ДЕВЯТЬ АЛГОРИТМОВ, КОТОРЫЕ ИЗМЕНИЛИ БУДУЩЕЕ

ОСТРОУМНЫЕ ИДЕИ, ЛЕЖАЩИЕ В ОСНОВЕ СОВРЕМЕННЫХ КОМПЬЮТЕРОВ

ДЖОН МАККОРМИК

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Джон Маккормик

ДЕВЯТЬ АЛГОРИТМОВ, которые изменили мир

Остроумные идеи, лежащие в основе современных компьютеров

Nine Algorithms That Changed the Future

The Ingenious Ideas That Drive Today's Computers

John MacCormick

with a foreword by Chris Bishop

PRINCETON UNIVERSITY PRESS
PRINCETON AND OXFORD

Девять алгоритмов, которые изменили мир

Остроумные идеи, лежащие в основе современных компьютеров

Джон Маккормик

с предисловием Криса Бишопа

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2014

УДК 004
ББК 32.97
М15

М15 Дж. Маккормик

Девять алгоритмов, которые изменили мир. Остроумные идеи, лежащие в основе современных компьютеров / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2014. – 236 с.: ил.

ISBN 978-5-94074-940-0

Ежедневно мы используем впечатляющие технологические достижения, даже не задумываясь об этом. Мы передаем по сети гигабайты информации, просматриваем тысячи документов в поисках необходимого, совершаем покупки в интернет-магазинах. Мы архивируем объемные материалы, так чтобы их можно было отправить по электронной почте, и пользуемся искусственным интеллектом компьютеров, которые автоматически исправляют опечатки в тексте, ретушируют фотографии и делают за нас многое другое...

Все это при нынешнем уровне развития технологий воспринимается как должное. Но ведь такие «чудеса» были бы невозможны без величайших идей информатики, родившихся в XX веке!

Эта книга – о том, как эти идеи зародились и как воплощались в жизнь.

Издание рассчитано на широкую аудиторию. Предварительного знакомства с информатикой от читателей не требуется.

УДК 004
ББК 32.97

ISBN 978-0-691-15819-8 (англ.)
ISBN 978-5-94074-940-0 (рус.)

Мир созрел до появления дешевых и в то же время сложных, и в высшей степени надежных устройств, – что-то должно случиться.

— Ванневар Буш
«*Возможный способ нашего мышления*», 1945

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Компьютеры преобразуют наше общество не в меньшей степени, чем достижения физики и химии в два предшествующих столетия. Действительно, вряд ли остался в нашей жизни уголок, который не затронут – а чаще изменен до неузнаваемости – цифровыми технологиями. И коль скоро компьютеры так важны для современного общества, не странно ли, что люди так мало знают об основополагающих идеях, благодаря которым все это стало возможно? Изучение этих идей составляет основу информатики, а новая книга Маккормика – одна из немногих попыток донести их до широкой аудитории.

Одна из причин недооценки информатики как научной дисциплины заключается в том, что ее редко преподают в средней школе. Если начальный курс по таким предметам, как физика и химия, принято считать обязательным, то информатику как отдельную дисциплину обычно изучают только в колледжах или университетах. А если уж в школе есть предмет «информатика» или «информационные и телекоммуникационные технологии», то сводится он к приобретению навыков работы с некоторыми программными пакетами. Неудивительно, что ученикам это кажется скучным, а их естественная склонность к использованию компьютеров для развлечений и общения умеряется впечатлением, будто созданию таких технологий недостает интеллектуальной глубины. Есть мнение, что именно такое умонастроение стало причиной снижения количества студентов, изучающих информатику в университетах, на 50 % за последние 10 лет. Учитывая критическое значение цифровых технологий для современного общества, крайне важно возродить интерес населения к обаянию информатики.

В 2008 году мне была оказана честь открывать 180 серию рождественских лекций Королевского института, начало которым положил Майкл Фарадей в 1826 году. Лекции 2008 года впервые были посвящены теме информатики. Готовясь к ним, я много размышлял о том, как объяснить смысл информатики непрофессиональной аудитории, и обнаружил, что существует очень мало ресурсов и почти никаких

научно-популярных книг на эту тему. Поэтому новая книга Маккормика пришлась очень кстати.

Маккормик проделал великолепную работу и сумел изложить сложные идеи информатики в виде, доступном для широкой публики. Многие из этих идей необычайно красивы и элегантны, что само по себе делает их достойными внимания. Приведу всего один пример: бурный рост Интернет-торговли стал реальностью только потому, что появилась возможность безопасно посылать по сети конфиденциальную информацию (в частности, номера кредитных карт). В течение многих десятилетий задача о безопасном обмене данными по «открытым» каналам считалась неразрешимой. А когда решение было найдено, оно оказалось исключительно элегантным, и Маккормик объясняет его с помощью точных аналогий, не требующих предварительного знакомства с информатикой. Благодаря таким жемчужинам книга займет достойное место на полке с научно-популярной литературой, и я ее всячески рекомендую.

Крис Бишоп

заслуженный научный работник, Microsoft Research, Кэмбридж
вице-президент, Королевский институт Великобритании
профессор информатики, Эдинбургский университет

ГЛАВА 1. Введение: необычные идеи, каждодневно используемые в компьютерах

Природа дарования, коим я обладаю, очень-очень проста. Ум мой от рождения предрасположен к фантазии, причудливо выражающейся в образах, фигурах, формах, предметах, понятиях, представлениях, порывах и отступлениях.

— Вильям Шекспир «*Бесплодные усилия любви*»[1]

Как появились на свет великие идеи информатики? Вот несколько примеров.

- В 1930-х годах, еще до создания первого цифрового компьютера, гениальный британский ученый открывает научную дисциплину информатики, а затем доказывает, что существуют задачи, которые не сможет решить ни один будущий компьютер, каким бы он ни был быстрым и мощным и как бы хитроумно ни был спроектирован.
- В 1948 году сотрудник телефонной компании публикует статью, которая заложила основы теории информации. Эта работа доказала, что компьютеры могут передавать сообщения без искажения, даже если большая часть данных изменена в результате воздействия помех.
- В 1956 году группа ученых приезжает на конференцию в Дартмуте с четко осознанной дерзкой идеей основать новую науку – искусственный интеллект. После многочисленных ярких достижений и не менее многочисленных глубочайших разочарований мы все еще ждем появления компьютерной программы, обладающей настоящим интеллектом.

[1] Перевод Ю. Корнеева.

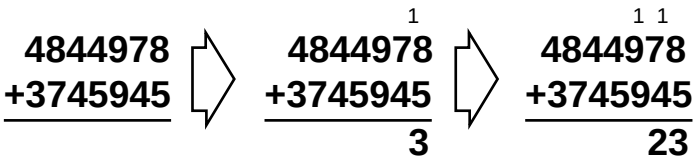
- В 1969 году исследователь из компании IBM открывает новый элегантный способ организовать информацию в базе данных. Сегодня эта технология применяется для хранения и извлечения информации, участвующей в большинстве сделок, совершаемых в онлайновом режиме.
- В 1974 году исследователи из финансируемой Британским правительством лаборатории по секретным коммуникациям открывают способ безопасного обмена данными между двумя компьютерами в ситуации, когда третий компьютер может наблюдать за всеми передаваемыми данными. Эти исследователи были связаны обязательством хранить государственную тайну, но по счастью три американских профессора независимо открыли и развили эту идею, которая ныне лежит в основе всех безопасных коммуникаций в Интернете.
- В 1996 году два студента Стэнфордского университета, работающих над диссертацией, решают объединить усилия и построить поисковую систему для веб. Спустя несколько лет они создали компанию Google – первый цифровой гигант эпохи Интернета.

Сегодня, в 21 веке мы наслаждаемся плодами поражающего воображение развития технологий, но никаких вычислительных устройств – будь то кластер из самых мощных доступных сегодня машин или модный гаджет последней модели – не существовало бы без основополагающих идей информатики, возникших в 20 столетии. Подумайте: делали ли вы сегодня что-то поразительное? Ответ, конечно, зависит от точки зрения. Быть может, вы перелопатили миллиарды документов в поисках двух-трех, которые содержат нужные сведения? Или сохранили либо передали по сети миллионы единиц информации без единой ошибки, несмотря на электромагнитные помехи, которым подвержены все электронные устройства? Или благополучно купили что-то через Интернет, хотя вместе с вами к тому же серверу обращались тысячи других пользователей? Или безопасно передали по проводам конфиденциальные данные (к примеру, номер своей кредитной карты), хотя вас могли подслушать десятки других компьютеров. А, быть может, вы воспользовались магией сжатия, позволяющий уменьшить размер многомегабайтной фотографии до величины, позволяющей передать ее по электронной почте? Или, даже не подозревая об этом, прибегли к скрытому в наладонном устройстве искусственному интеллекту, который исправляет опечатки, когда вы вводите текст с помощью его крохотной клавиатуры? Каждое из этих

деяний достойно изумления, и все они основаны на замечательных открытиях, о которых было сказано выше. Стало быть, любой пользователь компьютера каждый день по много раз применяет эти гениальные идеи, даже не осознавая этого! Цель этой книги – объяснить величайшие идеи информатики максимально широкой аудитории. Никакого предварительного знакомства с информатикой не предполагается.

Алгоритмы – чародейство услужливого джинна

До сих пор я говорил о великих «идеях» информатики, но сами ученые употребляют слово «алгоритм». Так в чем же разница между идеей и алгоритмом? Что вообще такое алгоритм? Простейший ответ звучит так: алгоритм – это рецепт, в котором описывается точная последовательность шагов, приводящая к решению задачи. Один такой алгоритм всем нам известен со школьной скамьи: сложение двух больших чисел. Пример его применения показан ниже. Описание последовательности шагов этого алгоритма начинается так: «Сначала сложить две последние цифры, записать последнюю цифру результата и перенести избыток в следующий столбец слева; затем сложить цифры в следующем столбце и прибавить к ним перенесенный из предыдущего столбца избыток...» – и так далее.



Первые два шага алгоритма сложения двух чисел.

Обратите внимание, что шаги этого алгоритма почти механические. И это одна из важнейших особенностей любого алгоритма: каждый шаг должен быть описан абсолютно точно, его выполнение не должно требовать от человека догадки или озарения. И тогда эти механические шаги можно будет оформить в виде программы для компьютера. Еще одна важная особенность алгоритма заключается в том, что он должен работать всегда, какие бы данные ни были поданы на вход. Алгоритм сложения, который мы учили в школе, обладает этим свойством: каковы бы ни были два складываемых числа, этот алго-

ритм рано или поздно даст правильный ответ. Так, его, безусловно, можно применить для сложения двух тысячезначных чисел, хотя на это и потребуется довольно много времени.

Возможно, у вас возник вопрос в связи с определением алгоритма как точного, механического рецепта. А настолько точным должен быть рецепт? Какие фундаментальные операции разрешены? Например, если речь идет об алгоритме сложения, то достаточно ли просто сказать «сложить две цифры» или нужно выписать полную таблицу сложения однозначных чисел? Такие детали могут показаться несущественными или даже излишне педантичными, но это представление очень далеко от истины: нахождение правильных ответов на такие вопросы как раз и составляет самую суть информатики, и здесь имеются связи с философией, физикой, неврологией и генетикой. Глубокие вопросы о том, что такое алгоритм на самом деле, в конечном счете сводятся к так называемому *тезису Чёрча-Тьюринга*. Мы вернемся к этой теме в главе 10, где будем обсуждать теоретические пределы вычислений и некоторые аспекты тезиса Чёрча-Тьюринга. А пока неформального представления об алгоритме как об очень точном рецепте нам будет вполне достаточно.

Итак, мы знаем, что такое алгоритм, но какое отношение это имеет к компьютерам? Суть дела в том, что для программирования компьютеров нужно формулировать инструкции очень точно. Прежде чем поручить компьютеру решение какой-то задачи, мы должны разработать соответствующий алгоритм. В других научных дисциплинах, например в математике и физике, важные результаты зачастую можно выразить одной формулой (из широко известных примеров можно назвать теорему Пифагора, $a^2 + b^2 = c^2$, или формулу Эйнштейна $E = mc^2$). Напротив, в информатике для формулирования важной идеи нужно описать, *как* решить задачу – разумеется, с применением какого-то алгоритма. Поэтому главная цель этой книги заключается в том, чтобы объяснить, что превращает ваш компьютер в личного джинна: великие алгоритмы, лежащие в основе его работы.

Какой алгоритм считать великим?

Это подводит нас к непростому вопросу: какие алгоритмы действительно «великие»? Перечень потенциальных кандидатов обширен, поэтому чтобы уложиться в объем книги, я применил несколько критериев отбора. Первый и самый главный критерий: алгоритм должен

ежедневно использоваться в обычном компьютере. Второй важный критерий: алгоритм должен решать конкретную реально возникающую задачу, например, сжатие файла или его передачу по зашумленному каналу. Для читателей, немного знакомых с информатикой, во врезке ниже описаны некоторые следствия, вытекающие из этих двух критериев.

Третий критерий заключается в том, что алгоритмы должны относиться преимущественно к *теоретической* информатике. Это исключает все связанное с компьютерным оборудованием: процессорами, мониторами, сетями и т. д. Кроме того, это означает, что устройству инфраструктуры, например Интернета, мы уделяем меньше внимания. Почему я решил сосредоточиться на теоретических основах информатики? Отчасти потому что существует перекос в общественном восприятии информатики: широко распространено мнение, будто информатика сводится в основном к программированию (то есть «софту») и конструированию оборудования («железа»). На самом же деле, многие из самых красивых идей информатики совершенно абстрактны и не попадают ни в одну из этих категорий. Ставя во главу угла теоретические идеи, я льщу себя надеждой, что больше людей придут к пониманию природы информатики как интеллектуальной дисциплины.

Вы, наверное, заметили, что мои критерии призваны отсеять потенциально великие алгоритмы, но я не ответил на гораздо более трудный вопрос: чем вообще определяется величие? Тут я решил положиться на собственную интуицию. В основе любого рассматриваемого в этой книге алгоритма лежит некий остроумный трюк, благодаря которому все и работает. Именно желание воскликнуть «ага!», когда трюк становится понятен, и доставляло мне радость, когда я объяснял алгоритм, и, надеюсь, доставит вам не меньшую радость, когда вы будете это объяснение читать. Поскольку я часто буду употреблять слово «трюк», то хочу сразу сказать, что не имею в виду ничего постыдного или мошеннического – такого, что вытворяют шулеры за карточным столом. Скорее, речь идет о профессиональных приемах или трюках фокусника: хитроумной технике, позволяющей достичь цели, когда другие способы затруднительны или даже невозможны.

Первый критерий – повседневное использование обычным пользователем компьютера – исключает алгоритмы, применяемые в основном профессионалами, например, компиляторы и способы верификации программ. Второй критерий – применение к конкретной задаче – исключает многие великие алгоритмы, изучаемые на млад-

ших курсах факультетов информатики. К ним относятся алгоритмы сортировки, например быстрая сортировка, алгоритмы на графах, в том числе алгоритм Дейкстры для поиска кратчайшего пути, и различные структуры данных, например хэш-таблицы. Нет сомнения, что это великие алгоритмы и что они отвечают первому критерию, поскольку активно используются в большинстве программ, запускаемых обычными пользователями. Но это алгоритмы общие: их можно применить к самым разным задачам. А в этой книге я решил сосредоточиться на алгоритмах решения конкретных задач, так как считаю, что это будет интереснее неискушенным пользователям.

Дополнительные сведения о принципах отбора алгоритмов для этой книги. Вообще говоря, не предполагается, что читатели знакомы с информатикой. Но если у вас есть познания в этой области, то из этой врезки станет ясно, почему многие из ваших любимых алгоритмов в книгу не попали.

Итак, я положился на собственную интуицию при выборе того, что мне кажется самыми остроумными трюками в мире информатики. Английский математик Г. Х. Харди в своей книге «Апология математика», пытаясь объяснить публике, почему математики занимаются своей наукой, выразил эту мысль такими словами: «Красота служит первым критерием: в мире нет места безобразной математике». Тот же критерий красоты применим и к теоретическим основам информатики. Таким образом, последний критерий отбора алгоритмов для этой книги – критерий красоты Харди. Надеюсь, что мне хотя бы отчасти удалось передать то ощущение красоты, который я испытывал, объясняя вошедшие в книгу алгоритмы.

Но перейдем к отобранным мной алгоритмам. Всепроникающие поисковые системы – пожалуй, самый наглядный пример алгоритмической технологии, которая оказывает влияние на всех пользователей компьютеров, поэтому неудивительно, что я включил несколько базовых алгоритмов поиска в Интернете. В главе 2 описывается, как поисковые системы используют *индексирование*, чтобы находить документы, отвечающие запросу, а в главе 3 объясняется алгоритм *PageRank*, который применялся в первой версии Google для того, чтобы поместить самые релевантные документы в начало списка результатов поиска.

Даже не задумываясь надолго над этим вопросом, большинство из нас *нутром чуют*, что своими поистине чудесными результатами поисковые системы обязаны каким-то глубоким научным идеям. Но есть и другие великие алгоритмы, которые часто вызываются, хотя пользователь об этом даже не подозревает. Примером такого алгоритма является криптография с открытым ключом, которой посвящена

глава 4. Всякий раз, заходя на защищенный сайт (адрес которого начинается с `https`, а не с `http`), вы пользуетесь так называемым алгоритмом *обмена ключами*, чтобы организовать защищенный сеанс. А это как раз часть криптографии с открытым ключом. В главе 4 объясняется, как устроен этот обмен ключами.

В главе 5 рассматриваются коды, исправляющие ошибки, – еще один класс алгоритмов, которыми мы пользуемся, даже не задумываясь об этом. На самом деле, если попытаться назвать какую-то одну великую идею, которая используется чаще всего, то это, пожалуй, будут коды, исправляющие ошибки. Благодаря им компьютер может обнаруживать *и исправлять* ошибки при хранении или передаче данных, не прибегая к резервному копированию или повторной передаче. Эти коды присутствуют всюду: во всех жестких дисках, во многих механизмах передачи по сети, на CD и DVD и даже в некоторых устройствах памяти. Но они так хорошо справляются со своей работой, что мы о них и не подозреваем.

Глава 6 выбивается из общего ряда. В ней обсуждаются алгоритмы распознавания образов, которые вкрались в список великих идей информатики, хотя и нарушают первый критерий: повседневное применение обычными пользователями. Распознавание образов – это целый класс алгоритмов, с помощью которых компьютеры выделяют значимые признаки из информации, характеризующейся высокой степень изменчивости: рукописный текст, речь, лица людей. На самом деле, в первой декаде 21 века в повседневных компьютерных приложениях эти методы не использовались. Но в 2011 году, когда я пишу эти слова, важность распознавания образов стремительно возрастает: в мобильных устройствах с миниатюрными экранными клавиатурами необходимо автоматически исправлять опечатки, планшеты должны распознавать рукописный ввод, и все больше подобных устройств (особенно смартфонов) управляются голосом. На некоторых вебсайтах распознавание образов даже применяется, чтобы показывать пользователю интересную ему рекламу. Ко всему прочему, я питаю особую склонность к распознаванию образов, потому что сам работаю в этой области. В силу указанных причин в главе 6 рассматриваются три наиболее интересных и успешно применяемых алгоритма распознавания образов: классификаторы по ближайшим соседям, деревья решений и нейронные сети.

Алгоритмы сжатия – тема главы 5 – еще один кластер великих идей, благодаря которым компьютер превращается в услужливого джинна. Иногда пользователи применяют сжатие самостоятельно,

например, чтобы сэкономить место на диске или уменьшить размер фотографии перед отправкой ее по электронной почте. Но гораздо чаще сжатие используется подспудно: хотя мы этого не знаем, скачиваемые или закачиваемые данные сжимаются для экономии полосы пропускания, а центры обработки данных нередко сжимают пользовательские данные для снижения затрат. Те 5 ГБ, которые почтовый провайдер предоставляет вам бесплатно, на его диске, скорее всего, занимают куда меньше места!

В главе 8 мы рассмотрим кое-какие фундаментальные алгоритмы, лежащие в основе работы баз данных. Мы уделим особое внимание хитроумным методам обеспечения *непротиворечивости* данных, хранящихся в базе. Без них наша жизнь в онлайне (в том числе покупки в Интернет-магазинах и общение в социальных сетях типа Facebook) погрязла бы в нагромождении машинных ошибок. В этой главе я объясню, в чем состоит проблема непротиворечивости и как ученые решают ее, не жертвуя феноменальной эффективностью, которой мы ожидаем от онлайновых систем.

В главе 9 речь пойдет об одной из неоспоримых жемчужин теоретической информатики: цифровой подписи. На первый взгляд кажется, что «подписать» электронный документ цифровым способом невозможно. Ну, действительно – такая подпись по определению состоит из цифровой информации, которую легко может скопировать любой желающий подделать подпись. Разрешение этого парадокса стало одним из самых значительных достижений информатики.

В главе 10 мы повернем совсем в другую сторону: вместо того чтобы описывать уже существующий великий алгоритм, мы поговорим об алгоритме, который *был бы* великим, если бы существовал. Как это ни удивительно, мы обнаружим, что этот конкретный великий алгоритм невозможен. Это полагает некие абсолютные пределы умению компьютеров решать задачи, и мы вкратце обсудим следствия этого результата для философии и биологии.

В заключение мы сведем воедино некоторые общие черты великих алгоритмов и поразмышляем о возможном будущем. Существуют ли еще какие-нибудь великие алгоритмы или мы уже открыли все?

Сейчас удобный момент упомянуть об одной особенности стиля этой книги. В научной литературе принято указывать источники, но изобилие цитат нарушило бы ритм текста и придало бы ему излишнюю академичность. Поскольку понятность и удобочитаемость стояли для меня на первом месте, я не стал включать цитаты в основной текст. Однако же все источники скрупулезно перечислены – зачас-

тую даже с дополнительными замечаниями – в разделе «Источники и литература для дальнейшего чтения» в конце книги. Там интересующиеся читатели найдут ссылки на дополнительные материалы, из которых смогут узнать гораздо больше о великих алгоритмах информатики.

И раз уж зашла речь об особенностях, упомяну заодно о небольшой поэтической вольности в названии книги. Наши «Девять алгоритмов, которые изменили мир» без сомнения революционны, но действительно ли их девять? Это вопрос спорный, ответ на него зависит от того, что считать отдельным алгоритмом. Посмотрим, откуда взялось число «девять». Если исключить введение и послесловие, то в книге девять глав, и в каждой обсуждаются алгоритмы, оказавшие революционное влияние на различные типы вычислительных задач: криптографию, сжатие, распознавание образов и т. д. Таким образом, «девять алгоритмов» – это в действительности девять классов алгоритмов для решения девяти разных проблем.

А какое нам, собственно, дело до великих алгоритмов?

Хочется надеяться, что этот краткий обзор пленительных идей заронил в вас желание копнуть глубже и понять, как они все-таки работают. Но, возможно, вы все еще недоумеваете: какова же конечная цель? Позвольте мне сказать несколько слов об истинном назначении этой книги. Определенно, это не техническое руководство. Прочитав книгу, вы не станете специалистом по компьютерной безопасности, искусственному интеллекту или еще какой-то области информатики. Конечно, кое-какие практически полезные знания вы приобретете. Например, вы будете знать, как проверить атрибуты «безопасных» веб-сайтов и «подписанных» программных пакетов. Вы также сможете осознанно решать, какой алгоритм сжатия – с потерей или без потери информации – лучше подходит для конкретной задачи. И, возможно, понимая некоторые особенности индексирования и ранжирования страниц, вы станете более эффективно пользоваться поисковыми системами.

Но это все мелочи по сравнению с главной целью книги. Прочитав ее, вы *не* станете гораздо более опытным пользователем компьютера. Но вы сумеете по-настоящему оценить красоту идей, с которыми имеете дело каждый день, пользуясь своими компьютерными устройствами.

Почему это хорошо? Попробую прибегнуть к аналогии. Я, безусловно, не специалист по астрономии – больше того, я почти полный невежда в этой области и хотел бы знать больше. Но всякий раз как я гляжу на ночное небо, те слабые познания в астрономии, которые у меня все-таки есть, усиливают наслаждение от этого времяпрепровождения. Почему-то понимание того, что я вижу, вселяет ощущение изумления и благоговения. Я очень надеюсь, что, прочитав эту книгу, вы хотя бы иногда будете испытывать то же чувство изумления и благоговения при работе с компьютером. Вы научитесь по достоинству ценить самый вездесущий и непроницаемый черный ящик всех времен: свой персональный компьютер, своего услужливого джинна.

ГЛАВА 2. Индексирование в поисковых системах: поиск иголки в самом большом в мире стоге сена

Ну, Гек, с того места, где ты стоишь, можно удочкой достать до входа. А попробуй-ка разыскать его.

— Марк Твен «*Приключения Тома Сойера*»

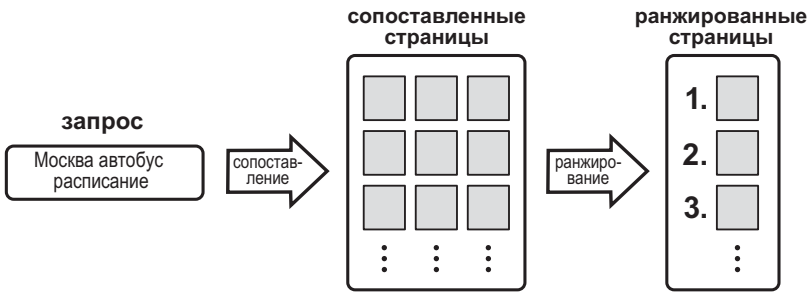
Поисковые системы оказывают колоссальное влияние на нашу жизнь. Но обращаясь к ним с запросами много раз на дню, мы редко задумываемся над тем, как этот замечательный инструмент работает. Огромный объем доступной информации, скорость поиска и качество его результатов кажутся настолько естественными, что мы раздражаемся, когда ответ не приходит за одну-две секунды. Мы как-то забываем, что каждый успешный поиск находит иголку в самом большом в мире стоге сена: Всемирной паутине.

Но в действительности, достойный восхищения сервис поисковых систем объясняется не только тем, что на проблему брошены гигантские технические ресурсы. Да, это правда, что любая компания, владеющая крупной поисковой системой, располагает разбросанной по всему миру сетью мощнейших центров обработки данных, в которых стоят тысячи серверов и самое передовое сетевое оборудование. Но вся эта груда железа была бы бесполезна без умных алгоритмов, с помощью которых производится организация информации и поиск в ней. В этой и следующей главе мы рассмотрим алгоритмические жемчужины, которые приводят в движение весь механизм всякий раз, как мы обращаемся с запросом. Как мы скоро увидим, среди задач, решаемых поисковой системой, две занимают особое место: *сопоставле-*

ние и *ранжирование*. В этой главе мы рассмотрим остроумную технику сопоставления: трюк с метасловами. А в следующей обратимся к задаче ранжирования и изучим знаменитый алгоритм PageRank, придуманный Google.

Сопоставление и ранжирование

Для начала будет полезно посмотреть, что вообще происходит, когда мы отправляем запрос на поиск в веб. Как уже было сказано, обработка запроса состоит из двух основных шагов: сопоставление и ранжирование. На практике оба шага для повышения эффективности объединяются в один. Но концептуально они различны, поэтому будем предполагать, что сопоставление завершается до начала ранжирования. На рисунке показана обработка запроса «Москва автобус расписание». На шаге сопоставления мы получаем ответ на вопрос «какие веб-страницы соответствуют моему запросу?» – в данном случае все страницы, на которых упоминаются расписания автобусов в Москве.



Два этапа поиска в веб: сопоставление и ранжирование. Результатом первого этапа (сопоставление) могут быть тысячи или миллионы страниц, которые необходимо отсортировать по релевантности на втором этапе (ранжирование).

Но в реальных поисковых системах многие запросы находят сотни, тысячи и даже миллионы подходящих страниц. А пользователи обычно предпочитают видеть не больше пяти, ну в крайнем случае десяти результатов. Поэтому система должна как-то отобрать лучшие результаты из очень большого множества. Хорошая поисковая система не только отберет несколько лучших результатов, но и покажет их в наиболее полезном порядке – сначала самая подходящая страница, затем подходящая чуть меньше и т. д.

Задача выбора и правильного упорядочения нескольких лучших результатов называется «ранжированием». Этот второй этап, следующий за сопоставлением, и является самым важным. В жестоком мире индустрии поиска выживают системы с лучшим механизмом ранжирования, остальные вымирают. В 2002 году каждая из трех крупнейших поисковых систем в США – Google, Yahoo и MSN – занимала примерно одинаковую долю рынка, чуть меньше 30 %. (MSN впоследствии поменяла название сначала на Live Search, а потом на Bing.) Но за несколько следующих лет Google резко увеличила свою долю, оставив Yahoo и MSN меньше, чем по 20 %. Принято считать, что феноменальный подъем Google на вершину индустрии поиска связан с изобретенными ей алгоритмами ранжирования. Не будет преувеличением сказать, что сама жизнь и смерть поисковых систем целиком зависят от алгоритмов ранжирования. Но эти алгоритмы мы будем обсуждать в следующей главе. А сейчас займемся этапом сопоставления.

AltaVista: первый алгоритм сопоставления масштаба веб

С чего начинается история алгоритмов сопоставления в поисковых системах? Очевидный – но неправильный – ответ: с Google, величайшего примера технологического успеха в 21 веке. И действительно, история Google, начавшаяся с диссертации двух аспирантов Стэнфордского университета, волнует и впечатляет. В 1998 году Ларри Пейдж и Сергей Брин собрали из разношерстных компьютерных компонентов поисковую систему нового типа. А спустя 10 лет их компания стала крупнейшим цифровым гигантом эпохи Интернета.

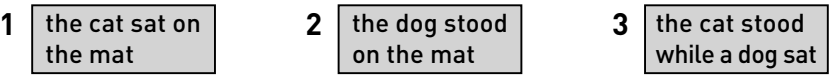
Но сама идея поиска в веб к тому моменту уже несколько лет как существовала. В числе первых коммерческих компаний были Infoseek и Lycos (та и другая запустили свои системы в 1994 году) и AltaVista, стартовавшая в 1995 году. В середине 1990-х годов AltaVista несколько лет считалась королем поисковых систем. Я тогда был аспирантом факультета информатики и ясно помню свое восхищение полнотой результатов поиска AltaVista. Впервые поисковая система полностью индексировала весь текст на каждой странице в веб и, более того, возвращала результаты чуть ли не мгновенно. Мы начнем знакомство с этим сенсационным технологическим достижением с древней (в самом буквальном смысле) идеи: индексирования.

Старое доброе индексирование

В основе любой поисковой системы лежит фундаментальная идея *индекса*. Но не создатели поисковых систем придумали индексы; на самом деле, идея стара, как сама письменность. Так, археологи нашли в древнем вавилонском храме возрастом 5000 лет библиотеку, в которой глиняные таблички были каталогизированы по темам. Так что у индексирования есть все основания претендовать на звание самой старой полезной идеи в информатике.

В наши дни словом «индекс» (по-русски «указатель») обычно называют раздел в конце книги. В нем в определенном порядке (обычно по алфавиту) перечислены все понятия, которые могли бы заинтересовать читателя, и против каждого понятия указаны места (обычно номера страниц), где это понятие встречается. Так, индекс в книге о животных мог бы содержать запись «гепард 124, 156», которая означает, что слово «гепард» встречается на страницах 124 и 156 (забавы ради можете поискать в индексе к этой книге слово «индекс» – попадете как раз на эту страницу).

В поисковой системе индекс выполняет те же функции, что в книге. «Страницами» теперь являются веб-страницы во Всемирной паутине, и поисковая система присваивает каждой веб-странице уникальный номер. (Да, страниц уйма – на момент последнего подсчета их количество исчислялось многими миллиардами – но компьютеры отлично справляются с большими числами.) Пример на рисунке проясняет ситуацию. Представьте, что во Всемирной паутине всего-то и есть что три таких коротких страницы, и им присвоены номера 1, 2 и 3.



Воображаемая Всемирная паутина, состоящая из трех страниц с номерами 1, 2, 3

a	**3**
cat	**1 3**
dog	**2 3**
mat	**1 2**
on	**1 2**
sat	**1 3**
stood	**2 3**
the	**1 2 3**
while	**3**

Простой индекс с номерами страниц

Чтобы построить индекс для этих трех веб-страниц, компьютер мог бы сначала составить список всех встречающихся на них слов, а затем отсортировать его по алфавиту. Назовем результат *списком слов* – в данном случае он будет выглядеть так: «a, cat, dog, mat, on, sat, stood, the, while». Затем компьютер перебрал бы все страницы и просмотрел бы все слова на каждой из них. Для каждого слова он записал бы номер текущей страницы против соответствующей позиции в списке слов. Окончательный результат показан на рисунке. Сразу видно, что слово «cat» встречается на страницах 1 и 3, но не на странице 2. А слово «while» есть только на странице 3.

Уже при таком элементарном подходе поисковая система может дать ответы на многие простые запросы. Пусть, например, вы вводите запрос `cat`. Поисковая система быстро найдет слово `cat` в списке слов. (Поскольку слова в списке расположены по алфавиту, компьютер может очень быстро найти нужное – точно так же, как человек находит слово в словаре.) А найдя слово `cat`, система может вернуть связанный с ним список страниц – в данном случае 1 и 3. Современные поисковые системы красиво форматируют результаты, включая в каждый небольшой фрагмент найденной страницы, но мы на такие детали не будем обращать внимания, а сосредоточимся на том, как система узнает, какие номера страниц соответствуют введенному запросу.

Рассмотрим еще один простой пример – порядок обработки запроса `dog`. В данном случае система быстро находит в списке слово `dog` и возвращает номера страниц 2 и 3. А что, если запрос содержит несколько слов, например: `cat dog`? Это означает, что мы хотим найти все страницы, содержащие оба слова «cat» и «dog». При наличии уже имеющегося индекса поисковая система легко справится и с этой задачей. Сначала она найдет каждое слово порознь и определит, какие им соответствуют страницы. Получится 1, 3 для «cat» и 2, 3 для «dog». Затем компьютер может просмотреть оба списка результатов и найти номера страниц, встречающиеся как в одном, так и в другом. В данном случае страницы 1 и 2 отбрасываются, но страница 3 присутствует в обоих списках, поэтому окончательный список результатов содержит только страницу 3. Аналогичная стратегия работает для запросов, в которых больше двух слов. Например, в ответ на запрос `cat the sat` будут возвращены страницы 1 и 3, поскольку они встречаются в каждом из трех списков для слов «cat» (1, 3), «the» (1, 2, 3) и «sat» (1, 3).

Пока что складывается впечатление, что построить поисковую систему довольно легко. Простейшая техника индексирования вроде бы

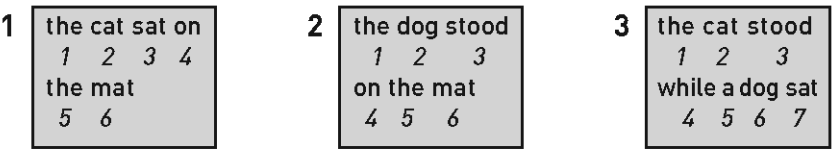
отлично работает, даже для запросов с несколькими словами. Увы, как выясняется, такой примитивный подход для современных поисковых систем совершенно не годится. Причин несколько, но мы рассмотрим только одну проблему: *фразовые запросы*. Фразовым называется запрос, в котором мы ищем точную фразу, не довольствуясь тем, что слова встречаются в разных местах страницы. В большинстве поисковых систем фразовые запросы заключаются в кавычки. Например, запрос `"cat sat"` по смыслу сильно отличается от запроса `cat sat`. По запросу `cat sat` ищутся страницы, на которых где-то встречаются оба слова «cat» и «sat» в любом порядке. Напротив, по запросу `"cat sat"` ищутся страницы, на которых сразу после слова «cat» идет слово «sat». В нашем случае в ответ на запрос `cat sat` возвращаются страницы 1 и 3, а на запрос `"cat sat"` – только страница 1.

Как поисковая система могла бы эффективно обработать фразовый запрос? Вернемся к примеру `"cat sat"`. Вроде бы на первом шаге можно было бы поступить так же, как при обработке запроса из двух слов `cat sat`: по списку слов построить список страниц, на которых встречается каждое слов; в данном случае мы получим 1, 3 для «cat» и точно такой же список – 1, 3 – для «sat». Но на этом поисковая система упирается в тупик. Она точно знает, что оба слова встречаются на страницах 1 и 3, но не может сказать, стоят ли они рядом и в нужном порядке. Можно было бы предположить, что в этот момент система просматривает исходные веб-страницы, чтобы понять, есть там нужная фраза или нет. Да, так поступить можно, но это было бы крайне неэффективно. Системе пришлось бы читать все содержимое каждой страницы, которая *могла бы* содержать фразу, а таких страниц может быть очень-очень много. Напомним, что в нашем примере всего-то три странички, а настоящая поисковая система должна правильно работать, когда страниц десятки миллиардов.

Трюк с позициями слов

Решение проблемы основано на первой из остроумных идей, благодаря которым современные поисковые системы так хорошо работают: в индексе нужно хранить не только *номера* страниц, но и *позиции* слов на страницах. Ничего таинственного в понятии позиции нет: это просто номер слова на странице. Так, позиция третьего слова равна 3, позиция двадцать девятого – 29 и т. д. На рисунке сверху показан весь наш набор данных из трех страницы, и для каждого слова указана его позиция. Ниже мы видим индекс, в котором хранятся номера страниц

и позиции слов. Такой способ построения индекса мы будем называть «трюком с позициями слов». Рассмотрим два примера, чтобы лучше понять, как этот трюк работает. Первая строка индекса имеет вид «a 3-5». Это означает, что слово «a» встречается в наборе данных ровно один раз – на странице 3 – и является пятым словом на этой странице. Самая длинная строка в индексе «the 1-1 1-5 2-1 2-5 3-1». Она говорит о позициях всех вхождений слова «the» в набор данных. Это слово дважды встречается на странице 1 (в позициях 1 и 5), дважды на странице 2 (в позициях 1 и 5) и один раз на странице 3 (в позиции 1).



a	3-5				
cat	1-2	3-2			
dog	2-2	3-6			
mat	1-6	2-6			
on	1-4	2-4			
sat	1-3	3-7			
stood	2-3	3-3			
the	1-1	1-5	2-1	2-5	3-1
while	3-4				

Вверху: наши три страницы с добавленными позициями слов
Внизу: новый индекс, включающий номера страниц и позиции слова на странице

Теперь вспомним, зачем нам понадобились позиции слов на странице: чтобы решить задачу об эффективной обработке фразовых запросов. Так посмотрим, как фразовый запрос обрабатывается с помощью нового индекса. Мы возьмем тот же запрос, что и раньше: `"cat sat"`. Первые шаги будет такими же, как для старого индекса: найти по индексу позиции отдельных слов; для «cat» мы получим 1-2, 3-2, а для «sat» – 1-3, 3-7. Пока все хорошо: мы знаем, что фразовому запросу `"cat sat"` могут удовлетворять только страницы 1 и 3. Но, как и раньше, мы еще не знаем, встречается точно такая фраза странице или нет, – может случиться так, что два слова по отдельности встречаются, но не рядом или не в том порядке. К счастью, это легко выяснить, зная позиции. Сначала займемся страницей 1. Из информации, хранящей-

ся в индексе, мы знаем, что слово «cat» встречается в позиции 2 на странице 1 (именно об этом говорит запись 1-2). Мы также знаем, что слово «sat» встречается в позиции 3 на странице 1 (запись 1-3). Но если «cat» находится в позиции 2, а «sat» в позиции 3, значит, «sat» следует сразу за «cat» (так как 3 – следующее целое число после 2). А это означает, что вся искомая фраза «cat sat» встречается на этой странице, начиная с позиции 2!

Я понимаю, что становлюсь занудливым, но мы рассматриваем этот пример так подробно, чтобы *точно* понять, какая информация используется для нахождения ответа. Обратите внимание – чтобы получить результат для фразового запроса `"cat sat"`, нам понадобилась лишь информация из индекса (1-2, 3-2 для слова «cat» и 1-3, 3-7 для слова «sat»), к исходным веб-страницам мы не обращались. Это очень важно, потому что мы должны только прочитать две записи из индекса, а не перелопачивать все страницы, которые потенциально могли бы подойти, – в настоящей поисковой системе таких страниц могут быть миллионы. Подведем итог: включение в индекс позиции слова на странице позволило обработать фразовый запрос, глядя только на две строки индекса, и обойтись без чтения большого числа веб-страниц. Этот простой трюк с позициями слов – одна из основных составляющих успешной работы поисковых систем!

На самом деле, мы еще не закончили рассматривать пример `"cat sat"`. Мы обработали только страницу 1, но не страницу 3. Однако рассуждение для страницы 3 аналогично: мы видим, что слово «cat» встречается в позиции 2, а слово «sat» – в позиции 7, поэтому они никак не могут быть рядом. Таким образом, страница 3 *не* удовлетворяет фразовому запросу `"cat sat"`, хотя и удовлетворяет запросу с несколькими словами `cat sat`.

Кстати говоря, трюк с позициями слов важен далеко не только для фразовых запросов. В качестве еще одного примера рассмотрим задачу поиска слов, расположенных близко друг к другу. В некоторых поисковых системах такой запрос формулируется с помощью ключевого слова `NEAR`. Система AltaVista предлагала эту возможность с самого начала и предлагает по сей день. Конкретно, предположим, что в некоей поисковой системе запрос `cat NEAR dog` ищет страницы, в которых между словами «cat» и «dog» находится не более пяти слов. Как можно эффективно выполнить этот запрос на нашем наборе данных? Имея позиции слов, легко! Слову «cat» в индексе соответствует запись 1-2, 3-2, а слову «dog» – запись 2-2, 3-6. Отсюда сразу видно, что страница 3 могла бы подойти. При этом на странице 3 «cat» встре-

чается в позиции 2, а «dog» – в позиции 6. Расстояние между этими словами равно 6 – 2 = 4. Следовательно, «cat» *действительно* отстоит от «dog» не более чем на пять слов, и страница 3 удовлетворяет запросу `cat NEAR dog`. Еще раз обращаю ваше внимание на эффективность выполнения запроса: нам не пришлось читать сами веб-страницы, хватило анализа двух записей индекса.

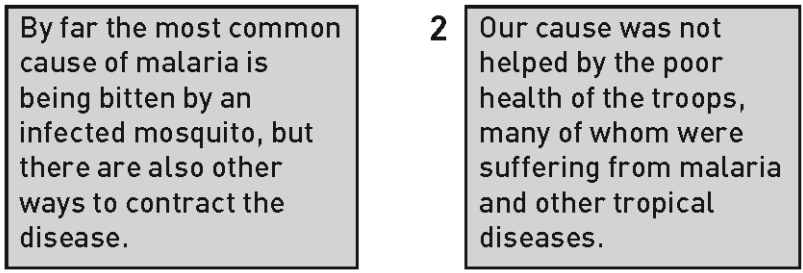
Но вообще-то запросы со словом `NEAR` не очень существенны для пользователей поисковых систем. Ими почти никто не пользуется, а большинство основных поисковых систем их даже не поддерживает. Тем не менее, возможность выполнять такие вопросы чрезвычайно важна, потому что поисковые системы постоянно делают это за кулисами. Чтобы понять, почему, нужно сначала познакомиться еще с одной из основных задач, стоящих перед современными поисковыми системами: *ранжированием*.

Ранжирование и близость

До сих пор мы занимались этапом сопоставления – задачей об эффективном поиске всех результатов, удовлетворяющих данному запросу. Но, как уже отмечалось, поисковая система не может считаться высококачественной, если плохо умеет ранжировать результаты, т. е. выбирать из них лучшие – достойные показа пользователю.

Рассмотрим идею ранжирования более пристально. От чего в действительности зависит «ранг» страницы? Правильный вопрос – не «*Соответствует* ли эта страница запросу?», а «*Релевантна* ли эта страница запросу?». Специалисты по информатике используют термин «релевантность», чтобы описать, насколько полезна данная страница применительно к конкретному запросу.

Для определенности предположим, что нас интересуют причины малярии, и мы вводим запрос `malaria cause`. Простоты ради допустим, что в ответ на этот запрос система возвращает всего две страницы, показанные на рисунке. Взгляните на них. Человеку с первого взгляда понятно, что на странице 1 речь действительно идет о причинах малярии, тогда как страница 2 – это описание какой-то военной кампании, в котором случайно встречаются слова «cause» и «malaria». Стало быть, страница 1, без сомнения, более «релевантна» запросу `malaria cause`, чем страница 2. Но компьютеры – не люди, им куда сложнее понять, о чем идет речь на этих страницах, так что на первый взгляд кажется, что поисковая система не сможет их правильно ранжировать.



also	1-19	
...		
cause	1-6	2-2
...		
malaria	1-8	2-19
...		
whom	2-15	

Вверху: две страницы, на которых упоминается малярия[1].
Внизу: часть индекса, построенного по этим страницам.

Однако в данном случае существует способ правильного ранжирования – и даже очень простой. Оказывается, что страницы, в которых встречающиеся в запросе слова расположены близко друг к другу, с большей вероятностью релевантны, чем страницы, где эти слова отстоят далеко. В нашем примере на странице 1 слова «malaria» и «cause» находятся друг от друга на расстоянии 2, а на странице 2 они разделены 17 словами. (И помните, что поисковая система может установить этот факт очень эффективно, глядя лишь на записи в индексе, а не на сами страницы.) Таким образом, хотя компьютер и не «понимает» смысла запроса, он может *догадаться*, что страница 1 более релевантна, чем страница 2, потому что интересующие пользователя слова на ней расположены гораздо ближе друг к другу.

Итак: хотя люди редко задают запросы со словом `NEAR`, сами поисковые системы постоянно пользуются информацией о близости для улучшения ранжирования, а эффективно делать это они могут исключительно благодаря трюку с позициями слов.

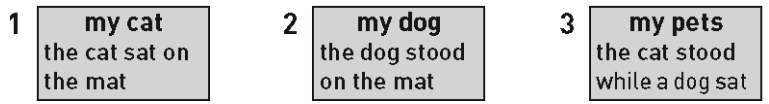
Мы уже знаем, что жители древнего Вавилона применяли индексирование за 5000 лет до появления поисковых систем. Но оказывает-

[1] Текст слева: «Наиболее распространенной причиной малярии является укус зараженного комара, но есть и другие способы подхватить эту болезнь». Текст справа: «Успеху нашей миссии не способствовало и плохое состояние здоровья солдат, многие из которых страдали от малярии и других тропических болезней». *Прим. перев.*

ся, что и трюк с позициями слов не был изобретен специально для поисковиков, это хорошо известная техника, которая использовалась в других информационно-поисковых системах задолго до пришествия Интернета. Зато в следующем разделе мы рассмотрим трюк, который был изобретен именно авторами поисковых систем: *трюк с метасловами*. Изобретательное использование этого трюка и связанных с ним идей позволили поисковой системе AltaVista вырваться на первое место в индустрии поиска в конце 1990-х годов.

Трюк с метасловами

До сих пор мы рассматривали совсем простые примеры веб-страниц. Вы, наверное, знаете, что у большинства страниц есть развитая структура: название, заголовки, ссылки, картинки, но мы пока считали их простым списков слов. А теперь поговорим о том, как поисковые системы учитывают структуру веб-страниц. Для простоты мы будем рассматривать только один вид структуры: наличие в начале страницы *названия*, за которым следует *тело*. На рисунке показаны уже знакомые нам три страницы, в каждую из которых добавлено название.



Пример набора веб-страниц, у каждой из которых есть название и тело.

Чтобы проанализировать структуру веб-страницы так, как это делают поисковые системы, нужно кое-что знать о том, как эти страницы создаются. Веб-страницы записываются на специальном языке, который позволяет браузеру аккуратно форматировать страницу (чаще всего для этой цели применяется язык HTML, но его детали для нас несущественны). Инструкции форматирования названий, заголовков, ссылок, картинок и других подобных элементов записываются с помощью специальных слов, которые называются *метасловами*. Например, в начале названия страницы могло бы находиться метаслово `<titleStart>`, а в конце – метаслово `<titleEnd>`. Аналогично тело страницы могло бы начинаться метасловом `<bodyStart>`, а заканчиваться метасловом `<bodyEnd>`. Пусть вас не смущают знаки «<» и «>». Они есть почти на всех компьютерных клавиатурах и часто именуются по своему смыслу в математике: «меньше» и «больше». Но в данном случае никакого отношения к математике они не имеют,

а употребляются просто как удобный способ отличить метаслова от обычных слов.

Тот же набор веб-страниц, что на предыдущем рисунке, но теперь показано, как их можно было бы записать с помощью метаслов, а не как они выглядят в окне браузера.

Взгляните на рисунок выше – содержимое страниц здесь в точности то же, что на предыдущем рисунке, но теперь мы видим, как страницы написаны, а не как они могли бы выглядеть в браузере. Большинство браузеров позволяют просмотреть исходный код веб-страницы, выбрав из меню пункт «Просмотр кода страницы» – попробуйте, когда будет возможность. (Кстати, упомянутые выше метаслова, например `<titleStart>` и `<titleEnd>`, выдуманы, просто чтобы было понятнее, о чем речь. В настоящем языке HTML метаслова называются *тегами*. Название страницы располагается между тегами `<title>` и `</title>` – поищите их, когда будете просматривать код страницы.)

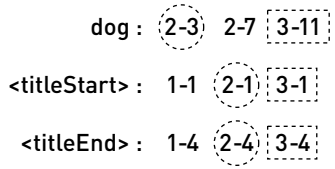
При построении индекса совсем нетрудно включить в него все метаслова. Никаких новых трюков не понадобится, нужно лишь сохранить позиции метаслов точно так же, как для обычных слов. На следующем рисунке показан индекс, построенный по трем страницам и включающий метаслова. Убедитесь, что ничего таинственного в этом индексе нет. Например, слову «mat» соответствует запись 1-11, 2-11, означающая, что это слово является одиннадцатым как на первой, так и на второй странице. С метасловами все обстоит точно так же: метаслову «`<titleEnd>`» соответствует запись 1-4, 2-4, 3-4, означающая, что оно является четвертым на страницах 1, 2 и 3.

Это простой трюк – индексирование метаслов так же, как обычных, – мы будем называть «трюком с метасловами». На первый взгляд, он прост, как правда, однако же играет важнейшую роль, позволяя поисковой системе точно выполнять поиск и качественно ранжировать результаты. Рассмотрим простой пример. Предположим, что поисковая машина поддерживает специальный тип запросов с ключевым словом `IN`: веб-страница удовлетворяет запросу `boat IN TITLE`, только если слово «boat» встречается в ее названии, а запрос `giraffe IN BODY` ищет те страницы, для которых тело содержит слово «giraffe».

Отметим, что в большинстве настоящих поисковых систем запросы со словом `IN` точно в таком виде не поддерживаются, но в некоторых того же эффекта можно добиться, перейдя в раздел «расширенный поиск», где можно указать, что слова запроса должны встречаться в названии или еще в какой-то части документа. Мы сделаем вид, что ключевое слово `IN` действительно существует, чтобы упростить объяснение. На самом деле, на момент написания этой книги Google позволял искать в названии документа с помощью ключевого слова `intitle:`, т. е. в Google запрос `intitle:boat` находит страницы, для которых слово «boat» встречается в названии. Попробуйте сами!

a	3-10
cat	1-3 1-7 3-7
dog	2-3 2-7 3-11
mat	1-11 2-11
my	1-2 2-2 3-2
on	1-9 2-9
pets	3-3
sat	1-8 3-12
stood	2-8 3-8
the	1-6 1-10 2-6 2-10
while	3-9
<bodyEnd>	1-12 2-12 3-13
<bodyStart>	1-5 2-5 3-5
<titleEnd>	1-4 2-4 3-4
<titleStart>	1-1 2-1 3-1

Индекс для веб-страниц, показанных на предыдущем рисунке, с включением метаслов.



Как поисковая система выполняет запрос `dog IN TITLE`.

Посмотрим, как поисковая система могла бы эффективно выполнить запрос `dog IN TITLE` на примере трех страниц, показанных на двух рисунках выше. Сначала она извлекает из индекса запись 2-3, 2-7, 3-11 соответствующую слову «dog». Затем (это несколько неожиданно, но пока просто поверьте мне) она извлекает записи для *обоих*

метаслов <titleStart> и <titleEnd>: соответственно 1-1, 2-1, 3-1 и 1-4, 2-4, 3-4. На рисунке выше показано, что извлечено к этому моменту – на кружочки и прямоугольники пока не обращайте внимания.

Далее поисковая система начинает просматривать запись индекса, соответствующую слову «dog»: для каждого результата проверяется, находится это слово в названии или нет. Первый результат для «dog» – обведенная кружочком пара 2-3 – соответствует третьему слову на странице 2. Просматривая запись для <titleStart>, поисковая система может узнать, где начинается название страницы 2, – это должна быть первая пара, начинающаяся с «2-». В данном случае просмотр завершается на обведенной кружочком паре 2-1, т. е. название страницы 2 начинается словом с номером 1. Точно так же система может узнать, где заканчивается название страницы 2. Для этого нужно просмотреть запись для слова <titleEnd> и найти пару, начинающуюся с «2-»; в данном случае это оказывается пара 2-4. Таким образом, название страницы 2 заканчивается словом с номером 4.

Все, что мы на данный момент знаем, сконцентрировано в обведенных кружочками элементах, которые говорят нам, что название страницы 2 начинается со слова 1 и заканчивается словом 4, а слово «dog» имеет номер 3. Последний шаг прост: так как 3 больше 1 и меньше 4, то мы с уверенностью можем сказать, что слово «dog» действительно встречается в названии, и, следовательно, страница 2 удовлетворяет запросу `dog IN TITLE`.

Теперь поисковая система может перейти к следующему потенциальному результату для слова «dog». Ему соответствует пара 2-7 (седьмое слово на странице 2), но поскольку мы уже знаем, что страница 2 удовлетворяет запросу, то эту пару можно пропустить и перейти к следующей – 3-11, которая обведена прямоугольником. Эта пара говорит, что «dog» – одиннадцатое слово на странице 3. Мы начинаем просмотр записей индекса, соответствующих <titleStart> и <titleEnd>, с пар, следующих за теми, что сейчас обведены кружочками, и ищем пары, начинающиеся с «3-». (Важно отметить, что возвращаться к началу каждой записи нет нужды – мы можем продолжить с того места, где остановились при анализе предыдущего результата.) В этом простом примере пара, начинающаяся с «3-», в обоих случаях оказывается следующей – 3-1 для <titleStart> и 3-4 для <titleEnd>. Обе они для наглядности обведены прямоугольниками. Вспомним, что наша задача – определить, находится слово «dog» в позиции 3-11 внутри названия или нет. Глядя на пары в прямоугольниках, мы видим, что на странице 3 «dog» – одиннадцатое

слово, а название это страницы начинается словом 1 и заканчивается словом 4. Поскольку 11 больше 4, значит, это вхождение слова «dog» оказывается за названием, а не внутри него; таким образом, страница 3 не удовлетворяет запросу `dog IN TITLE`.

Итак, трюк с метасловами позволяет поисковой системе весьма эффективно отвечать на запросы к структуре документа. В рассмотренном примере мы искали слова внутри названия страницы, но аналогичную технику можно применить к поиску в гиперссылках, описаниях картинок и других значимых частях веб-страницы. И на все такие запросы можно будет отвечать столь же эффективно. Как и в ранее обсуждавшихся примерах, поисковой системе не нужно заглядывать в сами веб-страницы, для ответа достаточно просмотреть небольшую часть индекса. И что не менее важно, каждую запись индекса необходимо просмотреть только *один раз*. Вспомните, что произошло, после того как мы закончили обработку первого результата на странице 2 и перешли к потенциальному результату на странице 3: вместо того чтобы возвращаться к началу записей для метаслов `<titleStart>` и `<titleEnd>`, поисковая система продолжила просмотр с того места, где остановилась. Это очень важно для повышения эффективности обработки запросов со словом `IN`.

