструментов и будет темой данной книги. Алгоритмы прекрасны, и представление о том, как они создаются и работают, может улучшить наш собственный образ мышления.

Но для начала развеим распространенное заблуждение о том, что алгоритмы неотделимы от компьютеров. Это так же неверно, как связывать числа с калькуляторами.

Способ решения задач

Головоломки, музыка, делимость чисел и ускорители нейтронов в физике элементарных частиц — вы увидите, что у всего этого есть общий алгоритм, который применяется в разных предметных областях, но при этом основан на тех же фундаментальных принципах. Как так?

Само слово «алгоритм» не раскрывает своего значения в полной мере. Оно происходит от имени Мухаммада ибн Мусы аль-Хорезми (ок. 783 — ок. 850), персидского ученого, который работал в областях математики, астрономии и географии. Аль-Хорезми внес большой вклад во многие сферы науки. Термин «алгебра» происходит от арабского названия его самой фундаментальной работы, «Краткая книга о восполнении и противопоставлении». Его вторая по известности работа, «Книга об индийском счете», была посвящена арифметике и переведена на латынь; именно из нее европейцы узнали об индо-арабской системе счисления. Имя Аль-Хорезми в латинском варианте выглядит как *Algorismus*; его стали использовать для обозначения численных методов с десятичными числами. Впоследствии термин *Algorismus* под влиянием греческого слова *arithmos* (ἀριθμός — число) превратился в *algorithm* (алгоритм) и все так же обозначал десятичную арифметику, пока в девятнадцатом веке не обзавелся современным значением.

Многие думают, что алгоритмы имеют какое-то отношение к компьютерам, но это не так. Они существовали задолго до появления компьютеров. Первый алгоритм, о котором нам известно, существовал еще во времена древнего Вавилона. Кроме того, алгоритмы решают не компьютерные проблемы. Они описывают, что и как нужно делать, в виде последовательности шагов. Звучит немного туманно. Вы можете спросить, о каких шагах идет речь? Мы могли бы внести ясность и дать четкое математическое определение самого алгоритма и того, что он делает (оно действительно существует), но это было бы излишним. Вам, скорее всего, будет достаточно знать, что алгоритм — это набор шагов, которые можно выполнить с помощью ручки и листа бумаги; несмотря на внешнюю простоту, это определение достаточно близко к тем, которые используют математики и специалисты в области информатики.

Начнем наше знакомство с алгоритмами с задачи, которую можно решить вручную. Представьте, что у нас есть два множества объектов и мы хотим как можно равномернее распределить объекты одного множества среди объектов другого. Объекты в первом множестве будут обозначаться как \times , а во втором — как \bullet . Мы хотим распределить крестики между точками.

Многие думают, что алгоритмы имеют какое-то отношение к компьютерам, но это не так. Они существовали задолго до появления компьютеров.

Если количество крестиков точно делится на количество точек, все просто: крестики между точками достаточно разместить так, как будто мы выполняем деление. Например, если всего у нас 12 объектов, из которых три — это крестики, а остальные девять — точки, мы рисуем один крестик, затем три точки, затем один крестик, затем три точки и, наконец, еще один крестик и три точки.

× • • • × • • • × • • •

Но что, если общее количество объектов нельзя точно разделить на число крестиков? Что, если крестиков у нас пять, а точек семь?

Вначале мы размещаем в один ряд все крестики и затем все точки, как показано ниже:

× × × × × • • • • • • •

После этого помещаем пять точек снизу от крестиков:

× × × × × • •
• • • • •

В результате справа остаются два столбца с одной точкой в каждом. Поместим их под двумя первыми столбцами, образуя третью строку:

× × × × ×
• • • • •
• •

Теперь можно заметить три оставшихся столбца. Два из них (крайние справа) помещаем под двумя первыми столбцами:

× × ×
• • •
• •
× ×
• •

Остается всего один лишний столбец, поэтому мы останавливаемся. Объединим столбцы, перемещаясь слева направо, и получим:

× • • × • × • • × • × •

Это наш результат. Мы распределили крестики между точками. Не так равномерно, как раньше, но это закономерно, ведь, как вы помните, пять

не делится ровно на 12. Тем не менее мы избежали нагромождения крестиков и создали последовательность, которая не выглядит совсем уж бессистемной.

Вам, наверное, интересно, есть ли у этой последовательности какая-то особенность; попробуйте подставить DUM вместо крестиков и da вместо точек. У вас получится что-то вроде DUM-da-da-DUM-da-DUM-da-da-DUM-da-DUM-da. Это ритм. Ритм состоит из акцентированных и тихих отрезков. Тот, который был получен выше, придумали не мы. Идея принадлежит пигмеям ака из Центральноафриканской республики; ритм под названием «Венда», который отбивают руками, служит аккомпанементом одной южноафриканской песни; аналогичный ритм встречается в Македонии и на Балканах. Но это еще не все. Если начать его со второго крестика (то есть с акцентированной части), получится следующее:

× • × • • × • × • × • •

Это популярный ритм колоколов на Кубе и в Западной Африке, а также барабанный ритм в Кении, хотя он используется и в Македонии (опять). Если начать его с третьей, четвертой или пятой акцентированной части, получатся другие ритмы, популярные в разных уголках мира.

Уникальное ли это явление? Мы можем создать 12-элементный ритм из семи акцентированных и пяти тихих частей — своеобразное зеркальное отражение последовательности, которую мы рассматривали выше. Если в точности следовать той же процедуре, получится следующее:

× • × × • × • × × • × •

Это тоже ритм. Он встречается в провинции Ашанти, Гана; а если его начать с последней акцентированной части, получится ритм, который использует народность йоруба в Нигерии, а также население Центральной Африки и Сьерра-Леоне.

Давайте расширим нашу географию. Начнем с пяти ударов и 11 тихих отрезков. Получится следующее:

× • • × • • × • • × • • × • • •

Это ритм босанова, только немного смещенный. Оригинал начинается с третьего удара и выглядит так:

× • • × • • × • • × • • × • • •

Возьмем три удара и четыре тихих отрезка:

× • × • × • •

Ритм в пропорции семь/четыре довольно популярен, и не только в традиционной музыке. Среди прочих мелодий его можно услышать в песне Monkey.



Таким образом, составляя столбцы из крестиков и точек и перемещая их туда-сюда, как мы только что делали, можно вывести много других ритмов. Чтобы проиллюстрировать эту процедуру, для наглядности мы изменили оставшиеся столбцы. Вместо создания столбцов, проверки их расположения и перемещения их из стороны в сторону мы могли бы оформить все это в виде простых численных операций. Давайте вернемся к примеру из 12 частей и семи ударений. Для начала разделим 12 на 7, что даст нам целую часть 1 и остаток 5:

$$12 = 1 \times 7 + 5.$$

Это говорит о том, что семь ударений нужно поместить в начало. Получится семь акцентированных отрезков, за которыми следуют оставшиеся пять тихих:

× × × × × × × • • • • •

Давайте еще раз выполним деление, но на этот раз разделим сам делитель из предыдущей операции 7 на полученный остаток 5. Мы снова получим целую часть 1 и остаток 2:

$$7 = 1 \times 5 + 2.$$

Это означает, что нам нужно взять пять столбцов справа и поместить их под пятью столбцами слева. Останется 2:

× × × × × × ×
• • • • •

Повторим тот же шаг: разделим делитель предыдущей операции 5 на ее остаток 2. Получится 2 с остатком 1:

$$5 = 2 \times 2 + 1.$$

Из этого следует, что мы должны *дважды* взять два крайних справа столбца и поместить их под двумя столбцами, крайними слева. Останется 1:

```

× × ×
• • •
× ×
× ×
• •

```

Обратите внимание на то, что *дважды* — это эквивалент двух шагов без использования деления, которые мы выполняли ранее. Сначала мы перешли бы от

```

× × × × × × ×
• • • • •

```

к

```

× × × × ×
• • • • •
× ×

```

а затем к

```

× × ×
• • •
× ×
× ×
• •

```

Если свести столбцы вместе, получится ритм Mре:

```

× • × × • × • × × • × •

```

Наш первый алгоритм

Метод, который мы применяли выше, можно записать чуть более формально, разбив его на следующие шаги. Предполагается, что мы начинаем с двух чисел, a и b . Пусть a будет общим количеством отрезков. Если ударений больше, чем тихих отрезков, b обозначает количество ударений. В противном случае это количество тихих отрезков. Вначале мы выставляем в ряд ударения, за которыми идут тихие отрезки.

1. Выполняем деление a на b . Так мы получим целую часть (q) и остаток (r). Выходит $a = q \times b + r$. Это знакомое всем целочисленное деление. Берем b крайних справа столбцов q раз и размещаем их под крайними слева столбцами, оставляя справа r столбцов.
2. Если остаток r равен нулю или единице, мы останавливаемся. В противном случае возвращаемся к шагу 1, но на этот раз роль a играет b , а роль b — r . Иными словами, мы повторяем первый шаг, делая a равным b , и b равным r .

Повторяем деление в эти два этапа, пока это имеет смысл. Выполненные нами шаги можно оформить в виде следующей таблицы (как и раньше, начинаем с $a = 12$ и $b = 7$ и получаем в каждой строке $a = q \times b + r$):

a	q	b	r
12	1	7	5
7	1	5	2
5	2	2	1

Если проанализировать эту таблицу, можно убедиться в том, что каждая строка соответствует одному этапу формирования и перемещения столбцов. Но этот метод можно описать еще точнее. На самом деле мы получили последовательность шагов, которые можно выполнить с помощью ручки и листа бумаги. Это наш первый алгоритм! Он позволяет создавать последовательности, которые соответствуют удивительно большому количеству музыкальных ритмов. Меняя смещения и число тихих отрезков, мы можем получить около 40 последовательностей, которые встречаются в разных ритмах по всему миру. Этот простой алгоритм (всего два шага, которые повторяются) выдает столько интересных результатов!

Но наш алгоритм этим не ограничивается. Поскольку речь идет о делении двух чисел, давайте подумаем о следующей проблеме общего характера: как найти максимальный делитель, который подходит как для a , так и для b ? Это *наибольший общий делитель* двух чисел. Он встречается в элементарной арифметике, например в следующей задачке: если у нас есть 12 пакетов с крекерами и 4 пакета с сыром, как распределить их по корзинам таким образом, чтобы в каждой корзине крекеры и сыр были в одной и той же пропорции? Так как 12 делится на четыре, вы получите четыре корзины с тремя пакетами крекеров и одним пакетом сыра в каждой; наибольший общий делитель для 12 и 4 равен 4. Все становится интереснее, если взять 12 пакетов с крекерами и восемь пакетов с сыром. Одно на другое не делится, но наибольшее число, на которое можно поделить 12 и 8, равно 4; это означает, что у вас опять получится четыре корзины, но на этот раз с тремя пакетами крекеров и двумя пакетами сыра в каждой.

Так как же найти наибольший общий делитель для двух произвольных целых чисел? Мы уже видели, что, если одно число делится на другое, этот делитель и есть наибольший общий. В противном случае достаточно найти наибольший общий делитель остатка от деления этих двух чисел и второго числа. Это проще изобразить с помощью математических символов. Если у нас есть два числа, a и b , их наибольший общий делитель равен наибольшему общему делителю остатка от $a \div b$ и b . Это возвращает нас к ритмам: *их подбор подобен поиску наибольшего общего делителя двух чисел.*

Метод поиска наибольшего общего делителя для двух чисел называется *алгоритмом Евклида*, в честь древнегреческого математика Евклида, который впервые описал его в своей книге «Начала» (300 г. до н. э.). Основная идея этого метода в том, что наибольший общий делитель для двух чисел остается неизменным, если вместо большего числа подставить результат вычитания из него меньшего числа. Возьмем, к примеру, 56 и 24. Их наибольший общий делитель равен 8, что является наибольшим общим делителем и для $56 - 24 = 32$ и 24, а также для 32 и 24 и т. д. Повторение вычитания — это фактически деление, поэтому алгоритм Евклида состоит из следующих шагов.

1. Чтобы найти наибольший общий делитель для a и b , делим a на b . Это даст нам целую часть (q) и остаток (r). Тогда получается $a = q \times b + r$.
2. Если остаток r равен 0, мы останавливаемся, а наибольший общий делитель для a и b будет равен b . В противном случае возвращаемся к шагу 1, но на этот раз роль a играет b , а роль b — r . Иными словами, мы повторяем первый шаг, делая a равным b и b равным r .

Это, в сущности, те же шаги, которые мы видели ранее. Единственное отличие в том, что при поиске ритма мы останавливаемся, если остаток во втором шаге равен 0 или 1, а алгоритм Евклида останавливается, только если остаток равен 0. Но на самом деле это то же самое: если остаток равен 1, при следующем повторении первого шага получается нулевой остаток, так как на 1 делится любое целое число. Попробуем 9 и 5: $9 = 1 \times 5 + 4$, поэтому мы делаем $5 = 1 \times 4 + 1$ и затем $4 = 1 \times 4 + 0$; таким образом наибольший общий делитель 9 и 5 равен 1.

Чтобы вам было легче понять, как это работает, взгляните на следующую таблицу с $a = 136$ и $b = 56$, похожую на ту, которую мы уже видели при обсуждении наших ритмов. Получается, что наибольшим общим делителем для 136 и 56 является число 8.

a	q	b	r
136	2	56	24
56	2	24	8
24	3	8	0

Как уже отмечалось с 9 и 5, алгоритм Евклида выдает правильный результат во всех случаях, даже когда у двух чисел нет никакого общего делителя, кроме 1. Именно это произошло с $a = 9$ и $b = 5$. Вы можете сами увидеть, что случится, если попробовать выполнить этот алгоритм для $a = 55$ и $b = 34$; он пройдет несколько шагов, но в итоге определит, что единственный общий делитель равен 1.

Шаги алгоритма Евклида выполняются в строго определенном порядке. Ниже описано, как они объединяются в целое.

1. Шаги формируют *последовательность*.
2. Шаги могут описывать *выборку*, которая определяет, что делать дальше. В шаге 2 мы проверяем, равен остаток 0 или нет. Затем в зависимости от результата предоставляются две альтернативы: мы либо останавливаемся, либо возвращаемся к шагу 1.
3. Шаги могут находиться в *цикле* и выполняться многократно. Если в шаге 2 остаток не равен 0, мы возвращаемся к шагу 1.

Эти три способа объединения шагов называются *управляющими структурами*, поскольку они определяют, какие действия будут выполнены в процессе работы алгоритма. Все алгоритмы структурированы подобным образом.

Они состоят из шагов, на которых производятся вычисления и обрабатываются данные; эти шаги собираются в единое целое и регулируются с помощью приведенных выше управляющих структур. Более сложные алгоритмы состоят из большего числа шагов, и их механизмы регулирования могут быть более замысловатыми. Но этих трех управляющих структур достаточно для описания того, как скомпоновать шаги любого алгоритма.

Кроме того, шаги алгоритма могут работать с вводом, который мы предоставляем. Ввод — это данные, обрабатываемые алгоритмом. Если поставить данные во главу угла, можно сказать, что алгоритмы предназначены для преобразования информации, описывающей задачу, в нечто, соответствующее ее решению.

Мы обнаружили алгоритм, представляющий собой операции деления, за музыкальными ритмами, но в действительности нам необязательно погружаться так глубоко, ведь операция деления сама по себе является алгоритмом. Даже если вам ничего не известно о Евклиде, вы знаете, как поделить два больших числа; в начальной школе все учатся умножать и делить в столбик. Учителя час за часом вдавливали нам в головы, как выполнять эти операции; это были последовательности шагов, в ходе которых мы размещали числа в нужных местах и делали с ними всякие вещи, то есть алгоритмы. Но алгоритмы не ограничены одними лишь числами. Вы сами видели, что с их помощью можно создавать музыку. Это способ распределения ударений на отрезке времени, но тот же принцип применим и для упаковывания крекеров с сыром.

Источник применения алгоритма Евклида к ритмам довольно необычный — лаборатория SNS в Ок-Ридж, Теннесси. SNS расшифровывается как (нейтронный источник); он генерирует интенсивное импульсное нейтронное излучение, которое используют для экспериментов в физике элементарных частиц (корень слова *spallation*, *spall*, означает *дробление* материи на более мелкие части; если обратиться к ядерной физике, тяжелое ядро излучает большое количество протонов и нейтронов, если попасть в него высокоэнергетической частицей). В работе SNS участвуют такие компоненты, как блоки питания высокого напряжения, которые позволяют сделать пульсацию более равномерной. Алгоритм, выведенный для распределения импульсов, по своему принципу не отличается от евклидового или того, который мы использовали для получения ритмов. Таким образом мы пришли от музыки к числам, а затем к субатомным частицам.

Алгоритмы, компьютеры и математика

Мы сказали, что алгоритмы не имеют прямого отношения к компьютерам, но на сегодня большинство людей связывает эти понятия. Действительно, компьютер позволяет проявить настоящий потенциал алгоритмов, однако это всего лишь устройство, которое выполняет отдаваемые ему команды. Чтобы командовать компьютером, мы его *программируем*, и обычно это делается для выполнения алгоритмов.

Это подводит нас к самому программированию. Программирование — это преобразование наших намерений в такой вид, в котором их может понять компьютер. Этот «вид» мы называем *языком программирования*, потому что иногда это действительно выглядит так, будто мы пишем текст на человеческом языке. Но языки программирования несравнимы по своему разнообразию и сложности с человеческой речью. Компьютер, конечно же, ничего в действительности не понимает. Возможно, в будущем что-то изменится и нам удастся создать по-настоящему разумные устройства, но пока, когда мы говорим о том, что компьютер что-то понимает, мы имеем в виду, что язык переводится в набор инструкций для управления током в электронных схемах (вместо тока также можно использовать свет, но принцип тот же).

Если алгоритм — это последовательность шагов, которые мы можем выполнить самостоятельно, то программирование — это процесс записи этих шагов в формате, понятном компьютеру. И компьютер сам занимается их выполнением. Компьютеры работают намного быстрее людей, поэтому они могут выполнять те же шаги за меньшее время. Основопологающим фактором вычислений является *скорость*. Компьютеры не могут делать что-то принципиально отличное от того, что делаем мы, люди, но они могут делать это быстрее — намного быстрее. Алгоритм становится существенно полезнее на компьютере, поскольку там он может быть выполнен на порядок быстрее, хотя *шаги*, из которых он состоит, *все те же*.

Язык программирования позволяет нам описать для компьютера шаги алгоритма. Он также дает возможность структурировать их с помощью трех фундаментальных управляющих структур: последовательности, выбора и итерации. Мы записываем эти шаги и определяем, как они должны быть скомпонованы, используя словарь и синтаксис нашего языка программирования.

Помимо скорости, у компьютеров есть еще одно преимущество. Помните, как вы учились делить и умножать в столбик — это требовало много практики и было довольно скучно. Как уже отмечалось выше, эти вещи

Программирование — это преобразование наших намерений в такой вид, в котором их может понять компьютер. Этот «вид» мы называем *языком программирования*.

вбивались нам в головы в раннем возрасте, и процесс не самый приятный. Но компьютерам не бывает скучно, поэтому еще одна причина доверять им алгоритмы состоит в том, что так мы можем освободить себя от рутинной работы и заняться чем-то более интересным.

Алгоритмы в основном выполняются на компьютере, но на языках программирования их, как правило, записывают люди, поэтому мы должны понимать, как они работают и как их можно использовать. Это подводит нас к кое-чему очень важному, о чем иногда забывают даже бывалые программисты и опытные специалисты в области компьютерных наук. Алгоритм по-настоящему можно понять только одним способом, — выполнив его вручную. Мы должны уметь выполнять алгоритмы так же, как компьютеры выполняют программы, которые реализуют эти алгоритмы. Нам повезло жить во времена, когда для изучения всевозможных вещей доступно невероятное количество материала: превосходные видеоуроки, анимированные курсы и симуляции находятся на расстоянии одного щелчка мышью. Это все замечательно, но на случай каких-либо трудностей всегда держите при себе ручку и блокнот. Это относится и к данной книге. Вы уверены, что поняли, как создаются ритмы? Пробовали делать это сами? Можете ли вы найти наибольший общий делитель для 252 и 24?

Все программы реализуют последовательность шагов для выполнения каких-то задач, поэтому может возникнуть соблазн приравнять их к алгоритмам. Но мы относимся к этому более строго и хотим, чтобы наши шаги обладали определенными характеристиками.

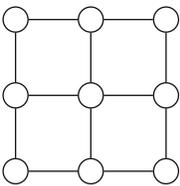
1. Количество шагов должно быть конечным. Алгоритмы не могут работать вечно (программа может работать до тех пор, пока работает компьютер, на котором она запущена; но такая программа была бы не реализацией алгоритма, а просто вычислительным процессом).
2. Шаги должны быть точными, чтобы их можно было выполнять без двусмысленностей.
3. Алгоритм может работать с каким-нибудь вводом; например, алгоритм Евклида работает с двумя целыми числами.
4. У алгоритма есть какой-то вывод; в этом весь его смысл: вернуть что-то в качестве результата. В алгоритме Евклида это наибольший общий делитель.
5. Алгоритм должен быть эффективным. У человека должна быть возможность выполнить каждый его шаг за разумное время с помощью ручки и листа бумаги.

Если алгоритм — это
последовательность шагов,
которые мы можем выполнить
самостоятельно,
то программирование — это
процесс записи этих шагов
в формате, понятном
компьютеру.

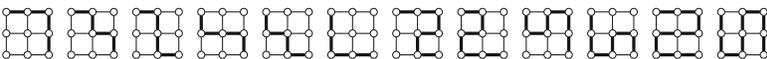
Эти характеристики гарантируют, что алгоритм что-то делает. Он существует для того, чтобы делать что-то полезное. Бессмысленные алгоритмы тоже существуют, и специалисты в области компьютерных наук могут изобретать их для развлечения или по ошибке, но нас с вами интересуют алгоритмы, имеющие реальное применение. При работе с алгоритмами недостаточно просто показать возможность выполнить какую-либо задачу. Мы преследуем практические цели, поэтому наши алгоритмы должны выполнять свою работу хорошо.

В этом заключается принципиальное отличие между алгоритмами и математикой. На заре развития информатики большинство компьютерщиков были математиками, и в компьютерных науках используется много математических концепций, но это не математическая дисциплина. Математик хочет доказать какое-то *тождество*; специалист в области информатики хочет *применить* это тождество на практике.

Первая характеристика алгоритма, которую мы привели выше, требует, чтобы количество шагов было конечным. Но это не совсем точная формулировка. Конечного числа шагов недостаточно. Мы хотим, чтобы этих шагов было достаточно мало для того, чтобы их можно было выполнить на практике и чтобы наш алгоритм мог завершиться за разумное время. Это означает, что изобретение алгоритма — это лишь полдела; его еще нужно сделать эффективным. Давайте рассмотрим пример, чтобы проиллюстрировать отличие между знанием чего-то и умением сделать что-то эффективно. Представьте, что у вас есть такая сетка:



Мы хотим найти кратчайший путь из левого верхнего угла сетки в правый нижний угол, но без посещения одного и того же места дважды. Длина каждого пути равна количеству звеньев между точками сетки. Это можно сделать так: найти все возможные пути, измерить длину каждого из них и выбрать самый короткий (или любой из них, если их несколько). Всего путей 12, и все они изображены ниже.



Существуют пять путей длиной 4, поэтому мы можем выбрать любой из них.

Но сетки бывают разных размеров: 4×4 , 5×5 и больше. Получается, что наш метод не очень хорошо масштабируется. В сетке размером 4×4 существуют 184 пути из левого верхнего угла в правый нижний; в сетке 5×5 таких путей и вовсе 8512. Количество путей быстро растет (на самом деле очень быстро), поэтому даже просто подсчитать их бывает затруднительно. Если взять сетку размером 26×26 , получится 8 402 974 857 881 133 471 007 083 745 436 809 127 296 054 293 775 383 549 824 742 623 937 028 497 898 215 256 929 178 577 083 970 960 121 625 602 506 027 316 549 718 402 106 494 049 978 375 604 247 408 путей. Это число состоит из 151 цифры; оно было вычислено компьютерной программой, реализующей алгоритм. Все верно: мы используем один алгоритм, чтобы разобраться в поведении другого.

Процедура прохождения всех возможных путей с последующим выбором кратчайшего, вне всяких сомнений, является правильной и всегда дает нам самый короткий путь (или один из них, если их несколько), но ее точно нельзя назвать практичной. К тому же она совершенно бесполезна, так как существуют алгоритмы, которые выдают тот же результат без прохождения всех возможных путей и тем самым экономят нам уйму времени и позволяют работать с сетками любого размера. Для поиска ответа в сетке 26×26 количество необходимых шагов исчисляется сотнями; в следующей главе вы сможете сами в этом убедиться.

Вопрос, что такое практичный алгоритм и каким образом один алгоритм оказывается практичнее другого, лежит в основе их практического применения. Читая эту книгу, вы будете сталкиваться с тем, что для решения одной и той же проблемы существует несколько алгоритмов, и выбирать следует тот, который лучше всего подходит в каждой конкретной ситуации. Как это бывает с любыми инструментами, некоторые алгоритмы оказываются более подходящими для определенных сценариев использования, чем другие. Но, в отличие от многих других инструментов, алгоритмы можно оценивать четко определенным способом.

Оценивание алгоритмов

При подборе алгоритма для решения какой-либо задачи мы хотим знать, как он себя поведет. Важный фактор — скорость. Алгоритмы выполняются на компьютерах, потому что так быстрее.

Аппаратное обеспечение постоянно развивается, и нам зачастую недостаточно знать, как программа, реализующая алгоритм, работает на отдельном компьютере. Наш компьютер может быть быстрее или медленнее того, на котором анализировался алгоритм, и спустя несколько лет замеры производительности на устаревшем устройстве будут представлять разве что историческую ценность. Нам нужен метод оценки эффективности алгоритма в отрыве от оборудования.

Однако то, как мы анализируем алгоритм, должно отражать масштаб задачи, которую мы пытаемся решить. Нас мало интересует, сколько времени займет сортировка 10 элементов; в крайнем случае это можно сделать даже вручную. А вот то, как долго будет сортироваться миллион элементов, уже интереснее. Мы хотим оценить поведение алгоритма в контексте нетривиальных задач.

Для этого нам нужен способ определения масштабов задач, которые решает алгоритм. Но подходящая величина зависит от конкретной задачи. Если мы хотим отсортировать на нашем компьютере ряд элементов, нужной нам величиной будет их количество (а не, скажем, их размер или то, как они структурированы). Если нам нужно умножить два числа, подходящей величиной будет количество цифр в обоих числах (это имеет смысл и для нас, людей: сложность умножения в столбик зависит от того, из скольких разрядов состоит каждое число). Анализируя задачу и подбирая алгоритм для ее решения, мы всегда учитываем ее масштаб.

Масштаб каждой конкретной задачи можно оценить по-разному, но в любом случае мы обозначаем его целым числом, n . Если вернуться к предыдущим примерам, n — это либо количество элементов для сортировки, либо количество разрядов в числах, которые нужно умножить. И только после этого можно говорить о производительности алгоритмов, решающих задачи масштаба n .

Время, нужное алгоритму, зависит от его *вычислительной сложности*. Вычислительная сложность алгоритма — это количество ресурсов, которое требуется для его работы. Ресурсы бывают двух основных видов: время, необходимое для выполнения, и объем компьютерной памяти.

Пока сосредоточимся на времени. Компьютеры могут иметь разную производительность, поэтому время выполнения алгоритма на конкретном оборудовании может служить лишь косвенным показателем того, что следует ожидать. Нам нужно что-то более универсальное. Скорость работы компьютера определяется тем, как долго он выполняет базовые операции. Чтобы не углубляться в технические детали, мы будем говорить о *количестве*

операций, необходимых для работы алгоритма, а не о том, насколько быстро они выполняются на конкретном компьютере.

Однако следует отметить, что мы позволим себе небольшую вольность в использовании терминологии и будем считать слова «операции» и «время» синонимами. Строго говоря, эти понятия нужно разделять, отмечая, что алгоритму требуется « x операций», но вместе с этим мы будем указывать, что алгоритму нужно «время x »; это время, которое требуется алгоритму для выполнения x операций на любом компьютере. И хотя на самом деле время будет варьироваться в зависимости от программного обеспечения, это неважно, если мы хотим сравнить два алгоритма, которым нужно «время x » и «время y » на *одном* и том же компьютере, каким бы он ни был.

Теперь вернемся к масштабу задач, которые решает алгоритм. Поскольку тривиальные задачи нас не интересуют, мы не станем останавливаться на мелком масштабе. Мы не можем назвать конкретные цифры, но отметим, что масштаб должен быть значительным.

Существует определение сложности, полезность которого доказана на практике. У него есть название и условное обозначение. Мы записываем его как $O(\cdot)$ и называем *большое*. Вместо точки в скобках указывается выражение. Эта запись означает, что алгоритму нужно время, не больше, чем множитель этого выражения. Давайте посмотрим, что это означает.

- Если вам нужно найти элемент в последовательности длиной n , которая никоим образом не упорядочена, сложность этой задачи будет $O(n)$. То есть, если у нас есть n элементов, время, необходимое, чтобы найти один из них, будет не больше, чем некое число, умноженное на n .
- Если вы хотите умножить в столбик два числа, состоящих из n разрядов, сложность этой операции будет $O(n^2)$. То есть время, необходимое на ее выполнение, будет не больше, чем некое число, умноженное на квадрат n .

Если сложность нашего алгоритма равна $O(n)$, то при вводе размером 10 000 следует ожидать, что на работу ему потребуется количество шагов, кратное десяти тысячам. Если сложность алгоритма $O(n^2)$, то для ввода аналогичного размера потребуется около ста миллионов шагов. Для многих задач этот масштаб не является большим. Компьютеры постоянно сортируют десятки тысяч элементов. Но вы можете видеть, что количество шагов, представленное сложностью алгоритма, может быть довольно крупным.

Давайте рассмотрим несколько примеров, чтобы вы могли получить представление о масштабах некоторых показателей, с которыми вы будете сталкиваться. Возьмем число 100 миллиардов, или 10^{11} ; оно состоит из 11 нолей. Если взять 100 миллиардов гамбургеров и разложить их один за другим, этой длины хватит, чтобы обогнуть земной шар 216 раз или, например, дойти до Луны и обратно.

Миллиард эквивалентен приставке *giga-*, по крайней мере в мире компьютеров. За миллиардом (гига-) идет триллион (тера-); это 1000 миллиардов, или 10^{12} . Если вы начнете произносить по одному числу в секунду, то, чтобы досчитать до триллиона, вам понадобится 31 000 лет. Умножим это еще на 1000 и получим квадриллион, 10^{15} или *пета-*; если верить биологу Э. О. Уилсону, общее число муравьев на нашей планете находится в пределах от 1 до 10 квадриллионов. Иными словами, в природе существует от 1 до 10 «петамуравьев».

Вслед за квадриллионом идет квинтиллион, или *екса-*; это 10^{18} — примерно столько, сколько песчинок на десяти больших пляжах. Например, на 10 Копакабанах умещается одна «эксапесчинка». Идем дальше. 10^{21} — это один секстиллион, или *зетта-*. В наблюдаемой вселенной существует одна «зеттазвезда». Последний общепринятый префикс, *йотта-*, обозначает 10^{24} , или один септиллион. Но всегда можно взять большее число. Например, 10^{100} называется *гугол* (англ. googol) — вы, наверное, слышали о компании с похожим названием, в котором специально допущена ошибка. Если возвести 10 в степень гугол, $10^{10^{100}}$, получится *гуголплекс*.

Эти аналогии помогут нам оценить относительные достоинства конкретных алгоритмов, которые мы будем рассматривать на страницах этой книги. Теоретически алгоритмы могут иметь любую сложность, но те из них, с которыми мы обычно сталкиваемся, можно разделить на несколько разных групп.

Категории сложности алгоритмов

Самыми быстрыми являются алгоритмы, работающие не дольше фиксированного времени, независимо от ввода. Мы обозначаем эту сложность как $O(1)$; например, алгоритм, который проверяет, является ли последняя цифра числа четной, не зависит от того, насколько большое это число, и всегда будет выполняться за одно и то же время. 1 в $O(1)$ означает, что максимальное количество шагов, необходимое для выполнения алгоритма, кратно 1, то есть оно всегда одно и то же.

Прежде чем мы познакомимся со следующей категорией сложности, следует обсудить, как именно вещи могут расти и уменьшаться. Многократное сложение эквивалентно умножению. Многократное умножение — это возведение в степень (экспоненциальный рост). Мы только видели, насколько большими могут становиться числа, если они растут экспоненциально (как в случае с 10^{12}). Возведение в степень может дать на удивление большую величину — это явление называется *экспоненциальным ростом*.

Проиллюстрируем его с помощью (вероятно, недостоверной) легенды об изобретении шахмат. Правитель страны, где были придуманы шахматы, спросил их создателя о том, что он хотел бы получить в качестве подарка (в таких историях это всегда «он»). Тот ответил, что хочет одно зерно риса на первой клетке шахматной доски, два зерна на второй, четыре на третьей и т. д. Правитель подумал, что легко отделался, и приказал исполнить желание. Но все оказалось не так просто. Эта последовательность растет вместе со степенью двойки: $2^0 = 1$ в первой клетке, $2^1 = 2$ во второй, $2^2 = 4$ в третьей; таким образом, в последней клетке количество зерен равно 2^{63} — это недостижимая величина, эквивалентная 9 223 372 036 854 775 808, или примерно 9 квинтиллионам.

Экспоненциальный рост может также объяснить, почему так сложно сложить кусочек бумаги много раз подряд. При каждом сгибании количество слоев бумаги удваивается. После десяти сгибаний получается $2^{10} = 1024$ слоя. Если бумага имеет толщину 0,1 миллиметра, у вас получится комочек толщиной 10 сантиметров. Сложить такой ком вдвое может быть физически невозможно, даже если приложить грубую силу, так как для этого его длина должна быть больше его толщины.

Именно благодаря экспоненциальному росту компьютеры становятся все более мощными год от года. Согласно *закону Мура*, количество транзисторов на интегральной схеме удваивается каждые два года. Этот закон назван в честь Гордона Мура, основателя Fairchild Semiconductor и Intel. Он сделал это наблюдение в 1965 году, и оно оказалось пророческим. В 1971 году в процессоре Intel 4004 насчитывалось около 2000 транзисторов, а в 2017-м этот показатель достиг 19 миллиардов (в 32-ядерном процессоре AMD Epyc).

С ростом мы разобрались, теперь давайте поговорим о его противоположности. Если у вас есть множитель какого-то числа, для получения исходного значения нужно выполнить деление. Но, если у вас есть степень чего-то, a^n , как обернуть эту операцию вспять? Противоположностью возведения в степень является *логарифм*.

Иногда говорят, что логарифмы находятся на границе между математикой для всех и математикой для посвященных; даже само название звучит

непонятно. Если эта операция кажется вам запутанной, помните, что логарифм числа — это противоположность возведения числа в степень. Возведение в степень предполагает многократное умножение, а логарифм заключается в многократном делении.

Логарифм отвечает на вопрос: «в какую степень возвести число, чтобы получить нужное значение?» Число, которое мы возводим, называется *основанием* логарифма. Поэтому, если спросить: «в какую степень нужно возвести 10, чтобы получить 1000?», ответом будет 3, поскольку $10^3 = 1000$. Конечно, мы можем взять для возведения в степень другое число, то есть другое основание. Логарифм записывается как $\log_a x$ и отвечает на вопрос: «в какую степень нужно возвести a , чтобы получить x ?». Если $a = 10$, подстрочное значение опускается ввиду распространенности логарифмов по основанию 10, поэтому вместо $\log_{10} x$ мы пишем просто $\log x$.

Существуют два других распространенных основания. Если основание равно математической константе e , мы пишем $\ln x$. Эта константа называется *числом Эйлера* и равна приблизительно 2,71828. В естественных науках $\ln x$ встречается повсеместно, поэтому его называют *натуральным логарифмом*. Другое распространенное основание — 2, и вместо $\log_2 x$ мы пишем $\lg x$. Это основание в основном применяется в информатике и алгоритмах, и в других областях его почти никогда не используют. Мы уже встречали его в примере со сгибанием кусочка бумаги: если результат состоит из 1024 слоев, он был сложен $\lg 1024 = \lg 2^{10} = 10$ раз. В истории с шахматами количество зерен риса зависит от количества удваиваний: $\lg 2^{63} = 63$.

$\lg x$ часто встречается в алгоритмах, потому что этот логарифм всегда используют при разбиении одной большой задачи на две мелких; этот принцип называется *разделяй и властвуй*, и он работает точно так же, как сгибание листа бумаги вдвое. Самый эффективный способ поиска по набору отсортированных элементов имеет сложность $O(\lg n)$. Поразительно, не так ли? Из этого следует, что для поиска по миллиарду упорядоченных элементов достаточно всего $\lg 10^9 \approx 30$ шагов.

Вторым лучшим выбором после алгоритмов, выполняемых за фиксированное время, являются алгоритмы с логарифмической сложностью. Дальше идут алгоритмы, которые выполняются за $O(n)$; их называют алгоритмами *линейного времени*, поскольку время их выполнения растет пропорционально n , то есть они растут в качестве множителей n . Мы уже видели, что время, необходимое для поиска по списку неупорядоченных элементов, пропорционально количеству этих элементов, $O(n)$. Обратите внимание на то, как повышается сложность по сравнению с упорядоченным списком; то, как

организованы вводные данные, может существенно повлиять на способ решения задачи. В целом линейное время — это лучшее, что можно ожидать от алгоритма, которому необходимо прочитать весь ввод, так как для n элементов это потребует $O(n)$ времени.

Если совместить линейное и логарифмическое время, получится алгоритм *логлинейного времени*, продолжительность выполнения которого равна n , умноженному на его логарифм, $n \lg n$. Лучший алгоритм для сортировки (то есть для упорядочивания элементов) имеет сложность $O(n \lg n)$. Это может показаться немного неожиданным, ведь если у вас есть n элементов и вы хотите сравнить каждый из них со всеми остальными, вам потребуется время $O(n^2)$, что больше, чем $O(n \lg n)$. Кроме того, если вы хотите отсортировать n элементов, для их полного перебора точно понадобится время $O(n)$. Их сортировка требует умножения этого n на коэффициент, меньший, чем само это число. Позже вы увидите, как этого можно добиться.

К следующей категории вычислительной сложности относится n , возведенное в фиксированную степень, $O(n^k)$; это так называемая *полиномиальная сложность*. Алгоритмы полиномиального времени эффективны, если не считать редких случаев, когда k является большим числом. Мы обычно радуемся, когда при решении вычислительной задачи удастся разработать алгоритм полиномиального времени.

Сложность вида $O(k^n)$ называют *экспоненциальной*. Обратите внимание на то, что, в отличие от полиномиальной сложности, степень здесь изменяется. Мы уже видели примеры взрывного экспоненциального роста. Времени, оставшегося до тепловой смерти вселенной, не хватит для вычисления экспоненциальных алгоритмов с нетривиальным вводом. Такие алгоритмы интересны с теоретической точки зрения, так как они показывают, что решение можно найти. Мы можем попытаться подобрать более подходящие алгоритмы с меньшей сложностью или доказать, что их не существует; в последнем случае можно пойти на компромисс и удовлетвориться, к примеру, приблизительным решением.

Но кое-что растет даже быстрее, чем экспонента. Речь идет о *факториале*. Напомним, что факториал натурального числа n (записывается как $n!$) — это просто произведение всех натуральных чисел вплоть до (и включая) n : $100! = 1 \times 2 \times 3 \times \dots \times 100$. Но если с $100!$ вы еще не сталкивались, то факториал $52!$ вам уже, наверное, попадался, даже если вы об этом не знаете. Это количество возможных комбинаций в колоде карт. Алгоритмы, время выполнения которых измеряется факториалами, имеют *факториальную сложность*.

Числа вроде $100!$ являются довольно экзотическими, но они встречаются во многих повседневных ситуациях, и не только в карточных играх. Возьмем, к примеру, следующую задачу: «Если у нас есть список городов и расстояний между каждой парой, как найти кратчайший маршрут для посещения каждого города с последующим возвращением в исходную точку?». Это так называемая *задача коммивояжера*, и ее очевидное решение состоит в том, чтобы проверить каждый возможный маршрут, охватывающий все города. К сожалению, это будет $n!$ для n городов. Если количество городов превышает, скажем, 20, задача становится практически нерешаемой. Существуют алгоритмы с немного лучшим временем выполнения, чем $O(n!)$, но даже они являются непрактичными. Как ни удивительно, единственный способ решения такой прямолинейной задачи за приемлемое время — нахождение неоптимального, но достаточно близкого к нему решения. Многие задачи, имеющие большое практическое значение, являются *труднорешаемыми*, то есть нам неизвестен практичный алгоритм, который может дать точный результат. Тем не менее поиск *аппроксимационных алгоритмов* остается динамичной областью информатики.

В приведенной ниже таблице приводятся результаты различных функций, разделенных по рассмотренным только что категориям сложности, для разных значений n . В первой строке указаны значения n , соответствующие линейной сложности; в следующих строках сложность возрастает. Вместе с n увеличивается и результат функции, но это происходит по-разному. Функция n^3 переходит от миллиона сразу к квинтиллиону, но это ничто по сравнению с 2^{100} или $100!$. Мы сделали отступ после n^k , чтобы отделить практичные алгоритмы от непрактичных. Границей здесь выступают алгоритмы полиномиального времени, которые, как мы уже видели, можно использовать на практике. Все, что сложнее, обычно не имеет практического применения.

n	1	10	100	1000	1 000 000
$\lg n$	0	3,32	6,64	9,97	19,93
$n \lg n$	0	33,22	664,39	9 965,78	$1,9 \times 10^7$
n^2	1	100	10 000	1 000 000	10^{12}
n^3	1	1000	1 000 000	10^9	10^{18}
n^k	1	10^k	100^k	1000^k	$1\,000\,000^k$
2^n	2	1024	$1,3 \times 10^{30}$	10^{301}	$10^{10^{5,5}}$
$n!$	1	3 628 800	$9,33 \times 10^{157}$	4×10^{2567}	$10^{10^{6,7}}$

В восемнадцатом веке славные жители Кёнигсберга прогуливались по своему городу воскресными днями. Кёнигсберг был построен на берегах реки Прегель, которая сформировала внутри городских стен два больших острова; эти острова соединялись с континентальной частью и между собой семью мостами.

Из-за превратностей европейской истории Кёнигсберг часто переходил из рук в руки: от рыцарей Тевтонского ордена к Пруссии, России, Веймарской республике и нацистской Германии; после Второй мировой войны он стал частью СССР и был переименован в Калининград — это название он носит и по сей день. В наши дни он принадлежит России, хотя и не имеет с ней прямого сухопутного сообщения. Калининград находится в российском анклав на берегу Балтийского моря, зажатый между Польшей и Литвой.

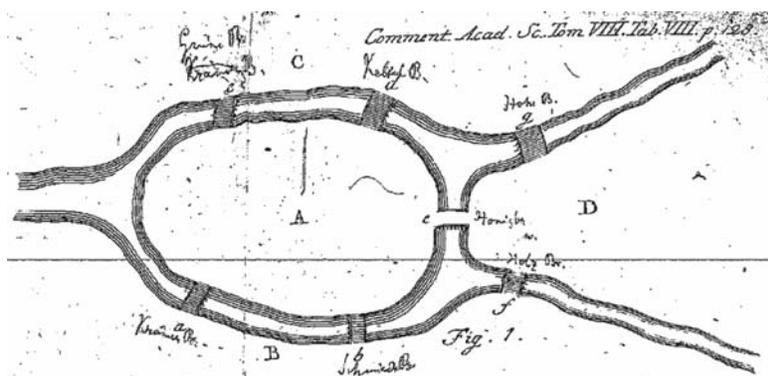
Когда-то славные горожане озаботились таким вопросом: возможно ли прогуляться по городу, перейдя все семь мостов ровно по одному разу? Это так называемая *задача о семи кёнигсбергских мостах*. Чтобы понять, о чем идет речь, взгляните на изображение Кёнигсберга того времени. Мосты обведены овалами. У города есть два острова, но на гравюре полностью виден только один из них; другой уходит вправо за пределы карты.



Мы точно не знаем как, но об этой задаче узнал известный швейцарский математик Леонард Эйлер; она упоминается в письме от 9 марта 1736 года, которое ему отправил мэр Данцига, прусского города, расположенного 129 километрами восточнее Кёнигсберга (в настоящее время Данциг называется Гданьск и принадлежит Польше). Переписка с Эйлером, по всей видимости, была одной из инициатив мэра по развитию математики в Пруссии.

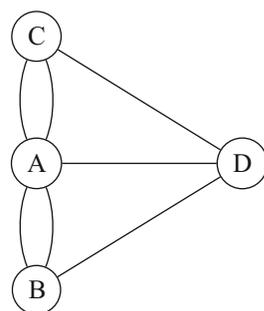
В то время Эйлер жил в Санкт-Петербурге. Он занялся этой задачей и 26 августа 1735 года представил свое решение членам Санкт-Петербургской академии наук. В следующем году Эйлер написал на латыни научный доклад с объяснением своего решения. Ответ на изначально поставленный вопрос был *отрицательным*: по городу нельзя было прогуляться так, чтобы перейти каждый мост всего один раз. Это не просто занимательный исторический факт: решив эту задачу, Эйлер положил начало целому новому разделу математики, который посвящен *графам*.

Но, прежде чем переходить к графам, давайте посмотрим, как Эйлер подошел к решению этой задачи. Для начала он сделал ее максимально абстрактной. Для ее представления не требуется подробной карты Кёнигсберга. Эйлер нарисовал следующую диаграмму.



Он обозначил два острова буквами A и D, а два берега континентальной части — буквами B и C. Эту диаграмму можно сделать еще более абстрактной, избавившись от физической геометрии, и свести все к связям между мостами, островами и континентальной частью, так как именно это имеет отношение к задаче.

Мы обозначили части суши окружностями и соединили их линиями — мостами. Теперь



задачу можно переформулировать следующим образом: если у вас есть карандаш, возможно ли приложить его к любому кругу и, не отрывая его от бумаги, пройтись им по каждой линии ровно один раз?

Эйлер предложил следующее решение. При посещении суши вы должны ее покинуть, за исключением тех случаев, когда это начало и конец вашей прогулки. Для этого каждая часть суши, если не считать начала и конца, должна иметь четное количество мостов, чтобы вы могли зайти на нее по одному мосту, а выйти по другому. Теперь взгляните на рисунок и посчитайте количество мостов, соединяющих каждую часть суши. Вы обнаружите, что в каждом случае это число нечетное: у A пять мостов, а у B , C и D их три. Где бы вы ни решили начать и закончить свою прогулку, у вас все равно останется две части суши, которые вам нужно посетить, и каждая из них имеет нечетное количество мостов. Мы не можем пройтись по каждому из этих мостов всего один раз.

Действительно, если в какой-то момент нашей прогулки мы посетим B , это означает, что мы уже прошли по мосту, который ведет к этой части суши. Чтобы ее покинуть, мы переходим по второму мосту. Позже нам нужно будет пройти по третьему мосту, так как этого требует условие. Но в этом случае мы застрянем в B , так как четвертого моста не существует, а мосты, которые мы уже посетили, нельзя переходить повторно. То же самое относится к C и D , у которых также по три моста. Но тот же принцип применим и к точке A , если она является промежуточной: после перехода по всем ее пяти мостам мы не сможем ее покинуть, так как шестого моста попросту нет.

Наша диаграмма состоит из кругов и линий, которые их соединяют. Используя правильную терминологию, мы создали структуру, составленную из узлов, или вершин, соединенных ребрами, или звеньями. Структура, состоящая из множества узлов и ребер, называется графом; Эйлер был первым, кто начал считать графы структурами и исследовать их свойства. Говоря современным языком, задача о кёнигсбергских мостах имеет отношение к путям; путь в графе — это последовательность ребер, которые соединяют последовательность узлов. Таким образом данная задача сводится к поиску эйлерова пути: маршрута, который проходит через каждую вершину графа ровно один раз. Путь, начинающийся и заканчивающийся в одном и том же узле, называется циклом. Если совпадают начало и конец эйлерова пути, получается эйлеров цикл.

Графы имеют настолько обширное применение, что им посвящены целые книги. Все, что можно смоделировать в виде узлов, соединенных с другими узлами, подходит под определение графа. Это позволяет задавать

всевозможные интересные вопросы; здесь эта тема затрагивается лишь поверхностно.

Но, прежде чем двигаться дальше, упомянем об одной детали, чтобы удовлетворить самых скрупулезных читателей. Мы упоминали о том, что граф — это структура, состоящая из множества вершин и ребер. В математике элементы множества не повторяются. Но в нашем представлении Кёнигсберга некоторые ребра используются больше одного раза — например, те, что соединяют A и B . Ребро определяется его начальной и конечной точками, поэтому два моста между A и B на самом деле являются двумя экземплярами одного ребра. Таким образом, совокупность ребер является не множеством, а *мультимножеством*, которое допускает дублирование элементов. Поэтому диаграмма мостов Кёнигсберга — это не граф, а, скорее, *мультиграф*.

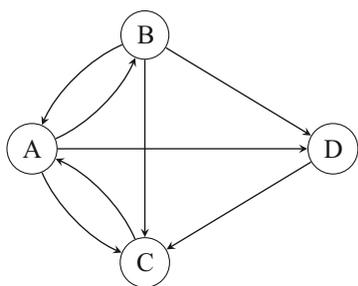
От графов к алгоритмам

Довольно общее определение графа охватывает все, что можно описать в виде объектов, соединенных с другими объектами. Граф также имеет некоторое отношение к топологии местности, но узлы и звенья могут существовать в отрыве от пространства.

В качестве примера графа можно привести *социальную сеть*. Узлами в социальной сети выступают социальные субъекты (это могут быть люди или организации), а звенья представляют взаимодействия между ними. Роль социальных субъектов могут играть настоящие актеры, а звенья — их совместные съемки в фильмах. Мы сами можем быть социальными субъектами, а звенья могут быть представлены нашими связями с другими людьми на сайте социальной сети. Социальную сеть можно использовать для поиска сообществ, состоящих из людей, которые взаимодействуют между собой. Существуют алгоритмы, способные эффективно находить сообщества в графах с миллионами узлов.

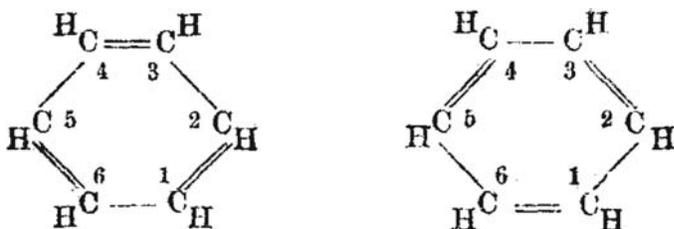
Ребра в кёнигсбергском графе не направленные, то есть их можно проходить в обоих направлениях; например, мы можем перейти из A в B и из B в A . То же самое относится и к социальным сетям, но только если связи взаимные, что происходит не всегда. В зависимости от области применения ребра в графе могут быть направленными. В этом случае они обозначаются стрелками. Направленный граф еще называют *ориентированным* (или *орграфом* для краткости). Пример орграфа показан ниже. Обратите внимание на то, что это не мультиграф; ребро, идущее из A в B , отличается от ребра, которое идет из B в A .

Довольно общее определение графа охватывает все, что можно описать в виде объектов, соединенных с другими объектами.



Еще один (огромный) пример графа — Всемирная паутина. Ее можно представить в виде узлов (веб-страниц) и ребер (гиперссылки, соединяющих каждую пару страниц). Это направленный граф, поскольку страница, на которую ссылаются, может не содержать обратной ссылки.

Если граф можно обойти, вернувшись в исходный узел, это означает, в нем есть *цикл*. Циклы имеют не все графы. У кёнигсбергского графа есть цикл, хоть и не эйлеров. В истории науки известен знаменитый циклический граф (на самом деле мультиграф): модель молекулярной структуры бензола, которую создал Август Кекуле:



Граф, у которого нет цикла, называется *ациклическим*. Направленные ациклические графы относятся к отдельной важной категории. Их обычно обозначают аббревиатурой DAG (*от англ. directed acyclic graph*). У них есть много применений; например, они используются для представления приоритетов выполнения задач (задачи — узлы, а приоритеты — ребра между ними), зависимостей, предварительных условий и других похожих конструкций. Пока оставим ациклические графы и займемся циклическими; это будет наше первое знакомство с алгоритмами для работы с графами.

Пути и ДНК

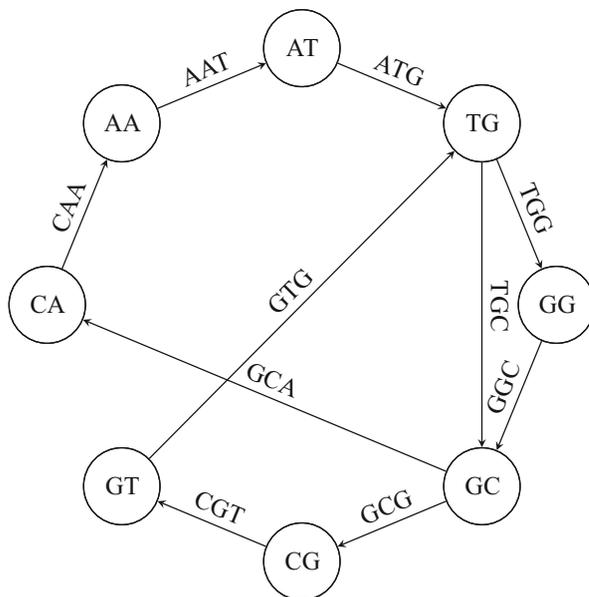
Одним из самых важных научных достижений последних десятилетий была расшифровка человеческого генома. Благодаря методикам, созданным в ходе

этой работы, мы теперь можем исследовать генетические заболевания, распознавать мутации, изучать геномы вымерших видов и многое другое.

Геномы закодированы в ДНК — большой органической молекуле в форме двойной спирали. Эта двойная спираль состоит из четырех нуклеотидов: цитозина (C), гуанина (G), аденина (A) и тимина (T). Каждая спираль представляет собой последовательность нуклеотидов, такую как ACCGTATAG. Соответствующие нуклеотиды, находящиеся в разных спиральях, связаны между собой по принципу A-T и C-G. Таким образом, если одна спираль имеет вид ACCGTATAG, то другая выглядит как TGGCATATC.

Определить структуру неизвестного участка ДНК можно следующим образом. Мы создаем множество копий цепочки и разбиваем их на мелкие фрагменты — например, по три нуклеотида в каждом. Эти фрагменты легко распознать с помощью специальных инструментов. В итоге мы получаем набор известных фрагментов. Теперь нам остается собрать их в последовательность ДНК, чью структуру мы уже будем знать.

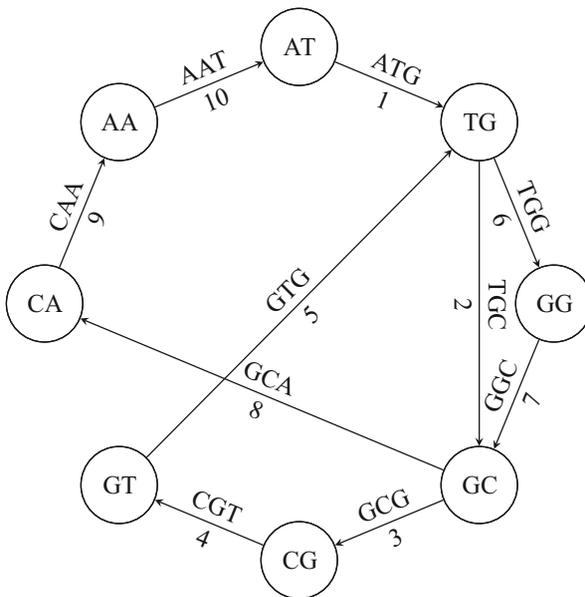
Представьте, что у нас есть следующие фрагменты (или, как их называют, полимеры): GTG, TGG, ATG, GGC, GCG, CGT, GCA, TGC, CAA и AAT. Все они тройные; чтобы получить последовательность ДНК, которую они изначально составляли, создадим граф. Вершинами в этом графе будут двойные полимеры, выведенные из тройных путем отбора первых двух и последних двух нуклеотидов. Таким образом из GTG получается GT и TG, а из TGG — TG и GG. Каждая пара вершин, полученная из исходного тройного полимера, соединяется ребром. Мы присваиваем название полимера этому ребру. То есть из ATG получаются вершины AT и TG, а также ребро ATG. Итоговый граф показан ниже.



Теперь нам остается только найти путь в этом графе, который проходит по каждому ребру ровно один раз (то есть эйлеров цикл), чтобы получить исходную цепочку ДНК. В 1873 году Карл Хирхольцер опубликовал алгоритм для поиска эйлеровых циклов, названный в его честь. Выглядит он так.

1. Выбираем начальный узел.
2. Идем от узла к узлу, пока не вернемся в начало. Пройденный нами путь может охватывать не все ребра.
3. Если существует вершина, которая является частью пути, но при этом принадлежит ребру, которое мы не проходили, мы начинаем с нее еще один путь, используя ранее не пройденные ребра, пока не возвращаемся в исходную точку. Таким образом у нас получается еще один замкнутый путь. Затем мы совмещаем его с путем, пройденным ранее.

Если использовать этот алгоритм для нашего графа, получится следующий путь.



Мы начинаем в AT и проходим по маршруту AT → TG → GG → GC → CA → AA → AT. Маршрут пройден, но он не охватывает все ребра. Например, у TG есть ребро TGC, которое мы еще не покрыли. Поэтому мы переходим в TG и начинаем еще один маршрут с ребра TGC; в результате получается TG → GC → CG → GT → TG. Совместим этот путь с первым и получим маршрут, представленный

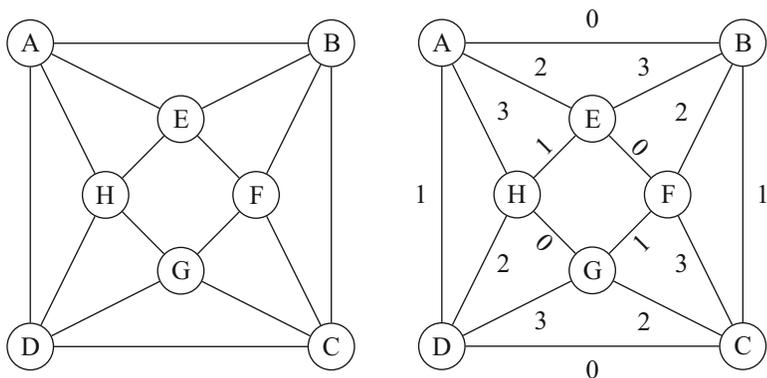
на рисунке, $AT \rightarrow TG (\rightarrow GC \rightarrow CG \rightarrow GT \rightarrow TG) \rightarrow GG \rightarrow GC \rightarrow CA \rightarrow AA \rightarrow AT$. Если пройти его от начала и до конца, не доходя до последнего узла, и соединить вершины с общими нуклеотидами ровно по одному разу, получится цепочка ДНК $ATGCGTGGCA$. Вы можете убедиться в том, что она содержит все полимеры, с которых мы начинали; CAA и CAT находятся, если при достижении последней вершины перейти в самое начало.

В этом конкретном примере мы нашли всего один дополнительный циклический путь, который затем был совмещен с исходным. Но таких путей может быть больше; шаг 3 повторяется до тех пор, пока не закончатся вершины с непокрытыми ребрами. Алгоритм Хирхольцера довольно быстрый: если его как следует реализовать, он выполняется за линейное время, $O(n)$, где n — количество ребер в графе.

Планирование турнира

Представьте, что вы занимаетесь организацией турнира, участники которого будут состязаться попарно в цепочке матчей. У нас есть восемь участников, каждый из которых сыграет в четырех матчах. Нам нужно создать такой график соревнований, чтобы у каждого участника оказалось по одному матчу в день.

Очевидное решение заключается в том, чтобы ограничить весь турнир одним матчем в день, растянув его настолько, насколько потребуется. У нас есть восемь участников, каждый из которых играет в четырех матчах, поэтому турнир займет 16 дней ($8 \times 4 / 2$; мы делим на два, чтобы не считать каждый матч дважды). Назовем наших участников Элис (A), Боб (B), Кэрл (C), Дэйв (D), Иви (E), Фрэнк (F), Грейс (G) и Хайди (H). В скобках указаны буквы, которыми мы будем их обозначать.



Мы можем найти лучшее решение, если смоделируем эту задачу в виде графа. Каждый игрок будет представлен отдельной вершиной, а каждый матч — ребром. В результате получится граф, показанный в левой части рисунка (см. выше). Справа рядом с ребрами указаны дни, на которые запланированы соответствующие матчи. Как мы получили это решение?

Мы решили пронумеровать дни последовательно. Пусть турнир начинается в нулевой день и все матчи проходят один за другим.

1. Берем матч, который мы еще не запланировали. Если все матчи уже запланированы, останавливаемся.
2. Планируем матч на ближайший день, но так, чтобы ни у одного из игроков не было в этот день других матчей. Возвращаемся к шагу 1.

Этот алгоритм выглядит настолько просто, что вы можете даже засомневаться, решает ли он нашу задачу. Это обманчивое впечатление. Давайте пройдемся по нему и посмотрим, что получится. В таблице ниже указаны матчи, следующие один за другим, и дни, на которые они запланированы, — все в соответствии с нашим алгоритмом. Сначала читаем первые два столбца, а затем другие два.

Матч	День	Матч	День
A, B	0	C, F	3
A, D	1	C, G	2
A, E	2	D, G	3
A, H	3	D, H	2
B, C	1	E, F	0
B, E	3	E, H	1
B, F	2	F, G	1
C, D	0	G, H	0

В первом матче встречаются Элис и Боб. Ни один из этих игроков больше не будет участвовать в турнире в нулевой день (в день, на который мы планируем матч).

Берем еще один матч, который пока не был запланирован, — скажем, Элис против Дэйва. Мы отбираем игроков в алфавитном порядке их однобуквенных обозначений, но имейте в виду, что порядок может быть любой, даже

случайный; главное, чтобы матчи не повторялись. У Элис уже есть матч, запланированный на эту дату, поэтому ближайший свободный день — первый.

Дальше идет матч между Элис и Иви. Элис уже занята в первые два дня, поэтому запланируем матч на день 2. Свой заключительный матч Элис проведет против Хайди; дни 0, 1 и 2 распланированы, поэтому данное событие должно произойти в день 3.

С Элис мы закончили. Переходим к матчам Боба и сразу пропускаем тот, который у него должен быть с Элис, так как мы его уже запланировали. Берем матч Боба против Кэрол. Боб уже занят в нулевой день (матч с Элис), поэтому матч будет проведен в день 1. При планировании матча против Иви мы видим, что Боб уже занят в дни 0 и 1, а Иви играет против Элис в день 2, поэтому выбираем день 3. Берем матч Боба против Фрэнка; у Боба есть матчи в дни 0, 1 и 3, а Фрэнк полностью свободен, поэтому выбираем день 2 (на день раньше матча Боба против Иви).

После Боба мы переходим к Кэрол. Ни у Кэрол, ни у Дэйва нет матчей в нулевой день, поэтому они сразятся в первый день турнира. Матч Кэрол против Фрэнка пройдет в день 3, поскольку Кэрол состязается в дни 0 (только что запланировали) и 1 (с Бобом), а у Фрэнка есть матч с Бобом. День 2 (это мы тоже запланировали ранее). Матч Кэрол против Грейс пройдет *раньше*, в день 2, так как Грейс все еще свободна, а у Кэрол нет матчей в этот день.

Аналогичным образом планируются оставшиеся матчи; интересно, что матчи вдоль внешнего и внутреннего прямоугольников графа состоятся в течение первых двух дней. У нас получилось две группы игроков, которые сначала состязаются параллельно, а затем встречаются между собой. В итоге найденное нами решение существенно превосходит простейший вариант с 16-дневным графиком; нам достаточно всего четырех дней!

На самом деле планирование турнира — это разновидность более общей задачи: *раскраски ребер*. Эта задача состоит в распределении цветов между ребрами таким образом, чтобы никакие два смежных ребра не имели один и тот же цвет. Конечно, цвет здесь — образное понятие. В нашем примере вместо цветов использовались номера дней; в целом это может быть любой другой набор уникальных значений. Если вместо ребер раскрашивать вершины и делать это так, чтобы никакие две из них, соединенные общим ребром, не имели один и тот же цвет, получится задача *раскраски вершин*. Как вы уже могли догадаться, обе эти задачи принадлежат к более общей категории, известной как задача *раскраски графа*.

Алгоритм раскраски ребер, описанный выше, отличается простотой и эффективностью (он последовательно проходит по каждому ребру, и делает

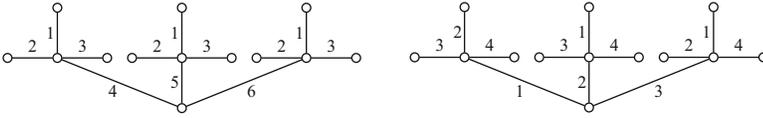
это всего один раз). Алгоритмы такого рода называют *жадными*. Они пытаются решить задачу путем поиска оптимального решения *на каждом этапе*, не заботясь об общем результате. Жадные алгоритмы подходят для многих случаев, в которых на каждом этапе необходимо принимать решение, «оптимальное в текущий момент». Стратегии, которыми руководствуется алгоритм при принятии решений, называются *эвристиками* от греч. εὐρίσκω — «искать» [решение].

Но, если немного подумать, такой недальновидный способ принятия решений может оказаться не лучшим. Иногда имеет смысл проявить некоторую сдержанность; выбор, кажущийся оптимальным в текущий момент, может привести к последствиям, о которых мы позже пожалеем. Представьте, что вы взбираетесь на гору. Если следовать жадной эвристике, на каждом этапе нужно выбирать самый крутой подъем (предположим, что вы опытный скалолаз). Но так не всегда можно добраться до вершины: есть вероятность очутиться на плоскогорье, с которого придется спускаться обратно. Настоящий путь к вершине может пролегать по более пологим склонам.

Метафору со скалолазанием часто используют при решении задач в информатике. Задача моделируется так, чтобы решение находилось на «вершине», к которой ведут разные потенциальные действия; идея в том, чтобы найти правильное сочетание действий. Этот подход называется *поиск восхождением к вершине*. Плато в таком случае является *локальным оптимумом*, а вершина, на которую мы взбираемся, — *глобальным оптимумом*.

Вернемся от скалолазания к планированию турнира. Мы выбрали ближайший свободный день для каждого матча. К сожалению, это может быть не лучшим способом планирования. Действительно, раскраска графа — сложная задача. Составленный нами алгоритм *не* гарантирует получение оптимального результата — такого, для которого требуется наименьшее количество дней (или цветов, если говорить в целом). У узла графа есть *степень* — число выходящих из него ребер. Мы можем доказать, что в случае, если наивысшая степень любого узла в графе равна d , для раскраски всех ребер достаточно d или $d + 1$ цветов; количество цветов, необходимое для раскраски ребер, называется *хроматическим индексом* графа. В конкретном примере наше решение является оптимальным, $d = 4$ (четыре дня). Но в других графах алгоритм может выдать не лучший результат. Плюс жадной раскраски графа в том, что мы знаем, насколько сильно решение способно отклониться от оптимума: потенциальное количество цветов может превышать d и достигать $2d - 1$, но не более того.

Если вам интересно, как это может произойти, взгляните на граф, который состоит из «звезд», соединенных с центральным узлом.



Если число звезд равно k и у каждой звезды есть k ребер, не считая того, которое ведет к центральному узлу, для раскраски ребер звезд понадобится k цветов. Нам нужно еще k цветов, чтобы соединить звезды с центральным узлом. Всего получается $2k$ цветов. Это то, что было сделано слева. Но это не идеальное решение. Если мы начнем с раскраски ребер, соединяющих звезды с центральным узлом, нам хватит k цветов. А для раскраски самих звезд будет достаточно одного дополнительного цвета. Итого $k + 1$ цветов. Этот подход проиллюстрирован справа. Все это соответствует теории, так как степень каждой звезды равна $k + 1$.

Но проблема в том, что жадный алгоритм выбирает порядок раскраски ребер, который в итоге оказывается неоптимальным (или, если быть более точным, *не глобально оптимальным*). Он может выдать лучшее решение, а может и нет. Но, опять же, отклонение от оптимального решения не такое уж большое. Это смягчает проблему, так как задача раскраски графа очень сложная, и алгоритм, который находит лучшее решение в *каждом* случае, имеет экспоненциальную сложность, около $O(2^n)$, где n — количество ребер в графе. Такие точные алгоритмы зачастую применимы только для крошечных графов.

У жадного алгоритма, который мы здесь представили, есть одна замечательная особенность (если не считать его практичность). Это *онлайн-алгоритм*. Он работает, даже если ввод известен не с самого начала, а предоставляется по ходу продвижения вперед. Для его запуска не нужно знать обо всех ребрах. Алгоритм будет работать правильно, даже если граф генерируется прямо на лету, по одному ребру за раз. Это может произойти, если игроки продолжают записываться для участия в турнире даже после того, как мы начали планировать матчи. Мы можем раскрашивать каждое ребро (матч) при его появлении, и по завершении создания графа он уже будет раскрашен. Более того, если граф создается постепенно, этот жадный алгоритм оказывается оптимальным; для графа, который формируется прямо в процессе решения задачи, не существует точного алгоритма, даже самого неэффективного.

Кратчайшие пути

Как мы уже видели, жадный алгоритм принимает лучшее решение на каждом этапе, что может не привести к лучшему итоговому результату. Можно сказать, что он имеет оппортунистический, недальновидный характер. К сожалению, как мы знаем из басни Эзопа, кузнечик, живущий сегодняшним днем, может пожалеть о своем решении зимой, а муравей, который думает о будущем, окажется в тепле и уюте. Но, если говорить о планировании турнира, участь кузнечика может быть не так уж печальной. Теперь давайте посмотрим, что на это ответит муравей.

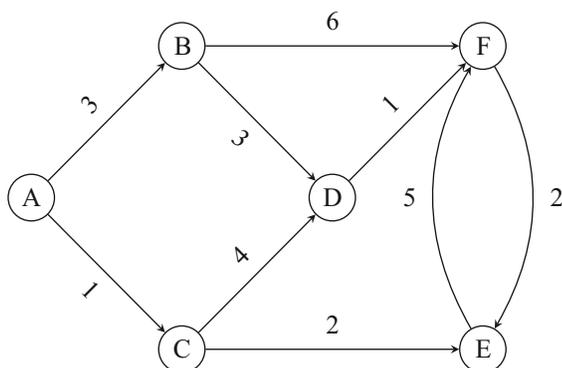
В главе 1 мы отмечали, что поиск кратчайшего пути между двумя точками на сетке лучше не делать путем перебора всех возможных вариантов. Вы сами видели, что на практике этот подход неосуществим, так как количество путей растет огромными темпами. Но теперь, когда мы познакомились с графами, эта задача кажется не такой безнадежной. Давайте поднимемся на уровень выше. Сетка имеет простую геометрию с равными расстояниями между соседними точками; давайте исходить из того, что мы имеем дело с сеткой любой формы, точки в которой могут находиться на разных расстояниях друг от друга.

Для этого мы создадим граф, узлы и ребра которого представляют карту, и попробуем найти кратчайший путь между двумя его узлами. Кроме того, каждому ребру будет назначен *вес*. Это нулевое или положительное значение, соответствующее расстоянию между двумя соединенными узлами. Расстояние может измеряться в километрах или времени, которое занимает его прохождение; подойдут любые неотрицательные величины. Таким образом *длина пути* будет суммой весов всех его ребер; *кратчайшим путем* между двумя узлами является тот, который имеет наименьшую длину. Если все веса равны 1, длина пути равна количеству ребер, из которых он состоит. Если допускаются другие значения, это утверждение неверно.

Следующий граф состоит из шести узлов, соединенных девятью ребрами с разными весами. Мы хотим найти кратчайший путь из А в F.

Если применить жадную эвристику, вначале мы перейдем из А в С, затем лучшим выбором будет Е, а оттуда уже можно попасть в F. Общая длина пути А, С, Е и F равна 8, что не является оптимальным решением. Лучше всего пройти из А в С, затем в D и, наконец, в F; длина этого варианта — 6. Итак, жадная эвристика не работает, и, в отличие от примера с планированием турнира, мы не знаем, насколько хуже ее результат может быть по сравнению с настоящим кратчайшим путем. Тем не менее и, опять же, в отличие от планирования

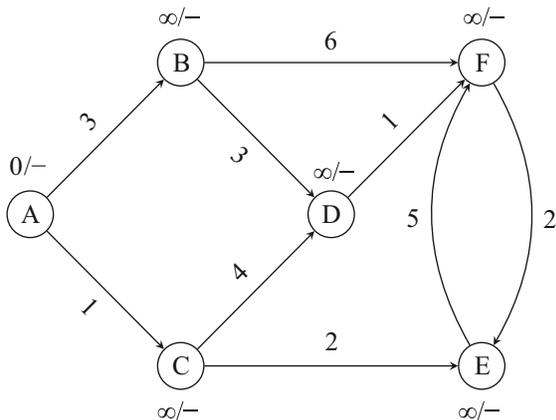
турнира, у задачи поиска кратчайшего пути существует эффективный алгоритм, поэтому нет никакого смысла использовать жадную эвристику.



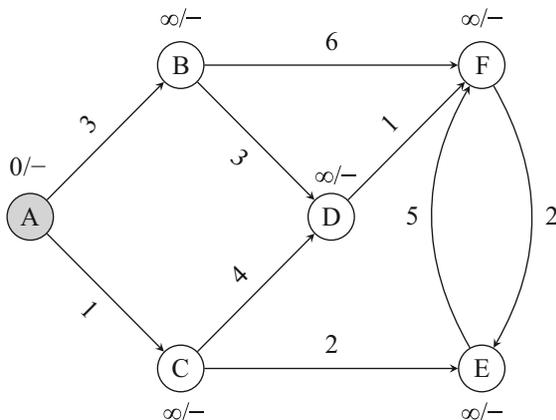
В 1956 году молодой голландский информатик, Эдсгер Дейкстра, делал покупки в амстердамских магазинах вместе со своей невестой. Устав, они присели на террасе кафе, чтобы выпить чашечку кофе. Дейкстра начал обдумывать задачу поиска лучшего пути из одного города в другой. Он набросал решение за 20 минут, хотя сам алгоритм был опубликован лишь спустя три года. Дейкстра имел блестящую карьеру, но, к его удивлению, именно это 20-минутное озарение сделало его знаменитым.

Так как же выглядит этот алгоритм? Мы хотим найти кратчайшие пути из одного узла графа ко всем другим. В этом алгоритме используется принцип *релаксации*: значениям, которые нужно найти (в данном случае расстояниям), дается приблизительная оценка. Вначале наши оценки максимально плохие. Затем по мере работы алгоритма они постепенно улучшаются, пока мы не получим правильные значения.

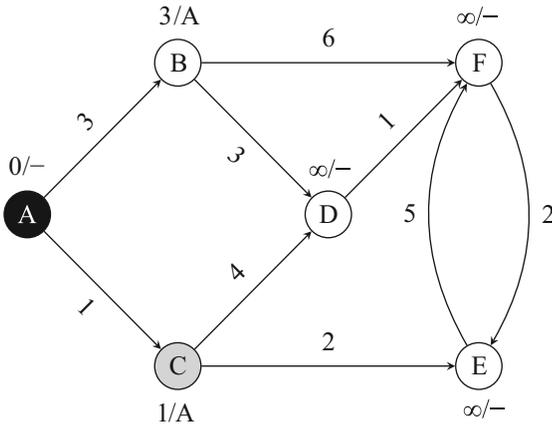
В алгоритме Дейкстры релаксация происходит следующим образом. Вначале мы присваиваем расстояниям от одного узла ко всем остальным наихудшее из возможных значений: бесконечность; очевидно, что длиннее ничего быть не может! На следующем рисунке мы указали рядом с узлами начальную оценку кратчайшего пути и предыдущего узла в нем. Для узла A это 0/-, поскольку расстояние от A к A равно нулю, и у A нет предыдущего узла. Рядом со всеми остальными узлами указано ∞/-, потому что расстояние бесконечно, и мы не знаем, какой путь к ним является кратчайшим.



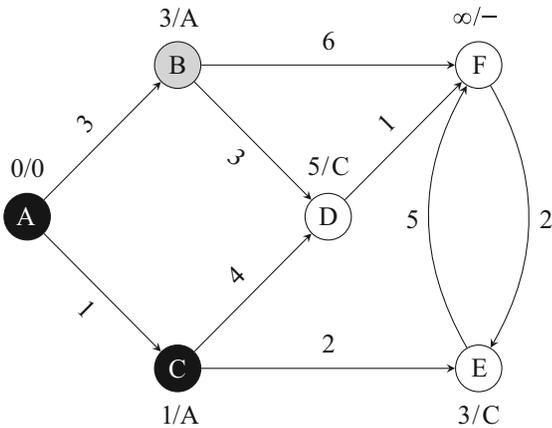
Мы берем узел с кратчайшим расстоянием от A (о котором нам известно). Это сам узел A. Он текущий, поэтому выделим его серым.



Дальше мы можем проверить оценочное расстояние от узла A к его соседям, B и C. Изначально мы сделали их бесконечно удаленными, но теперь оказывается, что путь к B равен 3, а к C — 1. Обновим наши оценки и выразим их через A: укажем 3/A над B и 1/A под C. К узлу A мы больше не вернемся. Обновим нашу диаграмму соответствующим образом, выделив A черным цветом. Перейдем к узлу с лучшей оценкой, который еще не посещался, то есть к C.

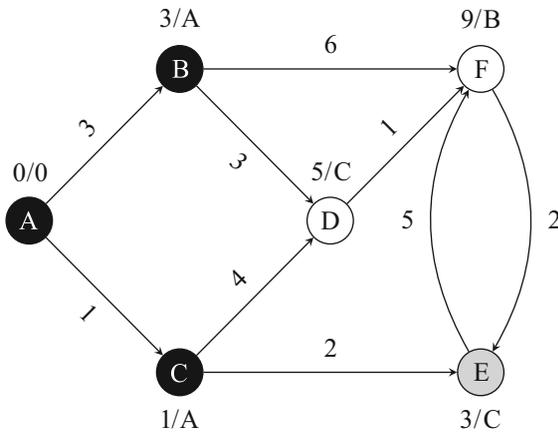


Находясь в узле С, проверяем оценки кратчайших путей к его соседям, D и E. Обе они бесконечные, но теперь мы видим, к каждому из этих узлов можно добраться через С. Путь от А до D через С имеет общую длину 5, поэтому записываем 5/С над D. Общая длина пути от А к E через С равна 3, поэтому указываем 3/С под E. С узлом С мы закончили, поэтому выделяем его черным и переходим к еще не посещенному узлу с лучшей текущей оценкой. Оценки узлов В и E одинаково хороши — 3. Мы можем выбрать любой из них. Пусть это будет В.

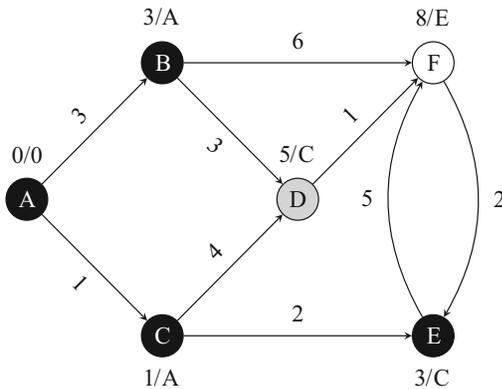


Продолжаем в том же духе. Находясь в узле В, проверяем оценки кратчайших путей к его соседям, D и F. Мы уже знаем, что путь к D из С равен 5; это лучше, чем если бы мы начинали из В (6). Поэтому оценка удаленности D остается без изменений. Путь к F сейчас считается бесконечно длинным,

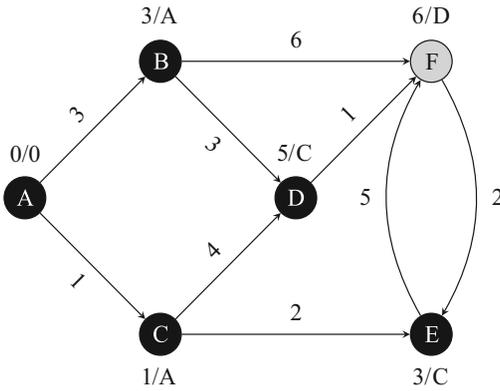
поэтому мы обновляем эту оценку до 9. Помечаем узел В как посещенный и переходим к узлу с лучшей оценкой, который мы еще не проверяли, то есть к Е.



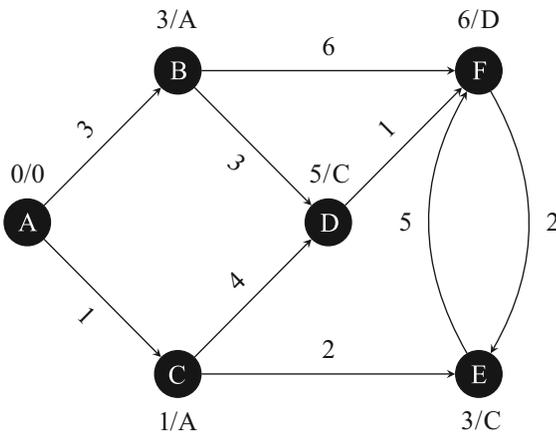
Е соседствует с F. Путь к F из E имеет длину 8; это лучше, чем путь, пролегающий через В. Обновим нашу оценку, пометим узел Е как посещенный и перейдем к следующему узлу с лучшей оценкой, который мы еще не посетили, то есть к D.



D соседствует с узлом F, к которому, как мы обнаружили, можно добраться через E по пути длиной 8. Путь к F через D равен 6, поэтому мы обновляем нашу предыдущую оценку. Как и прежде, переходим к узлу с лучшей оценкой, который мы еще не проверяли, то есть к F.



Находясь в узле F, проверяем, нужно ли обновить оценку для его соседа, E. Текущий путь к E имеет длину 3, тогда как путь через F равен 10. Оставляем E без исправлений. Посещение F ничего не изменило, но мы не могли знать это заранее. Мы посетили все узлы, поэтому алгоритм завершается.



Выполняя алгоритм, мы записывали длину путей и предшественника каждого узла. Мы делали это для того, чтобы по окончании алгоритма можно было найти кратчайший путь из A в любой другой узел графа; например, если взять F, мы начинаем с конца и двигаемся в начало. Узнаем предыдущий узел: D. Потом узнаем предшественника D, C, затем предшественника C, A. Кратчайший путь из A в F выглядит как A, C, D, F и имеет общую длину 6; мы уже упоминали об этом в начале нашего обсуждения.

В конце алгоритм Дейкстры нашел *все кратчайшие пути* из начального узла графа во все остальные. Он эффективен, так как его сложность равна $O((m + n)\log n)$, где m — количество ребер в графе, а n — количество узлов. Вот из каких шагов он состоит:

1. Назначить бесконечное расстояние всем узлам, кроме начального; назначить начальному узлу нулевое расстояние.
2. Найти узел на минимальном расстоянии, который еще не посещали. Это будет наш текущий узел. Если все узлы уже посещались, останавливаемся.
3. Проверяем всех соседей текущего узла. Если расстояние к соседу больше того, которое получилось бы при прохождении к нему через текущий узел, мы уменьшаем расстояние и обновляем путь, ведущий к соседу. Переходим к шагу 2.

Если нас интересует кратчайший путь только к одному конкретному узлу, мы можем остановиться после того, как этот узел будет выбран для посещения в шаге 2. На этом этапе мы уже знаем кратчайший путь к этому узлу, и он не поменяется в ходе дальнейшей работы алгоритма.

Алгоритм Дейкстры можно использовать в любом графе, независимо от того, направленный он или нет, даже если в нем есть циклы; главное, чтобы у него не было отрицательных весов. Это может произойти, если ребра между узлами представляют какого-то рода награды и штрафы. Хорошая новость в том, что для работы с отрицательными весами есть другие эффективные алгоритмы, но это подчеркивает, что у алгоритма может быть ограниченное применение. При подборе алгоритма для решения задачи необходимо убедиться в том, что наша задача удовлетворяет его требованиям. В противном случае он не будет работать; но имейте в виду, что сам алгоритм не может сообщить о том, что он не подходит. Если мы реализуем его на компьютере, он будет выполнять свои шаги, даже если в этом нет никакого смысла. Это даст бредовый результат. Мы сами должны заботиться о выборе подходящих инструментов для наших задач.