Запросы к названию документа и другие «структурные запросы», ответ на которые зависит от *структуры* веб-страницы, аналогичны запросам со словом `NEAR` в том смысле, что люди их задают редко, зато сами поисковые системы выполняют их постоянно для внутренних надобностей. Причина та же, что и раньше: само существование поисковой системы зависит от качества ранжирования, а его можно значительно повысить, учитывая структуру страницы. Например, страницы, в которых слово «dog» встречается в заголовке, с гораздо большей вероятностью содержат информацию о собаках, чем страницы, в которых это слово упоминается только в теле страницы. Поэтому, когда пользователь вводит простой запрос `dog`, поисковая система может внутри себя выполнить запрос `dog IN TITLE` (хотя пользователь этого явно и не просил), чтобы найти страницы, *посвященные* собакам, а не просто такие, где собаки случайно упоминаются.

Трюки индексирования и сопоставления — это еще не все

Создание поисковой системы для веб – отнюдь не простая задача. Конечный продукт можно сравнить с невообразимо сложной маши-

ной, в которой множество разных колесиков, шестеренок и рычагов, и все они должны быть правильно подогнаны, чтобы машина работала. Поэтому важно понимать, что два представленных в этой главе трюка сами по себе не решают задачу построения эффективного поискового индекса. Однако они дают *почувствовать*, как создаются и используются индексы в реальной поисковой системе.

Трюк с метасловами позволил AltaVista добиться успеха там, где другие потерпели поражение, – эффективно производить поиск во всей сети веб. Мы знаем об этом, потому что этот трюк описан в заявке на патент США 1999 года, поданной компанией AltaVista и названной «Поиск с ограничениями по индексу». Однако замечательного алгоритма сопоставления, придуманного AltaVista, не хватило для удержания на плаву в бурных водах зарождающейся индустрии поиска. Как мы уже знаем, эффективное сопоставление – лишь половина успеха поисковой системы; вторая половина – задача о *ранжировании* подходящих страниц. И в следующей главе мы увидим, что появления нового типа алгоритмов ранжирования оказалось достаточно, чтобы затмить AltaVista и вывести Google на передний край технологии поиска в веб.

ГЛАВА 3. PageRank: технология, породившая Google

Компьютер в Стар Треке не представляет собой ничего особенного. Ему задают случайные вопросы, он некоторое время думает. Я полагаю, что мы сумеем сделать кое-что получше.

— Ларри Пейдж (сооснователь *Google*)

С точки зрения архитектуры, гараж – обычно весьма скромное строение. Но в Кремниевой долине гаражам отведена особая предпринимательская роль: многие из крупнейших обосновавшихся там технологических компаний родились или, по крайней мере, вынашивались в гаражах. Это началось не во время бума Интернет-компаний 1990-х годов. За 50 лет до того, в 1939 году, когда мировая экономика еще не успела оправиться от Великой депрессии, в гараже Дэйва Хьюлитта в Калифорнии зародилась компания Hewlett-Packard. Несколько десятилетий спустя, в 1976 году, Стив Джобс и Стив Возняк вышли в мир из гаража Джобса в Лос Альтосе, Калифорнии, основав ставшую теперь легендарной компанию Apple. (Хотя популярное предание гласит, что Apple была основана в гараже, на самом деле Джобс и Возняк поначалу работали в спальне. Позднее им перестало хватать места, и они перебрались в гараж.) Но истории успеха HP и Apple перебивает запуск поисковой системы Google, которая тоже обреталась в гараже в Менло-Парк, Калифорния, когда была зарегистрирована как компания в сентябре 1998.

К тому времени Google уже больше года эксплуатировала свою службу поиска в веб – сначала на серверах Стэнфордского университета, где писали диссертации оба ее сооснователя. И лишь когда требования к полосе пропускания стремительно набирающей популярность службы оказались чрезмерными для Стэнфорда, оба аспиранта, Ларри Пейдж и Сергей Брин, перенесли операции в ставший с тех

пор знаменитым гараж в Менло-Парк. Наверное, они что-то сделали очень правильно, потому что не прошло и трех месяцев с момента регистрации компании, как журнал *PC Magazine* включил Google в число 100 лучших веб-сайтов 1998 года.

Тут-то и начинается наша история: по словам *PC Magazine*, элитного статуса Google удостоилась за «непревзойденное мастерство в возврате удивительно релевантных результатов». Вспомните, в предыдущей главе было сказано, что первые коммерческие поисковые системы были запущены четырьмя года раньше, в 1994 году. Как же работающая в гараже Google смогла преодолеть колоссальное отставание в четыре года, одним скачком оставив позади уже популярные к тому времени Lycos и AltaVista в части качества поиска? Простого ответа на этот вопрос не существует. Но одним из самых важных факторов, особенно в те ранние дни, был новаторский алгоритм ранжирования результатов поиска, который известен под названием *PageRank*.

В названии «PageRank» есть игра слов: с одной стороны, это алгоритм ранжирования веб-страниц, а, с другой, – алгоритм Ларри Пейджа, его главного изобретателя. Пейдж и Брин опубликовали алгоритм в 1998 году в статье, представленной на научную конференцию: «The Anatomy of a Large-scale Hypertextual Web Search Engine[1]». Как следует из названия, статья посвящена не только алгоритму PageRank. На самом деле это полное описание системы Google в том виде, в каком она существовала в 1998 году. Но в трясине технических деталей скрывается описание, пожалуй, первой алгоритмической жемчужины, появившейся на свет в 21 веке: алгоритма PageRank. В этой главе мы рассмотрим, как и почему этому алгоритму удается находить иголки в стогах сена и неизменно помещать самые релевантные результаты поиска в начало списка.

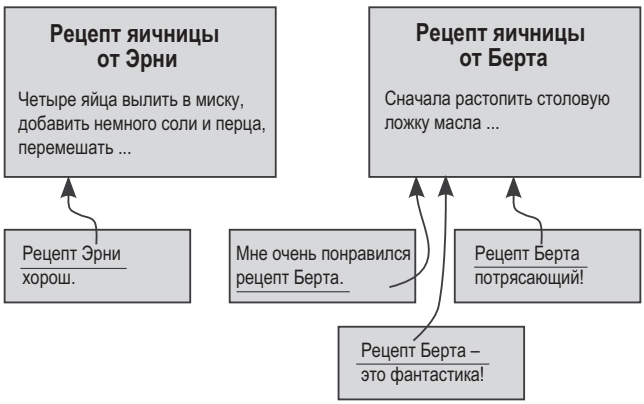
Трюк с гиперссылками

Вы, наверное, знаете, что такое гиперссылка: это фраза на веб-странице, щелчок по которой приводит к переходу на другую страницу. В большинстве браузеров гиперссылки отображаются синим цветом и подчеркиваются, чтобы они бросались в глаза.

Гиперссылки – на удивление старая идея. В 1945 году – примерно в то время, когда разрабатывались первые электронные компьютеры, – американский инженер Ванневар Буш опубликовал пророческое эссе под названием «Возможный способ нашего мышления». В этой рабо-

[1] Анатомия системы крупномасштабного гипертекстового Интернет-поиска.

те, охватывающий широкий круг вопросов, Буш описал целый спектр потенциальных новых технологий, в том числе гипотетическую машину, которую он назвал *memex*. Она могла хранить и автоматически индексировать документы, но этим отнюдь не ограничивалась. Машина позволяла осуществить «ассоциативное индексирование, ... посредством которого из любого документа можно мгновенно и автоматически перейти к другому» – иными словами, рудиментарная форма гиперссылки!



Смысл трюка с гиперссылками. В прямоугольниках показаны шесть веб-страниц. Две страницы содержат рецепты яичницы, а остальные четыре – гиперссылки на эти рецепты. Согласно трюку с гиперссылками, страница Берта имеет больший ранг, чем страница Эрни, потому что на первую указывают три ссылки, а на вторую – только одна.

Гиперссылки появились в 1945 году. Это одно из самых важных средств, применяемых поисковыми системами для ранжирования, и основа технологии Google PageRank, которой мы теперь и займемся по-настоящему.

Первое, что нужно понять, – простая идея, которую мы будем называть *трюком с гиперссылками*. Объяснить его проще всего на примере. Допустим, вы хотите узнать, как приготовить яичницу, и решаете поискать ответ в сети. Поиск в реальной сети даст миллионы результатов, но для простоты предположим, что найдено всего две страницы: «рецепт яичницы от Эрни» и «рецепт яичницы от Берта». На рисунке выше показаны как эти страницы, так и другие, ссылающиеся на первую или вторую. Продолжим идти по пути упрощения и предположим, что кроме этих четырех страниц, на наши рецепты приготовления яичницы в сети больше нет ссылок. На рисунке гиперссылки подчеркнуты, а стрелки показывают, куда ссылки ведут.

Вопрос: у какого из двух результатов должен быть более высокий ранг? Человеку не составит труда прочитать страницы, ссылающиеся на оба рецепта, и составить свое суждение. Похоже, что оба рецепта годятся, но народу рецепт Берта нравится гораздо больше, чем рецепт Эрни. Поэтому, если никакой другой информации нет, то, наверное, имеет смысл поставить Берта выше Эрни.

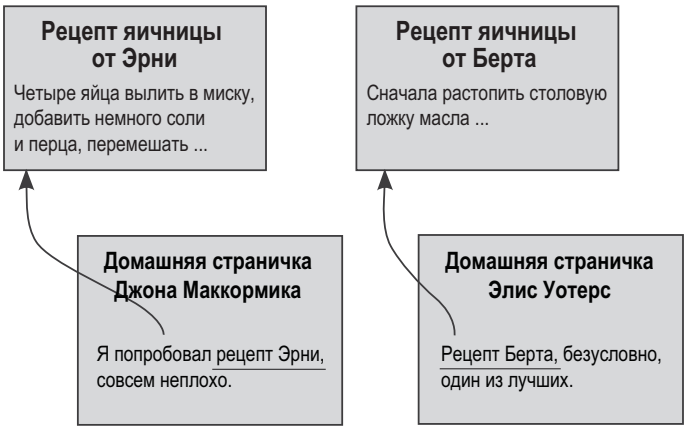
К сожалению, компьютеры не понимают смысла текста, поэтому поисковая система не может проанализировать четыре страницы, ссылающиеся на результаты, и оценить, насколько горячо рекомендуют каждый рецепт. С другой стороны, компьютеры отлично умеют считать. Поэтому напрашивается простое решение – *подсчитать* количество страниц, ссылающихся на каждый рецепт, – в данном случае одна для Эрни и три для Берта – и ранжировать рецепты по числу ведущих на них ссылок. Разумеется, этот подход гораздо менее точен, чем суждение человека, прочитавшего все страницы и присвоившего рецептам ранги вручную, но тем не менее эта техника полезна. Как выясняется, в отсутствие другой информации количество ссылок, ведущих на веб-страницу, – значимый показатель ее полезности или «авторитетности». В нашем примере Берт получает оценку 3, а Эрни – оценку 1, поэтому поисковая система ставит страницу Берта выше, чем страницу Эрни, в списке результатов поиска.

Вероятно, вы уже заметили кое-какие проблемы, свойственные этому «трюку с гиперссылками». Очевидный недостаток состоит в том, что иногда ссылки намеренно ведут не на хорошие страницы, а на *плохие*. Представьте, например, страницу, которая ссылается на рецепт Эрни со словами «Я попробовал рецепт Эрни – такая гадость». Подобные ссылки, где страница критикуется, а не рекомендуется, приводят к тому, что странице присваивается более высокий ранг, чем она заслуживает. Но, как показывает практика, ссылки все же чаще ставятся, чтобы порекомендовать, а не покритиковать страницу, поэтому описанный трюк остается полезным, несмотря на этот очевидный изъян.

Трюк с авторитетностью

Возможно, вы уже задались вопросом, почему все ведущие на страницу ссылки следует считать одинаково важными. Ведь ясно же, что рекомендация эксперта весомее рекомендации новичка. Чтобы разобраться в этом вопросе, возьмем тот же пример с яичницей, но другой набор ссылок. Новая ситуация изображена на следующем рисунке:

число ссылок на рецепты Берта и Эрни одинаково (по одной), но на страницу Эрни ссылается моя домашняя страничка, а на страницу Берта – знаменитый шеф-повар Элис Уотерс. Если никакой другой информации нет, чей рецепт вы предпочтете? Понятно, что лучше выбрать рецепт, который рекомендует известный повар, а не автор книги по информатике. Этот базовый принцип мы будем называть «трюком с авторитетностью»: ссылки с «авторитетных» страниц дают больший вклад в ранг, чем ссылки с малоавторитетных страниц.

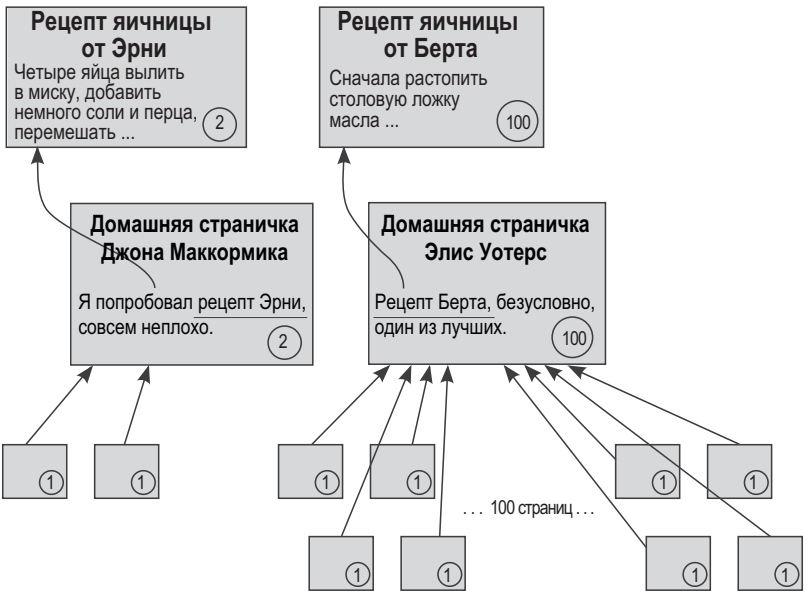


Смысл трюка с авторитетностью. Показаны четыре веб-страницы: два рецепта яичницы и две ссылки на эти рецепты. Одна из ссылок ведет со странички автора этой книги (который отнюдь не является известным поваром), другая – с домашней странички знаменитого шеф-повара Элис Уотерс. Благодаря трюку с авторитетностью страница Берта получает более высокий ранг, чем страница Эрни, потому что ссылка, ведущая на страницу Берта, «авторитнее», чем ссылка на страницу Эрни.

Принцип-то хорош, но в таком виде для поисковых систем бесполезен. Откуда компьютеру знать, что Элис Уотерс – больший авторитет по яичницам, чем я? Но есть идея: давайте объединим трюки с гиперссылками и авторитетностью. Первоначально каждой странице назначается авторитетность 1, но если на страницу ведут какие-нибудь ссылки, то ее авторитетность вычисляется путем суммирования авторитетностей всех ссылающихся на нее страниц. Другими словами, если страницы X и Y ссылаются на страницу Z, то авторитетность Z равна сумме авторитетностей X и Y.

На следующем рисунке приведен подробный пример вычисления авторитетности двух рецептов яичницы. Окончательные оценки обведены кружочками. На мою домашнюю страничку ссылаются две

страницы; на них никто не ссылается, поэтому авторитетность каждой из них равна 1. Авторитетность моей домашней странички равна сумме авторитетностей входящих ссылок, то есть 2. На домашнюю страничку Элис Уотерс ссылаются 100 страниц, авторитетность каждой из которых равна 1, поэтому итоговая авторитетность странички Элис равна 100. На рецепт Эрни ведет одна ссылка со страницы с авторитетностью 2, поэтому сумма авторитетностей всех входящих ссылок (их всего одна) равна 2. На страницу Берта тоже ведет одна ссылка, но ее авторитетность равна 100, так что итоговая авторитетность рецепта Берта – тоже 100. А коль скоро 100 больше 2, ранг страницы Берта оказывается выше.



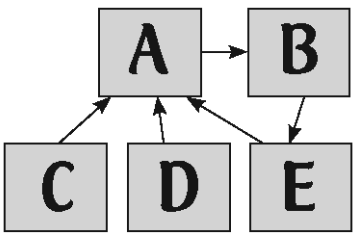
Простое вычисление «оценок авторитетности» для двух рецептов яичницы. Оценки обведены кружочками.

Трюк со случайным посетителем

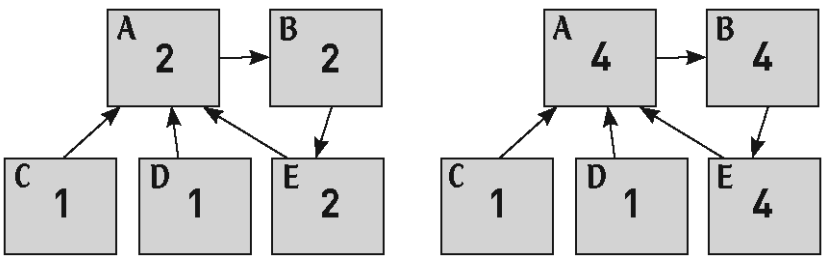
Похоже, мы нащупали стратегию автоматического вычисления авторитетности, которая не требует от компьютера понимать, что написано на странице. Увы, у этого решения есть серьезный недостаток. Может случиться, что гиперссылки образуют так называемый «цикл». Цикл возникает, если, переходя по гиперссылкам, можно вернуться в исходное место.

На следующем рисунке приведен пример. Мы видим 5 веб-страниц, обозначенных буквами *A*, *B*, *C*, *D*, *E*. Начав с *A*, мы можем перейти от *A* к *B*, затем от *B* к *E* и, наконец, от *E* к *A*, т. е. туда, откуда начали. Это означает, что страницы *A*, *B* и *E* образуют цикл.

Оказывается, что данное выше определение «оценки авторитетности» (полученное в результате комбинирования трюков с гиперссылками и авторитетностью) никуда не годится, если имеется цикл. Посмотрим, что происходит в этом примере. На страницы *C* и *D* нет ссылок, поэтому они получают оценку 1. Страницы *C* и *D* ссылаются на *A*, поэтому авторитетность *A* равна сумме авторитетностей *C* и *D*, то есть 2. Далее *B* получает оценку 2 от *A*, а *E* – оценку 2 от *B* (ситуация на этот момент показана на рисунке слева). Но теперь *A* отстала от жизни: она по-прежнему получает 1 от *C* и *D*, но еще и 2 от *E*, так что ее общая оценка равна 4. А теперь от жизни отстала *B*: она получает 4 от *A*. А дальше нужно обновить *E*, поскольку она получает 4 от *B* (новая ситуация изображена на рисунке справа). И так далее: оценка *A* равна 6, значит, оценка *B* равна 6, значит, оценка *E* равна 6, значит, оценка *A* равна 8 .... Ну, вы поняли. Мы должны бесконечно увеличивать оценки, снова и снова проходя по циклу.



Пример цикла гиперссылок.
Страницы A, B и E образуют цикл, потому что, начав с A, можно перейти к B, затем к E и вернуться в исходную точку A.



Проблема, вызванная циклами. Оценки авторитетности A, B и E растут бесконечно и никогда не становятся актуальными.

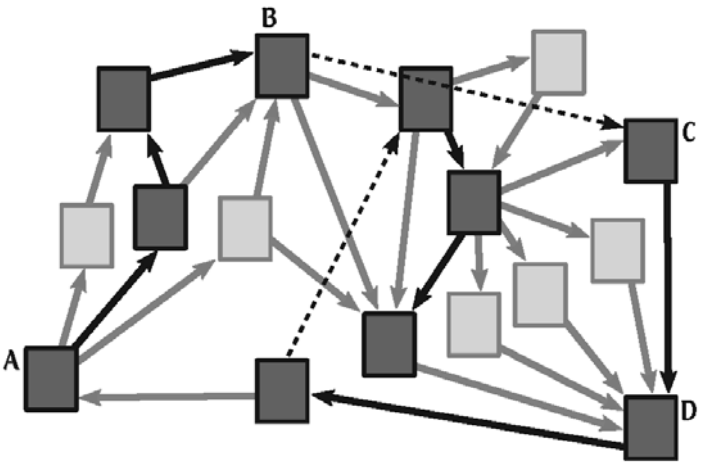
При таком вычислении оценок авторитетности возникает проблема яйца и курицы. Если бы мы знали истинную оценку авторитетности *A*, то могли бы вычислить оценки авторитетности *B* и *E*. А если бы мы знали истинные оценки авторитетности *B* и *E*, то могли бы вычислить оценку *A*. Но поскольку каждая оценка зависит от остальных, задача, похоже, неразрешима.

По счастью, решить проблему яйца и курицы можно с помощью приема, который мы будем называть *трюком со случайным посетителем*. Внимание: первоначальное описание этого трюка никак не соотносится с рассмотренными выше трюками с гиперссылками и авторитетностью. Но проанализировав лежащий в основе трюка механизм, мы обнаружим у него примечательные свойства: он сочетает желательные характеристики трюков с гиперссылками и авторитетностью, но работает, даже когда гиперссылки образуют цикл.

Чтобы разобраться в идее трюка, представим себе человека, который случайно блуждает в Интернете. Точнее, наш посетитель стартует с некоторой веб-страницы, случайно выбранной из всей Всемирной паутины. Затем из всех имеющихся на этой странице гиперссылок он случайно выбирает одну и щелкает по ней. Далее он случайно выбирает ссылку на новой странице. Этот процесс продолжается – на каждом шаге случайно выбирается ссылка на текущей странице и таким образом производится переход на следующую страницу. На рисунке ниже показан пример в предположении, что сеть состоит всего из 16 страниц. Прямоугольниками представлены веб-страницы, а стрелками – гиперссылки между ними. Четыре страницы помечены буквами. Страницы, на которые посетитель заходил, окрашены темным цветом, ссылки, по которым он переходил, – черные, а пунктирными стрелками обозначены случайные рестарты, о которых мы поговорим ниже.

Но в этом процессе есть одна тонкость: при посещении каждой страницы с фиксированной *вероятностью рестарта* (скажем, 15 %) посетитель *не станет* щелкать по имеющимся на странице гиперссылкам, а вместо этого начнет всю процедуру заново, случайно выбрав какую-то другую страницу из всей сети. Можете считать, что с вероятностью 15 % посетителю надоедает данная страница, и он решает последовать по новой цепочке ссылок. Взгляните на рисунок ниже. Наш посетитель начал со страницы *A* и проследовал по трем случайным гиперссылкам, потом на странице *B* ему стало скучно и он рестартовал, перейдя на страницу *C*. До следующего рестарта он проследовал еще по двум случайным ссылкам. (Кстати, во всех приме-

рах из этой главы случайные посетители рестартуют с вероятностью 15 %, именно такое значение выбрали сооснователи Google Пейдж и Брин в оригинальной статье, где описывался прототип их поисковой системы.)

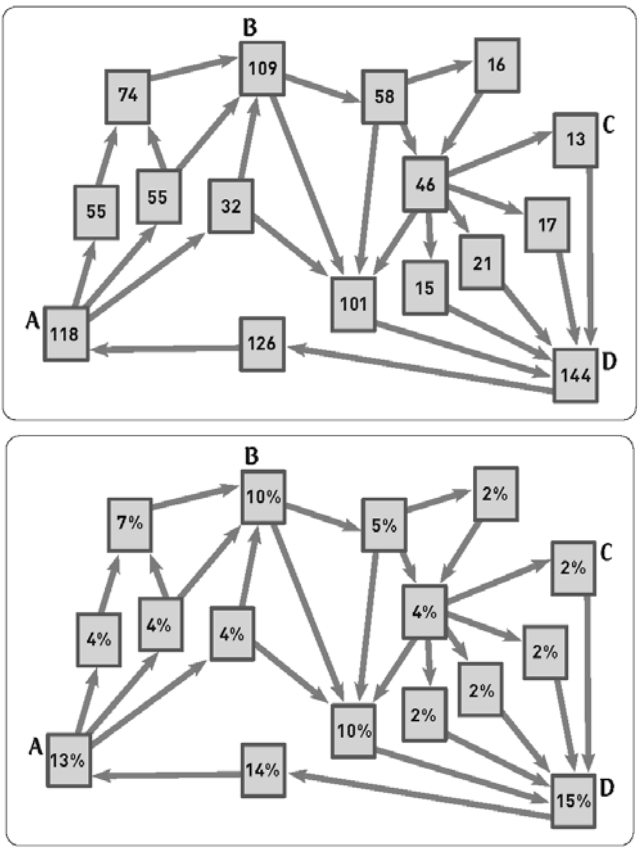


Модель случайного посетителя.
Страницы, на которые посетитель заходил, закрашены темным цветом, а пунктирными стрелками обозначены случайные рестарты. Тропа начинается на странице A и переходит по случайно выбранным гиперссылкам, прерываясь в двух местах случайными рестартами.

Этот процесс легко смоделировать на компьютере. Я написал такую программу и гонял ее, пока посетитель не побывал на 1000 страниц. (Разумеется, страницы необязательно должны быть различны. Несколько посещений одной и той же страницы тоже считаются, а в этом небольшом примере каждая страница посещалась много раз.) Результаты 1000 смоделированных посещений показаны на верхнем рисунке на следующей странице. Легко видеть, что чаще всего – 144 раза – посещалась страница *D*.

Как в опросах общественного мнения, мы можем повысить точность модели, увеличив объем случайной выборки. Я запустил программу еще раз и ждал, пока посетитель посетит миллион страниц (если вам интересно, на моем компьютере это заняло полсекунды!). При таком большом количестве посещений представлять результаты лучше в процентах. То, что получилось, показано на нижнем рисунке на следующей странице. Как и раньше, чаще всего посещалась страница *D* – на нее пришлось 15 % посещений.

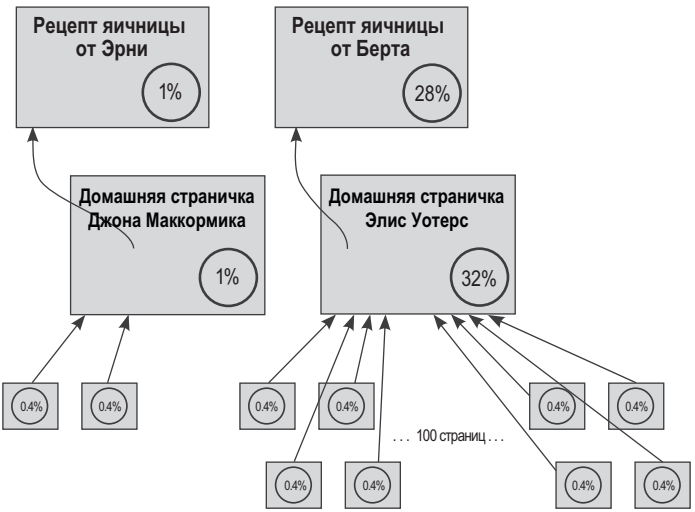
Какова связь между моделью случайного посетителя и трюком с авторитетностью, который мы хотели бы использовать для ранжирования веб-страниц? Вычисленные по модели случайного посетителя проценты – как раз и есть мера авторитетности страницы. Назовем *посетительской оценкой авторитетности* веб-страницы долю времени, которую случайный посетитель проводит на этой странице. Примечательно, что посетительская оценка авторитетности включает оба описанных выше трюка ранжирования веб-страниц по важности. Рассмотрим их по очереди.



Моделирование случайного посетителя.
Вверху: количество посещений каждой страницы
при моделировании 1000 посещений.
Внизу: процентная доля посещений каждой страницы
при моделировании миллиона посещений.

Сначала трюк с гиперссылками: его основная идея заключается в том, что ранг страницы тем выше, чем больше ссылок на нее ведет. Это верно и для модели случайного посетителя, поскольку у страницы, на которую ведет много ссылок, больше шансов быть посещенной. Страница *D* на нижнем рисунке с предыдущей страницы дает хороший пример: на нее ведет пять ссылок – больше, чем на любую другую страницу в модели, – в результате чего она получает наивысшую посетительскую оценку авторитетности (15 %).

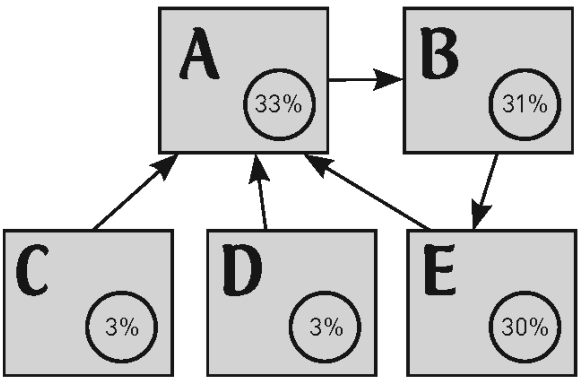
Теперь трюк с гиперссылками. Его основная идея в том, что ссылка с более авторитетной страницы должна давать больший вклад в ранг страницы, чем ссылка с менее авторитетной. Модель случайного посетителя обладает и этим свойством. Почему? Потому что по ссылке с популярной страницы проследуют с большей вероятностью, чем с непопулярной. Чтобы убедиться в этом, сравните страницы *A* и *C* на нижнем рисунке с предыдущей страницы: на каждую ведет ровно одна ссылка, но у страницы *A* посетительская оценка авторитетности гораздо выше (13 % против 2 %) из-за качества ведущей на нее ссылки.



Посетительские оценки авторитетности для примера рецептов яичницы. На страницы Берта и Эрни ведет по одной ссылке, определяющей их авторитетность, ранг страницы Берта относительно запроса «яичница» выше. Отметим, что модель случайного посетителя включает одновременно трюк с гиперссылками и трюк с авторитетностью. Иными словами, во внимание принимается как количество, так и качество входящих ссылок. Это наглядно демонстрирует страница B: она получает относительно высокую оценку (10 %), потому что на нее ведут три ссылки, хотя оценка каждой из них невысока: от 4 % до 7 %.

Прелесть трюка со случайным посетителем в том, что в отличие от трюка с авторитетностью он работает вне зависимости от того, образуют гиперссылки цикл или нет. Несложно прогнать модель случайного посетителя для нашего примера с рецептами яичницы. После нескольких миллионов посещений я получил посетительские оценки авторитетности, показанные на рисунке выше. Обратите внимание, что, как и при вычислениях, основанных на трюке с авторитетностью, страница Берта получает гораздо более высокую оценку, чем страница Эрни (28 % против 1 %), – несмотря на то, что на каждую ведет ровно одна ссылка. Поэтому Берт оказался бы выше в списке результатов поиска по запросу «яичница».

А теперь обратимся к гораздо более сложному примеру показанному на рисунке ниже, который из-за наличия циклов оказался неразрешимой задачей для исходного трюка с авторитетностью. И в этом случае нетрудно прогнать модель случайного посетителя, которая дает показанные на рисунке выше посетительские оценки авторитетности. Это как раз и есть ранги, которые поисковая система будет использовать при возврате результатов: у страницы *A* самый высокий ранг, за ней идут *B* и *E*, а *C* и *D* делят последнее место.



Посетительские оценки авторитетности для примера с циклом гиперссылок. При использовании трюка со случайным посетителем вычисление оценок не вызывает никаких затруднений, несмотря на наличие цикла (A → B → E → A).

Алгоритм PageRank на практике

Трюк со случайным посетителем был описан сооснователями Google в ставшей знаменитой статье «The Anatomy of a Largescale Hypertextual Web Search Engine». В сочетании со многими другими приемами этот

трюк все еще используется в большинстве основных поисковых систем. Однако существует целый ряд осложнений, из-за которых реальные современные системы применяют технику, несколько отличающуюся от описанной выше.

Одно из таких осложнений бьет в самое сердце PageRank: предположение о том, гиперссылка несет с собой повышение авторитетности, иногда сомнительно. Мы уже говорили о том, что хотя некоторые ссылки представляют собой критику, а не рекомендацию, на практике это не вызывает особых проблем. Куда серьезнее другая проблема – желание некоторых людей употребить во вред трюк с гиперссылками, чтобы искусственно повысить ранг собственных веб-страниц. Допустим, вы создали сайт BooksBooksBooks.com, на котором продаются – ну, естественно – книги. С помощью техники автоматизации сравнительно нетрудно создать много – скажем, 10 000 – разных веб-страниц, которые будут ссылаться на BooksBooksBooks.com. Если бы поисковая система вычисляла авторитетность, как описано выше, то BooksBooksBooks.com незаслуженно получил бы оценку в тысячи раз большую, чем у других книжных магазинов, а вместе с ней более высокое место в результатах поиска и больший объем продаж.

В поисковых системах такого рода злоупотребления называются *веб-спамом*. (Название дано по аналогии с почтовым спамом: непрошеные сообщения в вашем почтовом ящике похожи на никому не нужные веб-страницы, засоряющие результаты поиска.) Обнаружение и устранение различных видов веб-спама является важной и постоянной задачей во всех поисковых системах. Например, в 2004 году исследователи из Microsoft нашли 300 000 сайтов, на каждый из которых вела *ровно* 1001 ссылка – ситуация, вызывающая серьезные подозрения. Исследование этих сайтов вручную показало, что подавляющее большинство входящих ссылок – веб-спам.

Таким образом, поисковые системы вынуждены втягиваться в гонку вооружений с веб-спамерами и постоянно совершенствовать алгоритмы, чтобы получаемые ранги отражали реальное положение вещей. Стремление улучшить алгоритм PageRank дало начало многочисленным исследованиям в академических и промышленных кругах с целью найти иные алгоритмы, в которых гипертекстовая структура веб используется для ранжирования результатов. Подобные алгоритмы часто называют *алгоритмами ссылочного ранжирования*.

Еще одно осложнение связано с эффективностью вычислений в алгоритме PageRank. Мы для вычисления посетительских оценок авторитетности использовали случайную модель, но прогон такой модели

для всей сети занял бы слишком много времени, поэтому практически он бесполезен. В поисковых системах величина PageRank рассчитывается не с помощью моделирования случайного посетителя, а математически; ответ при этом получается такой же, но объем вычислений гораздо меньше. Мы рассматривали технику моделирования, поскольку она интуитивно понятна и описывает, *что* вычисляет поисковая система, а не *как* она это делает.

Следует также отметить, что в коммерческих поисковых системах вычисление ранга отнюдь не ограничивается применением алгоритма ссылочного ранжирования типа PageRank. Даже в оригинальной опубликованной в 1998 году статье с описанием системы Google ее сооснователи упоминали еще несколько применяемых в ней методов ранжирования результатов поиска. Нетрудно догадаться, что с тех пор технология шагнула вперед: на момент написания этой книги на сайте Google говорилось о «более чем 200 сигналах», используемых при оценке важности страницы.

Несмотря на многочисленные усложнения, характерные для современных поисковых систем, красивая идея, лежащая в основе алгоритма PageRank, – тот факт, что авторитетные страницы могут вносить вклад в авторитетность других страниц посредством гиперссылок, – остается в силе. Именно эта идея помогла Google свергнуть AltaVista с пьедестала и за несколько бурных лет превратить Google из небольшой начинающей компании в короля поиска. Без основополагающей идеи PageRank результаты большинства операций поиска в веб давали бы тысячи подходящих, но нерелевантных страниц. PageRank поистине алгоритмическая жемчужина, которая позволяет иголке подняться на самую вершину стога сена.

ГЛАВА 4. Криптография с открытым ключом: отправка секретов почтовой открыткой

Кто знает о моих заветных тайнах, скрытых от мира?

— Боб Дилан, песня *Covenant Woman*

Люди любят сплетничать. И секреты они тоже любят. А поскольку задача криптографии состоит в передаче секретов, все мы немного криптографы. Однако людям обменяться секретом проще, чем компьютерам. Если вы хотите сообщить секрет другу, можете просто пошептать ему на ушко. У компьютеров такой возможности нет. Компьютер не может «прошептать» номер кредитной карты другому компьютеру. Если два компьютера связаны через Интернет, то они не контролируют, куда попадет номер кредитной карты и какие еще компьютеры могут им завладеть. В этой главе мы рассмотрим, как компьютеры обходят эту проблему, применяя одну из самых изобретательных идей информатики, оказавшую огромное влияние на нашу жизнь: криптографию с открытым ключом.

Вы, наверное, недоумеваете, почему эта глава называется «отправка секретов почтовой открыткой». Ответ дает рисунок на обороте страницы: передачу информации с помощью почтовых открыток можно использовать для демонстрации действенности криптографии с открытым ключом. Если бы в реальной жизни вы захотели отправить почтой секретный документ, то, наверное, положили бы его в надежно запечатанный конверт. Это не гарантирует конфиденциальности, но является шагом в правильном направлении. С другой стороны, написав секретное сообщение на обороте почтовой открытки, вы очевидным образом нарушаете конфиденциальность: всякий, в чьи руки

попадет открытка (работники почты, например), сможет прочитать сообщение.

Именно с этой проблемой и сталкиваются компьютеры при попытке конфиденциальной передачи информации через Интернет. Любое сообщение в Интернете как правило проходит через многочисленные промежуточные компьютеры, которые называются маршрутизаторами, и его содержимое может прочитать любой, кто имеет к маршрутизатору доступ – в том числе любители подслушивать с враждебными целями. А значит, любые сведения, передаваемые вашим компьютером через Интернет, можно было бы с тем же успехом написать на открытке.

Аналогия с почтовой открыткой: очевидно, что текст на открытке, отправленной обычной почтой, не остается в секрете. По той же причине номер кредитной карты, отправленный с вашего ноутбука на Amazon.com, легко может быть перехвачен по пути, если его не зашифровать.

Вы, наверное, уже придумали, как по-быстрому исправить ситуацию с почтовой открыткой. Почему бы просто не шифровать секретным кодом все сообщения перед отправкой? Да, это сработает, если вы знаете человека, которому посылаете открытку. То есть когда-то в прошлом договорились о том, какой секретный код использовать. Проблема возникает, когда вы посылаете открытку незнакомцу. Если

вы зашифруете текст сообщения, то работники почты не смогут его прочесть, но этого не сможет сделать и предполагаемый получатель! А смысл криптографии с открытым ключом в том и состоит, чтобы использовать секретный код, который сможет расшифровать только получатель и никто другой – хотя у вас и не было возможности заранее договориться о коде.

Отметим, что перед компьютерами стоит та же проблема – обмен информацией с получателями, которых они не «знают». Так, в первый раз, когда вы что-то покупаете на Amazon.com, расплачиваясь кредитной картой, ваш компьютер должен передать номер карты серверу, принадлежащему Amazon. Но ваш компьютер никогда раньше не общался с сервером Amazon, поэтому у них не было возможности договориться о секретном коде. А любое соглашение, которое они попытаются заключить, может быть подслушано всеми маршрутизаторами на пути через Интернет.

Вернемся к аналогии с почтовой открыткой. На первый взгляд, налицо неразрешимый парадокс: получатель видит ту же информацию, что работники почты, но каким-то образом получатель узнает, как расшифровать сообщение, а работники почты – нет. Этот парадокс разрешает криптография с открытым ключом. И в этой главе мы объясним, как.

Шифрование с помощью общего секрета

Начнем с простого мысленного эксперимента. Оставим в стороне почтовые открытки и возьмем ситуацию попроще: словесное общение в комнате. Допустим, что в комнате находятся ваш друг Арнольд и ваш враг Ева. Вы хотите сообщить что-то секретное Арнольду, так чтобы Ева не смогла понять, что именно. Возможно, секретом является номер кредитной карты, но простоты ради предположим, что номер очень короткий – всего одна цифра от 1 до 9. Кроме того, предположим, что единственный способ общения с Арнольдом – произнесение вслух, поэтому Ева все слышит. Никакие уловки – произнесение шепотом, передача записки или еще что-то в этом роде – не допускаются.

Для определенности предположим, что кредитная карта имеет номер 7, и это-то число необходимо сообщить. Вот как можно было бы подойти к решению задачи. Во-первых, попробуйте придумать какое-нибудь число, которое Арнольд знает, а Ева нет. Пусть, напри-

мер, вы с Арнольдом старые друзья и в детстве жили на одной улице. Допустим, вы часто играли во дворе вашего дома по адресу Веселая улица, 322. Еще допустим, что Ева в детстве не была с вами знакома и не знает адреса дома, где вы с Арнольдом когда-то играли. Тогда вы можете сказать Арнольду: «Слушай, помнишь номер нашего дома на Веселой, где мы играли в детстве? Так вот, если к этому числу прибавить однозначный номер кредитной карты, который я задумал, то получится 329».

Трюк со сложением: сообщение 7 шифруется путем прибавления его к общему секрету 322. Арнольд может расшифровать сообщение, вычтя общий секрет, но Ева этого сделать не может.

Итак, если Арнольд помнит номер дома, то сможет узнать номер кредитной карты, вычтя из сообщенного вами числа 329 номер дома: 329 – 322 = 7. Ева же при этом ничего не узнает о номере кредитной карты, хотя слышала все, что вы говорили Арнольду. Этот процесс продемонстрирован на рисунке выше.

Почему этот метод работает? Потому что у вас с Арнольдом есть нечто, что специалисты по информатике называют *общим секретом*: число 322. Поскольку вы оба знаете это число, а Ева не знает, то его можно использовать, чтобы безопасно сообщить любое другое число, – достаточно вычислить и объявить сумму, а другая сторона сможет вычесть из нее общий секрет. Подслушивание суммы бесполезно для Евы, потому что она не знает, какое число из нее вычитать.

Хотите верьте, хотите нет, но если вы поняли этот простой «трюк со сложением» общего секрета и секретного сообщения, например номера кредитной карты, то уже представляете общий принцип шифрования в Интернете! Компьютеры пользуются этим трюком постоянно,

но чтобы по-настоящему обезопасить его, нужно позаботиться о некоторых деталях.

Во-первых, общие секреты компьютеров должны быть гораздо длиннее номера дома (322). Если секрет слишком короткий, то любой, кто подслушал разговора, можете попросту перебрать все возможности. Пусть, например, мы используем трехзначный номер дома для шифрования *настоящих* 16-значных номеров кредитных карт. Всего существует 999 трехзначных чисел, поэтому противнику, подслушавшему наш разговор, например Еве, достаточно перебрать 999 чисел, из которых одно обязательно окажется номером карты. Для такого перебора компьютеру потребуется лишь краткое мгновение, поэтому если мы хотим, чтобы общий секрет представлял хоть какую-то ценность, в нем должно быть куда больше цифр.

На самом деле, слыша, как кто-то упоминает в разговоре о шифровании количество бит, например «128-битное шифрование», знайте – это и есть описание длины общего секрета. Часто общий секрет называют «ключом», поскольку он позволяет открыть, или «расшифровать» сообщение. Взяв 30 % от количества бит в ключе, мы получим приблизительное число цифр в ключе. Поскольку 30 % от 128 равно 38, ключ 128-битного шифрования представляет собой 38-значное число[1]. 38-значное число больше миллиарда миллиардов миллиардов миллиардов, а поскольку любому из ныне существующих компьютеров для перебора такого количества возможностей понадобятся миллиарды лет, общий секрет такой длины считается в высшей степени безопасным.

Есть и еще один изъян, мешающий использовать трюк со сложением в реальной жизни: результат сложения поддается статистическому анализу, то есть ключ можно определить, проанализировав много зашифрованных им сообщений. Поэтому в современных методах шифрования используются так называемые «блочные шифры», в которых применяется некий вариант трюка со сложением.

Во-первых, длинные сообщения разбиваются на короткие «блоки» фиксированного размера, обычно 10–15 символов. Во-вторых, блок не просто складывается с ключом, а несколько раз преобразуется по определенным правилам, чтобы цифры как следует перемешались. Например, правило может звучать так: «прибавить первую половину ключа к последней половине блока, переставить цифры результата в

[1] Для читателей, знакомых с системами счисления, применяемыми в компьютерах, скажу, что я имею в виду десятичные цифры, а не двоичные (биты). А для тех, кто помнит, что такое логарифм, сообщу, что коэффициент преобразования числа бит в число десятичных цифр равен 30 %, потому что $\log_{10}2 \approx 0{,}3$.

обратном порядке и прибавить вторую половину ключа к последней половине блока». В реальности, впрочем, правила несколько сложнее. В современных блочных шифров обычно применяется 10, а то и больше «раундов» таких операций, т. е. набор операций применяется повторно. Если число раундов достаточно велико, то исходное сообщение настолько основательно перемешано, что устоит против любых статистических атак, и всякий, кто знает ключ, должен будет для восстановления незашифрованного сообщения проделать все операции в обратном порядке.

На момент написания этой книги самым популярным блочным шифром был AES (Advanced Encryption Standard). Этот шифр можно использовать в разных режимах, но как правило берутся блоки по 16 символов, 128-битные ключи и 10 раундов операций перемешивания.

Открытая выработка общего секрета

Пока все идет хорошо. Мы выяснили, как работает подавляющее большинство схем шифрования в Интернете: разбить сообщение на блоки и использовать вариант трюка со сложением для шифрования каждого блока. Но, как выясняется, это самая простая часть. Проблема в том, как *определить* общий секрет. В примере с Арнольдом и Евой мы немного смошенничали – воспользовались тем фактом, что вы с Арнольдом росли вместе и потому знаете общий секрет (номер вашего дома), который Ева знать не может. Ну а если сыграть в ту же игру, когда все трое незнакомы между собой? Можете ли вы с Арнольдом выработать общий секрет, так чтобы Ева о нем ничего не узнала? (Напоминаю: никакого жульничества – нельзя шептать Арнольду на ушко или передавать записку, которую Ева не может видеть. Все общение должно быть открыто.)

На первый взгляд, это невозможно, но оказывается, что есть остроумный способ решить задачу. В информатике это решение называют *протоколом обмена ключами Диффи-Хеллмана*, а мы будем говорить о *трюке со смешиванием красок*.

Трюк со смешиванием красок

Чтобы понять, в чем идея трюка, мы на время забудем о передаче номера кредитной карты и представим, что секрет, который требуется

сделать общим, – это цвет краски. (Да, звучит странно, но, как мы скоро увидим, это очень полезный способ рассуждения о данной задаче.) Итак, предположим, что вы находитесь в одной комнате с Арнольдом и Евой и у каждого есть большой набор банок с разными красками. Каждый располагает одним и тем же набором цветов – есть много разных цветов и у каждого есть много банок с краской каждого цвета. Можно считать, что краска никогда не кончается. На каждой банке имеется наклейка с цветом краски, поэтому нетрудно сформулировать инструкции о том, как смешивать краски: «смешай одну банку "небесно-голубого" с шестью банками "бледно-желтого" и пятью банками "аквамарина"». Но поскольку имеются сотни, а то и тысячи цветов любого мыслимого оттенка, то, взглянув на результат смешивания, невозможно сказать, из каких цветов он был получен. А коль скоро цветов так много, то попытка определить состав смеси методом проб и ошибок тоже ничего не даст.

Теперь немного изменим правила игры. Пусть у каждого в комнате отгорожен уголок за занавеской – место, где вы храните банки с красками и можете смешивать их, так чтобы никто не видел. Но правила общения такие же, как и раньше – все в открытую. Приглашать Арнольда в свой уголок вы не вправе! Еще одно правило касается способа передачи результата смешивания красок в общее пользование. Вы можете передать смесь любому из находящихся в комнате людей, но только одним способом: поставить банку на пол в середине комнаты и подождать, пока кто-то ее заберет. Оптимальный путь – изготовить достаточно смеси для всех и оставить несколько порций в середине комнаты. Тогда каждый, кто захочет взять порцию, сможет это сделать. Это правило просто вытекает из того, что все общение должно быть открытым: если вы даете какую-то смесь Арнольду, а Еве не даете, значит, между вами и Арнольдом образовался некий «закрытый» канал связи, а это против правил.

Напомню, что игра в смешивание красок призвана объяснить, как вырабатывается общий секрет. Наверное, вы недоумеваете, какое отношение смешивание красок имеет к криптографии, но всему свое время. Сейчас вы узнаете о поразительном трюке, которым компьютеры реально пользуются для выработки общих секретов в открытом пространстве, каковым является Интернет!

Для начала определим цель игры. Для вас с Арнольдом она состоит в том, чтобы создать *одну и ту же* смесь красок, не раскрыв Еве, как она получается. Если эта цель будет достигнута, то мы скажем, что вам с Арнольдом удалось выработать «общую секретную смесь».

Разрешается вести открытые переговоры сколь угодно долго, а также переносить банки с краской между серединой комнаты и вашим личным уголком.

Теперь отправимся в путешествие по миру остроумных идей, стоящих за криптографией с открытым ключом. Разобьем наш трюк со смешиванием красок на четыре шага.

Шаг 1. Вы и Арнольд порознь выбрали свой «закрытый цвет».

Ваш закрытый цвет – не то же самое, что общая секретная смесь, которая получится в конечном итоге, но он станет одним из ее ингредиентов. В качестве закрытого можно выбрать любой цвет, но вы должны его запомнить. Разумеется, ваш закрытый цвет почти наверняка не совпадет с закрытым цветом Арнольда, потому что цветов очень много. Для определенности предположим, что вы выбрали лавандовый цвет, а Арнольд – малиновый.

Шаг 2. Один из вас открыто объявляет ингредиенты еще одного цвета, который мы будем называть «открытым цветом».

И на этот раз можно выбрать любой цвет. Допустим, вы объявили открытым цветом ромашковый. Отметим, что существует всего один открытый цвет (а не свой у вас и у Арнольда), и, само собой, Ева знает, что это за цвет, потому что вы объявили о нем во всеуслышание.

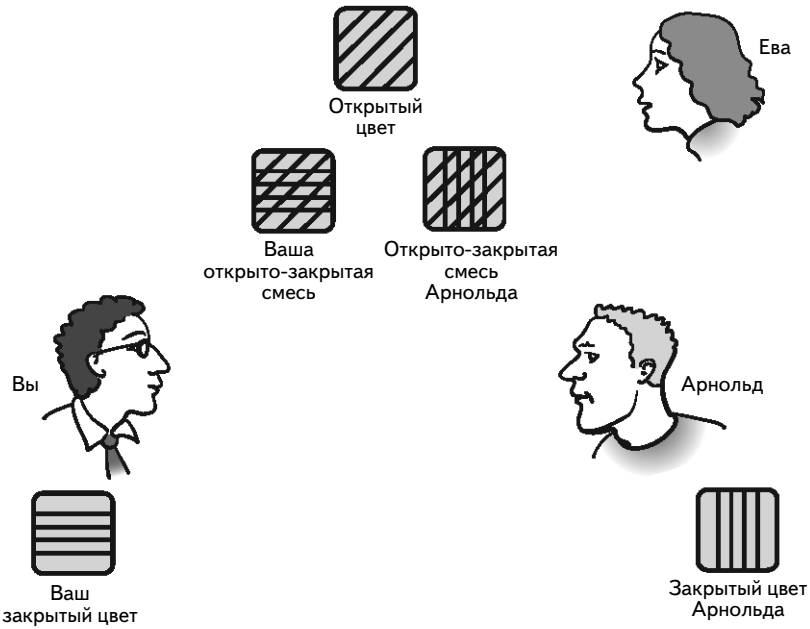
Шаг 3. И вы, и Арнольд смешиваете одну банку открытого цвета с одной банкой своего закрытого. Получается «открыто-закрытая смесь».

Очевидно, открыто-закрытая смесь Арнольда не будет совпадать с вашей, поскольку его закрытый цвет отличается от вашего. Ваша открыто-закрытая смесь будет состоять из банки лавандовой и банки ромашковой краски, а смесь Арнольда – из банки малиновой и банки ромашковой.

В этот момент вы и Арнольд хотели бы обменяться образцами своих открыто-закрытых смесей, но не забывайте, что не разрешается передавать смесь красок напрямую кому-то из присутствующих. Единственное, что вы можете, – приготовить несколько порций смеси и оставить их посредине комнаты, чтобы мог взять любой желающий. Так вы с Арнольдом и поступаете. Ева может украсть порцию-другую,

если хочет, но, как мы скоро убедимся, это ей ничего не даст. На следующем рисунке изображена ситуация после третьего шага трюка со смешиванием красок.

Отлично, мы движемся в верном направлении. Немного поразмыслив, вы, возможно, поймете, какое финальное действие позволит вам с Арнольдом создать одинаковую общую секретную смесь, не выдав секрет Еве. Ответ выглядит так:



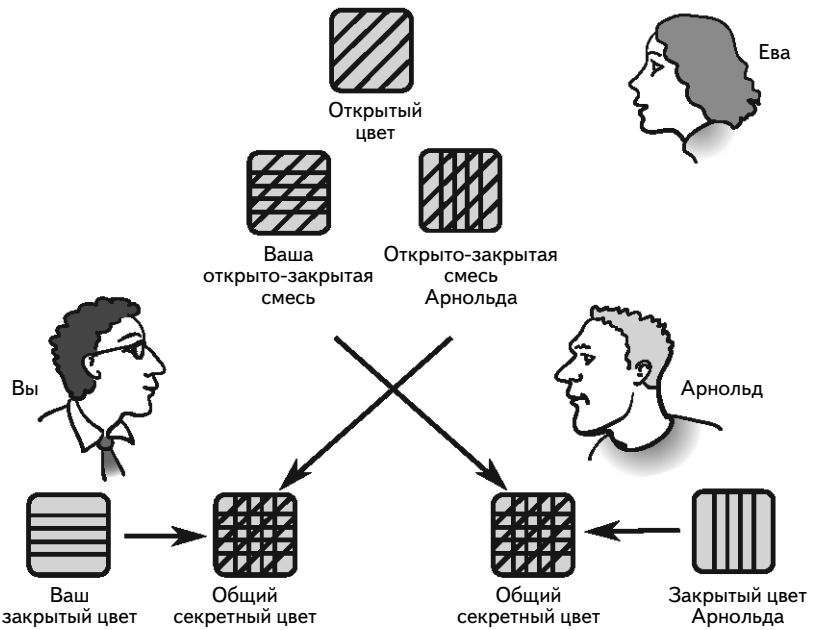
Трюк со смешиванием красок, шаг 3: открыто-закрытые смеси доступны любому желающему.

Шаг 4. Вы берете порцию открыто-закрытой смеси Арнольда, несете ее в свой угол и добавляете банку своего закрытого цвета. Тем временем Арнольд несет к себе порцию вашей смеси и добавляет в нее банку своего закрытого цвета.

Ура, вы создали совершенно одинаковые смеси! Убедимся: вы добавили свой закрытый цвет (лавандовый) в открыто-закрытую смесь Арнольда (малиновый и ромашковый), получив в результате смесь, состоящую из 1 части лавандового, 1 части малинового и 1 части ромашкового. А что Арнольд? Он добавил свой закрытый цвет (малино-

вый) в вашу открыто-закрытую смесь (лавандовый и ромашковый) и получил смесь из 1 части малинового, 1 части лавандового и 1 части ромашкового. То есть в точности такую же, как и вы. Действительно налицо общая секретная смесь. На следующем рисунке показана ситуация после финального шага.

А что же Ева? Почему она не может создать порцию общей секретной смеси? Да потому, что не знает ни вашего секретного цвета, ни Арнольдова. А для создания общей секретной смеси нужно знать хотя бы один из них. Вам удалось провести Еву, потому что вы не оставляли посредине комнаты своих закрытых цветов в чистом виде, а лишь в виде смеси с открытым цветом, а Ева не умеет «разделять» смесь на составные части.



Трюк со смешиванием красок, шаг 4: только вы и Арнольд можете изготовить общий секретный цвет, объединив смеси, как показано стрелками.