Давайте возьмем крайний случай: граф, у которого есть не только отрицательные веса, но и цикл с отрицательной суммой ребер (отрицательный цикл). У такого графа *нет алгоритма* для поиска кратчайших путей, поскольку *их не существует*. Если у нас есть отрицательный цикл, мы можем проходить по его ребрам раз за разом, и на каждой итерации длина пути будет уменьшаться. Это может продолжаться сколько угодно, и путь вдоль этого цикла получится бесконечно отрицательным.

При подборе алгоритма для решения задачи необходимо убедиться в том, что наша задача удовлетворяет его требованиям. В противном случае он не будет работать; но сам он не может об этом сообщить. Среди программистов и специалистов в компьютерных науках бессмысленные программы принято называть «мусор на входе — мусор на выходе». Люди должны сами выявлять такой мусор и понимать, что и в каких ситуациях использовать. Это является важной частью университетских курсов по алгоритмам.

Тот факт, что алгоритмы могут делать все, что угодно, от перевода текста до вождения автомобиля, создает обманчивое впечатление о том, для чего их в основном применяют. Реальность оказывается прозаичной. Сложно найти компьютерную программу, которая делает что-либо полезное без использования алгоритмов поиска по данным.

Это вызвано тем, что поиск в том или ином виде присутствует почти в любом контексте. Программе, принимающей данные, зачастую нужно что-то в них найти, и для этого почти наверняка используется поисковый алгоритм. Поиск — это не только очень распространенная, но и трудоемкая операция, которую необходимо выполнять постоянно. Хороший поисковый алгоритм может кардинально повысить скорость работы программы.

Поиск подразумевает выбор определенного элемента из группы элементов. Это общее определение включает в себя несколько разновидностей поиска. Большую роль играет то, упорядочены ли элементы, по которым мы ищем. Совсем другая ситуация возникает, когда мы получаем элементы один за другим и должны сразу же распознать тот, который ищем, без возможности пересмотреть наше решение. Если мы неоднократно выполняем поиск по множеству элементов, важно знать, являются ли какие-либо из них популярнее других. Мы исследуем все эти разновидности поиска в данной главе, но имейте в виду, что существуют и другие. Например, мы будем рассматривать только задачи с *точным поиском*, но во многих приложениях требуется *приблизительный поиск*. Возьмем, к примеру, проверку орфографии: когда вы делаете опечатку, программа должна найти слова, похожие на те, которые ей не удалось распознать.

По мере увеличения объемов данных все важнее становится возможность эффективного поиска по огромному количеству элементов. Вы увидите, что поиск масштабируется чрезвычайно хорошо, если наши элементы упорядочены. В главе 1 мы утверждали, что найти что-то среди миллиарда

Поиск в том или ином виде присутствует почти в любом контексте... Хороший поисковый алгоритм может кардинально повысить скорость работы программы.

отсортированных элементов можно примерно за 30 шагов; теперь же мы покажем, как этого добиться.

Эта глава даст нам представление об опасностях, которые поджидают нас при реализации алгоритма в виде компьютерной программы, рассчитанной на работу в рамках определенного устройства.

Иголка в стоге сена

Простейший пример поиска — это попытка найти иголку в стоге сена из известной поговорки. Если мы хотим найти что-то в группе объектов, которые никак не структурированы, нам остается только проверить их один за другим, пока не будет найдено то, что мы ищем, или пока закончатся все элементы.

Если вы ищете определенную игральную карту в колоде, вы можете начать сверху и двигаться вниз, пока вам не попадется нужная карта или пока колода не исчерпается. Вы также можете продвигаться снизу вверх или вытягивать карты из колоды случайным образом. Это ничего принципиально не изменит.

При работе с компьютерами мы обычно имеем дело не с физическими объектами, а с их цифровым представлением. Набор данных на компьютере часто представляют в виде *списка*. Список — это структура данных, содержащая группу элементов, которые следуют один за другим. Обычно считается, что элементы в списке *связаны*, то есть каждый элемент указывает на следующий, и так до самого конца; последний элемент указывает в никуда. Эта метафора недалека от реальности, поскольку для хранения элементов компьютер использует адреса памяти. В *связном списке* каждый элемент содержит две вещи: собственно данные и адрес следующего элемента. Участок памяти, который хранит адрес другого участка, называется *указателем*. Поэтому в связном списке каждый элемент содержит указатель на следующий элемент. Начальный элемент списка называется *головным*. Элементы списка иногда называют *узлами*. Последний узел никуда не указывает (или указывает на *null*: ничто в компьютерной терминологии).

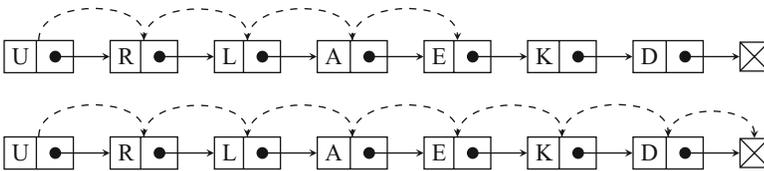
Список — это последовательность элементов, но он не всегда упорядочен по какому-то определенному критерию. Например, следующий список содержит несколько букв из алфавита:



Если список неупорядоченный, поиск элемента в нем будет проходить так.

1. Переходим к головному элементу списка.
2. Если это искомый элемент, сообщаем о его нахождении и останавливаемся.
3. Переходим к следующему элементу списка.
4. Если мы перешли в null, сообщаем о том, что искомый элемент не найден, и останавливаемся. В противном случае повторяем шаг 2.

Это называется *линейным*, или *последовательным*, поиском. В нем нет ничего особенного; это реализация простого перебора элементов, который происходит до тех пор, пока мы не найдем то, что нам нужно. В реальности алгоритм заставляет компьютер переходить от одного указателя к другому, пока не будет обнаружен искомый элемент или null. Ниже показано, как происходит поиск букв *E* и *X*.



Если искать среди n элементов, лучшее, что может случиться, — это когда нам сразу же попадетсЯ искомый элемент (то есть если он является головным). В худшем случае он находится в самом конце или вообще отсутствует в списке. В таком сценарии придется перебрать все n элементов. Таким образом производительность последовательного поиска равна $O(n)$.

С этим ничего нельзя поделать, если элементы собраны в случайную последовательность. Это видно на примере колоды карт: если она как следует перетасована, мы не можем заранее знать, где именно находится та или иная карта.

Иногда это создает проблемы в реальной жизни. Если нам нужно найти что-то конкретное в большой кипе бумаг, этот процесс может оказаться утомительным. Мы даже можем подумать, что из-за нашей невезучести искомый документ находится в самом низу. Поэтому мы прекращаем последовательный перебор и начинаем искать снизу. В этом нет ничего плохого, но не следует надеяться на то, что это как-то улучшит наши шансы и поможет быстрее завершить поиск. Если кipa бумаг никак не упорядочена, искомый документ может находиться где угодно: в начале, в конце или ровно посередине. Все

позиции равновероятны, поэтому продвижение сверху вниз ничем не хуже любой другой стратегии (при условии, что каждый элемент проверяется ровно один раз). Тем не менее выбор определенного порядка помогает отслеживать уже выполненные действия, поэтому мы предпочитаем придерживаться последовательного поиска.

Но все это верно только в том случае, если у нас нет причин полагать, что искомый элемент находится в какой-то определенной позиции. Если ж такие причины есть, мы можем воспользоваться любой дополнительной информацией для ускорения поиска.

Эффект Матфея и поиск

Вы, наверное, замечали, что на плохо организованном рабочем столе некоторые вещи «всплывают» наверх, а другие оказываются в самом низу. Автор был приятно удивлен, когда в процессе наведения порядка обнаружил вещи, которые считал давно утерянными. Это, наверное, случалось со всеми. Мы обычно держим ближе к себе вещи, которые часто используем, а то, что используется редко, постепенно пропадает из нашего поля зрения.

Представьте, что у вас есть кипа документов, над которыми вам нужно работать. Эти документы никак не упорядочены. Когда мы заканчиваем работу с документом, мы кладем его не туда, где он был найден, а наверх. И это повторяется раз за разом.

Может случиться так, что некоторые документы нам требуются чаще других. К некоторым из них мы можем возвращаться снова и снова, а к другим — лишь изредка. Если продолжить откладывать документы, с которыми мы уже поработали, наверх, через некоторое время обнаружится, что самые популярные из них находятся сверху, а те, которые требуются нам реже всего, оказались где-то внизу. Это удобно, поскольку часто используемые документы всегда под рукой, что экономит время.

Вырисовывается общая стратегия поиска, согласно которой одни и те же элементы ищутся многократно, и некоторые из них более востребованы, чем другие. Найдя нужный документ, мы откладываем его наверх, чтобы в следующий раз его было проще искать.

Насколько практичной была бы такая стратегия? Это зависит от того, насколько часто наблюдаются подобные расхождения в популярности элементов. В реальности это случается повсеместно. Вы, наверное, слышали пословицу: «Богатый богатеет, а бедный беднеет». Она касается не только богатых

и бедных людей. Эта тенденция встречается в удивительно большом количестве ситуаций в разных сферах деятельности. У нее даже есть название, *эффект Матфея*, данное в честь следующей цитаты из Евангелия от Матфея (25:29): «...ибо всякому имеющему дастся и приумножится, а у неимеющего отнимется и то, что имеет».

В этом стихе речь идет о материальных благах, поэтому давайте задумаемся на минуту о богатстве. Представьте, что у вас есть большой стадион, способный вместить 80 000 человек. Вы можете измерить средний рост посетителя стадиона. Пусть это будет около 1,7 метра. Представьте, что вы выбрали случайного болельщика и посадили на его место самого высокого человека в мире. Изменится ли средний рост? Даже если найти трехметрового гиганта (такой рост никогда не был зафиксирован), средний рост останется практически неизменным — разница с предыдущим показателем составит десятые доли миллиметра.

Теперь представьте, что вместо среднего роста мы замеряем среднее благосостояние. Допустим, для наших 80 000 человек это будет \$1 миллион (к нам на стадион ходит небедная публика). Теперь опять посадим на место случайного посетителя самого богатого человека в мире, владеющего \$100 миллиардами. Имело ли бы это какое-то влияние? Да, и существенное. Средний показатель вырос бы с \$1 миллиона до \$2 249 987,5 — более чем в два раза. Мы знаем, что богатство распределено неравномерно, но такая разница все равно может удивить. Она намного превышает расхождения естественных величин, таких как рост.

То же самое можно наблюдать во многих других сферах жизни. Есть множество актеров, о которых вы никогда не слышали. А есть горстка звезд, которые постоянно снимаются в фильмах и зарабатывают миллионы долларов. Термин «эффект Матфея» в 1968 году предложил социолог Роберт Мергтон, который заметил, что известные ученые получают большее признание за свою работу, чем их менее популярные коллеги, даже если делают аналогичный вклад. Чем известнее ученый, тем больше внимания он получает.

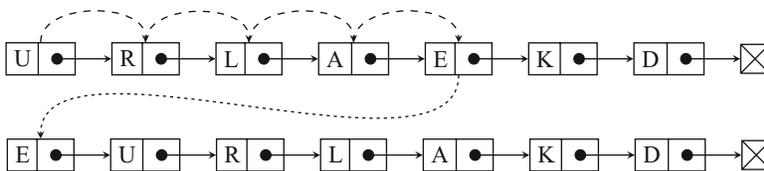
По тому же принципу распределяются слова в языке: некоторые из них используются куда чаще других. Такое очевидное неравенство присуще размерам городов (мегаполисы намного больше среднего города) и количеству/популярности веб-сайтов (на большинство сайтов заходят лишь время от времени, а у некоторых есть миллионы посетителей). За последние несколько лет распространенность таких неравномерных распределений, когда небольшая часть населения владеет непропорционально большим объемом ресурсов,

стала предметом обширных исследований. Ученые пытаются найти причины и законы, которые лежат в основе этого явления.

Возможно, элементам, по которым мы ищем, тоже присуща подобная разница в популярности. Если это так, то поисковый алгоритм, пользующийся этим, может работать по тому же принципу, который мы описывали ранее: каждый найденный документ ложится сверху.

1. Ищем элемент последовательным способом.
2. Если элемент найден, сообщаем о его нахождении, помещаем его в начало списка (делаем головным) и останавливаемся.
3. В противном случае сообщаем о том, что элемент не найден, и останавливаемся.

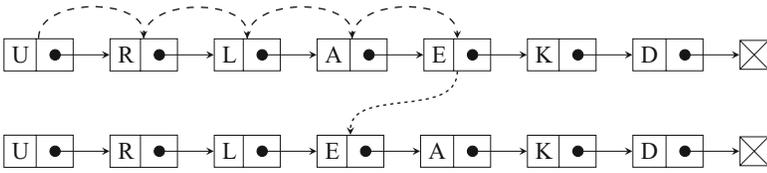
На следующей диаграмме найденный элемент E переносится в начало списка.



Потенциальный недостаток этого алгоритма в том, что в начало списка станут попадать даже те элементы, которые редко ищутся. Это правда, но, если элемент не популярный, он постепенно будет вытеснен ближе к концу при поиске других элементов. Тем не менее мы можем принять определенные меры и сделать нашу стратегию не такой кардинальной. Найденный элемент можно переносить не в самое начало, а лишь на одну позицию вперед. Этот подход называется *методом перегруппировки*.

1. Ищем элемент последовательным способом.
2. Если элемент найден, сообщаем о его нахождении, меняем его местами с предыдущим (если он не первый) и останавливаемся.
3. В противном случае сообщаем о том, что элемент не найден, и останавливаемся.

Таким образом популярные элементы постепенно перейдут в начало списка, а остальные окажутся ближе к концу, без внезапных скачков.



Перемещение в начало и метод перегруппировки являются разновидностями *самоорганизующегося поиска*; этот термин выбран потому, что список организуется по мере выполнения операций поиска и отражает популярность его элементов. Экономия времени зависит от того, насколько сильно варьируется эта популярность. Если от последовательного поиска можно ожидать производительности вида $O(n)$, то самоорганизующийся поиск с перемещением в начало может демонстрировать скорость порядка $O(n/\lg n)$. Если у нас есть миллион элементов, это позволит обработать их так же быстро, как если бы их было около 50 000. Метод перегруппировки может давать еще лучшие результаты, но для их достижения ему нужно больше времени. Это обусловлено тем, что оба метода требуют «прогрева», в течение которого выявляются популярные элементы. В алгоритмах с перемещением в начало этот период короткий; при использовании метода перегруппировки прогрев занимает больше времени, но зато потом получаются лучшие результаты.

Кеплер, автомобили и невесты

В 1611 году прославленный астроном Иоганн Кеплер (1571–1630) овдовел — его жена умерла от холеры. Он решил жениться повторно, но, будучи методическим человеком, не захотел полагаться на волю случая. Свой подход он описал в письме барону Страхлендорфу. Прежде чем делать выбор, Кеплер планировал провести собеседования с 11 потенциальными невестами. Ему очень понравилась кандидатура под номером пять, но друзья его переубедили, указав на ее низкое положение в обществе. Вместо нее они посоветовали еще раз присмотреться к четвертой кандидатке, но она ему отказала. В итоге после знакомства со всеми 11 претендентками, Кеплер все же женился на пятой, 24-летней Сюзанне Рюттингер.

Эта небольшая история является немного притянутым за уши примером поиска; Кеплер искал идеальную пару среди потенциальных кандидатур. Но у этого процесса была одна особенность, которую он мог сначала

не заметить: у нас не всегда есть возможность вернуться к варианту, который мы уже отклонили.

Эту задачу можно переформулировать с учетом современных реалий. Представьте, что мы ищем лучший способ определения того, какой автомобиль следует покупать. Мы заранее составили список автосалонов, которые хотим посетить. Кроме того, наше самолюбие не позволит нам вернуться в салон, из которого мы уже вышли с пустыми руками. Отклонив предложение, мы обязательно должны сохранить лицо; мы не можем снова прийти в то же место и сказать, что мы передумали. Так или иначе, в каждом автосалоне придется принимать окончательное решение: либо покупать, либо никогда больше не возвращаться.

Это разновидность *задачи оптимальной остановки*. Мы должны принять решение, пытаясь максимизировать награду или минимизировать расходы. В нашем примере нам нужно выбрать лучший автомобиль, который только можно купить. Если принять решение слишком рано, может оказаться, что впереди нас ждал более оптимальный вариант. А запоздавшее решение может означать, что мы уже видели лучший автомобиль, но не решились его купить. Так когда следует остановиться и сделать покупку?

Более формальное описание этой проблемы называется *задачей о секретаре* (или *о разборчивой невесте*). Мы хотим выбрать секретаря из группы кандидатов. Вы можете по очереди проводить собеседования с каждым кандидатом. В конце каждого собеседования необходимо принять решение: нанимать или нет. Если кандидат отклонен, вы уже не сможете к нему вернуться (его может перехватить кто-то другой, так как он слишком хорош). Как же сделать правильный выбор?

Ответ на удивление прост. Мы интервьюируем первые 37% кандидатов, отклоняем их всех, но отмечаем для себя лучшего из них. Число 37 может показаться случайным, но это результат деления $1/e \approx 37\%$, где e — это постоянная Эйлера, которая равна примерно 2,7182 (мы уже упоминали о ней в главе 1). Затем мы встречаемся с остальными кандидатами и останавливаем свой выбор на том, который лучше отмеченного нами ранее. Запишем это в виде алгоритма для n кандидатов.

1. Вычисляем n/e , чтобы получить 37% от n кандидатов.
2. Проверяем и отклоняем первые n/e кандидатов. Выбираем лучшего из них в качестве эталона.
3. Проверяем остальных кандидатов. Выбираем того, который оказывается лучше эталона, и останавливаемся.

Этот алгоритм не всегда находит лучшего кандидата; в конце концов, это может быть тот самый эталон, который выбран среди первых 37% и затем отклонен. Но мы можем доказать, что он выбирает лучшего кандидата в 37% случаев (опять же, $1/e$); более того, не существует другого алгоритма, который способен превзойти этот показатель. Иными словами, это лучший алгоритм из всех имеющихся: он может не дать оптимального результата в 63% случаев, но любая другая стратегия будет проваливаться еще чаще.

Возвращаясь к автомобилям, представим, что мы составили список из 10 автосалонов. Мы должны посетить первые четыре и найти лучшее предложение, не делая покупку. Затем мы начинаем посещать другие шесть автосалонов и принимаем то предложение, которое превзойдет отмеченное нами ранее (остальные отклоняем). Возможно, ни один из шести салонов не сделает лучшего предложения. Но никакая другая стратегия не даст более высоких шансов на оптимальную покупку.

Нам нужен был лучший кандидат из возможных, и мы не хотели соглашаться на меньшее. Но что, если эти высокие требования можно немного снизить? Конечно, в идеале мы хотели бы сделать лучший выбор, будь то секретарь или автомобиль, но нас может удовлетворить и другой вариант (хотя и не так сильно). Если поставить вопрос таким образом, то поиск лучше проводить с помощью того же алгоритма, но вместо 37% взять и отклонить \sqrt{n} кандидатов. В этом случае вероятность принятия оптимального решения растет вместе с количеством кандидатов, n , стремясь к 1 (то есть к 100%).

Мы рассмотрели разные способы выполнения поиска, рассчитанные на разные сценарии. Общим во всех этих примерах было то, что перебираемые нами элементы не находятся в каком-то определенном порядке; в лучшем случае мы постепенно упорядочиваем их по популярности (самоорганизующийся поиск). Совсем другое дело, когда элементы изначально упорядочены.

Представьте, что у нас есть стопка папок, каждой из которых присвоен отдельный номер. Документы в стопке упорядочены согласно этим номерам, по их возрастианию (номера могут не быть последовательными). Если нам нужно найти документ с определенным номером, было бы глупо начинать с начала и просто двигаться в конец. Намного лучше перейти сразу в середину и сравнить идентификатор документа с искомым номером. Здесь может произойти одно из трех.

1. Если нам повезет, мы попадем сразу на нужный нам документ. В этом случае мы закончили, поиск останавливается.
2. Номер искомого документа превышает номер, который нам попался. Это означает, что мы можем отклонить как этот, *так и все предыдущие*

документы. Они упорядочены, поэтому у всех у них будут меньшие номера. У нас получился «недолет».

3. Происходит противоположное: номер искомого документа меньше того, который нам попался. Мы можем спокойно отклонить как этот, так и все последующие документы. Наша цель находится позади.

В двух последних случаях у нас остается стопка, которая как минимум вдвое меньше исходной. Если начать с нечетного количества документов (скажем, с n) и разделить их посередине, получится две равные части по $n / 2$ элементов в каждой (игнорируем дробную часть).

○ ○ × ○ ○

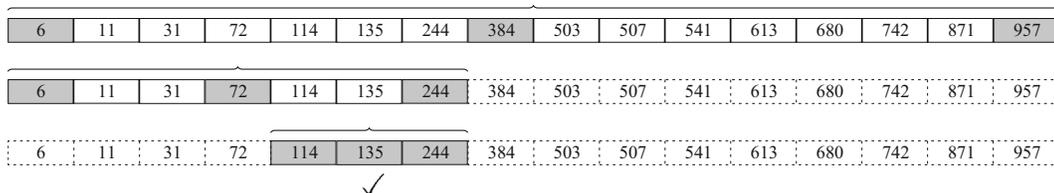
Если у нас четное количество элементов, в результате их деления получится две части: в одной $n / 2 - 1$, а в другой $n / 2$.

○ × ○ ○

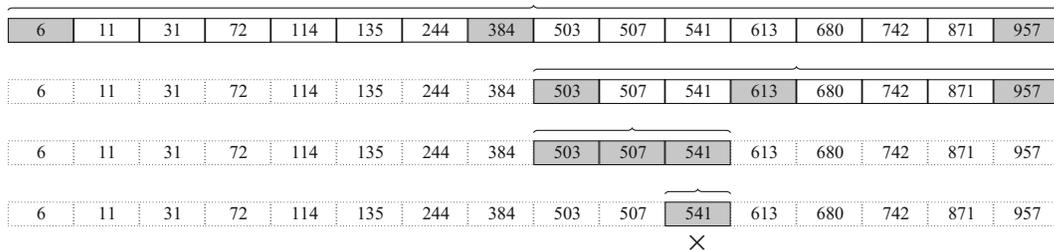
Мы все еще не нашли то, что искали, но мы находимся в куда лучшей позиции, чем прежде; у нас осталось намного меньше элементов, которые нужно перебрать. Итак, продолжаем. Проверяем средний документ в оставшейся стопке и повторяем ту же процедуру.

На диаграмме, показанной на следующей странице, можно видеть, как этот процесс работает в случае с 16 элементами, среди которых нужно найти элемент 135. Мы выделяем серым цветом рамки, в которых происходит поиск, и средний элемент.

Вначале область поиска охватывает весь список. Мы переходим к среднему элементу и узнаем, что он имеет номер 384. Это больше, чем 135, поэтому мы отбрасываем его и все элементы справа от него. Берем средний элемент в оставшемся списке; он имеет номер 72. Это меньше, чем 135, поэтому мы отбрасываем его вместе со всеми элементами, которые находятся слева от него. Область поиска сократилась до всего лишь трех элементов. Мы берем средний из них и обнаруживаем, что это тот, который нам нужен. На выполнение поиска ушло всего три шага, и нам даже не пришлось проверять 13 элементов из 16.



Этот процесс будет работать и в ситуации, когда искомого элемента не существует. Это можно наблюдать на следующей диаграмме, где в том же списке ищется элемент 520.



На этот раз 520 больше, чем 384, поэтому мы ограничиваем область поиска правой половиной списка. В ней посередине находится элемент 613, который больше, чем 520. У нас остается всего три элемента, средний из которых 507. Он меньше 520. Мы его отбрасываем и проверяем оставшиеся два элемента, ни один из которых не имеет искомого номера. Мы можем завершить поиск, сообщив о том, что он оказался безрезультатным. Для этого потребовалось всего четыре шага.

Описанный нами метод называется *двоичным поиском*, потому что на каждом этапе область поиска уменьшается вдвое. Этот принцип можно оформить в виде алгоритма, состоящего из следующих шагов.

1. Если область поиска пустая, нам нечего проверять, поэтому сообщаем о неудаче и останавливаемся. В противном случае находим средний элемент.
2. Если средний элемент меньше искомого, ограничиваем область поиска элементами, которые идут за ним, и возвращаемся к пункту 1.
3. В противном случае, если средний элемент больше искомого, ограничиваем область поиска элементами, которые идут перед ним, и возвращаемся к пункту 1.
4. Если средний элемент равен искомому, сообщаем об успехе и останавливаемся.

Таким образом мы сокращаем вдвое количество элементов, по которым нам нужно искать. Это метод типа «разделяй и властвуй». Он состоит в многократном делении, которое, как мы видели в главе 1, дает нам логарифм. Многократное деление на два — это логарифм по основанию 2. В худшем случае двоичный поиск продолжит разделять список элементов, пока ничего

Не отчаивайтесь, если вам когда-либо придется лихорадочно размышлять над строчкой кода, которая делает не то, чего вы от нее ожидали. Вы не одиноки. Это случается со всеми, даже с лучшими из нас.

не останется (как в примере с неудачным поиском). Если у нас есть n элементов, это может произойти не больше, чем $\lg n$ раз; следовательно, двоичный поиск имеет сложность вида $O(\lg n)$.

Это довольно впечатляющее преимущество перед последовательным поиском, даже если он самоорганизующийся. Поиск по миллиону элементов займет не больше 20 шагов. С другой стороны, ста шагов хватит на то, чтобы найти любой элемент в списке длиной $2^{100} \approx 1,27 \times 10^{30}$, что больше одного нониллиона.

Поразительная эффективность двоичного поиска сравнима разве что с его известностью. Это интуитивный алгоритм. Но, несмотря на свою простоту, он вновь и вновь вызывает затруднения при реализации его в компьютерных программах. Это связано не с самим двоичным поиском, а скорее с тем, как мы переводим алгоритмы в настоящий компьютерный код с помощью языков программирования. Программисты постоянно становятся жертвами коварных ошибок, которые закрадываются в их реализации. И это касается не только новичков: иногда даже программистам мирового класса не удается реализовать этот алгоритм как следует.

Чтобы понять, откуда берутся эти ошибки, подумайте о том, как мы находим средний элемент в списке на первом этапе алгоритма. Вот простой подход: середина между m -м и n -м элементами равна $(m + n) / 2$; если результат не натуральное число, он округляется. Это правильный принцип, основанный на элементарной математике, поэтому он применим везде.

Везде, кроме компьютеров. Компьютеры обладают ограниченными ресурсами, к которым относится и память. Поэтому на компьютере невозможно представить любые числа; некоторые из них просто слишком большие. Если компьютер накладывает ограничение на максимальный размер чисел, с которыми он может работать, то m и n должны быть меньше этого лимита. Конечно, $(m + n) / 2$ вписывается в допустимый диапазон, но, чтобы вычислить это значение, нам нужно сначала выполнить $m + n$ и затем поделить на два, *и эта сумма может превысить установленное ограничение!* Это называется *переполнением*: выход за пределы допустимых значений. Поэтому у вас может возникнуть ошибка, которая, как вы считали, вам не страшна. Результатом будет не среднее значение, а что-то совсем другое.

Осознав эту опасность, вы можете легко от нее защититься. Средний элемент нужно вычислять не как $(m + n) / 2$, а как $m + (m - n) / 2$. Результат тот же, но без переполнения. Теперь, когда вы знаете, в чем загвоздка, это решение кажется простым, но любой может быть пророком задним числом.

Конечно, эта книга посвящена алгоритмам, а не программированию, но позвольте автору поделиться опытом с теми, кто пишет или хочет писать компьютерные программы. Не отчаивайтесь, если вам когда-либо придется лихорадочно размышлять над строчкой кода, которая делает не то, чего вы от нее ожидали. Не волнуйтесь, если на следующий день окажется, что ошибка все это время была прямо у вас перед носом. Как же вы могли ее не заметить? Вы не одиноки. Это случается со всеми, даже с лучшими из нас.

Двоичный поиск требует, чтобы элементы были отсортированы. Поэтому, чтобы воспользоваться его преимуществами, нам нужно сначала научиться эффективно сортировать элементы. Таким образом мы плавно переходим к следующей главе, в которой вы узнаете, как сортировать вещи с помощью алгоритмов.

СОТИРОВКА

Конституция США гласит, что раз в десять лет должна проходить перепись населения, чтобы распределить налоги и представителей между разными штатами. Первая перепись после Американской революции состоялась в 1790 году, и с тех пор ее проводят каждое десятилетие.

На протяжении следующих ста лет после 1790 года население США стремительно выросло — с 4 миллионов во время первой переписи до более чем 50 миллионов в 1880 году. Но вместе с этим возникла проблема: время, уходившее на подсчеты, достигло восьми лет. Когда в 1890 году пришел черед следующей переписи, население выросло еще сильнее. Если бы этот процесс проходил как раньше, он, скорее всего, не был бы завершен до следующей переписи 1900 года.

В то время в Бюро переписи населения США работал Герман Холлерит, молодой выпускник Горной школы при Колумбийском университете (он окончил обучение в 1879 году в возрасте 19 лет). Зная о насущной проблеме с нехваткой времени, он пытался придумать, как ускорить этот процесс с помощью механических устройств. Холлерит вдохновился тем, как кондукторы записывали данные пассажиров, пробивая дыры в их железнодорожных билетах; он изобрел способ записи информации о переписи населения с использованием *перфокарт*. Затем эти карты можно было обработать *табуляторами* — электромеханическими машинами, которые считывали пробитые дыры и проводили на их основе вычисления.

Табулятор Холлерита применялся в переписи 1890 года и позволил сократить время подсчетов до шести лет; оказалось, что население США выросло примерно до 63 миллионов человек. Холлерит представил свое изобретение Королевскому статистическому обществу, отметив: «Не следует считать, что эта система все еще находится на стадии экспериментов. Эти машины подсчитали более 100 000 000 перфокарт по нескольку раз, и это создало широкое поле для проверки их возможностей». После проведения переписи Холлерит

основал частную компанию под названием Hollerith Electric Tabulating System. В 1924 году после череды переименований и слияний она превратилась в International Business Machines () .

В наши дни сортировка стала настолько вездесущей, что мы ее почти не замечаем. Всего несколько десятилетий назад в офисах были громадные картотеки с папками, и корпоративный персонал следил за тем, чтобы они хранились в нужном порядке (алфавитном или хронологическом). Для сравнения: всего одним щелчком мыши все письма в нашем почтовом ящике можно отсортировать по таким критериям, как тема, дата и отправитель. Наши контакты хранятся в отсортированном виде на цифровых устройствах без какого-либо внимания с нашей стороны; всего пару лет назад нам приходилось самостоятельно организовывать наши контакты в записных книжках.

Вернемся к переписи населения в США. Сортировка была одним из первых примеров автоматизации офисной работы и одним из первых применений цифровых вычислительных машин. За это время разработано множество разных алгоритмов сортировки. Целый ряд из них активно используется в программировании, предлагая разные сравнительные преимущества и недостатки, хотя некоторые не применяются на практике. Сортировка — это настолько фундаментальный аспект работы компьютеров, что в любой книге, посвященной алгоритмам, им всегда отводится отдельное место. И именно благодаря их большому разнообразию можно оценить важную сторону деятельности компьютерных специалистов и программистов. В их распоряжении находится множество инструментов, и некоторые из них подходят для одной и той же задачи. Представьте себе набор отверток: плоских, крестообразных, шестигранных, квадратных и т. д. У всех у них одно и то же назначение, но для определенных болтов нужны подходящие отвертки. Иногда плоская отвертка подходит для крестообразного болта; но обычно в каждой ситуации следует выбирать соответствующий инструмент. То же самое с сортировкой. Все сортировочные алгоритмы занимаются упорядочиванием, но каждый из них предназначен для определенных задач.

Прежде чем приступить к исследованию этих алгоритмов, давайте попробуем объяснить, что именно они делают. Очевидно, что они что-то сортируют, но здесь сразу же напрашивается вопрос: что мы имеем в виду под *сортировкой данных*?

Предполагается, что у нас есть набор взаимосвязанной информации, которую обычно называют *записями*. Это, к примеру, могут быть электронные письма в нашем почтовом ящике. Мы хотим организовать эти данные так, чтобы они находились в удобном для нас порядке. Эта перестановка должна

У всех у них одно и то же назначение, но для определенных болтов нужны подходящие отвертки... То же самое с сортировкой. Все сортировочные алгоритмы занимаются упорядочиванием, но каждый из них лучше подходит для определенных задач.

учитывать определенное свойство или свойства данных. В нашем примере с письмами упорядочивание может быть основано на дате получения (хронологический порядок) или имени отправителя (алфавитный порядок). Это можно делать по возрастанию, от старых писем к новым, или по убыванию, от новых писем к старым. Результатом процесса сортировки должны быть те же данные, которые были на входе; говоря техническим языком, это должна быть *перестановка* исходных данных, то есть те же элементы, только в другом порядке, без каких-либо изменений.

Свойство, по которому мы сортируем информацию, обычно называется *ключом*. Ключ может быть *атомарным*, если его нельзя разложить на части, или *составным*, если он основан на нескольких свойствах. Если нам нужно отсортировать письма по дате доставки, достаточно атомарного ключа (дату можно разбить на год, месяц и день, и она может также содержать точное время доставки, однако нам это неважно). Но мы также можем захотеть, чтобы письма были отсортированы сначала по имени отправителя, и чтобы все письма *каждого отправителя* шли в порядке доставки. Сочетание даты и отправителя представляет собой составной ключ для нашей сортировки.

В качестве ключа для сортировки можно использовать любое свойство — главное, чтобы его значение можно было упорядочить. Среди прочего это, естественно, относится к числам. Если нам нужно отсортировать информацию о продажах по количеству проданных товаров, наш ключ будет целым числом. Если ключи текстовые, такие как имена отправителей писем, мы обычно используем лексикографический порядок. Чтобы иметь возможность упорядочить наши данные, алгоритмы сортировки должны знать, как их сравнивать, но способ сравнения может быть любым (главное, чтобы он был корректным).

Начнем исследование методов сортировки с двух алгоритмов, которые могут быть вам знакомы, так как они, наверное, являются самыми интуитивно понятными. Их применяют для упорядочивания разных вещей даже люди, которые ничего не знают об алгоритмах.

Простые методы сортировки

Наша задача состоит в том, чтобы отсортировать следующие элементы.



Следует признать, что это довольно тривиальный пример; мы имеем дело с числами от одного до десяти. Но такое упрощение позволит нам сосредоточиться на логике сортировки.

Для начала мы пройдемся по всем элементам, возьмем меньший из них и переместим его в начало. Минимальное число равно 1, поэтому его следует сделать первым. Но эта позиция уже занята, поэтому нам нужно куда-то деть число 4, которое в ней сейчас находится; мы не можем его просто выбросить. Его можно поменять местами с наименьшим значением: минимум перемещается в самое начало, а то число, которое находилось там изначально, занимает освободившуюся позицию. Таким образом из этого списка, где минимум закрашен черным,

4 6 10 1 7 9 3 2 8 5

получается следующий,

1 6 10 4 7 9 3 2 8 5

где минимум закрашен белым, указывая на то, что он находится в правильной, упорядоченной позиции.

Теперь мы повторяем то же самое для всех чисел, кроме уже найденного минимума, то есть начиная со второй позиции и дальше (серые числа). Находим их минимум (2) и меняем его местами с первым из неотсортированных чисел (6).

1 6 10 4 7 9 3 2 8 5

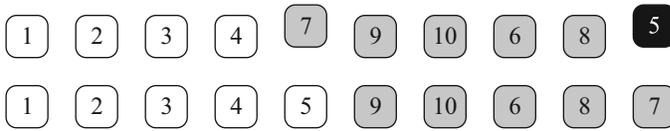
1 2 10 4 7 9 3 6 8 5

И снова делаем то же самое, перебирая элементы, начиная с третьей позиции. Находим минимум (3) и меняем его местами с тем элементом, который сейчас находится на третьем месте (10).

1 2 10 4 7 9 3 6 8 5

1 2 3 4 7 9 10 6 8 5

Если продолжить в том же духе, число 4 останется на месте, так как оно уже находится в правильной позиции. Затем мы поместим в нужную позицию число 5.



На каждом этапе у нас остается все меньше и меньше элементов, среди которых нужно искать минимальный. В конце мы найдем наименьший элемент из двух оставшихся, и на этом сортировка будет закончена.

Этот подход называется *сортировка выбором*, поскольку каждый раз из неотсортированных элементов выбирается минимальный и помещается туда, где он должен быть. Как и любой другой сортировочный алгоритм, который мы здесь рассмотрим, сортировка выбором отлично справляется с элементами, у которых одинаковый порядок. Если при поиске по неотсортированным элементам находится больше одного минимума, мы просто выбираем любой из них, а другие найдутся на следующих этапах и будут помещены рядом с ним.

Сортировка выбором — довольно прямолинейный алгоритм. Но можно ли его назвать хорошим? Внимательно присмотритесь к тому, что он делает: он последовательно проходит по всем элементам, которые нужно отсортировать, и каждый раз пытается найти минимум среди оставшихся неотсортированных элементов. Если количество элементов n , сложность сортировки выбором составит $O(n^2)$. Этот подход не плохой сам по себе; такая сложность является приемлемой. Мы можем решать огромные задачи (то есть сортировать много элементов) за разумное количество времени.

Но дело в том, что именно по причине большого значения сортировки для нее были созданы другие, более быстрые алгоритмы. Сортировка выбором не так уж и плоха, но обычно, если у нас есть большое количество элементов, мы отдаем предпочтение другим, более совершенным методам. С другой стороны, сортировка выбором проста для понимания, и ее легко можно реализовать на компьютере эффективным образом. Поэтому она совершенно точно не ограничивается академическим интересом; ее действительно применяют на практике.

То же самое можно сказать о другом простом сортировочном алгоритме, который мы сейчас опишем. Как и в случае с сортировкой выбором, для его

понимания не нужно разбираться в компьютерах. На самом деле именно так многие люди сортируют свои игральные карты.

Представьте, что вы участвуете в карточной игре с десятью картами (например, вы можете играть в Румми). Вы выбираете одну карту за другой и сортируете их в своей руке. Допустим, карты разделены по старшинству, начиная с младшей.

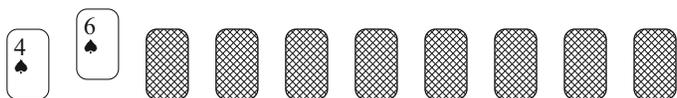
2 3 4 5 6 7 8 9 J Q K A

На самом деле во многих играх (включая Румми) туз может быть как младшей, так и старшей картой, но здесь мы исходим из того, что порядок всегда один.

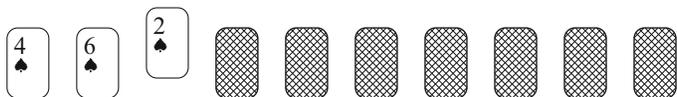
Вы начинаете с одной карты и затем получаете еще девять.



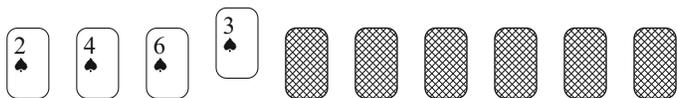
В качестве второй карты вам попалась шестерка.



Шестерка вполне может находиться рядом с четверкой, поэтому вы оставляете ее как есть и берете следующую карту. Вам попалась двойка.



На этот раз, чтобы упорядочить ваши карты, вам нужно поместить двойку слева от четверки, то есть сместить четверку и шестерку на одну позицию вправо. После этого вам выпадает следующая карта, тройка.



Вы вставляете ее между двойкой и четверкой. Следующая карта, девятка, уже находится в правильной позиции.



Дальше вам могут попасться такие карты, как 7, Q, J, 8 и 5. В конце у вас получится отсортированный набор.

Каждая новая карта вставляется в правильную позицию по отношению к предыдущей, уже упорядоченной. По этой причине данный подход называется *сортировкой вставками*, и он подходит для любого рода объектов, не только для карт.

Сортировку вставками, как и сортировку выбором, довольно просто реализовать. Оказывается, она имеет ту же сложность: $O(n^2)$. Но у нее есть одна особенность: как вы могли видеть на примере нашей карточной игры, *нам не нужно заранее знать элементы, которые мы сортируем*. В сущности, мы упорядочиваем их по мере поступления. Это означает, что сортировку вставками можно использовать в ситуациях, когда элементы передаются в виде какого-то потока прямо на лету. Нам уже встречался подобный алгоритм при обсуждении графов на примере задачи с турниром в главе 2. Он называется *онлайн-алгоритмом*. Если нам нужно отсортировать неизвестное количество элементов или мы должны предоставить упорядоченный список по требованию в любой момент времени, сортировка вставками — подходящий выбор.

Поразрядная сортировка

Давайте вернемся к Холлериту. В его табуляторах не использовалась ни сортировка выбором, ни сортировка вставками. Вместо них применялся предшественник метода, который до сих пор находится в употреблении, — *поразрядная сортировка*. Если вам интересен первый в истории пример автоматизированного упорядочивания данных, уделите некоторое время, чтобы понять, как это работает. Интересно также то, что при сортировке данным методом не происходит сравнения упорядочиваемых элементов друг с другом. По крайней мере, не полностью (подробнее об этом чуть ниже). Кроме того, поразрядная сортировка представляет не только исторический интерес, так как она до сих пор прекрасно работает. Кому не понравится почтенный, но в то же время практичный алгоритм?

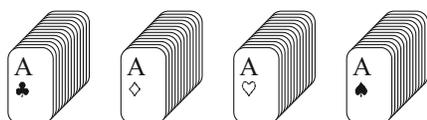
Проще всего поразрядную сортировку проиллюстрировать на примере все тех же игральных карт. Представьте, что у вас есть полная перетасованная колода, и вы хотите ее отсортировать. Чтобы это сделать, колоду можно разделить на 13 стопок, по одной для каждого достоинства. Мы проходимся по колоде, выбираем по одной карте и кладем ее в соответствующую стопку.

У нас получится 13 стопок по четыре карты в каждой: одна со всеми тузами, другая со всеми двойками и т. д.



Затем мы собираем все стопки вместе, размещая каждую следующую снизу. Таким образом у нас в руках окажется полная, частично отсортированная колода. Первые четыре карты будут тузами, следующие четыре будут двойками и так далее вплоть до королей.

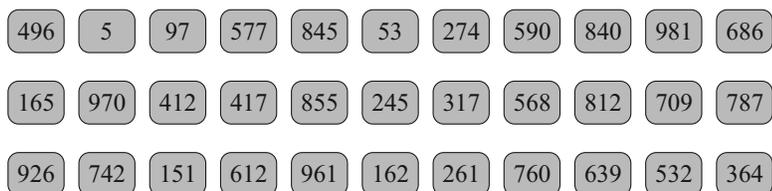
Теперь создадим четыре новых стопки, по одной для каждой масти. Мы пройдемся по всем картам, откладывая каждую из них в соответствующую стопку. У нас получится четыре стопки. Поскольку достоинства уже были отсортированы, в каждой стопке окажутся карты одной масти, упорядоченные по старшинству.



Чтобы закончить сортировку, остается просто собрать карты стопка за стопкой.

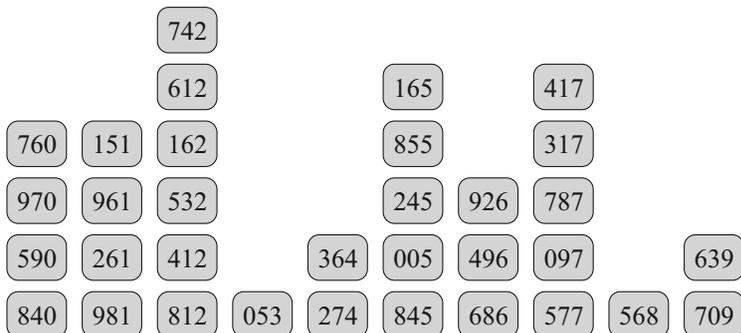
Это основная идея поразрядной сортировки. Мы не сравнивали все карты между собой. Сравнение было частичным: сначала по достоинству, затем по масти.

Конечно, если бы поразрядная сортировка годилась только для карт, она не стоила бы нашего внимания. Давайте посмотрим, как она работает с целыми числами. Представьте, что у нас есть следующий набор значений.

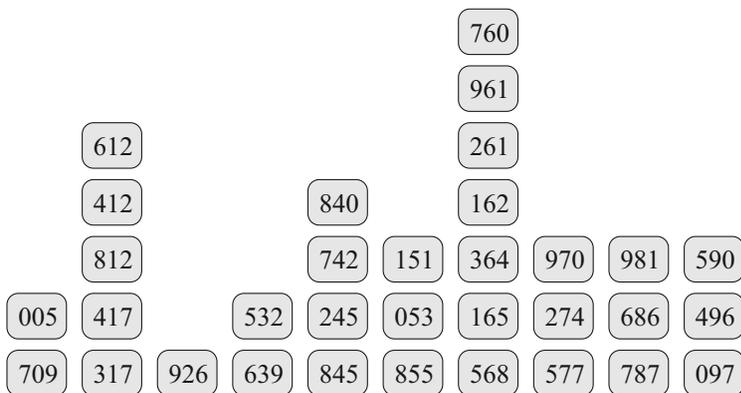


Следует позаботиться о том, чтобы все числа имели одинаковое количество разрядов. Поэтому при необходимости добавляем слева ноли, превращая

5 в 005, 97 в 097 и 53 в 053. Перебираем все наши числа и разделяем их на десять групп по крайней справа цифре.



Мы сделали числа чуть светлее, чтобы сигнализировать о том, что они частично упорядочены; в каждой группе находятся значения с одним и тем же младшим разрядом. В первой группе числа заканчиваются на ноль, во второй — на один, а в последней — на девять. Мы собираем эти десять групп, двигаясь слева направо и добавляя каждую следующую снизу от предыдущей (числа в группах ни в коем случае не должны перемешаться). Затем мы разделяем их на десять других групп, используя вторую цифру справа. Получится следующее.



На этот раз все числа в первой группе имеют одинаковый предпоследний разряд, равный нулю; во второй группе предпоследняя цифра равна 1, то же самое с остальными группами. В то же время числа в каждой группе уже

отсортированы по последнему разряду, так как именно это мы сделали, когда объединили их на первом этапе.

Снова собираем группы и перераспределяем числа, на этот раз используя третий разряд справа.

				532		709	812	926	
005	151	245		412	568	612	742	840	961
053	162	261	317	417	577	639	760	845	970
097	165	274	364	496	590	686	787	855	981

Теперь элементы в каждой группе начинаются с одной и той же цифры; при этом они отсортированы по второму и третьему разрядам (второй и первый этапы). Чтобы закончить сортировку, достаточно объединить группы в последний раз.

Поразрядная сортировка подходит для слов, любых алфавитно-цифровых символов и целых чисел. В информатике последовательность алфавитно-цифровых символов называется *строкой*. Этот подход работает со строками; строка может состоять из цифр, как в нашем примере, но необязательно. Количество групп для алфавитно-цифровых строк равно количеству букв в алфавите (например, 26 для английского алфавита), но проводимые операции будут точно такими же. Отличительная сторона поразрядной сортировки состоит в том, что строки всегда воспринимаются в качестве алфавитно-цифровых последовательностей, а не чисел, даже если они состоят из одних цифр. Если еще раз взглянуть на наш пример, можно заметить, что нас не интересовали конкретные значения; каждый раз мы работали с отдельным разрядом числа. Точно так же мы извлекали бы символы из слова, двигаясь справа налево. Вот почему этот подход еще называют *методом сортировки строк*.

Но не думайте, что среди всех представленных здесь методов сортировки только поразрядный умеет упорядочивать строки. Все они могут это делать. Главное, чтобы сами символы, из которых эти строки состоят, могли быть упорядочены. Компьютер воспринимает человеческие имена как строки, поэтому их можно сортировать, ведь буквы могут быть расставлены в алфавитном порядке, а имена можно сравнивать лексиграфически. Название «сортировка строк» связано с тем, что поразрядная сортировка работает со всеми ключами как со строками, даже с числами. Другие методы, представленные в этой главе, разделяют понятия чисел и строк и сравнивают их

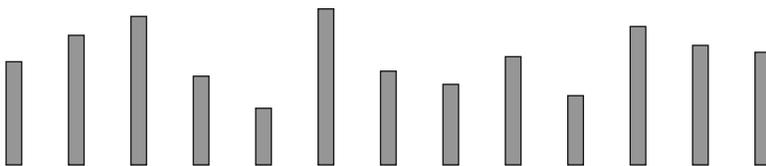
соответствующим образом. В наших примерах цифровые ключи используются сугубо для удобства.

Поразрядную сортировку делает эффективной тот факт, что она обрабатывает элементы цифра за цифрой (или символ за символом). Если нам нужно упорядочить n элементов, каждый из которых состоит из w цифр или символов, сложность алгоритма составит $O(nw)$. Это намного лучше, чем сложность $O(n^2)$, присущая сортировке выбором и вставками.

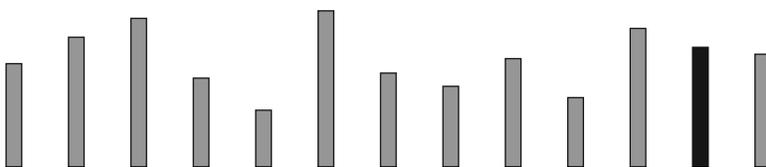
Таким образом мы снова возвращаемся к табуляторам. Табулятор работал похожим образом, сортируя перфорированные карты. Представьте, что у вас есть набор карт по десять столбцов в каждой. Отверстия в каждом столбце обозначают цифру. Устройство способно распознавать эти дыры и тем самым определять соответствующие значения. Оператор кладет карты в табулятор, а тот распределяет их по десяти выходным корзинам в зависимости от последнего столбца, то есть младшего разряда. Оператор собирает карты из корзин, стараясь не перемешать их никоим образом, и затем снова подсовывает их устройству, которое на этот раз распределяет их по выходным корзинам на основе предпоследнего столбца — разряда, который находится перед младшим. Повторив этот процесс десять раз, оператор собирает упорядоченный набор карт. Вуаля!

Быстрая сортировка

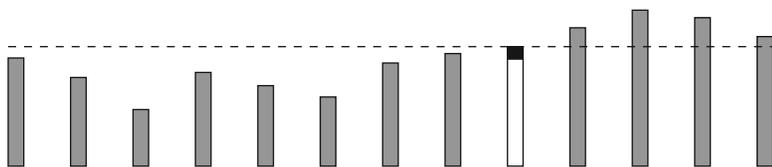
Представьте, что у нас есть группа детей, которые бегают по двору (возможно, школьному), и вы хотите выстроить их в ряд от самого низкого до самого высокого. Сначала мы просим их построиться в любом порядке на их выбор. Вот что получается.



Теперь выбираем ребенка наугад.



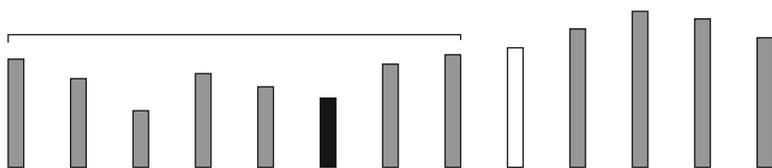
Мы просим детей, которые ниже выбранного ребенка, стать слева от него, а всех остальных — справа. На следующей диаграмме показано, где в итоге оказался выбранный ребенок; вы можете видеть, что дети, которые выше его, находятся справа, а те, которые ниже — слева.



Мы не просили детей построиться в правильном порядке. Они всего лишь переместились относительно выбранного нами ребенка. У нас получилось две группы: слева и справа. Дети в этих группах не выстроены по росту. Но мы точно знаем, что *один* ребенок (тот, которого мы выбрали) уже находится в правильной позиции в том ряду, который мы хотим сформировать. Все дети по левую сторону ниже его, а по правую — как минимум такого же роста или выше. Назовем выбранного нами ребенка *точкой вращения*, так как все остальные дети будут перемещаться относительно него.

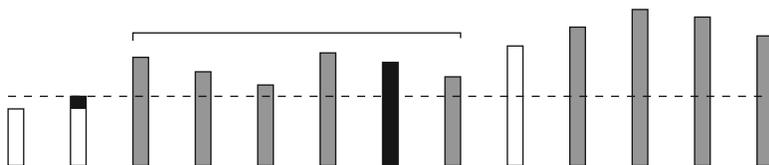
Чтобы вам было легче ориентироваться, мы станем закрашивать детей, которые уже находятся в правильной позиции, белым. Точка вращения, которую мы выбираем, выделим черным; после перемещения остальных детей мы нарисуем небольшую черную «шляпку», чтобы обозначить итоговую позицию точки вращения (ее закрасим белым, поскольку она уже в правильной позиции, с черным верхом, который сигнализирует о выполненном перемещении).

Теперь сосредоточимся на одной из двух групп — например, на левой. Опять выбираем наугад точку вращения в этой группе.

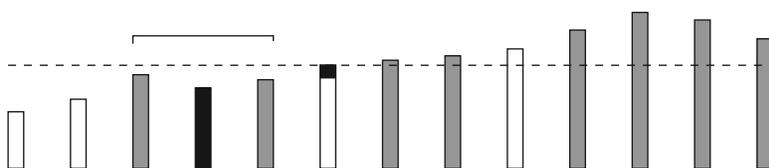


Мы просим детей в этой группе сделать то же, что и раньше: переместиться влево от выбранного ребенка, если они ниже его, или вправо, если нет. Как видно ниже, у нас опять получилось две группы поменьше. Одна из них состоит из одного ребенка, который, очевидно, находится в правильной позиции. Остальные дети находятся справа от второй точки вращения. Вторая точка тоже находится в нужном месте: все дети, которые ниже ростом,

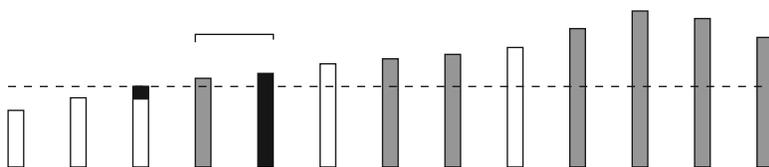
находятся слева, а остальные справа. Эта вторая группа охватывает всех детей вплоть до первой точки вращения. Выбираем в ней третью точку.



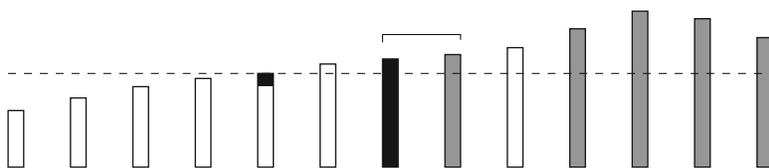
После того как мы скажем детям переместиться по тому же принципу, в зависимости от того, какой у них рост по сравнению с третьей точкой вращения, у нас получится еще две группы меньшего размера. Сосредоточимся на той, которая находится слева. Делаем то же, что и раньше. Выбирает точку вращения, уже четвертую по счету, и просим трех детей в этой группе построиться вокруг нее.



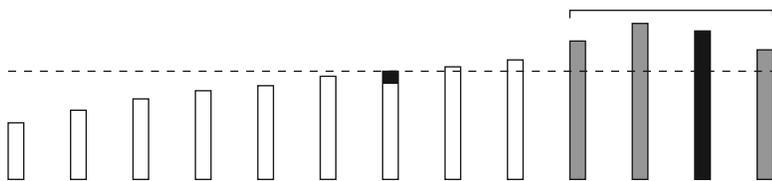
Когда они это сделают, точка вращения окажется первой в этой группе из трех детей; у нас остается группа по правую сторону, в которую входят два ребенка. Выбираем одного из них в качестве точки вращения, и другой ребенок при необходимости переместится вправо относительно нее.



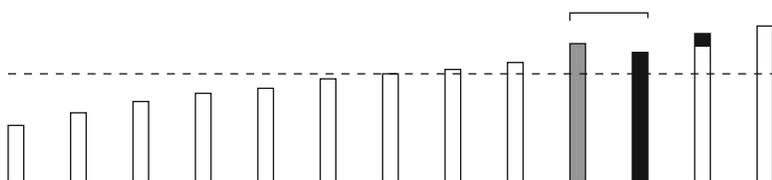
Оказывается, детям уже не нужно куда двигаться. Итак, нам удалось выстроить примерно половину детей; у нас остается две группы, которые мы покинули при работе с предыдущими точками вращения. Вернемся к первой из них (той, что слева), выберем точку вращения и повторим весь процесс.



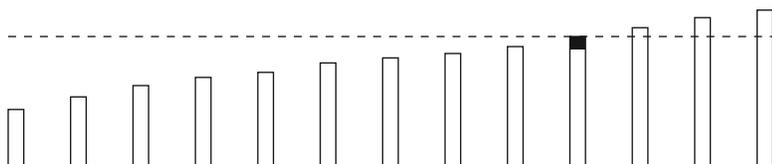
И снова не потребовалось никаких перемещений, поэтому мы берем последнюю группу неупорядоченных детей и выбираем точку вращения.



Справа у нас получилась группа из одного ребенка, а слева — из двух. Берем левую группу и выбираем одного из двух детей в качестве точки вращения.



Вот и все. Все дети выстроены по росту.

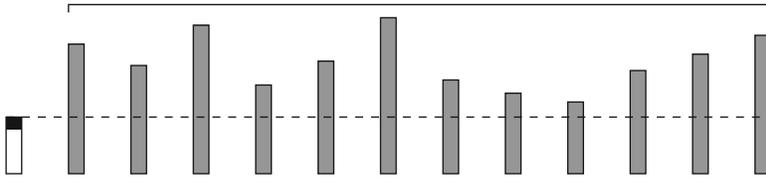


Давайте подведем итоги того, что мы только что сделали. Мы упорядочили всех детей, выбирая по одному ребенку и размещая его в правильной позиции. Для этого нам было достаточно попросить остальных детей выстроиться относительно него. Это всегда работает, и, конечно же, не только с детьми, но с любыми объектами, которые нужно отсортировать. Если необходимо упорядочить набор чисел, мы можем выполнить аналогичный процесс, выбирая произвольное число и группируя остальные числа вокруг него так, чтобы меньшие оказались перед ним, а остальные за ним. То же самое повторяется в группах меньшего размера, которые у нас получились; в итоге все числа окажутся упорядоченными. Это процесс, лежащий в основе *быстрой сортировки*.

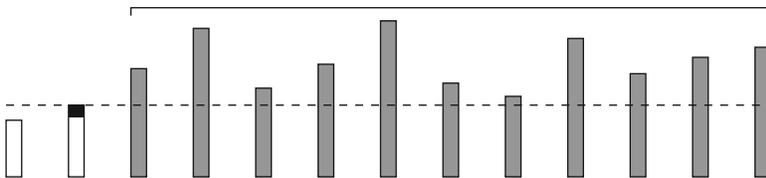
Быстрая сортировка основана на следующем наблюдении: если нам удастся расположить один элемент в правильной позиции относительно всех остальных (какой бы она ни была) и затем повторить то же самое для каждого оставшегося элемента, все элементы в итоге окажутся в правильных

позициях. Если вспомнить сортировку выбором, там мы тоже брали каждый элемент и перемещали его в нужное место относительно остальных, но этот элемент всегда был наименьшим из тех, что остались. Это ключевое отличие: в быстрой сортировке точка поворота *не должна* быть минимальным элементом среди оставшихся. Давайте посмотрим, что получится, если все же выбирать минимум.

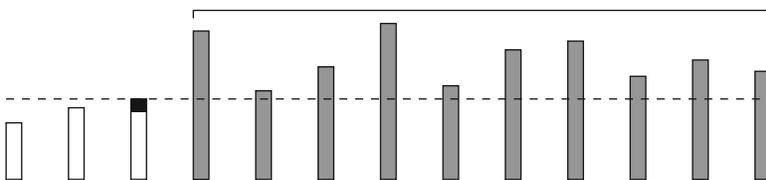
Если снова начать с той же группы детей, точкой вращения будет самый низкий ребенок. Он перейдет в начало шеренги, а все остальные окажутся за ним.



Затем мы выберем самого низкого ребенка из оставшихся и сделаем его вторым. Все остальные опять окажутся справа от него.



Повторим то же самое с третьим ребенком и получим следующее.



Обратите внимание на то, как сильно это напоминает сортировку выбором: шеренга выстраивается слева направо путем отбора самого низкого ребенка из оставшихся.

Теперь мы видим, что в качестве точки вращения не следует выбирать минимальный элемент. Во-первых, это требует определенных усилий: каждый раз нам приходится искать наименьшее значение. Во-вторых, этот метод похож на уже известный нам алгоритм, поэтому использовать его нет особого смысла.

На самом деле быстрая сортировка лучше сортировки выбором, потому что «обычно» (вскоре мы увидим, что это означает) выбираемая нами точка

вращения разделяет данные более равномерно. Выбор минимального элемента приводит к самому неравномерному распределению: все данные уходят вправо, а слева ничего не остается. Таким образом, на каждом этапе мы перемещаем только саму точку вращения.

При более равномерном распределении перемещается не только выбранный нами элемент. Правильные позиции *относительно него* занимают все элементы, которые оказываются слева. Конечно, речь не идет об их окончательных позициях, но в целом они находятся в лучшем положении, чем раньше. Таким образом один элемент оказывается в идеальной позиции, а остальные улучшают свое положение.

Это оказывает важное влияние на производительность быстрой сортировки: ее ожидаемая сложность составляет $O(n \lg n)$, что намного лучше, чем $O(n^2)$. Если мы хотим отсортировать 1 миллион элементов, $O(n^2)$ понадобится 10^{12} (триллион) операций, а $O(n \lg n)$ — около 20 миллионов.

Все зависит от выбора правильной точки вращения. Не следует каждый раз искать такие точки, которые разделяют наши данные оптимальным образом; это лишь усложнило бы весь процесс. Вместо этого лучше положиться на волю случая. Просто выберите произвольный элемент и используйте его для разделения данных.

Чтобы показать, что это хорошая стратегия, давайте сначала объясним, почему ее нельзя назвать плохой. Плохая стратегия ведет себя так, как мы только что видели: когда быстрая сортировка вырождается в сортировку выбором. Это происходит, если на каждом этапе выбирать такую точку вращения, которая в действительности не разделяет элементы. Например, мы можем каждый раз выбирать наименьший или наибольший элемент (результат будет один и тот же). Общая вероятность этого равна $2^{n-1}/n!$.

Вероятность вида $1/n!$ сложно себе представить, так как она крайне низкая. Для сравнения: если взять колоду с 52 игральными картами и перетасовать ее случайным образом, вероятность того, что она окажется упорядоченной, составляет $1/52!$. Это примерно то же самое, что получить решку 226 раз подряд при подкидывании монеты. Если умножить это на 2^{n-1} , мало что поменяется. $2^{51}/52!$ равно примерно $2,8 \times 10^{-53}$. Если взглянуть на это с точки зрения космических величин, наша планета состоит где-то из 10^{50} атомов. Если вы с вашим другом выберете по одному случайному атому, вероятность того, что это окажется один и тот же атом, будет равна 10^{-50} , что на самом деле больше, чем вероятность патологической быстрой сортировки колоды карт ($2^{51}/52!$).

Это объясняет, почему «обычно» мы разделяем данные более равномерно. Если не брать во внимание полосу невезения космических масштабов,

не стоит волноваться о том, что на каждом этапе будет выбрана худшая точка вращения из всех возможных. Наши шансы довольно неплохие: сложность $O(n \lg n)$ получается при выборе точек вращения случайным образом. Конечно, теоретически нам может не повезти, но вероятность этого представляет разве что академический интерес. На практике быстрая сортировка будет работать настолько хорошо, насколько можно ожидать.

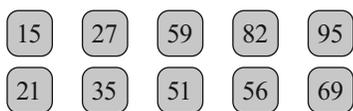
Быстрая сортировка изобретена в 1959–1960 годах британским ученым Тони Хоаром, который специализировался в области информатики. Это, наверное, самый популярный алгоритм сортировки на сегодня, потому что, если его правильно реализовать, он превосходит все остальные. Это также первый из рассмотренных нами алгоритмов, который демонстрирует не совсем детерминированное поведение. Он всегда сортирует правильно, но мы не можем гарантировать, что у него всегда будет одинаковая производительность. Тем не менее вероятность того, что он поведет себя патологическим образом, крайне мала. Это важная концепция, которая подводит нас к теме так называемых *вероятностных алгоритмов*, которые используют в своей работе элемент случайности. Это может показаться нелогичным, ведь алгоритмы должны быть детерминированными, послушными существами, которые четко следуют инструкциям, расставленным в заранее определенном порядке. И все же в последние годы можно наблюдать расцвет вероятностных алгоритмов. Оказывается, элемент случайности помогает решать задачи, с которыми не получается справиться более стандартными способами.

Сортировка слиянием

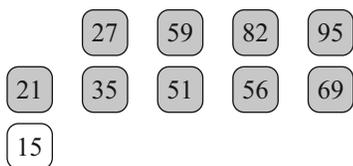
Мы уже рассматривали поразрядную сортировку, которая фактически упорядочивает элементы, распределяя их на каждом этапе по подходящим группам. Теперь же мы познакомимся с еще одним методом, который вместо разделения элементов *сливает* их вместе. Речь идет о *сортировке слиянием*.

Сортировка слиянием изначально исходит из того, что ее возможности ограничены. Она не пытается упорядочивать элементы, которые вообще никак не организованы. Вместо этого она занимается объединением двух групп элементов, каждая из которых уже отсортирована.

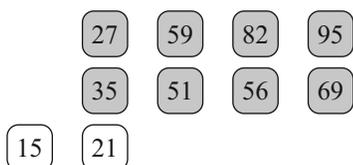
Представьте, к примеру, что наши элементы разделены на два ряда (в этом примере каждая группа состоит из одинакового количества элементов, но это вовсе не обязательно).



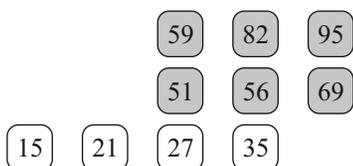
Как видите, каждый ряд уже отсортирован. Мы хотим их объединить, чтобы создать единую упорядоченную группу. Это очень просто. Мы сравниваем первые элементы каждой группы. В нашем случае число 15 меньше, чем 21, поэтому с него будет начинаться третья группа.



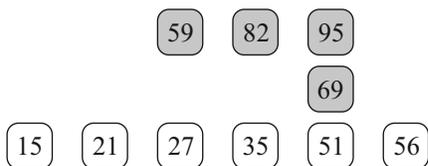
Снова проверяем первые элементы в двух группах. На этот раз 21 из второй группы меньше, чем 27 из первой. Поэтому мы берем число 21 и добавляем его в третью группу.



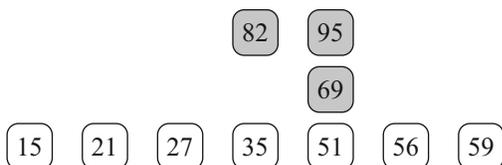
Продолжим в том же духе. Возьмем 27 из первой группы и 35 из второй и добавим их в конец третьей.



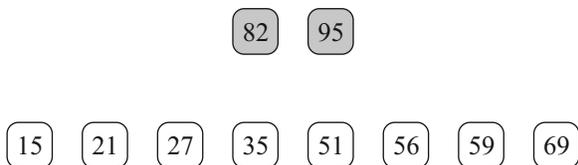
Теперь 51 меньше 59 и 56 меньше 59. Поскольку число 35 уже попало в конец третьей группы, получается, что мы выбрали из второй группы три элемента подряд. Это нормально, поскольку таким образом мы упорядочиваем элементы в третьей группе. Первые две группы вовсе не должны уменьшаться одинаковыми темпами.



Возвращаемся к первой группе. Число 59 меньше, чем 69, поэтому добавляем его в третью группу.



После перемещения 69 вторая группа полностью исчерпывается.



В конце мы перемещаем оставшиеся элементы первой группы в третью — они точно больше последнего перемещенного элемента, иначе мы их уже переместили бы. Итак, наши элементы полностью отсортированы.



Это хорошо, что мы можем получить единую упорядоченную группу из двух отдельных, но как это поможет нам отсортировать список неупорядоченных элементов? Действительно, никак. Тем не менее это важный аспект общего решения.

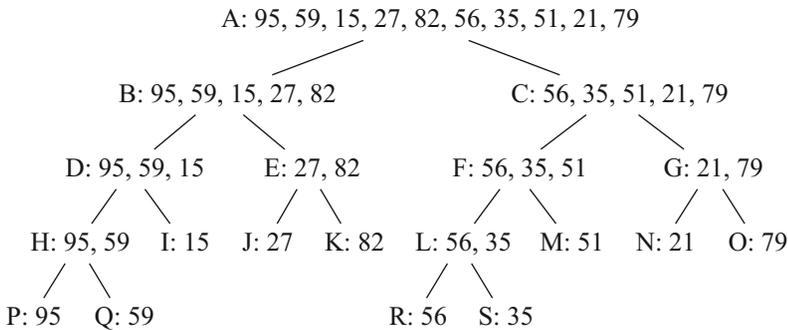
Представьте, что у вас есть группа людей. Мы поручаем одному человеку отсортировать набор элементов. Этот человек не знает, как сортировать, но зато он умеет объединять два набора отсортированных элементов в итоговый упорядоченный набор. Поэтому он разделяет набор элементов на две части и передает их двум другим людям. Каждому из них он говорит следующее: «Возьми эти элементы, отсортируй их и верни мне».

Первый человек не умеет сортировать, но если два других упорядочат элементы, которые им доверили, и вернут их ему, он сможет составить из них

итоговый отсортированный набор. Проблема в том, что два других человека знают о сортировке не больше первого: они могут лишь объединять уже упорядоченные наборы с помощью алгоритма слияния. Так чего же мы этим добились?

На самом деле мы решили задачу, если исходить из того, что все люди в группе делают одно и то же: разделяют полученные элементы на две части, делегируют их двум другим людям, затем ждут, когда те завершат свою работу и вернут два отсортированных набора.

Это выглядит как перекалывание ответственности из рук в руки, но давайте посмотрим, что произойдет, если применить данный метод к нашему примеру. Вначале у нас есть числа 95, 59, 15, 27, 82, 56, 35, 51, 21 и 79. Мы отдаем их Элис (А), которая разделяет их на две части и передает Бобу (В) и Кэрол (С). Это можно наблюдать на первом уровне перевернутого дерева, представленного ниже.

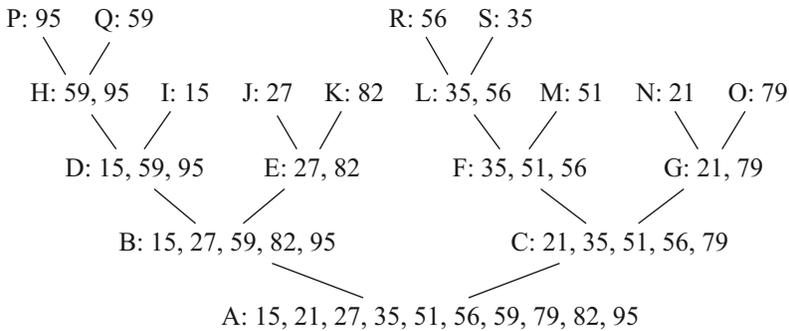


Затем Боб разделяет свои числа на две части и передает их Дейву (D) и Иви (E). Тем временем Кэрол разделяет свои числа между Фрэнком (F) и Грейс (G). Ответственность продолжает передаваться из рук в руки. Дейв разделяет числа между Хайди (H) и Иваном (I); Иви распределяет свои два числа между Джуди (J) и Карен (K); Фрэнк и Грейс передают элементы Лео (L) и Мэллори (M) и, соответственно, Нику (N) и Оливии (O). В конце Хайди разделяет свою пару между Пегги (P) и Квентином (Q), а Лео — между Робертом (R) и Сибил (S).

Тем, кто находится на листьях дерева, не остается никакой работы. Пегги и Квентин получили по одному числу и теперь должны выполнить сортировку. Но одно число уже и так упорядочено по определению: оно находится в правильном положении относительно самого себя. Поэтому Пегги и Квентин просто возвращают свои числа Хайди. То же самое делают Иван, Джуди, Карен, Роберт, Сибил, Мэллори и Оливия.

Теперь давайте перейдем к трем сортировщикам на следующей странице. В этом дереве мы двигаемся от листьев, расположенных сверху, к корню, который находится внизу (мы перевернули диаграмму, чтобы она больше походила на нормальное дерево). Давайте сосредоточимся на Хайди. Она получает обратно два числа, каждое из которых (само собой) отсортировано. Хайди знает, как объединить две группы упорядоченных чисел в одну, поэтому она получает набор 59, 95. Затем она возвращает его Дейву. То же самое делает и Лео: он получил числа 35 и 56, которые уже (сами собой) отсортированы, поэтому он объединяет их в упорядоченный набор 35, 56 и возвращает Фрэнку.

Дейв, который сначала не знал, что делать с числами 95, 59, 15, получил набор 59, 95 от Хайди и 15 от Ивана. Оба эти набора уже отсортированы, поэтому Дейв может их объединить в 15, 59, 95. Точно так же Фрэнк, получив 35, 56 от Лео и 51 от Мэллори, может вернуть 35, 51, 56.



Если все продолжат в том же духе, то к моменту, когда очередь дойдет до Элис, останется два отсортированных списка: один от Кэрол, а другой от Боба. Элис объединит их в итоговый упорядоченный список.

Эти два дерева лежат в основе сортировки слиянием. Упорядочивание делегируется настолько, насколько это возможно, вплоть до того, что его вообще не нужно проводить, поскольку отдельные элементы и так упорядочены сами по себе. Затем мы «сливаем» все большие и большие наборы, пока не объединим все элементы в финальный отсортированный список.

Каждый из участников должен обладать лишь минимальными умениями. Вы можете видеть, что в первом дереве Иви получила от Боба набор чисел, которые оказались уже упорядоченными: 27, 82. Но это неважно. Она не принялась проверять, в каком порядке они находятся, и мы не хотели, чтобы она тратила на это время. Она просто разделила эти числа и передала их дальше. Позже она получила их обратно и создала тот же набор, который у нее был

изначально. Это нормально; такое бессмысленное па-де-труа в исполнении Иви, Джуди и Карен не оказало особого влияния на производительность алгоритма.

Сложность сортировки слиянием не хуже, чем у быстрой сортировки, $O(n \lg n)$. Это означает, что у нас есть два алгоритма с одинаковой производительностью. На практике программисты руководствуются в своем выборе дополнительными факторами. Программы с быстрой сортировкой обычно работают быстрее программ на основе сортировки слиянием, потому что они лучше реализованы на конкретном языке программирования. В сортировке слиянием данные сначала разделяются, а затем объединяются; это означает, что этот процесс можно распараллелить так, чтобы огромные объемы информации сортировались в компьютерном кластере. В этом случае каждый компьютер играет роль отдельного сортировщика.

Этот подход появился вместе с первыми компьютерами. Его автором является американец венгерского происхождения, Янош Лайош Нейман, более известный под своим американским именем, Джон фон Нейман (1903–1957). В 1945 году он написал от руки 23-страничный документ, посвященный одному из первых цифровых компьютеров, EVDAC (Electronic Discrete Variable Automatic Computer). Вверху первой страницы было написано карандашом (и позже стерто) «СОВЕРШЕННО СЕКРЕТНО», так как в 1945 году работу над компьютерами тесно связывали с военными и поэтому засекретили. Темой документа стало нечисленное применение компьютеров: сортировка. Метод, описанный фон Нейманом, позже стал известен как сортировка слиянием.

Если вы достаточно молоды, слова *HotBot*, *Lycos*, *Excite*, *AltaVista* и *Infoseek* могут ничего для вас не значить. Но даже если вы о них слышали, то, наверное, не как о поисковых системах. Тем не менее какое-то время назад все они боролись за наше внимание, пытались стать нашим окном в Интернет.

Но это дела уже давно минувших дней, так как на рынке поисковых систем сейчас доминируют Google от Aphabet и Bing от Microsoft. Появление множества конкурирующих решений на новом рынке и их последующая консолидация — привычное явление во многих отраслях. Однако рынок поисковых систем примечателен тем, что большое влияние на его развитие оказал успех Google, ставший возможным благодаря алгоритмам, которые разработали основатели этой компании. Ларри Пейдж и Сергей Брин, получившие свои докторские степени в Стэнфордском университете, назвали изобретенный ими алгоритм PageRank (в честь Пейджа, а не от слов page и rank, как можно было бы подумать).

Прежде чем приступить к описанию PageRank, необходимо понять, чем именно занимаются поисковые системы. В действительности ответ состоит из двух частей. Во-первых, они обходят Интернет, читая и индексируя все веб-страницы, которые им встречаются. Таким образом, когда вы вводите свой запрос, поисковая система анализирует уже сохраненные ею данные и находит то, что вы ищете. Например, если ввести «изменение климата», система пройдет по данным, которые она насобираала, и выдаст веб-страницы, содержащие эти ключевые слова.

Если наш поисковый запрос относится к популярной теме, результатов может быть огромное множество. На момент написания этой книги запрос «изменение климата» в Google выдает более 700 миллионов результатов; когда вы будете читать эти строки, цифры могут быть другими, но вы можете себе представить, о каких масштабах идет речь. Это подводит нас ко второй составляющей того, чем занимаются поисковые системы. Они должны

представить результаты поиска таким образом, чтобы они выводились в порядке их актуальности: сначала должны идти страницы, которые имеют самое прямое отношение к нашему запросу, а затем то, что может нас не заинтересовать. Если вам интересны факты, связанные с изменением климата, в числе первых результатов следует ожидать страницы, принадлежащие ООН, NASA или «Википедии». Вы, наверное, удивились бы, увидев в качестве первого результата веб-страницу с описанием того, что на эту тему думают члены Общества плоской Земли. Из тех сотен миллионов страниц, которые могут относиться к вашему запросу, многие окажутся малоинформативными или слишком многословными, но при этом содержащими сущую чепуху. Очевидно, вам захочется сосредоточиться на достоверной информации, которая относится к делу.

Когда на сцене появилась поисковая система Google (автор родился достаточно давно, чтобы застать этот момент), люди (включая автора) стали переходить на нее с более старых и уже несуществующих систем, так как она выдавала лучшие результаты и делала это быстрее. На руку компании Google сыграло и то, что ее веб-страница была простой и содержала только нужную информацию, хотя в то время зачастую использовали всевозможные броские элементы. Второй фактор оказался весьма поучительным (в Google понимали, что пользователям нужны качественные и быстрые поисковые результаты, а не всякие прибабасы), но мы сосредоточим наше внимание на первом. Как Google удалось превзойти конкурентов по качеству результатов и скорости их выдачи?

Если бы Интернет был небольшим, мы могли бы поручить группе редакторов составить его каталог и назначить его записям (веб-страницам) разные степени важности. Но масштаб Интернета делает такой подход невозможным, хотя попытки этого предпринимались, пока не стало очевидно, что это непосильная задача.

Интернет состоит из веб-страниц, связанных между собой так называемыми *гиперссылками*; текст, содержащий перекрестные ссылки на другие свои участки или внешние документы, называется *гипертекстом*. Понятие гипертекста предшествовало появлению Всемирной паутины. Описание системы для организации знаний путем взаимного связывания документов впервые предложил американский инженер Вэнивар Буш в 1945 году в журнале *Atlantic*. Всемирную паутину, или просто *веб*, как ее позже начали называть, разработал британский ученый в области информатики Тим Бернерс-Ли в 1980-х. Бернерс-Ли работал в CERN, Европейской организации по ядерным исследованиям, рядом с Женевой, Швейцария, и хотел создать систему,

Если вам интересны факты, связанные с изменением климата... вы, наверное, удивились бы, увидев в качестве первого результата веб-страницу с описанием того, что на эту тему думают члены Общества плоской Земли.

которая помогла бы ученым обмениваться документами и информацией. Они могли делать свои документы доступными онлайн и ссылаться в них на другие документы, тоже доступные онлайн. Веб начал и до сих пор продолжает развиваться естественным образом за счет того, что люди добавляют новые веб-страницы. Авторы этих страниц добавляют ссылки на существующие страницы, которые имеют к ним какое-то отношение.

Представьте, что вы написали и разместили онлайн статью, в которой проводится краткий обзор эффектов изменения климата в вашей стране. Было бы неплохо, если бы читатель мог перейти на веб-страницу, которая, по вашему мнению, является достоверным источником на данную тему. Поэтому вы добавили в свою статью соответствующую ссылку. Таким образом вы помогаете своим читателям, позволяя им ознакомиться с более глубоким материалом, и в то же время делаете собственную работу более солидной, подкрепляя свои утверждения информацией с веб-страницы, которой доверяете.

Но есть много других людей, которые тоже пишут статьи об эффектах изменения климата в своих странах или регионах. И все они тоже хотят сослаться на достоверный, по их мнению, источник. Таким образом все эти онлайн-статьи будут содержать гиперссылки, указывающие на подходящие источники информации.

Причина, по которой NASA может выводиться на первом месте при поиске «изменения климата», связана с тем, что множество авторов, пишущих собственные статьи, решили разместить у себя гиперссылку на веб-страницу NASA, посвященную этой теме. Авторы приняли это решение независимо друг от друга, но многие из них, скорее всего, сослались на одну и ту же страницу (например, на страницу веб-сайта NASA). Поэтому данную страницу логично считать важной по отношению к другим веб-страницам.

Вся система работает по принципу демократии. Авторы одних веб-страниц ссылаются на другие. Чем больше ссылок указывает на страницу, тем более важной она считается, тем больше причин на нее сослаться, и тем важнее она в итоге становится.

Но у этого подхода есть одно концептуальное отличие от того, как мы практикуем демократию. Не все написанные статьи равны. Некоторые из них публикуются на более престижных веб-сайтах, чем другие. Статья в блоге, прочитанная горсткой людей, имеет меньший вес, чем статья в онлайн-издании с сотнями тысяч читателей. Из этого следует, что для оценки важности веб-страницы следует учитывать не только количество ссылок, которые на нее ведут. Большую роль также играет то, кто именно на нее ссылается.

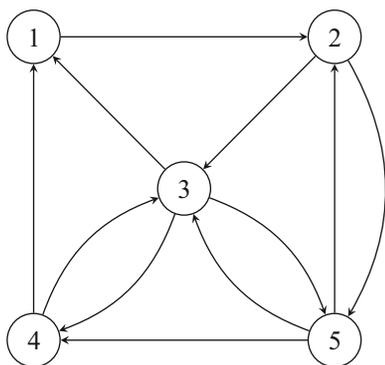
Вся система работает по принципу демократии. Авторы одних веб-страниц ссылаются на другие. Чем больше ссылок указывает на страницу, тем важнее она в итоге становится.

Было бы разумно предположить, что ссылка в престижном издании является более весомой, чем ссылка, размещенная на малоизвестном сайте. Конечно, не стоит судить книгу по обложке, но одобрение от выдающегося автора значит больше, чем положительный отзыв неизвестного рецензента. Каждая ссылка — это своего рода одобрение, выраженное одной страницей в отношении к другой, и весомость этого одобрения зависит от статуса его автора. В то же время, если страница ссылается на множество других источников, ее одобрение должно быть поделено между ними.

Набор страниц, связанных гиперссылками, составляет огромный граф, содержащий миллиарды страниц и еще большее количество ссылок между ними. Каждая веб-страница выступает узлом этого графа. Каждая ссылка из одной страницы на другую является направленным ребром. Фундаментальной особенностью PageRank было то, что по причинам, которые мы только что описали, структуру этого веб-графа можно использовать для определения важности каждой веб-страницы. Если быть более точным, каждой странице присваивается число — рейтинг, который определяет, насколько она важна относительно других веб-страниц. Чем важнее страница, тем выше ее рейтинг. Алгоритм PageRank применяет этот принцип в грандиозных масштабах — в графе, который представляет весь веб.

Основные принципы

Зайдя на любую веб-страницу, можно увидеть ссылки на другие ресурсы, имеющие какое-то отношение к текущему. Само наличие ссылки говорит о важности веб-страницы, на которую он ведет, иначе автор просто не добавил бы ее на свой сайт. Взгляните на следующий граф, представляющий небольшой набор веб-страниц, ссылающихся друг на друга.



Ссылки такого графа, указывающие на веб-страницы, называются *обратными*; поэтому мы будем также называть *обратными ссылками* страницы, на которых они размещены. Таким образом, обратные ссылки веб-страницы 3 являются ребрами, которые ведут к ней (входными ребрами), а также узлами, из которых они исходят: веб-страницы 2, 4 и 5. В этой главе нас интересуют графы, состоящие из веб-страниц, поэтому термины «узел» и «страница» будут взаимозаменяемыми.

Мы создадим алгоритм для определения важности каждой веб-страницы, основанный на следующих двух принципах.