Таким образом, у Евы есть доступ *только* к обеим открыто-закрытым смесям. Если она смешает одну порцию вашей открыто-закрытой смеси с одной порцией смеси Арнольда, то получит смесь, содержащую 1 часть малинового, 1 часть лавандового и 2 части ромашкового. То есть будет на 1 часть ромашкового больше, чем в общей секретной

смеси. Ее смесь окажется слишком желтой, а поскольку способа разделить смесь не существует, то удалить лишний желтый не получится. Если вы думаете, что проблему можно решить, добавив больше малинового и лавандового, то вспомните, что Ева не знает закрытых цветов, т. е. не знает, что именно нужно добавить. Она может добавлять только комбинации «малиновый плюс ромашковый» или «лавандовый плюс ромашковый», и в обоих случаях смесь получится чрезмерно желтой.

Числа вместо красок

Если вы понимаете трюк со смешиванием красок, то понимаете существо выработки общих секретов двумя компьютерами в Интернете. Но, конечно, красками они не пользуются. Компьютеры работают с числами, а «смешивание» чисел основано на математике. Вообще-то эта математика не слишком сложна, но поначалу может вызвать затруднения. Поэтому, чтобы идти дальше, мы воспользуемся «нечестной» математикой. Проблема состоит в том, что для перевода трюка со смешиванием красок на язык чисел нам необходимо *одностороннее действие*: нечто, что можно *сделать*, но нельзя *обратить*. В описанном выше трюке таким действием было «смешивание красок». Смешать краски для получения нового цвета легко, но разделить их и получить исходные цвета невозможно. Потому-то смешивание красок и является односторонним действием.

Выше мы сказали, что воспользуемся нечестной математикой. И вот в чем состоит обман: *умножение двух чисел – одностороннее действие*. Вы, конечно, понимаете, что это неправда. Операцией, противоположной умножению, является деление, и, выполнив деление, умножение легко «обратить». Например, если начать с числа 5 и умножить его на 7, то мы получим 35. Операцию легко обратить: если разделить 35 на 7, то получится 5 – то, с чего мы и начинали.

Но, тем не менее, мы смиримся с этим обманом и сыграем еще в одну игру между вами, Арнольдом и Евой. На этот раз будем предполагать, что все умеют перемножать числа, но никто не умеет делить одно число на другое. Задача почти такая же, как и прежде: вы с Арнольдом пытаетесь выработать общий секрет, только теперь это будет число, а не цвет краски. Действуют те же правила общения: все производится открыто, так что Ева слышит все, о чем вы с Арнольдом говорите.

Итак, вот что нужно сделать, чтобы перевести трюк со смешиванием красок на язык чисел.

Шаг 1. Вместо выбора «закрытого цвета» вы и Арнольд выбираете по «закрытому числу».

Допустим, вы выбрали 4, а Арнольд – 6. Теперь подумаем, как должны выглядеть остальные шаги трюка со смешиванием красок: объявление открытого цвета, составление открыто-закрытой смеси, обмен открыто-закрытыми смесями и добавление своего закрытого цвета в чужую смесь для получения общего секретного цвета. Все это нетрудно сформулировать в виде операций с числами, если в качестве односторонней операции взять умножение вместо смешивания красок. Прежде чем читать дальше, подумайте, сможете ли довести этот пример до конца самостоятельно.

Решение понять нетрудно. Вы оба уже выбрали закрытые числа (4 и 6), и следующий шаг выглядит так:

Шаг 2. Один из вас объявляет «открытое число» (а не открытый цвет, как в трюке со смешиванием красок).

Допустим, в качестве открытого числа выбрано 7.

Следующий шаг – составление открыто-закрытой смеси. Раз мы решили вместо смешивания красок перемножать числа, то очередной шаг звучит так:

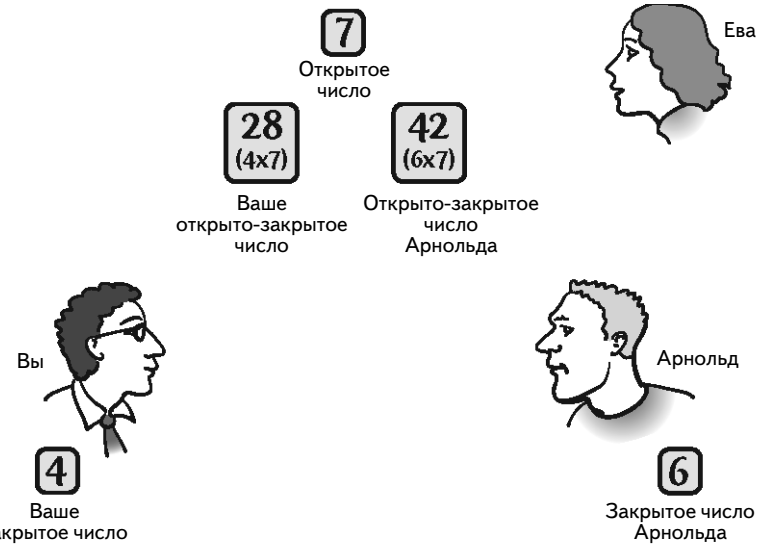
Шаг 3. Умножить свое закрытое число (4) на открытое число (7), получив в результате «открыто-закрытое число» 28.

Вы можете объявить это во всеуслышание, чтобы Арнольд и Ева знали, что ваше открыто-закрытое число равно 28 (таскать банки с краской больше не нужно). Арнольд делает то же самое со своим закрытым числом: умножает его на открытое число и объявляет свое открыто-закрытое число: $6 \times 7 = 42$. На следующем рисунке изображена ситуация в этот момент.

Помните, каким был последний шаг трюка со смешиванием красок? Вы взяли открыто-закрытую смесь Арнольда и добавили банку своего закрытого цвета, чтобы получить общий секретный цвет. То же самое происходит и сейчас, только вместо смешивания красок перемножаются числа.

Шаг 4. Вы берете открыто-закрытое число Арнольда – 42 – и умножаете его на свое закрытое число 4. В результате получается *общее секретное число* 168.

Тем временем Арнольд берет *ваше* открыто-закрытое число – 28 – и умножает его на *свое* закрытое число 6, получая – вот ведь сюрприз! – то же самое общее секретное число, т. к. 28 × 6 = 168. Окончательный результат показан на следующем рисунке.

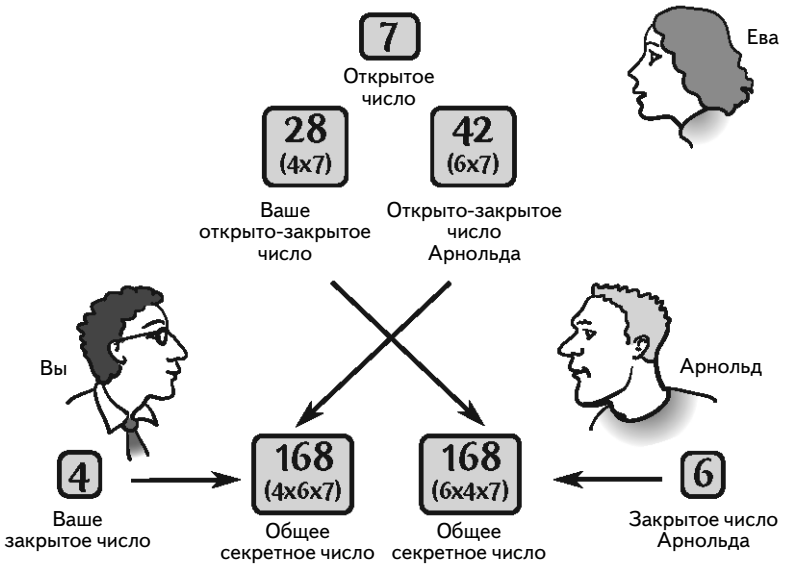


Трюк со смешиванием чисел, шаг 3: открыто-закрытые числа доступны любому желающему.

На самом деле, если подумать, никакого сюрприза нет. Вам с Арнольдом удалось получить один и тот же общий секретный цвет, потому что вы смешивали одни и те же краски, только в разном порядке: каждый добавил свой закрытый цвет в общедоступную смесь двух других цветов. То же самое происходит и с числами. Один и тот же результат получился, потому что перемножались одни и те же числа: 4, 6 и 7. (Убедитесь сами: 4 × 6 × 7 = 168.) Но *вы* пришли к этому результату, закрыв от всех прочих число 4 и «подмешав» его (умножив) к общедоступной смеси чисел 6 и 7 (т. е. 42), объявленной Арнольдом. А *Арнольд* пришел к общему секрету, закрыв число 6 и подмешав его к общедоступной смеси чисел 4 и 7 (т. е. 28), объявленной вами.

Как и в трюке со смешиванием красок, проверим, что у Евы нет возможности узнать общий секрет. Ева знает значения обоих открыто-закрытых чисел, она слышала, как вы сказали «28», а Арнольд – «42». Она также знает, что открытое число равно 7. *Если бы* Ева умела выполнять деление, она тут же раскрыла бы все ваши секреты, вы-

числив, что 28 / 7 = 4, а 42 / 7 = 6. А затем она вычислила бы общий секрет: 4 × 6 × 7 = 168. На ваше счастье, мы пользуемся нечестной математикой, в которой умножение – одностороннее действие, и, стало быть, делить Ева *не* умеет. Таким образом, у нее есть три числа 28, 42 и 7. Она может их перемножить, но это никак не поможет ей узнать общий секрет. Например, результат 28 × 42 = 1176 очень далек от правильного. Если в игре со смешиванием красок в ее смеси было слишком много желтого, то теперь в произведении слишком много семерок. В общем секрете есть только один множитель, равный 7, т. к. 168 = 4 × 6 × 7. А при попытке Евы раскрыть секрет, получилось два таких множителя, потому что 1176 = 4 × 6 × 7 × 7. И избавиться от лишней семерки невозможно, потому что делить Ева не умеет.



Трюк со смешиванием чисел, шаг 4: только вы и Арнольд можете изготовить общее секретное число, перемножив числа, как показано стрелками.

Смешивание красок в реальной жизни

Мы рассмотрели все идеи, благодаря которым компьютеры могут вырабатывать общие секреты в Интернете. Единственный изъян в схеме «смешивания чисел» – использование «нечестной математики», в которой ни одна из сторон якобы не умеет выполнять деление. Чтобы довести рецепт до конца, нам нужна настоящая математическая опе-

рация, которую очень легко выполнить (не труднее, чем смешать краски), но практически невозможно обратить (как невозможно разделить смесь красок на составляющие). В реальной жизни выполняемая компьютерами операция смешивания называется *дискретным возведением в степень*, а операция разделения красок – *дискретным логарифмированием*. Поскольку неизвестен эффективный способ вычисления дискретного логарифма компьютером, дискретное возведение в степень оказывается как раз той односторонней операцией, которую мы ищем. Чтобы объяснить, в чем суть дискретного возведения в степень, нам понадобятся две простые математические идеи. И придется написать несколько формул. Если вы не любите формул, просто пропустите этот раздел – почти все, что относится к данной теме, вы уже и так понимаете. Но если вы хотите по-настоящему разобраться в том, как компьютеры вершат свое волшебство, читайте дальше.

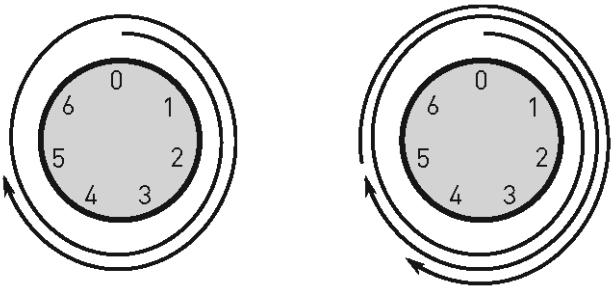
Первая важная идея называется *арифметикой часов*. В общем-то, все мы с ней давно знакомы: на циферблате часов есть всего 12 делений, поэтому когда часовая стрелка пересекает 12, отсчет начинается снова с 1. Мероприятие, которое начинается в 10 часов и продолжается 4 часа, закончится в 2 часа, поэтому можно сказать, что в этой 12-часовой системе 10 + 4 = 2. В математике арифметика часов работает аналогично, но с двумя отличиями: (1) количество делений на циферблате может быть произвольным и (2) отсчет начинается не с 1, а с 0.

На следующей странице изображен пример циферблата с 7 делениями. Обратите внимание, что деления обозначены числами 0, 1, 2, 3, 4, 5, 6. Для выполнения математических операций в арифметике часов с 7 делениями нужно складывать и умножать числа как обычно, но в качестве результата брать остаток от деления суммы или произведения на 7. Так, чтобы вычислить 12 + 6, мы сначала производим обычное сложение и получаем 18. А затем замечаем, что 7 умещается в 18 дважды (получается 14) и еще 4 остается. Поэтому окончательный ответ будет:

$$12 + 6 = 4 \text{ (при 7 делениях на циферблате)}$$

В примерах ниже мы предполагаем, что на циферблате 11 делений. (Как уже отмечалось, в реальной системе количество делений будет намного больше. Мы взяли маленький циферблат, чтобы упростить объяснение.) Вычислить остаток от деления на 11 нетрудно, нужно только вспомнить, что в (двузначных) числах, кратных 11, все цифры одинаковы, например, 66 или 88. Вот несколько примеров вычислений в арифметике часов с 11 делениями:

$$7 + 9 + 8 = 24 = 2 \text{ (при 11 делениях на циферблате)}$$
$$8 \times 7 = 56 = 1 \text{ (при 11 делениях на циферблате)}$$



Слева: если на циферблате 7 делений, то число 12 превращается в 5 – чтобы убедиться в этом, начните с нуля и сдвиньтесь на 12 делений по часовой стрелке, в направлении, показанном на рисунке. Справа: пользуясь тем же циферблатом с 7 делениями, мы обнаруживаем, что 12 + 6 = 4 – начните с цифры 5, на которой мы остановились на рисунке слева, и сдвиньтесь еще на 6 делений по часовой стрелке.

Вторая математическая идея – это *степенная нотация*. Тут ничего хитрого нет: это просто короткий способ записать насколько операций умножения числа на себя самого. Вместо того чтобы писать $6 \times 6 \times 6 \times 6$, можно написать 6^4. Степенную нотацию можно сочетать с арифметикой часов. Например:

$$34 = 3 \times 3 \times 3 \times 3 = 81 = 4 \text{ (при 11 делениях на циферблате)}$$
$$72 = \quad 7 \times 7 \quad = 49 = 5 \text{ (при 11 делениях на циферблате)}$$

В таблице на следующей странице показаны первые десять степеней чисел 2, 3 и 6 в арифметике часов с 11 делениями. Эта таблица пригодится при обсуждении примера ниже. Но прежде чем приступать к этому, давайте разберемся, как эта таблица была построена. Взгляните на последний столбец. Первое число в нем – 6, это то же самое, что 6^1. Следующим должно быть число 6^2, или 36, но поскольку на циферблате всего 11 делений, а 36 на 3 больше, чем 33, то в таблице находится число 3. Вы, наверное, думаете, что для вычисления третьего числа в этом столбце нужно сначала вычислить $6^3 = 6 \times 6 \times 6$, но можно поступить проще. Мы уже знаем, чему равно 6^2 при интересующем нас количестве делений – 3. Чтобы вычислить 6^3, достаточно умножить предыдущий результат на 6. Получается $3 \times 6 = 18 = 7$ (при 11 делениях на циферблате). Следующим будет число $7 \times 6 = 42 = 9$ (при 11 делениях на циферблате) и так далее до конца столбца.

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

Ну вот, теперь мы наконец готовы выработать общий секрет, как это делают компьютеры в реальной жизни. Пусть, как и прежде, вы с Арнольдом пытаетесь выработать общий секрет, а Ева подслушивает и стремится узнать, что это за секрет.

Шаг 1. Вы и Арнольд независимо выбираете *закрытое число*, каждый свое.

Чтобы не усложнять вычисления, мы в этом примере будем использовать очень маленькие числа. Допустим, вы в качестве закрытого выбрали число 8, а Арнольд – 9. Сами по себе эти числа не являются общими секретами, но, как и закрытые цвета в трюке со смешиванием красок, они станут *ингредиентами*, подмешиваемыми в общий секрет.

Шаг 2. Вы с Арнольдом открыто договариваетесь о двух *открытых числах*: количестве делений на циферблате (в нашем примере 11) и так называемом *основании* (мы выберем основание 2).

Эти открытые числа – 11 и 2 – аналогичны открытому цвету, о котором вы с Арнольдом договаривались в начале трюка со смешиванием красок. Правда, аналогия со смешиванием красок здесь немного хромает: открытый цвет был один, а открытых чисел нужно два.

Шаг 3. Каждый из вас независимо вычисляет *открыто-закрытое число* (ОЗЧ), применяя к своему закрытому числу и обоим открытым операции возведения в степени и арифметику часов.

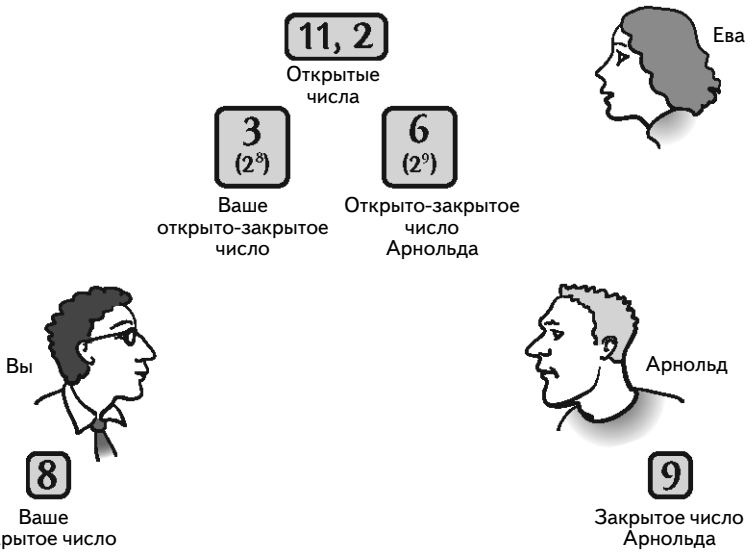
Точнее, смешивание производится по формуле

$$\text{ОЗЧ} = \text{основание}^{\text{закрытое число}}$$
(при заданном количестве делений на циферблате)

Записанная словами, эта формула выглядит тяжеловато, но на практике ничего сложного в ней нет. В нашем примере для вычислений можно воспользоваться приведенной выше таблицей:

ваше ОЗЧ = $2^8 = 3$ (при 11 делениях на циферблате)
ОЗЧ Арнольда = $2^9 = 6$ (при 11 делениях на циферблате)

На следующем рисунке изображена ситуация после этого шага. Открыто-закрытые числа в точности аналогичны «открыто-закрытым смесям», которые вы составляли на третьем шаге трюка со смешиванием красок. Там вы смешивали банку краски открытого цвета с банкой краски своего закрытого цвета. Здесь вы смешиваете свое закрытое число с двумя открытыми числами, применяя степенную нотацию и арифметику часов.



Смешивание чисел в реальной жизни, шаг 3: открыто-закрытые числа (3 и 6), вычисленные путем возведения в степень с применением арифметики часов, доступны любому желающему. Число 2^8 под числом 3 напоминает нам о том, как было получено 3, однако тот факт, что $3 = 2^8$ при 11 делениях на циферблате не разглашается. Точно так же остается закрытой информация о том, что 6 – это 2^9.

Шаг 4. Каждый из вас берет открыто-закрытое число другого и смешивает его со своим закрытым числом.

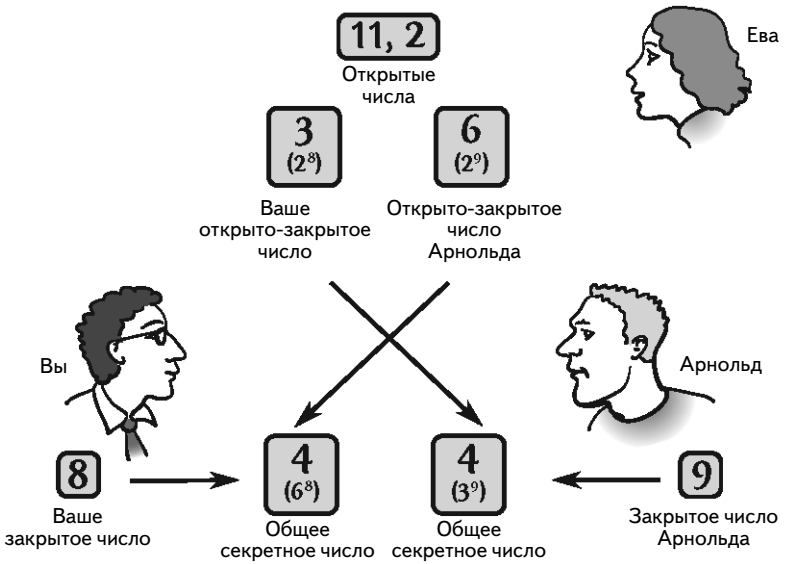
Это делается по такой формуле:

$$\text{общий секрет} = \text{ОЗЧ партнера}^{\text{закрытое число}}$$
(при заданном количестве делений на циферблате)

И снова читать формулу трудно, но применять на практике легко, если воспользоваться приведенной выше таблицей:

ваш общий секрет = $6^8 = 4$ (при 11 делениях на циферблате)
общий секрет Арнольда = $3^9 = 4$ (при 11 делениях на циферблате)

Конечная ситуация показана на следующем рисунке.



Смешивание чисел в реальной жизни, шаг 4: только вы и Арнольд можете вычислить общее секретное число, объединив показанные стрелками элементы с помощью операций возведения в степень и арифметики часов.

Естественно, у вас с Арнольдом получился один и тот же общий секрет (в данном случае 4). Чтобы объяснить, почему это так, понадобилась бы сложная математика, но идея та же, что и раньше: вы смешивали ингредиенты в разном порядке, но сами ингредиенты были одинаковы, потому и общий секрет получился одним и тем же.

Как и в предыдущих вариантах этого трюка, Ева осталась в дураках. Она знает оба открытых числа (2 и 11) и оба открыто-закрытых числа (3 и 6). Но воспользоваться этим знанием, для того чтобы вычислить общий секрет, она не может, так как не имеет доступа к секретным ингредиентам (закрытым числам), которые вы с Арнольдом храните в тайне.

Криптография с открытым ключом на практике

Окончательный вариант трюка со смешиванием красок – смешивание чисел с помощью возведения в степень и арифметики часов – один из реально применяемых способов выработки компьютерами общего секрета в Интернете. Описанный здесь метод называется протоколом обмена ключами Диффи-Хеллмана по имени Уитфилда Диффи и Мартина Хеллмана, которые впервые опубликовали этот алгоритм в 1976 году. Всякий раз как вы заходите на защищенный сайт (адрес которого начинается с «`https:`», а не «`http:`»), ваш компьютер и веб-сервер обмениваются данными для выработки общего секрета, применяя протокол Диффи-Хеллмана или подобный ему. После того как общий секрет выработан, компьютеры могут шифровать все передаваемые данные, применяя тот или иной вариант описанного выше трюка со сложением.

Важно понимать, что при практическом использовании протокола Диффи-Хеллмана используются гораздо большие числа, чем в нашем примере. Мы выбрали очень маленькое количество делений на циферблате (11), чтобы упростить вычисления. Но при таком выборе количества делений количество возможных закрытых чисел тоже мало (поскольку закрытое число должно быть меньше количества делений). А это означает, что противник может с помощью компьютера перебрать все возможные закрытые числа, пока ней найдет то, что дает ваше открыто-закрытое число. В нашем примере всего было 11 возможных закрытых чисел, такая система взламывается на раз-два-три. В настоящих же реализациях протокола Диффи-Хеллмана количество делений обычно выражается числом с несколькими сотнями цифр, так что количество возможных закрытых чисел невообразимо велико (гораздо больше триллиона триллионов). Но даже при таких условиях открытые числа следует выбирать с осторожностью, следя за тем, чтобы они обладали нуж-

ными математическими свойствами, – если вам интересно, прочитайте врезку ниже.

> Открытые числа в протоколе Диффи-Хеллмана – и это самое главное их свойство – должны быть простыми, т. е. не иметь делителей, кроме самого себя и единицы. Еще одно требование – основание должно быть *первообразным корнем по модулю*, равному количеству делений на циферблате. Это означает, что степени основания пробегают все присутствующие на циферблате значения. Взглянув на таблицу на стр. 67, вы убедитесь, что и 2 и 6 являются первообразными корнями по модулю 11, а 3 – нет, т. к. среди степеней 3 есть числа 3, 9, 5, 4, 1, но нет чисел 2, 6, 7, 8, 10.

При выборе используемых в протоколе Диффи-Хеллмана количества делений на циферблате и основания должны быть соблюдены некоторые математические свойства.

Описанный выше алгоритм Диффи-Хеллмана – это лишь один из многих хитроумных приемов обмена информацией с помощью (электронных) почтовых открыток. В информатике протокол Диффи-Хеллмана называется *алгоритмом обмена ключами*. Другие алгоритмы с открытым ключом работают иначе, они позволяют зашифровать сообщение с помощью открытой информации, объявленной предполагаемым получателем, так что прочесть его сможет только этот получатель. Алгоритм же обмена ключами позволяет выработать общий секрет на основе открытой информации от получателя, но само шифрование производится с помощью трюка со сложением. Для большинства операций обмена данными в Интернете последний вариант – тот, что мы изучали в этой главе, – предпочтительнее, так как требует гораздо меньше вычислительных ресурсов.

Однако существуют приложения, в которых необходима полноценная криптография с открытым ключом. Пожалуй, самое интересное из них – цифровые подписи, о которых речь пойдет в главе 9. Из нее вы узнаете, что идеи, применяемые в полноценной криптографии с открытым ключом, очень похожи на те, что вы уже видели: секретная информация «смешивается» с открытой способом, не допускающим обращения, – как краски разных цветов. Самая известная криптосистема с открытым ключом называется RSA по начальным буквам фамилий трех ее авторов: Рональда Райвеста (Ronald **R**ivest), Ади Шамира (Adi **S**hamir) и Леонарда Адлемана (Leonard **A**dleman). В главе 9 система RSA служит основой для демонстрации принципа работы цифровых подписей.

За изобретением этих первых алгоритмов с открытыми ключами стоит сложная и интригующая история. Диффи и Хеллман действительно были первыми, кто опубликовал носящий их имя алгоритм в 1976 году. Райвест, Шамир и Адлеман действительно первыми опубликовали алгоритм RSA в 1978 году. Но это не всё! Позднее выяснилось, что Британскому правительству подобные системы были известны уже несколько лет. К несчастью для математиков, придумавших предвестников алгоритмов Диффи-Хеллмана и RSA, они работали в правительственной лаборатории по связи GCHQ. Их работы были засекречены, и гриф был снят только в 1997 году.

RSA, протокол Диффи-Хеллмана и другие криптосистемы с открытым ключом – не просто остроумные идеи. Из них выросли коммерческие технологии и стандарты Интернета, имеющие неоценимое значение для промышленности и отдельных людей. Подавляющее большинство сделок в сети, которые мы совершаем ежедневно, были бы невозможны без криптографии с открытым ключом. Изобретатели RSA запатентовали свою систему в 1970-е годы, и срок их патента истек только в конце 2000 года. В тот день в мюзик-холле Great American в Сан-Франциско состоялся торжественный вечер – наверное, в ознаменование того факта, что криптография с открытым ключом пришла, чтобы остаться навсегда.

ГЛАВА 5. Коды, исправляющие ошибки: ошибки, которые исправляются сами собой

Но одно дело – показать человеку, что он заблуждается, а другое дело – заставить его усвоить истину.

— Джон Локк «*Опыт о человеческом разумении*» (1690)[1]

Нам уже стало привычно обращаться к компьютеру, когда в том возникает необходимость. Но Ричарду Хэммингу, работавшему исследователем в лабораториях Bell Telephone Company в 1940-х годах, не так повезло: на нужном ему компьютере работали сотрудники других отделов, а в его распоряжении он оказывался только по выходным. Представьте себе досаду, которую он испытывал, когда компьютер раз за разом отказывал из-за ошибок при чтении своих же данных. Вот что сам Хэмминг писал по этому поводу:

> *Два выходных подряд я приходил и обнаруживал, что все мои данные выгружены и ровным счетом ничего не сделано. Я был в ярости, потому что мне нужны были ответы, а два выходных оказались потеряны зря. Тогда я сказал себе: «Черт, если машина может обнаружить ошибку, то что мешает ей определить, где эта ошибка произошла, и исправить ее?».*

Мало найдется ситуаций, когда нужда с такой очевидностью оказывалась матерью изобретения. Вскоре Хэмминг создал первый *код, исправляющий ошибки*, – алгоритм, который как по волшебству обнаруживает и исправляет ошибки в компьютерных данных. Без таких кодов наши компьютеры и системы связи работали бы многократно медленнее, были бы куда менее мощными и надежными, чем сейчас.

[1] Перевод А.Н. Савина.

Нужда в обнаружении и исправлении ошибок

Компьютеры решают три фундаментальные задачи. Самая важная – вычисления. Это означает, что компьютеру подаются на вход какие-то данные, а он преобразовывает их для получения полезного ответа. Но умение вычислять ответы было бы практически бесполезно без двух других задач, решаемых компьютерами: хранение и передача данных. (Компьютеры хранят данные по большей части в памяти и на дисках. А передают их как правило через Интернет.) Чтобы вам стало понятнее, попробуйте представить себе компьютер, который не умеет ни хранить, ни передавать информацию. Ну да, вы смогли бы проделать какие-то сложные вычисления (например, подготовить замысловатую электронную таблицу с деталями бюджета компании), но не сумели бы ни передать результаты коллеге, ни даже сохранить их, чтобы продолжить работу позже. Таким образом, передача и хранение данных – неотъемлемая черта современных компьютеров.

Но с передачей и хранением данных связана серьезнейшая проблема: данные должны быть *абсолютно правильны* – во многих случаях даже малейшая ошибка может сделать данные бесполезными. Людям необходимость хранить и передавать информацию без ошибок тоже знакома. Например, когда вы записываете чей-то номер телефона, важно записать все цифры правильно и в нужном порядке. Стоит допустить ошибку хотя бы в одной цифре, как телефон становится практически бесполезным. А бывает так, что ошибка *хуже*, чем бесполезна. Например, ошибка в файле с компьютерной программой может привести к тому, что программа «грохнется» или сделает не то, что должна делать. (Она может даже удалить какие-то важные файлы или «грохнуться» раньше, чем вы успели сохранить проделанную работу.) А ошибки в финансовых данных, хранящихся в компьютере, могут привести к материальным потерям (если, например, вы думаете, что покупали акции по $5,34, а на самом деле заплатили $8,34).

Но людям приходится хранить сравнительно немного не допускающей ошибок информации, и не так уж трудно избежать ошибок, внимательно проверив данные, которые считаются важными, – номера банковских счетов, пароли, адреса электронной почты и т. п. Напротив, объем информации, которую компьютеры должны хранить и передавать без единой ошибки, поистине огромен. Чтобы составить представление об этой величине, представьте, что обладаете каким-то вычислительным устройством с памятью емкостью 100 гигабайт

(на момент написания этой книги диски такой емкости были типичны для дешевых ноутбуков). 100 гигабайт – это эквивалент примерно 15 миллионов страниц текста. Так что даже если компьютерная система хранения делает всего одну ошибку на миллион страниц, в заполненном данными устройстве будет (в среднем) 15 ошибок. То же рассуждение относится и к передаче данных: если вы скачиваете программу размером 20 мегабайт, и компьютер неправильно интерпретирует всего один из миллиона полученных знаков, в скачанной программе, вероятно, будет больше 20 ошибок – и каждая из них может привести к краху в самый неподходящий момент.

Мораль сей басни заключается в том, что для компьютера точность в 99,9999 % случаев абсолютно недостаточна. Компьютеры должны хранить и передавать миллиарды единиц информации без единой ошибки. Но компьютеры, как и любые другие устройства, сталкиваются с проблемами коммуникации. Самым наглядным примером являются телефоны: очевидно, что они передают информацию неточно, иначе не было бы искажений, статических помех и других шумов во время разговора. Но от этого страдает не только телефонная сеть: на электрические провода тоже влияют самые разные возмущения; беспроводная связь испытывает помехи постоянно, а физические носители – жесткие диски, CD, DVD – можно поцарапать или сломать, даже простая пылинка или еще какое-то физическое препятствие может привести к ошибке при чтении. И как можно надеяться на частоту ошибок, меньшую 1 на много миллиардов, при таких очевидных дефектах коммуникации? В этой главе мы рассмотрим идеи, с помощью которых информатика справляется с этой трудностью. Оказывается, что если использовать правильные трюки, то даже при исключительно ненадежных каналах связи данные можно передать с поразительно низкой частотой ошибок – настолько низкой, что на практике можно считать, что ошибок нет вовсе.

Трюк с повторением

Основополагающий трюк, благодаря которому возможна надежная передача по ненадежному каналу, всем нам хорошо знаком: чтобы гарантировать правильную передачу информации, ее нужно передать несколько раз. Если вам диктуют номер телефона или банковского счета по телефону, а связь плохая, то вы, скорее всего, попросите собеседника хотя бы один раз повторить, чтобы убедиться, что все поняли правильно.

Компьютеры поступают точно так же. Предположим, что компьютер в банке пытается передать баланс вашего счета через Интернет. Баланс равен $5213,75, но, к несчастью, сеть не очень надежна, поэтому с вероятностью 20 % любая цифра может оказаться измененной. Так, при первой попытке был передан баланс $5293,75. Понятно, что вы не можете узнать, правильно это или нет. *Возможно*, все цифры правильны, а, возможно, в какой-то есть ошибка – кто знает? Но, применив трюк с повторением, вы сумеете почти наверняка узнать истинный баланс. Допустим, вы просите передать баланс пять раз и получаете следующие ответы:

```
передача 1: $ 5 2 9 3 . 7 5
передача 2: $ 5 2 1 3 . 7 5
передача 3: $ 5 2 1 3 . 1 1
передача 4: $ 5 4 4 3 . 7 5
передача 5: $ 7 2 1 8 . 7 5
```

Обратите внимание, что есть передачи, где неправильны несколько цифр, а в одной (с номером 2) вообще нет ошибок. Существенно, что у вас нет никакого способа выяснить, где имеются ошибки, поэтому вы не можете взять передачу 2 и объявить, что именно она правильная. Вместо этого приходится исследовать каждую цифру отдельно. Мы смотрим на первую цифру во всех передачах и выбираем значение, которое встречается чаще других. Приведем ту же табличку, что и выше, но добавим к ней строку с наиболее часто встречающимися цифрами.

```
передача 1:                        $ 5 2 9 3 . 7 5
передача 2:                        $ 5 2 1 3 . 7 5
передача 3:                        $ 5 2 1 3 . 1 1
передача 4:                        $ 5 4 4 3 . 7 5
передача 5:                        $ 7 2 1 8 . 7 5
самая часто встречающаяся цифра: $ 5 2 1 3 . 7 5
```

Рассмотрим несколько примеров, чтобы прояснить эту идею до конца. Исследуя первую цифру во всех передачах, мы видим, что в передачах 1–4 она была равна 5, а в передаче 5 – 7. Иными словами, четыре передачи сказали «5», и только одна сказала «7». Поэтому, хотя полной уверенности и нет, но *скорее всего* первая цифра баланса равна 5. Перейдем ко второй цифре: 2 встретилась четыре раза, а 4 – только один раз, т. е. наиболее вероятная цифра – 2. Третья цифра интереснее, поскольку тут есть три возможности: 1 встретилась 3 раза, 9 – один раз, 4 – один раз. Но, применяя тот же принцип, мы заключаем, что наиболее вероятно значение 1. Повторяя то же рассуждение

для всех цифр, мы приходим к окончательной гипотезе о величине баланса: $5213,75. В нашем случае это правильный ответ.

Что ж, это было несложно. Но можно ли сказать, что мы уже решили задачу? В каком-то смысле, да. Но возникает неудовлетворенность по двум причинам. Во-первых, частота ошибок в этом канале связи составляла всего 20 %, а иногда компьютерам приходится общаться по каналам куда худшего качества. Во-вторых, и это, пожалуй, самое важное: наша гипотеза – всего лишь догадка о наиболее вероятной величине баланса. К счастью, оба возражения снимаются очень легко: мы попросту увеличиваем количество повторных передач, пока не достигнем желаемой надежности.

Пусть, например, в предыдущем примере частота ошибок составляет 50 %, а не 20 %. Вы могли бы попросить банк передать баланс 1000 раз вместо 5. Сосредоточимся только на первой цифре, поскольку с остальными дело обстоит аналогично. Поскольку частота ошибок равна 50 %, то примерно в половине случаев цифра передана правильно, как 5, а в половине заменена каким-то другим случайным значением. Следовательно, цифра 5 встретится приблизительно 500 раз, а остальные (0–4 и 6–9) – приблизительно по 50 раз каждая. Математик может подсчитать, с какой вероятностью какая-нибудь из этих цифр встретится чаще, чем 5; оказывается, что даже если бы мы таким способом передавали баланс каждую секунду, то чтобы предположение о его величине оказалось неверным, должно было бы пройти много триллионов лет. Мораль сей басни заключается в том, что, повторяя передачу ненадежного сообщения достаточно часто, его можно сделать настолько надежным, насколько необходимо. (Мы предполагали, что ошибки происходят *случайно*. Но если бы злонамеренный противник намеренно вмешался в передачу и выбирал, какие ошибки создавать, то трюк с повторением оказался бы гораздо более уязвимым. Впрочем, некоторые коды, с которыми мы познакомимся ниже, хорошо работают даже в условиях такой атаки.)

Итак, применяя трюк с повторением, задачу ненадежной связи можно решить, и вероятность ошибки практически сводится к нулю. К сожалению, для современных компьютерных систем этот трюк не подходит. Если речь идет о сравнительно небольшом объеме данных, например о балансе, то его повторная передача 1000 раз обходится не слишком дорого, но очевидно, что передавать 1000 копий большой программы (мегабайт этак на 200) при скачивании практически нереально. Ясно, что компьютерам нужно что-то более совершенное, чем трюк с повторением.

Трюк с избыточностью

Хотя в компьютерах трюк с повторением в том виде, в каком он описан выше, не используется, мы рассмотрели его первым, чтобы вы поняли общий принцип организации надежной связи. Принцип этот состоит в том, что одно исходное сообщение послать недостаточно, для повышения надежности нужно передать еще что-то. В случае трюка с повторением это «что-то» – дополнительные копии исходного сообщения. Но, как выясняется, повысить надежность можно, передавая и другие виды дополнительной информации, – и этих видов не так уж мало. В информатике эту дополнительную информацию называют «избыточностью». Иногда избыточность добавляется к исходному сообщению. С такой техникой «дописывания» мы познакомимся ниже, когда будем рассматривать трюк с контрольной суммой. Но сначала рассмотрим другой вид избыточности, когда исходное сообщение заменяется другим, более длинным – «избыточным». Получив более длинное сообщение, мы затем можем восстановить из него исходное, даже если в процессе передачи по плохому каналу оно было искажено. Эту технику мы будем называть *трюком с избыточностью*.

Поясним на примере. Вспомните, что мы пытались передать банковский баланс $5213,75 по ненадежному коммуникационному каналу, который случайным образом изменяет 20 % цифр. Вместо того чтобы передавать сообщение «$5213,75», давайте попробуем преобразовать его в более длинное (и, стало быть, «избыточное»), но содержащее ту же информацию. В данном случае заменим каждую цифру соответствующим английским числительным:

```
five two one three point seven five
```

Снова предположим, что примерно 20 % знаков в этом сообщении случайно изменены из-за плохого качества канала связи. Полученное сообщение могло бы выглядеть так:

```
fiqe kwo one thrxp point sivpn fivq
```

Выглядит непривычно, но я думаю, вы согласитесь, что всякий, кто знает английский язык, догадается, что искаженное сообщение представляет правильный баланс $5213,75.

Главное здесь то, что благодаря *избыточности* оказалось возможно с высокой степенью надежности обнаружить и исправить единичное изменение сообщения. Если я скажу вам, что строчка «fiqe» представляет английское числительное, то вы с полной уверенностью

сможете утверждать, что исходным сообщением была строчка «five», потому что нет другого английского числительного, которое можно было бы получить из «fiqe» изменением одного знака. Напротив, если я скажу вам что цифры «367» представляют некоторое число, но одна цифра изменена, то вы никак не сможете догадаться, каким было исходное число, потому что в этом сообщении нет избыточности.

И хотя пока еще точно не ясно, как работает избыточность, мы уже понимаем, что она как-то связана с *удлинением* сообщения и что каждая часть сообщения должна отвечать определенному *образцу*. Тогда мы сначала сможем обнаружить любое единичное изменение (поскольку сообщение не отвечает образцу), а затем исправить ошибку (сделав так, чтобы сообщение отвечало образцу). В информатике такие известные образцы называются «кодовыми словами». В нашем случае кодовые слова – это просто английские числительные: «one», «two», «three» и так далее.

Теперь настало время объяснить, как на самом деле работает трюк с избыточностью. Сообщения состоят из так называемых «символов». В нашем примере символами являются цифры от 0 до 9 (для простоты мы игнорируем знак доллара и десятичную точку). Каждому символу сопоставляется кодовое слово. В нашем примере символу 1 сопоставляется кодовое слово «one», символу 2 – кодовое слово «two» и т. д.

Для передачи сообщения мы сначала берем каждый символ и заменяем его соответствующим кодовым словом. Затем преобразованное таким образом сообщение передается по ненадежному каналу связи. Получатель анализирует каждую часть сообщения и проверяет, является ли она допустимым кодовым словом. Если это так (например, «five»), то мы просто производим обратную замену кодового слова на соответствующий ему символ (например, 5). В противном случае (например, получено «fiqe»), мы ищем кодовое слово, наиболее близкое к полученному (в данном случае «five»), и заменяем *это* кодовое слово соответствующим ему символом (в данном случае 5). Примеры приведены на следующем рисунке.

Кодирование

```
1 → one
2 → two
3 → three
4 → four
5 → five
```

Декодирование

```
five → 5 (точное соответствие)
fiqe → 5 (ближайшее соответствие)
twe → 2 (ближайшее соответствие)
```

Код, в котором цифры заменены английскими числительными.

Вот, собственно, и всё. Компьютеры используют этот трюк с избыточностью всегда, когда сохраняют и передают информацию. Математики придумали более хитроумные кодовые слова, чем английские числительные, но в остальном принцип обеспечения надежной связи тот же самый. На следующем рисунке показан реальный пример. Этот код в информатике называется кодом Хэмминга (7,4), и это лишь один из кодов, изобретенных Ричардом Хэммингом в Bell Labs в 1947 году в ответ на еженедельные «падения» компьютера. (Из-за требования Bell Labs запатентовать коды Хэмминг опубликовал их лишь спустя три года, в 1950.) Очевидное отличие этого кода от предыдущего состоит в том, что все делается с помощью нулей и единиц. Поскольку при хранении или передаче компьютер преобразует любые данные в цепочки нулей и единиц, всякий код, применяемый на практике, оперирует только этими двумя цифрами.

Кодирование

```
0000 → 0000000
0001 → 0001011
0010 → 0010111
0011 → 0011100
0100 → 0100110
```

Декодирование

```
0010111 → 0010 (точное соответствие)
0010110 → 0010 (ближайшее соответствие)
1011100 → 0011 (ближайшее соответствие)
```

Реальный код, применяемый в компьютерах. В информатике он называется кодом Хэмминга (7,4). В разделе «Кодирование» перечислены только пять из 16 возможных 4-значных цепочек. Остальным тоже соответствуют кодовые слова, но здесь они для краткости опущены.

Но если отвлечься от этого различия, то все остальное работает в точности, как раньше. При кодировании в каждую группу из четырех цифр добавляется избыточность, так что получается кодовое слово из семи цифр. При декодировании мы сначала ищем точное соответствие семи полученным цифрам, а если такого нет, то ближайшее соответствие. Однако этот конкретный код придуман так хитро, что любую ошибку в одном из 7 знаков можно однозначно исправить. За проектированием кодов, обладающих таким свойством, стоит красивая математика, но здесь мы не будем вдаваться в детали.

Поясним еще раз, почему трюк с избыточностью предпочтительнее трюка с повторением. Основная причина – их сравнительная *стоимость*. В информатике стоимость схем исправления ошибок измеряют в терминах «накладных расходов», т. е. объема дополнительной информации, которую нужно передать, чтобы сообщение можно было принять правильно. Накладные расходы трюка с повторением

огромны, поскольку приходится передавать полные копии сообщения. Накладные расходы трюка с избыточностью зависят используемого набора кодовых слов. В примере, где использовались английские числительные, длина избыточного сообщения составляла 35 знаков, тогда как исходного – только 6 цифр, так что накладные расходы этой схемы весьма велики. Но математики придумали наборы кодовых слов с гораздо меньшей избыточностью и при этом такие, что вероятность пропустить ошибку крайне мала. Низкие накладные расходы этих кодовых слов и есть та причина, по которой в компьютерах применяется трюк с избыточностью, а не с повторением.

До сих пор мы во всех примерах предполагали использование кодов для *передачи* информации, но все сказанное равным образом относится и к задаче *хранения* информации. Чтобы обеспечить высочайшую надежность, которую мы видим, CD, DVD и жесткие диски опираются на коды, исправляющие ошибки.

Трюк с контрольной суммой

Пока что мы знакомились с методами, позволяющими как *обнаружить*, так и *исправить* ошибки в данных. Трюки с повторением и с избыточностью как раз на это и нацелены. Но существует и другой подход к задаче: забыть об *исправлении* ошибок, а сосредоточиться только на их *обнаружении*. (Живший в 17 веке философ Джон Локк отлично понимал разницу между обнаружением и исправлением ошибок, о чем свидетельствует эпиграф к этой главе.) Во многих приложениях простого обнаружения ошибки вполне достаточно, потому что, зная о наличии ошибки, можно затребовать другую копию данных. И делать так до тех пор, пока не придут данные без ошибок. Эта стратегия используется очень часто. Например, именно так передаются данные, когда вы подключаетесь к Интернету. Мы будем называть эту технику «трюком с контрольной суммой», а почему, скоро станет понятно.

Чтобы лучше разобраться в трюке с контрольной суммой, удобно считать, что сообщения целиком состоят из чисел. Это вполне реалистичное допущение, потому что компьютеры хранят всю информацию в числовом виде, а преобразуют числа в текст или изображения, только когда нужно представить ее человеку. Впрочем – и это важно отчетливо понимать – конкретный выбор символов, присутствующих в сообщении, не имеет значения для описываемой в этой главе техники. Иногда удобнее использовать числовые символы (цифры от 0 до 9), а иногда буквенные (буквы от *a* до *z*). В любом случае можно

договориться о том, как преобразовывать один набор символов в другой. Например, очевидное преобразование букв в числа выглядит так: $a \rightarrow 01$, $b \rightarrow 02$, ..., $z \rightarrow 26$. Поэтому неважно, будем ли мы изучать технику передачи на примере числовых или буквенных сообщений; впоследствии ее можно будет применить к сообщениям любого другого вида, выполнив предварительно простое преобразование символов.

Теперь нужно выяснить, что же такое контрольная сумма. Есть много вариантов контрольных сумм, но мы остановимся на той, что попроще, и назовем ее «простой контрольной суммой». Вычислить простую контрольную сумму числового сообщения действительно просто, проще быть не может: нужно сложить все цифры сообщения и отбросить все цифры результата, кроме последней. Вот эта последняя цифра и называется простой контрольной суммой. Рассмотрим пример. Пусть сообщение имеет вид:

```
4 6 7 5 6
```

Сумма всех его цифр равна 4 + 6 + 7 + 5 + 6 = 28, но мы оставляем от нее только последнюю цифру, так что простая контрольная сумма этого сообщения равна 8. А как используется контрольная сумма? Тоже просто: дописываем контрольную сумму в конец исходного сообщения, перед тем как отправлять его. Теперь получатель сможет снова вычислить контрольную сумму и сравнить ее с той, что вы отправили. Иными словами, он «контролирует сумму» сообщения, отсюда и название «контрольная сумма». Продолжим начатый пример. Простая контрольная сумма сообщения «46756» равна 8, и мы передаем ее вместе с сообщением:

```
4 6 7 5 6 8
```

Человек, получивший сообщение, должен знать, что вы применили трюк с контрольной суммой. В таком случае он понимает, что последняя цифра, 8, не является частью исходного сообщения, откладывает ее в сторонку и вычисляет контрольную сумму того, что осталось. Если при передаче сообщения не было ошибок, то он получит 4 + 6 + 7 + 5 + 6 = 28, оставит от него только последнюю цифру (8) и сравнит ее с той цифрой, которую раньше отложил в сторонку. Увидев, что они равны (а это так и есть), получатель делает вывод, что сообщение передано правильно. А что произойдет, если сообщение передано с ошибкой? Предположим, что вместо 7 была передана цифра 3. Тогда будет получено сообщение:

```
4 6 3 5 6 8
```

Мы откладываем в сторонку 8 для последующего сравнения, вычисляем контрольную сумму 4 + 6 + 3 + 5 + 6 = 24 и оставляем только последнюю цифру. Она *не* равна отложенной ранее цифре 8, поэтому мы точно знаем, что в процессе передачи сообщение было искажено. И тогда мы просим повторно передать сообщение, ждем получения новой копии, снова вычисляем и сравниваем контрольную сумму. Так продолжается до тех пор, пока не будет получено сообщение с правильной контрольной суммой.

Все это выглядит слишком хорошо, чтобы быть правдой. Вспомните про «накладные расходы» системы исправления ошибок – объем дополнительной информации, которую нужно передать вместе с самим сообщением. Получается, что мы придумали систему с фантастически низкими накладными расходами – каким бы длинным ни было сообщение, для обнаружения ошибки нужно добавить всего одну цифру (контрольную сумму)!

Увы, в жизни так не бывает. Проблема вот в чем: описанная выше контрольная сумма способна обнаружить не более *одной* ошибки в сообщении. Если ошибок две или больше, то простая контрольная сумма, возможно, их обнаружит, а, возможно, и нет. Рассмотрим несколько примеров:

		Контрольная сумма
Исходное сообщение	4 6 7 5 6	8
Сообщение с одной ошибкой	1 6 7 5 6	5
Сообщение с двумя ошибками	1 5 7 5 6	4
Сообщение с двумя (другими) ошибками	2 8 7 5 6	8

Исходное сообщение (46756) то же, что и прежде, и его контрольная сумма равна 8. В следующей строке показано сообщение с одной ошибкой (первая цифра равна 1 вместо 4), и его контрольная сумма оказалась равна 5. На самом деле, еще немного поэкспериментировав, вы убедитесь, что изменение любой *одной* цифры сообщения приводит к контрольной сумме, отличной от 8, так что единственную ошибку мы гарантированно обнаружим. Нетрудно строго доказать, что это действительно так: если в сообщении есть только одна ошибка, то простая контрольная сумма ее обязательно обнаружит.

В следующей строке таблицы мы видим сообщение с двумя ошибками: изменены две первые цифры. В этом случае контрольная сумма оказывается равна 4. И поскольку 4 отличается от исходной контрольной суммы – 8, то получатель сообщения знает, что ошибка была. Но «облом» происходит в последней строке таблицы. Здесь тоже показано сообщение с двумя ошибками и тоже в двух первых цифрах. Вот только контрольная сумма этого сообщения с двумя ошибками равна 8 – как и у исходного сообщения! Таким образом, получатель не сможет обнаружить, что сообщение передано с ошибками.

По счастью, проблему можно решить, немного усовершенствовав наш трюк с контрольной суммой. Для начала определим новый вид контрольной суммы. Назовем ее «лестничной» контрольной суммой, потому что процесс ее вычисления напоминает подъем по лестнице. Представьте, что вы находитесь у подножия лестницы, ступеньки которой пронумерованы числами 1, 2, 3 и т. д. Чтобы вычислить лестничную контрольную сумму, вы складываете цифры, как и раньше, только каждая цифра умножается на номер ступеньки, на которой вы стоите, а после прибавления каждой цифры вы поднимаетесь на одну ступеньку вверх. В конце все цифры, кроме последней, отбрасываются, как и в случае простой контрольной суммы. Так, для того же сообщения

```
4 6 7 5 6
```

что и раньше, при вычислении лестничной контрольной суммы мы сначала вычисляем выражение:

```
(1 × 4) + (2 × 6) + (3 × 7) + (4 × 5) + (5 × 6)
                                  = 4 + 12 + 21 + 20 + 30
                                  = 87
```

а затем отбрасываем все цифры, кроме последней – 7. Таким образом, лестничная контрольная сумма сообщения «46756» равна 7.

И в чем смысл сих манипуляций? Оказывается, что если включить *обе* контрольные суммы – простую и лестничную, то можно гарантированно обнаружить любые две ошибки в сообщении. Поэтому модифицированный трюк с контрольной суммой заключается в том, чтобы дописать в конец исходного сообщения две дополнительные цифры: сначала простую контрольную сумму, а потом лестничную. Например, сообщение «46756» нужно было бы передать в виде:

```
4 6 7 5 6 8 7
```

Получатель сообщения, как и раньше, должен заранее знать, какой трюк используется. Если ему это известно, то он сможет без труда проверить наличие ошибок, как и в трюке с простой контрольной суммой. Только теперь нужно отложить в сторонку две последние цифры (8 – простую контрольную сумму и 7 – лестничную контрольную сумму). Затем он вычисляет простую и лестничную контрольную сумму остатка сообщения (46756) – получается соответственно 8 и 7. Если обе вычисленные контрольные суммы совпадают с переданными (в данном случае это так), то сообщение либо правильно, либо содержит не менее трех ошибок.

В таблице ниже демонстрируется практическое применение этой схемы. Таблица отличается от предыдущей только наличием лестничной контрольной суммы в каждой строке и добавлением еще одной строки с примером. Мы видим, что если в сообщении имеется одна ошибка, то и простая, и лестничная контрольная сумма отличаются (5 вместо 8 и 4 вместо 7). Если ошибки две, то может случиться, что отличаются обе контрольные суммы, как в третьей строке таблицы (4 вместо 8 и 2 вместо 7). Но, как мы уже убедились, бывают случаи, когда при наличии двух ошибок простая контрольная сумма не изменяется. Пример показан в четвертой строке, где простая контрольная сумма по-прежнему равна 8. Но поскольку лестничная контрольная сумма отличается от исходной (9 вместо 7), то мы все равно знаем, что ошибка имеется. А в последней строке показана противоположная ситуация: две ошибки, при которых простая контрольная сумма изменилась (9 вместо 8), а лестничная осталась прежней (7). Однако же ошибка все равно обнаруживается, потому что хотя бы одна из контрольных сумм отличается от исходной. И хотя доказать это не очень просто, описанный факт не является случайностью: оказывается, что этот трюк всегда позволяет обнаружить ошибки, если их не больше двух.

		Простая и лестничная контрольная сумма
Исходное сообщение	4 6 7 5 6	8 7
Сообщение с одной ошибкой	1 6 7 5 6	5 4
Сообщение с двумя ошибками	1 5 7 5 6	4 2
Сообщение с двумя (другими) ошибками	2 8 7 5 6	8 9
Сообщение с двумя (снова другими) ошибками	6 5 7 5 6	9 7

Основной принцип мы поняли, но следует знать, что описанный трюк с контрольной суммой работает только для сравнительно коротких сообщений (менее 10 цифр). Однако аналогичные идеи применимы и к более длинным сообщениям. Можно определить алгоритмы вычисления контрольных сумм в виде последовательностей простых операций – сложения цифр, умножения цифр на «лестницы» разной формы, перестановку некоторых цифр. На словах это звучит довольно сложно, но компьютеры способы вычислять контрольные суммы с поразительной скоростью, и это оказывается чрезвычайно полезным на практике методом обнаружения ошибок в сообщениях.

В описанном выше трюке вычисляются только две цифры контрольной суммы (простая и лестничная), в настоящих же контрольных суммах цифр гораздо больше – иногда до 150 (далее в этой главе я всюду имею в виду десять *десятичных* цифр, 0 – 9, а не две *двоичных* – 0 и 1, хотя последние куда более употребительны при передаче информации между компьютерами). Важно то, что количество цифр в контрольной сумме (будь то 2, как в примере выше, или 150, как в некоторых реальных контрольных суммах) *фиксировано*. Впрочем, это не мешает вычислять контрольные суммы сообщений любой длины. Если сообщение очень длинное, то даже сравнительно большая контрольная сумма длиной 150 цифр составляет пренебрежимо малую часть самого сообщения. Пусть, к примеру, вы используете контрольную сумму длиной 150 цифр для проверки правильности 20-мегабайтного программного пакета, скачанного с какого-то сайта. Тогда контрольная сумма занимает менее одной тысячной процента от размера самого пакета. Уверен, вы согласитесь, что это приемлемый уровень накладных расходов! А математик скажет вам, что вероятность пропустить ошибку при использовании контрольной суммы такой длины настолько мала, что с любой практической точки зрения такое событие можно считать невозможным.

Как обычно, есть несколько важных технических деталей. Неправда, что любая контрольная сумма длиной 100 цифр обладает такой высокой отказоустойчивостью. Необходимы контрольные суммы особого вида, которые в информатике называются *криптографическими хэш-функциями* – особенно если изменения в сообщение внесены злонамеренным противником, а не просто появились в результате плохого качества канала. Эта проблема вполне реальна, потому что злобный хакер вполне может попробовать изменить вышеупомянутый 20-мегабайтный пакет, так что у него будет точно такая же 100-значная контрольная сумма, но совершенно другое наполнение, при-

званное перехватить управление вашим компьютером! Применение криптографических хэш-функций исключает такую возможность.

Трюк с указкой

Теперь, узнав о контрольных суммах, мы можем вернуться к первоначальной задаче обнаружения *и* исправления ошибок связи. Мы уже умеем это делать – неэффективно с помощью трюка с повторением и эффективно с помощью трюка с избыточностью. А возвращаемся мы потому, что до сих пор не знаем, как создавать кодовые слова, которые лежат в основе этого трюка. В одном из примеров мы использовали в этом качестве английские числительные, но такой набор кодовых слов хуже того, что реально применяется в компьютерах. Мы также видели пример кода Хэмминга, но не объяснили, откуда берутся используемые в нем кодовые слова.

Теперь нам предстоит узнать еще об одном наборе кодовых слов, которые можно использовать в трюке с избыточностью. Поскольку это очень частный случай трюка с избыточностью, позволяющий быстро указать место ошибки, назовем его «трюком с указкой».

Как и в трюке с контрольной суммой, мы будем работать с числовыми сообщениями, состоящими из цифр от 0 до 9, но помните, что это только для удобства. Очень просто преобразовать буквенное сообщение в числовое, поэтому описанная ниже техника применима к любым сообщениям.

Для простоты предположим, что сообщение содержит ровно 16 цифр, но и это допущение никак не ограничивает общность техники. Если имеется длинное сообщение, то его можно разбить на куски длиной по 16 цифр и работать с каждым куском отдельно. А если сообщение короче 16 цифр, то можно дополнить его до нужной длины нулями.

Первый шаг трюка с указкой состоит в том, чтобы расположить 16 цифр сообщения в виде квадрата, просматриваемого слева направо и сверху вниз. Если исходное сообщение имеет вид:

```
4 8 3 7 5 4 3 6 2 2 5 6 3 9 9 7
```

то после преобразования оно становится таким:

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

Затем вычислим простую контрольную сумму каждой строки и поместим ее справа:

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

Простые контрольные суммы вычисляются так же, как и раньше. Например, для получения контрольной суммы второй строки вычисляем 5 + 4 + 3 + 6 = 18 и оставляем последнюю цифру – 8.

Следующий шаг – вычислить простые контрольные суммы каждого столбца и поместить их внизу, в новой строке:

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

Ничего необычного в этих контрольных суммах тоже нет. Например, контрольная сумма третьего столбца вычисляется как 3 + 3 + 5 + 9 = 20, последняя цифра этого числа равна 0.

Следующий шаг трюка с указкой – упорядочить все цифры, так чтобы их можно было сохранять или передавать последовательно. Делается это очевидным образом – читаем цифры слева направо и сверху вниз. В результате получается такое сообщение из 24 цифр:

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

Теперь представьте, что вы получили сообщение, сформированное с помощью трюка с указкой. Как восстановить из него исходное сообщение и исправить ошибки, допущенные при передаче? Рассмотрим пример. Исходное 16-значное сообщение будет таким же, как и раньше, но для интереса предположим, что при передаче была допущена ошибка в одной цифре. В какой именно, я пока не скажу – мы выясним это с помощью трюка с указкой.

Итак, пусть *получено* такое 24-значное сообщение:

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

Первым делом расположим цифры в виде квадрата 5×5, понимая, что последний столбец и последняя строка содержат цифры контрольных сумм, переданные вместе с сообщением:

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

Затем вычислим простые контрольные суммы первых четырех цифр в каждой строке и в каждом столбце и запишем результаты в новой строке и столбце рядом с полученными значениями контрольных сумм:

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

Важно помнить, что теперь у нас есть два набора контрольных сумм: *полученный* и *вычисленный*. Как правило, эти наборы совпадут. На самом деле, если они совпали, то можно заключить, что с большой вероятностью сообщение правильно. Но если при передаче была ошибка, то некоторые вычисленные значения будут отличаться от полученных. В рассматриваемом примере таких отличий два: в третьей строке отличаются 5 и 0, а во втором столбце – 3 и 8. Несовпавшие контрольные суммы обведены рамочкой на рисунке ниже.

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

И вот ключевое озарение: положения отличающихся значений точно говорят, в какой цифре произошла ошибка передачи! Она *должна* находиться в третьей строке (потому что во всех остальных строках контрольная сумма правильна) и во втором столбце (потому что во всех остальных столбцах контрольная сумма правильна). И как показано на следующем рисунке, это оставляет нам только одну возможность – число 7, обведенное сплошной рамкой.

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

Но это еще не всё – мы нашли ошибку, но пока не исправили ее. К счастью, это нетрудно: нужно только заменить 7 таким числом, которое сделает обе контрольные суммы правильными. Мы видим, что контрольная сумма второго столбца должна быть равна 3, а на самом деле равна 8 – значит, ее нужно уменьшить на 5. После уменьшения 7 на 5 остается 2:

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

Можно даже проверить правильность этого изменения, посчитав контрольную сумму третьей строки; она теперь равна 5, что совпадает с полученной контрольной суммой. Ошибка найдена и исправлена! Последний, самый простой, шаг – восстановить исправленное исходное 16-значное сообщение из квадрата 5×5. Для этого нужно прочитать его сверху вниз и слева направо (конечно, пропуская последнюю строку и последний столбец). Получаем:

```
4 8 3 7 5 4 3 6 2 2 5 6 3 9 9 7
```

а это и есть то самое сообщение, с которого мы начали.

В информатике трюк с указкой называется «двухмерным контролем четности». Слово *четность* означает то же самое, что простая контрольная сумма, но относится к двоичным числам, с которыми обычно работают компьютеры. А *двухмерной* она называется, потому что сообщение представлено в виде таблицы с двумя измерениями (строки и столбцы). Двухмерный контроль четности раньше применялся в некоторых реальных компьютерных системах, но он не так эффективен, как некоторые другие трюки с избыточностью. Я решил объяснить именно этот способ, потому что его легко представить наглядно и на его примере видно, что для обнаружения и исправления ошибок необязательно нужна сложная математика, на которой основаны коды, употребляемые в современных компьютерных системах.

Обнаружение и исправление ошибок в реальном мире

Коды, исправляющие ошибки, появились в 1940-х годах, спустя не так уж много времени после рождения самого электронного компьютера. Оглядываясь назад, нетрудно понять, почему так произошло: ранние компьютеры были ненадежны, а их компоненты часто давали сбой. Но своими истинными корнями коды, исправляющие ошибки, уходят в еще более далекое прошлое и связаны с таким системами связи, как телеграф и телефон. Поэтому вовсе не удивительно, что два важнейших события, приведших к появлению кодов, исправляющих ошибки, произошли в исследовательских лабораториях компании Bell Telephone Company. Оба героя нашей истории, Клод Шеннон и Ричард Хэмминг, работали исследователями в Bell Labs. С Хэммингом мы уже встречались: именно его недовольство еженедельными сбоями компьютера компании привело прямиком к изобретению первых кодов, исправляющих ошибки, которые ныне известны как коды Хэмминга.

Однако коды, исправляющие ошибки, – это лишь часть более обширной дисциплины, которая называется *теорией информации*, и большинство историков науки считают, что начало этой дисциплине положила работа, опубликованная в 1948 году Клодом Шенноном. Эта выдающаяся работа, озаглавленная «Математическая теория связи», названа в биографии Шеннона «Великой хартией информационного века». Ирвинг Рид (один из изобретателей упоминаемых ниже кодов Рида-Соломона) писал о ней: «Немного найдется в этом

столетии работ, которые оказали большее влияние на науку и технику. В этой основополагающей статье... он перевернул практически все аспекты теории и практики связи». С чем связана такая высокая оценка? Шеннон математически доказал, что в принципе возможно достичь поразительно низкой частоты ошибок при передаче по зашумленному каналу. Но лишь спустя много десятилетий ученым удалось на практике приблизиться к теоретически максимальному коэффициенту качества связи, предсказанному Шенноном.

Кстати, Шеннон был человеком разносторонних интересов. Как один из четырех основных организаторов конференции по искусственному интеллекту в Дартмуте в 1956 году (мы вернемся к ней в конце главы 6) он активно участвовал в основании еще одной отрасли науки: искусственного интеллекта. Но на этом он не остановился. Он любил ездить на одноколесных велосипедах и построил парадоксальный моноцикл с эллиптическим (т. е. не круглым) колесом; это означает, что при движении моноцикла вперед ездок поднимается и опускается.

Благодаря работе Шеннона коды Хэмминга оказались в более широком теоретическом контексте и задали направление многочисленным позднейшим исследованиям. Таким образом, коды Хэмминга использовались в некоторых из самых первых компьютеров и до сих пор находят широкое применение в некоторых типах памяти. Еще одно важное семейство кодов получило название коды *Рида-Соломона*. Их можно настроить для исправления большого количества ошибок в кодовом слове (сравните с кодом Хэмминга (7,4) на стр. 80, который способен исправить только одну ошибку в 7-значном кодовом слове). Коды Рида-Соломона основаны на разделе математики, который называется «алгебры над конечным полем», но в первом приближении можно считать, что это комбинация лестничной контрольной суммы и двухмерного трюка с указкой. Эти коды используются при записи на CD, DVD и жесткие компьютерные диски.

Контрольные суммы также широко применяются на практике, особенно когда нужно обнаружить, а не исправить ошибки. Пожалуй, самый вездесущий пример – сетевой протокол Ethernet, который ныне используется едва ли не в каждом компьютере на планете. В Ethernet для обнаружения ошибок применяется контрольная сумма CRC-32. В самом распространенном протоколе Интернета – TCP (Transmission Control Protocol) контрольные суммы тоже присутствуют в каждой порции, или *пакете*, передаваемых данных. Пакеты с неправильными контрольными суммами просто отбрасываются, пос-

кольку TCP устроен так, что впоследствии автоматически передает их повторно. Программные пакеты, публикуемые в Интернете, также часто сопровождаются контрольными суммами, чтобы можно было убедиться в их правильности; для этой цели часто используются контрольные суммы MD5 и SHA-1. Обе являются криптографическими хэш-функциями, т. е. защищают не только от случайных ошибок при передаче, но от злонамеренного изменения программы. Контрольная сумма MD5 содержит примерно 40 цифр, в SHA-1 – примерно 50 цифр. В том же семействе есть и еще более отказоустойчивые контрольные суммы: SHA-256 (примерно 75 цифр) и SHA-512 (примерно 150 цифр).

Наука о кодах, обнаруживающих и исправляющих ошибки, продолжает развиваться. Начиная с 1990-х годов, пристальное внимание привлекают *коды с контролем четности низкой плотности*. Теперь они используются в самых разных приложениях: от спутникового телевидения до связи с зондами для исследования далекого космоса. В следующий выходной, когда будете наслаждаться передачей высокой четкости по спутниковому каналу, вспомните о восхитительной иронии судьбы: возможностью развлечься в выходной сегодня вы обязаны досаде, которую Ричард Хэмминг – тоже в выходной – испытал, сражаясь с одним из первых компьютеров.

ГЛАВА 6. Распознавание образов: обучение на опыте

Аналитическая Машина не притворяется, будто может что-то изобрести. Она может сделать что-то лишь при условии, что мы сумеем рассказать ей, как это сделать.

— Ада Лавлейс, из заметок 1843 года
об *Аналитической Машине*

В предыдущих главах мы рассматривали области, в которых возможности компьютеров намного превосходят возможности человека. Например, компьютер может зашифровать или расшифровать большой файл за секунду-другую, тогда как у человека на те же вычисления ушли бы многие годы. Можно привести еще более убедительный пример: подумайте, сколько времени понадобилось бы человеку, чтобы вручную вычислить PageRank миллиардов веб-страниц по алгоритму, описанному в главе 3. Эта задача настолько необозрима, что человеку она, скорее всего, вообще не по силам. А компьютеры в компаниях, занимающихся поиском, производят такие вычисления постоянно.

Напротив, в этой главе мы рассмотрим область, в которой у людей имеется естественное преимущество: *распознавание образов*. Это одна из дисциплин искусственного интеллекта, занимающаяся решением таких задач, как распознавание лиц, объектов, речи и рукописного текста. Вот несколько более конкретных примеров: понять, изображена ли на фотографии ваша сестра, или распознать город и область, написанные от руки на конверте. Таким образом, распознавание образов можно определить более общо: заставить компьютеры действовать «разумно», исходя из входных данных, характеризуемых высокой изменчивостью.

Слово «разумно» взято в кавычки по веской причине: вопрос о том, сможет ли компьютер когда-нибудь продемонстрировать настоящий

разум, далек от своего решения. В эпиграфе к этой главе приведена одна из первых оговорок в этом споре: замечание Ады Лавлейс, сделанное в 1843 году по поводу проекта одного из первых механических компьютеров, получившего название Аналитическая Машина. Леди Лавлейс иногда называют первым в мире программистом за ее глубокие мысли относительно Аналитической Машины. Но в этом высказывании она подчеркивает отсутствие у машин творческого начала: они должны рабски следовать инструкциям людей, которые их программируют. В наши дни у специалистов по информатике нет общего мнения о том, могут ли компьютеры в принципе мыслить. И этот спор грозит стать еще более запутанным, если в него втянутся философы, нейробиологи и теологи.

По счастью, нам здесь не нужно разрешать парадоксы машинного разума. Мы вполне можем заменить слово «разумный» словом «полезный». Таким образом, основная задача распознавания образов – взять данные с высокой степенью изменчивости, например фотографии разных лиц, сделанные при разном освещении, или образцы разных слов, написанные разными людьми, и сделать с ними что-то полезное. Без сомнения, люди умеют обрабатывать такие данные разумно: мы способны распознавать лица с поразительной точностью и разбирать практически любой почерк, который раньше никогда не видели. Как выясняется, в этом отношении компьютеры сильно уступают людям. Но появились кое-какие изобретательные алгоритмы, которые позволяют компьютерам добиваться хороших результатов при решении некоторых задач распознавания. В этой главе мы поговорим о трех таких алгоритмах: классификаторы по ближайшим соседям, деревья решений и искусственные нейронные сети. Но сначала нужно более научно поставить задачу, которую мы пытаемся решить.

В чем состоит задача?

На первый взгляд, задачи распознавания образом кажутся безнадежно разнообразными. Может ли компьютер использовать одни и те же средства для распознавания рукописного текста, лиц, речи и т. д.? Один из возможных ответов у нас прямо перед глазами (в буквальном смысле): наш мозг демонстрирует высочайшую скорость и точность при решении широкого круга задач распознавания. Но можно ли написать компьютерную программу, которая будет делать то же самое?

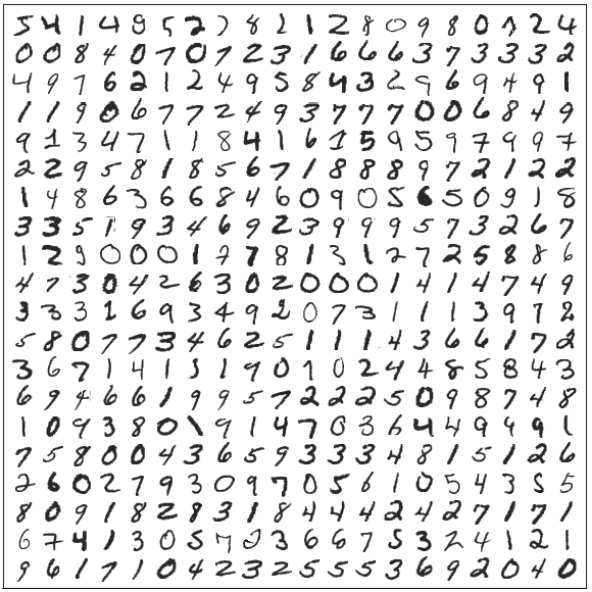
Прежде чем пускаться в обсуждение методов, которые можно было бы использовать в такой программе, нужно как-то унифици-

ровать приводящее в замешательство разнообразие ситуаций и определить одну задачу, которую мы пытаемся решить. Стандартный подход к распознаванию образов – рассматривать эту деятельность как задачу *классификации*. Предположим, что подлежащие обработке данные некоторым разумным образом разбиты на несколько порций, называемых *образцами*, и что каждый образец принадлежит одному из фиксированного набора возможных *классов*. Например, в задаче распознавания лиц образцом может быть фотография лица, а классами – личности лиц, которые система умеет распознавать. В некоторых задачах есть всего два класса. Типичный пример – медицинский диагноз болезни, когда имеются классы «здоров» и «болен», а образец состоит из совокупности результатов анализов конкретного пациента (артериальное давление, вес, рентгенограммы и, возможно, много других данных). Задача компьютера – обработать образцы данных, которые он никогда раньше не видел, и выполнить *классификацию* – отнести каждый образец к одному из возможных классов.

Для определенности сосредоточимся на какой-нибудь одной задаче распознавания образов. Возьмем задачу распознавания рукописных цифр. На рисунке ниже показаны типичные образцы данных. В задаче имеется десять классов: цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Требуется классифицировать образцы рукописных цифр, отнеся их к одному из этих классов. Эта проблема имеет большое практическое значение, потому что почтовые индексы в США и многих других странах состоят из цифр. Если компьютер способен быстро и точно распознавать индексы, то для сортировки почты можно использовать машины, которые работают гораздо эффективнее человека.

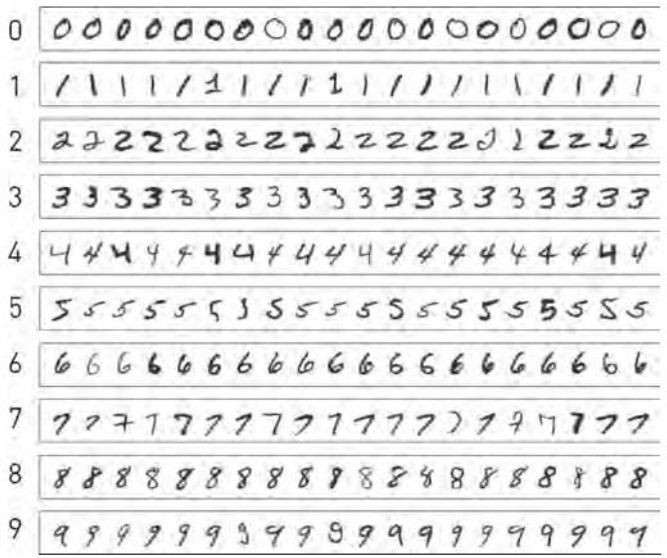
Очевидно, в компьютеры не встроены знания о том, как выглядят рукописные цифры. Да и в мозг человека такие знания не встроены: мы *учимся* распознавать цифры и другие рукописные тексты, сочетая явное обучение другими людьми и самостоятельное рассматривание примеров. Эти две стратегии (явное обучение и обучение на примерах) применяются также и при компьютерном распознавании образов. Однако оказывается, что явное обучение компьютеров эффективно только на самых простых задачах. Так, управление климатом в моем доме можно считать простой системой классификации. Образец состоит из текущей температуры и времени суток, а три возможных класса – «включить обогрев», «включить кондиционирование», «выключить все». Поскольку днем я нахожусь на работе, то запрограммировал систему, так чтобы в дневные часы она находилась в режиме «выключить все», а в остальное время работала в режиме «включить

обогрев», если температура низкая, или «включить кондиционирование», если высокая. Таким образом, программируя термостат, я в некотором смысле «обучил» систему выполнять классификацию по трем классам.



Большинство задач распознавания образов можно сформулировать как задачи классификации. В данном случае задача заключается в том, чтобы классифицировать рукописные цифры, отнеся их к одному из 10 классов: 0, 1, …, 9. Источник данных: база данных MNIST, LeCun et al. 1998.

К сожалению, никому не удалось явно «научить» компьютер решению более интересных задач классификации, например, распознаванию цифр, показанных на рисунке выше. Поэтому ученые обратились к другой стратегии: заставить компьютер автоматически «обучаться», как классифицировать образцы. Основная стратегия – предложить компьютеру достаточно большой набор *помеченных данных*: уже классифицированных образцов. На следующем рисунке показан пример помеченных данных для задачи распознавания рукописных цифр. Поскольку каждый образец снабжен меткой (своим классом), то компьютер может с помощью различных аналитических ухищрений выделить характеристики каждого класса. Если впоследствии предложить компьютеру непомеченный образец, то он сможет выдвинуть гипотезу о его классе, выбрав класс с наиболее похожими характеристиками.



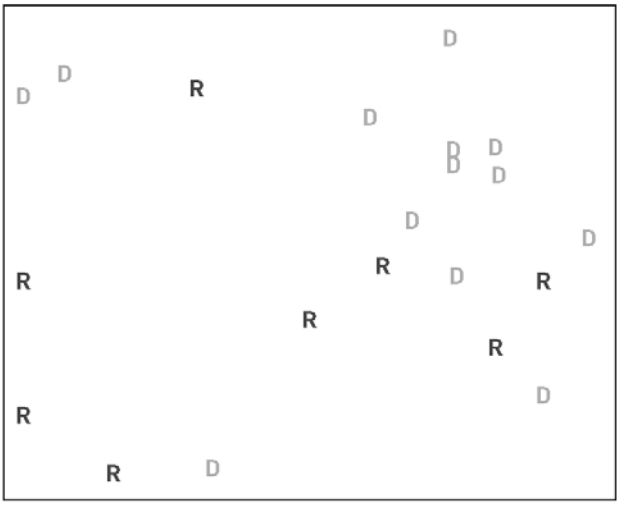
Для обучения классификатора компьютеру нужны помеченные данные. В данном случае каждый образец данных (рукописная цифра) снабжен меткой, содержащей одну из 10 возможных цифр. Метки показаны слева, а обучающие образцы справа. Источник данных: база данных MNIST, LeCun et al. 1998.

Процесс формирования характеристик каждого класса часто называют «обучением», а сами помеченные данные – «обучающими данными». Таким образом, по существу задача распознавания образов распадается на два этапа: этап обучения, когда компьютер узнает о классах на основе помеченных обучающих данных, и этап классификации, когда компьютер классифицирует новые, еще не помеченные образцы.

Трюк с ближайшими соседями

Вот интересная задача классификации: сумеете ли вы, зная только домашний адрес человека, предсказать, в фонд какой политической партии он сделает взнос? Очевидно, решить эту задачу абсолютно точно не сможет даже человек: одного адреса недостаточно для прогнозирования политических пристрастий. Тем не менее, хотелось бы обучить систему классификации на основе домашнего адреса предсказывать, кому человек окажет финансовую поддержку *с наибольшей вероятностью*.

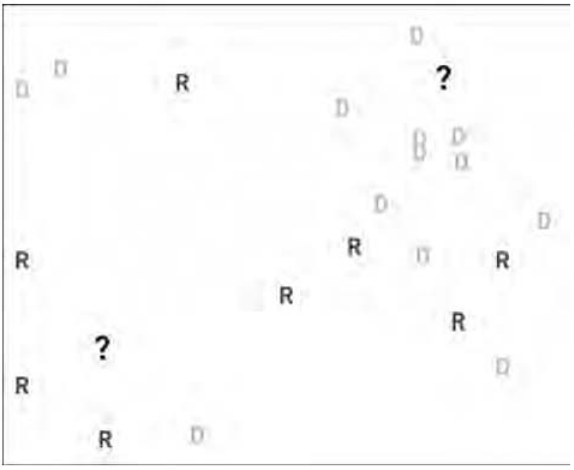
На следующем рисунке показано, какие обучающие данные можно было бы использовать для этой цели. Здесь мы видим карту реальных взносов жителей одного округа в Канзасе во время президентских выборов 2008 года в США. Для простоты улицы на карте не показаны, но фактическое местоположение каждого дома, хозяева которого делали взнос, отражено точно. Дома приверженцев демократов обозначены буквой «D», республиканцев – буквой «R».



Обучающие данные для предсказания политических пристрастий. «D» обозначает дом, владельцы которого перечислили взнос в фонд демократов, а «R» – дом, помогавший республиканцам. Источник данных: проект Fundrace, Huffington Post.

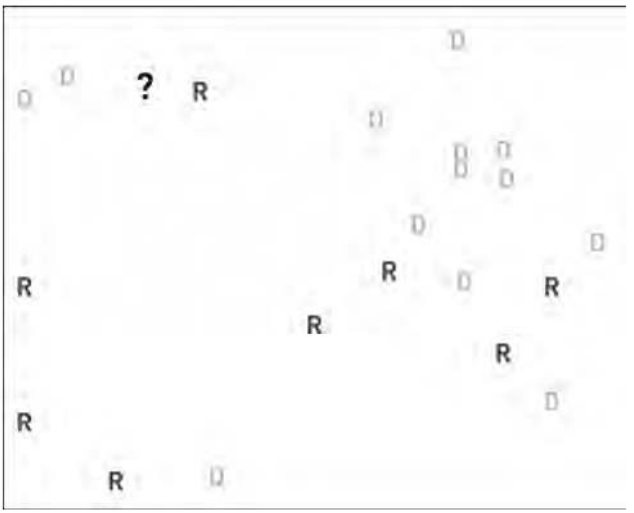
С обучающими данными понятно. Но что мы будем делать, получив новый образец, который нужно классифицировать в пользу демократов или республиканцев? На рисунке (стр. 100) изображена эта ситуация. Обучающие данные те же, что и раньше, но есть еще два дома, обозначенные воспросительными знаками. Сначала займемся тем, что выше. К какому классу вы отнесли бы этот вопросительный знак, глядя на карту и не прибегая ни к каким научным методам? Он окружен приверженцами демократов, поэтому класс «D» кажется вполне вероятным. А как насчет нижнего вопросительного знака? Нельзя сказать, что он прямо-таки окружен сторонниками республиканцев, но все же находится скорее на республиканской, чем на демократической территории, так что «R» представляется разумной гипотезой.

Хотите верьте, хотите нет, но мы сейчас придумали один из самых действенных и полезных методов распознавания образов – в информатике его называют *классификатором по ближайшим соседям*. В простейшей форме трюк с «ближайшим соседом» делает именно то, что сказано в названии. Получив неклассифицированный образец, мы первым делом ищем ближайший к нему классифицированный образец и в качестве предсказания берем класс этого образца. В применении к рисунку ниже это означает, что мы берем ближайшую к вопросительному знаку букву.



Классификация по ближайшему соседу.
Каждому вопросительному знаку назначается класс его ближайшего соседа.
Верхний вопросительный знак заменяется буквой «D», нижний – буквой «R».
Источник данных: проект Fundrace, Huffington Post.

Чуть более сложный вариант этого трюка называется «*K* ближайших соседей», где *K* – небольшое число, скажем, 3 или 5. В такой формулировке исследуются *K* ближайших соседей вопросительного знака и выбирается класс, наиболее популярный среди этих соседей. Результат применения этого метода показан на стр. 101. В данном случае самый ближний сосед вопросительного знака – сторонник республиканцев, поэтому трюк с ближайшим соседом в простейшей форме классифицировал бы вопросительный знак как «R». Но если взять трех ближайших соседей, то выяснится, что среди них два демократа и один республиканец, так что при таком выборе соседей демократы оказываются более популярны, и вопросительный знак классифицируется как «D».



Пример использования «K ближайших соседей». Если брать только одного ближайшего соседа, то вопросительный знак классифицируется как «R», а если трех ближайших соседей – то как «D». Источник данных: проект Fundrace, Huffington Post.

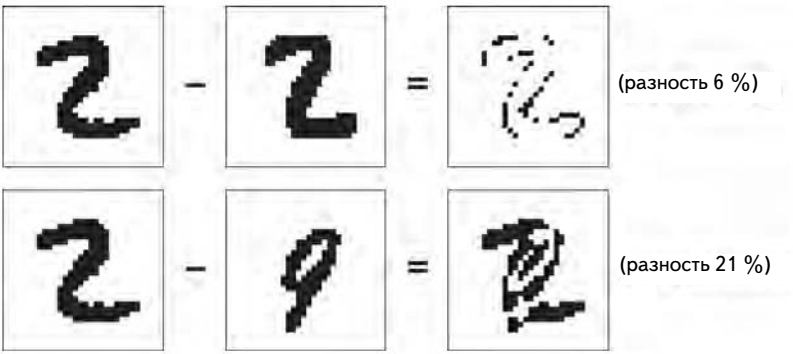
Так сколько же ближайших соседей брать? Ответ зависит от решаемой задачи. На практике обычно пробуют несколько значений и смотрят, какое оказалось лучше. Звучит как-то ненаучно, но такой подход отражает реальность – эффективные системы распознавания образов как правило представляют собой сплав математического понимания, экспертной оценки и практического опыта.

Различные виды «ближайших» соседей

До сих пор мы работали над задачей, в которой понятие «близости» образцов имеет простую и интуитивно понятную интерпретацию. Поскольку точки нанесены на карту, то для определения ближайших домов можно взять просто географическое расстояние. Но как быть, если образцами являются рукописные цифры, как на стр. 97? Необходим какой-то способ вычислить «расстояние» между двумя образцами рукописных цифр. На следующем рисунке показано, как это можно сделать.

Основная идея состоит в том, чтобы измерить разницу между изображениями цифр, а не географическое расстояние между ними. Разницу мы будем измерять в процентах, т. е. изображения, отличаю-

щиеся на 1 %, считаются очень близкими соседями, а отличающиеся на 99 % – очень далекими. На рисунке показано несколько примеров. (Как обычно бывает в распознавании образов, входные данные предварительно подвергнуты обработке. В данном случае все цифры приведены к единому масштабу и расположены по центру области.) В верхней строке мы видим два изображения рукописной цифры 2. Произведя операцию своеобразного «вычитания» изображений, мы получаем изображение, показанное справа, – белое всюду, кроме нескольких мест, где изображения различаются. С другой стороны, в нижней строке показаны результаты вычитания изображений разных цифр (2 и 9). Как видим, в разности справа черных пикселей гораздо больше, так как изображения различаются во многих местах. На самом деле, черным закрашено примерно 21 % площади результата, поэтому эти изображения нельзя назвать особенно близкими соседями.



Вычисление «расстояния» между двумя рукописными цифрами. Второе изображение в каждой строке вычитается из первого, справа показано результирующее изображение, из которого видно, насколько исходные различаются. Процентную долю черных пикселей в разности можно считать «расстоянием» между исходными изображениями. Источник данных: база данных MNIST, LeCun et al. 1998.

Теперь, зная, как вычислять «расстояние» между рукописными цифрами, легко построить систему их распознавания. Сначала нужно подготовить много обучающих данных – по типу примеров на стр. 98, только гораздо больше. В типичных системах такого рода количество помеченных образцов может достигать 100 000. Если после этого предъявить системе новую непомеченную цифру, то она сможет просмотреть все 100 000 образцов и найти тот, который является ближайшим соседом классифицируемого. Напомним, что, говоря «ближай-

ший сосед», мы имеем в виду наименьшее процентное расхождение, вычисленное, как показано на рисунке выше. Непомеченной цифре сопоставляется та же метка, что ее ближайшему соседу.

Оказывается, что система с так определенным «расстоянием» неплохо работает, распознавая цифры с точностью около 97 %. Ученые потратили массу усилий, чтобы найти более изощренные определения «расстояния до ближайшего соседа». При современном определении классификаторы рукописных цифр могут достигать точности свыше 99,5 %, что сравнимо с точностью гораздо более сложных систем распознавания образов с такими загадочными названиями, как «машины опорных векторов» и «сверточные нейронные сети». Трюк с ближайшими соседями – воистину чудо информатики, так как в нем элегантная простота сочетается с замечательной эффективностью.

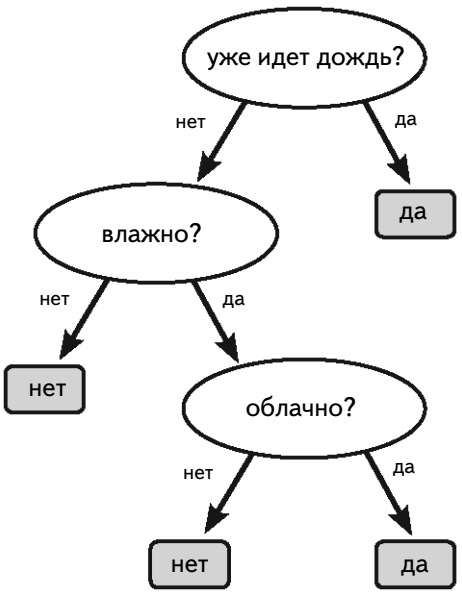
Ранее мы отметили, что в работе систем распознавания образов есть два этапа: *обучение* (или тренировка), в ходе которого производится обработка обучающих данных с целью выделения характеристик классов, и *классификация*, когда классифицируется новый, непомеченный образец. Так что же происходило на этапе обучения рассмотренного выше классификатора по ближайшему соседу? Выглядит так, будто мы взяли обучающие данные, ничему на них не научились, а сразу перешли к классификации. На самом деле, это особое свойство классификаторов по ближайшему соседу: они не нуждаются в явном этапе обучения. В следующем разделе мы рассмотрим классификатор другого типа, в котором обучение играет куда более важную роль.

Трюк с двадцатью вопросами: деревья решений

Игра в двадцать вопросов обладает для специалистов по информатике особой прелестью. В этой игре один участник загадывает объект, а остальные должны выяснить, что он загадал, задав не более 20 вопросов, на которые можно отвечать только да или нет. Продаются даже наладонные устройства для этой игры. И хотя в эту игру чаще всего играют с детьми, для взрослых она тоже оказывается на удивление интересна. Спустя всего несколько минут вы начинаете понимать, какие вопросы «хорошие», а какие – «плохие». Хорошие вопросы дают много «информации» (что бы под этим ни понимать), а плохие почти ничего не дают. Например, бессмысленно в самом начале игры спра-

шивать «сделано ли это из меди?», потому что если получен ответ «нет», то диапазон возможностей почти не сузился. Эти интуитивные представления о том, что такое хорошие и плохие вопросы, и лежит в основе завораживающей науки, которая называется теорией информации. И заодно – в основе простого и мощного метода распознавания образов, называемого *деревьями решений*.

Дерево решений по существу представляет собой предварительно составленный план игры в 20 вопросов. На следующем рисунке приведен тривиальный пример. Это дерево решений для ответа на вопрос, нужно ли брать с собой зонтик. Вначале мы находимся в вершине дерева и идем в ту или другую сторону в зависимости от ответа на вопрос. Дойдя до какого-нибудь прямоугольника внизу дерева, мы получаем окончательный ответ.



Вы не понимаете, как это связано с распознаванием образов и классификацией? Дело в том, что при наличии достаточного количества обучающих данных, дерево решений можно *научить* выполнять точную классификацию.

Рассмотрим пример малоизвестной, но чрезвычайно важной задачи о *веб-спаме*. Мы уже встречались с ней в главе 3, когда говорили о некоторых беззастенчивых владельцах сайтов, которые пытаются манипулировать алгоритмами ранжирования, искусственно создавая

много гиперссылок на определенные страницы. Стратегия, применяемая этими неразборчивыми в средствах веб-мастерами, заключается в создании страниц со специально подобранным содержимым, для людей бесполезных. Ниже на рисунке показана выдержка из спамной страницы. В этом бессмысленном тексте многократно повторяются поисковые слова, относящиеся к онлайновому обучению. Задача такого веб-спама – попытаться увеличить ранг тех сайтов, предлагающих онлайновое обучение, на которые такие страницы ссылаются.

управление человеческими ресурсами
обучение, дистанционное обучение через веб

Изучение языков волшебство онлайн mba сертификат самостоятельное изучение – онлайновое юридическое образование разного уровня, онлайновое послевузовское образование. Проживание консультации компьютерные курсы. Продолжающееся медицинское образование через веб конференция, новости онлайнового обучения в индиане, не требуется диплом об окончании колледжа онлайновый сервис программа информационных систем управления – компьютерные технологии онлайновые классы и mba изучение нового языка получение диплома в онлайне образование медсестры кредиты на дополнительное образование заочное дистанционное образование получение диплома горячее обслуживание ПК курс по технической поддержке.

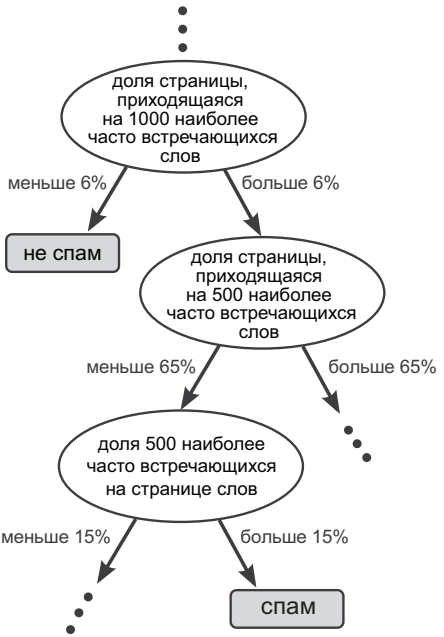
Фрагмент спамной веб-страницы[1]. Эта страница не содержит полезной для человека информации, ее единственная задача – манипуляция рангом при поиске в веб. Источник: Ntoulas et al. 2006

Естественно, поисковые системы тратят немало сил на выявление и устранение веб-спама. Это отличное применение методов распознавания образов: мы можем получить сколько угодно обучающих данных (в данном случае веб-страниц), пометить их признаками «спам» или «не спам» и обучить некий классификатор. Именно так и поступили исследователи из Microsoft Research в 2006 году. Как оказалось, лучше всего для этой цели подходит старый знакомец: дерево решений. На стр. 106 показана небольшая часть построенного ими дерева решений.

Конечно, в полном дереве решений много других атрибутов, та часть, что показана на рисунке, касается популярности слов на странице. Веб-спамеры любят включать множество популярных слов, чтобы увеличить свой ранг, поэтому если процентная доля популярных слов мала, то мала и вероятность спама. Этим объясняется пер-

[1] В оригинале спам, естественно, английский. Но и в русскоязычной части Интернета нет недостатка в такой прелести. *Прим. перев.*

вое решение в дереве, остальные следуют такой же логике. Точность этого дерева доходит до 90 % – от идеала далеко, но и это бесценное оружие в войне с веб-спамерами.



Часть дерева решений для выявления веб-спама. Точками обозначены части дерева, которые для простоты опущены. Источник: Ntoulas et al. 2006

Важно понимать не столько детали конкретного дерева, сколько тот факт, что все дерево было создано автоматически компьютерной программой на основе обучающих данных в виде 17 000 веб-страниц. Классификация этих страниц по признаку спам/не спам была выполнена человеком. Для построения хорошей системы распознавания образов иногда требуется приложить немало ручного труда, но это одноразовые затраты, которые окупаются многократно.

В отличие от рассмотренного выше классификатора по ближайшему соседу, этап обучения классификатору по дереву решений существует и занимает много времени. Как устроено обучение? Так же, как в игре в двадцать вопросов, немалую роль играет интуиция. Компьютер проверяет огромное количество возможных вариантов первого вопроса, пытаясь найти тот, который дает больше всего информации. Затем он делит обучающие примеры на две группы в зависимости от ответа

на первый вопрос и для каждой группы ищет второй наилучший вопрос. Так продолжается на всем пути движения вниз по дереву: всякий раз выбирается наилучший вопрос для множества оставшихся в данной точке дерева обучающих примеров. Если в какой-то точке остается «чистое» множество примеров, т. е. содержащее только спамные или только неспамные страницы, то компьютер перестает порождать новые вопросы, а вместо этого выводит ответ, соответствующий оставшимся страницам.

Подведем итоги. Этап обучения классификатора по дереву решений может быть сложным, но он полностью автоматический и выполняется только один раз. После этого мы получаем требуемое дерево решений, а этап классификация оказывается поразительно простым: как и в игре в двадцать вопросов, нужно спускаться вниз по дереву в зависимости от ответов на вопросы, пока не достигнем прямоугольника с окончательным решением. Как правило, хватает всего нескольких вопросов, так что этап классификации оказывается очень эффективным. Сравните с методом ближайшего соседа, когда на этапе обучения не пришлось прикладывать ни малейших усилий, зато на этапе классификации нужно было сравнивать предъявленный образец со всеми обучающими (а таких в случае задачи о рукописных цифрах 100 000).

В следующем разделе мы познакомимся с нейронными сетями: техникой распознавания образов, при которой этап обучения не только важен, но и построен по образу и подобию процесса обучения, свойственного человеку и животным.

Нейронные сети

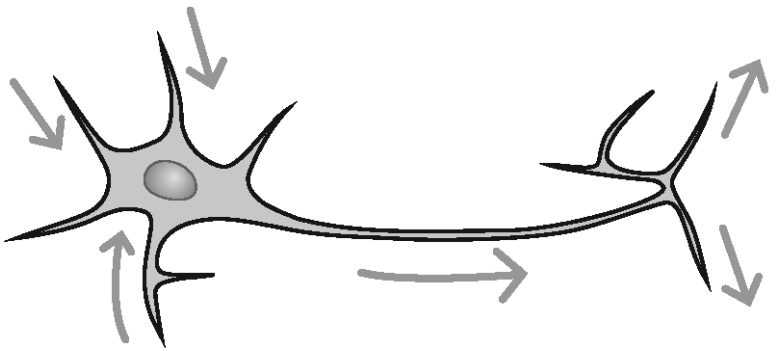
Поразительные способности человеческого мозга приводили в восхищение и вдохновляли многих ученых в области информатики еще со времен создания первых цифровых компьютеров. Одну из самых ранних дискуссий по поводу моделирования мозга с помощью компьютера инициировал Алан Тьюринг, британский ученый, проявивший себя выдающимся математиком, инженером и криптологом. Классическая работа Тьюринга «Вычислительные машины и разум», опубликованная в 1950 году, – самое знаменитое философское обсуждение вопроса о том, может ли компьютер притвориться человеком. В этой работе впервые описан научный способ оценки сходства между компьютерами и людьми, который теперь называется «тест Тьюринга». Но менее известна та часть этой статьи, в которой Тьюринг прямо

анализирует возможность моделирования человеческого мозга с помощью компьютера. Он считал, что для этого будет достаточно всего лишь нескольких гигабайтов памяти.

Прошло шестьдесят лет, и теперь все согласны, что Тьюринг сильно недооценил объем работы, необходимой для моделирования человеческого мозга. Но ученые тем не менее продолжают преследовать эту цель, предстающую во многих разных обличьях. Один из результатов в этой области – *искусственные нейронные сети*, или просто нейронные сети.

Биологические нейронные сети

Чтобы понять, что такое искусственная нейронная сеть, сначала нужно познакомиться с тем, как работают природные биологические нейронные сети. Мозг животного состоит из клеток, которые называются нейронами, и каждый нейрон связан со многими другими. По этим связям нейроны могут посылать электрические и химические сигналы. Некоторые связи предназначены для *приема* сигналов от других нейронов, остальные – для *передачи* сигналов (см. рисунок ниже).



Типичный живой нейрон. Электрические сигналы передаются в направлениях, показанных стрелками. Выходной сигнал передается, только если сумма входных сигналов достаточно велика.

Есть простой способ описания сигналов: в каждый момент времени нейрон может быть «пассивным» или «возбужденным». Пассивный нейрон не передает никаких сигналов, возбужденный с большой частотой посылает пачки сигналов во всех исходящих направлениях. Как нейрон решает, когда возбудиться? Все зависит от силы поступающих ему сигналов. Обычно, если суммарная величина всех входящих сиг-

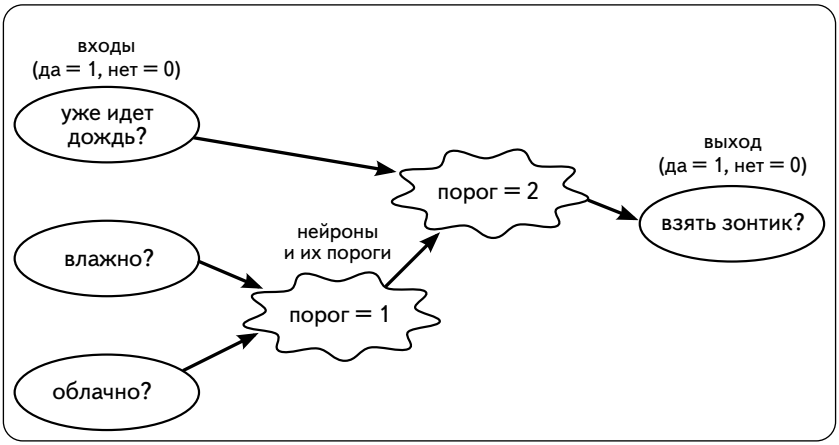
налов достаточно велика, нейрон переход в возбужденное состояние; в противном случае остается пассивным. Грубо говоря, нейрон «суммирует» все получаемые входы и возбуждается, если сумма достаточно большая. Это описание необходимо уточнить в одном важном отношении – существуют два типа входных сигналов: возбуждающие и тормозящие. Сила возбуждающих сигналов прибавляется к итогу, как мы только что описали, тогда как сила тормозящих сигналов вычитается из итога – то есть тормозящие сигналы препятствуют переходу нейрона в возбужденное состояние.

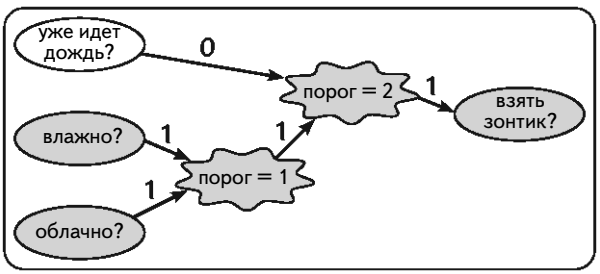
Нейронная сеть для задачи о зонтике

Искусственная нейронная сеть – это компьютерная модель, описывающая крохотную часть мозга в очень упрощенном виде. Начнем с обсуждения простейшего варианта нейронной сети, который годится для рассмотренной выше задачи о зонтике. Затем мы воспользуемся более изощренной нейронной сетью, чтобы решить так называемую «задачу о солнечных очках».

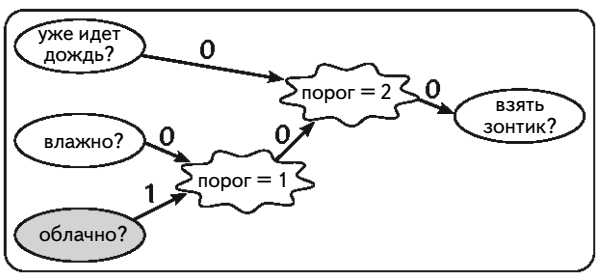
Каждому нейрону в нашей простой модели сопоставляется число, называемое его *порогом*. При прогоне модели каждый нейрон складывает получаемые входные сигналы. Если их сумма не меньше порога, то нейрон возбуждается, иначе остается пассивным. На следующей странице показана нейронная сеть для совсем простой задачи о зонтике. Слева мы видим три входа в сеть. Можете считать их аналогами сигналов, поступающих в мозг от органов чувств. Как наши глаза и уши генерируют электрические и химические сигналы, передаваемые в мозг с помощью нейронов, так и три показанных на рисунке входа посылают сигналы нейронам искусственной сети. Все три входа в этой сети возбуждающие. Каждый передает сигнал силы +1, если соответствующее ему условие истинно. Например, если на улице облачно, то вход с меткой «облачно?» передает возбуждающий сигнал силы +1; в противном случае он не передает ничего, что эквивалентно передаче сигнала силы 0.

Если оставить в стороне входы и выходы, то эта сеть состоит всего из двух нейронов с разными порогами. Нейрон, для которого входами являются влажность и облачность, возбуждается, только если активны оба его входа (т. е. его порог равен 2), а второй нейрон возбуждается, если активен хотя бы один из его входов (т. е. его порог равен 1). На рисунке показано, как конечный выход может изменяться в зависимости от входов.





влажно и облачно, но дождя нет



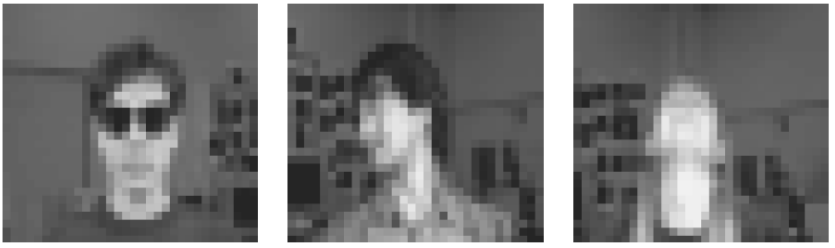
облачно, но не влажно и дождя нет

Вверху: нейронная сеть для задачи о зонтике.
Посредине и внизу: нейронная сеть в действии. Нейроны, входы и выходы, находящиеся в возбужденном состоянии, закрашены серым цветом. Входы на среднем рисунке говорят, что дождя нет, но влажно и облачно, поэтому принимается решение взять зонтик. На нижнем рисунке активен только вход «облачно?», в связи с чем принимается решение не брать зонтик.

Сейчас я рекомендую вернуться к дереву решений для задачи о зонтике на стр. 104. Как выясняется, дерево решений и нейронная сеть дают в точности одинаковые результаты на одних и тех же входных данных. Для этой простой, искусственно придуманной задачи дерево решений, пожалуй, является более подходящим представлением. Но далее мы рассмотрим гораздо более сложную задачу, которая демонстрирует истинную мощь нейронных сетей.

Нейронная сеть для задачи о солнечных очках

В качестве примера настоящей задачи, которую можно успешно решить с помощью нейронных сетей, рассмотрим «задачу о солнечных очках». Входными данными для нее является база данных, содержащая фотографии лиц, сделанные с низким разрешением. Люди сняты по-разному: одни смотрят прямо в камеру, другие – вверх, третьи – влево или вправо, а некоторые носят солнечные очки. На рисунке ниже приведено несколько примеров.

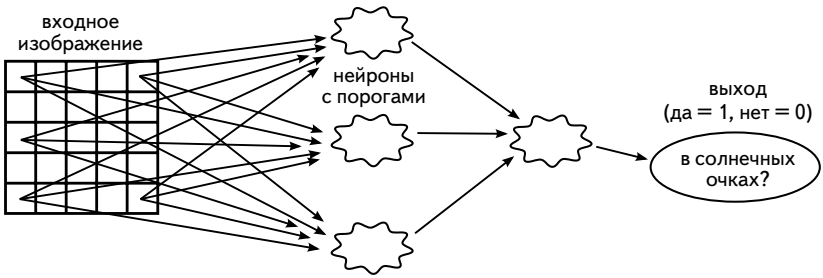


Лица, подлежащие «распознаванию» нейронной сетью. На самом деле, мы не распознаем лица, а решаем более простую задачу – определить, есть на лице солнечные очки или нет. Источник: Tom Mitchell «Machine Learning», McGraw-Hill (1998). Используется с разрешения правообладателя.

Мы сознательно взяли изображения низкого разрешения, чтобы было проще описать нейронную сеть. Разрешение каждого изображения составляет всего лишь 30×30 пикселей. Но, как мы скоро увидим, нейронная сеть способна выдать удивительно хорошие результаты при таком грубом разрешении.

Нейронные сети можно использовать для стандартного распознавания лиц из этой базы данных, т. е. чтобы установить личность изображенного на фотографии человека независимо от того, смотрит он в камеру или скрыл глаза за солнечными очками. Но мы займемся

задачей попроще, на которой сможем более отчетливо продемонстрировать свойства нейронных сетей. Наша цель – определить, есть на данном лице солнечные очки или нет.



Нейронная сеть для задачи о солнечных очках.

На рисунке выше схематически представлена структура сети. Здесь не показаны каждый нейрон и каждая связь. Прежде всего, в глаза бросается единственный выходной нейрон в правой части, который порождает 1, если на входном изображении присутствуют солнечные очки, и 0 в противном случае. В центре сети мы видим три нейрона, которые получают сигналы непосредственно от входного изображения и посылают сигналы выходному нейрону. Наиболее сложна часть сети слева, где видны многочисленные связи между входным изображением и центральными нейронами. В реальной сети каждый пиксель входного изображения связан с каждым из центральных нейронов, но на рисунке, конечно, показаны не все эти связи. Произведя несложные вычисления, можно убедиться, что таких связей довольно много. Напомним, что мы работаем с изображениями низкого разрешения 30×30 пикселей. Но даже такое, по современным стандартам, крохотное изображение состоит из $30 \times 30 = 900$ пикселей. А если учесть, что центральных нейронов три, то всего получится $3 \times 900 = 2700$ связей в левой части сети.

Почему выбрана имена такая структура сети? Нельзя ли соединить нейроны по-другому? Да, можно, существует много вариантов структуры сети, дающих хорошие решения задачи о солнечных очках. Выбор структуры часто основан на предыдущем опыте, который показывал хорошие результаты. Повторим еще раз, в системах распознавания образов важны как глубокое понимание, так и интуиция.

К сожалению, как мы скоро увидим, чтобы сеть заработала правильно, каждую из 2700 связей необходимо «настроить». Как справиться с задачей такой сложности? Ответ прост: настройка производится автоматически путем обучения на примерах.

Добавление взвешенных сигналов

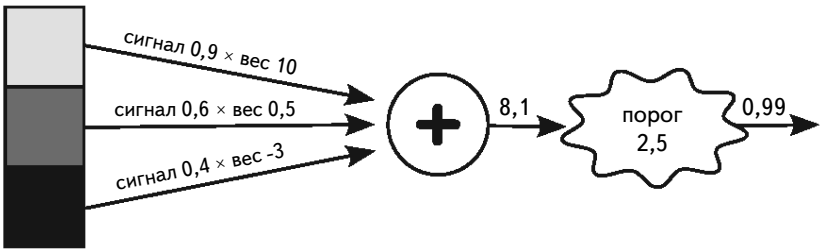
Выше уже было отмечено, что для задачи о зонтике мы использовали очень простую нейронную сеть. В задаче о солнечных очках мы добавили три важных улучшения.

Улучшение 1. Сигналы могут принимать любое значение от 0 до 1 включительно (напомним, что в сети для задачи о зонтике входные и выходные сигналы могли принимать только значения 0 и 1). Другими словами, сигнал в новой сети может быть равен, например 0,0023 или 0,755. Давайте подумаем, что это значит в контексте задачи о солнечных очках. Яркость пикселя входного изображения соответствует значению сигнала, посылаемого по исходящим из пикселя связям. Поэтому белый пиксель посылает значение 1, черный – значение 0. А различным оттенкам серого соответствуют значения между 0 и 1.