1. Важность веб-страницы зависит от важности тех страниц, которые на нее ссылаются, то есть от важности ее обратных ссылок.
2. Веб-страница разделяет свою важность поровну между страницами, на которые она ссылается.

Представьте, что нам нужно определить важность страницы 3. Мы знаем, что у нее есть обратные ссылки 2, 4 и 5. Возьмем по очереди каждую из них и предположим, что нам известна их важность. Страница 2 разделяет свою важность между страницами 3 и 5; следовательно, странице 3 отводится только половина. Страница 4 тоже разделяет свою важность между двумя страницами, 3 и 1, поэтому странице 3 достается половина ее важности. Наконец, страница 5 разделяет свою важность между страницами 2, 3 и 4, поэтому страница 3 получает треть ее важности. Чтобы не быть слишком многословными, обозначим важность страницы i как $r(P_i)$; r от слова *rank* («рейтинг»). Таким образом, страница 3 будет иметь следующую важность.

$$r(P_3) = \frac{r(P_2)}{2} + \frac{r(P_4)}{2} + \frac{r(P_5)}{3}$$

В целом, если нам нужно узнать важность определенной веб-страницы, и при этом нам уже известна важность всех ее обратных ссылок, процесс будет довольно простым: важность каждой обратной ссылки нужно разделить на количество веб-страниц, на которые она ссылается, и сложить все полученные результаты.

Вычисление важности — это своего рода конкурс, на котором веб-страницы голосуют друг за друга. Каждая страница имеет голос определенного веса, который она может отдать за важные, по ее мнению, ресурсы. Если она считает важной только один ресурс, ему достается весь ее голос. Если таких

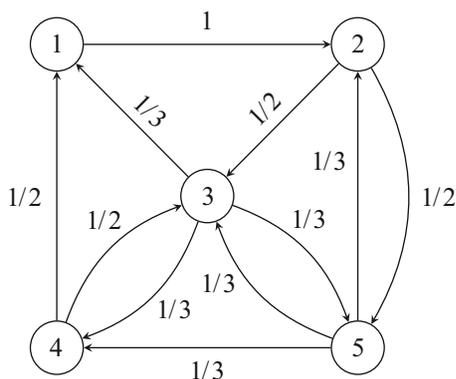
ресурсов несколько, часть голоса достается каждому из них. Таким образом, если веб-страница хочет проголосовать за три важных ресурса, каждый из них получит одну треть ее голоса. Каким ресурсам должна отдать свой голос веб-страница? Тем, на которые ведут ее гиперссылки, то есть тем, на которые она ссылается. И чем определяется важность веб-страницы? Важностью ее обратных ссылок.

Эти два принципа придают ранжированию веб-страниц ауру демократичности. Нет какой-то одной стороны, которая определяет, что важно, а что нет. Важность веб-страницы зависит от ресурсов, которые на нее ссылаются, и эти ресурсы голосуют своими ссылками. Но, в отличие от настоящих выборов в большинстве стран мира, не все участники имеют одинаковые голоса. Голос веб-страницы зависит от ее значимости, которая, опять-таки, определяется другими веб-страницами.

Это может показаться казуистикой, так как для определения важности веб-страницы нам фактически нужно определить важность ее обратных ссылок. Если следовать тому же принципу, то важность каждой обратной ссылки определяется важностью ее собственных обратных ссылок. Этот процесс может продолжаться сколько угодно, и в итоге мы так и не узнаем, как вычислить значимость веб-страницы, с которой все началось. Что еще хуже, может обнаружиться, что мы ходим по кругу. В нашем примере для вычисления важности страницы 3 нужно определить, насколько важны страницы 2, 4 и 5. Чтобы вычислить важность страницы 2, нужно знать важность страницы 1 (и 5, но давайте пока оставим это в стороне). Важность страницы 1 определяется важностью страницы 4, для получения которой нужно знать важность страницы 3. Мы вернулись к тому, с чего начали.

Пример

Чтобы понять, как выйти из этой ситуации, представим, что еще до вычисления их важности всем страницам присвоен одинаковый рейтинг. Если вернуться к нашему сравнению с голосованием, каждая веб-страница получает ровно один голос. Когда начинается голосование, каждая страница распределит свой голос между страницами, на которые она ссылается, — так, как описывалось ранее. Затем каждая страница получит голоса ото всех своих обратных ссылок. Передача голосов будет выглядеть так:



Страница 1 отдает свой голос странице 2 (единственному ресурсу, на который она ссылается). Страница 2 разделяет свой голос на две части и отправляет по $1/2$ страницам 3 и 5. Страница 3 разделяет свой голос на три части и передает по $1/3$ страницам 1, 4 и 5. Страницы 4 и 5 голосуют аналогичным образом.

По окончании голосования каждая страница подсчитает итоговую сумму голосов (или их частей), которые она получила от своих обратных ссылок. Например, страница 1, получившая голоса от страниц 3 и 4, будет иметь $1/2 + 1/3 = 5/6$ голоса, а страница 3, получившая голоса от страниц 2, 4 и 5, будет иметь $1/2 + 1/2 + 1/3 = 4/3$ голоса. Мы видим, что изначальная доля голосов страницы 1 уменьшилась, а у страницы 3 она увеличилась.

Давайте немного изменим условия и выдадим каждой странице перед началом голосования не один голос, а $1/5$, чтобы в сумме получалась единица. В целом, если у нас есть n страниц, каждая из них получает $1/n$ голоса. В остальном процесс не меняется. Общая важность снова распределяется равномерно между всеми веб-страницами, но на этот раз она равна 1.

По окончании голосования важность каждой веб-страницы изменится. Раньше все они имели одинаковый рейтинг, $1/5 = 0,2$, но после вычислений их рейтинги будут следующими (по порядку): 0,17, 0,27, 0,27, 0,13 и 0,17. Страницы 2 и 3 стали более важными, а страницы 1, 4 и 5 потеряли в значимости. Суммарная важность всех веб-страниц равна 1.

Теперь можно начать следующий тур голосования с использованием тех же правил. Страницы распределяют собранные ими голоса между ресурсами, на которые они ссылаются. В конце второго тура каждая страница подсчитает свои голоса, чтобы определить собственную относительную значимость. После всех вычислений страницы будут иметь такие рейтинги: 0,16, 0,22, 0,26, 0,14 и 0,22.

Еще раз повторим этот же процесс. На самом деле мы будем повторять его снова и снова. В результате голоса (то есть важность каждой веб-страницы) будут изменяться так, как показано в следующей таблице. Вы можете видеть начальные значения и результаты после каждого тура голосования.

Тур	Страница 1	Страница 2	Страница 3	Страница 4	Страница 5
начало	0,20	0,20	0,20	0,20	0,20
1	0,17	0,27	0,27	0,13	0,17
2	0,16	0,22	0,26	0,14	0,22
3	0,16	0,23	0,26	0,16	0,20
4	0,17	0,22	0,26	0,15	0,20
5	0,16	0,23	0,25	0,15	0,20
6	0,16	0,23	0,26	0,15	0,20

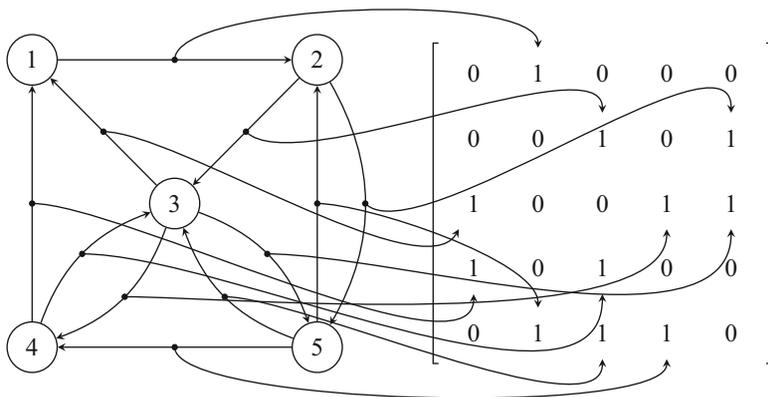
Если провести еще один, седьмой по счету тур голосования, ситуация не поменяется по сравнению с предыдущим туром. Голоса и, следовательно, важность веб-страниц останутся прежними. Таким образом мы получим конечный результат. Согласно вычисленному рейтингу, самой важной является страница 3; за ней идет страница 2, затем страница 5, затем 1 и в самом конце страница 4.

Давайте остановимся на минуту и подумаем о том, что мы только что сделали. Мы отталкивались от двух принципов, которые определили правила для вычисления важности веб-страниц на основе важности их обратных ссылок. Прежде чем начинать вычисления, мы назначили n веб-страницам одинаковый рейтинг, равный $1/n$. Затем мы посчитали важность каждой веб-страницы, сложив те доли голосов, которые они получили от своих обратных ссылок. Это дало нам новые рейтинги веб-страниц, которые отличались от изначальных, $1/n$. Мы повторили весь процесс, но уже с новыми значениями, и получили еще один набор рейтингов. После ряда повторений обнаружилось, что ситуация стабилизировалась: степень важности перестала меняться при последующих повторениях. В этот момент мы остановились и сообщили о найденных нами значениях.

Вопрос, конечно в том, работает ли описанный нами подход в целом, а не только в этом конкретном примере. И, что еще важнее, дает ли он осмысленные результаты?

Матрица гиперссылок и степенной метод

У метода вычисления важности страницы на основе важности ее обратных ссылок есть элегантное определение. Все начинается с графа, который описывает связи между нашими веб-страницами. Этот граф можно представить в виде *матрицы* чисел; назовем ее *матрицей смежности*. Создать ее довольно просто. Количество ее строк и столбцов должно быть равно числу узлов в графе. Затем каждое пересечение, относящееся к ссылке, будет обозначено единицей, а все остальные пересечения — нолями. Матрица смежности для нашего примера выглядит так:



Важность веб-страниц можно представить с помощью отдельной строки или вектора.

$$[r(P_1) \quad r(P_2) \quad r(P_3) \quad r(P_4) \quad r(P_5)]$$

Поскольку мы начали рассматривать внутреннее устройство алгоритма PageRank, давайте будем называть важность веб-страницы рейтингом. В уместности этого термина вы сможете убедиться, когда мы вычислим рейтинги (то есть важность) всех страниц, доступных онлайн. Поскольку наша строка содержит все рейтинги, мы будем называть ее *рейтинговым вектором* графа.

Важность каждой веб-страницы разделена между ресурсами, на которые она ссылается. Теперь, когда у нас есть матрица смежности, мы можем пройти по каждой строке и поделить каждую единицу на количество единиц в этой строке. Это то же самое, что разделить голос каждой страницы между всеми ее исходящими ссылками. Если это сделать, получится следующая матрица.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/3 & 1/3 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 1/3 & 1/3 & 1/3 & 0 \end{bmatrix}$$

Назовем ее *матрица гиперссылок*.

Если внимательно присмотреться к этой матрице, можно заметить, что каждый ее столбец иллюстрирует вычисление важности страницы на основе рейтингов страниц, ссылающихся на нее. Возьмем первый столбец, который относится к странице 1; ее важность основана на страницах 3 и 4. Страница 3 дает ей $1/3$ своего рейтинга, а страница 4 — $1/2$, так как она ссылается на две страницы. Страница 1 не получает рейтинг ни от каких других страниц в графе, поскольку они на нее не ссылаются. Это можно выразить так:

$$r(P_1) \times 0 + r(P_2) \times 0 + \frac{r(P_3)}{3} + \frac{r(P_4)}{2} + r(P_5) \times 0 = \frac{r(P_3)}{3} + \frac{r(P_4)}{2}$$

Но это точное определение $r(P_1)$, рейтинга страницы 1. Мы получили рейтинг, сложив произведения элементов рейтингового вектора с соответствующими элементами первого столбца матрицы гиперссылок.

Давайте посмотрим, что получится, если взять рейтинговый вектор и сложить произведения его элементов с соответствующими элементами второго столбца матрицы гиперссылок.

$$r(P_1) \times 1 + r(P_2) \times 0 + r(P_3) \times 0 + r(P_4) \times 0 + \frac{r(P_5)}{3} = r(P_1) + \frac{r(P_5)}{3}$$

Это точное определение $r(P_2)$, рейтинга страницы 2. Аналогичным образом сумма произведений элементов рейтингового вектора и содержимого третьего столбца матрицы гиперссылок даст нам $r(P_3)$, рейтинг страницы 3.

$$r(P_1) \times 0 + \frac{r(P_2)}{2} + r(P_3) \times 0 + \frac{r(P_4)}{2} + \frac{r(P_5)}{3} = \frac{r(P_2)}{2} + \frac{r(P_4)}{2} + \frac{r(P_5)}{3}$$

Можете убедиться в том, что четвертый и пятый столбцы матрицы гиперссылок получают $r(P_4)$ и, соответственно, $r(P_5)$. Операция сложения произведений элементов рейтингового вектора и содержимого каждого столбца матрицы в действительности является произведением этой матрицы и рейтингового вектора.

Если вы незнакомы с матричными операциями, этот процесс может показаться запутанным, поскольку, когда речь идет о произведении, обычно имеется в виду умножение двух чисел, а не таких вещей, как векторы и матрицы. Математические операции при желании можно проводить и с другими сущностями, а не только с числами. Примером таких операций является произведение вектора и матрицы. В этом нет ничего таинственного: это просто операция, которую мы выполняем с элементами вектора и матрицы.

Представьте, что мы выпекаем рогалики и круассаны, продавая их по \$2 и, соответственно, \$1,5. У нас есть два магазина; за отдельно взятый день один магазин продает 10 рогаликов и 20 круассанов, а другой — 15 рогаликов и 10 круассанов. Как вычислить общие продажи в каждом из магазинов?

Чтобы узнать доход первого магазина, умножим цену рогалика на количество проданных рогаликов в этом магазине. То же самое сделаем с круассанами и затем сложим эти два результата:

$$2 \times 10 + 1,5 \times 20 = 50$$

Проведем те же вычисления для второго магазина:

$$2 \times 15 + 1,5 \times 10 = 45$$

Это можно выразить более лаконично, если записать цены рогаликов и круассанов в виде вектора:

$$[2,00 \quad 1,50]$$

Теперь запишем в виде матрицы суточные продажи. Матрица будет состоять из двух столбцов, по одному на магазин, и двух строк, по одной для рогаликов и круассанов:

$$\begin{bmatrix} 10 & 15 \\ 20 & 10 \end{bmatrix}$$

Теперь, чтобы найти общие продажи в каждом магазине, умножим элементы вектора на каждый столбец матрицы продаж и результаты просуммируем. Таким образом получится произведение вектора и матрицы:

$$\begin{aligned} [2,00 \quad 1,50] \times \begin{bmatrix} 10 & 15 \\ 20 & 10 \end{bmatrix} \\ = [2,00 \times 10 + 1,50 \times 20 \quad 2,00 \times 15 + 1,50 \times 10] \\ = [50 \quad 45] \end{aligned}$$

Произведение вектора и матрицы — это частный случай произведения двух матриц. Давайте расширим наш пример и заменим вектор с ценами рогаляков и круассанов матрицей с ценами и прибылью для каждой продажи:

$$\begin{bmatrix} 2,00 & 1,50 \\ 0,20 & 0,10 \end{bmatrix}$$

Чтобы найти общие продажи и прибыль для каждого магазина, создадим такую матрицу, в которой записи i -й строки и j -го столбца будут суммой произведений i -й строки в матрице с ценами и прибылью и j -го столбца матрицы продаж. Это определение произведения двух матриц:

$$\begin{aligned} \begin{bmatrix} 2,00 & 1,50 \\ 0,10 & 0,20 \end{bmatrix} \times \begin{bmatrix} 10 & 15 \\ 20 & 10 \end{bmatrix} \\ = \begin{bmatrix} 2,00 \times 10 + 1,50 \times 20 & 2,00 \times 15 + 1,50 \times 10 \\ 0,10 \times 10 + 0,20 \times 20 & 0,10 \times 15 + 0,20 \times 10 \end{bmatrix} \\ = \begin{bmatrix} 50 & 45 \\ 5 & 3,5 \end{bmatrix} \end{aligned}$$

Вернемся к рейтингам страниц. На каждом этапе вычисление рейтингового вектора в действительности является произведением этого вектора из предыдущего этапа и матрицы гиперссылок. Переходя от этапа к этапу, мы получаем последовательные оценки рейтингов, то есть последовательные оценки рейтингового вектора, который из них состоит.

Чтобы получить эти последовательные оценки рейтингового вектора, нам нужно лишь умножить его на каждом этапе на матрицу гиперссылок и получить тем самым вектор для следующего этапа.

На первом этапе все элементы рейтингового вектора равны $1/n$, где n — количество страниц. Если обозначить исходный рейтинговый вектор как π_1 , вектор в конце первого этапа как π_2 , а матрицу гиперссылок как H , получится

$$\pi_2 = \pi_1 \times H.$$

На каждом этапе мы берем текущий рейтинговый вектор и вычисляем его версию для следующего этапа. Во втором туре голосования, в котором мы получили третий столбец с оценками рейтингов (то есть наш третий рейтинговый вектор), мы выполнили следующие вычисления:

$$\pi_3 = \pi_2 \times H = (\pi_1 \times H) \times H = \pi_1 \times (H \times H) = \pi_1 \times H^2.$$

В третьем туре голосования мы получили наш четвертый рейтинговый вектор:

$$\pi_4 = \pi_3 \times H = (\pi_1 \times H^2) \times H = \pi_1 \times (H^2 \times H) = \pi_1 \times H^3.$$

На каждом этапе мы умножаем результат предыдущих вычислений на матрицу гиперссылок. В конце получается набор произведений последовательных оценок этой матрицы и рейтингового вектора. Как видим, это эквивалент умножения начального рейтингового вектора на растущие степени матрицы гиперссылок. Такой подход к вычислению последовательных приближенных значений называется *степенным методом*. Поэтому вычисление рейтингов веб-страниц является практическим применением степенного метода к рейтинговому вектору и матрице гиперссылок снова и снова, пока этот вектор не перестанет изменяться — или, как еще говорят, пока он не *придет* к стабильному значению, нашим итоговым рейтингам.

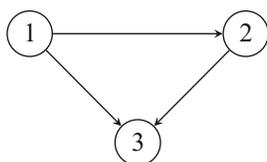
Итак, получили более точное определение того, как вычисляются рейтинги в веб-графе.

1. Формируем на основе графа матрицу гиперссылок.
2. Начинаем с исходной оценки рейтингов, назначая каждой странице рейтинг $1/n$, где n — общее количество страниц.
3. Применяем степенной метод, умножая рейтинговый вектор на матрицу гиперссылок до тех пор, пока значения вектора не сойдутся.

Эта сжатая формулировка позволяет переместить задачу в область линейной алгебры — раздела математики, посвященного матрицам и операциям с ними. Существуют устоявшиеся теоретические наработки, с помощью которых можно исследовать степенной метод и высокопроизводительные реализации операций с матрицами, такие как описанное выше умножение. Определение, основанное на матрицах, также поможет определить, *всегда ли* сходится степенной метод, то есть всегда ли мы можем найти рейтинги всех страниц в графе.

Висячие узлы и блуждающий пользователь

Давайте возьмем в качестве примера более простой граф, состоящий всего из трех узлов.



Нам нужно найти рейтинги этих трех узлов. Мы будем следовать тому же алгоритму. Инициализируем рейтинговый вектор путем назначения всем узлам одинакового рейтинга, $1/3$. Затем умножим рейтинговый вектор на матрицу гиперссылок.

$$\begin{bmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Если приступить к выполнению степенного метода, поэтапно обновляя рейтинговый вектор путем его умножения на матрицу гиперссылок, окажется, что после четырех этапов все рейтинги свелись к нулю.

Тур	Страница 1	Страница 2	Страница 3
начало	0,33	0,33	0,33
1	0,00	0,17	0,50
2	0,00	0,00	0,17
3	0,00	0,00	0,00

Это явная проблема. Мы не рассчитывали на то, что все страницы будут иметь нулевую важность. В конце концов, у страницы 3 есть две обратные ссылки, а у страницы 2 только одна, поэтому можно было бы ожидать, что это как-то отразится на результатах, не говоря уже о том, что общая сумма всех рейтингов должна быть равна 1. Но все пошло совсем не так, как мы надеялись.

Проблема кроется в узле 3. Наличие обратных ссылок должно было дать ему какой-то рейтинг, но дело в том, что у него отсутствуют выходные ссылки. Поэтому можно сказать, что он вобрал в себя рейтинги остальной части графа, но никуда их не распределил. Он ведет себя как эгоистичный узел или черная дыра: все, что в него заходит, никогда не выходит наружу. На протяжении нескольких этапов он вел себя как слив, в котором очутились значения всех рейтингов.

Такие узлы называются *висячими*, поскольку они висят (мертвым грузом) на окончаниях графа. В вебе существованию таких страниц ничто не препятствует. Конечно, страницы обычно содержат входные и выходные ссылки,

но вам вполне может встретиться страница без выходных ссылок, которая приведет в негодность степенной метод, описанный ранее.

Чтобы обойти эту проблему, воспользуемся метафорой. Представьте себе пользователя, который бродит по Интернету, переходя с одной страницы на другую. Для этого он обычно щелкает по ссылке. Но в какой-то момент пользователь заходит на висячий узел — страницу, которая не ссылается ни на какой другой ресурс. Мы не хотим, чтобы он там застрял, поэтому наделяем его возможностью перейти на любую другую страницу, доступную онлайн. Например, мы ходим по Интернету и случайно попадаем в тупик. Но это нас не останавливает. Мы всегда можем ввести другой адрес в нашем браузере и перейти на любой другой ресурс, даже если висячая страница на него не ссылается. Именно этого мы хотим от нашего пользователя. Если ему больше некуда идти, он выбирает произвольную страницу и переходит на нее, чтобы продолжить свое виртуальное путешествие. Пользователь становится *блуждающим*; у него в распоряжении есть устройство телепортации, с помощью которого он может мгновенно очутиться в совершенно любом месте.

Применим эту метафору к ранжированию страниц. Матрица гиперссылок дает нам вероятность, с которой пользователь перейдет по ссылке на определенную страницу. В нашем примере с тремя узлами первая строка матрицы гиперссылок говорит о том, что, находясь на странице 1, пользователь может в равной степени выбрать одну из двух страниц: 2 или 3. Во второй строке мы видим, что, находясь на странице 2, он всегда будет переходить на страницу 3. Вернемся на секунду к нашему первому примеру. Если пользователь попадет на страницу 5, перед ним открывается возможность перехода на страницы 2, 3 и 4 с одинаковой вероятностью, $1/3$.

Висячий узел проявляется в виде строки, состоящей из одних нулей. Вероятность перехода куда-либо является нулевой. Здесь нам поможет блуждающий пользователь. Как уже отмечалось, он может перейти на любую страницу графа. Это фактически означает, что в матрице гиперссылок больше не будет строк с нулями. Пользователь может выбрать любую веб-страницу с одинаковой вероятностью, поэтому вместо нулей эта строка будет содержать значения $1/n$ (в нашем примере это $1/3$). Таким образом матрица примет следующий вид:

$$\begin{bmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

Теперь пользователь, очутившийся на странице 3, может перейти на любую страницу графа с одинаковой вероятностью. Он даже может временно застрять на той же странице, но это неважно, так как он станет повторять свои попытки снова и снова, пока случайным образом не выберет другую страницу.

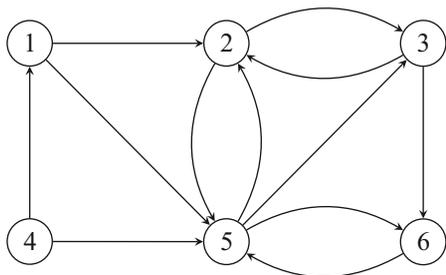
Мы будем называть эту модифицированную матрицу, в которой нулевые строки заменены строками со значениями вида $1/n$, S-матрицей. Если использовать ее в степенном методе, рейтинги изменятся следующим образом:

Тур	Страница 1	Страница 2	Страница 3
начало	0,33	0,33	0,33
1	0,11	0,28	0,61
2	0,20	0,26	0,54
3	0,18	0,28	0,54
4	0,18	0,27	0,55
5	0,18	0,27	0,54

На этот раз алгоритм сходится на ненулевых значениях; высасывания рейтингов не происходит. К тому же результаты выглядят осмысленными. Наивысший рейтинг получила страница 3 с двумя обратными ссылками; дальше идет страница 2 с одной обратной ссылкой. Страница 1, на которую никто не ссылается, заняла последнее место.

Матрица Google

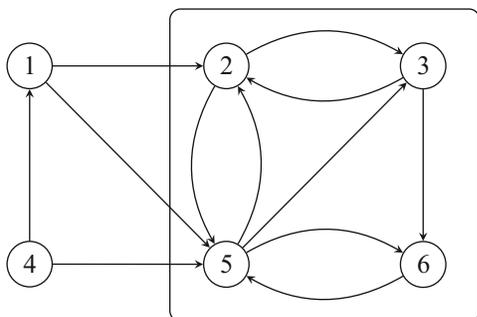
Кажется, проблема решена, но в более сложных ситуациях возникают аналогичные затруднения. У следующего графа нет висячих узлов.



Если выполнить алгоритм, у двух узлов, страниц 1 и 4, окажется нулевой рейтинг.

Тур	Страница 1	Страница 2	Страница 3	Страница 4	Страница 5	Страница 6
начало	0,17	0,17	0,17	0,17	0,17	0,17
1	0,08	0,22	0,14	0,00	0,42	0,14
2	0,00	0,25	0,25	0,00	0,29	0,21
3	0,00	0,22	0,22	0,00	0,33	0,22

Произошло следующее: несмотря на отсутствие висячих узлов, у нас все равно есть узлы, которые ведут себя в качестве слива для остального графа. Если присмотреться, можно заметить, что у узлов 2, 3, 5 и 6, если их объединить, есть только входные ссылки. Мы можем перейти в эту группу из узлов 1 или 4, но после этого перемещаться можно будет только внутри этой группы. Мы не можем выйти за ее пределы. Наш блуждающий пользователь окажется в ловушке, которая представляет собой не одну, а сразу несколько страниц, ссылающихся только друг на друга.



Нам опять нужно помочь блуждающему пользователю выбраться из этой ловушки. На этот раз в матрицу гиперссылок придется внести более комплексные изменения. Наша исходная матрица позволяла пользователю переходить между страницами только с помощью ссылок, существующих в графе. Затем мы ее модифицировали, чтобы избежать появления строк, состоящих из одних нулей; так у нас получилась S-матрица, позволявшая при попадании в висячий узел переходить в любое место графа. Теперь мы еще немного поменяем поведение блуждающего пользователя, модифицировав S-матрицу.

Сейчас, когда пользователь оказывается на узле, его возможные переходы определяются S-матрицей. В последнем примере ввиду отсутствия нулевых строк S-матрица ничем не отличалась от матрицы гиперссылок:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 1/2 & 0 & 0 & 0 & 1/2 \\ 1/2 & 0 & 0 & 0 & 1/2 & 0 \\ 0 & 1/3 & 1/3 & 0 & 0 & 1/3 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Если блуждающий пользователь окажется на странице 5, то, как показывает S -матрица, он может перейти только на страницу 2, 3 или 6, и в каждом случае вероятность будет равна $1/3$. Давайте сделаем его более проворным, чтобы он мог следовать S -матрице *не всегда*, а только с какой-то выбранной нами вероятностью a ; таким образом шансы на то, что он перейдет в любую точку графа, без учета S -матрицы, равны $(1 - a)$.

Способность переходить куда угодно откуда угодно означает, что в матрице никогда не может быть нулевых строк, ведь ноль обозначает переход, который невозможно выполнить. Чтобы этого достичь, нам придется *увеличить* значения нулевых записей и *уменьшить* значения ненулевых, чтобы вся строка всегда давала в сумме 1. Итоговые значения матрицы можно вычислить с помощью линейной алгебры, основываясь на S и вероятности a . Новая производная матрица называется *матрицей Google*; обозначим ее символом G . Если она определяет поведение блуждающего пользователя, все работает так, как мы того хотели: пользователь станет следовать S -матрице с вероятностью a и двигаться независимо с вероятностью $(1 - a)$. В нашем примере матрица Google будет выглядеть так:

$$\begin{bmatrix} \frac{3}{120} & \frac{54}{120} & \frac{3}{120} & \frac{3}{120} & \frac{54}{120} & \frac{3}{120} \\ \frac{3}{120} & \frac{3}{120} & \frac{54}{120} & \frac{3}{120} & \frac{54}{120} & \frac{3}{120} \\ \frac{3}{120} & \frac{54}{120} & \frac{3}{120} & \frac{3}{120} & \frac{3}{120} & \frac{54}{120} \\ \frac{54}{120} & \frac{3}{120} & \frac{3}{120} & \frac{54}{120} & \frac{3}{120} & \frac{3}{120} \\ \frac{3}{120} & \frac{37}{120} & \frac{37}{120} & \frac{3}{120} & \frac{3}{120} & \frac{37}{120} \\ \frac{3}{120} & \frac{3}{120} & \frac{3}{120} & \frac{3}{120} & \frac{105}{120} & \frac{3}{120} \end{bmatrix}$$

Сравните это с S -матрицей. Обратите внимание на то, что в первой строке две записи имели значения $1/2$, а стальные были нулевыми. Теперь же

в матрице Google две записи $1/2$ превратились в $54/120$, а остальные стали равны $3/120$ вместо 0. Аналогичные преобразования произошли и в других строках. Теперь, если блуждающий пользователь окажется на странице 1, вероятность перехода на страницы 2 и 5 составит $54/120$ в каждом из случаев, а вероятность перехода на любую другую страницу будет равна $3/120$.

Теперь мы можем дать окончательное определение алгоритму PageRank.

1. Формируем из графа матрицу Google.
2. Начинаем с исходных оценок рейтингов, назначая каждой странице рейтинг $1/n$, где n — общее количество страниц.
3. Применяем степенной метод, умножая рейтинговый вектор на матрицу Google до тех пор, пока не сойдутся значения этого вектора.

Мы просто заменили «матрицу гиперссылок», которая была в исходном алгоритме, «матрицей Google». Если пройтись этим алгоритмом по нашему графу со «сливной» группой узлов, получится следующее:

Тур	Страница 1	Страница 2	Страница 3	Страница 4	Страница 5	Страница 6
начало	0,17	0,17	0,17	0,17	0,17	0,17
1	0,10	0,14	0,14	0,10	0,31	0,21
2	0,07	0,15	0,17	0,07	0,31	0,23
3	0,05	0,14	0,18	0,05	0,32	0,26
4	0,05	0,14	0,17	0,05	0,33	0,27

Все сработало как нужно; мы больше не получаем нулевых рейтингов.

Степенной метод в сочетании с матрицей Google работает всегда. Линейная алгебра говорит о том, что он всегда сходится на финальном наборе рейтингов, сумма которых равна 1; на это не влияют ни висячие узлы, ни участки графа, которые высасывают рейтинги из остальных узлов. Нам даже не нужно делать начальные рейтинги равными $1/n$. Для инициализации можно использовать любые значения, которые в сумме дают единицу.

PageRank на практике

Итак, мы выработали метод поиска рейтингов в любом графе. Вопрос в том, являются ли конечные результаты осмысленными.

Рейтинговый вектор (в том виде, в котором мы его определили) играет особую роль в отношении матрицы Google. Когда степенной метод заканчивает работу, этот вектор больше не меняется. Следовательно, умножив его на матрицу Google, мы просто получим тот же вектор. В линейной алгебре это называется *первым собственным вектором* матрицы Google. Если не углубляться в математику, эта теория подтверждает, что этот вектор имеет особое значение для нашей матрицы.

Окончательным и важнейшим показателем того, что алгоритм PageRank подходит для назначения рейтингов веб-страницам, является то, насколько полезны его результаты людям. Поисковая система Google выдает хорошие результаты, то есть то, что мы, пользователи, получаем в ответ на наши запросы, соответствует нашему представлению о важной информации. Если бы рейтинговый вектор был математической диковинкой, не имеющей отношения к веб-страницам, он бы нас не заинтересовал.

Еще одно преимущество алгоритма PageRank в том, что его можно эффективно реализовать. Матрица Google имеет огромный размер; нам нужны одна строка и один столбец для каждой страницы в Интернете. Тем не менее матрица Google, как мы видели, является производной от S -матрицы, которая в свою очередь была получена из матрицы гиперссылок. Нам вовсе не нужно создавать и хранить саму матрицу Google; мы можем получить ее динамически, применяя матричные операции к матрице гиперссылок. Это удобно. В отличие от матрицы Google, у которых нет нулевых записей, матрица гиперссылок хранит множество нулей. Интернет может состоять из миллиардов страниц, но каждая из них ссылается лишь на горстку других ресурсов. Матрица гиперссылок является *разреженной*, то есть она в основном состоит из нулей, с небольшим количеством ненулевых значений (которых на несколько порядков меньше, чем нулевых). Благодаря этому мы можем хранить нашу матрицу с применением ловких методик, позволяющих размещать в памяти не все сразу, а лишь те позиции, в которых находятся ненулевые значения. Нам нужна не вся матрица гиперссылок, а только координаты ненулевых записей, занимающих намного меньше места. Это дает существенное преимущество при практическом применении алгоритма PageRank.

Но есть один важный нюанс. Нам известно, что этот алгоритм сыграл ключевую роль в успехе Google, но мы не знаем, в каком качестве он используется сейчас и используется ли вообще. Поисковая система Google постоянно развивается, и проводимые изменения не являются публичными. Известно, что Google использует историю поисковых запросов для оптимизации последующих результатов. Оптимизация может зависеть от страны, в которой мы

проживаем, или от общих тенденций в поисковых запросах, выполняемых людьми со всего мира. Все эти ингредиенты составляют секретный рецепт, с помощью которого Google улучшает свой продукт и удерживает позицию на рынке поисковых систем. Тем не менее все это не умаляет эффективности PageRank для назначения рейтингов веб-страницам, представленным в виде узлов графа.

PageRank подчеркивает еще один важный фактор успеха алгоритма. Это не только элегантность и эффективность, но и то, насколько он подходит для решения конкретной задачи. Это творческий аспект. Если мы хотим искать по Интернету, нам нужно преодолеть одну важную проблему — его огромный размер. Но если представить его в виде графа, размер из препятствия превратится в преимущество. Именно ввиду такого большого количества взаимосвязанных страниц можно ожидать, что для решения задачи подойдет метод, основанный на ссылочной структуре графа. Способность смоделировать задачу является первым шагом на пути к ее алгоритмическому решению.

ГЛУБОКОЕ ОБУЧЕНИЕ

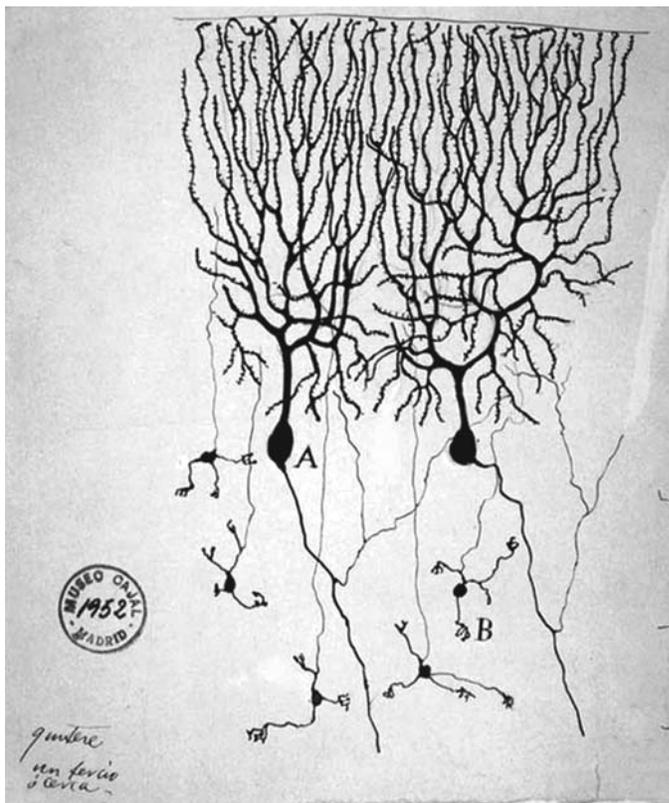
В последние годы приобрели популярность системы глубокого обучения. О них даже начали писать в средствах массовой информации. С их помощью компьютеры могут делать вещи, которые раньше считались прерогативой человека. Еще более неловким является тот факт, что эти системы часто преподносятся так, будто они работают подобно человеческому разуму. Это, конечно, намек на то, что ключом к искусственному интеллекту может быть эмуляция того, как мыслят люди.

Но, если не обращать внимания на рекламные преувеличения, большинство ученых, работающих в сфере глубокого обучения, не разделяют мнение о том, что эти системы похожи по своему принципу на разум человека. Их задача в том, чтобы продемонстрировать какое-то полезное поведение, которое зачастую ассоциируют с интеллектом. Однако мы не пытаемся копировать природу; строение человеческого мозга слишком сложное для того, чтобы эмулировать его на компьютере. Но мы все же поглядываем на природу, сильно упрощаем наши наблюдения и пытаемся проектировать системы, которые в некоторых сферах могут заменять биологические организмы, эволюционировавшие на протяжении миллионов лет. Более того, системы глубокого обучения можно рассматривать с точки зрения алгоритмов, которые они используют, чем мы и займемся в этой главе. Это поможет получить некоторое представление о том, что именно и как они делают. Благодаря этому мы сможем увидеть, что за их сложным поведением скрываются простые фундаментальные принципы. Вы убедитесь в том, что для извлечения из них какой-то пользы требуется недюжинная изобретательность.

Чтобы понять, что такое глубокое обучение, нужно начать с менее масштабных и более скромных вещей. На их основе мы постепенно сформируем более развернутую картину. В конце этой главы вы будете знать, что такого «глубокого» в этом обучении.

Нейроны, настоящие и искусственные

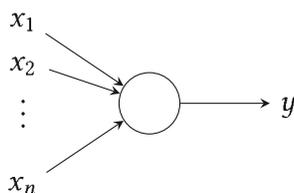
Нашей отправной точкой будет главная составляющая систем глубокого обучения, позаимствованная из биологии. Мозг — это часть нервной системы, основными компонентами которой выступают клетки под названием *нейроны*. Нейронам присуща определенная форма; они не похожи на шарообразные структуры, которые обычно ассоциируют с клетками. Ниже представлено одно из первых изображений нейронов, нарисованное в 1899 году испанцем Сантьяго Рамоном-и-Кахалем, отцом современных нейронаук.



В центре изображения находятся два нейрона голубиного мозга. Как видите, нейрон состоит из тела клетки и отростков, которые из него исходят. Эти отростки соединяются друг с другом с помощью *синапсов*, объединяя нейроны в сеть. Нейроны асимметричны. На одной стороне каждого из них расположено множество отростков, а на другой — всего один. Сторону с большим количеством отростков можно считать вводом нейрона, а противоположную сторону — его выводом. Ввод имеет вид электрических сигналов,

поступающих по входным отросткам. Многие нейроны могут также отправлять сигналы. Чем больший ввод принимает нейрон, тем большая вероятность того, что он пошлет (*сгенерирует*) сигнал. В этом случае говорят, что нейрон *активируется*.

Человеческий мозг — это огромная сеть нейронов, которых насчитывается порядка ста миллиардов, и каждый из них соединен в среднем с тысячей других. У нас нет возможности создать ничего подобного, но мы можем строить системы на основе упрощенных, идеализированных моделей нейронов. Модель искусственного нейрона показана ниже.



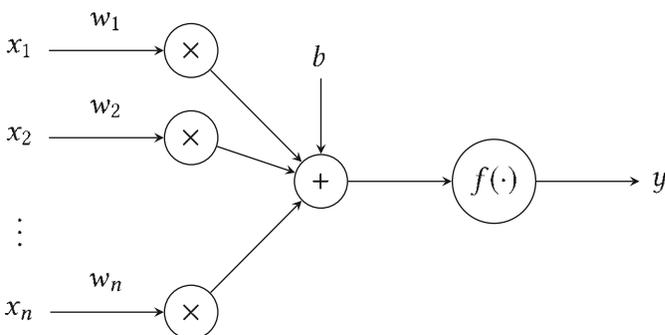
Это абстрактная версия биологического нейрона, представляющая собой структуру с несколькими входами и одним выходом. Выход биологического нейрона зависит от его входного сигнала; точно так же искусственный нейрон должен активироваться с учетом полученного ввода. Нас интересует мир компьютеров, а не биохимии мозга, поэтому нашему искусственному нейрону нужна вычислительная модель. Мы исходим из того, что в качестве сигналов нейрон получает и отправляет числа. Иными словами, он собирает весь ввод, вычисляет на его основе какое-то арифметическое значение и отправляет на выход какой-то результат. Для реализации искусственного нейрона не требуется никакой специальной электронной схемы. Можете считать, что это крошечная компьютерная программа, которая, как и любая другая, принимает входные значения и генерирует вывод. Нам не нужно создавать нейронные сети в буквальном смысле; мы можем их симулировать.

Частью процесса обучения биологических нейронных сетей является усиление или ослабление синапсов между нейронами. Развитие новых когнитивных способностей и поглощение знаний приводит к тому, что одни синапсы становятся сильнее, а другие ослабевают или даже полностью расходятся. Более того, синапсы могут не только стимулировать, но и блокировать активацию нейрона; когда сигнал доходит до такого синапса, нейрон не должен активироваться. На самом деле у маленьких детей больше мозговых синапсов, чем у взрослых. В процессе взросления нейронная сеть в нашей голове упрощается. Представьте, что мозг младенца — это глыба мрамора; с каждым

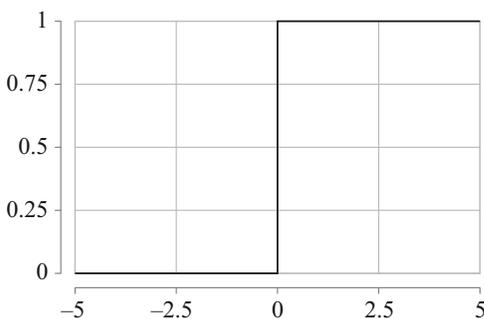
годом мы становимся опытнее и учимся новому, откалывая от нее кусочек за кусочком. В итоге она начинает приобретать определенную форму.