Улучшение 2. Результирующий вход вычисляется в виде взвешенной суммы. В сети для задачи о зонтике нейроны суммировали величины входов, не принимая во внимание, что связи могут иметь разную силу. Сила связи представлена числом, которое называется весом связи. Вес может как положительным, так и отрицательным. Большими положительными весами (например, 51,2) представлены сильные возбуждающие связи – если сигнал передается по такой связи, то находящийся на ее конце нейрон с большой вероятностью возбудится. Большими отрицательными весами (например, –121.8) представлены сильные тормозящие связи – если сигнал передается по такой связи, то находящийся на ее конце нейрон, скорее всего, останется пассивным. Связи с малыми весами (например, 0,03 или –0,0074) оказывают небольшое влияние на состояние принимающего нейрона. (В действительности, вес можно назвать «большим» или «малым» только в сравнении с другими весами, поэтому приведенные выше численные значения имеют смысл, только если относятся к связям, ведущим в один и тот же нейрон.) Когда нейрон вычисляет сумму входов, каждый входной сигнал умножается на вес его связи, и результат прибавляется к сумме. Поэтому большие веса влияют на сумму больше, чем малые, и вполне может статься, что возбуждающие и тормозящие сигналы уравновесят друг друга.

Улучшение 3. Эффект порога смягчается. Выходной сигнал не обязан либо полностью присутствовать (быть равным 1), либо

полностью отсутствовать (быть равным 0); его значение может быть любым числом от 0 до 1 включительно. Если суммарный входной сигнал существенно ниже порога, то выход будет близок к 0, а если существенно выше, то к 1. Если же величина суммарного входа оказывается вблизи порога, то выходной сигнал будет близок к 0,5. Рассмотрим, к примеру, нейрон с порогом 6,2. Вход величиной 122 может породить выход величиной 0,995, потому что он существенно выше порога. А вход величиной 6,1 близок к порогу и может породить выход 0,45. Этот эффект относится ко всем нейронам, в том числе и к конечному выходному. В нашей задаче о солнечных очках, выходное значение, близкое к 1, означает, что с большой вероятностью очки есть, а выходное значение, близкое к 0, – что очков, скорее всего, нет.



Перед суммированием сигнал умножается на вес связи.

На рисунке выше показан искусственный нейрон нового типа со всеми тремя улучшениями. Этот нейрон получает входы от трех пикселей: яркого (сигнал 0,9), среднего (сигнал 0,6) и темного (сигнал 0,4). Веса соответствующих связей с этим нейроном равны 10, 0,5 и –3. Величины сигналов умножаются на веса и складываются, в результате чего получается суммарный входной сигнал величиной 8,1. Поскольку 8,1 значительно больше порога нейрона, равного 2,5, то выход очень близок к 1.

Настройка нейронной сети посредством обучения

Теперь мы готовы определить, что значит настроить искусственную нейронную сеть. Во-первых, каждой связи (а их, напомним, могут быть многие тысячи) необходимо назначить вес: положительный (возбуждающая связь) или отрицательный (тормозящая связь). Во-вторых,

каждому нейрону нужно назначить порог. Можно считать, что веса и пороги – это небольшие рукоятки, которые можно поворачивать, как реостат электрической лампы, регулирующий ее яркость.

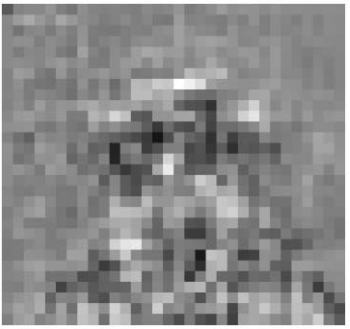
Разумеется, если крутить такие рукоятки вручную, то потребуется недопустимо много времени. Но мы можем поручить это компьютеру на этапе обучения. Первоначально каждая рукоятка устанавливается в случайное положение (такое решение может показаться чрезмерно произвольным, но именно так поступают профессионалы в реальных приложениях). Затем компьютеру предъявляется первый обучающий образец. В нашем приложении это фотография человека в солнечных очках или без них. Образец прогоняется через сеть, которая порождает одно выходное значение между 0 и 1. Но поскольку образец *обучающий*, известно «целевое» значение, которое должна вернуть сеть в идеале. Основной трюк заключается в том, чтобы немного изменить сеть, так чтобы выход оказался ближе к целевому значению. Допустим, к примеру, что на первом обучающем образце изображен человек в очках. Тогда целевое значение равно 1. Следовательно, каждая рукоятка в сети немного поворачивается в направлении, которое приближает выходное значение к 1. Если же на первом обучающем образце очков нет, то каждая рукоятка немного поворачивается в обратном направлении. Надо полагать, вы уже поняли, как продолжается этот процесс. Сети один за другим предъявляются обучающие образцы, и всякий раз рукоятки поворачиваются, так чтобы показатели работы сети улучшились. После многократного прогона всех обучающих образцов сеть достигает хороших показателей, обучение прекращается и рукоятки фиксируются в текущих положениях.

Как именно вычисляются малые корректировки положения рукояток, весьма важно, но для понимания этого требуется знание математики в объеме, выходящем за рамки этой книги. Соответствующая отрасль математики называется многопараметрической оптимизацией, ее обычно изучают на среднем курсе университета. Да, без математики никуда! Отметим также, что описанный здесь подход, который специалисты называют «стохастическим градиентным спуском» – лишь один из многих методов обучения нейронных сетей.

У всех этих методов один и тот же принцип, поэтому поговорим об общей картине: этап обучения нейронной сети – достаточно длительный процесс, в котором многократно корректируются все веса и пороги, пока сеть не станет показывать хорошие результаты на обучающих образцах. Однако все это может автоматически сделать компьютер, а в результате получится сеть, которую можно будет исполь-

зовать для классификации новых образцов простым и эффективным способом.

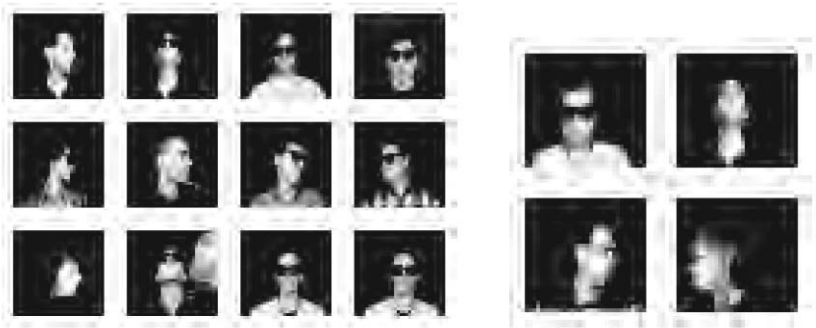
Посмотрим, как все это работает в задаче о солнечных очках. По завершении этапа обучения каждой из нескольких тысяч связей между исходным изображением и центральными нейронами назначен числовой вес. Если взять только связи, идущие от каждого пикселя к одному какому-нибудь нейрону (скажем, верхнему), то можно будет очень наглядно представить эти веса, преобразовав их в изображение. Такая визуализация весов для одного из центральных нейронов показана на рисунке ниже. На рисунке сильные возбуждающие связи (с большими положительными весами) закрашены белым цветом, сильные тормозящие связи (с большими отрицательными весами) – черным. Различными оттенками серого представлены связи промежуточной силы. В соответствующем пикселе показан вес связи. Внимательно приглядитесь к рисунку. В той области, где обычно находятся солнечные очки, явственно видна полоса сильных тормозящих связей – можно даже убедить себя в том, будто вы на самом деле видите изображение очков. Можно назвать это «призраком» очков, потому что никаких реальных очков здесь, конечно, не изображено.



Веса (т. е. сила) связей с одним из центральных нейронов в сети для задачи о солнечных очках

Проявление такого призрака покажется особенно удивительным, если вспомнить, что веса назначались без участия человека, который знает о типичном цвете и местоположении солнечных очков. *Единственная* информация, предоставленная человеком, – набор обучающих изображений, для каждого из которых всего лишь сказано, есть на нем очки или нет. Призрак солнечных очков проявился автоматически в результате повторяющейся корректировки весов на этапе обучения.

С другой стороны, ясно видны и другие области сильных весов, которые теоретически не должны влиять на решение о наличии солнечных очков. Как быть с этими бессмысленными и кажущимися случайными связями? Тут мы столкнулись с одним из важных уроков, который получили исследователи, изучающие искусственный интеллект в течение нескольких последних десятилетий: псевдоразумное поведение может проявиться в системах, кажущихся случайными. Вообще-то, это не должно вызывать удивления. Если бы у нас была возможность заглянуть в наш мозг и проанализировать силу связей между нейронами, то подавляющее их большинство показались бы случайными. И тем не менее, эти беспорядочные связи непредсказуемой силы порождают наше разумное поведение!



Результаты работы сети для задачи о солнечных очках.
Источник: Tom Mitchell «Machine Learning», McGraw-Hill (1998).
Используется с разрешения правообладателя.

Использование сети для задачи о солнечных очках

Итак, мы имеем сеть, которая может выдавать произвольное значение от 0 до 1, и возникает вопрос: как все же узнать, есть на человеке солнечные очки или нет. Ответ удивительно прост: если на выходе получается число больше 0,5, считаем, что очки есть, если меньше – что очков нет.

Для проверки построенной нейронной сети я провел эксперимент. В базе данных о лицах есть около 600 изображений, из них 400 я использовал для обучения сети, а на остальных 200 проверял качество ее работы. Точность сети оказалась в районе 85 %. Другими словами,

сеть давала правильный ответ на вопрос «носит ли данный человек солнечные очки» примерно для 85 % изображений, которые раньше не видела. На рисунке выше показано, что часть изображений классифицирована правильно, а часть – неправильно. Всегда интересно исследовать образцы, на которых алгоритм распознавания дает сбой, и эта нейронная сеть – не исключение. Одно или два неправильно классифицированных изображения – действительно трудные задачи, на которых даже человек может ошибиться. Но по меньшей мере одно (левое верхнее на правом рисунке) нам представляется абсолютно однозначным – человек смотрит прямо в камеру и, безусловно, носит очки. Таинственные сбои такого рода – вовсе не редкость в задачах распознавания образов.

Разумеется, современные нейронные сети на этой задаче способны достигать гораздо более высокого уровня распознавания, чем 85 %. Мы лишь хотели понять основные идеи на примере простой сети.

Распознавание образов: прошлое, настоящее и будущее

Выше уже отмечалось, что распознавание образов – часть более обширной области знаний, искусственного интеллекта (ИИ). При распознавании образов имеют дело с сильно изменяющимися данными: аудиозаписями, фотографиями и видео, тогда как ИИ в целом решает и многие другие задачи, в том числе, обучение компьютера игре в шахматы, моделирование роботов в чатах, создание человекоподобных роботов.

Рождение ИИ было подобно взрыву: на конференции в Дартмутском колледже в 1956 году группа из десяти ученых по существу основала новую науку, впервые введя в обращение само выражение «искусственный интеллект». В дерзкой заявке на финансирование конференции, направленной в фонд Рокфеллера, организаторы писали, что дискуссия «исходит из предположения о том, что любой аспект обучения или иное проявление разума в принципе возможно описать настолько точно, что можно будет построить моделирующую его машину».

Дартмутская конференция обещала много, но последующие десятилетия принесли мало. Большие надежды ученых, убеждавших общество в том, что прорыв к истинно «разумным» машинам уже маячит на горизонте, раз за разом разбивались в прах, когда прототипы

упорно продолжали демонстрировать механистическое поведение. И даже достижения в развитии нейронных сетей мало что изменили: после первых многообещающих результатов ученые упирались все в ту же стену механистического поведения.

Однако медленно, но верно ИИ сокращал совокупность мыслительных процессов, считающихся присущими только человеку. В течение многих лет царило мнение, что благодаря интуиции и озарению шахматные гроссмейстеры способны одолеть любую компьютерную программу, которая по необходимости следует детерминированным правилам, а не интуиции. И все же этот камень преткновения для ИИ был убедительно разрушен в 1997 году, когда созданный в IBM компьютер Deep Blue обыграл чемпиона мира Гарри Каспарова.

Тем временем успехи ИИ постепенно начинали оказывать влияние и на обыденную жизнь. Автоматизированные телефонные системы, в которых при обслуживании клиентов применяется распознавание речи, стали нормой. Управляемые компьютером противники в видеоиграх стали демонстрировать свойственные человеку стратегии, даже особенности характера и слабости. Такие онлайновые сервисы, как Amazon и Netflix, научились рекомендовать предметы, исходя из автоматически выведенных интересов покупателя, и часто это приносит на удивление полезные результаты.

И само наше восприятие этих достижений оказало важнейшее влияние на прогресс в области искусственного интеллекта. Возьмите задачу, которая в 1990 году безусловно требовала разумного вмешательства человека, которому платили за его опыт и знания: планирование маршрута полета с несколькими остановками. В 1990 году выбор удобного и дешевого маршрута во многом зависел от туристического агента.

Но в 2010 году компьютеры лучше, чем люди, решают эту задачу. Рассказ о том, как именно они это делают, сам по себе был бы очень интересен, потому что для планирования маршрута применяется несколько красивейших алгоритмов. Но важен даже не столько результат, сколько наше *восприятие*. Смею утверждать, что в 2010 планирование маршрута большинством людей воспринимается как чисто механическая задача – разительный контраст с положением дел 20-ю годами раньше.

Такая постепенная трансформация задач – от кажущихся интуитивными к очевидно механистическим – продолжается. ИИ вообще и распознавание образов в частности медленно расширяют сферу охвата и совершенствуются. Описанные в этой главе алгоритмы – клас-

сификаторы по ближайшим соседям, деревья решений и нейронные сети – применимы к широчайшему кругу практических задач. К ним относится и исправление ошибок при вводе текста толстым пальцем на виртуальной клавиатуре сотового телефона, и помощь в постановке диагноза по результатам большого количества анализов, и распознавание автомобильных номеров на автоматизированном пункте оплаты проезда по платной дороге, и выбор рекламного объявления, демонстрируемого конкретному пользователю, – и это только малая часть. Таким образом, эти алгоритмы – важнейшие составные части систем распознавания образов. Неважно, считаете вы их по-настоящему «разумными» или нет, в ближайшем будущем вас ждет много нового.

ГЛАВА 7. Сжатие данных: кое-что задаром[1]

Эмма, ободренная таким приемом, была готова доказать, что с нею самой обстоит иначе, когда бы ее не остановил голос миссис Элтон, долетевший из дверей гостиной; пришлось ограничиться очень крепким, искренним рукопожатием, в которое она постаралась вложить и свои дружеские чувства, и поздравленье.

— Джейн Остин «*Эмма*»[2]

Все мы знакомы с идеей *сжатия* физических предметов: пытаясь втиснуть кучу одежды в маленький чемодан, мы уминаете ее, чтобы все поместилось, хотя в нормальном состоянии объем одежды больше, чем объем чемодана. Вы *сжимаете*, или *упаковываете* одежду. Впоследствии вы *распаковываете* одежду, доставая ее из чемодана, надеясь, что исходный размер и форма восстановятся.

Но то же самое можно сделать и с информацией: компьютерные файлы и другие данные можно сжать, уменьшив размер для удобства хранения и транспортировки. Позже их можно распаковать и использовать в исходном виде.

В компьютерах большинства пользователей достаточно места на диске, поэтому необходимости в сжатии файлов не возникает. Поэтому складывается впечатление, будто сжатие к нам не имеет отношения. Но это ошибочное мнение: на самом деле, за кулисами сжатие используется в компьютерных системах очень часто. Например, многие сообщения, передаваемые через Интернет, сжимаются без ведома пользователя, и почти все скачиваемое программное обеспечение представлено в сжатом виде, благодаря чему на передачу файлов уходит в несколько раз меньше времени. Даже ваша речь сжимается, когда вы говорите по телефону: за счет сжатия голосовых данных до

[1] Something for Nothing – так назывался широко известный фантастический рассказ Р. Шекли в переводе Т. Озерской. *Прим. перев.*

[2] Перевод М. Канн.

передачи телефонным компаниям удается использовать свои ресурсы гораздо эффективнее.

Сжатие используется и в более очевидных ситуациях. В популярном файловом формате ZIP применяется остроумный алгоритм сжатия, который мы опишем в этой главе. И, возможно, вы знакомы с компромиссом при сжатии цифрового видео: чем выше качество видео, тем больше размер содержащего его файла.

Сжатие без потери информации: бесплатный сыр бывает не только в мышеловке

Важно понимать, что есть два очень разных вида сжатия: без потери и с потерей информации. Сжатие без потери информации – это самый что ни на есть бесплатный сыр, когда вы действительно получаете кое-что задаром. Алгоритм сжатия без потери информации получает на входе файл, сжимает его в несколько раз, а затем распаковывает, получая *в точности* исходный файл. Напротив, в случае сжатия с потерей информации файл, получающийся после распаковки, немного отличается от исходного. Сжатие с потерей информации мы обсудим позже, а пока займемся сжатием без потерь. В качестве примера рассмотрим исходный файл, содержащий текст этой книги. После его сжатия и последующей распаковки получается тот же самый текст – не изменилось ни одно слово, ни один пробел, ни одна запятая. Но хочется предостеречь вас от чрезмерных восторгов при виде бесплатного сыра: алгоритмы сжатия без потери информации дают впечатляющее уменьшение размера отнюдь не для *всех* файлов. Однако хороший алгоритм сжатия может существенно уменьшить размеры файлов большинства часто встречающихся типов.

Но откуда берется этот бесплатный сыр? Как это возможно – взять кусок данных (информации) и уменьшить их «истинный» размер, ничего не испортив, так что впоследствии все можно в точности восстановить? Вообще-то, люди так поступают постоянно, даже не задумываясь. Возьмем, к примеру, ваш ежедневник. Для простоты предположим, что вы работаете пять дней в неделю по восемь часов в день и что ежедневник разбит на интервалы по одному часу. Таким образом, в каждом из пяти дней восемь интервалов, а в неделе их всего 40. Чтобы сообщить кому-то о содержании своего расписания на неделю, вы должны передать 40 единиц информации. Но если кто-то хочет по телефону условиться о

встрече на будущей неделе, будете вы рассказывать ему, когда у вас есть время, перечисляя все 40 единиц информации? Конечно, нет! Скорее всего, вы скажете что-то типа: «Понедельник и вторник полностью забиты, в четверг и пятницу я занят с часу до трех, а в остальное время свободен». Это и есть пример сжатия данных без потери информации! Ваш собеседник может точно реконструировать сведения о вашей занятости во всех 40 интервалах, хотя вы явно об этом не говорили.

Быть может, вам подумалось, что такой вид «сжатия» – жульничество, потому что опирается н тот факт, что многие фрагменты расписания одинаковы. Точнее, все интервалы в понедельник и вторник заняты, так что их можно описать очень быстро, а вся остальная неделя свободна за исключением двух интервалов, которые тоже легко описать. Да, действительно, это очень простой пример. И тем не менее, сжатие данных в компьютере устроено по тому же принципу: основная идея заключается в том, чтобы найти одинаковые фрагменты данных и с помощью того или иного трюка описать их более эффективно.

Это особенно легко, когда в данных есть повторы. Например, вам, наверное, не составит труда придумать хороший способ сжатия следующих данных:

```
AAAAAAAAAAAAAAAAAAAAABCBCBCBCBCBCBCBCBCBCAAAAAADEFDEFDEF
```

Если это сразу не очевидно, подумайте, как бы стали диктовать эти данные по телефону. Уверен, что вместо «A, A, A, A, ..., D, E, F» вы сказали бы что-то вроде «21 раз A, потом 10 раз BC, потом еще 6 раз A, потом 3 раза DEF». А если бы нужно было сокращенно записать эта данные на бумажке, вы, наверное, написали бы что типа «21A,10BC,6A,3DEF». В данном случае вы сжали исходные данные – 56 знаков – в цепочку, содержащую всего 16 знаков. Это меньше трети исходного размера – совсем неплохо! В информатике этот конкретный трюк называется *кодированием длин серий*, потому что мы кодируем «серию» повторяющихся знаков с помощью ее «длины».

К сожалению, кодирование длин серий полезно только для сжатия очень специфических данных. Оно применяется на практике, но, как правило, в сочетании с другими алгоритмами сжатия. Например, в факсах кодирование длин серий комбинируется еще с одним методом – кодом Хаффмана, о котором мы будем говорить ниже. Основная проблема заключается в том, что повторяющиеся данные должны быть *соседними*. С помощью кодирования длин серий легко сжать цепочку `ABABAB` (представив ее в виде `3AB`), но невозможно сжать цепочку `ABXABYAB`.

Вероятно, вы понимаете, почему кодирование длин серий можно использовать в факсимильных аппаратах. Факсы, по определению, черно-белые документы, которые преобразуются в большое количество точек черного или белого цвета. При чтении точек по порядку (слева направо, сверху вниз) встречаются длинные серии белых точек (фон) и короткие серии черных точек (напечатанный или рукописный текст). Это и позволяет эффективно использовать кодирование длин серий. Но, как уже было сказано, данные, к которым эта техника применима, встречаются нечасто.

Поэтому ученые изобрели целый ряд более сложных трюков, основанных на той же идее (найти повторения и эффективно их описать), но работающих и тогда, когда данные не являются соседними. Мы рассмотрим два таких трюка: «то же, что и раньше» и «более короткий символ». Чтобы создать ZIP-файл, ничего, кроме этих двух трюков, не нужно, а формат ZIP чаще всего применяется для сжатия файлов на персональных компьютерах. Поэтому, поняв стоящие за этими трюками идеи, вы будете понимать, как ваш компьютер использует сжатие, – в большинстве случаев.

Трюк «то же, что и раньше»

Представьте, что перед вами стоит пугающая задача продиктовать по телефону вот такую строчку:

```
VJGDNQMYLH-KW-VJGDNQMYLH-ADXSGF-O-
VJGDNQMYLH-ADXSGF-VJGDNQMYLH-EW-ADXSGF
```

В ней 63 знака (черточки мы игнорируем, они включены только для того, чтобы бы проще читать данные). Нельзя ли придумать что-то получше произнесения всех 63 знаков по одному? Прежде всего, мы замечаем, что в данных много повторов. На самом деле, большая часть «блоков», разделенных черточками, повторяется хотя бы один раз. Поэтому, диктуя данные, можно существенно упростить себе задачу, если сказать что-то типа «эта часть совпадает с той, что я тебе говорил раньше». Правда, нужно будет уточнить, насколько раньше и сколько знаков в повторяющейся части, например, так: «вернись назад на 27 знаков и, начиная с этого места, скопируй 8 знаков».

Посмотрим, как эта стратегия работает на практике. В первых 12 знаках нет повторов, поэтому придется произнести их по одному: «V, J, G, D, N, Q, M, Y, L, H, K, W». Но следующие 10 знаков совпадают с уже встречавшимися, поэтому можно просто казать «вернись на 12, скопируй 10». Следующие семь знаков новые, диктуем их по одному:

«A, D, X, S, G, F, O». Но следующие 16 – один большой повтор, поэтому говорим: «вернись на 17, скопируй 16». Очередные 10 знаков тоже повторяют встречавшиеся ранее, поэтому говорим: «вернись на 16, скопируй 10». Следующие два знака повторами не являются, поэтому диктуем, как есть: «E, W». Наконец, последние 6 знаков совпадают с встречавшимися ранее, и мы говорим: «вернись на 18, скопируй 6».

Попробуем кратко сформулировать наш алгоритм сжатия. Будем использовать аббревиатуру `b` вместо «вернись на» и `c` вместо «скопируй». Тогда инструкция «вернись на 18, скопируй 6» сокращенно записывается в виде `b18c6`. А все инструкции во время диктовки можно записать так:

```
VJGDNQMYLH-KW-b12c10-ADXSGF-O-b17c16-b16c10-EW-b18c6
```

Эта цепочка состоит всего из 44 знаков. В исходной было 63 знака, так что мы сэкономили 19, или почти треть от первоначальной длины.

У трюка «то же, что и раньше» есть еще одна интересная особенность. Как применить его для сжатия сообщения `FG-FG-FG-FG-FG-FG-FG-FG`? (Как и раньше, черточки не являются частью сообщения, а добавлены только для удобочитаемости.) Группа `FG` повторяется 8 раз, поэтому мы могли бы продиктовать первые четыре группы по отдельности, а потом воспользоваться инструкцией вернись-и-скопируй: `FG-FG-FG-FG-b8c8`. Это сэкономит несколько знаков, но можно добиться лучшего результата. Для этого понадобится инструкция, которая на первый взгляд кажется бессмысленной: «вернись на 2, скопируй 14», или `b2c14`. Таким образом, сжатое сообщение записывается в виде `FG-b2c14`. Но как можно скопировать 14 знаков, если доступно всего два? На самом деле, никакой проблемы нет, если копируются знаки из *восстанавливаемого*, а не сжатого сообщения. Проделаем это шаг за шагом. После того как продиктованы первые два знака, мы имеем `FG`. Далее поступает инструкция `b2c14`, поэтому мы возвращаемся на 2 символа назад и начинаем копировать. Пока есть только два символа (`FG`) , их и скопируем: добавив их к тому, что уже есть, получаем `FG-FG`. Но теперь-то у нас больше двух знаков! Копируем дальше и после добавления к уже имеющемуся восстановленному сообщению получаем `FG-FG-FG`. Появилось еще два знака, так что мы можем продолжить копирование. И так можно действовать, до тех пор пока не будет скопировано требуемое количество знаков (в данном случае 14). Чтобы проверить, насколько хорошо вы поняли, попробуйте распаковать следующее сжатое сообщение: `Ab1c250`[3].

[3] Ответ: буква A, повторенная 251 раз.

Трюк «более короткий символ»

Чтобы понять, в чем смысл трюка «более короткий символ», нужно поближе познакомиться с тем, как в компьютерах хранятся сообщения. Возможно, вам доводилось слышать, что компьютеры не хранят буквы: *a*, *b*, *c* и т. д. Все буквы хранятся в виде чисел, а затем интерпретируются как буквы в соответствии с некоторой фиксированной таблицей (техника преобразования букв в цифры и наоборот уже упоминалась при обсуждении контрольных сумм на стр. 81). Например, можно договориться представлять букву «a» числом 27, «b» – числом 28, а «c» –числом 29. Тогда строчка «abc» будет храниться в компьютере в виде «272829», но перед отображением на экране или печатью на бумаге ее легко преобразовать обратно в «abc».

На следующей странице приведен полный список 100 символов, сохраняемых в компьютере, и для каждого символа двузначный код. Отметим попутно, что этот конкретный набор двузначных кодов в реальных компьютерных системах не используется, хотя настоящие коды очень на него похожи. Основное отличие состоит в том, что компьютеры работают не с десятичной системой, как люди, а с так называемой двоичной. Впрочем, эти детали для нас несущественны. Трюк «более короткий символ» работает как в десятичной, так и в двоичной системе, поэтому для простоты объяснений будем считать, что компьютеры пользуются десятичной системой.

Приглядитесь к таблице символов внимательнее. Обратите внимание, что в первой строке указан числовой код пробела между словами – «00». Затем идут заглавные буквы от *A* («01») до *Z* («26»), а вслед за ними – строчные буквы от *a* («27») до *z* («52»). Далее следуют знаки препинания и некоторые буквы из европейских алфавитов, от *á* («80») до *Ù* («99»).

Как с помощью этих двузначных кодов сохранить в компьютере фразу «Meet your fiancé there.»? Просто: нужно лишь заменить каждый знак соответствующим ему кодом и записать все коды друг за другом:

```
M  e  e  t     y  o  u  r     f  i  a  n  c  é     t  h  e  r  e  .
13 31 31 46 00 51 41 47 44 00 32 35 27 40 29 82 00 46 34 31 44 31 66
```

Важно четко понимать, что в компьютере пары цифр не разделяются, т. е. это сообщение хранится в виде непрерывной цепочки из 46 цифр: «13131460051414744003235274029820046343144 3166». Конечно, человеку такую цепочку интерпретировать несколько сложнее, но для компьютера никаких трудностей здесь нет, поскольку он

с легкостью может составить пары цифр, перед тем как преобразовывать их в знаки для отображения на экране. Главное, что нет никакой неоднозначности в том, как выделять числовые коды, поскольку каждый код состоит ровно из двух цифр. На самом деле, именно по этой причине буква *A* представлена как «01», а не просто «1», *B* – как «02», а не «2» и т. д. до буквы *I* («09», а не «9»). Если бы мы представляли *A* как «1», *B* как «2» и т. д., то однозначно интерпретировать сообщение было бы невозможно. Действительно, сообщение «1123» можно было бы разбить как «1 1 23» (что соответствует AAW) или как «11 2 3» (KBC) или даже как «1 1 2 3» (AABC). Так что запомните важную мысль: преобразование между числовыми кодами и знаками должно быть однозначным, даже если коды хранятся подряд, без каких-либо разделителей. Скоро мы встретимся с этим положением в весьма удивительной ситуации!

пробел	00	T	20	n	40	(	60	á	80
A	01	U	21	o	41	)	61	à	81
B	02	V	22	p	42	*	62	é	82
C	03	W	23	q	43	+	63	è	83
D	04	X	24	r	44	,	64	í	84
E	05	Y	25	s	45	-	65	ì	85
F	06	Z	26	t	46	.	66	ó	86
G	07	a	27	u	47	/	67	ò	87
H	08	b	28	v	48	:	68	ú	88
I	09	c	29	w	49	;	69	ù	89
J	10	d	30	x	50	<	70	Á	90
K	11	e	31	y	51	=	71	À	91
L	12	f	32	z	52	>	72	É	92
M	13	g	33	!	53	?	73	È	93
N	14	h	34	“	54	{	74	Í	94
O	15	i	35	#	55	|	75	Ì	95
P	16	j	36	$	56	}	76	Ó	96
Q	17	k	37	%	57	~	77	Ò	97
R	18	l	38	&	58	Ø	78	Ú	98
S	19	m	39	’	59	ø	79	Ù	99

Числовые коды, которые компьютер мог бы использовать для хранения символов.

А пока вернемся к трюку «более короткий символ». Как и для многих других, на первый взгляд, сугубо технических идей, описанных в этой книге, этот трюк люди применяют постоянно, даже не задумываясь. Основная идея проста – если что-то используется достаточно часто, то имеет смысл завести сокращение. Каждый знает, что «США» – сокращение от «Соединенные Штаты Америки», и мы экономим уйму времени, когда набираем на клавиатуре или произносим трехбуквенный код «США» вместо полного названия, содержащего 25 знаков. Но, с другой стороны, мы не вводим 3-буквенные коды для каждой фразы из 25 знаков. Вы знаете сокращение для фразы «Какое над нами синее небо»? Нет, конечно. А почему? В чем разница между фразами «Соединенные Штаты Америки» и «Какое над нами синее небо»? Да в том, что одна из них используется гораздо чаще, чем другая, и, сократив первую, мы сможем сэкономить гораздо больше времени, чем сократив вторую.

А теперь попробуем применить эту идею к системе кодирования, показанной выше. Мы уже знаем, что можем сэкономить максимально много, используя сокращения для того, что часто встречается. Что ж, в английском языке чаще всего встречаются буквы «e» и «t», поэтому давайте сопоставим им более короткие коды. В данный момент букве «e» соответствует код 31, а букве «t» – код 46, т. е. каждая представлена двумя цифрами. А что если взять только одну цифру? Пусть «e» представляется цифрой 8, а «t» – цифрой 9. Отличная идея! Вспомните, как была раньше закодирована фраза «Meet your fiancé there» – потребовалось 46 цифр. А теперь мы можем обойтись всего 40 цифрами:

```
M  e  e  t     y  o  u  r     f  i  a  n  c  é     t  h  e  r  e  .
13 8  8  9 00 51 41 47 44 00 32 35 27 40 29 82 00 9  34 8  44 8  66
```

Увы, у этого плана есть фатальный изъян. Вспомните, что компьютер не хранит пробелы между буквами. Поэтому закодированная фраза выглядит не как «13 8 8 9 00 51 ... 44 8 66», а как «138890051... 44866». Вы уже видите, в чем проблема? Возьмем только первые пять цифр – 13889. Код 13 соответствует букве «M», код 8 – букве «e», а код 9 – букве «t», поэтому эту строчку можно разбить так: 13-8-8-9, что дает слово «Meet». Но ведь код 88 соответствует букве «ú» с диакритическим знаком, поэтому строчку 13889 можно разбить и так: 13-88-9, и тогда получится слово «Mút». На самом деле, ситуация еще хуже, потому что код 89 соответствует букве щ с другим диакритическим знаком, поэтому строчку 13889 можно разбить и третьим спо-

собом: 13-8-89, а это будет слово «Meù». Нет никакой возможности решить, какая из трех интерпретаций правильна.

Облом! Наш хитроумный план взять более короткие коды для букв «e» и «t» привел к неработоспособной системе кодирования. К счастью, ситуацию можно исправить с помощью еще одного трюка. В чем, собственно, проблема? В том, что видя цифру 8 или 9, мы не можем сказать, является ли она частью кода из одной цифры (например, «e» или «t») или кода из двух цифр, начинающегося с 8 или 9 (различные буквы с диакритическими знаками типа «á» или «è»). Чтобы решить эту проблему, мы вынуждены пойти на жертву: некоторые наши коды должны стать *длиннее*. Неоднозначные коды из двух цифр, начинающиеся с 8 и 9, теперь будут содержать три цифры, но начинаться не с 8 и не с 9. В таблице на стр. 130 показан один из способов сделать это. Под топор попали и некоторые знаки препинания, но в итоге мы получили весьма симпатичную ситуацию: с цифры 7 начинаются только трехзначные коды, тогда как коды, начинающиеся с 8 или 9, состоят всего из одной цифры. Ну а все коды, начинающиеся с 0, 1, 2, 3, 4, 5 или 6, по-прежнему содержат две цифры. Теперь разбить строчку 13889 можно только одним способом (13-8-8-9 – «Meet»), и это справедливо для всех вообще правильно сформированных последовательностей цифр. Неоднозначность устранена, и исходное сообщение кодируется следующим образом:

```
M  e  e  t     y  o  u  r     f  i  a  n  c  é      t  h  e  r  e  .
13 8  8  9 00 51 41 47 44 00 32 35 27 40 29 782 00 9  34 8  44 8  66
```

В первоначальной кодировке было 46 цифр, в этой – только 41. Вроде бы, экономия копеечная, но для длинных сообщений выигрыш может оказаться очень значительным. Например, текст этой книги (только слова, без картинок) занимает примерно 500 килобайт – это полмиллиона знаков! А после сжатия с помощью двух описанных выше трюков размер сократился до 160 килобайт, то есть более чем в три раза.

Резюме: откуда берется бесплатный сыр?

Сейчас мы понимаем все важные идеи, лежащие в основе создания типичного сжатого ZIP-файла. Вот что происходит.

Шаг 1. Исходный несжатый файл преобразуется с помощью трюка «то же, что и раньше», в результате чего большинство повторяющихся данных заменяется гораздо более короткими

инструкциями – вернуться назад и скопировать данные из другого места.

Шаг 2. Определяется, какие символы встречаются в преобразованном файле чаще всего. Например, если исходный файл написан по-английски, то компьютер, скорее всего, обнаружит, что чаще всего встречаются буквы «e» и «t». Затем компьютер строит таблицу по типу показанному ниже, сопоставляя часто встречающимся символам короткие числовые коды, а редко встречающимся – длинные.

Шаг 3. Файл еще раз преобразуется – путем замены символов соответствующими им числовыми кодами, вычисленными на шаге 2.

пробел	00	T	20	n	40	(	60	**á**	**780**
A	01	U	21	o	41	)	61	**à**	**781**
B	02	V	22	p	42	*	62	**é**	**782**
C	03	W	23	q	43	+	63	**è**	**783**
D	04	X	24	r	44	,	64	**í**	**784**
E	05	Y	25	s	45	-	65	**ì**	**785**
F	06	Z	26	**t**	**9**	.	66	**ó**	**786**
G	07	a	27	u	47	/	67	**ò**	**787**
H	08	b	28	v	48	:	68	**ú**	**788**
I	09	c	29	w	49	;	69	**ù**	**789**
J	10	d	30	x	50	<	**770**	**Á**	**790**
K	11	**e**	**31**	y	51	=	**771**	**À**	**791**
L	12	f	32	z	52	>	**772**	**É**	**792**
M	13	g	33	!	53	**?**	**773**	**È**	**793**
N	14	h	34	“	54	**{**	**774**	**Í**	**794**
O	15	i	35	#	55	**|**	**775**	**Ì**	**795**
P	16	j	36	$	56	**}**	**776**	**Ó**	**796**
Q	17	k	37	%	57	~	**777**	**Ò**	**797**
R	18	l	38	&	58	**Ø**	**778**	**Ú**	**798**
S	19	m	39	’	59	**ø**	**779**	**Ù**	**799**

Числовые коды для трюка «более короткий символ». Отличия от таблицы на стр. 127 показаны полужирным шрифтом. Коды двух самых употребительных букв укорочены ценой удлинения кодов многих редко употребляемых символов. В результате общая длина большинства сообщений уменьшается.

Таблица числовых кодов, построенная на шаге 2, также сохраняется в ZIP-файле – иначе было бы невозможно декодировать (и, стало быть, распаковать) файл. Отметим, что при сжатии разных файлов получаются разные таблицы числовых кодов. В действительности исходный файл разбивается на блоки, и для каждого блока строится собственная таблица кодов. Все это делается эффективно и автоматически, что позволяет добиться отличной степени сжатия для многих типов файлов.

Сжатие с потерей информации: не бесплатный сыр, но отличная сделка

До сих пор мы говорили о так называемом сжатии *без потери информации*. Это название означает, что сжатый файл можно восстановить точно в исходном виде, т. е. ни один знак не будет изменен. Но иногда гораздо полезнее сжатие *с потерей информации*, при котором восстановленный файл оказывается очень похож на исходный, но может не совпадать с ним в точности. Например, с потерей информации часто сжимаются файлы, содержащие изображения или звуковые данные: если две картинки для человека *выглядят* одинаково, то не имеет значения, совпадают файлы, хранящиеся в компьютере и в камере, или нет. То же самое верно и для звуковых данных: неважно, совпадает файл, воспроизводимый цифровым плеером, с файлом на компакт-диске, если на слух они воспринимаются одинаково.

На самом деле, сжатие с потерей информации иногда используется куда более радикальным способом. Все мы видели низкокачественное видео и изображения в Интернете, когда либо картинка размыта, либо качество звука оставляет желать лучшего. Это результат агрессивного применения сжатия с потерей информации, когда целью является максимальное уменьшение размера файла, содержащего видео или изображение. Здесь уже не ставится задача сделать видео неотличимым от оригинала, главное – чтобы вообще хоть что-то можно было различить. Настраивая уровень «потерь» при сжатии, владелец веб-сайта может соблюсти баланс между большим высококачественным файлом, в котором изображение и звук почти идеальны, и низкокачественным файлом с очевидными дефектами, зато требующим гораздо более узкую полосу пропускания при передаче. Возможно, вы делали нечто похожее со своей камерой, которая обычно позволя-

ет настраивать качество фотографий и видео. Если выбрать высокое качество, то в памяти камеры можно будет сохранить меньше фотографий и видео, чем в случае качества пониже. Объясняется это тем, что чем выше качество мультимедийного файла, тем больше места он занимает. А регулируется это настройками уровня потерь при сжатии. В этом разделе мы рассмотрим несколько трюков, применяемых при такой настройке.

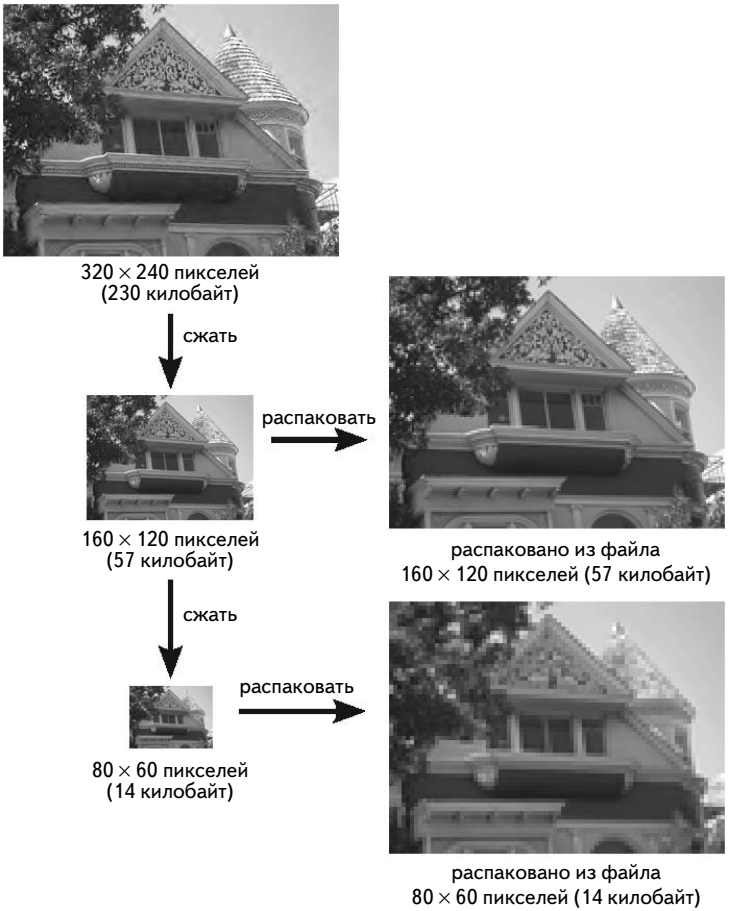
Трюк с пропуском

Простой и полезный трюк, применяемый для сжатия с потерей информации, – пропустить часть данных. Посмотрим, как это работает для черно-белых фотографий. Но сначала нужно понять, как черно-белые фотографии хранятся в компьютере. Фотография состоит из большого количества крохотных точек, называемых пикселями. У каждого пикселя строго определенный цвет – черный, белый или промежуточный оттенок серого. Вообще говоря, мы не думаем о пикселях, потому что они очень маленькие, но их можно увидеть, если рассмотреть картинку на экране монитора или телевизора через лупу.

Каждый пиксель черно-белой фотографии, хранящейся в компьютере, представлен числом. Исключительно для этого примера предположим, что чем больше число, тем ярче цвет, а максимально возможное число равно 100. Таким образом, 100 соответствует белому, 0 – черному, 50 – среднему оттенку серого, 90 – светло-серому и т. д. Пиксели образуют прямоугольную таблицу, состоящую из строк и столбцов, причем каждый пиксель представляет цвет очень малого участка фотографии. Количество строк и столбцов – это «разрешение» изображения. Например, во многих телевизорах высокой четкости разрешение составляет 1920 × 1080, т. е. имеется 1920 столбцов и 1080 строк пикселей. Общее число пикселей на таком экране получается умножением 1920 на 1080 – больше двух миллионов! Словом «мегапиксель» называется просто миллион пикселей. Таким образом, в 5-мегапиксельной камере строк и столбцов столько, что их произведение превышает 5 миллионов. В компьютере фотография хранится в виде списка чисел – по одному для каждого пикселя.

Разрешение фотографии дома с башенкой на левом верхнем рисунке на следующей странице гораздо меньше, чем в телевизоре высокой четкости: всего 320 × 240. Тем не менее, количество пикселей все равно достаточно велико (320 × 240 = 76 800), и размер несжатого файла с этой фотографией больше 230 килобайт. Кстати говоря, килобайт

приблизительно равен 1000 знаков текста – таков, к примеру, размер среднего электронного письма с одним абзацем. Следовательно, можно сказать – хотя это очень грубая оценка – что файл с левой верхней фотографией занимает на диске столько же места, сколько 200 коротких писем.



Сжатие с помощью трюка с пропуском.
В левой колонке показано исходное изображение и два уменьшенных. Каждое уменьшенное изображение получено путем пропуска половины строк и столбцов предыдущего изображения.
В правой колонке мы видим, что получается после распаковки сжатого изображения до исходного размера. Реконструкция не идеальна, реконструированное изображение заметно отличается от оригинала.

Мы можем сжать этот файл с помощью такого чрезвычайно простого приема: проигнорировать, или «пропустить» каждую вторую строку и каждый второй столбец пикселей. Трюк с пропуском действительно на диво прост! В данном случае получится фотография меньшего разрешения – 160 × 120, она показана на рисунке под исходной фотографией. Размер файла составляет примерно четверть от исходного (около 57 килобайт). Это потому, что, уменьшив ширину и высоту изображения вдвое, мы оставили только четверть пикселей.

Мы можем повторить этот трюк. Возьмем новое изображение размером 160 × 120 и еще раз выкинем каждую вторую строку и каждый второй столбец – результат показан на левом нижнем рисунке. Размер изображения снова сократился на 75 %, так что получился файл размером всего 14 килобайт. Это 6 % от исходного размера – впечатляющая степень сжатия.

Но вспомните – мы применяем сжатие с потерей информации, так что ожидать бесплатного сыра не приходится. Сыр дешевый, но заплатить за него все же придется. Взгляните, что происходит, если попытаться восстановить из сжатого файла изображение исходного размера. Поскольку часть строк и столбцов удалена, компьютеру приходится догадываться, какого цвета были отсутствующие пиксели. Простейшая догадка – сделать отсутствующий пиксель такого же цвета, как один из его соседей. Годится любой сосед, но мы взяли пиксель слева сверху от отсутствующего.

Результат такого способа распаковки показан на рисунке в правой колонке. Как видите, большая часть зрительных характеристик сохранена, но качество и детализация определенно пострадали, особенно в таких неоднородных областях, как крона дерева, крыша башенки и резьба на фронтоне дома. Кроме того, особенно на картинке, восстановленной из файла 80 × 60, видны неприятные зубчатые края, например, на диагональных линиях крыши. Все это называется «артефактами сжатия»: не просто утрата деталей, но заметные глазу новые черты, обязанные своим появлением конкретному методу сжатия с потерей информации и последующей распаковке.

Хотя трюк с пропуском полезен для объяснения основной идеи сжатия с потерей информации, на практике он редко применяется в такой простой форме. Компьютеры действительно «пропускают» информацию, чтобы добиться сжатия, но гораздо более осторожны в решениях о том, что именно пропустить. Хорошо известный пример – графический формат со сжатием JPEG. Это тщательно продуманная техника сжатия изображения, дающая куда более качествен-

ные результаты, чем выбрасывание каждой второй строки и каждого второго столбца. Сверху, на следующем рисунке, показано изображение в формате JPEG, размер которого составляет 35 килобайт, но при этом визуально оно практически неотличимо от исходного. Пропустив еще больше информации, но придерживаясь формата JPEG, мы можем уменьшить размер изображения до 19 килобайт (в центре), по-прежнему сохранив отличное качество. Правда, теперь уже видно некоторое размытие и утрата кое-каких деталей резьбы. Но даже JPEG страдает от артефактов сжатия, если степень сжатия слишком велика: внизу показано то же изображение в формате JPEG, сжатое до 12 килобайт, теперь видны пятна света и тени на небе и неприятная пятнистость вблизи диагональной линии крыши.

JPEG (35 kilobytes)

JPEG (19 kilobytes)

JPEG (12 kilobytes)

В случае сжатия с потерей информации чем выше степень сжатия, тем хуже качество. Одно и то же изображение показано при разных уровнях качества JPEG. Качество верхнего изображения самое высокое, но и места для файла требуется больше всего. Изображение внизу самого низкого качества, места оно занимает вдвое меньше, но отчетливо видны артефакты сжатия – особенно на небе и вдоль границы крыши.

Детали стратегии пропуска в формате JPEG слишком сложны, чтобы описывать их здесь в полном объеме, но основной принцип вполне доступен для понимания. Сначала все изображение разбивается на квадратики 8 × 8 пикселей. Каждый квадратик сжимается отдельно. Отметим, что без сжатия для представления квадратика понадобилось бы 8 × 8 = 64 пикселя. (Мы предполагаем, что фотография черно-белая – для цветного изображения понадобилось бы три разных цвета, поэтому количество чисел было бы в три раза больше, но об этой детали мы сейчас забудем.) Если все пиксели в квадратике одного цвета, то квадратик целиком представляется одним числом, так что компьютер может «пропустить» 63 числа. Если в квадратике превалирует один цвет, а отличия только в оттенках (скажем, некоторый участок неба почти одного серого цвета), то компьютер все равно решает представить квадратик одним числом, что дает хорошую степень сжатия данного квадратикв ценой небольшой погрешности при последующей распаковке. На нижнем рисунке на предыдущей странице видно, что некоторые квадратики 8 × 8 сжаты именно так, что привело к появлению небольших квадратных участков одного цвета.

Если в квадратике 8 × 8 имеется плавный переход от одного цвета к другому (скажем, от темно-серого слева к светло-серому справа), то 64 числа можно свести к двум: значения темно-серого и светло-серого. Алгоритм JPEG работает не совсем так, но идея похожа: если квадратик 8×8 достаточно близок к некоторой комбинации известных образцов (например, постоянный цвет или плавно изменяющийся цвет), то большую часть информации можно отбросить, оставив только характерные для образца значения.

JPEG отлично работает для фотографий, но как быть со звуковыми и музыкальными файлами? Они тоже сжимаются с потерей информации, и алгоритм основан на тех же идеях: пропустить информацию, которая мало влияет на результат. В популярных форматах сжатия музыки, например MP3 и AAC, вообще говоря, используется тот же подход, что в JPEG. Аудиофайл разбивается на блоки, и каждый блок сжимается независимо. Как и в JPEG, блоки, которые изменяются предсказуемо, можно описать всего несколькими числами. Но в форматах сжатия звука можно также учесть некоторые особенности человеческого слуха. В частности, определенные звуки человек не различает, поэтому алгоритм сжатия может выкинуть их, не снижая качества результата.

Истоки алгоритмов сжатия

Описанный в этой главе трюк «то же, что раньше» – один из основных методов сжатия, применяемый в ZIP-файлах, – известен в информатике как алгоритм LZ77. Его изобрели два израильских ученых, Абрахам Лемпель и Якоб Зив, опубликован он был в 1977 году. Но если мы хотим проследить истоки алгоритмов сжатия, то нужно углубиться в историю еще на три десятилетия. Мы уже встречались с Клодом Шенноном, ученым из компании Bell Labs, основавшим теорию информации в работе 1948 года. Шеннон был одним из героев рассказа о кодах, исправляющих ошибки (глава 5), но он же со своей статьей 1948 года играет важную роль и в возникновении алгоритмов сжатия.

Это не случайное совпадение. На самом деле, коды, исправляющие ошибки, и алгоритмы сжатия – две стороны одной медали. То и другое сводится к понятию *избыточности*, о котором мы так много говорили в главе 5. Если в файле есть избыточность, значит, он длиннее, чем необходимо. Возвращаясь к примеру из главы 5, рассмотрим файл, в котором вместо числа «5» встречается слово «five». В этом случае ошибку типа «fivq» можно легко обнаружить и исправить. Следовательно, коды, исправляющие ошибки, можно рассматривать как способ *добавления* избыточности в сообщение или файл.

Алгоритмы сжатия делают прямо противоположное: *устраняют* избыточность из сообщения или файла. Легко представить себе алгоритм сжатия, который замечает, что слово «five» часто встречается в файле, и заменяет его более коротким символом (быть может, даже символом «5»). Это в точности обращение процесса кодирования с помощью алгоритма, исправляющего ошибки. На практике, впрочем, сжатие и исправление ошибок не аннулируют друг друга, как описано выше. Хороший алгоритм сжатия устраняет неэффективную избыточность, а кодирование с целью исправления ошибок добавляет другую, более эффективную избыточность. Таким образом, часто сообщение сначала сжимают, а потом добавляют в него данные для исправления ошибок.

Но вернемся к Шеннону. Его основополагающая работа 1948 года наряду с другими необычайно плодотворными идеями содержала описание одного из самых ранних методов сжатия. Профессор Массачусетского технологического института Роберт Фано открыл этот метод примерно в то же время, и теперь он известен под названием кода Шеннона-Фано. На самом деле, код Шеннона-Фано представ-

ляет собой конкретную реализацию трюка «более короткий символ», описанного в этой главе. Как мы скоро увидим, этот код был вскоре вытеснен другим алгоритмом, но сам метод очень эффективен и до сих пор существует как один из возможных способов сжатия в формате ZIP.

Шеннон и Фано знали, что несмотря на эффективность их подхода, он не является наилучшим из возможных. Шеннон математически доказал, что должны существовать лучшие способы сжатия, но не сумел их найти. Тем временем Фано начал читать в МТИ курс по теории информации для студентов старших курсов и поставил задачу об оптимальном сжатии как тему для курсовой работы. Поразительно, но один из его студентов эту задачу решил, предложив метод достижения наилучшего возможного сжатия каждого отдельного символа. Этим студентом был Дэвид Хаффман, а его метод – который теперь называют кодом Хаффмана – является еще одним примером трюка «более короткий символ». Код Хаффмана остается важнейшим алгоритмом сжатия и широко используется в системах связи и хранения данных.

ГЛАВА 8. Базы данных: в поисках непротиворечивости

– Ничего не знаю, ничего! – раздраженно восклицал он. – Когда под рукой нет глины, из чего лепить кирпичи?

— Артур Конан Дойл «*Медные буки*»[1]

Подумайте о таком таинственном обряде. Человек берет со стола специально отпечатанный блокнот (называемый чековой книжкой), пишет в нем какие-то цифры и ставит подпись с росчерком. Затем он вырывает из блокнота верхний лист, кладет его в конверт и наклеивает на конверт еще один кусочек бумаги (называемый *маркой*). Наконец, человек выносит конверт из дома и идет по улице к большому ящику, куда засовывает конверт.

До наступления 21 века этот обряд совершал всякий, кто оплачивал счета: за телефон, за электричество, за обслуживание кредитных карт и т. д. С той поры получили развитие системы онлайнового выставления счетов и онлайнового банкинга. Простота и удобство таких систем сделали прежний бумажный метод трудоемким до нелепости и совершенно неэффективным.

Благодаря каким технологиям произошла такая трансформация? Самый очевидный ответ – благодаря пришествию Интернета, без которого не было бы никаких онлайновых операций. Еще одна ключевая технология – криптография с открытым ключом, которую мы обсуждали в главе 4. Без нее было бы невозможно безопасно передавать через Интернет конфиденциальную финансовую информацию. Но есть еще одна технология, неотъемлемая от онлайновых сделок: *базы данных*. Пользователи компьютеров по большей части пребывают в блаженном неведении о ее существовании, однако же практически

[1] Перевод Н. Л. Емельянниковой. В оригинале «Data! Data! Data!» – буквально «Данные! Данные! Данные!», почему эта фраза и вынесена в эпиграф к главе о базах данных. *Прим. перев.*

все онлайновые сделки обрабатываются с помощью изощренных баз данных, разработанных учеными с 1970-х годов.

Базы данных отвечают за два важнейших аспекта обработки сделок: эффективность и надежность. Эффективность обеспечивается алгоритмами, которые дают тысячам клиентов возможность одновременно совершать сделки, не опасаясь конфликтов или противоречий. А надежность – алгоритмами, благодаря которым данные не пропадают даже в случае поломки дисков компьютера, хотя обычно это приводит к масштабной потере данных. Онлайновый банкинг – канонический пример приложения, которое должно быть исключительно эффективным (чтобы обслуживать одновременно много пользователей без ошибок и не создавая противоречий) и стопроцентно надежным. Поэтому для придания конкретности нашим рассуждениям мы часто будем возвращаться к примеру онлайнового банкинга.

В этой главе мы узнаем о трех фундаментальных – и очень красивых – идеях, лежащих в основе баз данных: упреждающая запись в журнал, двухфазная фиксация и реляционные базы данных. Благодаря этим идеям базы данных стали доминирующей технологией хранения важной информации определенного вида. Как обычно, мы постараемся проникнуть в суть каждой идеи, выделив тот трюк, вследствие которого она работает. Упреждающая запись в журнал основана на трюке со списком дел, им мы займемся в первую очередь. Затем перейдем к протоколу двухфазной фиксации, который можно описать с помощью простого, но действенного трюка «подготовить и зафиксировать». И наконец, мы заглянем в мир реляционных баз данных и узнаем о трюке с виртуальной таблицей.