В искусственном нейроне пластичность синапсов и их способность блокировать сигнал определяются *весами*, которые мы применяем к вводу. В нашей модели есть n входных значений, x_1, x_2, \dots, x_n . К каждому из них мы применяем веса, w_1, w_2, \dots, w_n . Каждый вес умножается на соответствующий ввод. Это итоговое значение, полученное нейроном, является суммой произведений: $w_1x_1 + w_2x_2 + \dots + w_nx_n$. К этому *взвешенному* вводу мы добавляем *сдвиг* b , который можно считать склонностью нейрона к активации; чем больший сдвиг, тем вероятнее активация. Отрицательный сдвиг, добавленный к взвешенному вводу, будет блокировать нейрон.

Веса и сдвиг — это *параметры* нейрона, поскольку они влияют на его поведение. Вывод искусственного как и биологического нейрона зависит от его ввода. Это достигается за счет передачи входных значений специальной *функции активации* (или *передаточной функции*), которая и генерирует вывод. Ниже показана схема этого процесса, на которой функция активации обозначена как $f(\cdot)$.



Простейшей функцией активации является ступенчатая, которая выдает либо 0, либо 1. Нейрон генерирует 1, если поданное на вход значение больше нуля. В остальных случаях возвращается 0 (то есть нейрон не срабатывает).



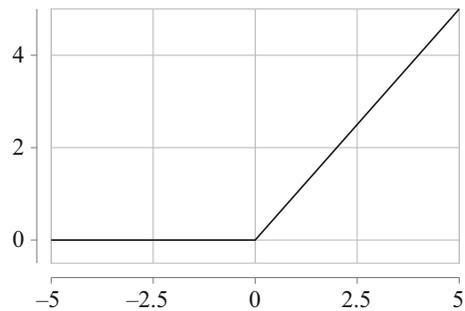
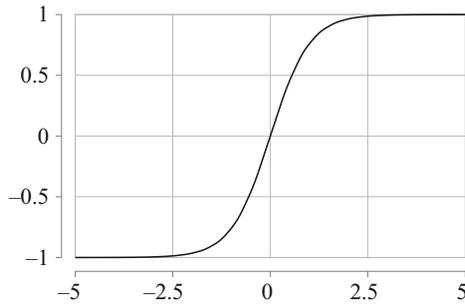
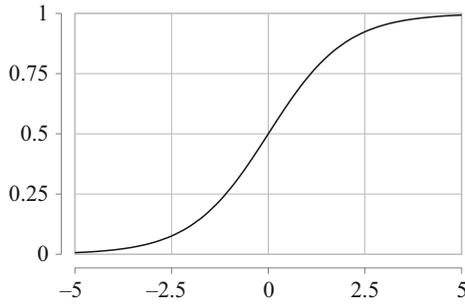
Сдвиг можно представить себе в качестве порогового значения. Нейрон возвращает 1, если взвешенный ввод превышает определенную величину, или 0, если нет. Действительно, если записать поведение нейрона в виде формулы, первым условием будет $w_1x_1 + w_2x_2 + \dots + w_nx_n + b > 0$ или $w_1x_1 + w_2x_2 + \dots + w_nx_n > -b$. Если $t = -b$, получится $w_1x_1 + w_2x_2 + \dots + w_nx_n > t$, где t — противоположность сдвигу; это пороговое значение, которое должен превысить ввод, чтобы нейрон активировался.

На практике вместо ступенчатой обычно используют другие, родственные функции активации. На следующей странице показаны три такие функции, которые часто можно встретить.

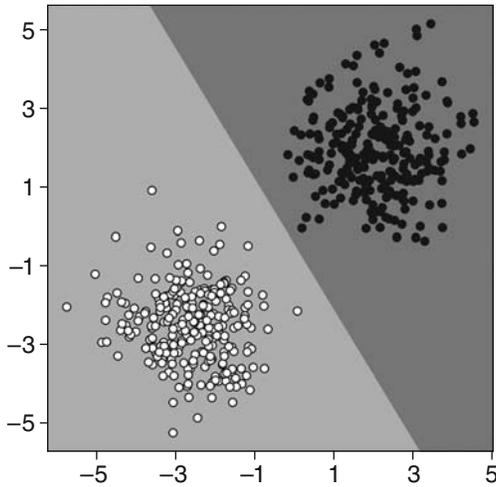
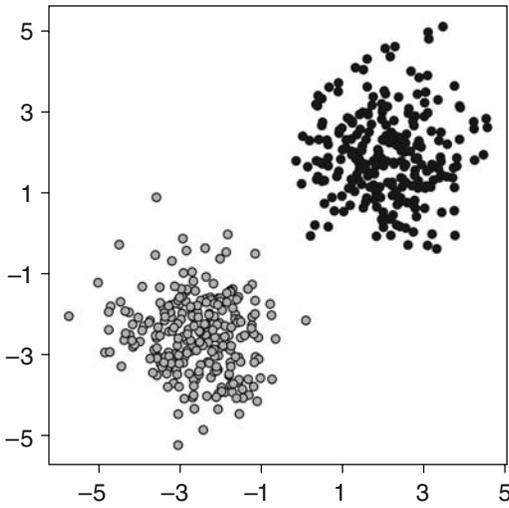
Та, что сверху, называется *сигмоидой*, так как она имеет форму буквы S. Ее вывод находится в диапазоне от 0 до 1. Крупный положительный ввод дает вывод, близкий к 1; крупный отрицательный ввод дает вывод, близкий к 0. Это примерно то, как работает биологический нейрон, который активируется только при сильном сигнале; это также плавное приближение ступенчатой функции. Функция активации, размещенная в центре, называется *гиперболическим тангенсом*, или, сокращенно, *tanh* (произносится по-разному: тап-Н, как в then, или thents с мягким th, как в thanks). Она похожа на сигмоиду, но находится в диапазоне от -1 до +1; крупный отрицательный ввод дает отрицательный вывод, что эквивалентно блокирующему сигналу. Нижняя функция называется *выпрямителем*. Она превращает любой отрицательный ввод в 0; в противном случае ее вывод прямо пропорционален вводу. В таблице, представленной ниже, показан вывод этих трех функций активации с учетом разного ввода.

С чем связано такое разнообразие функций активации (ведь существуют и другие)? Дело в том, что, как показывает практика, каждая из них имеет определенную сферу применения. Функции активации играют ключевую роль в поведении нейронов, что часто отражается на названиях последних. Нейрон, который использует ступенчатую функцию, называется *перцептроном*. Существуют также сигмоидные и tanh-нейроны. Нейрон является структурно-функциональной *единицей* (unit) нервной системы, поэтому нейроны на основе выпрямителей называются ReLU (rectified linear unit — выпрямленная линейная единица).

	-5	-1	0	1	5
сигмоида	0,01	0,27	0,5	0,73	0,99
tanh	-1	-0,76	0	0,76	+1
выпрямитель	0	0	0	1	5



Отдельный искусственный нейрон можно научить отличать два множества элементов. Например, взгляните на диаграмму, представленную вверху следующей страницы; на ней изображены два свойства: x_1 — на горизонтальной оси и x_2 — на вертикальной. Мы хотим создать систему, которая сможет отличить два скопления элементов. Она будет способна взять любой элемент и определить, к какой группе он принадлежит. Она фактически создаст *границу решений* (как на нижней диаграмме). Система сможет взять элемент с любыми координатами (x_1, x_2) и определить, к какой области он относится: светлой или темной.



У нейрона будет всего два входа. Он станет принимать каждую пару (x_1, x_2) и вычислять вывод. Если в качестве функции активации использовать сигмоиду, вывод окажется между 0 и 1. Значения, которые больше 0,5, будут принадлежать одной группе, а все остальные — другой. Таким образом нейрон получит роль *классификатора*, который распределяет наши данные по отдельным классам. Но как он будет это делать? Откуда у нейрона может взяться способность классифицировать данные?

Процесс обучения

В момент своего создания нейрон не способен распознавать никакие виды данных; он должен *научиться* это делать. Обучение происходит на примерах. Весь процесс напоминает то, как студент изучает какую-то тему путем анализа большого количества задач и их решений. Мы просим студента проанализировать условие и ответ. Если он как следует постарается, то после некоторого количества повторений этого процесса ему удастся понять, откуда берутся решения. Он даже сможет решать новые задачи, связанные с уже проанализированными, но уже без доступа к готовым решениям.

В ходе этого процесса мы *учим* компьютер находить ответы; набор решенных задач называется *учебным набором данных*. Это пример обучения с учителем, поскольку решения, словно учитель, помогают компьютеру находить правильные ответы. Это самый распространенный вид *машинного обучения*, отдельного раздела информатики, посвященного методам обучения компьютера решению задач. Существует также *обучение без учителя* — это когда мы предоставляем компьютеру учебный набор данных, но без каких-либо решений. У этого подхода есть важные сферы применения, такие как распределение наблюдений по разным кластерам (у задачи выбора подходящего кластера наблюдений нет универсального решения). Но в целом обучение с учителем дает лучшие результаты, так как мы предоставляем больше информации для анализа. Именно этот метод мы будем рассматривать в данной главе.

После обучения студент обычно сдает какие-то экзамены, чтобы показать, насколько хорошо он усвоил материал. То же самое с машинным обучением: мы даем компьютеру другой, *тестовый набор данных*, который он раньше не видел, и просим его найти соответствующие решения. Затем мы оцениваем производительность системы машинного обучения с учетом того, насколько хорошо ей удалось решить задачи в тестовом наборе данных.

В задаче с классификацией обучение с учителем проходит так: мы даем нейронной сети большое количество наблюдений (задач) вместе с их классами (решениями). Мы ожидаем, что нейрон каким-то образом научится приходить от наблюдения к его классу. Затем, если мы дадим ему наблюдение, с которым он раньше не сталкивался, он должен будет его успешно классифицировать (в разумных пределах).

Поведение нейрона в ответ на любой ввод определяется его весами и сдвигом. Вначале мы выбираем эти параметры случайным образом; на этом этапе нейрон подобен бестолковому студенту, который еще ничего не знает. Мы подаем на вход пару вида (x_1, x_2) . Нейрон сгенерирует вывод. Он будет таким же

В момент своего создания нейрон не способен распознавать никакие виды данных; он должен *научиться* это делать. Обучение происходит на примерах.

случайным, как наши текущие веса и сдвиг. Но при анализе каждого наблюдения в учебном наборе данных мы знаем, каким должен быть ответ. Благодаря этому мы способны определить, насколько сильно вывод нейрона отличается от желаемого ответа. Это называется *потерей*: мера того, насколько ошибся нейрон при анализе заданного ввода.

Например, если нейрон выдает в качестве ответа 0,2, а желаемый вывод равен 1,0, потеря является разницей этих двух значений. Чтобы не заморачиваться со знаками, потеря обычно вычисляется как квадрат разницы; в данном случае это будет $(1,0-0,2)^2 = 0,64$. Если бы желаемый вывод был равен 0,0, потеря была бы $(0,0-0,2)^2 = 0,04$. Зная потерю, мы можем ее минимизировать, откорректировав веса и сдвиг.

Вернемся к нашему студенту. После каждой неудачной попытки решить задачу, мы мотивируем его к достижению лучших результатов. Студент понимает, что ему нужно немного изменить свой подход и повторить попытку со следующим примером. Если он опять провалится, мы еще раз его мотивируем. И еще раз. Пока после целой кучи примеров в учебном наборе данных его решения не начнут становиться все ближе к верным и он не сможет справиться с тестовым набором.

Согласно нейробиологии, в ходе обучения структура мозга студента претерпевает изменения; некоторые синапсы между нейронами становятся сильнее, а другие ослабляются или даже рассоединяются. У искусственного нейрона нет прямого аналога этого процесса, но с ним происходит нечто похожее. Как вы помните, поведение нейрона зависит от его ввода, весов и сдвига. Мы не можем контролировать ввод, так как он поступает извне. Но мы можем регулировать веса и сдвиги. Именно это и происходит. Мы обновляем значения весов и сдвига таким образом, чтобы минимизировать ошибки нейрона.

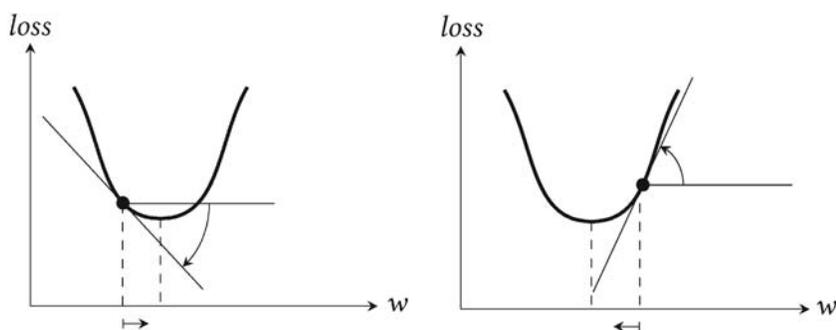
В этом нам помогает сама природа задачи, которую должен выполнить нейрон. Мы хотим, чтобы он взял каждое наблюдение, вычислил вывод в соответствии с определенным классом и откорректировал веса и сдвиг для минимизации потери. Таким образом нейрон пытается решить *задачу минимизации*. Она звучит так: *имея ввод и полученный вывод, откалибруйте веса и сдвиг, чтобы минимизировать потерю*.

Это требует принципиально нового подхода. До сих пор мы описывали нейрон как некую сущность, которая принимает ввод и генерирует вывод. В этом смысле нейрон представляет собой одну большую функцию, которая берет входные значения, применяет веса, слагает произведения, добавляет сдвиг, пропускает результат через функцию активации и выдает итоговый вывод. Но если взглянуть на это с другой стороны, ввод и вывод у нас уже есть (это

наш учебный набор данных), а вот веса и сдвиг могут меняться. Поэтому нейрон в целом можно рассматривать как функцию с *весами и сдвигом* в качестве переменных, поскольку именно на эти значения мы можем повлиять; более того, мы хотим менять их с каждым вводом, чтобы минимизировать потери.

Возьмем в качестве иллюстрации простой нейрон с одним весом и без сдвига. Отношение между весом и потерей показано в левой части диаграммы на следующей странице. Толстая кривая представляет потерю в виде функции от веса для заданного ввода. Нейрон должен отрегулировать свой вес, чтобы функция возвращала минимальное значение. На диаграмме показана точка с текущей потерей для заданного ввода. К сожалению, нейрон не знает, каким должен быть идеальный вес, который свел бы потерю к минимуму. Ему известно лишь значение функции в обозначенной точке; он не может свериться с представленной ниже диаграммой, как это делаем мы. К тому же вес может корректироваться (увеличиваться или уменьшаться) лишь совсем немного, приближая результат к минимуму.

Чтобы понять, что делать дальше (увеличивать или уменьшать вес), нейрон может найти касательную прямую, проходящую через текущую точку. Затем он может вычислить ее наклон, то есть угол между ней и горизонтальной осью. Стоит отметить, что нейрону для этого не требуется никаких особых возможностей; он лишь должен провести вычисления в одной локальной точке. Касательная прямая имеет отрицательный наклон, так как угол определяется по часовой стрелке. Эта величина показывает *скорость изменения функции*; следовательно, отрицательный наклон говорит о том, что с увеличением веса потеря будет уменьшаться. Таким образом нейрон узнает, что для уменьшения потери ему нужно сместиться вправо. Поскольку наклон отрицательный, а изменение веса положительное, нейрон делает вывод о том, что вес должен двигаться в направлении, противоположном тому, о котором сигнализирует наклон.



Теперь взгляните на правую часть диаграммы. На этот раз нейрон находится справа от минимальной потери. Он снова находит касательную прямую

и вычисляет ее наклон. Угол оказывается положительным. Положительный наклон говорит о том, что потеря увеличивается вместе с весом. Таким образом нейрон понимает, что для минимизации потери он должен уменьшить вес. Так как наклон положительный, а необходимое изменение веса отрицательное, нейрон опять делает вывод о том, что ему нужно двигаться в направлении, противоположном тому, о котором сигнализирует наклон.

В обоих случаях действует одно и то же правило: нейрон вычисляет наклон и изменяет вес в противоположном направлении. Нечто похожее можно встретить в математическом анализе. Наклон функции в заданной точке — это ее *производная*. Чтобы уменьшить потерю, нам нужно изменить вес на небольшую величину в направлении, противоположном производной потери.

Конечно, у нейрона обычно больше одного веса. К тому же у него есть сдвиг. Чтобы понять, как отрегулировать каждый отдельный вес и сдвиг, нейрон следует той же процедуре, которую мы описали выше, рассчитанной на один вес. Говоря математическим языком, он вычисляет так называемую *частную производную* потери относительно каждого отдельного веса и сдвига. Если у нас есть сдвиг и n весов, у нас в итоге получится $n + 1$ частных производных. Вектор, содержащий все частные производные функции, называется *градиентом*. Это эквивалент наклона для функций с множественными переменными; он показывает направление, в котором нужно двигаться, чтобы увеличить значение функции. Для его уменьшения следует выбрать противоположное направление. Таким образом, чтобы уменьшить потерю, нейрон смещает каждый вес и сдвиг в направлении, противоположном тому, о котором сигнализируют частные производные его градиента.

Для проведения вычислений нам вовсе не нужно рисовать касательные и измерять углы. Существуют эффективные методы нахождения частных производных и градиентов, но мы не станем вдаваться в подробности. Важно то, что у нас есть четко определенный способ регуляции весов и сдвига, который позволяет улучшать вывод нейрона. Таким образом процесс обучения можно описать в виде следующего алгоритма.

Для каждого ввода и желаемого вывода в учебном наборе данных:

- 1) вычисляется вывод нейрона и потеря;
- 2) обновляются веса и сдвиг нейрона, чтобы минимизировать потерю.

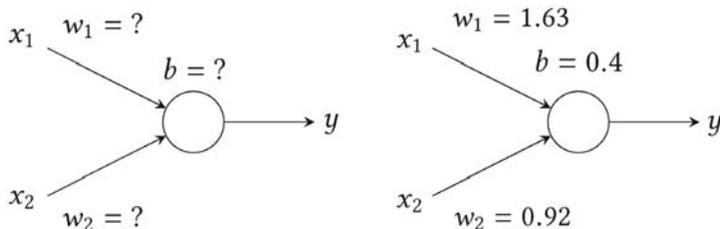
Пройдясь по всему учебному набору данных и закончив тем самым обучение, мы завершаем *эпоху*. Обычно на этом можно не останавливаться. Весь процесс повторяется на протяжении целого ряда эпох; это как если бы студент, изучивший весь учебный материал, начал с самого начала. Ожидается, что это даст лучшие

результаты, так как теперь он начинает не с чистого листа (то есть он не совсем бестолковый) — у него накопились кое-какие знания из предыдущей эпохи.

Чем больше повторений процесса обучения и накопленных эпох, тем лучше мы справляемся с учебными данными. Но главное здесь не перестараться. Студент, который анализирует один и тот же набор задач снова и снова, рано или поздно выучит их наизусть, но при этом не будет иметь никакого понятия о том, как решать другие задачи, с которыми он еще не сталкивался. Это бывает, когда студент, который вроде бы хорошо подготовился к экзамену, с треском проваливается. Говорят, что компьютер, проходящий через машинное обучение, *подгоняет* данные. Если с этим переборщить, происходит так называемая *переподгонка* (или переобучение): отличные результаты с учебным набором данных, но плохие — с тестовым.

Можно доказать, что, следуя этому алгоритму, нейрон может научиться классифицировать любые данные, которые являются *линейно разделяемыми*. Если ваши данные находятся в двух измерениях (как в нашем примере), это означает, что их можно разделить прямой линией. Если помимо (x_1, x_2) существуют другие свойства, применяется обобщенный принцип. В трех измерениях, когда ввод имеет вид (x_1, x_2, x_3) , данные являются линейно разделяемыми, если их можно разделить простой плоскостью в трехмерном пространстве. Если измерений больше, аналог прямой и плоскости называется *гиперплоскостью*.

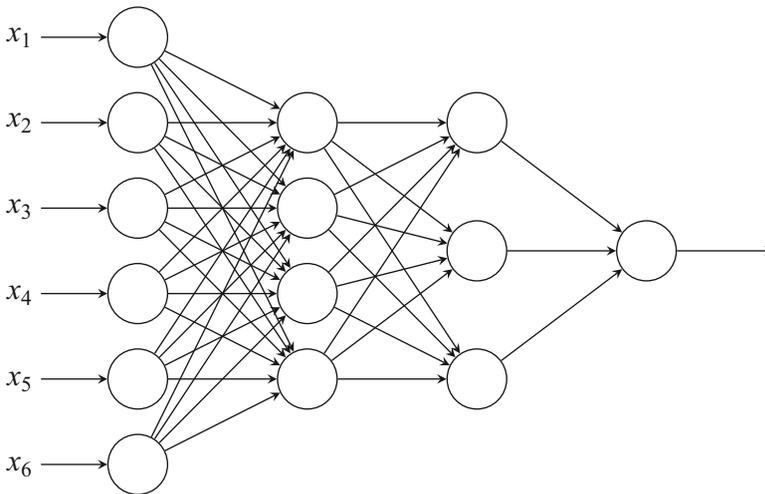
По окончании обучения наш нейрон научится разделять данные. «Научится» в том смысле, что он подберет подходящие веса и сдвиг, как это описано выше: он начнет со случайных значений и постепенно будет их обновлять, минимизируя потерю. Вспомните диаграмму с двумя скоплениями элементов, которые нейрон научился разделять с помощью границы решений. Слева показано то, с чего мы начинали, а справа — конечный результат. Вы можете видеть итоговые значения параметров.



Но так происходит не всегда. Отдельный нейрон, который действует сам по себе, может выполнять только определенные задачи, такие как классификация линейно разделяемых данных. Для решения более сложных проблем нужно использовать сети нейронов.

От нейронов к нейронным сетям

Из взаимосвязанных нейронов формируются *нейронные сети* (как биологические, так и искусственные). Входной сигнал одного нейрона может быть подключен к выходу другого, а его собственный выход может служить вводом для других нейронов. Таким образом можно создать нейронную сеть на подобие следующей:

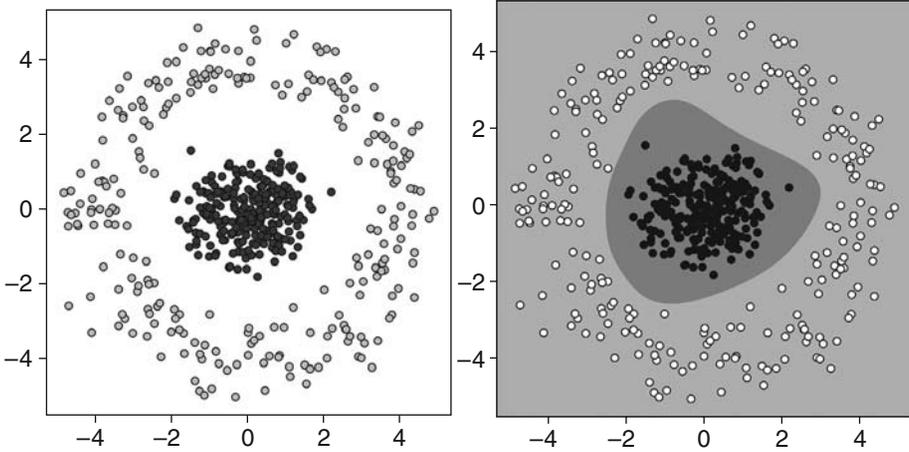


Нейроны этой искусственной нейронной сети разделены по слоям. Этот прием часто применяется на практике: многие нейронные сети состоят из слоев, размещенных один над другим. Еще мы сделали так, чтобы все нейроны текущего слоя соединялись со всеми нейронами следующего, в направлении слева направо. Это тоже распространенный подход, хотя использовать его необязательно. Такие слои называют *плотно связанными*.

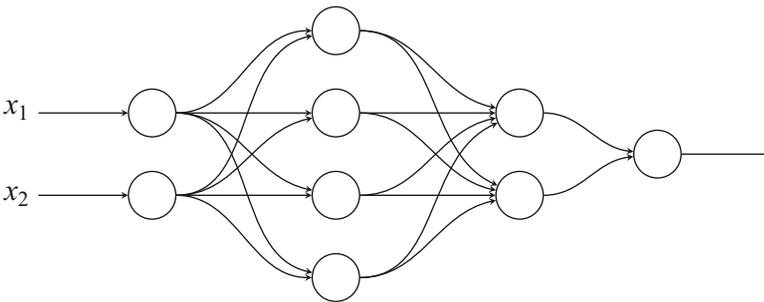
Первый слой не связан с предыдущим, точно так же как последний слой не связан со следующим. Вывод последнего слоя является выводом всей сети; это значения, которые мы хотим от нее получить.

Давайте вернемся к задаче классификации. Нам теперь нужно распределить два набора данных, представленных вверху следующей страницы. Данные собраны в концентрические круги. Любому человеку очевидно, что они принадлежат к двум отдельным группам. Также очевидно, что они не линейно разделяемые: эти два класса не сможет разделить никакая прямая линия. Нам нужно создать такую нейронную сеть, которая сможет отличать эти две группы и определять, к какой из них принадлежат любые последующие наблюдения. Это то, что показано на нижней диаграмме. Любое наблюдение

на светлом фоне будет причислено нейронной сетью к одной группе, а любое наблюдение на темном фоне — к другой.

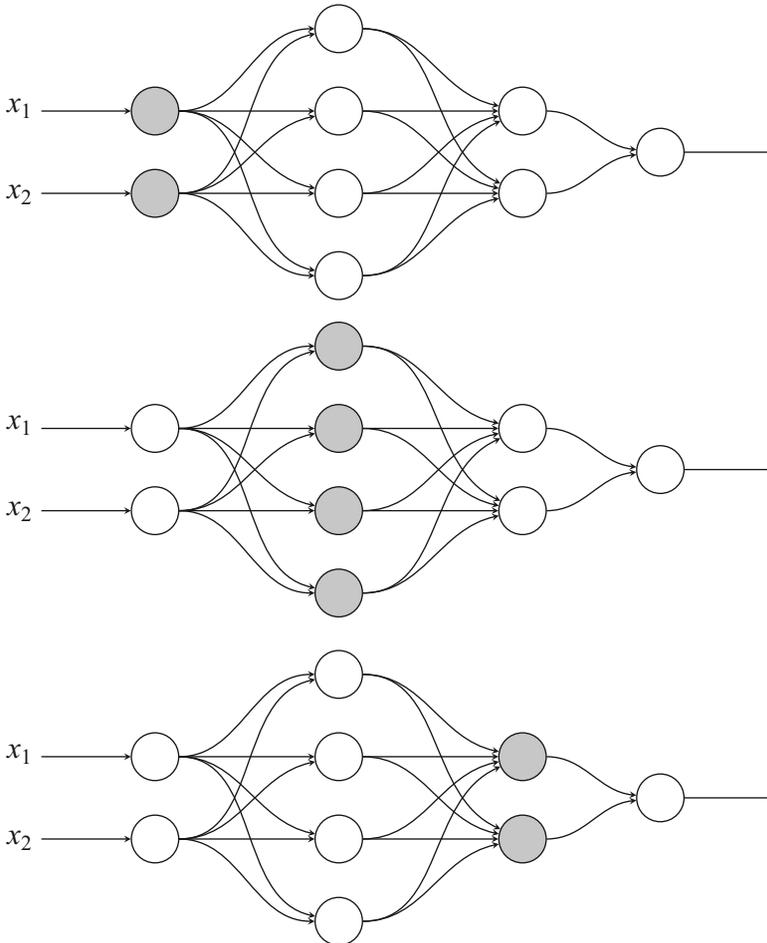


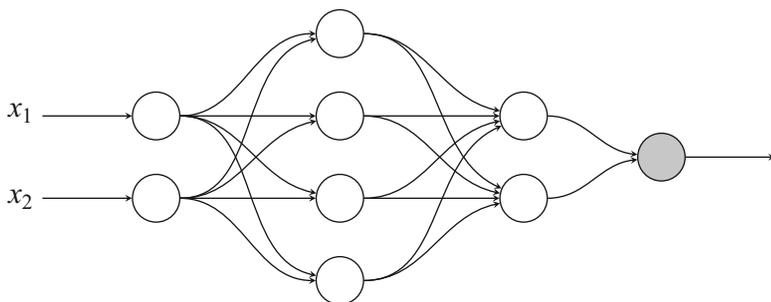
Чтобы получить результат, изображенный на нижней диаграмме, мы сформируем сеть, слой за слоем. Поместим два нейрона во входной слой, по одному для каждой координаты в наших данных. Добавим еще один слой с четырьмя нейронами, плотно соединенными с входным слоем. Поскольку у него нет прямого сообщения со входом или выходом, он является *скрытым*. Добавим еще один слой с двумя нейронами, тесно соединенными с первым скрытым слоем. В конце разместим выходной слой с одним нейроном, тесно связанным с последним скрытым слоем. Во всех нейронах применяется функция активации \tanh (гиперболический тангенс). Выходной нейрон генерирует значение между -1 и $+1$, демонстрирующее его уверенность в том, что данные принадлежат к той или иной группе. Мы возьмем это значение и дадим на его основе однозначный ответ, да или нет, в зависимости от того, превышает ли оно $0,0$. Вот как выглядит наша нейронная сеть:



Алгоритм обратного распространения потери

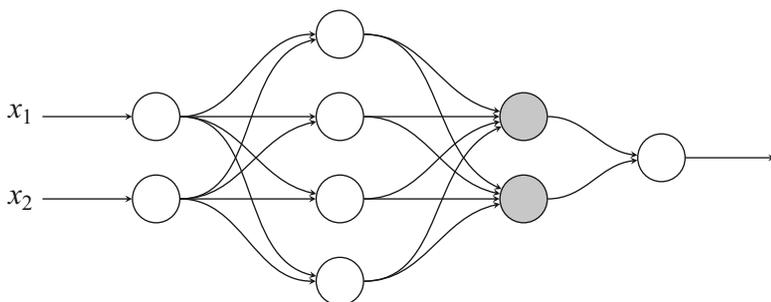
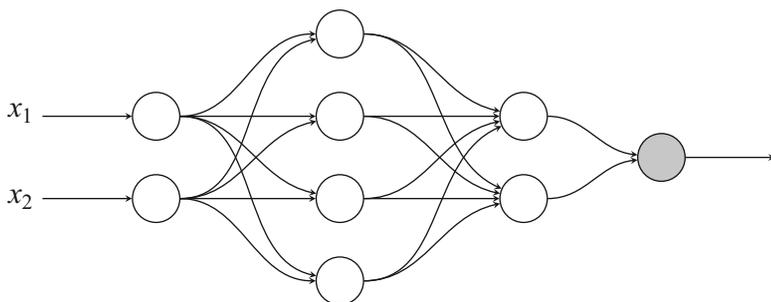
Вначале нейронная сеть ни о чем не знает, и еще ничего не отрегулировано; ее веса и сдвиги подбираются случайным образом. Так выглядит невежество в мире нейронных сетей. Затем мы подаем на вход наблюдение из нашего набора данных, то есть пару координат, x_1 и x_2 . Эти значения сначала попадают в два первых нейрона, а затем передаются в виде вывода первому скрытому слою. Все четыре нейрона в этом слое вычисляют свой вывод, который затем отправляется второму скрытому слою. Нейроны в этом слое передают свой вывод выходному слою, который генерирует итоговое выходное значение нейронной сети. По мере выполнения вычислений результаты проходят слой за слоем, от входного до выходного.

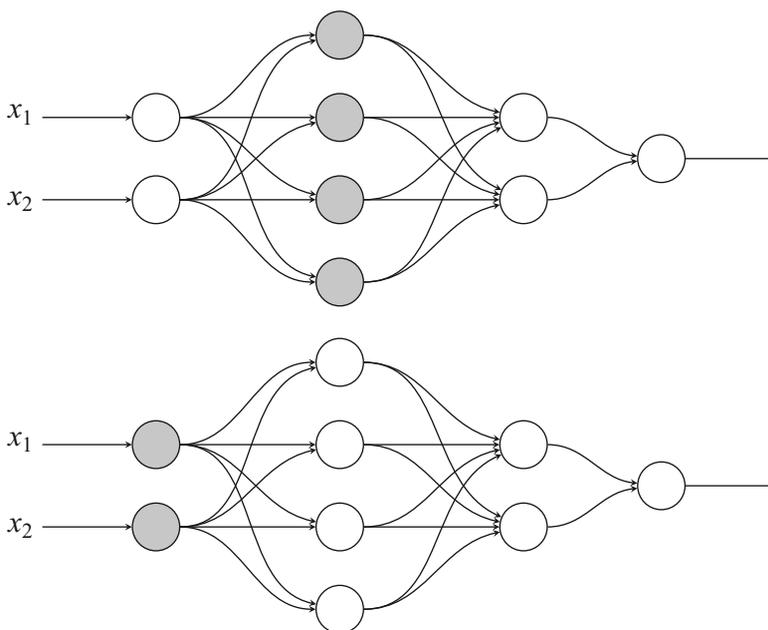




Дойдя до выходного слоя, мы вычисляем потерю, как в примере с одним нейроном. Затем нам нужно отрегулировать веса и сдвиг всех нейронов в сети, чтобы свести потерю к минимуму.

Оказывается, чтобы это сделать, можно пойти в обратном направлении, от выходного слоя к входному. Зная потерю, мы можем обновить веса и сдвиги нейронов выходного слоя (в нашем случае это всего один нейрон, но это не всегда так). Далее можно обновить веса и сдвиги нейронов предыдущего, последнего скрытого слоя. После этого повторяем данный процесс для предпоследнего скрытого слоя и, наконец, для входного.





Процесс обновления весов и сдвигов в нейронной сети похож на то, как мы обновляли отдельно взятый нейрон. Здесь опять используются математические производные. Нейронную сеть можно считать одной большой функцией, переменными которой выступают веса и сдвиги всех ее нейронов. Таким образом мы можем вычислить производную для каждого из этих значений с учетом потери и обновить нейрон с помощью этой производной. Мы подошли к ключевой части процесса обучения: *алгоритму обратного распространения потери*.

Для каждого ввода и желаемого вывода:

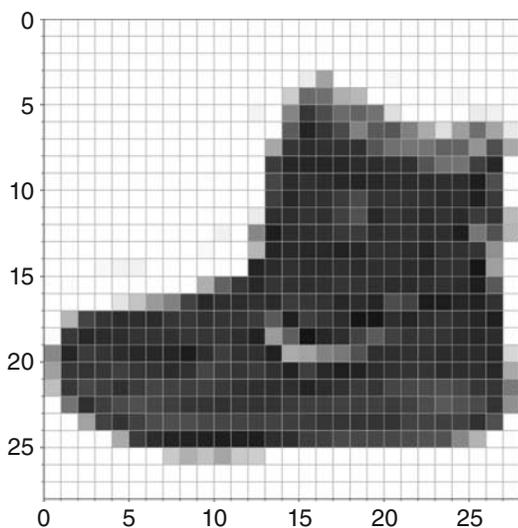
- 1) вычисляется вывод и потеря нейронной сети; мы проходим слой за слоем от входа к выходу;
- 2) обновляются веса и сдвиги нейронов, чтобы минимизировать потерю; теперь двигаемся от выходного слоя к входному.

Используя алгоритм обратного распространения, мы можем создавать сложные нейронные сети и учить их выполнять разные задачи. Системы глубокого обучения состоят из простых компонентов. Это искусственные нейроны с ограниченными вычислительными возможностями: они принимают ввод, умножают его на веса, слагают, добавляют сдвиг и применяют к полученному значению функцию активации. Но если их много и они соединены между собой особым образом в нейронные сети, их можно научить выполнять немало полезных задач.

Распознавание одежды

Давайте повернем разговор в более практичное русло и представим, что нам нужно создать нейронную сеть для распознавания элементов одежды, изображенных на рисунках. Это будет задача по *распознаванию образов*. Нейронные сети отлично проявили себя в этой области.

Каждый образ будет представлен небольшой фотографией размером 28×28 . Наш учебный набор данных состоит из 60 000 образов, а тестовый — из 10 000; первый будет использоваться для обучения нейронной сети, а второй — для оценивания того, насколько хорошо она обучилась. Вот пример образа, на который мы нанесли оси координат и сетку, чтобы вам было легче ориентироваться:



Образ разделен на отдельные мелкие участки — именно так происходит цифровая обработка изображений. Мы берем весь прямоугольный участок и делим его на небольшие фрагменты в количестве $28 \times 28 = 784$. Каждому фрагменту назначается целочисленное значение от 0 до 255, соответствующее определенному оттенку серого: 0 означает полностью белый цвет, а 255 — полностью черный. Приведенный выше образ можно представить в виде матрицы (см. следующую страницу).

В реальности нейронные сети обычно требуют сведения входных значений к узкому диапазону, такому как от 0 до 1, иначе они могут плохо работать; можете считать, что большие значения сбивают нейроны с толку. Из этого следует, что перед использованием матрицы каждую ячейку следовало бы поделить на 255, но мы намеренно пропустим данный этап.

На следующем рисунке показан случайный набор из десяти предметов каждого вида. Как видите, изображения довольно разнообразные, и не все они являются идеальными примерами того или иного класса. Это делает нашу задачу интереснее. Мы хотим создать нейронную сеть, которая принимает на вход подобные образы и возвращает вывод с информацией о том, к какой категории они, по ее мнению, принадлежат.

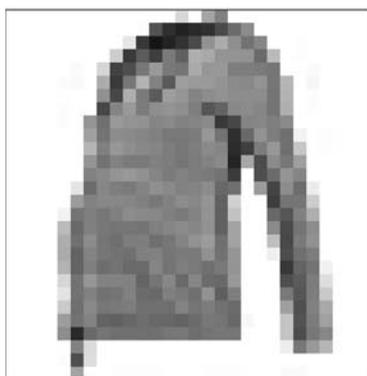
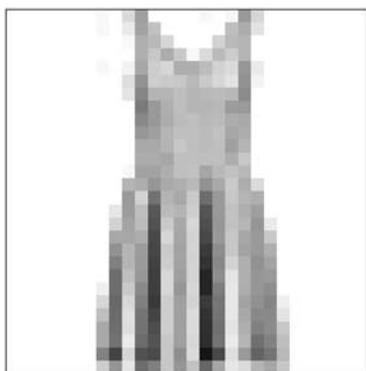
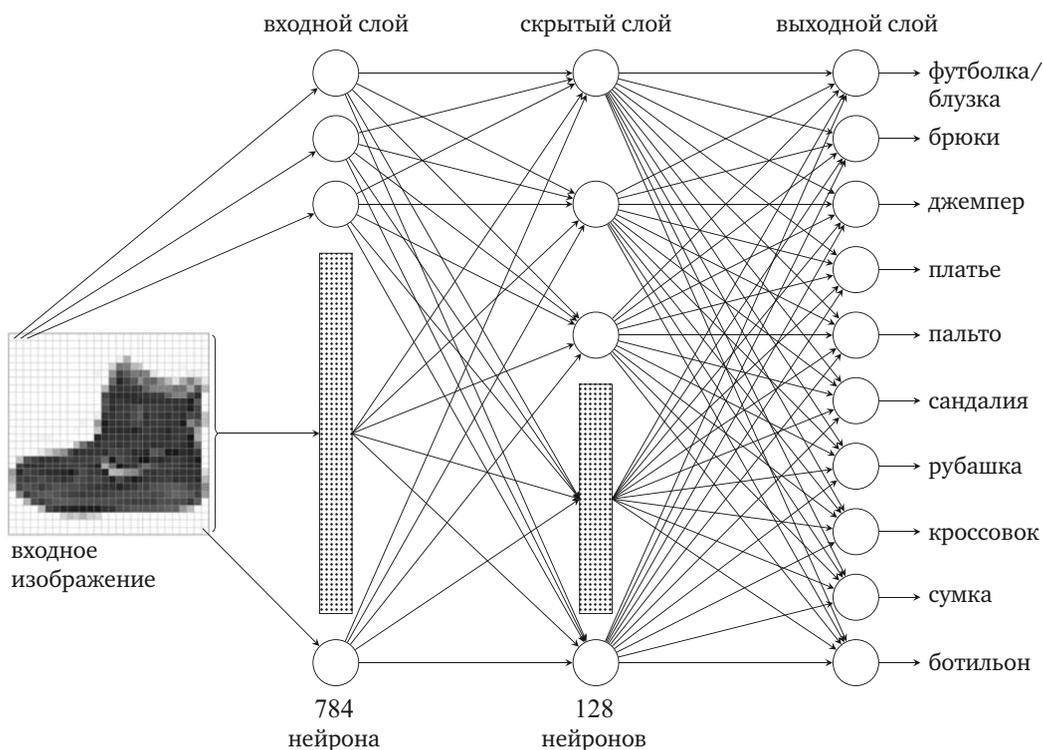


И снова наша нейронная сеть будет состоять из слоев. В первом слое окажутся 784 входных нейрона. Каждый из них станет принимать один ввод, соответствующий одному фрагменту изображения, и просто возвращать его в виде своего вывода. Если на рисунке изображен ботильон, первый нейрон получит на вход значение левого верхнего фрагмента, 0, и вернет этот 0. Остальные нейроны получают значения фрагментов строка за строкой, сверху вниз, слева направо. Фрагмент со значением 58 в правом конце каблука ботильона (четвертая строка снизу, третий столбец справа) будет передан нейрону и скопирован в качестве его вывода. Строки и столбцы в нашей нейронной сети отсчитываются сверху и слева, поэтому данный нейрон находится в 25-й строке сверху и 26-м столбце слева, что делает его порядковый номер равным $24 \times 28 + 26 = 698$.

Следующий слой будет плотно соединен с входным слоем. Он состоит из 128 нейронов типа ReLU. Этот слой не имеет прямой связи ни со входными изображениями (в отличие от входного слоя), ни с выводом (так как мы добавим еще один слой). Следовательно, он скрытый, так как его не видно из-за пределов нейронной сети. Поскольку соединение плотное, это приведет к большому числу связей между входным и скрытым слоями. Каждый нейрон в скрытом слое будет соединен с выводами всех нейронов входного слоя. Это означает, что у каждого нейрона окажется 784 входных соединения, а во всем слое — $784 \times 128 = 100\,352$.

Добавим последний слой, который содержит выходные нейроны, возвращающие результаты работы нейронной сети. Всего нейронов будет 10, по одному на каждый класс. Каждый из них будет соединен со всеми нейронами скрытого слоя, всего $10 \times 128 = 1280$ соединений. Общее количество связей между всеми слоями в нейронной сети составит $100\,352 + 1280 = 101\,632$. Принцип работы полученной системы проиллюстрирован на следующей странице. Мы не можем уместить все узлы и ребра, поэтому большинство узлов во входном и скрытом слоях обозначены прямоугольниками с точками внутри; в первом прямоугольнике 780 узлов, а во втором — 124. Все соединения, направленные к отдельным узлам в прямоугольниках, сведены в одну точку.