Но сначала попробуем развеять тайну, окутывающую вопрос о том, что такое вообще база данных. На самом деле, даже в технической литературе по информатике, под термином «база данных» понимают множество разных вещей, поэтому дать единственно верное определение невозможно. Но большинство специалистов согласны в том, что ключевым свойством баз данных, отличающим их от других способов хранения информации, является тот факт, что у информации в базе данных есть предопределенная структура.

Чтобы понять, что здесь означает слово «структура», рассмотрим для начала полную противоположность – пример *неструктурированной* информации:

> *Розине 35 лет, она дружит с Мэттом, которому 26 лет. Инь Ю 32 лет, Судип – 31. Мэтт, Инь Ю и Судип дружат между собой.*

Это как раз та информация о людях, которая хранится на сайтах социальных сетей типа Facebook или MySpace. Но, конечно, она хранится не в таком неструктурированном виде. Вот как можно представить ее в структурированной форме:

Имя	Возраст	Друзья
Розина	35	Мэтт
Инь Ю	37	Мэтт, Судип
Мэтт	26	Розина, Инь Ю, Судип
Судип	31	Инь Ю, Мэтт

В информатике структуру такого типа называют *таблицей*. В каждой строке таблицы находится информация об одном объекте (в данном случае о человеке). В каждом столбце хранится информация определенного вида, например, имя или возраст человека. База данных обычно насчитывает много таблиц, но в первых примерах мы не станем усложнять и ограничимся одной таблицей.

Очевидно, что и человек, и компьютер способны манипулировать данными гораздо эффективнее, если они представлены в виде структурированной таблицы, а не в виде не имеющего никакой структуры текста, как в примере выше. Однако назначение баз данных отнюдь не исчерпывается простотой использования.

Путешествие по миру баз данных мы начнем с простой концепции: непротиворечивости. Как мы скоро увидим, практики, работающие с базами данных, помешаны на непротиворечивости – и не без причины. По существу, «непротиворечивость» означает, что в базе данных нет внутренних противоречий. Противоречия – худший из кошмаров администратора баз данных. Но как вообще могут возникнуть противоречия? Представьте, что мы немного изменим первые две строки показанной выше таблицы:

Имя	Возраст	Друзья
Розина	35	Мэтт, Инь Ю
Инь Ю	37	Мэтт, Судип

Видите, в чем проблема? Согласно первой строке, Розина дружит с Инь Ю. Но если верить второй строке, Инь Ю не дружит с Розиной.

Этим нарушается смысл понятия дружбы, которая по определению взаимна. Впрочем, это еще довольно мягкий пример противоречия. Более серьезная проблема возникнет, если заменить «дружбу» на «брак». Тогда получится, что *A* состоит в брачных отношениях с *B*, а *B* состоит в брачных отношениях с *C* – ситуация, которая во многих странах попросту незаконна.

На самом деле, таких противоречий легко избежать при добавлении в базу новых данных. Компьютеры прекрасно умеют следовать правилам, поэтому нетрудно прописать в базе данных правило «если *A* состоит в брачных отношениях с *B*, то *B* состоит в брачных отношениях с *A*». При попытке добавить строку, нарушающую это правило, будет выдано сообщение об ошибке, и строка не добавится. Поэтому обеспечение непротиворечивости, основанное на соблюдении простых правил, не нуждается в каком-то хитром трюке.

Но бывают и другие типы противоречий, для предотвращения которых нужны куда более изобретательные решения. Ниже мы рассмотрим один такой пример.

Транзакции и трюк со списком дел

Транзакции – пожалуй, самая важная идея в мире баз данных. Но чтобы понять, что это такое и зачем они нужны, нужно знать два факта о компьютерах. С первым фактом все, вероятно, знакомы: компьютерные программы иногда «падают», а упавшая программа забывает обо всем, что она делала. Остается только та информация, которая была сохранена в файловой системе. Второй факт – менее известный, но очень важный: компьютерные устройства хранения, например жесткие диски и флэшки, за одну операцию записывают лишь небольшую порцию данных – обычно около 500 знаков. (Кому интересен технический жаргон, знайте, что я имею в виду *размер сектора* жесткого диска, обычно равный 512 байтам. В случае флэшки соответствующая величина называется размером страницы и тоже составляет от нескольких сотен до нескольких тысяч байтов.) Пользователи компьютера не замечают малости этой величины, потому что современные диски способны выполнять сотни тысяч таких операций записи в секунду. Но факт остается фактом: за одну операцию содержимое диска может измениться не более чем на несколько сотен знаков.

И какое отношение это имеет к базам данных? А самое прямое: как правило, компьютер может обновить за раз только одну строку табли-

цы в базе. На нашем небольшом простеньком примере этого, к сожалению, не видно. Вся таблица содержит меньше 200 знаков, поэтому в данном случае компьютер *мог бы* обновить две строки сразу. Но, вообще говоря, в базе данных сколько-нибудь значительного размера для изменения двух строк нужны две отдельные дисковые операции.

Установив эти непреложные факты, мы можем перейти к существу вопроса. Как выясняется, многие кажущиеся простыми изменения базы данных требуют модификации двух и более строк. А, как мы теперь знаем, две разные строки нельзя изменить одной дисковой операцией, поэтому обновление сводится к последовательности как минимум двух операций. Однако компьютер может в любой момент «накрыться». И что будет, если это случится *между* двумя дисковыми операциями? Компьютер-то можно перезагрузить, но он забудет обо всех операциях, которые собирался выполнить, поэтому некоторые важные изменения могут так никогда и не произойти. Иными словами, база данных может оказаться в противоречивом состоянии!

Возможно, сейчас вам кажется, будто проблема противоречивости носит академический характер. Но давайте рассмотрим два примера ее проявления. Начнем с еще более простой базы данных, чем раньше.

Имя	**Друзья**
Розина	нет
Инь Ю	нет
Мэтт	нет

В этой унылой и наводящей тоску базе данных перечислены три одиноких человека. А теперь предположим, что Розина и Инь Ю подружились, и мы хотим обновить базу данных, отразив это радостное событие. Для обновления придется изменить первую и вторую строки таблицы, а это, как мы уже знаем, требует как правило двух отдельных дисковых операций. Предположим, что строка 1 обновляется первой. Сразу после этого обновления, еще до того как у компьютера появился шанс выполнить вторую дисковую операцию для обновления строки 2, база данных выглядит так:

Имя	**Друзья**
Розина	Инь Ю
Инь Ю	нет
Мэтт	нет

Пока все хорошо. Программе осталось только обновить строку 2, и все будет в порядке. Но секундочку – а что если компьютер «накроется», не успев это сделать? Тогда после перезагрузки он не будет знать, что строка 2 осталась не обновленной. База данных окажется точно в таком состоянии, как показано выше: Розина дружит с Инь Ю, но Инь Ю *не* дружит с Розиной. Налицо пугающее противоречие.

Я уже упоминал, что люди, работающие с базами данных, помешаны на непротиворечивости, но в этом примере беда может показаться не такой уж страшной. В конце концов, так ли важно, что Инь Ю помечена как друг в одном месте и как не имеющая друзей – в другом? Можно даже вообразить автоматическую программу, которая периодически просматривает базу данных, находит и исправляет подобные несоответствия. Больше скажу – такие программы действительно существуют и могут применяться в базах данных, для которых непротиворечивость не стоит на первом месте. Возможно, вы даже встречались с ними, поскольку в некоторых операционных системах в процессе перезагрузки после аварийной остановки запускается проверка всей файловой системы на предмет отсутствия противоречий.

Но существуют ситуации, когда противоречивость – настоящее зло, и автоматизированный инструмент не поможет. Классический пример – перевод денег с одного банковского счета на другой. Вот еще одна простая база данных:

Владелец счета	Тип счета	Баланс счета
Зади	текущий	$800
Зади	сберегательный	$300
Педро	текущий	$150

Предположим, что Зади попросила перевести 200 долларов со своего текущего счета на сберегательный. Как и в предыдущем примере, мы должны будем обновить две строки с помощью двух последовательных дисковых операций. Сначала баланс текущего счета уменьшается до 600 долларов, а затем баланс сберегательного счета увеличивается до 500 долларов. И если нам не повезет, и между двумя этими операциями компьютер накроется, то база данных будет выглядеть так:

Владелец счета	Тип счета	Баланс счета
Зади	текущий	$600
Зади	сберегательный	$300
Педро	текущий	$150

Для Зади это катастрофа: до сбоя на двух ее счетах было 1100 долларов, а теперь осталось только 900. Она не снимала деньги со счета, но 200 долларов попросту испарились! И обнаружить этот факт нельзя, потому что после сбоя в базе данных нет никаких противоречий. Здесь мы столкнулись с куда более тонким типом противоречивости: новое состояние базы данных противоречит ее состоянию *до* сбоя.

Стоит рассмотреть этот важный момент более подробно. В первом нашем примере база данных оказалась в очевидно противоречивом состоянии: *A* дружит с *B*, но *B* не дружит с *A*. Такой тип противоречивости можно обнаружить, исследовав базу данных (хотя процесс исследования может занять очень длительное время, если база содержит миллионы – или даже миллиарды – записей). Во втором примере состояние базы данных вполне правдоподобно, если рассматривать его в любой отдельный момент времени. Не существует правила, которое бы говорило, каким должен быть баланс или соотношение между балансами. Но если взять состояние базы данных за период времени, то противоречивость бросается в глаза. Существенны три факта: (1) до перевода у Зади было 1100 долларов; (2) после сбоя у нее осталось 900 долларов; (3) между этими событиями она не снимала деньги со счетов. В совокупности эти факты противоречивы, однако противоречие невозможно обнаружить, исследуя состояние базы данных в любой конкретный момент времени.

Чтобы избежать подобных противоречий, ученые ввели понятие «транзакции» – набора изменений базы данных, которые должны произойти все вместе, чтобы база данных оставалась непротиворечивой. Простая, но чрезвычайно полезная идея. Программист базы данных может выполнить команду «начать транзакцию», затем внести ряд взаимозависимых изменений и закончить командой «завершить транзакцию». База данных гарантирует, что все изменения будут внесены, даже если компьютер, на котором работает база, накроется и будет перезапущен в середине транзакции.

Чтобы быть совсем уж точными, нужно упомянуть и о второй возможности: не исключено, что после сбоя и перезапуска база дан-

ных вернется в состояние до начала транзакции. Но если такое случится, то программист получит уведомление о том, что транзакция не выполнена и должна быть повторена, так что никакого вреда не будет. Эту возможность мы подробнее обсудим ниже, в разделе, посвященном «откату» транзакций. А пока подчеркнем ключевой момент: база данных остается непротиворечивой независимо от того, завершена транзакция или откачена.

Из приведенного описания может создаться впечатление, что мы зря уделяем столько внимания возможности сбоев – ведь в современных операционных системах с современными приложениями это бывает очень редко. На это есть два возражения. Во-первых, понятие «сбоя» весьма общее: оно охватывает любые причины, по которым компьютер перестает работать, что грозит потерей данных. Сюда входят и сбои в электроснабжении, и отказы дисков или другого оборудования, и ошибки в операционной системе или прикладных программах. Во-вторых, даже если такие обобщенные сбои редки, некоторые базы данных все равно не могут позволить ни малейшего риска: банки, страховые компании и иные организации, имеющие дело с денежными средствами, должны обеспечить непротиворечивость данных при любых обстоятельствах.

Описанное выше простое решение (начать транзакцию, выполнить произвольное количество операций и завершить транзакцию) может показаться слишком хорошим, чтобы быть правдой. Но на самом деле, его можно реализовать с помощью сравнительного простого трюка со списком дел.

Трюк со списком дел

Не все мы умеем четко организовывать свою жизнь. Но повезло нам в этом отношении или нет, все мы знакомы с орудием, которым пользуются организованные люди: списком предстоящих дел. Даже если лично вам вести такой список не нравится, возражать против его полезности трудно. Если за день нужно сделать 10 дел, то начать стоит с того, чтобы их записать – и лучше бы в порядке, гарантирующем эффективность. Список дел особенно полезен, если вам случается на что-то отвлечься (не сказать ли – «в случае сбоя»?) посередь дня. Если вы по какой-то причине забудете, что осталось сделать, то достаточного беглого взгляда на список, чтобы освежить память.

Транзакции баз данных – это тоже особого рода список дел. Потому-то мы и говорим о трюке со списком дел, хотя специалисты по информатике применяют термин «упреждающая запись в журнал».

Основная идея заключается в том, чтобы вести журнал действий, которые базе данных предстоит выполнить. Журнал хранится на жестком диске или на каком-нибудь другом постоянном запоминающем устройстве, поэтому информация не пропадает после сбоя и перезапуска. Все операции, входящие в состав транзакции, сначала записываются в журнал и, стало быть, сохраняются на диске и только потом выполняются. Если транзакция завершается успешно, то можно удалить связанный с ней список дел из журнала и тем самым сэкономить место. Таким образом, описанная выше транзакция перевода денег Зади должна быть разбита на два шага. Сначала, не трогая таблицу базы данных, мы записываем список дел транзакции в журнал.

Владелец счета	Тип счета	Баланс счета
Зади	текущий	$800
Зади	сберегательный	$300
Педро	текущий	$150

Журнал с упреждающей записью

1. Начать транзакцию перевода.
2. Изменить баланс текущего счета с $800 на $600.
3. Изменить баланс сберегательного счета с $300 на $500.
4. Завершить транзакцию перевода.

Убедившись, что все записи в журнал сохранены на постоянном запоминающем устройстве, например на диске, мы можем внести запланированные изменения в саму таблицу.

Владелец счета	Тип счета	Баланс счета
Зади	текущий	$600
Зади	сберегательный	$500
Педро	текущий	$150

Журнал с упреждающей записью

1. Начать транзакцию перевода.
2. Изменить баланс текущего счета с $800 на $600.
3. Изменить баланс сберегательного счета с $300 на $500.
4. Завершить транзакцию перевода.

В предположении, что все изменения сохранены на диске, записи можно удалить из журнала.

Но это был простой случай. А что если компьютер в середине транзакции неожиданно «грохнется»? Как и раньше, предположим, что это произошло после дебетования текущего счета, но до кредитования сберегательного. Компьютер перезагружается, база данных перезапускается и обнаруживает на жестком диске такую информацию:

Владелец счета	Тип счета	Баланс счета
Зади	текущий	$600
Зади	сберегательный	$300
Педро	текущий	$150

Журнал с упреждающей записью

1. Начать транзакцию перевода.
2. Изменить баланс текущего счета с $800 на $600.
3. Изменить баланс сберегательного счета с $300 на $500.
4. Завершить транзакцию перевода.

Теперь компьютер может определить, что в момент сбоя находился в середине транзакции, потому что в журнале имеется информация об этом. Однако же в журнале перечислены четыре запланированных действия. Как узнать, какие уже применены к базе данных, а какие еще нет? Ответ восхитительно прост: а неважно! Причина в том, что записи в журнале базы данных устроены так, что дают один и тот же результат, сколько бы раз их ни применять: один, два, сколько угодно.

Технический термин для этого свойства – *идемпотентный*, так что специалист скажет, что все действия, записанные в журнале, должны быть идемпотентными. Для примера возьмем запись с номером 2 «Изменить баланс текущего счета с $800 на $600». Неважно, сколько раз в баланс записывалось значение $600, – результат будет один и тот же. Поэтому если в процессе восстановления после сбоя база данных видит такую запись в журнале, она спокойно выполняет действие, не заботясь о том, было оно уже выполнено до сбоя или нет.

Следовательно, при восстановлении после сбоя база данных может просто воспроизвести все действия, хранящиеся в журнале для уже завершенных транзакций. С незавершенными транзакциями тоже все просто. Любой набор действий в журнале, который не завершается командой «завершить транзакцию», нужно выполнить в обратном порядке, оставив базу в состоянии, предшествующем началу транзакции. Мы еще вернемся к вопросу об «откате» транзакций, когда будем обсуждать репликацию.

Атомарность в большом и в малом

Существует еще один способ взглянуть на транзакции: с точки зрения пользователя базы данных любая транзакция *атомарна*. Физики, правда, уже много десятков лет умеют расщеплять атом, но изначально древнегреческое слово *атомарный* означает «неделимый». Когда в информатике говоря «атомарный», подразумевают именно это значение. Таким образом, атомарную транзакцию нельзя разбить на

более мелкие операции: либо вся транзакция целиком завершается успешно, либо база данных остается в исходном состоянии, как будто транзакция никогда и не начиналась.

Поэтому трюк со списком дел дает атомарные транзакции, которые, в свою очередь, гарантируют непротиворечивость. Это ключ к нашему каноническому примеру: эффективной и полностью надежной базе данных для онлайнового банкинга. Но мы еще не решили эту задачу. Сама по себе непротиворечивость не обеспечивает ни эффективности, ни надежности. Однако в сочетании с техникой блокировки, которую мы скоро опишем, трюк со списком дел позволяет обеспечить непротиворечивость, даже когда к базе данных одновременно обращаются тысячи клиентов. Вот это *действительно* означает потрясающую эффективность, потому что параллельно обслуживается много клиентов. И к тому же трюк со списком дел обеспечивает высокую надежность, так как предотвращает возникновение противоречий. Точнее, этот трюк препятствует *повреждению* данных, хотя не исключает *потерю* данных. Наш следующий трюк – «подготовить и зафиксировать» – знаменует значительный прогресс на пути продвижения к цели: предотвращению потери данных.

Трюк «подготовить и зафиксировать» для реплицированных баз данных

Путешествие в мир изобретательных методов работы с базами данных мы продолжим знакомством с алгоритмом, который будем называть трюком «подготовить и зафиксировать». Чтобы понять, зачем он нужен, необходимо усвоить еще два факта: во-первых, базы данных часто *реплицируются*, т. е. создается несколько копий одной и той же базы в разных местах, а, во-вторых, транзакции баз данных иногда необходимо отменять – эту операцию называют «откатом». Прежде чем переходить к трюку «подготовить и зафиксировать», мы вкратце рассмотрим обе эти концепции.

Реплицированные базы данных

Трюк со списком дел позволяет восстановить базу данных после некоторых видов сбоев – путем завершения или отката транзакций, которые были активны в момент сбоя. Однако при этом предполагается, что все данные, которые были сохранены перед сбоем, по-пре-

жнему в наличии. А что, если диск компьютера совсем сломался, и часть или вообще все данные оказались потеряны? Это лишь одна из многих ситуаций постоянной утраты данных. Есть и другие: ошибки в программе (в самой программе управления базой данных или в операционной системе) и аппаратные сбои. В любом из этих случаев компьютер можете перезаписать данные, которые вы считали надежно сохраненными на диске, – стереть их или заменить мусором. Очевидно, что трюк со списком дел здесь не поможет.

Однако в некоторых случаях потеря данных абсолютно недопустима. Если банк потеряет информацию о вашем счете, вы будете очень недовольны, а банк ждут серьезные юридические и финансовые неприятности. То же самое касается брокерской компании, которая обязалась выполнить ваш заказ на продажу акций, но потеряла сведения о сделке. На самом деле, любая компания с большим объемом продаж через Интернет (у всех на слуху примеры eBay и Amazon) просто не может позволить себе потерю или искажение сведений о своих клиентах. Однако в большом центре обработки данных, где стоят тысячи компьютеров, какие-то компоненты (особенно жесткие диски) выходят из строя каждый день. И значит, данные, хранившиеся на этих компонентах, тоже теряются каждый день. И как в таких непростых условиях банк умудряется сохранить ваши данные в безопасности?

Очевидное и широко применяемое решение – иметь две или больше копий базы данных. Каждая копия называется *репликой*, а совокупность всех копий – *реплицированной базой данных*. Часто реплики территориально разнесены (быть может, находятся в разных центрах обработки данных, отстоящих друг от друга на сотни километров), поэтому если одна пропадет в результате стихийного бедствия, останется вторая.

Я присутствовал на встрече, где директор одной компьютерной компании рассказывал о том, что случилось с ее клиентами после террористической атаки на Всемирный торговый центр в Нью-Йорке 11 сентября 2001 года. В башнях-близнецах у компании было пять крупных клиентов, и все они работали с территориально разнесенными реплицированными базами данных. Четыре из пяти клиентов смогли продолжить работу практически без перерыва, пользуясь уцелевшими репликами. У пятого, к несчастью, было по одной реплике в каждой башне, и обе пропали! Этот клиент смог возобновить работу только после восстановления базы данных из хранящейся в другом месте архивной резервной копии.

Отметим, что реплицированная база данных – совсем не то, что «резервное копирование» данных. Резервная копия – это мгновенный снимок данных *в конкретный момент времени*; в случае ручного копирования это снимок на момент запуска программы резервного копирования, а в случае автоматизированного – на момент, когда программа запускается по расписанию, скажем в 2 часа ночи ежедневно или еженедельно. Иными словами, резервная копия – это полный дубликат каких-то файлов, базы данных или еще чего-то, что вы хотите сохранить на всякий случай.

По самому своему определению, резервная копия может быть неактуальной: любые изменения, внесенные в базу данных после копирования, нигде больше не сохранены. Напротив, в реплицированной базе данных все изменения в любой момент времени синхронизированы. Как только какая-нибудь запись базы данных хоть немного изменяется, изменения сразу же отражаются во всех репликах.

Понятно, что репликация – отличный способ застраховаться от потери данных. Но у репликации есть и свои проблемы: новый тип противоречивости. Что делать, если по какой-то причине данные в двух репликах различаются? Такие реплики противоречат друг другу, и очень трудно, а подчас и невозможно, определить, в какой из них данные правильны. Мы вернемся к этому вопросу после обсуждения отката транзакций.

Откат транзакций

Рискуя показаться назойливым, я все же повторю, что такое транзакция: это набор изменений в базе данных, которые должны быть выполнены *все вместе*, если мы хотим, чтобы база осталась непротиворечивой. Выше нас интересовал в основном вопрос о том, как завершить транзакцию, даже если в середине нее база данных «грохнулась».

Но оказывается, что иногда по какой-то причине завершить транзакцию невозможно. Например, из-за того что в ходе транзакции в базу добавляется так много данных, что на диске заканчивается место. Это очень редкий, но от того не менее важный, случай.

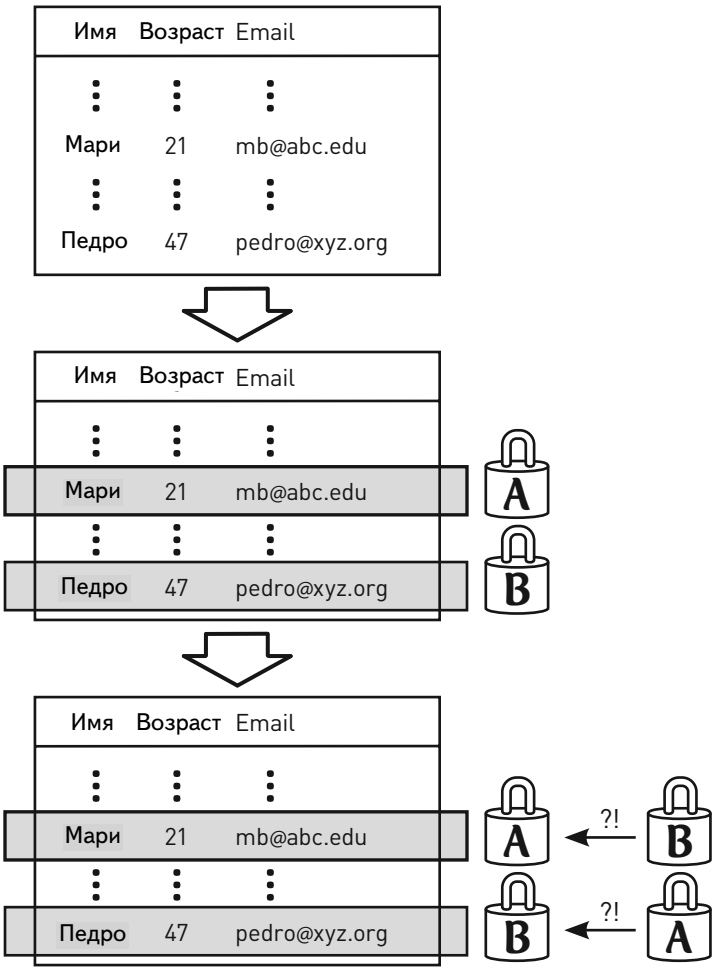
Гораздо более распространенная причина невозможности завершить транзакцию связана с концепцией *блокировки*. В активно используемой базе данных одновременно может выполняться много транзакций. (Подумайте, что случилось бы, если бы ваш банк разрешал переводить деньги только одному пользователю в каждый момент времени – производительность такой системы онлайнового банкинга

оказалась бы ужасающей.) Но часто бывает необходимо заморозить некоторую часть базы данных на время выполнения транзакции. Например, если транзакция *A* обновляет запись, дабы отразить тот факт, что Розина подружилась с Инь Ю, то получилось бы совсем нехорошо, если бы транзакция *B* одновременно удалила Инь Ю из базы данных. Поэтому транзакция *A* «блокирует» часть базы, в которой хранится информация об Инь Ю. Это означает, что данные заморожены, и никакая другая транзакция не может их изменить. В большинстве баз данных транзакции могут блокировать отдельные строки, отдельные столбцы или всю таблицу целиком. Очевидно, что в каждый момент времени одну и ту же часть базы данных может заблокировать только одна транзакция. После успешного завершения транзакция «разблокирует» все данные, которые были заблокированы, и затем другие транзакции могут беспрепятственно эти данные изменять.

На первый взгляд, отличное решение, но оно может привести к крайне неприятной ситуации, которую в информатике называют *взаимоблокировкой*; она показана на рисунке ниже. Допустим, что две длинные транзакции *A* и *B*, выполняются одновременно. Первоначально (верхний рисунок) ни одна строка в базе данных не заблокирована. Затем (средний рисунок) *A* блокирует строку с информацией о Мари, а *B* – строку с информацией о Педро. Чуть позже *A* обнаруживает, что должна заблокировать строку, относящуюся к Педро, а *B* – что необходимо заблокировать строку с данными о Мари. Эта ситуация показана на нижнем рисунке. Итак, *A* хочет заблокировать строку Педро, но в каждый момент времени блокировать строку может только одна транзакция, а *B* уже заблокировала строку Педро! Поэтому *A* должна ждать завершения *B*. Однако *B* не может завершиться, пока не заблокирует строку Мари, которая в данный момент заблокирована транзакцией *A*. Таким образом, *B* должна ждать завершения *A*. Сейчас *A* и *B* *взаимно блокированы*, потому что каждая может продолжить выполнение, только после того как другая завершится. Они будут ждать друг друга вечно, и ни одна не дойдет до конца.

Взаимоблокировки изучены очень детально, во многих базах данных периодически запускается специальная процедура для их обнаружения. Если найдена взаимоблокировка, то одна из виновных в ней транзакций попросту отменяется, чтобы другая могла продолжиться. Но, как и в случае с нехваткой места на диске в середине транзакции, для отмены необходим механизм, который бы «откатывал» еще не завершенную транзакцию. Итак, нам известны по меньшей мере две причины, по которым может возникнуть необходимость в откате

транзакции. Есть еще много других, но нам вдаваться в такие детали нет необходимости. Важно запомнить, что транзакции часто не могут успешно завершиться по непредсказуемой заранее причине.



Взаимоблокировка: когда две транзакции A и B одновременно пытаются заблокировать одни и те же строки – но в разном порядке – они блокируют друг друга, так что ни одна не может продвинуться дальше.

Для реализации отката нужно слегка изменить трюк со списком дел: в журнале с упреждающей записью должно быть достаточно дополнительной информации, чтобы при необходимости *отменить*

любую операцию. (Сравните с приведенным ранее описанием, где мы говорили, что в журнале должно быть достаточно информации, чтобы *повторить* любую операцию после сбоя.) Этого легко добиться на практике. На самом деле, в рассмотренных выше простых примерах информация для отмены и повтора одна и та же. Операцию «Изменить баланс текущего счета с $800 на $600» легко отменить – нужно лишь изменить баланс текущего счета с $600 на $800. Подведем итог: чтобы откатить транзакцию, программа управления базой данных может пробежаться по журналу с упреждающей записью (списку дел) и заменить каждую входящую в состав транзакции операцию на противоположную.

Трюк «подготовить и зафиксировать»

А теперь подумаем о проблеме отката транзакций в *реплицированной* базе данных. Сложность заключается в том, в одной из реплик при откате может возникнуть ошибка, а в других – нет. Например, так случится, если в одной из реплик закончилось место на диске, а у остальных его еще достаточно.

Нам поможет простая аналогия. Допустим, что вы с тремя друзьями хотели бы все вместе посмотреть недавно вышедший в фильм. Чтобы было интереснее, предположим, что это происходит в 1980-е годы, когда электронной почты еще не было, поэтому поход в кино приходится согласовывать по телефону. Как решить эту задачу? Вот один из возможных подходов. Определитесь, какой день и час подходит лично вам и – насколько вам известно – может подойти вашим друзьям. Пусть это будет четверг, 8 вечера. Далее нужно позвонить кому-то из друзей и спросить, свободен ли он в 8 вечера в четверг. Если да, вы говорите «отлично, зарезервируй это время, а я попозже перезвоню и подтвержу». Потом вы звоните следующему другу и делаете то же самое. Ну и напоследок обращаетесь к третьему другу с тем же предложением. Если все свободны в указанное время, то решение считается принятым, и вы звоните каждому по очереди, чтобы подтвердить встречу.

Это простой случай. А если кто-то не готов встречаться в 8 вечера в четверг? В таком случае придется «откатить» всю проделанную к этому моменту работу и начать заново. На практике вы, скорее всего, позвонили бы каждому другу и тут же предложили новый день и время, но для простоты будем считать, что вы сначала звоните со словами: «Извини, 8 вечера в четверг не годится, вычеркни это из ка-

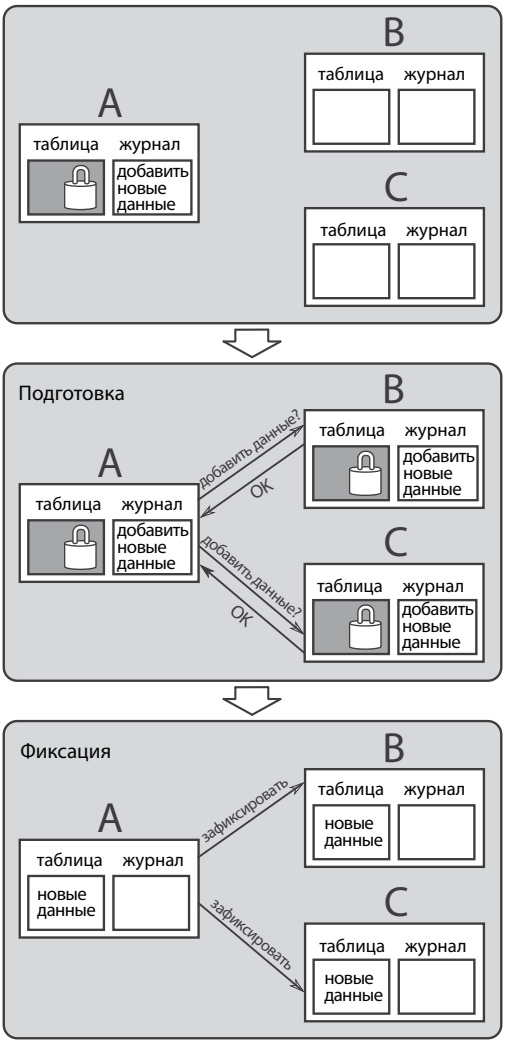
лендаря, а я тебе скоро перезвоню и скажу новое время». Покончив с этим, вы повторяете всю процедуру.

Обратите внимание, что в этой стратегии согласования похода в кино есть две разных фазы. На первой фазе вы предлагаете дату и время, но еще не уверены в них на 100 %. Выяснив, что предложение всех устраивает, *вы* знаете, что дата и время точно правильны, но остальные – еще нет. Поэтому нужна вторая фаза, когда вы обзваниваете всех друзей и подтверждаете время встречи. Если же хотя бы один человек не сможет прийти в это время, то на второй фазе вы обзваниваете всех, кому успели до этого позвонить, и отменяете встречу. В информатике это называется *протоколом двухфазной фиксации*, а мы придумали название «трюк подготовить и зафиксировать». Первая фаза называется «подготовкой», вторая – «фиксацией» или «отменой» в зависимости от того, было предложение принято всеми или нет.

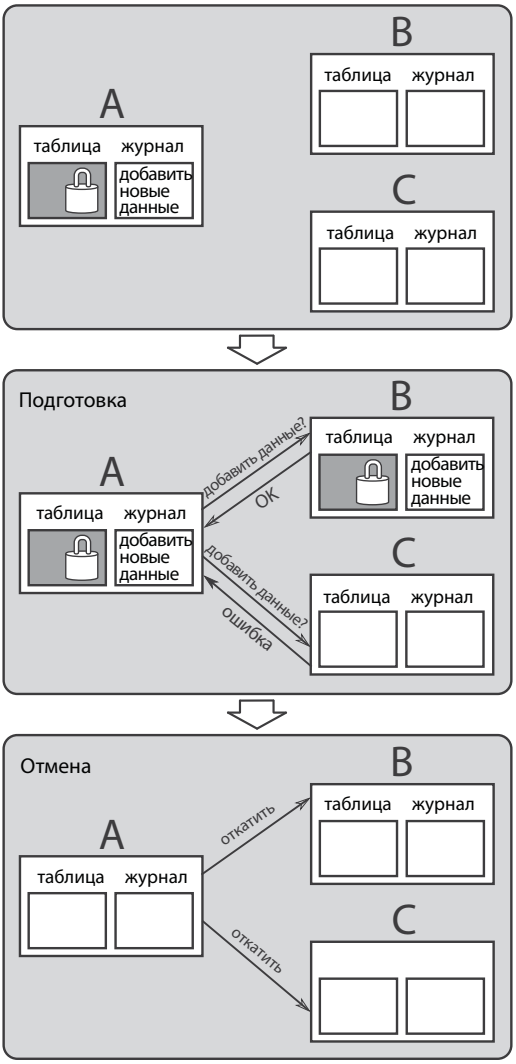
Интересно, что в этой аналогии присутствует понятие блокировки базы данных. Хотя явно мы об этом не упомянули, но каждый из друзей, резервируя время для похода в кино, неявно дает обещание больше ни с кем на это время не договариваться. И до тех пор пока вы не подтвердите или не отмените мероприятие, соответствующая графа в календаря остается «заблокированной» и не может быть изменена никакой другой «транзакцией». В ответ на другие предложения ваш друг должен отвечать: «Извини, у меня на это время другие планы. И пока не буду знать точно, не могу ничего обещать насчет поиграть в баскетбол».

Теперь посмотрим, как трюк «подготовить и зафиксировать» работает в реплицированной базе данных. Идея показана на рисунке ниже. Как правило, одна из реплик является «главной», то есть координирует транзакцию. Для определенности предположим, что есть три реплики *A*, *B* и *C*, причем *A* является главной. Допустим также, что необходимо выполнить транзакцию, которая вставляет новую строку в таблицу. Фаза подготовки начинается с того, что *A* блокирует эту таблицу, а затем записывает новые данные в журнал с упреждающей записью. Одновременно *A* отправляет новые данные репликам *B* и *C*, каждая из которых блокирует свои копии таблицы и записывает новые данные в свои журналы. После этого *B* и *C* сообщают *A*, удалось им сделать свое дело или нет. Затем начинается вторая фаза. Если хотя бы одна реплика столкнулась с проблемой (например, нехваткой места на диске или невозможностью заблокировать таблицу), то главная реплика *A* будет знать, что транзакцию необходимо откатить, и проинформирует об этом все остальные реплики (см. рисунок на

стр. 157). Если же все реплики уведомили об успешном завершении фазы подготовки, то *A* пошлет им сообщение, подтверждающее транзакцию, и реплики ее завершат (см. рисунок ниже).



Трюк «подготовить и зафиксировать»: главная реплика A координирует две другие реплики (B, C), чтобы добавить в таблицу новые данные. На фазе подготовки главная реплика проверяет, все ли реплики готовы завершить транзакцию. Когда все согласовано, главная реплика приказывает остальным зафиксировать данные.



Трюк «подготовить и зафиксировать» с откатом: верхний рисунок такой же, как и выше. Но в фазе подготовки одна из реплик сообщает об ошибке. Поэтому на нижнем рисунке изображена фаза «отмены», в ходе которой каждая реплика откатывает транзакцию.

Итак, в нашем распоряжении имеются два трюка для работы с базами данных: трюк со списком дел и трюк «подготовить и зафиксировать». Что это нам дает? Применяя сочетание обоих трюков, банк – да

и вообще любая онлайновая компания – может реализовать реплицированную базу данных с атомарными транзакциями. А это дает возможность одновременно и эффективно обслуживать тысячи клиентов, не оставляя практически никаких шансов непротиворечивости или потере данных. Однако мы еще не заглядывали в сердце базы данных: как организованы данные и как обрабатываются запросы? Последний наш трюк даст ответы на эти вопросы.

Реляционные базы данных и трюк с виртуальной таблицей

До сих пор все наши базы данных состояли из одной таблицы. Но истинная мощь современной технологии баз данных проявляется, когда в базе несколько таблиц. Основная идея состоит в том, что в каждой таблице хранится разная информация, но эти таблицы каким-то образом связаны между собой. Так, в базе данных компании могут быть отдельные таблицы с данными о заказчиках, о поставщиках и об изделиях. Но в таблице заказчиков могут упоминаться какие-то элементы из таблицы изделий, потому что заказчики заказывают изделия. А, быть может, в таблице изделий упоминаются элементы из таблицы поставщиков, потому что изделия изготавливаются из материалов, поставляемых поставщиками.

Рассмотрим небольшой, но реалистичный пример: информация из базы данных колледжа о том, какие курсы посещают студенты. Чтобы пример был обозримым, количество студентов и курсов невелико, но, надеюсь, будет понятно, что те же принципы применимы и тогда, когда данных гораздо больше.

Во-первых, разберемся, как можно было бы хранить данные в одной таблице, – как мы делали до сих пор. Это показано на верхнем рисунке ниже. Как видите, таблица состоит из 10 строк и 5 столбцов; чтобы измерить объем информации в базе данных, можно просто сказать, что в ней $10 \times 5 = 50$ элементов. Взгляните на этот рисунок более пристально. Вас ничего не раздражает? Например, что вы скажете о многократно повторяющихся данных? Нет ли более эффективного способа сохранить ту же информацию? Возможно, вы заметили, что информация о курсе дублируется для каждого студента, который этот курс посещает. Так, три студента ходят на курс ARCH101, и вся информация о нем (название, преподаватель, номер аудитории) присутствует в записи о каждом из этих студентов. Куда эффективнее

было бы завести две таблицы: одну с данными о том, на какие курсы ходят студенты, а другую – с данными о самих курсах. Такое решение с двумя таблицами показано на нижнем рисунке.

Имя студента	Номер курса	Название курса	Преподаватель	Номер аудитории
Франческа	ARCH101	Введение в археологию	Проф. Блэк	610
Франческа	HIST256	История Европы	Проф. Смит	851
Сюзанна	MATH314	Дифференциальные уравнения	Проф. Кирби	560
Эрик	MATH314	Дифференциальные уравнения	Проф. Кирби	560
Луиджи	HIST256	История Европы	Проф. Смит	851
Луиджи	MATH314	Дифференциальные уравнения	Проф. Кирби	560
Билл	ARCH101	Введение в археологию	Проф. Блэк	610
Роза	MATH314	Дифференциальные уравнения	Проф. Кирби	560
Роза	ARCH101	Введение в археологию	Проф. Блэк	610

Имя студента	Номер курса
Франческа	ARCH101
Франческа	HIST256
Сюзанна	MATH314
Эрик	MATH314
Луиджи	HIST256
Луиджи	MATH314
Билл	ARCH101
Роза	MATH314
Роза	ARCH101

Номер курса	Название курса	Преподаватель	Номер аудитории
ARCH101	Введение в археологию	Проф. Блэк	610
HIST256	История Европы	Проф. Смит	851
MATH314	Дифференциальные уравнения	Проф. Кирби	560

Вверху: база данных о студентах и курсах, состоящая из одной таблицы. Внизу: те же данные, организованные более эффективно – в виде двух таблиц.

Одно из преимуществ подхода с несколькими таблицами очевидно: общий объем потребной для хранения памяти уменьшился. В новом варианте есть одна таблица из 10 строк и 2 столбцов ($10 \times 2 = 20$ элементов) и еще одна из 3 строк и 4 столбцов ($3 \times 4 = 12$ элементов), всего 32 элемента. Напомним, что в первом случае для хранения той же информации нам понадобилось 50 элементов.

Откуда взялась экономия? Потому что мы устранили дублирование информации: теперь название курса, преподаватель и номер аудитории не повторяются в записи о каждом студенте, посещающем этот курс, а хранятся ровно в одном месте. Но кое-чем для этого пришлось пожертвовать: теперь в двух разных местах присутствует столбец «номер курса». Таким образом, мы обменяли *большое* число повторений (полная информация о курсе) на *меньшее* (только номер курса). В целом сделка получилась выгодной. В этом крохотном примере выигрыш невелик, но если представить, что каждый курс могут посещать сотни студентов, то экономия на памяти может оказаться очень существенной.

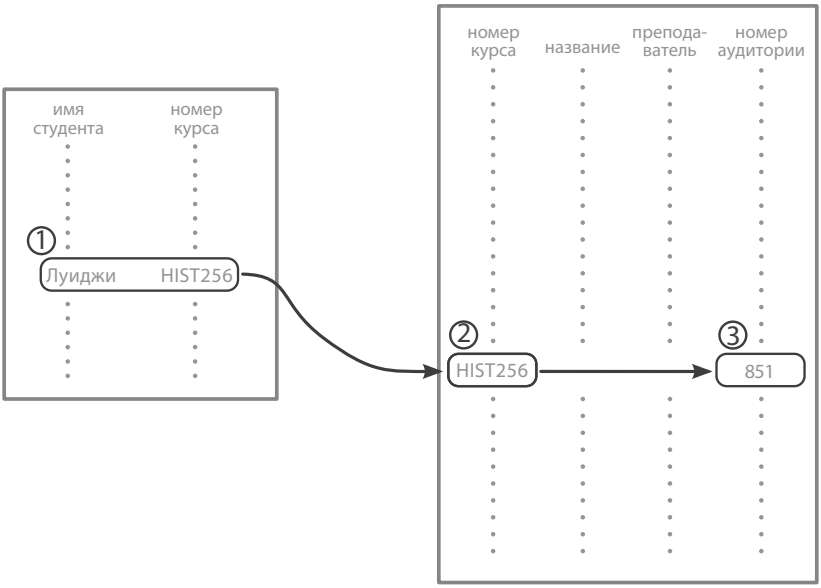
Решение с несколькими таблицами обладает еще одним большим преимуществом. Если таблицы правильно спроектированы, то вносить изменения в базу данных очень просто. Пусть, например, номер аудитории для курса MATH314 сменился с 560 на 440. При наличии одной таблицы нам пришлось бы обновить четыре разных строки, а, как мы говорили выше, эти четыре обновления нужно было бы включить в состав одной транзакции, чтобы база данных гарантированно осталась непротиворечивой. А в решении с двумя таблицами достаточно обновить всего одну строку в таблице курсов.

Ключи

Стоит отметить, что в реальных базах данных количество таблиц зачастую гораздо больше двух. Легко прикинуть, какие еще таблицы могли бы понадобиться в базе данных колледжа. Например, могла бы быть таблица с данными о каждом студенте: идентификационный номер, номер телефона, домашний адрес и т. д. Могла бы быть таблица с данными о преподавателях: адрес электронной почты, местонахождение кабинета, присутственные часы. Каждая таблица проектируется так, чтобы ее столбцы по возможности больше нигде повторялись, – идея в том, что когда потребуется детальная информация о каком-то объекте, ее можно будет найти в соответствующей таблице.

В терминологии баз данных любой столбец, по которому можно «поискать» детальную информацию, называется *ключом*. Подумайте,

к примеру, как вы стали бы искать номер аудитории, в которой читается курс по истории, посещаемый Луиджи. Если таблица всего одна, то нужно было бы просмотреть все строки, пока не найдется курс Луиджи, а затем найти в этой строке номер аудитории – 851. Но при наличии нескольких таблиц мы сначала нашли бы в первой номер курса по истории, который посещает Луиджи – «HIST256». А затем, используя «HIST256» как *ключ* другой таблицы, стали бы искать подробные сведения об этом курсе в строке с номером курса «HIST256». Найдя эту строку, мы извлекли бы из нее номер аудитории (снова 851). Этот процесс изображен на рисунке на следующей странице.



Поиск данных по ключу: чтобы найти номер аудитории, в которой читается курс по истории, посещаемый Луиджи, мы сначала находим номер этого курса в левой таблице. Найденное значение «HISR256» далее используется как ключ для другой таблицы. Поскольку номера курсов во второй таблице отсортированы по алфавиту, нужную строку можно найти очень быстро, а уже из нее извлекается номер аудитории (851).

Красота подобного использования ключей заключается в том, что поиск в базе по ключу можно производить очень эффективно. Делается это так же, как при поиске слова в словаре. Как вы стали бы искать слово «эпистемология» в обычном печатном словаре? Естественно, вы не стали бы листать страницы с первой до последней, а быстро нашли бы нужную страницу, обращая внимания только на заголов-

ки сверху. Сначала вы перелистывали бы сразу по много страниц, а, приближаясь к цели, постепенно уменьшали бы их количество. При поиске по ключу в базе данных применяется та же техника, только гораздо эффективнее, чем может сделать человек. Объясняется это тем, что база данных может заранее вычислить, по сколько страниц перелистывать, и сохранить сведения о заголовках в начале и конце таких групп страниц. Множество заранее вычисленных групп страниц для быстрого поиска по ключу в информатике называется *B-деревом*. Это еще одна остроумная идея, играющая огромную роль в современных базах данных. К сожалению, подробный рассказ о *B-деревьях* завел бы нас слишком далеко в сторону.

Трюк с виртуальной таблицей

Мы почти готовы оценить главный трюк, лежащий в основе современных баз данных с несколькими таблицами. Принципиальная идея проста: хотя вся информация хранится в фиксированном наборе таблиц, база данных может сгенерировать новые временные таблицы, когда это потребуется. Такие таблицы мы будем называть «виртуальными», чтобы подчеркнуть тот факт, что на самом деле они нигде не хранятся, – база данных создает их автоматически, когда нужно ответить на запрос, а потом сразу же удаляет.

Продемонстрируем трюк с виртуальными таблицами на простом примере. Допустим, что имеется база данных, изображенная на нижнем рисунке на стр. 159, и пользователь просит найти имена всех студентов, посещающих курс профессора Кирби. База может обработать этот запрос несколькими способами, мы рассмотрим только один из возможных подходов. Первый шаг – создать новую виртуальную таблицу, в которой перечислены студенты и преподаватели всех курсов. Делается это с помощью специальной операции – *соединения* двух таблиц. Основная идея заключается в том, чтобы объединить каждую строку одной таблицы с соответствующей ей строкой другой таблицы, причем соответствующими считаются строки, в которых одинаковы значения в столбцах, содержащих ключи. Например, результатом соединения двух таблиц, показанных на нижнем рисунке на стр. 159, по ключевому столбцу «номер курса» будет виртуальная таблица, в точности совпадающая с той, что изображена на верхнем рисунке, – имя каждого студента объединяется со всеми данными о соответствующем курсе из второй таблицы, а для поиска этих данных используется столбец «номер курса», выступающий в роли ключа. Но из-

начально мы просили только имена студентов и преподавателей, все прочие столбцы нам не нужны. По счастью, в базе данных есть также операция *проецирования*, которая позволяет отбросить ненужные столбцы. Таким образом, после операции соединения, объединяющей две таблицы, и операции проецирования, которая устраняет лишние столбцы, получается такая виртуальная таблица:

Имя студента	Преподаватель
Франческа	Проф. Блэк
Франческа	Проф. Смит
Сюзанна	Проф. Кирби
Эрик	Проф. Кирби
Луиджи	Проф. Смит
Луиджи	Проф. Кирби
Билл	Проф. Блэк
Билл	Проф. Смит
Роза	Проф. Кирби
Роза	Проф. Блэк

Следующая важная операция над базой данных называется *выборкой*. Она отбирает строки из таблицы, исходя из некоторого критерия, а остальные отбрасывает, в результате чего получается новая виртуальная таблица. В данном случае нас интересуют студенты, посещающие курс профессора Кирби, поэтому нужно выполнить операцию выборку, которая отбирает лишь те строки, в которых в качестве преподавателя указан «Проф. Кирби». Остается такая виртуальная таблица:

Имя студента	Преподаватель
Сюзанна	Проф. Кирби
Эрик	Проф. Кирби
Луиджи	Проф. Кирби
Роза	Проф. Кирби

Почти все готово. Осталось только сделать еще одну операцию проекции, чтобы убрать столбец «преподаватель». Получится виртуальная таблица, дающая ответ на первоначальный запрос:

Имя студента
Сюзанна
Эрик
Луиджи
Роза

Стоит добавить одно техническое замечание. Тем, кто знаком с языком запросов к базе данных SQL, наверное, показалось странным данное выше определение операции «выборки», потому что в SQL команда select делает куда больше, чем просто отбор строк. Использованная мной терминология берет начало в математической теории операций в базах данных, которая называется *реляционной алгеброй*. Там «выборка» означает именно отбор строк и ничего больше. Реляционная алгебра включает также операции соединения и проецирования, которыми мы пользовались для поиска студентов профессора Кирби.

Реляционные базы данных

База данных, в которой вся информация хранится в наборе взаимосвязанных таблиц типа тех, что были показаны выше, называется *реляционной*. Реляционные базы данных пропагандировал исследователь из IBM Э. Ф. Кодд в работе 1970 года «A Relational Model of Data for Large Shared Data Banks»[2], оказавшей необычайно сильное влияние. Как и многие величайшие научные идеи, в ретроспективе реляционные базы данных кажутся довольно простыми – но в то время это был гигантский скачок в сторону повышения эффективности хранения и обработки информации. Оказалось, что горстки операций (таких, как «выборка», «соединение» и «проецирование») достаточно для порождения виртуальных таблиц, которые позволяют ответить практически на любой запрос к реляционной базе. Таким образом, в таблицах реляционной базы можно эффективно хранить данные в структурированном виде, а трюк с виртуальными таблицами дает возможность отвечать на запросы, которые, на первый взгляд, нуждаются в данных, представленных в иной форме.

Вот почему реляционные базы данных применяются в большинстве компаний, занимающихся электронной торговлей. Всякий раз, покупая что-то через Интернет, вы, вероятно, взаимодействуете с

[2] Реляционная модель данных для больших разделяемых банков данных. *Прим. перев.*

целым выводком таблиц в реляционной базе, в которых хранится информация об изделиях, клиентах и совершенных ими покупках. В киберпространстве мы буквально окружены реляционными базами, хотя часто и не подозреваем об этом.

Базы данных с точки зрения человека

Случайному наблюдателю базы данных могут показаться самой скучной темой в этой книге. Да и что волнительного можно найти в хранении данных? Однако, если вдуматься, то гениальные идеи, заставляющие крутиться базы данных, предстанут совсем в другом свете. Работающие на оборудовании, которое может отказать в середине любой операции, базы данных тем не менее обеспечивают эффективность и стопроцентную надежность, которых мы привыкли ожидать от онлайнового банкинга и других подобных систем. Трюк со списком дел дал нам в руки атомарные транзакции, которые гарантируют непротиворечивость, даже если с базой одновременно работают тысячи пользователей. Столь высокая степень параллелизма вкупе с быстротой реакции на запросы, за которую нужно благодарить трюк с виртуальной таблицей, – основа эффективности баз данных. Трюк со списком дел гарантирует также непротиворечивость в условиях возможных отказов. В сочетании с трюком «подготовить и зафиксировать», который применяется к реплицированным базам, мы получаем данные, свободные от всяких противоречий и способные пережить любые отказы оборудования.

Блистательная победа баз данных над ненадежными компонентами, которую в информатике называют «отказоустойчивостью», – плод работы многих исследователей на протяжении нескольких десятилетий. Среди тех, чей вклад особенно велик, следует назвать Джима Грея, великолепного специалиста, который написал книгу по обработке транзакций (первое издание этой книги –«Transaction Processing: Concepts and Techniques» – вышло в 1992 году). Увы, карьера Грея закончилась слишком рано: однажды в 2007 году, выйдя на своей яхте из бухты Сан-Франциско, он прошел под мостом Золотые Ворота в открытый океан, намереваясь провести денек на близлежащих островах. И с тех пор ни Грея, ни его яхты никто не видел. У этой трагической истории есть хватающая за душу черточка – многочисленные друзья Грея по сообществу баз данных в попытках спасти его

пытались использовать им же созданные инструменты: они загрузили в базу данных только что сделанные спутниковые снимки океана в окрестностях Сан-Франциско, чтобы знакомые и коллеги могли отыскать хоть какие-нибудь следы пропавшего пионера баз данных. К сожалению, поиск не увенчался успехом, и информатика осталась без одного из своих выдающихся светил.

ГЛАВА 9. Цифровые подписи: кто на самом деле написал эту программу?

А чтобы убедить вас, что вы не правы, и доказать вам ошибочность вашего предположения, я сейчас покажу вам свидетельство ... Посмотрите сами, посмотрите, возьмите в руки, можете убедиться, оно не подделано.

— Чарльз Диккенс «*Повесть о двух городах*»[1]

Из всех идей, с которыми мы встретились в этой книге, концепция «цифровой подписи», пожалуй, самая парадоксальная. Если понимать буквально, то слово «цифровой» означает «состоящий из ряда цифр». Но, по самому своему определению, такой объект можно скопировать: достаточно просто копировать одну цифру за другой. Если можно прочитать, то можно и скопировать! С другой стороны, весь смысл «подписи» состоит в том, что прочитать ее можно, а скопировать (то есть подделать) не может никто, кроме автора. И как же возможно создать подпись, которая является цифровой, но не может быть скопирована? В этой главе мы покажем, как разрешается этот парадокс.

Для чего в действительности применяются цифровые подписи?

На первый взгляд, задавать вопрос, для чего применяются цифровые подписи, совершенно излишне. Ну конечно же, для тех же целей, что и подписи на бумаге: чтобы подписывать чеки и другие юридические

[1] Перевод Е. Бекетовой.

документы, например, договор аренды квартиры. Разве нет? Но немного подумав, вы поймете, что это не так. Когда вы расплачиваетесь за что-то через Интернет, кредитной картой или в системе онлайнового банкинга, разве вы ставите где-то свою подпись? Нет, не ставите. При оплате кредитной картой обычно никакой подписи не нужно. Системы онлайнового банкинга устроены немного иначе, потому что при входе в такую систему вы должны ввести пароль, подтвердив тем самым свою личность. Но когда впоследствии вы совершаете в системе платеж, подписи с вас опять-таки никто не требует.

Компьютер проверяет цифровую подпись автоматически. Вверху: это сообщение мой браузер показывает, когда я пытаюсь скачать и выполнить правильно подписанную программу. Внизу: а такой результат я вижу, когда цифровая подпись отсутствует или недействительна.

Как видите, есть две возможности. Если программа снабжена действительной подписью (как на верхнем рисунке), то компьютер может подтвердить, что все нормально – кто написал, тот и подписал. Конечно, это не гарантирует безопасность программы, но, по крайней мере, мы можете принять обоснованное решение, доверять автору

или нет. С другой стороны, если подписи нет или она недействительна (как на нижнем рисунке), то нет абсолютно никакой уверенности в источнике программы. Даже если вы думаете, что загружаете программу с сайта уважаемой компании, есть шанс, что злобный хакер каким-то образом подменил настоящую программу вредоносным чудовищем. Или программа написана любителем, у которого нет времени или достаточной мотивации для создания действительной цифровой подписи. Вам, пользователю, решать, доверять такой программе или нет.

Подписание программ – самое очевидное применение цифровых подписей, но ни в коем случае не единственное. На самом деле, ваш компьютер получает и проверяет цифровые подписи на удивление часто, потому что в некоторых протоколах Интернета такие подписи используются для подтверждения подлинности участвующих во взаимодействии компьютеров. Например, защищенные серверы, адреса которых начинаются с «https», обычно посылают вашему компьютеру подписанный цифровой подписью сертификат, перед тем как организовать защищенный сеанс. Цифровые подписи применяются также для проверки аутентичности многих программных компонентов, например, надстроек над браузером. Возможно, вам попадались предупреждения о таких вещах во время блуждания по сети.

Существует еще один тип онлайновой подписи, с которым вы могли сталкиваться: некоторые сайты просят ввести в качестве подписи свое имя в онлайновой форме. Иногда мне приходится это делать при составлении рекомендательного письма для своих студентов. Но это *совсем не то*, что в информатике называется цифровой подписью! Понятно, что такую напечатанную подпись легко подделает любой, кто знает, как вас зовут. В этой главе вы узнаете, как создать цифровую подпись, которую никто не сможет подделать.

Рукописные подписи

Мы будем объяснять, как устроены цифровые подписи, постепенно: начнем с хорошо знакомых подписей на бумаге и будем мелкими шажками двигаться в сторону цифровых подписей. Итак, возвратимся во времени назад – в мир, где еще не было компьютеров. В этом мире удостоверить подлинность документа можно было только одним способом – поставив рукописную подпись на бумаге. Отметим, что при таком положении дел одного лишь подписанного документа недостаточно для удостоверения его подлинности. Допустим, к примеру, что

вы нашли клочок бумаги, на котором написано: «Обещаю уплатить 100 долларов Франсуазе. Подписано, Рави» – вот буквально так.

Бумажный документ с рукописной подписью.

Как проверить, что именно Рави подписал этот документ? Чтобы это сделать, необходимо хранилище подписей, которому вы доверяете и куда можно сходить и проверить, что подпись Рави подлинная. В реальном мире эту роль берут на себя банки или государственные учреждения – там действительно хранятся образцы подписей клиентов, и при необходимости с этими физическими документами можно свериться. А в нашей гипотетической ситуации предположим, что имеется доверенное утверждение, которое назовем «банк рукописных подписей», где хранятся подписи всех людей. Схематический пример такого банка показан на следующем рисунке.

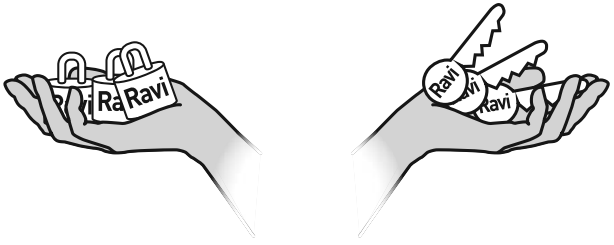
В банке хранятся удостоверения клиентов вместе с их рукописными подписями.

Чтобы проверить подлинность подписи Рави на документе с обещанием заплатить Франсуазе, необходимо зайти в банк рукописных подписей и попросить показать подпись Рави. Здесь мы делаем два важных допущения. Во-первых, предполагается, что банку можно доверять.

Теоретически можно представить себе, что служащие банка подменят подпись Рави подписью самозванца, но мы эту возможность отметаем. Во-вторых, предполагается, что самозванец не может подделать подпись Рави. Это допущение, как известно, не имеет ничего общего с действительностью: опытный фальсификатор без труда воспроизведет подпись, и даже любитель способен добиться приемлемого сходства. Тем не менее, допущение о невозможности подделки необходимо – без него рукописные подписи вообще были бы бесполезны. Ниже мы увидим, что цифровые подписи подделать практически невозможно. И в этом одно из основных их преимуществ над рукописными.

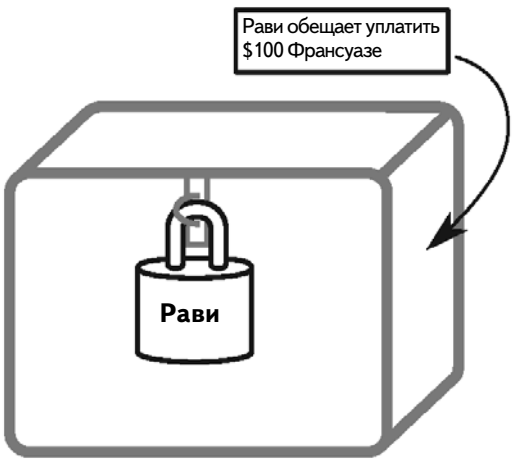
Подписание с помощью замка

Первый наш шаг в направлении цифровых подписей – отказаться от рукописных подписей вовсе и внедрить другой способ проверки подлинности документов, опирающийся на замки, ключи и запертые ящики. Каждый участник новой схемы (в нашем примере это будут Рави, Такеши и Франсуаза) приобретает большой запас навесных замков. Важно, что все замки, принадлежащие одному участнику, одинаковы. Кроме того, замки каждого участника *уникальны*: никто не может изготовить или раздобыть такой же замок, как у Рави. И наконец, все замки обладают довольно необычным свойством: они оснащены биометрическими датчиками, гарантирующими, что запереть замок может только его владелец. Так, если Франсуаза найдет открытый замок, принадлежащий Рави, то использовать его, чтобы запереть что-то, она не сможет. Само собой, у Рави имеется также запас ключей, отпирающих его замки. Поскольку все его замки идентичны, то ключи также идентичны. Эта ситуация схематически изображена на следующем рисунке. Мы описали сцену, на которой будет разворачиваться наш «трюк с физическим замком».



В трюке с физическим замком у каждого участника имеется запас изготовленных только для него идентичных замков и ключей.

Теперь предположим, как и прежде, что Рави должен Франсуазе 100 долларов, и Франсуаза хотела бы зафиксировать этот факт способом, допускающим проверку. Иначе говоря, Франсуаза хочет получить эквивалент документа, показанного на предыдущей странице, но не хочет полагаться на рукописную подпись. И вот как этот трюк работает. Рави составляет документ, на котором написано «Рави обещает уплатить 100 долларов Франсуазе», но не подписывает его. Он снимает с этого документа копию и кладет ее в сейф (сейфом мы будем называть прочный ящик, запираемый на навесной замок). Затем Рави запирает сейф одним из своих замков и вручает запертый сейф Франсуазе. Конечный результат показан на следующем рисунке. В некотором смысле – в каком именно, очень скоро станет ясно – запертый сейф и *есть* подпись документа. Отметим, что Франсуазе или уполномоченному ей свидетелю было бы разумно наблюдать за созданием подписи. В противном случае Рави мог бы смошенничать, положив в сейф совсем другой документ. (Пожалуй, эта схема была бы еще лучше, если бы сейф был прозрачным. Однако с идеей прозрачного сейфа смириться трудно, поэтому забудем про эту возможность.)



Чтобы создать допускающую проверку подпись с помощью трюка с физическим замком, Рави кладет копию документа в сейф и запирает его на свой замок.

Наверное, вы уже понимаете, как Франсуаза может проверить подлинность документа Рави. Если кто-то, в том числе сам Рави, попытается отрицать подлинность документа, то Франсуаза может сказать: «Так, Рави, дай-ка мне на минутку твой ключик. Я собираюсь открыть

им этот сейф». В присутствии Рави и других свидетелей (скажем, судьи) Франсуаза открывает замок и демонстрирует содержимое сейфа. После чего говорит: «Рави, ты единственный, у кого имеются замки, к которым подходит этот ключ, больше никто к содержимому этого сейфа не имеет отношения. Так что ты и только ты мог написать эту записку и положить ее в сейф. Отдавай мои 100 долларов!».

Хотя при первом знакомстве этот метод проверки подлинности выглядит несколько замысловато, на деле он и практичный, и действенный. Однако у него есть и недостатки. Главная проблема в том, что требуется добрая воля со стороны Рави: прежде чем что-то доказать, Франсуаза должна убедить Рави одолжить ей ключ. Но Рави может и отказаться или, хуже того, сделать вид, что готов сотрудничать, но подсунуть другой ключ – который не подходит к замку. А когда Франсуаза не сумеет открыть сейф, Рави заявит: «Вот видите, это не мой замок, должно быть, какой-то мошенник написал документ и положил его в сейф без моего ведома».

Чтобы остановить хитроумного Рави, нам снова придется прибегнуть к услугам доверенной третьей стороны, например банка. Но в отличие от банка рукописных подписей, наш новый банк будет хранить ключи. А участники отдают банку на хранение не образцы своих подписей, а физические ключи, которыми можно открыть их замки. Банк физических ключей показан на рисунке ниже.

В банке физических ключей хранятся ключи, отпирающие замки каждого участника. Все ключи различны.

Это последний кусочек головоломки, теперь трюк с физическим замком доведен до логического завершения. Если Франсуазе понадобится доказать, что расписку написал действительно Рави, то она

принесет сейф в банк и в присутствии свидетелей откроет его ключом Рави. Если ключ подошел к замку, значит, только Рави отвечает за содержимое сейфа, и, следовательно, в нем лежит подлинный экземпляр проверяемого Франсуазой документа.

Подписание с помощью перемножающего замка

Построенная нами инфраструктура ключей и замков – это именно тот подход, который необходим для цифровых подписей. Очевидно, однако, что для подписей, которые должны передаваться в электронном виде, физические замки и ключи не годятся. Поэтому следующий наш шаг – заменить замки и ключи их математическими аналогами, которые можно представить в цифровом виде. Точнее, мы представим замки и ключи *числами*, а запирание и отпирание – *операцией умножения в арифметике часов*. Если вы не знаете, что такое арифметика часов, самое время перечитать объяснение в главе 4.

Чтобы создать неподделываемые цифровые подписи, компьютеры используют часы с поистине гигантскими циферблатами – количество делений выражается числом с десятками, а то и сотнями цифр. Но для описания идеи мы возьмем небольшой циферблат, чтобы упростить вычисления.

Точнее, во всех примерах из этого раздела мы будем пользоваться часами с 11 делениями. Поскольку нам придется перемножать числа в такой арифметике, на следующей странице я привел таблицу умножения для всех чисел, меньших 11. Например, вычислим произведение 7×5. Не имея таблицы, мы должны были бы сначала перемножить числа в обычной арифметике: $7 \times 5 = 35$, а затем взять остаток от деления на 11. Поскольку $35 = 3 \times 11 + 2$, то остаток равен 2. Заглянув в таблицу, мы увидим, что на пересечении строки 7 и столбца 5 действительно стоит 2 (можете вместо этого взять число на пересечении строки 5 и столбца 7 – порядок не имеет значения). Выполните еще несколько упражнений на умножение и убедитесь, что все поняли.

Прежде чем двигаться дальше, немного изменим постановку решаемой задачи. Раньше нас интересовало, как Рави может «подписать» сообщение (расписку, если быть точными) Франсуазе. Сообщение было обычным текстом на естественном языке. Но, начиная с этого момента, нам будет гораздо удобнее работать с числами. Поэтому согласимся, что компьютер сможет без труда преобразовать сообщение

в последовательность чисел, которую Рави должен будет подписать. Если впоследствии кто-то захочет проверить подлинность цифровой подписи Рави на этой последовательности, то потом сможет легко восстановить из чисел исходный текст. Мы уже встречались с подобной задачей, когда обсуждали контрольные суммы (стр. 81) и трюк с более коротким символом (стр. 126). Если хотите разобраться в деталях, вернитесь к обсуждению трюка с более коротким символом – на стр. 127 приведен пример преобразования букв в числа и наоборот.

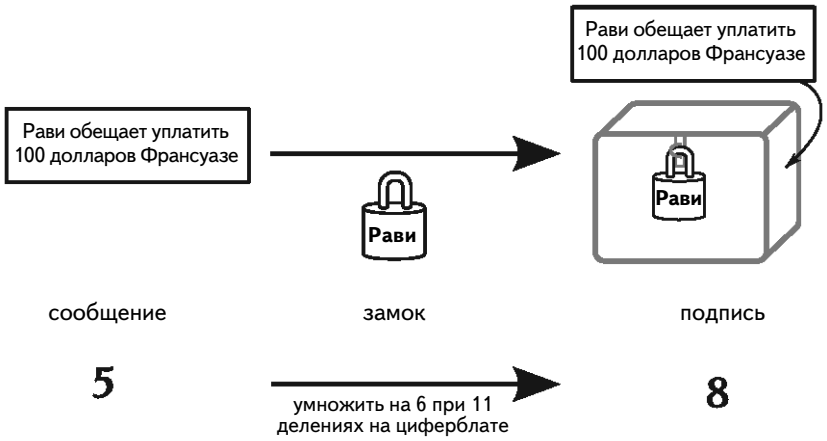
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	1	3	5	7	9
3	3	6	9	1	4	7	10	2	5	8
4	4	8	1	5	6	2	6	10	3	7
5	5	10	4	9	3	8	2	7	1	6
6	6	1	7	2	8	3	9	4	10	5
7	7	3	10	6	2	9	5	1	8	4
8	8	5	2	10	7	4	1	9	6	3
9	9	7	5	3	1	10	8	6	4	2
10	10	9	8	7	6	5	4	3	2	1

Таблица умножения для часов с 11 делениями на циферблате.

Итак, вместо текстового сообщения Рави должен подписать последовательность чисел, скажем «494138167543 ... 83271696129149». Но для простоты начнем с предельно короткого сообщения, состоящего всего из одной цифры, например, «8» или «5». Не расстраивайтесь, в конечном итоге мы научимся подписывать и сообщения нормальной длины. А пока не будем усложнять.

Покончив с предисловием, мы готовы к тому, чтобы разобраться с существом нового трюка, который назвали «трюком с перемножающим замком». Как и в трюке с физическим замком, Рави понадобится замок и отпирающий его ключ. Получить замок очень просто: Рави сначала выбирает количество делений на циферблате, а затем почти произвольное число, меньшее этой величины, которое и будет его числовым «замком» (на самом деле, некоторые числа более предпочтительны, но обсуждение таких деталей увело бы нас слишком далеко в сторону). Для определенности предположим, что Рави выбрал 11 делений и 6 в качестве своего замка.

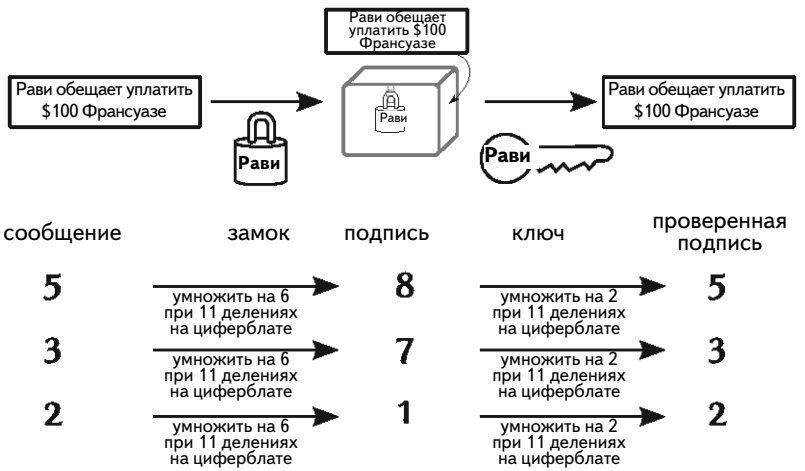
Ну и как Рави «запрет» свое сообщение в сейф с помощью такого замка? Как ни странно, для этого он воспользуется умножением: «запертым» сообщением будет произведение замка на само сообщение (в арифметике часов с 11 делениями, конечно). Напомню, что пока мы имеем дело с простым сообщением, состоящим из одной цифры. Предположим, что сообщение Рави – «5». Тогда «запертое» сообщение – $6 \times 5 = 8$ при 11 делениях, как обычно (убедитесь в этом с помощью приведенной выше таблицы умножения). Этот процесс изображен на рисунке. Конечный результат – «8» – это и есть цифровая подпись Рави под исходным сообщением. Разумеется, такое математическое «запирание» бессмысленно, если мы не сможем отпереть сообщение с помощью какого-нибудь математического «ключа». По счастью, такой способ есть. Трюк состоит в том, чтобы снова выполнить умножение (в арифметике часов), но на другое число – специально подобранное так, чтобы отпирать замок, основанный на ранее выбранном числе.



Как «запереть» числовое сообщение на «замок», создав тем самым цифровую подпись. Вверху показано, как сообщение физически запирается в сейф с помощью физического замка. Внизу продемонстрирован математический аналог этой операции, когда сообщение является числом (5), замок – другим числом (6), а процессу запирания соответствует умножение на количество делений на циферблате. Конечный результат (8) – это цифровая подпись сообщения.

Продолжим начатый пример. Если Рави выбрал часы с 11 делениями и 6 в качестве замка, то ключом будет число 2. Откуда мы это

взяли? К этому важному вопросу мы вернемся позже. А пока убедимся, что умножение на такой ключ действительно отпирает запертое сообщение. Мы уже видели на рисунке выше, что после запирания сообщения 5 на замок 6 получается запертое сообщение (цифровая подпись) 8. Чтобы отпереть запертое сообщение, мы берем число 8 и умножаем его на ключ 2, что в арифметике часов дает 5. Смотрите-ка, действительно, как по волшебству, получилось исходное сообщение 5! Весь процесс показан на рисунке выше, где представлены еще два примера: сообщение «3» после запирания превращается в «7», а после отпирания ключом возвращается к исходному значению «3». Аналогичную метаморфозу претерпевает сообщение «2».



Как «запереть» и потом «отпереть» сообщение с помощью числового замка и соответствующего ему числового ключа. Вверху показано, как сообщение запирается и отпирается физически. В следующих трех строках продемонстрировано числовое запирание и отпирание сообщений с помощью умножения. Отметим, что процесс запирания порождает цифровую подпись, а процесс отпирания – сообщение. Если после отпирания получилось сообщение, совпадающее с исходным, то цифровая подпись проверена и является подлинной.

На этом рисунке объясняется также, как проверить цифровую подпись. Нужно просто взять подпись и отпереть ее с помощью перемножающего ключа подписавшего. Если получившееся отпертое сообщение совпадает с исходным, то подпись подлинная. В противном случае она подделана. Процесс проверки более детально показан

в таблице ниже. Здесь мы по-прежнему считаем, что на циферблате 11 делений, но для доказательства того, что в выбранных ранее числовых ключе и замке нет ничего особенного, мы взяли другие значения. Конкретно, замок равен 9, а подходящий к нему ключ – 5. В первом примере сообщение равно «4», его подпись «3». После отпирания подписи получаем «4», т. е. исходное сообщение, а это означает, что подпись подлинная. Во втором примере аналогичный результат получается для сообщения «8» с подписью «6». В последнем примере показана, что происходит, когда подпись подделана. Здесь сообщение, как и раньше, равно «8», но его подпись – «7». После отпирания подписи получаем сообщение «2», которое отличается от исходного. Таким образом, подпись поддельная.

Сообщение	**Цифровая подпись** (для получения настоящей подписи нужно умножить сообщение на значение числового замка 9. Для подделывания возьмите любое другое число)	**Отпертая подпись** (для отпирания подписи умножьте ее на значение ключа 5)	**Сообщения совпадают?**	**Подделка?**
4	3	4	Да	Нет
8	6	8	Да	Нет
8	7	2	Нет!	Да!

Вспомните, что физический замок был оснащен биометрическими датчиками, которые не давали воспользоваться им никому, кроме владельца, – иначе фальсификатор мог бы воспользоваться замком Рави, чтобы запереть в сейфе любое нужное ему сообщение, подделав тем самым его подпись. То же самое справедливо и для числовых замков. Рави должен хранить свой замок – число – в *секрете*. Подписывая сообщение, он может предъявить любому желающему как само сообщение, так и его подпись, но не замок.

А как насчет количества делений на циферблате и числового ключа? Их тоже нужно хранить в секрете? Ответ – нет. Рави может сообщить то и другое во всеуслышание, даже опубликовать на сайте, это никак не компрометирует его подпись. Если Рави действительно опубликует количество делений и значение ключа, то любой желающий сможет узнать эти числа и проверить его подписи. На первый взгляд, способ очень удобный – но есть важные нюансы.

Например, устраняет ли этот подход необходимость в доверенном банке, который был нужен для хранения рукописных подписей и физических замков? Нет, не устраняет – доверенная третья сторона, например банк, все равно необходима. Без нее Рави смог бы распространять фальшивый ключ и тем самым отказаться от своих подписей. Хуже того, враги Рави могли бы создать новый числовой замок и соответствующий ему ключ, организовать сайт, на котором объявить, что этот ключ принадлежит Рави, а затем подписывать с помощью нового замка любые сообщения. Всякий поверивший, что новый ключ принадлежит Рави, поверит и в то, что вражеские сообщения подписаны Рави. Таким образом, роль банка не в том, чтобы хранить ключ и количество делений на циферблате в секрете. Банк выступает в качестве доверенного органа, распространяющего информацию о ключе и количестве делений, ассоциированных с Рави. Это показано на рисунке ниже.

Банк числовых ключей. Роль банка состоит не в том, чтобы хранить числовые ключи и количество делений в секрете. Банк является доверенным учреждением, у которого можно спросить, какой ключ и количество делений связаны с конкретным лицом, и получить честный ответ. Банк готов предоставить эту информацию любому желающему.

Подведем итог обсуждению: числовые замки *закрыты*, а числовые ключи и количество делений на циферблате *открыты*. Согласен, ключу как-то не пристало быть «открытым», в повседневной жизни мы стараемся всячески оберегать свои физические ключи. Но чтобы прояснить такое необычное использование ключей, вернемся к трюку с физическими замками. Там банк хранил копию ключа Рави и безропотно одалживал ее всем желающим проверить подпись Рави.

Поэтому физический ключ можно было в каком-то смысле считать «открытым». То же рассуждение справедливо и для перемножающих ключей.

Настало время рассмотреть важный практический вопрос: что, если нужно подписать сообщение, содержащее более одной цифры? На него есть несколько ответов. Для начала можно взять гораздо больший размер циферблата: если, например, для представления делений используются 100-значные числа, то с помощью уже описанного метода можно будет подписывать 100-значные сообщения 100-значными подписями. Если сообщение еще длиннее, то можно просто разбить его на 100-значные блоки и подписать каждый блок отдельно. Но ученые придумали способ лучше. Оказывается, что длинные сообщения можно – для подписания – свернуть в один-единственный блок (длиной, скажем, 100 знаков), применив преобразование, которое называется *криптографической хэш-функцией*. Мы уже встречались с такими функциями в главе 5, где они использовались в качестве контрольной суммы для проверки правильности длинного сообщения (например, программного пакета). Здесь идея очень похожа: свести длинное сообщение к гораздо более короткому блоку, перед тем как подписывать. Тогда даже очень длинные «сообщения» можно будет подписать эффективно. А коли так, то вопрос о длинных сообщениях будем считать закрытым, чтобы не усложнять изложение.

Другой важный вопрос: откуда берутся числовые замки и ключи? Выше отмечалось, что участники могут выбрать для своего замка практически любое значение. Увы, детали, скрывающиеся за словом «практически», требуют знания теории чисел в объеме университетского курса. Ну а для тех, кто немного знаком с этой теорией, намекну: если количество делений на циферблате – простое число, то в качестве замка подойдет любое число, меньшее этой величины. В противном случае ситуация усложняется. Простым называется число, у которого нет других делителей, кроме себя самого и единицы. Как легко убедиться, число 11, которое мы использовали в этой главе, простое.

Таким образом, выбрать замок легко – особенно, если на циферблате простое число делений. Но после выбора замка нам еще необходимо подобрать к нему ключ. И вот это оказывается интересной – и очень старой – математической задачей. Ее решение известно уже много столетий, а его идея еще старше: это алгоритм Евклида, который был описан древнегреческим математиком Евклидом свыше 2000 лет назад. Но не будем засорять голову деталями генерации ключа.

Достаточно знать, что, зная значение замка, компьютер может найти соответствующий ключ с помощью хорошо известного в математике алгоритма Евклида.

Если это объяснение вас не удовлетворяет, то, быть может, я смогу вас немного успокоить, открыв ужасную тайну, поджидающую нас впереди: все решение на основе «перемножающих» замков и ключей имеет фундаментальный изъян и должно быть отброшено. В следующем разделе мы применим другой подход к числовым замкам и ключам, который реально используется на практике. Так зачем тогда я тратил время на объяснение никуда не годной системы с перемножением? Главным образом затем, что с умножением знакомы все, поэтому принцип можно объяснить, не вводя сразу слишком много новых идей. Другая причина – восхитительные связи между дефектным подходом на основе умножения и правильным, который мы рассмотрим ниже.

Но прежде чем двигаться дальше, необходимо понять, в чем состоит изъян подхода с перемножением. Напомним, что значение замка *закрыто* (т. е. секретно), а значение ключа – *открыто*. Как уже было сказано, участник схемы подписания вправе выбрать любой размер циферблата (который объявляется открыто) и значение замка (которое остается тайным), после чего должен сгенерировать соответствующий ключ с помощью компьютера (применяя алгоритм Евклида, если речь идет о перемножаемом ключе, как было до сих пор). Ключ хранится в доверенном банке, и банк готов сообщить его любому желающему. Проблема же в том, что тот же самый трюк – алгоритм Евклида – который применялся для генерации ключа по замку, можно с тем же успехом применить и в обратном направлении, а, стало быть, получить замок, к которому подходит ключ! Это немедленно сводит на нет всю схему цифровой подписи. Поскольку значение ключа открыто, то значение замка, которое мы считали секретным, может вычислить кто угодно. А зная замок, злоумышленник сможет подделать цифровую подпись.

Подписание степенным замком

В этом разделе мы заменим дефектную систему цифровой подписи на основе перемножения схемой, известной под названием RSA, которая реально применяется на практике. В новой системе вместо умножения используется менее известная операция *возведения в степень*. На самом деле, все это мы уже проходили во время объяснения крип-

тографии с открытым ключом в главе 4: тогда мы тоже сначала разобрали негодную систему на основе умножения, а потом рассмотрели настоящий вариант с возведением в степень.

Если вы незнакомы со степенной нотацией – 5^9 или 3^4, – то обратитесь к стр. 66. Но на всякий случай напомню, что 3^4 («3 в степени 4») – это не что иное, как $3 \times 3 \times 3 \times 3$. Кроме того, нам потребуется еще несколько технических терминов. В выражении 3^4 число 4 называется *показателем степени*, а 3 – *основанием*. Сама же операция называется *возведением в степень*. Как и в главе 4, мы будем сочетать возведение в степень с арифметикой часов. Во всех примерах в этом разделе будет использоваться циферблат с числом делений 22. Из показателей степени нам понадобятся только 3 и 7, поэтому я поместил таблицу, в которой показаны значения n^3 и n^7 для всех значений n от 1 до 20 (при 22 делениях на циферблате).

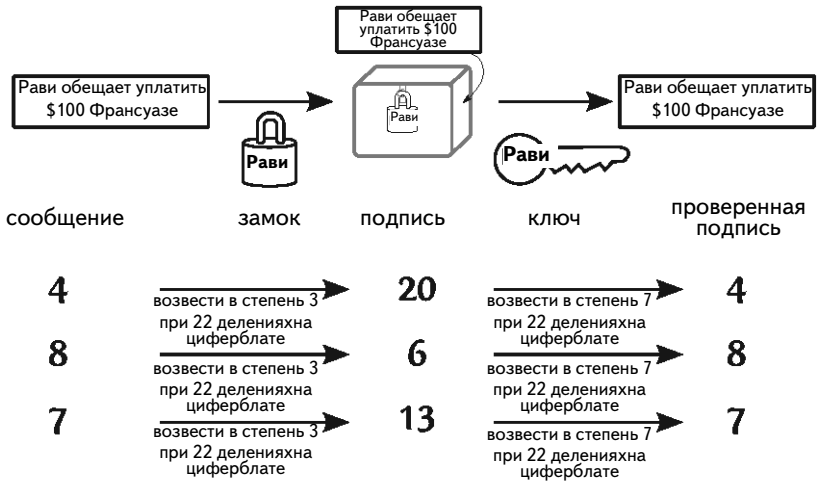
n	n^3	n^7	n	n^3	n^7
1	1	1	11	11	11
2	8	18	12	12	12
3	5	9	13	19	7
4	20	16	14	16	20
5	15	3	15	9	5
6	18	8	16	4	14
7	13	17	17	7	19
8	6	2	18	2	6
9	3	15	19	17	13
10	10	10	20	14	4

Результаты возведения в степень 3 и 7 при количестве делений 22.

Проверим два элемента в этой таблице, чтобы убедиться, что все правильно. Возьмем строку для $n = 4$. В обычной арифметике $4^3 = 4 \times 4 \times 4 = 64$. Теперь возьмем остаток от деления на 22: $64 = 3 \times 22 + 20$. Это объясняет, почему в столбце n^3 находится число 20. Аналогично в обычной арифметике $4^7 = 16\,384$ (уж поверьте мне на слово), и остаток от деления этого числа на 22 равен 16 (кому интересно, $22 \times 744 = 16\,368$). Потому-то в столбце n^7 стоит 16.

Вот теперь мы наконец готовы увидеть, как в действительности работает цифровая подпись. Все так же, как в методе на основе умножения, но с одним исключением: при запирании и отпирании сообще-

ний используется не умножение, а возведение в степень. Как и раньше, Рави сначала выбирает количество делений и открыто объявляет о своем выборе – в данном случае 22. Затем он выбирает секретное значение замка, которое может быть произвольным числом, меньшим количества делений (с одним примечанием мелким шрифтом, о котором я скажу ниже). Допустим, что Рави выбрал 3. Далее с помощью компьютера он находит ключ, соответствующий данному замку и размеру циферблата. Чуть позже мы рассмотрим некоторые детали этого процесса. Важно, впрочем, лишь то, что компьютер легко может вычислить ключ, зная замок и количество делений и применяя хорошо известные математические методы. В данном случае оказывается, что к замку 3 подходит ключ 7.



Запирание и отпираний сообщений с помощью возведения в степень.

На рисунке выше показаны конкретные примеры того, как Рави подписывает сообщения, а другие люди проверяют его подпись. Если сообщение равно «4», то его подпись равна «20»: чтобы вычислить ее, мы возводим сообщение в степень, равную замку. То есть мы должны вычислить 4^3, с учетом количества делений на циферблате получается 20. (Не забудьте, эти вычисления легко проделать с помощью приведенной выше таблицы.) Если теперь Франсуаза захочет проверить цифровую подпись Рави «20», то она сначала попросит банк сообщить ей, выбранные Рави количество делений и ключ (банк делает то же, что и раньше, только числа будут другими). Затем Франсуаза берет подпись, возводит ее в степень, равную значению ключа, и применяет

арифметику часов; получается $20^7 = 4$ (снова пользуемся таблицей). Если результат совпадает с исходным сообщением (так оно и есть), то подпись подлинная. На рисунке показаны также аналогичные вычисления для сообщений «8» и «7».

В таблице ниже тот же процесс показан еще раз, но на этот раз с упором на проверку подписи. Первые два примера совпадают с тем, что мы видели на предыдущем рисунке (для сообщений «4» и «8» соответственно), в обоих случаях подписи подлинные. В третьем примере сообщение равно «8», а подпись «9». Если отпереть ее известным ключом при известном количестве делений, то получится $9^7 = 15$, что отличается от исходного сообщения. Таким образом, эта подпись поддельная.

Сообщение	**Цифровая подпись** (для получения настоящей подписи нужно возвести сообщение в степень, равную значению числового замка 3. Для подделывания возьмите любое другое число)	**Отпертая подпись** (для отпирания подписи возведите ее в степень, равную значению ключа 7)	**Сообщения совпадают?**	**Подделка?**
4	20	4	Да	Нет
8	6	8	Да	Нет
8	9	15	Нет!	Да!

Как обнаружить, что подпись подделана, с помощью возведения в степень. В этих примерах значение замка равно 3, значение ключа – 7, количество делений на цифрблате – 22. Первые две подписи настоящие, третья поддельная.

Как уже было сказано ранее, схема со степенными замками и ключами называется цифровой подписью *RSA* по именам изобретателей (Рональд Райвест, Ади Шамир и Леонард Адлеман), которые опубликовали ее в 1970-х годах. Мы уже встречались с акронимом RSA в главе 4, посвященной криптографии с открытым ключом. На самом деле, RSA одновременно является схемой криптографии с открытым ключом и схемой цифровой подписи, и это не случайное совпадение, потому что между обоими алгоритмами существует глубокая теоретическая связь. В этой главе мы обсуждали только ту сторону RSA, которая относится к цифровой подписи, но, вероятно, вы отметили разительное сходство с идеями из главы 4.

Детали выбора количества делений, замка и ключа в системе RSA обворожительны, но для понимания общего принципа несущественны. Самое главное, что участник легко может вычислить ключ по замку. А вот обратить этот процесс невозможно: зная ключ и количество делений, нельзя подобрать соответствующий замок. Тем самым дефект рассмотренной выше системы на основе умножения можно считать исправленным.

По крайней мере, ученые так думают, хотя точно этого никто не знает. Вопрос о том, является ли система RSA действительно безопасной, один из самых увлекательных – и дразнящих – во всей информатике. Скажем лишь, что ответ на него связан как со старой нерешенной математической задачей, так и с горячей темой современных исследований на стыке физики и информатики. Задача называется *разложение на множители* (или *целочисленная факторизация*), а тема исследований – *квантовые вычисления*. Мы рассмотрим оба эти аспекта безопасности RSA по очереди, но сначала нужно понять, что вообще такое «безопасная» схема цифровой подписи, в частности, RSA.

Безопасность RSA

Безопасность любой схемы цифровой подписи сводится к вопросу «Может ли враг подделать мою подпись?». В случае RSA этот вопрос звучит более конкретно: «Может ли враг вычислить значение моего закрытого замка, зная значения открытого ключа и количество делений на циферблате?». Вы, наверное, будете расстроены, узнав, что ответ на этот вопрос положителен. Собственно, вы и так это знали: *всегда* можно подобрать значение замка методом проб и ошибок. В конце концов, у нас есть сообщение, количество делений и цифровая подпись. Мы знаем, что значение замка меньше, чем количество делений, поэтому можем просто пробовать каждое возможное значение замка, пока не найдем то, что дает правильную подпись. Всего-то и нужно, что возводить сообщение в степень. Подвох в том, что на практике в схеме RSA применяются циферблаты с гигантским количеством делений – длиной несколько тысяч цифр. Поэтому даже на самом быстром из существующих суперкомпьютеров для перебора всех возможных замков понадобилось бы несколько триллионов лет. Поэтому нам неинтересно, может ли враг вообще вычислить замок. Хотелось бы знать, может ли он это сделать *настолько эффективно*, чтобы представлять реальную угрозу. Если лучшая стратегия врага – метод проб и ошибок (или, как говорят в информатике, *полный перебор*), то мы всегда можем выбрать количество делений достаточно большим,

чтобы сделать эту атаку практически бессмысленной. С другой стороны, если у врага есть стратегия, работающая гораздо быстрее полного перебора, то мы в беде.

В схеме с перемножающими замками и ключами мы видели, что подписывающий может выбрать значение замка, а затем вычислить ключ, пользуясь алгоритмом Евклида. Проблема в том, что противнику не нужно прибегать к полному перебору, чтобы обратить этот процесс; можно применить тот же алгоритм Евклида, чтобы вычислить замок по ключу, а это куда быстрее, чем полный перебор. Поэтому схема на основе умножения и считается небезопасной.

Связь между RSA и разложением на множители

Я обещал вскрыть связь между безопасностью RSA и старой-престарой математической задачей о разложении на множители. Чтобы понять, как они связаны, нужно приоткрыть завесу тайны над тем, как выбирается количество делений в схеме RSA.

Сначала напомню определение *простого числа*: это число, не имеющее других делителей, кроме единицы и себя самого. Например, 31 – простое число, потому что может быть представлено только в виде произведения двух таких множителей: 1×31. Но 33 – не простое число, потому что $33 = 3 \times 11$.

Теперь мы готовы разобраться с тем, как наш старый приятель Рави выбирает количество делений на циферблате. Первым делом Рави должен выбрать два очень больших простых числа. В типичном случае каждое число состоит из сотен цифр, но мы, как обычно, будем работать с числами поменьше. Допустим, что Рави выбрал числа 2 и 11. Их произведение и будет количеством делений: в нашем случае $2 \times 11 = 22$. Как мы знаем, количество делений объявляется во всеуслышание наряду с выбранным значением ключа. Но – и это ключевой момент – два простых сомножителя количества делений хранятся в секрете и известны только Рави. Математические методы, стоящие за схемой RSA, позволяют, зная эти два простых сомножителя, вычислить значение ключа по значению замка и наоборот.

Детали описаны на врезке ниже, но для нашей главной цели они несущественны. Нужно лишь знать, что враги Рави не смогут вычислить секретное значение его замка, зная только открытую информацию (количество делений и значение ключа). Но *если бы* враги знали и оба простых сомножителя, то вычислить замок не составило бы труда. Иными

словами, враги Рави смогут подделать его подпись, если сумеют *разложить на множители* число, равное количеству делений. (Конечно, не исключено, что есть и другие способы взломать RSA. Эффективная факторизация – лишь одно из возможных направлений атаки.)

Рави выбирает два простых числа (в нашем примере 2 и 11), перемножает их и получает количество делений (22). Назовем эту величину «первичным» количеством делений – причины станут понятны чуть ниже. Затем Рави вычитает единицу из каждого простого числа и перемножает получившиеся результаты. Это дает ему «вторичное» количество делений. В нашем примере оно равно $1 \times 10 = 10$.

Теперь отметим удивительную связь с дефектной системой перемножающих замков и ключей, описанной ранее: Рави выбирает замок и ключ в соответствии с системой на основе умножения, только вместо первичного количества делений берет вторичное. Допустим, что в качестве замка Рави выбрал 3. Если вторичное количество делений равно 10, то к этому замку подходит перемножающий ключ 7. Легко убедиться, что это действительно так: после запирания сообщения «8» получается $8 \times 3 = 24$, или «4» при количестве делений 10. Отпирание подписи «4» ключом 7 дает $4 \times 7 = 28$, т. е. «8» при том же количестве делений. Стало быть, мы получили исходное сообщение.

На этом работа Рави закончена: он берет выбранные таким образом перемножающие замок и ключ и использует их как *степенные* замок и ключ в системе RSA. Разумеется, это будут показатели степени в арифметике с первичным количеством делений 22.

Технические детали генерации количества делений, замка и ключа в системе RSA.

В нашем крохотном примере разложить количество делений на множители (и значит, взломать схему цифровой подписи) до смешного просто: всем известно, что $22 = 2 \times 11$. Но если количество делений равно числу с сотнями цифр, то нахождение его множителей – задача чрезвычайно трудная. И хотя эта задача – «целочисленная фактори-

зация» – изучается уже несколько сотен лет, до сих пор никто так и не нашел общего способа решить ее настолько эффективно, чтобы скомпрометировать схему RSA с типичным для нее количеством делений на циферблате.

История математики пестрит нерешенными задачами, которые прельщали математиков одними лишь эстетическими свойствами, побуждая к глубокому исследованию, несмотря на отсутствие практической ценности. Удивительно, но многие из таких «интригующих, но кажущихся бесполезными» задач впоследствии приобрели огромную практическую значимость – а иногда для осознания их полезности должно было пройти несколько веков.

Разложение на множители относится именно к таким задачам. Первые серьезные исследования в этой области относятся к 17 веку – в работах математиков Ферма и Мерсенна. Эйлер и Гаусс – два величайших имени в математических святцах – внесли в нее важный вклад в следующих двух столетиях, а затем их работу продолжили многие другие ученые. Но лишь с появлением криптографии с открытым ключом в 1970-х годах трудность задачи факторизации больших чисел легла в основу практического применения. Как вы теперь знаете, тот, кто найдет эффективный алгоритм разложения больших чисел на множители, сумеет подделать все существующие цифровые подписи!

Но чтобы немного успокоить вас, скажу, что после 1970-х годов были придуманы и другие схемы цифровой подписи. Все они опираются на трудность решения различных фундаментальных математических задач. Поэтому открытие эффективного алгоритма разложения на множители «поломает» только схемы типа RSA.

С другой стороны, всем этим системам присуща общая проблема, которая по-прежнему смущает ученых: их безопасность не *доказана*. В основе каждой системы лежит некоторая математическая задача, которая долгое время изучалась и признана трудной. Но теоретики так и не смогли доказать отсутствие эффективного решения. Поэтому хотя специалисты согласны в том, что схемы криптографии с открытым ключом или цифровой подписи вряд ли удастся взломать, в принципе такая возможность не исключена.

Связь между RSA и квантовыми компьютерами

Я выполнил обещание вскрыть связь между RSA и одной старой математической задачей, но осталось еще объяснить, какое отношение

RSA имеет к области квантовых вычислений, в которой сейчас ведутся активные исследования. Чтобы разобраться в этом, мы должны сначала признать фундаментальный факт: в квантовой механике движение объектов носит *вероятностный* характер – в отличие от детерминированных законов классической физики. Таким образом, если построить компьютер из деталей, подверженных квантово-механическим эффектам, то он будет вычислять значения с некоторой вероятностью, а не строго детерминированные последовательности нулей и единиц, как классический компьютер. По-другому эту мысль можно выразить, сказав, что квантовый компьютер одновременно хранит несколько разных значений: у каждого имеется некоторая вероятность, но до тех пор пока вы не запросили у компьютера окончательный ответ, все они существуют одновременно. Следовательно, квантовый компьютер может вычислять много разных возможных ответов одновременно. Поэтому для некоторых типов задач можно реализовать метод полного перебора, когда будут опробоваться сразу все возможные решения!

Да, это годится не для всех задач, но, как выясняется, задача разложения на множители относится как раз к тому классу, который решается на квантовых компьютерах гораздо эффективнее, чем на классических. Поэтому если бы удалось построить квантовый компьютер, способный обрабатывать числа с тысячами цифр, то можно было бы взломать цифровую подпись RSA: разложить на множители количество делений на циферблате часов, на основе полученных множителей вычислить вторичное количество делений, а затем получить закрытый замок по открытому ключу.

В 2011 году, когда я пишу эти слова, теория квантовых вычислений далеко опережает практику. Исследователям удалось построить реальные квантовые компьютеры, но пока что они смогли лишь разложить на множители число $15 = 3 \times 5$ – до 1000-значных чисел, применяемых в схеме RSA, отсюда еще очень далеко! А для создания по-настоящему больших квантовых компьютеров предстоит разрешить необычайно сложные технические проблемы. Так что никто не знает, когда они станут достаточно мощными, чтобы раз и навсегда похоронить схему RSA, да и станут ли таковыми вообще.

Цифровые подписи на практике

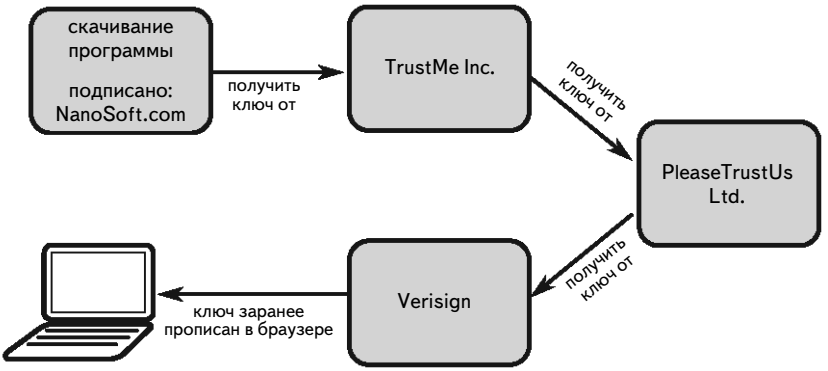
Выше в этой главе мы выяснили, что у конечных пользователей вроде вас и меня не возникает необходимость ставить свою цифровую под-

пись. Некоторые компьютерно грамотные пользователи таки подписывают свои электронные письма, но для большинства из нас основное применение цифровой подписи – проверка скачанного контента. Самый очевидный пример – скачивание какой-нибудь программы. Если она подписана, компьютер «отпирает» подпись, пользуясь открытым ключом подписавшего, и сравнивает результат с подписанным сообщением – в данном случае с самой программой (как уже было сказано, на практике программа перед подписанием сворачивается в гораздо меньшее сообщение, которой называется безопасным хэшем, или дайджестом). Если отпертая подпись соответствует программе, то выдается ободряющее сообщение, в противном случае сообщение будет зловещим; примеры того и другого показаны на стр. 168.

Я уже говорил, что во всех схемах необходим какой-то доверенный «банк», где хранятся значения открытых ключей и количества делений, выбранные владельцами цифровых подписей. К счастью, как вы, наверное, заметили, ходить в реальный банк при скачивании каждой программы не приходится. Доверенные организации, которые хранят открытые ключи, называются *удостоверяющими центрами*. В любом таком центре установлены серверы, к которым можно обратиться по сети и получить информацию об открытом ключе. Таким образом, получая цифровую подпись, ваша машина одновременно получает сведения о том, какой удостоверяющий центр может поручиться за открытый ключ подписавшего.

Вероятно, вы обратили внимание на таящуюся здесь проблему: конечно, ваш компьютер может обратиться к указанному удостоверяющему центру и проверить подпись, но почему мы должны доверять самому этому центру? Получается, что вместо проверки подлинности одной организации (от которой мы получили программу, например NanoSoft.com) мы должны проверить подлинность другой (удостоверяющего центра, например TrustMe Inc.) – но сама-то проблема никуда не делась. Хотите верьте, хотите нет, но обычно эта проблема решается за счет того, что удостоверяющий центр (TrustMe Inc.) ссылается на другой удостоверяющий центр (скажем, PleaseTrustUs Ltd.) – также с помощью цифровой подписи. Такая цепочка доверия может быть сколь угодно длинной, но проблема остается: как можно доверять организации в конце цепочки? Ответ, изображенный на рисунке ниже, заключается в том, что некоторые организации официально признаются так называемыми корневыми удостоверяющими центрами (сокращенно УЦ). Из наиболее известных корневых УЦ назову VeriSign, GlobalSign и GeoTrust. Контактная информация

(в том числе адрес в Интернете и открытый ключ) ряда корневых УЦ изначально прописана в браузере, поэтому цепочка доверия к цифровым подписям начинается в точке, которой можно доверять.



Для получения ключей и проверки цифровых подписей необходима цепочка доверия.

Парадокс разрешен

В начале этой главы я сказал, что само словосочетание «цифровая подпись» звучит как оксюморон: все записанное цифрами можно скопировать, а подпись по определению должна быть некопируемой. Как разрешился этот парадокс? Ответ состоит в том, что цифровая подпись зависит как от секрета, известного только подписавшему, так и от подписываемого сообщения. Секрет (который мы в этой главе называли замком) остается одним и тем же для всех сообщений, подписанных данным лицом, однако подпись для каждого сообщения разная. Таким образом, тот факт, что любой человек может беспрепятственно скопировать подпись, не имеет никакого значения: эту подпись нельзя перенести на другое сообщение, поэтому сам факт копирования не является подлогом.

Разрешение этого парадокса – не просто остроумная и красивая идея. Цифровые подписи играют огромную практическую роль: без них не было бы Интернета в том виде, в каком мы его знаем. Благодаря криптографии данными все же можно было бы безопасно обмениваться, но гораздо труднее было бы проверить *источник* полученных данных. Сочетание блестящей идеи со столь широким практическим применением означает, что цифровые подписи без сомнения являются одним из самых значительных достижений информатики.

ГЛАВА 10. Что можно вычислить?

Позвольте напомнить вам о некоторых проблемах вычислительных машин.

— Ричард Фейнман
(лауреат Нобелевской премии по физике 1965 года)

Мы с вами видели целый ряд изобретательных, мощных и красивых алгоритмов, превращающих голое железо компьютера в услужливого джинна. И в свете восторженного тона предыдущих глав было бы только естественно задаться вопросом, а есть ли что-то такое, чего компьютер сделать не может. Ответ абсолютно ясен, если говорить только о том, что компьютеры не умеют делать *сегодня*: таких задач масса и по большей части они связаны с тем или иным проявлением искусственного интеллекта. Нужны примеры? Пожалуйста: высококачественный перевод с одного языка на другой, например с английского на китайский, автоматическое управление автомобилем, так чтобы он двигался в оживленном городском потоке быстро и безопасно, или (мне как преподавателю это особенно интересно) оценка студенческих работ.

Да, мы видели, каких удивительных результатов можно достичь с помощью хитроумных алгоритмов. И, быть может, завтра кто-то изобретет алгоритм, который позволит вести машину или безукоризненно проверять работы моих студентов. По видимости, это трудные задачи, но неужели настолько трудные, что решить их невозможно? Да и вообще существуют ли задачи трудные до такой степени, что никто и никогда не сможет придумать алгоритм их решения? В этой главе мы увидим, что ответ – твердое да: *существуют* задачи, которые никогда не будут решены компьютерами. Этот глубокий факт – что одни вещи «вычислимы», а другие нет – составляет любопытный контрапункт к тем многочисленным триумфам алгоритмов, с которыми мы познакомились в этой книге. Сколь бы изощренные алгоритмы ни изобретали

будущие исследователи, всегда будут существовать задачи, ответ на которые невозможно «вычислить».

Факт существования невычислимых задач поражает сам по себе, но история его открытия еще более замечательна! То, что такие задачи существуют, было известно еще до создания первого электронного компьютера! Два математика, американский и британский, независимо установили существование невычислимых задач в конце 1930-х годов – на несколько лет раньше, чем появились настоящие компьютеры, а произошло это во время Второй мировой войны. Американцем был Алонзо Чёрч, чья революционная работа по теории вычислений по сей день остается основополагающей во многих отраслях информатики. А британцем – Алан Тьюринг, которого часто называют самой важной фигурой из всех, что стоят у истоков информатики. Работы Тьюринга охватывали весь спектр идей вычислений – от сложнейших математических теорий и глубоких философских мыслей до смелых инженерных решений. В этой главе мы пройдем по стопам Чёрча и Тьюринга в путешествии, которое приведет нас к демонстрации невозможности решения одной конкретной задачи с помощью компьютера. Мы начнем это путешествие в следующем же разделе с обсуждения программных ошибок и сбоев.

Ошибки, сбои и надежность программ

За последние годы надежность программного обеспечения компьютеров резко возросла, но все мы знаем – полагаться на безупречность работы программ все еще опасно. Время от времени даже хорошо написанные программы высокого качества делают нечто непредусмотренное. В худшем случае программа «падает», и вы теряете данные или документ, над которым работали (или вылетаете из видеоигры – очень досадно, знаю по собственному опыту). Но всякий, кто работал на домашних компьютерах в 1980–1990-е годы, засвидетельствует, что тогда программы падали куда чаще, чем в 21 веке. Улучшение дел в этой области объясняется разными причинами, но одна из основных – значительный прогресс автоматизированных средств проверки программ. Иными словами, после того как команда программистов написала большую и сложную программу, она может воспользоваться таким средством, чтобы проверить, есть ли в программе ошибки, которые могут привести к сбою. И эти автоматизированные инструменты ищут потенциальные ошибки все лучше и лучше.

Но тогда возникает естественный вопрос: а настанет ли такой день, когда автоматизированные средства проверки программ разовьются настолько, что смогут находить все ошибки в программе? Вот уж было бы действительно здорово, ведь сама возможность сбоев программы была бы ликвидирована раз и навсегда. Увы, в этой главе нас ждет печальное известие – программная нирвана никогда не наступит: доказуемо невозможно создать инструмент проверки программ, который будет находить все потенциальные причины сбоев во всех программах.

Уместно будет сказать еще несколько слов о том, что означают слова «доказуемо невозможно». В большинстве наук, например в физике и биологии, ученые выдвигают гипотезы о том, как могут вести себя те или иные системы, и ставят эксперименты, проверяющие правильность гипотез. Но поскольку эксперимент всегда несет в себе некую неопределенность, невозможно быть на 100 % уверенным, что гипотеза правильна, даже если эксперимент оказался успешным. Однако – и это составляет разительный контраст с естественными науками – в математике и информатике в правильности некоторых результатов можно быть уверенными на все сто процентов. Если вы принимаете основные аксиомы математики (например, что 1 + 1 = 2), то путем цепочки дедуктивных рассуждений можно с полной уверенностью доказать истинность ряда других утверждений (например, что любое число, оканчивающееся на 5, делится на 5). Для такого рода рассуждений не нужны компьютеры: математик доказывает неоспоримые факты с помощью одних лишь карандаша и бумаги.

Поэтому, когда мы в информатике говорим, что «*X* доказуемо невозможно», это не значит, что *X* очень трудно или, быть может, невозможно достичь на практике. Это означает, что со стопроцентной уверенностью *X* никогда не может осуществиться, потому что это было доказано с помощью цепочки дедуктивных математически строгих рассуждений. Простой пример: «доказуемо невозможно, что число, кратное 10, оканчивается на 3». И другой пример, который является окончательным выводом этой главы: доказуемо невозможно создать автоматизированный инструмент проверки программ, который будет обнаруживать все потенциальные сбои во всех программах.

Доказательство ложности чего-либо

При доказательстве невозможности универсальной программы обнаружения сбоев мы воспользуемся техникой *доказательства от про-*

тивного. Хотя математики любят заявлять исключительные права на эту технику, на самом деле люди постоянно пользуются ей в повседневной жизни, даже не осознавая. Приведу простой пример. Для начала мы должны признать следующие два факта, которые не будут оспаривать даже самые заядлые исторические ревизионисты.

1. Гражданская война в США происходила в 1860-е годы.
2. Авраам Линкольн был президентом во время Гражданской войны.

Рассмотрим теперь такое утверждение: «Авраам Линкольн родился в 1520 году». Оно истинно или ложно? Даже не зная об Аврааме Линкольне ничего, кроме двух названных выше фактов, вы все равно быстро установите, что это утверждение ложно, не так ли?

Скорее всего, вы будете рассуждать примерно так: (1) человек не живет дольше 150 лет, так что если Линкольн родился в 1520 году, то должен был умереть никак не позже 1670 года; (2) Линкольн был президентом во время Гражданской войны, поэтому Гражданская война должна была начаться до того, как он умер, т. е. раньше 1670 года; (3) но это невозможно, потому что, как все согласны, Гражданская война происходила в 1860-е годы; (4) *следовательно*, Линкольн не мог родиться в 1520 году.

Проанализируем это рассуждение более тщательно. Почему допустимо сделать вывод о ложности исходного утверждения? Потому что мы доказали, что оно противоречит некоторому заведомо истинному факту. Точнее, мы доказали, что из исходного утверждения следует, что Гражданская война началась раньше 1670 года, тогда как точно известно, что она происходила в 1860-е годы.

Доказательство от противного – чрезвычайно важная техника, поэтому возьмем пример, чуть более близкий к математике. Я утверждаю: «В среднем сердце человека совершает примерно 6000 ударов за 10 минут». Это утверждение истинно или ложно? Наверное, вы сразу заподозрили неладное, но как доказать, что я вру? Перед тем как читать дальше, потратьте несколько секунд на то, чтобы проанализировать ход своих мыслей.

Мы снова воспользуемся доказательством от противного. Во-первых, предположим, будто утверждение о том, что сердце человека совершает в среднем 6000 ударов за 10 минут, верно. Если это так, то сколько ударов оно совершает в среднем за одну минуту? Для этого нужно разделить 6000 на 10, получится 600 ударов в минуту. Даже далекий от медицины человек знает, что это намного выше нормальной частоты пульса, которая составляет от 50 до 150 ударов в минуту. По-

этому исходное утверждение противоречит общеизвестному факту и потому должно быть ложным: *неправда*, что сердце человека совершает в среднем 6000 ударов за 10 минут.