Вывод нашей нейронной сети будет состоять из 10 значений, по одному на каждый нейрон в слое. Каждый выходной нейрон представляет отдельный класс, а его вывод означает вероятность принадлежности изображения к этому классу; в сумме значения всех 10 нейронов дадут 1, как и полагается при работе с вероятностными величинами. Это пример очередной функции активации, которая называется *softmax*; она принимает на вход вектор вещественных чисел и преобразует их в распределение вероятностей. Давайте рассмотрим два следующих примера.



Если взять первый пример, показанный сверху, после обучения сеть выведет следующий вывод:

Выходной нейрон	Класс	Вероятность
1	T-shirt/top	0,09
2	Trouser	0,03
3	Pullover	0,00

Выходной нейрон	Класс	Вероятность
4	Dress	0,83
5	Coat	0,00
6	Sandal	0,00
7	Shirt	0,04
8	Sneaker	0,00
9	Bag	0,01
10	Ankle boot	0,00

Таким образом нейронная сеть выражает уверенность в том, что это платье, оценивая вероятность этого в 83%. Незначительными вероятностями того, что это может быть футболка/блузка, рубашка или брюки можно пренебречь.

Во втором примере, показанном справа, сеть возвращает такие результаты:

Выходной нейрон	Класс	Вероятность
1	T-shirt/top	0,00
2	Trouser	0,00
3	Pullover	0,33
4	Dress	0,00
5	Coat	0,24
6	Sandal	0,00
7	Shirt	0,43
8	Sneaker	0,00
9	Bag	0,00
10	Ankle boot	0,00

Нейронная сеть выражает 43-процентную уверенность в том, что она имеет дело с рубашкой — это ошибка; в действительности на фотографии изображен джемпер (если вы вдруг сами не догадались). Тем не менее на втором месте с 33% находится правильный вариант.

В одном примере сеть дала верный ответ, а в другом ошиблась. В целом, если предоставить ей все 60 000 изображений из нашего учебного набора данных, она сумеет правильно распознать около 86% в тестовом наборе, состоящем из 10 000 изображений. Довольно неплохо, учитывая, что эта нейронная

сеть хоть и намного сложнее предыдущей, все равно является относительно простой. На ее основе можно создавать более сложные сетевые структуры, способные давать лучшие результаты.

Несмотря на повышенную сложность, наша нейронная сеть учится точно так же, как и более простая сеть из предыдущего примера, которая распознавала скопления данных и концентрические круги. Для каждого ввода в ходе обучения мы получаем вывод, который затем сравнивается с желаемым результатом с целью вычисления потери. Теперь же в виде вывода возвращается не одно значение, а сразу десять, хотя принцип остается прежним. Когда нейронная сеть распознает рубашку с примерно 83-процентной вероятностью, мы можем сравнить это с идеальным результатом, то есть с вероятностью 100%. Таким образом у нас получаются два набора выходных значений: те, что сгенерировала сеть, и те, которые мы бы хотели получить (набор, в котором правильный ответ имеет вероятность 1, а все остальные — 0). Ниже приводится сравнение полученных и желаемых результатов в последнем примере.

Выходной нейрон	Класс	Вероятность	Цель
1	T-shirt/top	0,00	0,00
2	Trouser	0,00	0,00
3	Pullover	0,33	1,00
4	Dress	0,00	0,00
5	Coat	0,24	0,00
6	Sandal	0,00	0,00
7	Shirt	0,43	0,00
8	Sneaker	0,00	0,00
9	Bag	0,00	0,00
10	Ankle boot	0,00	0,00

Мы берем последние два столбца и вычисляем показатели потери — только на этот раз у нас много значений, поэтому здесь нельзя обойтись простой разницей в квадрате. Для получения разницы между наборами подобных значений существуют специальные величины. В нашей нейронной сети мы будем использовать одну из них, *категорийную перекрестную энтропию*, которая определяет, насколько сильно отличаются два распределения вероятностей. Вычислив потерю, мы обновляем нейроны в выходном слое. Дальше обновляются нейроны в скрытом слое. Если коротко, то мы выполняем обратное распространение.

Мы выполняем тот же процесс для всех изображений в нашем учебном наборе данных, то есть для всей эпохи. Закончив, повторяем все сначала для следующей эпохи. При этом мы пытаемся найти баланс: подобрать достаточное количество эпох для того, чтобы нейронная сеть извлекла как можно больше пользы из учебного набора данных, но не слишком много. В ходе обучения сеть скорректирует веса и сдвиги нейронов, которых довольно много. Входной слой всего лишь копирует значения в скрытый, поэтому его регулировать не нужно. Однако в скрытом слое находится 100 352 веса и 128 сдвигов, а в выходном — 1280 весов и 10 сдвигов, что в сумме дает 101 770 параметров.

Знакомство с глубоким обучением

Сами по себе нейроны мало на что способны, но нейронной сети по силам любая вычислительная задача, которую можно описать алгоритмически и выполнить на компьютере — это доказуемый факт. Следовательно, нейронная сеть может делать все то же самое, что и компьютер. Но, конечно, вся суть в том, что нейронной сети не нужно объяснять, как именно выполнять задачу. Достаточно предоставить ей примеры и обучить ее с помощью алгоритма. Одним из таких алгоритмов, как мы уже видели, является обратное распространение. В этой главе рассматривалась лишь классификация, но нейронные сети можно применять к задачам любого рода. Они могут предсказывать значения определенных величин (таких как кредитный рейтинг), переводить текст с одного языка на другой, а также распознавать и генерировать речь; они сумели победить чемпионов в игре го, продемонстрировав совершенно новые стратегии, которые поставили в тупик даже экспертов. Чтобы научиться играть в го, им достаточно было знать правила игры. Они не имели доступа к библиотеке уже проведенных поединков, и в процессе обучения они просто играли сами с собой.

Сегодня существует множество примеров успешного применения нейронных сетей, но принципы, на которых они основаны, не новы. Перцептрон изобрели в 1950-х, а алгоритму обратного распространения уже больше 30 лет. За это время интерес к потенциалу нейронных сетей то появлялся, то пропадал. Но в последние несколько лет кое-что изменилось: мы получили возможность создавать по-настоящему огромные сети. Это достигнуто благодаря прогрессу в сфере производства специализированных компьютерных чипов, способных эффективно выполнять вычисления, которые применяются в нейронах. Если представить себе все нейроны нейронной сети,

размещенные в памяти компьютера, то все необходимые вычисления можно проводить с помощью операций с огромными числовыми матрицами. Нейрон вычисляет сумму взвешенных произведений своих вводов; как вы, наверное, помните из обсуждения PageRank в предыдущей главе, сумма произведений — это суть умножения матриц.

Оказалось, что для этого идеально подходят *графические процессоры* (англ. graphics processing units или GPU). GPU — это компьютерный чип, специально предназначенный для создания и модификации изображений внутри компьютера; это родственник *центрального процессора* (англ. central processing unit или CPU), чипа, который выполняет программные инструкции. GPU рассчитан на работу с компьютерной графикой. Генерация и обработка компьютерной графики требует выполнения числовых операций с большими матрицами; сцена, сгенерированная компьютером, представляет собой огромную матрицу с числами (вспомните ботильон). Графические процессоры играют роль рабочих лошадок в игровых консолях, занимая наше воображение часами напролет. Но эта технология также ответственна за прогресс в сфере искусственного интеллекта.

Мы начинали с простейшей нейронной сети, состоящей из единственного нейрона. Затем количество нейронов было увеличено и впоследствии доведено до нескольких сотен. Тем не менее нейронная сеть для распознавания образов, которую мы создали, вовсе не является большой. И ее архитектуру нельзя назвать сложной. Мы просто добавляли один слой нейронов за другим. Исследователи в области глубокого обучения добились больших успехов в проектировании нейронных сетей. Предложенные ими архитектуры могут состоять из десятков слоев, и их геометрия вовсе не ограничена простым одномерным набором нейронов, как в наших примерах. Нейроны внутри слоя могут формировать двухмерные полотнообразные структуры. Более того, слои вовсе не обязательно связывать плотными соединениями; существуют и другие модели связей. Также необязательно подключать вывод предыдущего слоя к вводу следующего. Мы, например, можем соединить два несмежных слоя. Слои можно объединять в модули и составлять из этих модулей все более сложные конфигурации. Сегодня в нашем распоряжении есть целый зоопарк архитектур нейронных сетей, каждая из которых рассчитана на определенные задачи.

Но, какой бы ни была архитектура, слои нейронов в ходе обучения обновляют свои веса и сдвиги. Если задуматься, у нас есть набор входов, которые, обучаясь, преобразуют слои. В результате эти слои, внося изменения в свои параметры, вбирают в себя информацию, представленную входными данными. Конфигурация весов и сдвигов слоя является результатом полученных

им данных. Первый скрытый слой, который контактирует непосредственно с входным слоем, кодирует ввод нейронной сети. Второй скрытый слой кодирует вывод первого скрытого слоя, с которым он соединен напрямую. По мере того, как мы все глубже погружаемся в многослойную сеть, каждый последующий слой кодирует вывод, полученный из предыдущего. Каждое представление основано на предыдущем и поэтому каждый следующий слой находится на все более высоком уровне абстракции. Таким образом глубокие нейронные сети изучают иерархию концепций, переходя на все более абстрактные уровни. Вот почему мы называем это *глубоким* обучением. Мы имеем в виду архитектуру, в которой каждый следующий слой представляет более глубокие концепции, находящиеся на более высоких уровнях абстракции. Если взять распознавание образов, то первый слой в многослойной сети может научиться распознавать небольшие локальные участки, такие как края изображения. Затем второй слой может научиться распознавать концепции, основанные на участках, распознанных предыдущим слоем, такие как глаза, нос и уши. Третий слой может отталкиваться от результатов второго и научиться распознавать лица. Теперь вы можете видеть, что наша нейронная сеть для распознавания образов была немного наивной; мы даже не пытались реализовать настоящее глубокое обучение. Повышая уровень абстракции, мы ожидаем, что наша сеть будет находить те же образы, что и человек — от грамматических структур до признаков болезни на рентгеновских снимках; от символов, написанных от руки, до интернет-мошенничества.

Тем не менее вы можете сказать, что все это сводится к обновлению простых значений в простых составных компонентах — искусственных нейронах. И вы будете правы. Люди, которые это осознают, иногда чувствуют разочарование. Они хотят узнать, что такое машинное и глубокое обучение, но ответ кажется слишком уж простым: иногда то, что с виду обладает человеческими способностями, можно свести к элементарным по своей сути операциям. Возможно, мы предпочли бы получить более запутанный ответ, который пощекотал бы наше самолюбие.

Но не стоит забывать, что, согласно научному подходу, природу можно объяснить с использованием элементарных концепций — настолько простых, насколько возможно. Это вовсе не означает, что из простых правил и составных компонентов нельзя получить сложные структуры и модели поведения. Искусственные нейроны намного проще биологических, но даже если бы поведение последних можно было свести к простым моделям, только благодаря их огромному количеству и взаимосвязанности возникает то, что мы называем интеллектом.

Искусственные нейроны
намного проще
биологических, но даже
если бы поведение последних
можно было объяснить,
только благодаря их
огромному количеству
и взаимосвязанности
возникает... интеллект.

Это позволяет взглянуть на вещи под другим углом. Действительно, искусственные нейронные сети могут обладать необычайным потенциалом. Но, чтобы заставить их работать, требуются неординарная находчивость и потрясающие инженерные умения со стороны людей. Здесь мы лишь слегка коснулись этой темы. Возьмем, к примеру, обратное распространение. Это фундаментальный алгоритм, лежащий в основе нейронных сетей, который позволяет эффективно выполнять вычисления, являющиеся по сути поиском математических производных. Исследователи потратили много времени на создание эффективных вычислительных методик, таких как *автоматическое дифференцирование* — широко распространенный механизм получения производных. Или подумайте о том, как именно вычисляются изменения параметров в нейронной сети. Существуют различные *оптимизаторы*, позволяющие развертывать все более крупные, но в то же время эффективные сети. Если говорить об аппаратном обеспечении, то инженеры проектируют все лучшие и лучшие чипы, которые ускоряют выполнение все более сложных нейронных вычислений, но при этом снижают использование ресурсов. На смену существующим архитектурам нейронных сетей приходят новые. В этой области проводятся активные исследования и эксперименты; это в том числе касается попыток создания нейронных сетей, которые проектируют другие нейронные сети. Поэтому каждый раз, когда вам встречается новостной заголовок об очередном достижении в этой сфере, вспоминайте о тех трудолюбивых людях, благодаря которым это стало возможным.

ПОСЛЕСЛОВИЕ

15 июля 2019 года Марк Карни, глава Банка Англии, представил дизайн новой купюры номиналом £50, которую планировалось ввести в оборот двумя годами позже. В 2018 году Банк Англии решил посвятить новую купюру ученому и организовал публичный конкурс, длившийся шесть недель. Всего было получено 227 299 голосов за 989 кандидатов, которые соответствовали заявленным критериям. Затем Консультативный комитет по дизайну банкнот сократил этот список до 12 вариантов. Окончательное решение принял глава банка, выбрав Алана Тьюринга. Вот как он это прокомментировал: «Алан Тьюринг был выдающимся математиком, чей труд оказывает огромное влияние на то, как мы живем сегодня. Работы Алана Тьюринга, отца компьютерных наук и искусственного интеллекта, а также героя войны, были прорывными и имели далеко идущие последствия. Тьюринг — гигант, на плечах которого так многие стоят».

Тьюринг (1912–1954) был гением, который исследовал природу компьютерных вычислений и их пределы. Он предвидел восход устройств, демонстрирующих разумное поведение, задавался вопросами о том, могут ли компьютеры мыслить, сделал вклад в такие области, как математическая биология и механизмы морфогенеза, сыграл ключевую роль в расшифровке закодированных немецких сообщений во время Второй мировой войны (его участие десятилетиями оставалось засекреченным). В результате трагического поворота событий Тьюринг покончил с собой. В 1952 году его арестовали, осудили и приговорили к обязательной гормональной терапии за гомосексуализм, который в Великобритании тех времен был противозаконным. Официальное помилование состоялось в 2013 году. Его изображение на новой купюре — это своего рода реабилитация, которая еще несколько десятилетий назад была бы немислимой.

Все алгоритмы в этой книге описывались в виде простых шагов, достаточно элементарных для того, чтобы их можно было выполнить с помощью ручки и листа бумаги. Учитывая, что алгоритмы реализуются в компьютерных программах, понимание того, что они на самом деле собой представляют, поможет понять, какие вычисления в принципе возможны. Для этого необходимо глубже погрузиться в природу этих простых шагов. В конце концов, ученик начальной школы и выпускник колледжа могут по-разному обращаться с ручкой и листом бумаги. Возможно ли в точности определить, из какого рода

Возможно ли в точности
определить, из какого рода
шагов может состоять
алгоритм?.. В 1936 году
[Тьюринг] разработал модель
вычислительной машины,
которая определяет, на что
способен (любой) компьютер.

шагов может состоять алгоритм? Тьюринг предложил ответ на этот вопрос еще до появления первых цифровых компьютеров. В 1936 году он разработал модель вычислительной машины, которая определяет, на что способен (любой) компьютер. Ее сокращенно называют *машиной Тьюринга*. Она состоит из следующих компонентов.

1. *Лента*, разделенная на клетки или *ячейки*. Каждая ячейка может быть пустой или содержать символ из какого-либо алфавита. Она может быть бесконечно длинной.
2. *Головка* может последовательно двигаться по ленте влево и вправо. Головка может считывать символы в ячейке, которая под ней находится. Мы называем символ в этой ячейке *считанным*. Головка может стереть или перезаписать считанный символ.
3. *Управляющее устройство*, также известное как регистр состояния. Может находиться в любом состоянии, входящем в конечное множество. Представьте себе циферблат с состояниями, на любое из которых может указывать стрелка.
4. *Конечная таблица с инструкциями*. Каждая инструкция определяет следующее *действие* машины. Это то, что сделает машина, учитывая текущее состояние и считанный символ.

Машина Тьюринга показана в схематическом виде на следующей странице.

Алфавит этой отдельно взятой машины Тьюринга состоит из 1 и ★. Согласно управляющему устройству, машина может находиться в одном из семи состояний, q_0, q_1, \dots, q_6 . В таблице с инструкциями каждому состоянию выделена отдельная строка, а каждому возможному символу — отдельный столбец; чтобы вы могли видеть пустые ячейки, мы обозначим их как *B*. Текущее состояние определяется строкой, а считанный символ — столбцом. Каждая запись в таблице инструкций содержит либо тройное значение, описывающее действие, либо прочерк, который означает, что машине нечего делать в этом сочетании строки и столбца.

Действие машины состоит из следующих шагов.

1. Машина может изменить свое состояние или оставить его как есть. Новое состояние является первым элементом тройного значения в конечной таблице инструкций.

Наша демонстрационная машина Тьюринга выполняет алгоритм, который вычисляет разницу двух чисел, a и b , если $a > b$; в противном случае возвращается ноль. Эта операция называется *собственное вычитание*, обозначим ее как $a \dot{-} b$. Мы имеем $4 \dot{-} 2 = 2$ и $2 \dot{-} 4 = 0$.

Вначале мы помещаем на ленту *ввод* машины. Ввод — это конечная строка символов из машинного алфавита. Все остальные ячейки на ленте, слева и справа от нее, являются пустыми. В этой машине Тьюринга ввод имеет следующий вид: $1111 \star 11$; он представляет числа 4 и 2 в *унарной системе счисления*, разделенные символом \star .

Изначально головка машины находится в крайней слева ячейке. Управляющее устройство указывает на состояние q_0 . Затем машина начинает работать и выполнять свои действия. Если проследить первые шесть действий, можно увидеть следующую картину.

1. Мы начинаем в состоянии q_0 , а считанный символ равен 1.

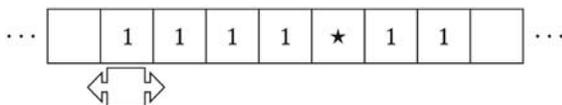
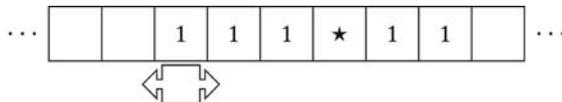
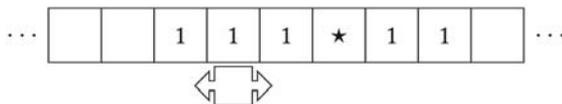


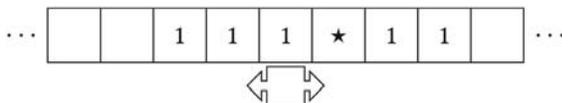
Таблица инструкций дает нам (q_1, B, R) , поэтому машина поменяет свое состояние на q_1 , сотрет 1 и переместится вправо. Головка и лента будут выглядеть так:



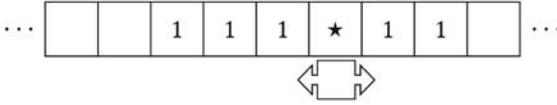
2. Для состояния q_1 и считанного символа 1 таблица инструкций дает нам $(q_1, 1, R)$. Машина прочитает и запишет 1, оставив ячейку без изменений, и переместится вправо, оставаясь в состоянии q_1 .



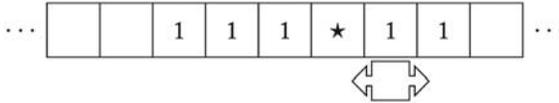
3. Машина делает то же самое на шаге 2: считывает и записывает 1, оставаясь в q_1 , и перемещается вправо.



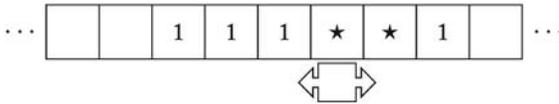
4. И снова машина прочитает и запишет 1, оставаясь в q_1 , и переместится вправо.



5. Головка перешла к символу ★, оставаясь в состоянии q_1 . Инструкция выглядит как $(q_1, ★, R)$. Машина меняет состояние на q_2 , оставит ★ на ленте и переместится вправо.



6. Головка перешла к символу 1, справа от ★, и теперь находится в состоянии q_2 . Инструкция выглядит как $(q_2, ★, L)$. Машина меняет состояние на q_3 , запишет ★ вместо 1 и переместится влево.



Машина продолжит работать в том же духе, выполняя действия, предписанные таблицей инструкций. Если взглянуть на весь процесс в целом, то становится очевидно, что машина выполняет цикл. На каждой итерации она находит самую левую единицу и заменяет ее пустым значением. Затем она ищет справа символ ★. Когда машина его находит, она продолжает перемещаться вправо, пока не обнаружит единицу и не запишет вместо нее ★. Таким образом на каждой итерации машина перезаписывает 1 слева и справа от ★. Через какое-то время эта операция станет невыполнимой. В этом случае машина сотрет символы ★ и завершит работу. Лента будет содержать значение 11, эквивалент числа 2, окруженное пустыми ячейками. Чтобы просигнализировать о завершении работы, машина войдет в состояние q_6 , в котором, согласно таблице инструкций, больше нечего делать, и остановится.

Если предоставить ввод 11 ★ 1111, машина будет работать до тех пор, пока лента не заполнится пустыми ячейками, эквивалентными значению 0. Если дать машине любой ввод, состоящий из a единиц, звездочки и b единиц, она будет выполнять свои действия, пока на ленте не останется $a - b$ единиц (если $a > b$), или пока вся лента не будет состоять из одних пустых ячеек.

Эта машина Тьюринга выполняет алгоритм для вычисления операции собственного вычитания на основе предоставленного ввода, следуя инструкциям, описанным в соответствующей таблице. Ее шаги настолько элементарны, что для выполнения операции головке приходится активно перемещаться туда-сюда. Чтобы найти $2 \div 4 = 0$ и $4 \div 2 = 2$ потребуется 21 и, соответственно, 34 действия. Действий много, но насколько же они просты! Для их выполнения требуется минимальный интеллект. Именно в простоте итераций весь смысл. Для выполнения действий машины Тьюринга не требуется особых умений; вам лишь нужно сверяться с таблицей, перемещаться по ленте, считывать и записывать по одному символу за раз и следить за своим состоянием. Вот и все. Нетривиальным этот подход делает то, что действия, которые способна выполнять машина Тьюринга, определяют, из каких шагов может состоять алгоритм.

В этой книге алгоритмы описывались на более высоком уровне с помощью более сложных шагов. Это делалось для удобства, так как машина Тьюринга работает на таком низком уровне, что описывать с ее помощью наши алгоритмы было бы слишком громоздким процессом. Но все шаги в любом рассмотренном нами алгоритме можно представить в виде действий корректно составленной машины Тьюринга. Мы продемонстрировали простую машину для реализации собственного вычитания. Для более сложных алгоритмов нам понадобилась бы машина Тьюринга с большим количеством действий, с более объемным алфавитом и более богатой таблицей инструкций. Но это была бы вполне выполнимая задача.

Несмотря на свою простоту, машина Тьюринга имеет широчайшую сферу применения; с ее помощью можно реализовать любой алгоритм. Машина Тьюринга способна выполнить любой алгоритм, который можно вычислить на компьютере. Иными словами, *с ее помощью можно делать все, на что способен алгоритм*. Это свободная трактовка тезиса Черча — Тьюринга, названного в честь Тьюринга и американского математика Алонсо Черча (1903–1995), одного из основателей теоретических компьютерных наук. Это тезис, поскольку у него нет доказательства, и мы не знаем, можно ли его доказать математически. Теоретически он может быть опровергнут, если кто-то разработает альтернативный вид вычислений, способный решать задачи, которые не под силу машине Тьюринга. Но вряд ли это когда-либо произойдет. Поэтому машина Тьюринга считается формальным описанием понятия *алгоритм*.

Представьте себе компьютер — настолько мощный, насколько хотите. Он будет куда быстрее описанной нами машины Тьюринга, которая работает с лентой символов. Но все, что он вычисляет алгоритмическим путем, может быть

вычислено машиной Тьюринга. Это касается даже тех компьютеров, которые мы еще не умеем производить. Наши устройства работают с *битами*, которые принимают лишь одно из двух состояний: 0 или 1. *Квантовые компьютеры* работают с *кубитами*. Если проверить состояние кубита, оно точно так же будет равно либо 0, либо 1. Но между проверками кубит может находиться в комбинированном состоянии, известном как *суперпозиция*. То есть он одновременно может быть равен 0 и 1, пока мы не решим его прочитать; только тогда он примет одно из этих двух значений. Это позволяет квантовым компьютерам представлять сразу несколько состояний вычисления. Они способны быстро решать сложные задачи, с которыми плохо справляются классические компьютеры. К сожалению, на текущем уровне технологического прогресса производство квантовых компьютеров сопряжено с большими трудностями. Тем не менее даже они не могут делать что-то такое, что не под силу машине Тьюринга. Они более эффективны в решении некоторых задач, чем классические ЭВМ или любые машины Тьюринга, если уж на то пошло. Но если машина Тьюринга не может решить какую-то задачу, квантовый компьютер тоже окажется бессильным.

Рамки, в которых проводятся вычисления, определяются машинами Тьюринга. Все, что делает компьютер, можно сделать с помощью ручки и листа бумаги, работая с лентой символов. Все, что выполняется на любом цифровом устройстве, в сущности, является последовательностью элементарных манипуляций с символами. В естественных науках мы наблюдаем окружающий мир и пытаемся объяснить его с помощью фундаментальных принципов. В вычислениях все наоборот: мы пытаемся найти для фундаментальных принципов различные практические применения.

Тьюринг предложил свою машину в качестве модели вычислений еще до появления первых цифровых компьютеров. Это не мешало ему исследовать возможности будущих вычислительных устройств. Говоря об ограничениях, свойственных компьютерам, следует помнить, что в их рамках человеческий интеллект способен творить чудеса. Эти ограничения не мешают нам создавать алгоритмы на все случаи жизни. Когда в Месопотамии была изобретена письменность, она предназначалась для учета, а не для литературных текстов. Первыми писателями, вероятно, стали бухгалтеры, но в итоге, несмотря на такое скромное начало, мы получили Уильяма Шекспира. Кто знает, что нам со временем принесут алгоритмы.

Рамки, в которых проводятся вычисления, определяются машинами Тьюринга. Все, что делает компьютер, можно сделать с помощью ручки и листа бумаги... Все, что выполняется на любом цифровом устройстве, является последовательностью элементарных манипуляций с символами.

Null

Ничто в компьютере.

PageRank

Алгоритм, который используется для ранжирования веб-страниц в зависимости от их значимости. Разработан основателями Google и стал фундаментом одноименной поисковой системы.

ReLU

Нейрон, который использует выпрямитель в качестве функции активации. ReLU расширяется как rectified linear unit (выпрямленная линейная единица).

Tanh (гиперболический тангенс)

Функция активации, похожая на сигмоиду, но со значениями в диапазоне от -1 до 1 .

Softmax

Функция активации, которая принимает на вход вектор вещественных чисел и превращает его в другой вектор, описывающий распределение вероятностей.

«O» большое

Обозначение сложности вычислений. Имея алгоритм и ввод, превышающий какой-то пороговый показатель, оно дает нам максимальное ожидаемое количество шагов, необходимых для выполнения алгоритма. Ввод должен превышать какое-то пороговое значение, поскольку нас интересует поведение алгоритма в контексте больших наборов данных. Вычислительная сложность алгоритма «O» большое гарантирует, что при достаточно большом вводе алгоритму не потребуется больше определенного количества шагов. Например, сложность $O(n^2)$ означает, что, имея ввод размером n , превышающий определенную пороговую величину, мы можем выполнить алгоритм за количество шагов, не превышающее фиксированный множитель n^2 .

Автоматическое дифференцирование

Набор методик для получения производной функции численным, а не аналитическим путем, который потребовал бы использования правил исчисления для дифференцирования функций.

Активация

Генерация вывода нейроном.

Акцентированный отрезок

Часть ритма, на которую делается акцент.

Алгоритм

1. Вернитесь к первой странице этой книги.
2. Прочитайте текущую страницу.
3. Если непонятно, вернитесь к пункту 2. В противном случае переходите к пункту 4.
4. Если следующая страница существует, сделайте ее текущей и перейдите к пункту 2. В противном случае остановитесь.

Алгоритм Дейкстры

Алгоритм для поиска кратчайшего расстояния между двумя узлами в графе, изобретенный в 1956 году молодым голландским ученым Эдсгером Дейкстрой. Он работает с графами, содержащими положительные веса.

Алгоритм Евклида

Алгоритм для поиска наибольшего общего делителя двух целых чисел, представленный в 13-томном сборнике «Начала», который написал древнегреческий математик Евклид (около 300 г. до н. э.). Этот труд посвящен геометрии и теории чисел, начиная с аксиом и заканчивая доказательством теорем, основанных на этих аксиомах. Это древнейшая дошедшая до нас работа в области математики, в которой используется этот дедуктивный подход, что делает ее одной из наиболее влиятельных книг в истории науки.

Алгоритм обратного распространения

Фундаментальный алгоритм обучения нейронных сетей. Сеть корректирует свою конфигурацию (веса и сдвиги) путем передачи обновленных значений от конечного слоя к первому.

Алгоритм Хирхольцера

Алгоритм поиска в графах циклов Эйлера. Опубликован в 1873 немецким математиком Карлом Хирхольцером.

Аппаратное обеспечение

Физические компоненты, из которых состоит компьютер или цифровое устройство. Этот термин дополняет программное обеспечение.

Ациклический граф

Граф, у которого нет циклов.

Бит

Основная единица информации, хранящаяся в компьютере. Бит может принимать одно из двух значений: 0 или 1. Слово «бит» (англ. bit) происходит от словосочетания binary digit (двоичная цифра).

Блуждающий пользователь

Человек, который переходит с одной веб-страницы на другую в соответствии с вероятностью, заданной в матрице Google.

Быстрая сортировка

Метод сортировки, который работает путем многократного перемещения значений относительно выбранного элемента так, чтобы те, которые меньше его, оказались по левую сторону, а остальные — по правую.

Вектор

Горизонтальная строка или вертикальный столбец чисел (или, в более общем смысле, математических выражений). Векторы обычно встречаются в геометрии, где они имеют длину и направление и представлены в виде строки или столбца с их числовыми координатами; но в действительности понятие вектора носит более общий характер — возьмите, к примеру, рейтинговый вектор. Вектор является частным случаем матрицы.

Вес (граф)

Число, назначенное ребру графа. Такое число, к примеру, может представлять награду или наказание, связанные со ссылкой между двумя узлами, соединенными ребром.

Вес (нейрон)

Числовое значение, закрепленное за синапсом нейрона. Из каждого синапса нейрон получает ввод, умноженный на вес этого синапса.

Взвешенный ввод (нейрон)

Сумма произведений входных значений и весов нейрона.

Висячий узел

Узел в алгоритме PageRank, у которого есть только входящие ребра.

Восхождение к вершине

Метафора для описания модели решения задач. Решение находится на вершине горы, и мы должны взобраться на нее, начиная с ее подножия. На каждом шагу может быть несколько альтернативных путей. Мы можем выбрать лучший путь в целом, не лучший путь, который, тем не менее, ведет к вершине, или же путь, ведущий к плато. Последний вариант самый плохой, так как он вынуждает нас вернуться в предыдущую позицию и выбрать другой путь.

Выбор

Выбор между альтернативными наборами шагов в алгоритмах и программировании.

Выпрямитель

Функция активации, которая превращает любой отрицательный ввод в ноли. Остальной вывод прямо пропорционален ее вводу.

Гиперплоскость

Обобщенная плоскость в более чем трех измерениях.

Гиперссылка

Ссылка на другой участок в том же тексте или на другой документ. В вебе гиперссылки ведут с одной веб-страницы на другую, и пользователь может по ним переходить.

Гипертекст

Текст, содержащий гиперссылки.

Глобальный оптимум

Лучшее решение задачи в целом.

Глубокое обучение

Нейронные сети, состоящие из множества скрытых слоев, организованных таким образом, чтобы каждый следующий слой представлял более глубокие концепции, соответствующие более высоким уровням абстракции.

Головной элемент

Первый элемент в списке.

Градиент

Вектор, содержащий все частные производные функции.

Граница решений

Значения одной или нескольких переменных, которые разделяют два разных исхода одного решения.

Граф

Набор узлов (вершин) и ребер (звеньев), которые их соединяют. С помощью графов можно моделировать всевозможные связные структуры, от людей до компьютерных сетей. В результате многие задачи можно смоделировать в виде графов, и на основе этого подхода разработано множество алгоритмов.

Графический процессор (GPU)

Чип, специально предназначенный для выполнения инструкций по созданию и редактированию изображений внутри компьютера.

Двоичный поиск

Поисковый алгоритм, который работает с упорядоченными данными. Мы берем элемент посередине пространства поиска. Если он совпадает с тем, который мы искали, нам повезло. Если нет, мы повторяем эту процедуру в левой или правой части, в зависимости от того, в какую сторону мы промахнулись.

Длина пути

Сумма весов вдоль пути в графе. Если у графа нет весов, это количество звеньев, которые составляют путь.

Жадный алгоритм

Алгоритм, в котором при выборе между двумя стратегиями мы предпочитаем ту, которая приносит лучший результат на текущий момент. Это может не привести к итоговому оптимальному исходу.

Задача коммивояжера

Имея список городов и расстояния между каждым двумя из них, мы должны найти кратчайший маршрут, который позволит посетить каждый город ровно один раз и вернуться в начало. Это, наверное, самая известная нерешаемая задача.

Задача минимизации

Задача, в которой, помимо возможных решений, мы пытаемся найти то, у которого минимальное значение.

Задача о секретарях

Задача об оптимальной остановке. У нас есть группа кандидатов на должность секретаря, и мы беседуем их по очереди. Решение о найме принимается на месте, без возможности пересмотра и знакомства с оставшимися кандидатами.

Задача об оптимальных покупках (марковский момент)

Выбор лучшего момента для остановки в попытке максимизировать вознаграждение, минимизировать наказание.

Закон Мура

Наблюдение, сделанное в 1965 году Гордоном Муром, основателем Fairchild Semiconductor и Intel. Заключается в том, что количество транзисторов на интегральных схемах удваивается каждые два года. Это пример экспоненциального роста.

Замкнутый путь

Путь, который начинается и заканчивается в одном и том же узле графа. Иногда его называют циклическим.

Запись

Набор взаимосвязанных данных, описывающих какую-то сущность для определенного приложения. Например, студенческая запись может содержать идентификационную информацию, год зачисления и копии документов.

Интернет

Глобальная сеть компьютеров и цифровых устройств, взаимосвязанных посредством общего набора коммуникационных протоколов.

Итерация

См. цикл.

Категорийная перекрестная энтропия

Функция потерь, которая вычисляет разницу между двумя распределениями вероятностей.

Квантовый компьютер

Компьютер, который использует квантовые явления для выполнения вычислений. Квантовые компьютеры работают с кубитами вместо битов и могут решать некоторые задачи намного быстрее, чем классические ЭВМ. Изготовление квантовых компьютеров сопряжено с определенными физическими трудностями.

Класс сложности

Набор задач, для решения которых требуется один и тот же объем ресурсов (таких как время или память).

Классификатор

Программа, которая относит наблюдение к одному из нескольких классов.

Ключ

Элемент записи, который мы используем для ее сортировки или поиска. Ключ может быть атомарным, если его нельзя разбить на отдельные части (например, идентификационный номер), или составным, если он состоит из более мелких фрагментов данных (как полное имя, состоящее из фамилии, имени и отчества).

Кратчайший путь

Кратчайший путь между двумя узлами графа.

Кубит

Элементарная единица квантовой информации. Кубит может существовать в суперпозиции двух состояний, 0 и 1. Но, если его измерять, он схлопывается

в одном из двух двоичных значений. Кубит может быть реализован с помощью таких квантовых свойств, как спин электрона.

Линейно разделяемый

Набор данных, наблюдения в котором можно разделить на две категории с помощью прямой линии (в двумерном пространстве), плоскости (в трехмерном пространстве) или гиперплоскостью (если измерений больше трех).

Линейное время

Время, пропорциональное вводу алгоритма. Записывается как $O(n)$.

Линейный поиск

Поисковый алгоритм, в котором каждый элемент анализируется по очереди, пока не будет найдено то, что мы ищем. Его еще называют последовательным поиском.

Логарифм

Операция, противоположная возведению в степень. Логарифм отвечает на следующий вопрос: «В какую степень нужно возвести число, чтобы получить нужное значение?» Если спросить, «в какую степень нужно возвести 10, чтобы получить 1000?», ответом будет 3, поскольку $10^3 = 1000$. Число, возводимое в степень, называется основанием. Если $a^x = b$, мы записываем $\log_a x$. Если $a = 2$, мы записываем $\lg x$.

Логарифмическое время

Время, пропорциональное логарифму ввода алгоритма, например $O(\lg n)$. Присуще хорошим поисковым алгоритмам.

Логлинейное время

Время, пропорциональное произведению размера ввода алгоритма и логарифма этого ввода, например $O(n \lg n)$. Присуще хорошим алгоритмам сортировки.

Локальный оптимум

Решение, которое превосходит все другие решения, находящиеся по соседству, но не является лучшим в целом. Соседним называется решение, которое находится в одном шаге от текущего.

Матрица

Прямоугольный массив, который, как правило, состоит из чисел или, в более общем виде, математических выражений. Содержимое матрицы организовано в виде горизонтальных строк и вертикальных столбцов.

Матрица Google

Специального рода матрица (модификация матрицы гиперссылок), которая используется в степенном методе в алгоритме PageRank.

Матрица гиперссылок

Матрица, представляющая структуру графа; похожа на матрицу смежности, но каждый элемент в ней делится на количество ненулевых элементов в соответствующем ряду.

Матрица смежности

Матрица, представляющая граф. Для каждой вершины графа предусмотрены строка и столбец. Каждая запись, строка и столбец которой соответствуют двум вершинам, соединенным ребром графа, содержит 1; все остальные записи равны 0.

Машина Тьюринга

Идеализированная (абстрактная) машина, описанная Аланом Тьюрингом. Состоит из бесконечной ленты и подвижной головки, которая считывает и записывает символы на этой ленте, руководствуясь набором заданных правил. Машина Тьюринга может реализовать любой алгоритм, поэтому ее можно использовать в качестве модели возможных вычислений.

Машинное обучение

Применение алгоритмов, которые решают задачи путем автоматического обучения на примерах.

Метка

В машинном обучении это значение, представляющее категорию, к которой принадлежит наблюдение. В ходе обучения компьютеру предоставляются задачи вместе с решениями; если задача состоит в классификации, решение представляет собой метку, соответствующую определенному классу.

Метод перегруппировки

Самоорганизующийся поисковый алгоритм. Найденный элемент меняется местами с предыдущим. Таким образом популярные элементы перемещаются ближе к началу.

Метод строковой сортировки

Метод сортировки, который работает со своими ключами как с последовательностью символов. Например, ключ 1234 воспринимается как строка символов 1, 2, 3, 4, а не как число 1234.

Мультиграф

Граф, в котором одно и то же ребро может повторяться.

Мультимножество

Множество, в котором элемент может встречаться в нескольких экземплярах; в математике элемент не может входить в обычное множество больше одного раза.

Мусор на входе — мусор на выходе

Если программе предоставлены неверные данные вместо ожидаемого ввода, не стоит ждать чудес: программа выдаст неверный результат вместо ожидаемого вывода.

Наибольший общий делитель

Наибольшее целое число, на которое делятся два других целых числа.

Направленный граф

Граф, в котором ребра направлены. Направленный граф также называют ориентированным (или оргграфом, если коротко).

Нейрон

Клетка, которая является элементарным составным элементом нервной системы. Она принимает сигналы от одних нейронов и передает их другим.

Ненаправленный граф

Граф с ненаправленными ребрами.

Нерешаемая задача

Задача, решение которой с использованием лучшего известного нам алгоритма заняла бы слишком много времени для того, чтобы рассматривать что-то кроме тривиальных случаев.

Обратная ссылка

Ссылка, которая указывает на просматриваемую нами веб-страницу (или же веб-страница с этой ссылкой).

Обучение

Процесс предоставления алгоритму машинного обучения демонстрационного ввода, на основе которого он может научиться выдавать правильный вывод.

Обучение без учителя

Подход к машинному обучению, в котором алгоритму предоставляются задачи без решений. Таким образом алгоритм должен определить, каким должен быть ввод для получения нужных решений.

Обучение с учителем

Подход к машинному обучению, в котором алгоритму предоставляются как сами задачи, так и их решения.

Онлайн-алгоритм

Алгоритм, которому не нужен полный ввод для решения задачи. Онлайн-алгоритм получает ввод постепенно, по мере его появления, и на каждом этапе генерирует решение на основе уже принятого ввода.

Оптимизаторы

Алгоритмы, которые оптимизируют значение функции. В машинном обучении оптимизаторы обычно минимизируют значение функции потерь.

Перемещение к началу

Самоорганизующийся поисковый алгоритм. При нахождении искомого элемента мы перемещаем его в начало.

Переподгонка (переобучение)

Эквивалент зубрежки в машинном обучении. Модель, которую мы пытаемся обучить, слишком хорошо адаптируется к учебным данным. В результате она

неспособна предсказывать правильные значения для других, неизвестных данных.

Переполнение

Выход за пределы диапазона допустимых значений на компьютере.

Перестановка

Изменение порядка размещения каких-то данных.

Перфокарты

Кусок тонкого картона с дырами, размещение которых обозначает какую-то информацию. Эти карты использовались в старых компьютерах и даже раньше, в устройствах вроде жаккардового ткацкого станка, в котором они описывали тканые узоры.

Перцептрон

Искусственный нейрон, для активации которого используется ступенчатая функция.

Плотное соединение

Слои в нейронной сети организованы так, что все нейроны одного слоя соединяются со всеми нейронами следующего.

Подгонка

Процесс машинного обучения на основе данных. В этом процессе создается модель, которая соответствует наблюдениям.

Полиномиальное время

Время, пропорциональное вводу алгоритма, возведенному в фиксированную степень. Например, $O(n^2)$.

Поразрядная сортировка

Метод сортировки, который работает путем разбиения ключей на составные части (например, на отдельные цифры, если это цифровой ключ) и распределения по группам в зависимости от значений этих составных частей (десять групп, по одной для каждой цифры). Вначале группы формируются на основе последней цифры, затем мы складываем их вместе и распределяем в зависимости от предпоследнего разряда и т. д. Дойдя до первого разряда,

мы получаем отсортированную группу. Это метод строковой сортировки, поскольку он работает с числовыми ключами как со строками цифр.

Последовательность

Последовательность шагов, выполняемых один за другим в алгоритмах и программировании.

Потеря

Разница между фактическим и желаемым выводом алгоритма машинного обучения. Обычно вычисляется функцией потерь.

Приближение

Решение задачи с помощью алгоритма, который может дать не оптимальный, но достаточно близкий к нему результат.

Программа

Набор инструкций, написанных на языке программирования, которые описывают вычислительный процесс.

Программирование

Искусство написания компьютерных программ.

Программная ошибка (англ. bug)

Ошибка в программе. С помощью этого термина Томас Эдисон описывал техническую неисправность. На ранних этапах развития компьютерной техники в оборудование пробирались настоящие насекомые (bugs) и выводили его из строя. Например, в 1947 году в компьютер Harvard Mark II залетел мотылек и отпечатался в его журнале, который впоследствии вошел в коллекцию Национального музея американской истории при Смитсоновском институте.

Программное обеспечение (software)

Набор программ, выполняющихся на компьютере или цифровом устройстве. Этот термин дополняет аппаратное обеспечение. Он использовался в разных областях еще до появления компьютеров. В 1850-х годах сборщики мусора называли разлагающиеся материалы soft-ware («мягкие изделия»), а остальные — hard-ware («твердые изделия»). Эта терминология может поднять настроение тем, кому никак не удастся заставить компьютер делать то, что от него требуется.

Производная

Наклон функции в заданной точке или скорость ее изменения. Например, ускорение — это производная скорости (быстрота изменения скорости по времени).

Пространство поиска

Область значений, по которым мы ищем.

Путь

Последовательность ребер графа, которые соединяют последовательность узлов.

Путь выполнения

Последовательность шагов, которые алгоритм выполняет в ходе своей работы.

Разделяй и властвуй

Метод решения задач, в котором задача разбивается на более мелкие подзадачи (обычно две) снова и снова, пока они не становятся настолько простыми, что их решение становится тривиальным.

Разреженная матрица

Матрица, большинство элементов которой равны нулю.

Рандомизация

Использование в алгоритмах элемента случайности. Это дает алгоритму возможность находить хорошие решения в большинстве случаев, даже если поиск оптимального решения является невыполнимым с вычислительной точки зрения.

Раскрашивание вершин

Назначение цветов вершинам графа таким образом, чтобы никакие две смежные вершины не были одного цвета.

Раскрашивание графа

Раскрашивание ребер или вершин графа.

Раскрашивание ребер

Назначение цветов ребрам графа таким образом, чтобы никакие два смежных ребра не были одного цвета.

Распознавание изображений

Вычислительная задача по распознаванию образов в изображениях.

Расщепление

Разбиение материала на более мелкие части. В ядерной физике роль такого материала играет тяжелое ядро, излучающее большое количество протонов и нейтронов в результате столкновения с высокоэнергетической частицей.

Рейтинговый вектор

Вектор, содержащий рейтинги страниц графа.

Релаксация

Метод, который используется в алгоритмах для работы с графами. Искомым элементам назначаются худшие значения из возможных, и затем алгоритм их постепенно корректирует. Таким образом мы начинаем с самых крайних значений и плавно их смягчаем, приближая их к конечному результату.

Самоорганизующийся поиск

Поисковый алгоритм, который учитывает популярность элементов, перемещая их туда, где их можно будет быстрее найти.

Сдвиг

Числовое значение, присвоенное нейрону, которое определяет его склонность к активации.

Сигмоида

Функция в форме буквы S, чьи значения находятся в диапазоне от 0 до 1.

Синапс

Соединение между нейронами.

Скрытый слой

Слой нейронной сети, не связанный напрямую с ее вводом или выводом.

Сложность (вычислительная сложность)

Время, необходимое для выполнения алгоритма. Время выражается в последовательности элементарных вычислительных шагов, которые нужно выполнить.

Собственный вектор

В линейной алгебре это вектор, дающий при умножении на определенную матрицу тот же вектор, умноженный на число (которое называют собственным). PageRank находит первый собственный вектор матрицы Google, то есть собственный вектор матрицы Google с наибольшим собственным значением (1).

Сортировка слиянием

Метод сортировки, который работает путем многократного слияния все больших наборов отсортированных элементов.

Сортировка вставками

Метод сортировки, в котором каждый элемент вставляется в подходящую позицию среди уже отсортированных элементов.

Сортировка выбором

Метод сортировки, в котором мы раз за разом находим наименьший из неупорядоченных элементов и помещаем его в правильную позицию.

Социальная сеть

Граф, в котором роль узлов играют люди, а связи между ними являются ребрами.

Список

Структура данных, содержащая элементы. Каждый элемент указывает на следующий; исключение составляет последний элемент, который указывает в никуда или, как еще говорят, на null. Таким образом элементы связаны между собой, поэтому подобные списки также называют связными.

Срабатывание (нейрон)

См. активация (нейрон).

Степенной метод

Алгоритм, который берет вектор, умножает его на матрицу и затем умножает результат на ту же матрицу снова и снова, пока он не придет к стабильному значению. Степенной метод — ключевой элемент PageRank; вектор, в котором он сходится, является первым собственным вектором матрицы Google.

Степень (узел)

Количество ребер узла.

Строка

Последовательность символов. Традиционно символами считали буквы алфавита, но в настоящее время содержимое строки зависит от контекста; это могут быть цифры, буквы, знаки препинания или даже недавно изобретенные символы, такие как эмодзи.

Структура данных

Способ организации данных таким образом, чтобы их можно было обрабатывать с помощью набора определенных готовых операций.

Табулятор

Электромеханическое устройство, которое могло считывать перфокарты и использовать эту информацию для выдачи числовых значений.

Тезис Черча — Тьюринга

Гипотеза о том, что любые вычисления, которые можно описать с помощью алгоритма, могут быть выполнены машиной Тьюринга.

Тестовый набор данных

Данные, которые откладываются в сторону во время обучения. С их помощью мы проверяем, насколько хорошо конкретный метод машинного обучения проявляет себя при работе с реальным вводом.

Узел

Элемент в различных структурах данных. Элементы списка называют узлами.

Указатель

Участок компьютерной памяти, хранящий адрес другого участка компьютерной памяти. Таким образом первый ссылается на второй.

Унарная система счисления

Система счисления, в которой числа представлены одним символом; например, один штрих обозначает единицу, а III — число 3.

Управляющая структура

Три возможных способа комбинирования шагов алгоритма или программы: последовательность, выбор и итерация.

Учебный набор данных

Данные, которые мы предоставляем алгоритмам машинного обучения, чтобы научить их решать задачи.

Факториал

Факториал натурального числа n является произведением всех чисел от 1 до n включительно. Мы используем обозначение $n!$, поэтому $n! = 1 \times 2 \times \dots \times n$. Это определение можно расширить до всех вещественных чисел, но здесь нас это не интересует.

Факториальная сложность

Вычислительная сложность, которая соответствует факториальному росту. В обозначении больше «О» это выглядит как $O(n!)$.

Функция активации

Функция, которая определяет вывод нейрона на основе его ввода.

Хроматический индекс

Минимальное количество цветов, которые нужны, чтобы раскрасить все ребра графа (задача раскраски графа).

Центральное процессорное устройство

Чип, который выполняет инструкции программы внутри компьютера.

Цикл

Путь, который начинается и заканчивается в одном и том же узле графа. А также повторяющаяся последовательность инструкций в компьютерной программе. Цикл завершается при выполнении условия. Цикл, который никогда не заканчивается, называется бесконечным и обычно является результатом ошибки, не давая программе остановиться. См. итерация.

Частная производная

Производная функции для одной из множества переменных, в то время как остальные переменные принимаются за константы.

Число Эйлера

Математическая константа e , приблизительно равная 2,71828. Это предел $(1 + 1/n)^n$, где n стремится к бесконечности.

Эвристика

Стратегия выбора между альтернативными решениями в алгоритме. Жадная эвристика склоняет нас к выбору, который выглядит лучше всего прямо сейчас (без учета того, что может произойти в будущем).

Эйлеров путь

Маршрут, проходящий по каждому ребру графа ровно один раз. Его еще называют «эйлерова цепь».

Эйлеров цикл

Эйлеров путь, который начинается и заканчивается в одном и том же узле.

Экспоненциальный рост

Модель роста, в которой количество элементов последовательно умножается само на себя. Например, мы начинаем с a элементов, затем получаем $a \times a$, затем $a \times a \times a$; в целом это Формула. При экспоненциальном росте числа быстро увеличиваются.

Эпоха

Этап в процессе машинного обучения, охватывающий весь набор учебных данных.

Эффект Матфея

Явление, состоящее в том, что богатые становятся богаче, а бедные беднеют. Названо в честь стиха из Евангелия от Матфея (25:29) и, как показывает практика, применимо ко многим сферам, а не только к материальному богатству.

Язык программирования

Искусственный язык, с помощью которого можно описывать этапы вычислений. Язык программирования может быть выполнен на компьютере. У языков программирования, как и у естественных языков, есть синтаксис и грамматика, которые определяют, что на них можно писать. Их существует большое количество, и постоянно появляются новые языки, призванные сделать программирование более продуктивным (или просто потому, что многие не могут удержаться от создания собственного языка в надежде на то, что он получит широкое распространение). Языки программирования бывают высокоуровневыми и низкоуровневыми; первые похожи на естественный язык, а последние состоят из элементарных конструкций, отражающих соответствующее аппаратное обеспечение.

Предисловие

1. Если вам интересны эти и другие индикаторы глобального успеха, достигнутого благодаря идеям Просвещения, ищите «Pinker 2018».

Глава 1

1. Передача The Algorithmic Age («Алгоритмическая эра») транслировалась 8 февраля 2018 года на *Radio Open Source*.
2. Подробности об алгоритмах древнего Вавилона ищите в [Кнут, 1972].
3. Алгоритм для распределения пульсации по временным отрезкам в SNS был предложен Эриком Бьерклундом (1999). В 2005 году Готфрид Туссен заметил сходство с ритмами, и его работа является основой для нашей трактовки. Подробности ищите в исследованиях Демайна и др. (2009). Если вас интересует пространственный анализ алгоритмов и музыки, см. [Туссен, 2013].
4. Эти критерии определил Дональд Кнут (1997, разд. 1), который тоже начинает свою трактовку с алгоритма Евклида.
5. Если вас интересует перечисление путей в сетке, см. [Кнут, 2011, 253–255]; там же дается пример и изображения путей. Алгоритм, который определяет количество возможных путей, ищите в [Ивашита и др., 2013].
6. Описания этого числа можно найти в [Тайсон, Страусс и Готт, 2016, 18–20]. В романе Дейва Эггерса, «Сфера», одна технологическая компания подсчитывает количество песчинок в пустыне Сахара.
7. Лист бумаги, который складывается n раз, должен быть достаточно большим. Если всегда складывать его вдоль одной и той же оси, он должен быть довольно длинным. Длина определяется по формуле $L = \frac{\pi t}{6}(2^n + 4)(2^n - 1)$, где t — толщина листа, а n — количество сгибов. Если сложить квадратный лист бумаги в чередующихся направлениях, его ширина должна быть равна $W \approx \pi t 2^{(3/2)(n-1)}$. Эти формулы сложнее, чем обычное возведение в квадрат, так как при каждом складывании часть листа уходит на формирование изгибов вдоль согнутого края; именно для вычисления этих изгибов

здесь используется число π . Эти формулы в 2002 году предложила Бритни Кристал Галливан, которая в то время была старшеклассницей. Она решила продемонстрировать, что рулон туалетной бумаги длиной 365 метров можно сложить пополам всего 12 раз. Хорошее введение в степени степеней (включая этот пример) можно найти в [Строгаз, 2012, глава 11].

8. См. Transistor count, «Википедия», https://en.wikipedia.org/wiki/Transistor_count.
9. Это вызвано тем, что для сравнения n элементов друг с другом вам нужно взять один элемент и сравнить его с остальными $n - 1$ элементами, затем взять еще один и сравнить его с $n - 2$ элементами (с первым вы его уже сравнили) и т. д. Таким образом получается $1 + 2 + \dots + (n - 1) = n(n - 1) / 2$ сравнений. Затем вы получаете $O(n(n - 1) / 2) = O(n^2 - n / 2) = O(n^2)$, потому что, согласно определению большого «О», алгоритм, время работы которого не превышает $O(n^2)$, точно успеет выполняться за время $O(n^2 - n / 2)$.

Глава 2

1. Изображение находится в общественном состоянии и взято из Wikipedia Commons по адресу: https://commons.wikimedia.org/wiki/File:Konigsberg_Bridge.png.
2. Документ (Eulerho, 1736) доступен на сайте Euler Archive (<http://eulerarchive.maa.org>), который находится в ведении Математической ассоциации США. Английский перевод находится в [Биггз, Ллойд и Уилсон, 1986].
3. Графам посвящен обширный раздел научной литературы. Хорошей отправной точкой будет [Бенджамин, Чартранд и Чанг, 2015].
4. Изображение, опубликованное в оригинальном издании (Eulerho, 1736), взято из Wikipedia Commons по адресу: https://commons.wikimedia.org/wiki/File:Solutio_problematism_ad_geometriam_situs_pertinentis,_Fig._1.png. Находится в общественном доступе.
5. Изображение из [Кекуле, 1872], взятое из «Википедии» по адресу: [https://en.wikipedia.org/wiki/Benzene#/media/File:Historic_Benzene_Formulae_Kekul%C3%A9_\(original\).png](https://en.wikipedia.org/wiki/Benzene#/media/File:Historic_Benzene_Formulae_Kekul%C3%A9_(original).png). Находится в общественном достоянии.
6. Оригинальное издание на немецком языке ищите в [Хирхольцер, 1873].
7. Если вам интересны подробности об алгоритме Хирхольцера и других алгоритмах для эйлерового пути, см. [Флейшнер, 1991]. Использование графов в процессе составления генома описывается в [Певзнер, Танг и Уотерман, 2001; Компью, Певзнер и Теслер, 2001].

8. Анализ оптимальности жадного онлайн-алгоритма для раскрашивания ребер, а также пример звездообразного графа, иллюстрирующего худший случай, можно найти в [Бар-Ной, Мотвани и Наор, 1992].
9. В оригинальной басне двумя персонажами были муравей и цикада. Они также встречаются в латинских переводах с древнегреческого и пересказе Жана де Лафонтена на французском языке.
10. История об изобретении алгоритма изложена Дейкстрой в интервью Misa and Frana в 2010 году.

Глава 3

1. Первое описание эффекта Матфея встречается в [Мертон, 1968]. Обзоры ряда явлений с неравномерными распределениями ищите в [Барабаси и Мартон, 2016; Уест, 2017]. Если вас интересует несоответствие распределений роста и богатства среди посетителей стадиона, см. [Талеб, 2007].
2. Самоорганизующийся поиск представлен Джоном Маккейбом в 1965 году. Анализ производительности методов перемещения в начало и перегруппировки можно найти в [Ривест, 1976; Бахраш, Эль-Янив и Рейнштедтлер, 2002].
3. Задача о секретарях опубликована в колонке Мартина Гарднера в февральском выпуске журнала *Scientific American* 1960 года. Решение дано в мартовском выпуске. Об истории этой задачи можно почитать в [Фергюсон, 1989]. В 2006 году Д. Нил Бирден предложил решение для варианта, составленного без использования принципа «все или ничего». В [Мэтт Паркер, 2014, 11 глава] находится описание этой задачи вместе с несколькими математическими идеями и введением в компьютеры.
4. Двоичный поиск появился на заре компьютерной эпохи [Кнут 1998]. Джон Мокли, один из архитекторов ENIAC, первого цифрового компьютера общего назначения, описал его в 1946 году. Если вас интересует богатая история двоичного поиска, см. [Бентли, 2000; Пэттис, 1988; Блох, 2006].

Глава 4

1. Холлерит, 1894.
2. Сортировки выбором и вставками существуют с момента появления первых компьютеров; они включены в исследование методов сортировки в 1950-х годах [Friend, 1956].

3. Согласно [Кнуту, 1998, 170], идея, стоящая за поразрядной сортировкой в том виде, в котором мы ее видели здесь, существует как минимум с 1920-х годов.
4. Подбрасывание монеты следует из $1 / 52! \approx (1 / 2)^{226}$. Пример с выбором случайного атома позаимствован у Дэвида Хэнда (2014), согласно которому, вероятность меньше одного из 10^{50} является пренебрежительно низкой в космических масштабах.
5. См. [Хоар, 1961a, 1961b, 1961c].
6. Больше о рандомизированных алгоритмах можно найти в [Митценмахер и Апфол, 2017].
7. О жизни фон Неймана и атмосферы, царившей в эпоху появления первых цифровых компьютеров, можно почитать в [Дайсон, 2012]. Презентацию программы для сортировки слиянием фон Неймана см. [Кнут, 1970].

Глава 5

1. Оригинальная версия алгоритма PageRank опубликована Брином и Пэйджем в 1998 году. Мы лишь поверхностно затронули ее математические аспекты. Более глубокий обзор ищите в [Брайан и Лейсе, 2006]. Введение в поисковые системы и PageRank представлено в [Лэнгвил и Мейер, 2006; Берри и Браун, 2005]. Еще одним важным алгоритмом ранжирования является Hupertext Induced Topic Search или HITS [Клейнберг, 1998, 1999], разработанный раньше, чем PageRank. Задолго до этого, еще в 1940-х годах, похожие идеи возникали в других областях, например в социометрии, науке об измерении социальных отношений, и эконометрике, науке о количественных аспектах экономических принципов [Францешет, 2011].

Глава 6

1. Современные технологии позволяют рассматривать нейроны гораздо более детально, но Рамон-и-Кахаль был пионером в этой области, и его рисунки считаются одними из самых элегантных иллюстраций в истории науки. В Интернете можно найти множество изображений нейронов, но этого рисунка нам достаточно; вы можете сами убедиться в красоте и неувядающей силе иллюстраций Рамон-и-Кахаля, выполнив простой поисковый запрос. Приведенный здесь рисунок находится в общественном достоянии и доступен по адресу: <https://commons.wikimedia.org/wiki/File:PurkinjeCell.jpg>.

2. Если быть точным, название «сигмоида» происходит от греческой буквы сигма, Σ , но по своему внешнему виду эта функция больше напоминает латинскую букву S .
3. Касательная угла определяется как отношение противоположной стороны прямоугольного треугольника к смежной или как синус угла, поделенный на косинус угла в единичной окружности. Гиперболический тангенс определяется как отношение между гиперболическим синусом и гиперболическим косинусом угла на гиперболе.
4. В 1943 году Уоррен Маккаллок и Уолтер Питтс предложили первый искусственный нейрон. В 1957 году Фрэнк Розенблатт описал перцептрон. Этим изобретениям уже больше полувека. Как же так получилось, что нейронные сети набрали популярность лишь недавно? В 1969 году Марвин Мински и Сеймур Пейперт в своей знаменитой одноименной книге нанесли сильный удар по идее перцептрона, указав на то, что ему свойственны фундаментальные вычислительные ограничения. Это в сочетании с несовершенным оборудованием тех дней привело к так называемой зимней спячке в нейронных вычислениях, которая длилась вплоть до 1980-х, пока исследователи не научились создавать и обучать сложные нейронные сети. Это возродило интерес к данной области, но, чтобы довести нейронные сети до уровня популярности, который наблюдается в последние пять лет, пришлось проделать много работы.
5. Одна из трудностей при изучении нейронных сетей связана с запутанными обозначениями, которые, как может показаться, понятны только избранным. Но на самом деле они не такие уж и сложные, если понимать, о чем идет речь. Мы часто можем встретить производные; производная функции $f(x)$ по x записывается как $\frac{df(x)}{dx}$. Частная производная функции f со многими переменными, такими как x_1, x_2, \dots, x_n , записывается как $\frac{\partial f}{\partial x_i}$. Градиент обозначает как $\nabla f = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})$.
6. Алгоритм обратного распространения появился на свет в середине 1980-х [Румельхарт, Хинтон и Уильямс, 1986], хотя различные его вариации встречались еще в 1960-х годах.
7. Это изображение взято из набора данных Fashion-MNIST [Сяо, Расул и Вольграф, 2017], разработанного для измерения производительности машинного обучения. Раздел вдохновлен учебным руководством по основам классификации в TensorFlow, доступным по адресу: https://www.tensorflow.org/tutorials/keras/basic_classification.
8. Описание первой системы, победившей чемпиона по го, можно найти в [Сильвер и др., 2016]. Ее улучшенная версия, не требующая человеческих

знаний в форме сыгранных ранее партий, описывается в [Сильвер и др., 2017].

9. Литература, посвященная глубокому обучению, является довольно обширной. Если вас интересует комплексное введение в эту область, см. [Гудфеллоу, Бенджио и Курвилль, 2016]. Краткий обзор можно найти в [Леун, Бенджио и Хинтон, 2015]. Темы глубокого и машинного обучения освещаются в [Алпайдин, 2016]. Исследование на тему автоматических поисковых методов в нейронной архитектуре можно почитать в [Елскен, Хендрик Метзен и Хаттер, 2018].

Послесловие

1. Помимо Тьюринга, другими известными исследователями в этой области были Мэри Эннинг, Пол Дирак, Розалинд Фрэнклин, Уильям и Кэролин Хершел, Дороти Ходжкин, Ада Лавлейс и Чарльз Бэббидж, Стивен Хокинг, Джеймс Клерк Максвелл, Сриниваса Рамануджан, Эрнест Резерфорд и Фредерик Сэнгер. Бэббидж, Лавлейс и Тьюринг были пионерами в сфере компьютеров. Бэббидж (1791–1871) изобрел первую вычислительную машину и выработал фундаментальные принципы, на которых основаны современные компьютеры. Лавлейс (1815–1852), дочь лорда Байрона, работала вместе с Бэббиджем, осознала потенциал его изобретения и создала первый алгоритм, который можно было выполнить на таком устройстве. В наши дни она считается первым компьютерным программистом. Дизайн купюры номиналом £50 можно посмотреть в официальном объявлении по адресу: <https://www.bankofengland.co.uk/news/2019/july/50-pound-banknote-character-announcement>.
2. См. замечательную биографию, составленную Эндрю Ходжесом (1983). Роль Тьюринга во взломе немецкой шифровальной машины Энигма освещена в фильме «Игра в имитацию», вышедшем в 2014 году.
3. Описание машины доступно в [Тьюринг, 1937, 1938].
4. Пример машины Тьюринга позаимствован из [Джон Хопкрофт, Раджив Мотвани и Джеффри Уллман, 2001, глава 8]. Рисунок основан на примере Себастьяна Сардины, доступном по адресу: <http://www.texample.net/tikz/examples/turing-machine-2/>.
5. Подробности о тезисе Черча — Тьюринга см. [Льюис и Пападимитрио, 1998, глава 5]. Обсуждение истории тезиса Черча — Тьюринга и его различных вариантов можно найти в [Копленд и Шагрир, 2019].

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Algorismus 15
directed acyclic graph 43
IBM 75
NASA 100
О большое 32
PageRank 11
Pink Floyd 19
- автоматическое дифференцирование 149
- Алан Тьюринг 150
алгоритм PageRank 97
алгоритм Евклида 22
алгоритмическая эра 13
алгоритм обратного распространения потери 137
алгоритмы линейного времени 35
Атомарный ключ 77
Ациклический граф 43
- Вероятностные алгоритмы 91
Вычислительная сложность 31
- Герман Холлерит 74
гиперболический тангенс 124
гиперплоскость 132
гипертекст 98
глубокое обучение 120
Гордон Мур 34
градиент 131
граница решений 125
графы 39
- Двоичный поиск 70
ДНК 44
- жадные алгоритмы 49
- задача коммивояжера 37
Задача о секретаре 67
задача о семи кёнигсбергских мостах 38
Закон Мура 34
- Иоганн Кеплер 66
- Карл Хирхольцер 45
категорийная перекрестная энтропия 144
Квантовые компьютеры 157
- Ларри Пейдж 97
Линейный поиск 62
Логлинейное время 36
Локальный оптимум 49
- матрица Google 116
матрица гиперссылок 108
Матрица смежности 107
машина Тьюринга 152
машинное обучение 127
Метод перегруппировки 65
метод сортировки строк 84
Мультиграф 41
мультимножество 41
Мухаммад ибн Муса аль-Хорезми 15

наибольший общий делитель 22
нейронные сети 133
обратные ссылки 103
обучение без учителя 127
обучение с учителем 127
организация турнира 46
ориентированный граф 41

переполнение 72
перестановка данных 77
перфокарты 74
перцептрон 124
Поиск восхождением к вершине 49
поразрядная сортировка 81
приблизительный поиск 59

разреженная матрица 118
распознавание образов 138
рейтинговый вектор графа 107
Роберт К. Мертон 64

самоорганизующийся поиск 66
Сантьяго Рамон-и-Кахаль 121
Связный список 61
Сергей Брин 97
сигмоида 124
синапсы 121
скорость изменения функции 130
сортировка вставками 81
сортировка выбором 79

Сортировка данных 75
сортировка слиянием 91
Составной ключ 77
социальная сеть 41
степенной метод 111
степень узла графа 49

тестовый набор данных 127

Управляющие структуры 23
учебный набор данных 127

Факториальная сложность 36
функция softmax 141
Функция активации 123
функция выпрямитель 124

Хроматический индекс 49

частная производная потери 131
число Эйлера 35

эвристики 49
Эдсгер Дейкстра 52
Эйлеров цикл 40
экспоненциальная сложность 36
Экспоненциальный рост 34
эффект Матфея 64

язык программирования 25

- Alpaydin, Ethem. 2016. *Machine Learning*. Cambridge, MA: MIT Press.
- Bachrach, Ran, Ran El-Yaniv, and Martin Reinstädler. 2002. “On the Competitive Theory and Practice of Online List Accessing Algorithms.” *Algorithmica* 32 (2): 201–245.
- Barabási, Albert-László, and Pósfai Márton. 2016. *Network Science*. Cambridge: Cambridge University Press.
- Bar-Noy, Amotz, Rajeev Motwani, and Joseph Naor. 1992. “The Greedy Algorithm Is Optimal for Online Edge Coloring.” *Information Processing Letters* 44 (5): 251–253.
- Bearden, J. Neil. 2006. “A New Secretary Problem with Rank-Based Selection and Cardinal Payoffs.” *Journal of Mathematical Psychology* 50:58–59.
- Benjamin, Arthur, Gary Chartrand, and Ping Zhang. 2015. *The Fascinating World of Graph Theory*. Princeton, NJ: Princeton University Press.
- Bentley, Jon. 2000. *Programming Pearls*. 2nd ed. Boston: Addison-Wesley.
- Berry, Michael W., and Murray Browne. 2005. *Understanding Text Engines: Mathematical Modeling and Text Retrieval*. 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics.
- Biggs, Norman L., E. Keith Lloyd, and Robin J. Wilson. 1986. *Graph Theory, 1736–1936*. Oxford: Clarendon Press.
- Bjorklund, Eric. 1999. “The Theory of Rep-Rate Pattern Generation in the SNS Timing System.” SNS-NOTE-CNTRL-99. Spallation Neutron Source. ics-web.sns.ornl.gov/timing/Rep-Rate%20Tech%20Note.pdf.
- Bloch, Joshua. 2006. “Extra, Extra — Read All about It: Nearly All Binary Searches and Mergesorts Are Broken.” *Google AI Blog*, June 2. google-research.blogspot.it/2006/06/extra-extra-read-all-about-it-nearly.html.
- Brin, Sergey, and Lawrence Page. 1998. “The Anatomy of a Large-Scale Hypertextual Web Search Engine.” *Computer Networks and ISDN Systems* 30 (1–7): 107–117.
- Bryan, Kurt, and Tanya Leise. 2006. “The \$25,000,000,000 Eigenvector: The Linear Algebra behind Google.” *SIAM Review* 48 (3): 569–581.
- Charniak, Eugene. 2018. *Introduction to Deep Learning*. Cambridge, MA: MIT Press.

- Compeau, Phillip E. C., Pavel A. Pevzner, and Glenn Tesler. 2011. “How to Apply de Bruijn Graphs to Genome Assembly.” *Nature Biotechnology* 29 (11): 987–991.
- Copeland, B. Jack, and Oron Shagrir. 2019. “The Church-Turing Thesis: Logical Limit or Breachable Barrier?” *Communications of the ACM* 62 (1): 66–74.
- Demaine, Erik D., Francisco Gomez-Martin, Henk Meijer, David Rappaport, Perouz Taslakian, Godfried T. Toussaint, Terry Winograd, and David R. Wood. 2009. “The Distance Geometry of Music.” *Computational Geometry: Theory and Applications* 42 (5): 429–454.
- Dyson, George. 2012. *Turing’s Cathedral: The Origins of the Digital Universe*. New York: Vintage Books.
- Elskens, Thomas, Jan Hendrik Metzen, and Frank Hutter. 2018. “Neural Architecture Search: A Survey.” ArXiv, Cornell University. August 16. arxiv.org/abs/1808.05377.
- Eulerho, Leonhardo. 1736. “Solutio Problematis Ad Geometrian Situs Pertinentis.” *Commentarii Academiae Scientiarum Imperialis Petropolitanae* 8:128–140.
- Ferguson, Thomas S. 1989. “Who Solved the Secretary Problem?” *Statistical Science* 4 (3): 282–289.
- Fleischner, Herbert, ed. 1991. “Chapter X Algorithms for Eulerian Trails and Cycle Decompositions, Maze Search Algorithms.” In *Eulerian Graphs and Related Topics*, 50: X.1– X.34. Amsterdam: Elsevier.
- Franceschet, Massimo. 2011. “PageRank: Standing on the Shoulders of Giants.” *Communications of the ACM* 54 (6): 92–101.
- Friend, Edward H. 1956. “Sorting on Electronic Computer Systems.” *Journal of the ACM* 3 (3): 134–168.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. Cambridge, MA: MIT Press.
- Hand, David J. 2014. *The Improbability Principle: Why Coincidences, Miracles, and Rare Events Happen Every Day*. New York: Farrar, Straus and Giroux.
- Hawking, Stephen. 1988. *A Brief History of Time*. New York: Bantam Books.
- Hierholzer, Carl. 1873. “Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu Umfahren.” *Mathematische Annalen* 6 (1): 30–32.
- Hoare, C. A. R. 1961a. “Algorithm 63: Partition.” *Communications of the ACM* 4 (7): 321.

- Hoare, C. A. R. 1961b. “Algorithm 64: Quicksort.” *Communications of the ACM* 4 (7): 321.
- Hoare, C. A. R. 1961c. “Algorithm 65: Find.” *Communications of the ACM* 4 (7): 321–322.
- Hodges, Andrew. 1983. *Alan Turing: The Enigma*. New York: Simon and Schuster.
- Hollerith, Herman. 1894. “The Electrical Tabulating Machine.” *Journal of the Royal Statistical Society* 57 (4): 678–689.
- Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. 2001. *Introduction to Automata Theory, Languages, and Computation*. 2nd ed. Boston: Addison-Wesley.
- Iwashita, Hiroaki, Yoshio Nakazawa, Jun Kawahara, Takeaki Uno, and Shinichi Minato. 2013. “Efficient Computation of the Number of Paths in a Grid Graph with Minimal Perfect Hash Functions.” Technical Report TCS-TR-A-13-64. Division of Computer Science, Graduate School of Information Science, Technology, Hokkaido University.
- Kekulé, August. 1872. “Ueber Einige Condensationsprodukte Des Aldehyds.” *Annalen der Chemie und Pharmacie* 162 (1): 77–124.
- Kleinberg, Jon M. 1998. “Authoritative Sources in a Hyperlinked Environment.” In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, 668–677. Philadelphia: Society for Industrial and Applied Mathematics.
- Kleinberg, Jon M. 1999. “Authoritative Sources in a Hyperlinked Environment.” *Journal of the ACM* 46 (5): 604–632.
- Knuth, Donald E. 1970. “Von Neumann’s First Computer Program.” *Computing Surveys* 2 (4): 247–261.
- Knuth, Donald E. 1972. “Ancient Babylonian Algorithms.” *Communications of the ACM* 15 (7): 671–677.
- Knuth, Donald E. 1997. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd ed. Reading, MA: Addison-Wesley.
- Knuth, Donald E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd ed. Reading, MA: Addison-Wesley.
- Knuth, Donald E. 2011. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Upper Saddle River, NJ: Addison-Wesley.
- Langville, Amy N., and Carl D. Meyer. 2006. *Google’s PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ: Princeton University Press.
- LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. “Deep Learning.” *Nature* 521 (7553): 436–444.

- Lewis, Harry R., and Christos H. Papadimitriou. 1998. *Elements of the Theory of Computation*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.
- McCabe, John. 1965. "On Serial Files with Relocatable Records." *Operations Research* 13 (4): 609–618.
- McCulloch, Warren S., and Walter Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* 5 (4): 115–133.
- Merton, Robert K. 1968. "The Matthew Effect in Science." *Science* 159 (3810): 56–63.
- Minsky, Marvin, and Seymour Papert. 1969. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press.
- Misa, Thomas J., and Philip L. Frana. 2010. "An Interview with Edsger W. Dijkstra." *Communications of the ACM* 53 (8): 41–47.
- Mitzenmacher, Michael, and Eli Upfal. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. 2nd ed. Cambridge: Cambridge University Press.
- Parker, Matt. 2014. *Things to Make and Do in the Fourth Dimension: A Mathematician's Journey through Narcissistic Numbers, Optimal Dating Algorithms, at Least Two Kinds of Infinity, and More*. London: Penguin Books.
- Pattis, Richard E. 1988. "Textbook Errors in Binary Searching." *SIGCSE Bulletin* 20 (1): 190–194.
- Pevzner, Pavel A., Haixu Tang, and Michael S. Waterman. 2001. "An Eulerian Path Approach to DNA Fragment Assembly." *Proceedings of the National Academy of Sciences* 98 (17): 9748–9753.
- Pinker, Steven. 2018. *Enlightenment Now: The Case for Reason, Science, Humanism, and Progress*. New York: Viking Press.
- Rivest, Ronald. 1976. "On Self-Organizing Sequential Search Heuristics." *Communications of the ACM* 19 (2): 63–67.
- Rosenblatt, Frank. 1957. "The Perceptron: A Perceiving and Recognizing Automaton." Report 85–460–1. Cornell Aeronautical Laboratory.
- Rumelhart, David E., Geoff rey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088): 533–536.
- Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, et al. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature* 529 (7587): 484–489.

- Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, et al. 2017. “Mastering the Game of Go without Human Knowledge.” *Nature* 550 (7676): 354–359.
- Strogatz, Steven. 2012. *The Joy of x : A Guided Tour of Math, from One to Infinity*. New York: Houghton Mifflin Harcourt.
- Taleb, Nassim Nicholas. 2007. *The Black Swan: The Impact of the Highly Improbable*. New York: Random House.
- Toussaint, Godfried T. 2005. “The Euclidean Algorithm Generates Traditional Musical Rhythms.” In *Renaissance Banff: Mathematics, Music, Art, Culture*, edited by Reza Sarhangi and Robert V. Moody, 47–56. Winfield, KS: Bridges Conference, Southwestern College.
- Toussaint, Godfried T. 2013. *The Geometry of Musical Rhythm: What Makes a “Good” Rhythm Good?* Boca Raton, FL: CRC Press.
- Turing, Alan M. 1937. “On Computable Numbers, with an Application to the Entscheidungsproblem.” *Proceedings of the London Mathematical Society* S2–42:230–265.
- Turing, Alan M. 1938. “On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction.” *Proceedings of the London Mathematical Society* S2–43:544–546.
- Tyson, Neil deGrasse, Michael Abram Strauss, and Richard J. Gott. 2016. *Welcome to the Universe: An Astrophysical Tour*. Princeton, NJ: Princeton University Press.
- West, Geoffrey. 2017. *Scale: The Universal Laws of Life, Growth, and Death in Organisms, Cities, and Companies*. London: Weidenfeld and Nicholson.
- Xiao, Han, Kashif Rasul, and Roland Vollgraf. 2017. “Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms.” August 28. arxiv.org/abs/1708.07747.

ДЛЯ ДАЛЬНЕЙШЕГО ЧТЕНИЯ

- Broussard, Meredith. 2018. *Artificial Unintelligence: How Computers Misunderstand the World*. Cambridge, MA: MIT Press.
- Christian, Brian, and Tom Griffiths. 2016. *Algorithms to Live By: The Computer Science of Human Decisions*. New York: Henry Holt and Company.
- Cormen, Thomas H. 2013. *Algorithms Unlocked*. Cambridge, MA: MIT Press.
- Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. 3rd ed. Cambridge, MA: MIT Press.
- Denning, Peter J., and Matti Tedre. 2019. *Computational Thinking*. Cambridge, MA: MIT Press.
- Dewdney, A. K. 1993. *The (New) Turing Omnibus: 66 Excursions in Computer Science*. New York: W. H. Freeman and Company.
- Dyson, George. 2012. *Turing's Cathedral: The Origins of the Digital Universe*. New York: Vintage Books.
- Erwig, Martin. 2017. *Once upon an Algorithm: How Stories Explain Computing*. Cambridge, MA: MIT Press.
- Fry, Hannah. 2018. *Hello World: How to Be Human in the Age of the Machine*. London: Doubleday.
- Harel, David, and Yishai Feldman. 2004. *Algorithmics: The Spirit of Computing*. 3rd ed. Harlow, UK: Addison-Wesley.
- Louridas, Panos. 2017. *Real-World Algorithms: A Beginner's Guide*. Cambridge, MA: MIT Press.
- MacCormick, John. 2013. *Nine Algorithms That Changed the Future: The Ingenious Ideas That Drive Today's Computers*. Princeton, NJ: Princeton University Press.
- O'Neil, Cathy. 2016. *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. New York: Crown Publishing Group.
- Petzold, Charles. 2008. *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine*. Indianapolis: Wiley Publishing.
- Sedgewick, Robert, and Kevin Wayne. 2017. *Computer Science: An Interdisciplinary Approach*. Boston: Addison-Wesley.

ОБ АВТОРЕ

Панос Луридаc — доцент на кафедре Научных и технических методов управления Афинского университета экономики и бизнеса. Он работает над алгоритмическими приложениями, проектированием программного обеспечения, безопасностью, практической криптографией и прикладным машинным обучением. Он автор книги *Real-World Algorithms: A Beginner's Guide*, опубликованной издательством MIT Press, и уже больше четверти века активно занимается программированием.

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Научно-популярное издание

БИБЛИОТЕКА MIT

Луридас Панос

АЛГОРИТМЫ

САМЫЙ КРАТКИЙ И ПОНЯТНЫЙ КУРС

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Ответственный редактор *Е. Истомина*
Литературный редактор *Р. Болдино*
Младший редактор *А. Захарова*
Художественный редактор *К. Доброслов*
Компьютерная верстка *Э. Брегис*
Корректоры *А. Баскакова, Л. Макарова*

Страна происхождения: Российская Федерация
Шығарылған елі: Ресей Федерациясы

ООО «Издательство «Эксмо»
123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб. 2013.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндiрушi: «ЭКМО» АҚБ Баспасы,
123308, Ресей, қала Мәскеу, Зорге көшесi, 1 үй, 1 ғимарат, 20 қабат, офис 2013 ж.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Тауар белгiсi: «Эксмо»
Интернет-магазин : www.book24.ru
Интернет-магазин : www.book24.kz
Интернет-дүкен : www.book24.kz
Импортер в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибутор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибутор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3-а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92. 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 на сайте Издательства «Эксмо»
www.eksmo.ru/certification
Өндiрген мемлекет: Ресей. Сертификация қарастырылмаған

Дата изготовления / Подписано в печать 17.02.2022.
Формат 70x100^{1/16}. Печать офсетная. Усл. печ. л. 15,56.
Тираж экз. Заказ

ПРИСОЕДИНЯЙТЕСЬ К НАМ!

БОМБОРА
ИЗДАТЕЛЬСТВО

БОМБОРА – лидер 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.

Мы в СОЦСЕТЯХ:

 [bomborabooks](https://www.instagram.com/bomborabooks)  [bombora](https://www.facebook.com/bombora)
bombora.ru



ISBN 978-5-04-115765-4



В электронном виде книги издательства вы можете
купить на www.litres.ru

ЛитРес:
один клик до книги



■ ■ ■ ЧИТАЙ · ГОРОД

book 24.ru

Официальный
интернет-магазин
издательской группы
«ЭКМО-АКТ»

ЛУЧШИЕ КНИГИ О БИЗНЕСЕ С ЛОГОТИПОМ ВАШЕЙ КОМПАНИИ? ЛЕГКО!

Удивить своих клиентов, бизнес-партнеров, сделать памятный подарок сотрудникам и рассказать о своей компании читателям бизнес-литературы? Приглашаем стать партнерами выпуска актуальных и популярных книг. О вашей компании узнает наиболее активная аудитория.

ПАРТНЕРСКИЕ ОПЦИИ:

- Специальный тираж уже существующих книг с логотипом вашей компании.
- Размещение логотипа на супер-обложке для малых тиражей (от 30 штук).
- Поддержка выхода новинки, которая ранее не была доступна читателям (50 книг в подарок).

ПАРТНЕРСКИЕ ВОЗМОЖНОСТИ:

- Рекламная полоса о вашей компании внутри книги.
- Вступительное слово в книге от первых лиц компании-партнера.
- Обращение первых лиц на суперобложке.
- Отзыв на обороте обложки вложение информационных материалов о вашей компании (закладки, листовки, мини-буклеты).



У вас есть возможность обсудить свои пожелания с менеджерами корпоративных продаж. Как?

Звоните:

+7 495 411 68 59, доб. 2261

Заходите на сайт:

eksmo.ru/b2b



ПАНОС ЛУРИДАС

компьютерные науки/математика

АЛГОРИТМЫ

Самый краткий и понятный курс

Эта книга поможет читателям не только полностью охватить тему алгоритмов, но и привьет **алгоритмический образ мышления**.

Вы поймете, **что такое алгоритмы**, в чем измеряется **их эффективность** и **в каких областях они применяются**.

В этой книге рассматриваются алгоритмы для решения задач, связанных с **графами, поиском, упорядочиванием элементов** и **ранжированием**.

Вы познакомитесь с **нейросетями** и **глубоким обучением**. А самое главное: для чтения этой книги достаточно лишь среднего школьного образования.

ISBN 978-5-04-115765-4



9 785041 157654 >

БОМБОРА
ИЗДАТЕЛЬСТВО

БОМБОРА – лидер 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 bombora.ru