В более абстрактных терминах доказательство от противного можно описать следующим образом. Допустим, мы подозреваем, что некоторое утверждение *S* ложно, но хотели бы доказать это неопровержимо. Сначала мы предполагаем, что *S* истинно. Применяя какое-то рассуждение, мы устанавливаем, что некое другое утверждение *T* тоже должно быть истинно. Если, однако, известно, что *T* ложно, то мы пришли к противоречию. Это доказывает, что исходное утверждение (*S*) обязано быть ложным. Математик сказал бы более кратко: «из *S* следует *T*, но *T* ложно, следовательно, *S* тоже ложно». Вот в этом и состоит суть доказательства от противного. В таблице ниже показано, как связать это абстрактное описание доказательства от противного с двумя приведенными выше примерами.

	Первый пример	**Второй пример**
S (исходное утверждение)	Линкольн родился в 1520 году	Сердце человека совершает 6000 ударов за 10 минут
T (следует из *S*, но заведомо ложно)	Гражданская война началась раньше 1670 года	Сердце человека совершает 600 ударов в минуту
Вывод: *S* ложно	Линкольн ***не*** родился в 1520 году	Сердце человека ***не*** совершает 6000 ударов за 10 минут

На этом будем считать отвлечение на тему доказательства от противного законченным. Конечная цель этой главы – доказать, от противного, что не может существовать программы, которая обнаруживает все возможные сбои во всех программах. Но перед тем как устремиться к этой цели, необходимо познакомиться с некоторыми интересными фактами относительно компьютерных программ.

Программы, анализирующие другие программы

Компьютеры рабски следуют точным инструкциям, содержащимся в программах. Делают они это абсолютно детерминированно, поэтому при каждом запуске программы получается один и тот же результат. Это правда? Или ложь? На самом деле, для ответа на этот вопрос я не предоставил достаточно информации. Правда, что некоторые простые программы при каждом запуске порождают в точности одинако-

вый результат, но большинство программ, с которыми мы ежедневно работаем, при каждом запуске ведут себя совершенно по-разному. Возьмите, к примеру, свой любимый текстовый редактор: что, всякий раз, как вы его открываете, экран выглядит одинаково? Если я открою файл «address-list.docx» в редакторе Microsoft Word, то увижу на экране список адресов, хранящийся у меню в компьютере. А если в том же Microsoft Word я открою файл «bank-letter.docx», то увижу текст письма, которое вчера отправил в свой банк. (Если «.docx» – для вас тайна за семью печатями, прочитайте врезку ниже, там рассказывается о расширениях имен файлов.)

> В этой главе нам будут часто встречаться имена файлов, например «abcd.txt». Часть имени после точки называется «расширением» – в данном случае расширением имени «abcd.txt» является «txt». В большинстве операционных систем расширение имени файла определяет тип содержащихся в файле данных. Например, файл с расширением «.txt» обычно содержит простой текст, файл с расширением «.html» – веб-страницу, а файл с расширением «.docx» – документ Microsoft Word. Некоторые операционные системы по умолчанию скрывают расширения, поэтому вы можете не знать об их существовании, пока не выключите режим «скрыть расширения». О том, как это делается, легко узнать, задав поисковой системе запрос «показать расширения файлов».

Технические сведения о расширениях имен файлов.

Давайте проясним одну вещь: в обоих случаях я запускаю одну и ту же программу, Microsoft Word. Но с разными *входными данными*. Не думайте, что все современные операционные системы позволяют запускать программы двойным щелчком по документу. Это просто любезность со стороны компании-производителя (скорее всего, Apple или Microsoft). При двойном щелчке по документу запускается некоторая программа, для которой этот документ является входными данными. Результат программы – это то, что вы видите на экране, и, естественно, он зависит от того, по какому документу вы щелкали.

На практике вход и выход компьютерных программ несколько сложнее. Например, выбирая пункт меню или набирая текст, вы предоставляете программе дополнительные данные. А сохраняя документ или другой файл, порождаете дополнительный результат. Но для простоты будем считать, что программа принимает ровно один элемент входных данных, каковым является имя хранящегося в компьютере файла. И еще будем считать, что программа порождает ровно один результат – графическое окно на экране монитора.

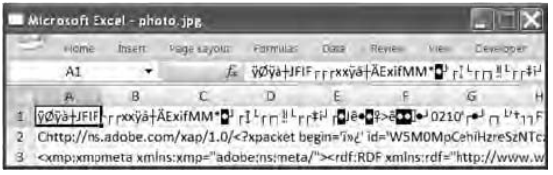
К сожалению, удобство, благодаря которому файл можно указать двойным щелчком по нему, затемняет важный момент. В операционной системе есть много уловок, с помощью которых она угадывает, какую программу запускать после двойного щелчка по файлу. Но важно понимать, что при запуске *любой* программы можно указать *любой* файл в качестве входных данных. Как это сделать? Ниже перечислено несколько способов. Не все они работают во всех операционных системах, не всегда позволяют выбрать любой файл – разные системы запускают программы по-разному, и иногда выбор входного файла ограничен из соображений безопасности. Тем не менее, я настоятельно рекомендую немного поэкспериментировать со своим компьютером и убедиться, что вы можете запустить свой любимый редактор с файлами разных типов.

Запустить программу Microsoft Word с входным файлом `stuff.txt` можно тремя способами.

- Щелкнуть правой кнопкой мыши по файлу `stuff.txt`, выбрать из меню команду «Открыть с помощью...» и затем выбрать приложение Microsoft Word.
- Сначала с помощью средства операционной системы поместить на рабочий стол ярлык приложения Microsoft Word. Затем перетащить файл `stuff.txt` на этот ярлык.
- Открыть приложение Microsoft Word непосредственно, выбрать из меню «Файл» команду «Открыть», установить в раскрывающемся списке режим «Все файлы», затем выбрать файл `stuff.txt`.

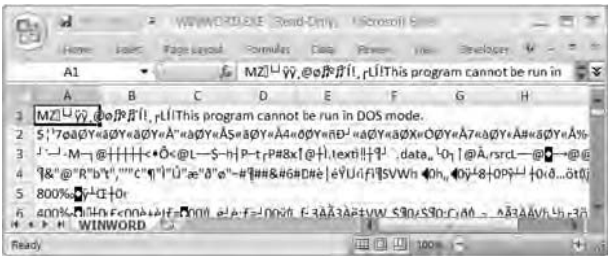
Различные способы запуска программы с указанием конкретного входного файла.

Понятно, что, открыв файл, на который программа не рассчитана, можно получить неожиданные результаты. На рисунке ниже показано, что получилось, когда я попытался открыть в приложении Microsoft Excel файл «photo.jpg», содержащий фотографию. На выходе я увидел совершенно бесполезный мусор. Но программа запустилась и что-то все-таки вывела.



Так выглядит Microsoft Excel, если открыть в ней файл «photo.jpg». В результате получается мусор, но важно, что в принципе на вход любой программе можно подать абсолютно любой файл.

На первый взгляд это может показаться нелепостью, но попробуем зайти в своем безумии еще дальше. Вспомните, что компьютерные программы сами хранятся в виде файлов на диске. Часто имена таких файлов имеют расширение «.exe» – сокращение от «executable» (исполняемый); это просто означает, что программу можно «выполнить», или запустить. Но раз программа – это файл на диске, то можно подать этот файл на вход другой программе. Например, программа Microsoft Word на моем компьютере хранится в файле «WINWORD.EXE». Поэтому я могу запустить программу Excel, указав в качестве входных данных файл WINWORD.EXE, и получить восхитительный мусор, показанный на рисунке ниже.



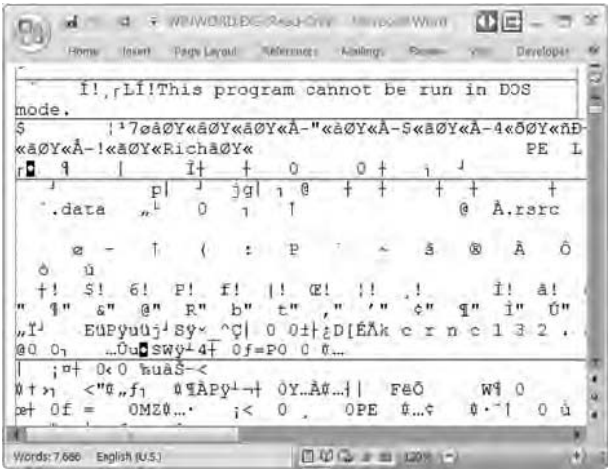
Microsoft Excel открывает Microsoft Word.
Результатом – что и неудивительно – является мусор на экране.

И снова призываю вас поэкспериментировать. Для этого нужно будет найти файл WINWORD.EXE. На моем компьютере он находится в папке «C:\Program Files\MicrosoftOffice\Office12», но точное местоположение зависит от операционной системы и установленной версии Microsoft Office. Возможно, также придется включить показ «скрытых файлов», чтобы вообще увидеть эту папку. И, кстати говоря, для этого (и следующего) эксперимента годится любая электронная таблица и текстовый редактор, так что наличие Microsoft Office необязательно.

И наконец, верх тупости. Что, если запустить программу, указав в качестве входных данных *ее саму*? Например, открыть в Microsoft Word файл WINWORD.EXE? Да попробуйте, это же нетрудно. На рисунке ниже показано, что получилось у меня на компьютере. Как и ранее, программа нормально запускается, но на экране мы видим преимущественно мусор.

И для чего все это? Цель этого раздела – познакомить вас с некоторыми неочевидными вещами, которые можно делать при запуске программы. Сейчас у вас в голове должны уложиться три несколько

странных идеи, которые впоследствии окажутся чрезвычайно важны. Во-первых, это понятие о программе, которую можно запускать с любым входным файлом, хотя, если программа не рассчитана на файл такого типа, то результатом обычно оказывается мусор. Во-вторых, мы выяснили, что программы хранятся в виде файлов на диске, поэтому на вход одной программы можно подать другую. В-третьих, мы осознали, что на вход программы можно подать файл, содержащий ее саму. До сих пор второе и третье действие порождали бессмысленный мусор, но в следующем разделе мы увидим завораживающий пример, в котором эти трюки начнут наконец приносить плоды.



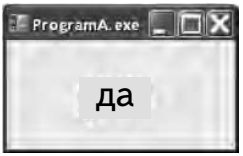
Программа Microsoft Word открывает саму себя. Открытый документ – это файл WINWORD.EXE, т. е. та самая программа, которая запускается при щелчке по значку Microsoft Word.

Некоторые программы невозможны

Компьютерам нет равных в выполнении простых инструкций – современные компьютеры выполняют несколько миллиардов команд в секунду. Поэтому возникает мысль, что любую задачу, которую можно точно описать на естественном языке, возможно записать в виде программы и отдать на выполнение компьютеру. Цель этого раздела – убедить вас в обратном: существуют простые точные предложения на русском языке, которые невозможно записать в виде компьютерной программы.

Простые программы да–нет

Для простоты мы в этом разделе будем рассматривать только очень скучный набор компьютерных программ. Назовем их программами «да–нет», потому что они умеют только показывать диалоговое окно, содержащее лишь одно слово: «да» или «нет». Так, несколько минут назад я написал программу ProgramA.exe, которая только и умеет что выводить такое окно:



В полосе заголовка этого окна мы видим имя программы – ProgramA.exe.

Я написал и еще одну программу, ProgramB.exe, которая выводит «нет» вместо «да»:

Программы ProgramA и ProgramB просты настолько, что даже не нуждаются во входных данных (а если при их запуске указать какой-то файл, они его проигнорируют). Иными словами, это примеры программ, которые ведут себя абсолютно одинаково при каждом запуске вне зависимости от входных данных.

В качестве более интересного примера программы «да–нет» я написал программу SizeChecker.exe. Она принимает на входе файл и выводит «да», если его размер больше 10 килобайт, и «нет» в противном случае. Если я щелкну правой кнопкой мыши по 50-мегабайтному файлу видео (например, mymovie.mpg), выберу из контекстного меню команду «Открыть с помощью...» и затем программу SizeChecker.exe, то увижу такое окно:

С другой стороны, если я запущу ту же программу для небольшого 3-килобайтного файла, содержащего сообщение электронной почты (скажем, myemail.msg), то увижу другое окно:

Таким образом, SizeChecker.exe – пример программы «да–нет», которая иногда выводит «да», а иногда «нет».

Теперь рассмотрим немного отличающуюся программу, которую назовем NameSize.exe. Она исследует *имя* входного файла. Если в имени есть хотя бы один знак, то NameSize.exe выводит «да», иначе «нет». Что может выводить такая программа? По определению, имя входного файла содержит хотя бы один символ (иначе у файла вообще нет имени, и мы не смогли бы его даже выбрать). Поэтому NameSize.exe всегда выводит «да» независимо от входных данных.

Кстати, вышеупомянутые программы – это первые примеры программ, которые не выводят мусор, когда им на вход подаются другие программы. Например, размер файла NameSize.exe оказался равен 8 килобайт. Поэтому если подать на вход SizeChecker.exe файл NameSize.exe, то появится окно «нет» (потому что размер NameSize.exe не превышает 10 килобайт). Можно даже запустить SizeChecker.exe, указав на входе ее саму. На этот раз будет выведено «да», потому что размер SizeChecker.exe оказался больше 10 килобайт (на самом деле, около 12). Можно и NameSize.exe запустить для нее самой; будет выведено «да», потому что имя файл «NameSize.exe» уж точно содержит хотя бы один символ. Согласен, все эти программы «да–нет» довольно скучные, но важно понять, как они себя ведут, поэтому убедитесь, что вы согласны со всеми строками в приведенной ниже таблице.

AlwaysYes.exe: программа да–нет, анализирующая другие программы

Теперь можно подумать и о гораздо более интересных программах «да–нет». Для начала изучим программу AlwaysYes.exe. Она исследует поданный на вход файл и выводит «да», если этот файл сам является программой «да–нет», которая *всегда* выводит «да». В против-

ном случае AlwaysYes.exe выводит «нет». Отметим, что AlwaysYes.exe отлично работает для любого входного файла. Если на вход подана не исполняемая программа (например, файл addresslist.docx), то она выведет «нет». Если подана исполняемая программа, не относящаяся к типу «да–нет» (например, WINWORD.EXE), то результатом будет «нет». Если подана программа «да–нет», которая иногда выводит «нет», то AlwaysYes.exe выведет «нет». И только если на вход подана программа «да–нет», которая *всегда* выводит «да», то AlwaysYes.exe выведет «да». Ранее мы видели две такие программы: ProgramA.exe и NameSize.exe. В таблице на следующей странице показано, что AlwaysYes.exe выводит для разных входных файлов, в том числе для себя самой. Как видно из последней строки таблицы, AlwaysYes.exe выводит «нет», если подать ей на вход ее саму, потому что для некоторых входных файлов она выводит «нет».

Программа	Входной файл	Результат
ProgramA.exe	address-list.docx	да
ProgramA.exe	ProgramA.exe	да
ProgramB.exe	address-list.docx	нет
ProgramB.exe	ProgramA.exe	нет
SizeChecker.exe	mymovie.mpg (50 МБ)	да
SizeChecker.exe	myemail.msg (3 КБ)	нет
SizeChecker.exe	NameSize.exe	нет
SizeChecker.exe	SizeChecker.exe (12 КБ)	да
NameSize.exe	mymovie.mpg	да
NameSize.exe	ProgramA.exe	да
NameSize.exe	NameSize.exe	да

Результаты, выводимые некоторыми простыми программами «да–нет». Обратите внимание на различие между программами, которые всегда выводят «да» независимо от входных данных (например, ProgramA.exe, NameSize.exe), и программами которые выводят «нет» иногда (например, SizeChecker.exe) либо всегда (например, ProgramB.exe).

В предпоследней строке этой таблицы присутствует еще не упоминавшаяся программа Freeze.exe. Она делает то, что больше всего ненавидят пользователи компьютеров: «виснет» (вне зависимости от входных данных). Возможно, вы сталкивались с этим явлением: видеоигра или приложение замирают (или «зависают») и отказываются реагировать на любые действия. После этого остается только

принудительно снять программу. Если и это не работает, то, возможно, придется отключить компьютер от питания (в случае ноутбука для этого иногда нужно вынимать аккумуляторы!) и перезагрузиться. Программы могут зависать по разным причинам. Иногда из-за «взаимоблокировки», обсуждавшейся в главе 8. А иногда потому, что программа занята бесконечным вычислением – например, раз за разом ищет данные, которых нет.

AlwaysYes.exe выводит

Входной файл	Результат
address-list.docx	нет
mymovie.mpg	нет
WINWORD.EXE	нет
ProgramA.exe	да
ProgramB.exe	нет
NameSize.exe	да
SizeChecker.exe	нет
Freeze.exe	нет
AlwaysYes.exe	нет

Что AlwaysYes.exe выводит для различных входных данных. «Да» выводится только для программ «да–нет», которые всегда выводят «да» – в данном случае это ProgramA.exe и NameSize.exe.

Понимать, что именно делают зависающие программы, нам необязательно. Достаточно знать, что должна сделать AlwaysYes.exe, если такая программа подана ей на вход. AlwaysYes.exe была определена достаточно четко, поэтому ответ на этот вопрос ясен: AlwaysYes.exe выводит «да», если поданная ей на вход программа всегда выводит «да», в противном случае выводится «нет». Поэтому, если на вход подана программа типа Freeze.exe, то AlwaysYes.exe обязана вывести «нет», что и показано в предпоследней строке таблицы.

YesOnSelf.exe: упрощенный вариант AlwaysYes.exe

Вы, наверное, уже подумали, что AlwaysYes.exe – изощренная и полезная программа, коль скоро она умеет анализировать другие программы и предсказывать результаты их работы. Признаюсь, я эту программу фактически не писал, а лишь рассказал, как она должна

была бы себя вести, если бы я ее написал. А теперь я опишу еще одну программу, YesOnSelf.exe. Она похожа на AlwaysYes.exe, только проще.

YesOnSelf.exe выводит

Входной файл	Результат
address-list.docx	нет
mymovie.mpg	нет
WINWORD.EXE	нет
ProgramA.exe	да
ProgramB.exe	нет
NameSize.exe	да
SizeChecker.exe	да
Freeze.exe	нет
AlwaysYes.exe	нет
YesOnSelf.exe	???

Что YesOnSelf.exe выводит для различных входных данных. «Да» выводится только для программ «да–нет», которые выводят «да», если на вход подана та же программа – в данном случае это ProgramA.exe, NameSize.exe и SizeChecker.exe. Последняя строка таблицы – загадка, потому что любой возможный результат вроде бы годится. Обсуждение приведено в тексте.

Вместо того чтобы выводить «да», когда поданная на вход программа *всегда* выводит «да», YesOnSelf.exe выводит «да», когда входная программа выводит «да», если *ей на вход подана она сама*. В противном случае YesOnSelf.exe выводит «нет». Иными словами, если на вход YesOnSelf.exe подать SizeChecker.exe, то YesOnSelf.exe подвергнет SizeChecker.exe анализу, чтобы определить, что выводит SizeChecker.exe, когда ей на вход подается SizeChecker.exe. Выше мы уже видели (см. таблицу на стр. 203), что SizeChecker.exe выводит «да», если запущена с SizeChecker.exe на входе. Поэтому YesOnSelf.exe выведет для SizeChecker.exe «да». Рассуждая аналогично, можно определить, что выводит YesOnSelf.exe для других входных данных. Отметим, что если входной файл не является программой «да–нет», то YesOnSelf.exe автоматически выводит «нет». В таблице выше показаны результаты YesOnSelf.exe для некоторых входных файлов – убедитесь, что вы понимаете, как получена каждая строка, потому что это очень важно для понимания поведения YesOnSelf.exe.

Необходимо отметить еще две особенности этой любопытной программы, YesOnSelf.exe. Во-первых, взгляните на последнюю строку таблицы. Что должна вывести YesOnSelf.exe, если ей на вход подать ее саму? По счастью, возможностей всего две, так что рассмотрим их по очереди. Если программа выводит «да», то мы знаем (согласно определению YesOnSelf.exe), что YesOnSelf.exe должна вывести «да», когда получает на входе себя же. Так сразу и не выговоришь, но хорошо подумав, вы поймете, что никаких противоречий здесь нет, поэтому возникает желание признать «да» правильным ответом.

Но не будем торопиться. Что если YesOnSelf.exe, получив на входе саму себя, выводит «нет»? Что ж, это означало бы (опять же согласно определению YesOnSelf.exe), что YesOnSelf.exe должна вывести «нет», когда ей на вход подана она же. И это утверждение ничуть не противоречиво! Выходит, что YesOnSelf.exe может выбрать, какой ответ выводить. И если она будет неукоснительно следовать этому выбору, то ответ будет правильным. Эта загадочная свобода выбора в поведении YesOnSelf.exe, как мы скоро увидим, – лишь невинно выглядящая верхушка предательского айсберга. А пока продолжим исследование этого вопроса.

Еще следует отметить, что и программу YesOnSelf.exe я тоже фактически не писал. Я лишь описал ее поведение. Однако, если предположить, что я *все же написал* AlwaysYes.exe, то и YesOnSelf.exe написать было бы просто. Спросите, почему? Потому что YesOnSelf.exe проще, чем AlwaysYes.exe: ей нужно исследовать только один вход, а не все возможные.

AntiYesOnSelf.exe: противоположность YesOnSelf.exe

Пора перевести дыхание и вспомнить, что мы собираемся сделать. Цель этой главы – доказать, что программа обнаружения сбоев существовать не может. Но непосредственная наша задача скромнее. В этом разделе мы просто пытаемся найти пример какой-нибудь невозможной программы. Это станет полезной вехой на пути к конечной цели, потому что, увидев, *как* доказывается невозможность существования программы, мы сможем воспользоваться той же техникой для программы поиска сбоев. И рад сообщить вам, что эта веха уже совсем недалеко. Осталось рассмотреть еще всего одну программу «да–нет» – и мы у цели.

Очередная программа называется AntiYesOnSelf.exe. Как следует из названия, она очень похожа на YesOnSelf.exe – на самом деле, сов-

падает с ней во всем, кроме результата. Там, где YesOnSelf.exe выводит «да», AntiYesOnSelf.exe выводит «нет». И наоборот.

Если входным файлом является программа «да–нет»,
то AntiYesOnSelf.exe отвечает на вопрос:

Будет ли поданная на вход программа при запуске для себя самой выводить «нет»?

Краткое описание поведения AntiYesOnSelf.exe.

Хотя это полное и точное определение поведения AntiYesOnSelf.exe, будет полезно сформулировать его еще более явно. Напомним, что YesOnSelf.exe выводит «да», если поданная на вход программа выводит «да», когда ей на вход подана она сама, и «нет» в противном случае. Следовательно, AntiYesOnSelf.exe выводит «нет», если поданная на вход программа выводит «да», когда ей на вход подана она сама, и «да» в противном случае. Или по-другому – AntiYesOnSelf.exe отвечает на следующий вопрос о поданном ей на вход файле: «Верно ли, что входной файл, когда ему на вход подан он сам, не выведет “да”?».

Соглашусь, что от такого описания AntiYesOnSelf.exe тоже можно свихнуться. Не проще было бы переформулировать его так: «Будет ли входной файл при запуске для себя самого выводить “нет”?»? Почему такая формулировка неправильна? К чему этот птичий язык адвокатов: «не выведет “да”», когда можно сказать просто и ясно: «выведет “нет”»? А дело в том, что иногда программа может не выводить ни «да», ни «нет», а делать что-то еще. Поэтому, когда нам говорят, что некоторая программа не выводит «да», мы не вправе автоматически заключить, что она выводит «нет». Например, программа может вывести мусор или зависнуть. Однако существует ситуация, в которой можно сделать более определенный вывод: если нам заранее известно, что программа относится к типу «да–нет», то мы знаем, что она никогда не зависает и никогда не выводит мусор – она всегда завершается и выводит либо «да», либо «нет». Поэтому для программ «да–нет» адвокатская невнятица насчет «не выводит “да”» эквивалентна более простому утверждению «выводит “да”».

Вот теперь, наконец, мы можем дать очень простое описание поведения AntiYesOnSelf.exe. Если входной файл является программой «да–нет», то AntiYesOnSelf.exe отвечает на вопрос «Будет ли поданная на вход программа при запуске для себя самой выводить “нет”?». Эта формулировка настолько важна для дальнейшего, что выше я поместил ее в рамочку.

С учетом работы, уже проделанной при анализе YesOnSelf.exe, легко составить таблицу результатов работы AntiYesOnSelf.exe. На самом деле, мы просто скопировали предыдущую таблицу, заменив всюду «да» на «нет» и наоборот. Получилась таблица, показанная выше. Как обычно, рекомендую тщательно осмыслить все строки таблицы и убедиться, что вы с ними согласны. Если входным файлом является программа «да–нет», то можно использовать простую формулировку в рамочке вместо приведенной выше более сложной.

AntiYesOnSelf.exe выводит

Входной файл	Результат
address-list.docx	да
mymovie.mpg	да
WINWORD.EXE	да
ProgramA.exe	нет
ProgramB.exe	да
NameSize.exe	нет
SizeChecker.exe	нет
Freeze.exe	да
AlwaysYes.exe	да
AntiYesOnSelf.exe	???

Что AntiYesOnSelf.exe выводит для различных входных данных. По определению AntiYesOnSelf.exe выводит ответ, противоположный YesOnSelf.exe, а, значит, эта таблица – за исключением последней строки – совпадает с таблицей на стр. 205 с тем отличием, что «да» всюду заменено на «нет» и наоборот. С последней строкой возникает серьезное затруднение, которое объясняется в тексте.

Из последней строки видно, что при попытке вычислить результат AntiYesOnSelf.exe при запуске для нее самой возникает проблема. Чтобы разобраться в ней, давайте сначала еще упростим описание AntiYesOnSelf.exe, заключенное выше в рамочку: вместо рассмотрения всех возможных программ «да–нет» на входе ограничимся тем, что происходит, когда AntiYesOnSelf.exe получает на входе саму себя. Тогда фразу «Будет ли поданная на вход программа ...» можно заменить на «Будет ли AntiYesOnSelf.exe ...» – потому что на вход как раз и подается AntiYesOnSelf.exe. Это и есть нужная нам окончательная формулировка, которая также заключена в рамочку.

Если входным файлом является она сама, то AntiYesOnSelf.exe отвечает на вопрос:

Будет ли AntiYesOnSelf.exe при запуске для себя самой выводить «нет»?

Краткое описание поведения AntiYesOnSelf.exe, когда ей на вход подана она сама. Отметим, что это упрощенный вариант описания на стр. 207, в котором рассматривается всего один случай: когда входным файлом является AntiYesOnSelf.exe.

Вот теперь мы и впрямь готовы выяснить, что выводит AntiYesOnSelf.exe, если на вход ей подана она сама. Есть всего две возможности («да» и «нет»), поэтому с их рассмотрением не должно возникнуть трудностей. Начнем по порядку.

Случай 1 (*выводится «да»*). Если выводится «да», то ответом на выделенный полужирным шрифтом вопрос будет «нет». Но ответом на этот вопрос, по определению, является результат, который выводит AntiYesOnSelf.exe (прочитайте обведенное рамочкой описания, чтобы убедиться), и, следовательно, должно выводиться «нет». Таким образом, мы только что доказали, что если выводится «да», то выводится «нет». Невозможно! По существу, мы пришли к *противоречию*. (Если вы незнакомы с техникой доказательства от противного, прочитайте обсуждение в начале этой главы. Ниже мы воспользуемся этой техникой еще несколько раз.) Коль скоро мы получили противоречие, значит, наше предположение о том, что выводится «да», было неверным. Мы доказали, что AntiYesOnSelf.exe, когда ей на вход подана она сама, не может выводить «да». Перейдем к другой возможности.

Случай 2 (*выводится «нет»*). Если выводится «нет», то ответом на выделенный полужирным шрифтом вопрос будет «да». Но, как и в случае 1, ответом на этот вопрос, по определению, является результат, который выводит AntiYesOnSelf.exe, и, следовательно, должно выводиться «да». Иными словами, мы только что доказали, что если выводится «нет», то выводится «да». Мы снова получили противоречие, значит, наше предположение о том, что выводится «нет», не может быть верным. Мы доказали, что AntiYesOnSelf.exe, когда ей на вход подана она сама, не может выводить «нет».

И что теперь? Мы исключили оба возможных варианта вывода при запуске AntiYesOnSelf.exe для самой себя. И это тоже противоречие: мы определили AntiYesOnSelf.exe как программу «да–нет», т. е. программу, которая всегда завершается и выводит либо «да», либо «нет». И вместе с тем только что продемонстрировали, что AntiYesOnSelf.

exe не может вывести ни один из этих результатов! Это противоречие доказывает, что исходное предположение было неверным: *невозможно* написать программу «да–нет», которая вела бы себя, как AntiYesOnSelf.exe.

Теперь вы понимаете, почему я честно признался, что не писал ни одной из программ AlwaysYes.exe, YesOnSelf.exe и AntiYesOnSelf.exe. Я лишь описал, как они вели бы себя, если бы я их написал. В последнем абзаце мы от противного доказали, что программа AntiYesOnSelf.exe существовать не может. Но мы можем доказать даже больше: AlwaysYes.exe и YesOnSelf.exe также невозможны! Почему? Как вы, наверное, догадались, нашим инструментом снова станет доказательство от противного. Выше я сказал, что если бы AlwaysYes.exe существовала, то, внеся в нее небольшие изменения, можно было бы создать YesOnSelf.exe. А если бы существовала YesOnSelf.exe, то было бы совсем просто получить AntiYesOnSelf.exe – достаточно было бы заменить выводимые результаты на противоположные («да» вместо «нет» и «наоборот»). Короче говоря, если AlwaysYes.exe, то существует и AntiYesOnSelf.exe. Но мы уже знаем, что AntiYesOnSelf.exe существовать не может, следовательно, не может существовать и AlwaysYes.exe. Аналогичное рассуждение показывает, что существование YesOnSelf.exe тоже невозможно.

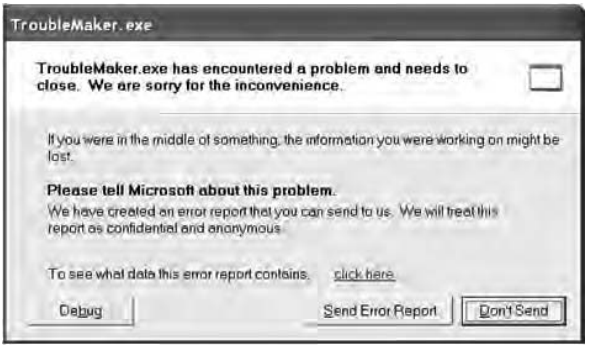
Напомню, весь этот раздел – лишь веха на пути к конечной цели: доказательству невозможности программ обнаружения сбоев. Пока мы ставили более скромную задачу: привести какие-нибудь примеры невозможных программ. Этой цели мы достигли, исследовав целых три программы, которые никак не могут существовать. Из них самой интересной была AlwaysYes.exe. Две другие не так прозрачны, поскольку в их описании участвует поведение программ, которые передаются сами себе в качестве входных данных. С другой стороны, AlwaysYes.exe – весьма мощная программа, поскольку, если бы она существовала, то могла бы проанализировать любую другую программу и сообщить, верно ли, что она всегда выводит «да». Увы, как мы только что видели, написать такую умную и многообещающую программу никому не под силу.

Невозможность обнаружения сбоев

И наконец, мы в силах приступить к рассмотрению программы, которая анализирует другие программы и определяет, будут они па-

дать или нет. Точнее, мы докажем, что такая программа невозможна. Прочитав несколько предшествующих страниц, вы, наверное, предчувствуете, что мы воспользуемся доказательством от противного. То есть сначала предположим, что наш Святой Грааль существует: что есть такая программа CanCrash.exe, которая способна сказать, будет какая-то другая программа падать или нет. Проделав с CanCrash.exe странные, таинственные и восхитительные манипуляции, мы придем к противоречию.

Один из шагов доказательства потребует от нас взять абсолютно правильную программу и изменить ее так, чтобы при определенных обстоятельствах она заведомо «падала». Как это сделать? Да очень просто. Программа может падать по самым разным причинам. Одна из наиболее распространенных – попытка деления на нуль. В математике результат такой операции называют «неопределенным». С точки зрения компьютера, «неопределенность» означает серьезную ошибку, после которой программа не может продолжать работу и аварийно завершается – падает. Поэтому, чтобы гарантированно заставить программу падать, нужно вставить в нее пару дополнительных команд, которые приведут к делению на нуль. Именно так я и получил программу TroubleMaker.exe, показанную на следующем рисунке.



Результат сбоя программы в одной операционной системе. В других операционных системах сбой может выглядеть иначе, но спутать его ни с чем невозможно. Программа TroubleMaker.exe специально была написана так, чтобы вызвать сбой и тем самым продемонстрировать, насколько это просто.

Теперь начнем собственно доказательство невозможности существования программы обнаружения сбоев. На рисунке ниже показан ход рассуждений. Сначала мы предположим, что существует программа CanCrash.exe, которая является программой типа «да–нет», всегда завершается и выводит «да», если на вход подана программа,

которая при определенных обстоятельствах может упасть, и «нет», если поданная на вход программа не падает никогда. Теперь подвергнем CanCrash.exe странному преобразованию: вместо того чтобы выводить «да», пусть она падает! (Как было сказано выше, это очень просто сделать, намеренно добавив деление на нуль.) Назовем получившуюся программу CanCrashWeird.exe. Итак, эта программа падает – с появлением диалогового окна, аналогичного показанному выше, – если поданная ей на вход программа может упасть, и выводит «нет», если та не падает никогда.

Имя программы	Поведение программы
CanCrash.exe	• выводит **да**, если поданная на вход программа может упасть • выводит **нет**, если поданная на вход программа никогда не падает
↓	
CanCrashWeird.exe	• **падает**, если поданная на вход программа может упасть • выводит **нет**, если поданная на вход программа никогда не падает
↓	
CrashOnSelf.exe	• **падает**, если поданная на вход программа падает, когда ей на вход подана она сама • выводит **нет**, если поданная на вход программа не падает, когда ей на вход подана она сама
↓	
AntiCrashOnSelf.exe	• выводит **да**, если поданная на вход программа падает, когда ей на вход подана она сама • **падает**, если поданная на вход программа не падает, когда ей на вход подана она сама

Последовательность четырех невозможных программ обнаружения сбоев. Последняя программа AntiCrashOnSelf.exe, очевидно, невозможна, потому что при попытке подать ей на вход ее же саму возникает противоречие. Однако каждая из этих программ получается из предыдущей путем небольшой модификации. Следовательно, ни одна из них существовать не может.

Следующий шаг – преобразовать CanCrashWeird.exe в еще более странное создание, которое называется CrashOnSelf.exe. Эту программу, как и программу YesOnSelf.exe из предыдущего раздела, интересует только, как ведут себя программы, которым на вход поданы они сами. Точнее, CrashOnSelf.exe исследует входную программу и намеренно падает, если эта программа упадет, когда ей на вход подана она сама. В противном случае она выводит «нет». Отметим, что полу-

чить CrashOnSelf.exe из CanCrashWeird.exe очень просто: процедура точно такая же, как при трансформации AlwaysYes.exe в YesOnSelf.exe (см. стр. 206).

Последней в последовательности четырех программ на рисунке является программа AntiCrashOnSelf.exe, получающаяся трансформацией CrashOnSelf.exe. На этом шаге поведение исходной программы просто меняется на противоположное: если поданная на вход программа падает, когда ей на вход подана она сама, то AntiCrashOnSelf.exe выводит «да». В противном случае AntiCrashOnSelf.exe падает.

Вот теперь мы может продемонстрировать противоречие. Что должна делать AntiCrashOnSelf.exe, если ей подать на вход ее же саму? Согласно своему определению, она должна выводить «да», если падает (противоречие, потому что она не может ничего вывести, если уже упала). И опять-таки согласно своему определению, AntiCrashOnSelf.exe должна падать, если не падает – очевидное противоречие. Мы исключили оба мыслимых варианта поведения AntiCrashOnSelf.exe, а, следовательно, такая программа существовать не может.

Наконец, мы можем использовать показанную выше цепочку преобразований, чтобы доказать невозможность CanCrash.exe. Действительно, *если бы* она существовала, то, переходя от одного шага к другому, мы могли бы ее преобразовать в AntiCrashOnSelf.exe – но, как мы уже знаем, AntiCrashOnSelf.exe невозможна. В силу этого противоречия предположение о существовании CanCrash.exe обязано быть ложным.

Проблема остановки и неразрешимость

На этом мы завершаем описание одного из самых значительных результатов информатики. Мы доказали, что никто и никогда не сможет написать программу наподобие CanCrash.exe – которая анализирует другие программы и находит в них все возможные ошибки, способные привести к падению, т. е. аварийному завершению.

На самом деле, когда Алан Тьюринг, основатель теоретической информатики, первым доказал подобный результат в 1930-х годах, он думал вовсе не об ошибках или сбоях. Тогда ведь и электронных компьютеров-то не было. Тьюринга интересовало, даст ли компьютерная программа какой-нибудь ответ в конечном итоге. И тесно связанный с этим вопрос: *завершится* ли когда-нибудь или будет работать вечно, не давая никакого ответа? Вопрос о том, завершится ли, или «остановится» ли данная программа, получил название «проблема остановки». Величайшее достижение Тьюринга заключалось в дока-

зательстве того, что сформулированный им вариант проблемы остановки является, как говорят в информатике, «неразрешимой» задачей. Неразрешимой называют задачу, которую невозможно решить, написав программу для компьютера. Таким образом, результат Тьюринга можно сформулировать иначе: невозможно написать программу AlwaysHalts.exe, которая выводит «да», если поданная ей на вход программа останавливается, и «нет» в противном случае.

В таком виде проблема остановки очень напоминает задачу, рассмотренную в этой главе, которую можно было бы назвать «проблемой падения». Мы доказали неразрешимость проблемы падения и с помощью аналогичной техники могли бы доказать неразрешимость проблемы остановки. Как нетрудно догадаться, в информатике существует еще много неразрешимых задач.

Что следует из невозможности некоторых программ?

Это последняя глава в книге, дальше только послесловие. Я сознательно включил ее в качестве контрапункта к предыдущим главам. Если в предыдущих главах рассматривались идеи, показывающие, что компьютеры в некоторых отношениях мощнее и полезнее людей, то в этой мы увидели одно из фундаментальных ограничений компьютеров. Мы убедились, что существуют задачи, который компьютер не сможет решить никогда, каким бы мощным он ни был и сколь бы искусно его ни программировали. И к числу таких неразрешимых задач относятся и потенциально полезные, например, анализ других программ на предмет возможности сбоя.

В чем значимость этого странного факта, который можно даже рассматривать как предвестника беды? Отражается ли существование неразрешимых задач на практическом использовании компьютеров? А как насчет вычислений, которые мы проделываем в мозгу, – для них тоже есть неразрешимые задачи?

Неразрешимость и использование компьютеров

Сначала поговорим о практических последствиях неразрешимости для применения компьютеров. Краткий ответ: нет, неразрешимость не оказывает существенного влияния на их повседневное применение. Тому есть две причины. Во-первых, неразрешимость связана с

вопросом о том, можно ли вообще получить от программы ответ, без каких бы то ни было временных ограничений. На практике, однако, вопрос об эффективности (сколько времени ждать ответа) чрезвычайно важен. Существует немало разрешимых задач, для которых неизвестны эффективные алгоритмы. Одна из самых известных – задача коммивояжера. В современной формулировке она звучит примерно так: пусть требуется облететь много городов (скажем, 20, 30 или 100). В каком порядке нужно их посещать, что общая стоимость авиабилетов была минимальна? Как мы уже сказали, эта задача разрешима – даже зеленый новичок сможет написать программу, которая находит самый дешевый маршрут. Беда в том, что такая программа будет работать миллионы лет. На практике это никуда не годится. Так что сам факт разрешимости еще не означает, что задачу можно решить практически.

Перейдем ко второй причине, по которой практические последствия неразрешимости не так существенны: как выясняется, часто неразрешимые задачи допускают решение *в большинстве случаев*. И это прекрасно иллюстрирует главный пример данной главы. Пройдя по цепочке рассуждений, мы доказали, что никакая программа не сможет найти все ошибки во всех компьютерных программах. Тем не менее, мы можем попытаться написать программу обнаружения сбоев в надежде, что она сможет найти большинство ошибок во многих типах программ. Скажу больше, в этой области ведутся активные исследования. И надежностью программного обеспечения, которая так выросла за последние несколько десятков лет, мы отчасти обязаны прогрессу именно в области программ обнаружения сбоев. Так что зачастую возможно найти весьма полезные частичные решения неразрешимых задач.

Неразрешимость и мозг

Имеет ли существование неразрешимых задач какие-то последствия для мыслительных процессов человека? Этот вопрос заводит нас прямиком в темные воды философии, точнее, таких ее классических проблем, как определение сознания и различие между разумом и мозгом. Но одно можно сказать точно: если вы полагаете, что мозг человека в принципе возможно смоделировать на компьютере, то мозг подвержен тем же ограничениям, что и компьютеры. Иначе говоря, придется согласиться, что некоторые задачи человеческому мозгу неподвластны, каким бы он ни был умным или хорошо тренированным. Этот вывод непосредственно следует из основного результата

данной главы. Если мозг можно смоделировать с помощью компьютерной программы и если мозг может решать неразрешимые задачи, то и компьютерная модель мозга смогла бы эти задачи решить – что противоречит тому факту, что программа не в состоянии решить неразрешимую задачу.

Разумеется, вопрос о том, сможем ли мы когда-нибудь построить точную компьютерную модель мозга, еще очень далек от своего разрешения. Похоже, что с научной точки зрения фундаментальных препятствий нет, поскольку низкоуровневые детали передачи химических и электрических сигналов в мозге довольно хорошо изучены. С другой стороны, существуют философские аргументы в пользу того, что физические процессы в мозге каким-то образом создают «разум», который качественно отличается от любой физической системы, которую можно было бы смоделировать на компьютере. Эти аргументы предстают в разных формах и могут апеллировать, например, к нашей способности к рефлексии и интуиции, а также к духовному началу.

Здесь существует знаменательная связь со статьей Алана Тьюринга 1937 года, посвященной неразрешимости, – статьей, которая, как многие считают, заложила основы информатики как научной дисциплины. К сожалению, название статьи несколько туманно: оно начинается невинно звучащей фразой «О вычислимых числах...», а заканчивается лишенным всякой гармонии «...с применением к Entscheidungsproblem[1]». (Мы не будем вникать во вторую часть названия!) Важно понимать, что в 1930-е годы слово «computer» имело совершенно другой смысл, чем сейчас. Для Тьюринга это был *человек*, который выполнял какие-то вычисления с помощью карандаша и бумаги. Поэтому «вычислимые числа» в названии статьи – это числа, который человек в принципе мог вычислить. В своих рассуждениях Тьюринг описывает машину определенного вида (для Тьюринга «машина» – это то, что мы сейчас называем «компьютером»). Часть статьи посвящена демонстрации того, что некоторые вычисления машинам недоступны, – это и есть доказательство неразрешимости, которое мы уже подробно обсудили. А в другой части той же статьи детально и убедительно обосновывается, что «машина» Тьюринга (читай: компьютер) может произвести любое вычисление, которое способен выполнить вычислитель (человек).

Наверное, вы начинаете понимать, почему основополагающая статья Тьюринга «О вычислимых числах...» имеет такое непреходящее значение. В ней не только поставлены и решены некоторые фундамен-

[1] Проблема разрешения. *Прим. перев.*

тальные проблемы информатики, автор еще и углубляется в область философии и убедительно показывает, что мыслительные процессы человека можно смоделировать на компьютере (который, напомню, тогда еще даже не был изобретен!). На современном философском языке эта идея – о том, что все компьютеры, а, возможно, и люди тоже, эквивалентны с точки зрения вычислительной мощности, – известна под названием *тезис Чёрча-Тьюринга*. В этом названии отдано должное не только Алану Тьюрингу, но и Алонзо Чёрчу, который (как уже отмечалось) независимо открыл существование неразрешимых задач. На самом деле, Чёрч опубликовал свою работу на несколько месяцев раньше Тьюринга, но формулировки Чёрча более абстрактны, и вычисления с помощью машин в них явно не упоминаются.

Споры о справедливости тезиса Чёрча-Тьюринга не утихают. Но если верна его самая сильная формулировка, то неразрешимость ставит пределы не только нашим компьютерам. Те же ограничения относятся как к джинну, подвластному кончикам наших пальцев, так и к джинну, стоящему за пальцами: нашим разумом.

ГЛАВА 11. Послесловие: еще один услужливый джинн?

Мы можем заглянуть вперед не очень далеко, но видим там многое, что нужно сделать.

— Алан Тьюринг «*Вычислительные машины и разум*», 1950

В 1991 году мне посчастливилось побывать на публичной лекции великого физика-теоретика Стивена Хокинга. В лекции с дерзким названием «Будущее Вселенной» Хокинг уверенно предсказал, что в течение ближайших 10 миллиардов лет, а то и больше Вселенная продолжит расширяться. Иронично усмехнувшись, он добавил: «Полагаю, к тому времени доказать мне, что я не прав, будет уже невозможно». К несчастью для меня, у предсказаний будущего информатики нет запаса в 10 миллиардов лет, который выторговали себе космологи. Любые сделанные мной прогнозы вполне могут быть опровергнуты, пока я жив.

Но это не должно помешать нашим размышлениям о будущем великих идей информатики. Будут ли исследованные нами великие алгоритмы во веки веков считаться «великими»? Устареют ли какие-то из них? Появятся ли новые великие алгоритмы? Чтобы ответить на эти вопросы, мы должны думать не как космологи, а как историки. И тут мне вспоминается, как много лет назад я смотрел по телевизору лекции известного, хотя и неоднозначного, историка из Оксфорда, А. Дж. П. Тейлора. В конце цикла лекций Тейлор прямо ответил на вопрос, будет ли третья мировая война. Он считал, что будет, потому что люди, скорее всего, «будут вести себя в будущем так же, как вели себя в прошлом».

Последуем примеру Тейлора и примем во внимание исторические качели. Великие алгоритмы, описанные в этой книге, возникли в результате случайностей и изобретений, рассыпанных по всему 20 веку.

Кажется разумным предположить, что такой же темп сохранится и в 21 веке, так что новые значительные идеи в области алгоритмов будут появляться каждые 20–30 лет. Иногда эти алгоритмы будут ошеломляюще неожиданными, совершенно новыми методами, придуманными учеными. Криптография с открытым ключом и связанные с ней алгоритмы цифровой подписи могут служить примерами подобного рода. А иногда окажется, что алгоритмы уже какое-то время существовали в недрах сообщества исследователей, ожидая, когда ветер поднимет волну новой технологии, в которой они смогут найти свое место. Таковы применяемые в поисковых системах алгоритмы индексирования и ранжирования; подобные им много лет существовали в дисциплине информационного поиска, но потребовался феномен веб, чтобы сделать их «великими» в том смысле, что они повседневно применяются широкими массами пользователей компьютеров. Конечно, эти алгоритмы были адаптированы к новой сфере применения, тому пример PageRank.

Отметим, что появление новой технологии не обязательно порождает новые алгоритмы. Взять хоть феноменальный рост количества ноутбуков в 1980–1990 годы. Ноутбуки произвели революцию в способах использования компьютеров, резко увеличив их доступность и мобильность. Они также послужили мощным стимулом прогресса в таких важных отраслях, как технология производства экранов и управления электропитанием. Но я бы сказал, что революция ноутбуков не принесла с собой никаких великих алгоритмов. А вот распространение Интернета, напротив, породило великие алгоритмы: предоставив инфраструктуру, в которой могли существовать поисковые системы, Интернет позволил алгоритмам индексирования и ранжирования достичь полной зрелости.

Поэтому бесспорное ускорение технологического развития, которое мы продолжаем наблюдать повсюду, само по себе не гарантирует появления новых алгоритмов. На самом деле, существует мощная историческая сила, которая действует в противоположном направлении и наводит на мысль, что темп алгоритмических инноваций в будущем, скорее всего, замедлится. Я имею в виду тот факт, что информатика превращается в зрелую научную дисциплину. По сравнению с такими науками, как физика, математика и химия, информатика еще очень молода: она берет начало в 1930-х годах. Поэтому осмелюсь предположить, что великие алгоритмы, открытые в 20 веке, были низко висящими плодами, а в будущем отыскивать изобретательные алгоритмы с широкой сферой применимости станет все труднее и труднее.

Таким образом, налицо две противоборствующих тенденции: новые ниши, открываемые новыми технологиями, иногда дают место новым алгоритмам, тогда как взросление науки сужает эти возможности. Я склонен думать, что эти тенденции уравновешивают друг друга, а в результате нас ждет медленный, но неуклонный процесс рождения новых алгоритмов, который растянется на долгие годы.

О некоторых потенциально великих алгоритмах

Разумеется, какие-то новые алгоритмы станут полной неожиданностью, поэтому сказать о них что-нибудь заранее невозможно. Но у некоторых ниш и методов определенно имеется потенциал. Одна из очевидных тенденций – растущее применение искусственного интеллекта (и, в частности, распознавания символов) в повседневных задачах, и будет очень интересно присутствовать при появлении какой-нибудь радикально новой алгоритмической жемчужины в этой области.

Еще одна плодородная нива – класс алгоритмов под названием «протоколы с нулевым разглашением». В них используются специальные криптографические методы, позволяющие добиться еще более удивительных результатов, чем цифровая подпись: две и более сторон могут объединить свою информацию, не раскрывая ее отдельные фрагменты. Одно из потенциальных применений – онлайновые аукционы. С помощью протокола с нулевым разглашением участники торгов могут обменяться заявками таким образом, чтобы можно было выявить победителя, но при этом ни одна сторона не будет знать о содержании заявок всех остальных сторон! Протоколы с нулевым разглашением – настолько остроумная идея, что они обязательно вошли бы в мой свод великих алгоритмов, если бы использовались на практике. Но пока они не получили широкого распространения.

Еще одна идея, которой посвящено огромное количество академических исследований, но которая пока не нашла практического применения – «распределенные хэш-таблицы». Это изобретательный способ хранить информацию в одноранговых системах, не имеющих центрального сервера, который управлял бы потоками информации. На момент написания этой книги многие системы, объявляющие себя одноранговыми, на самом деле пользуются центральными серверами для обеспечения части своей функциональности, поэтому нужды в распределенных хэш-таблицах у них не возникает.

Метод «византийской отказоустойчивости» попадает в ту же категорию: удивительно красивый алгоритм, который пока не может быть отнесен к разряду великих из-за недостаточно широкого признания. Этот алгоритм обеспечивает устойчивость некоторых компьютерных систем к ошибкам любого типа (при условии, что одновременно возникает не слишком много ошибок). Этим он отличается от традиционной отказоустойчивости, когда система способна продолжать работу лишь после сравнительно безобидных ошибок, например, выхода из строя диска или краха операционной системы.

Могут ли великие алгоритмы уйти в тень?

Помимо вопроса о том, какие алгоритмы смогут достичь величия в будущем, интересно порассуждать на тему о том, смогут ли нынешние «великие» алгоритмы – бесценные инструменты, которыми мы пользуемся постоянно, даже не задумываясь об этом, – утратить свою важность. И здесь тоже стоит обратиться к историческому опыту. Если ограничить рассмотрение только конкретными алгоритмами, то, конечно, некоторые из них могут потерять значимость. Наиболее наглядный пример дает криптография, в которой исследователи, изобретающие новые алгоритмы, ведут непрестанную гонку вооружений с другими исследователями, который изучают способы взлома этих алгоритмов. Вспомним, к примеру, о так называемых криптографических хэш-функциях. Хэш-функция под названием MD5 принята в качестве официального стандарта Интернета и широко использовалась в начале 1990-х годов. Но с тех пор в ней были обнаружены существенные изъяны в плане безопасности, и теперь применять ее не рекомендуется. Аналогично в главе 9 мы говорили о том, что схему цифровой подписи RSA будет очень легко взломать, если удастся построить квантовый компьютер достаточно большого размера.

Однако я думаю, что подобные примеры – слишком узкая трактовка поставленного вопроса. Конечно, MD5 имеет дефекты (как, кстати говоря, и пришедшая ее на смену функция SHA-1), но это не означает, что саму идею криптографических хэш-функций следует отвергнуть. В действительности, хэш-функции используются очень широко, и в безупречных – пока безупречных – нет недостатка. Так что, если взглянуть на ситуацию шире и быть готовыми изменить детали алгоритма, сохранив его основные идеи, то маловероятно, что многие

из признанных ныне великими алгоритмов утратят свою важность в будущем.

Чему мы научились?

Можно ли извлечь из представленных алгоритмов какие-то общие уроки? Один такой урок, ставший для меня как автора книги большим сюрпризом, – возможность объяснить все великие идеи на пальцах, не требуя от читателя знания информатики или умения программировать. Приступая к работе над книгой, я думал, что великие алгоритмы можно будет разбить на две категории. В первую попадут алгоритмы, в основе которых лежит какой-нибудь простой, но остроумный трюк – трюк, который удастся объяснить, не прибегая к техническим приемам. Во вторую категорию должны были попасть алгоритмы, настолько сильно опирающиеся на продвинутые знания в области компьютеров, что объяснить их читателям без технической подготовки невозможно. Я планировал включить упоминания об алгоритмах из второй категории в виде каких-нибудь по возможности интересных историй, объяснить, в каких областях они применяются, и голословно заявить, что алгоритм остроумен, хотя я и не в состоянии объяснить, как он работает. Представьте себе мое удивление и восторг, когда я обнаружил, что все отобранные мной алгоритмы попадают в первую категорию! Конечно, многие важные технические детали пришлось опустить, но основной механизм работы удалось объяснить словами, понятными неспециалисту.

Еще один важный вывод, относящийся ко всем нашим алгоритмам, заключается в том, что информатика отнюдь не ограничивается программированием. Читая вводный курс по информатике, я спрашиваю у студентов, что, по их мнению, такое информатика. Абсолютное большинство говорят о «программировании» или о чем-то эквивалентном, например «программной инженерии». На просьбу назвать еще какие-нибудь аспекты информатики многие тушуются. Бывает, что следует упоминание о чем-то, связанным с оборудованием, например «конструирование аппаратуры». Это наглядно свидетельствует об укоренившемся в обществе непонимании того, чем на самом деле занимается информатика. Надеюсь, что, прочитав эту книгу, вы станете лучше понимать, над какими задачами ломают голову ученые, работающие в области информатики, и какие решения они находят.

Здесь может помочь простая аналогия. Допустим, вы встретили профессора, основная сфера интересов которого – японская лите-

ратура. Скорее всего, профессор может говорить, читать и писать по-японски. Но если бы вас попросили высказать гипотезу о том, на какие размышления он тратит больше всего времени в ходе своих исследований, то вряд ли вы сказали бы «о японском языке». Конечно, японский язык – важная составная часть знаний, необходимых для изучения обычаев, культуры и истории, отраженных в японской литературе. С другой стороны, человек, в совершенстве владеющий японским, может вообще ничего не знать о японской литературе (и в Японии таких людей, наверное, миллионы).

Между языками программирования и базовыми идеями информатики связь аналогичная. Чтобы реализовать алгоритмы и экспериментировать с ними, ученым необходимо преобразовывать алгоритмы в компьютерные программы, а любая программа пишется на каком-то языке, например Java, C++ или Python. Поэтому ученый, работающий в области информатики, обязан знать язык программирования, но это лишь предварительное условие; основное же его занятие – придумывать, адаптировать и понимать алгоритмы. Надеюсь, что, познакомившись с великими алгоритмами, описанными в этой книге, читатель будет отчетливее представлять себе это различие.

Конец пути

Вот и закончилось наше путешествие по миру выдающихся и в то же время повседневно выполняемых вычислений. Достигли мы своей цели? И будете ли вы в результате по-другому работать с вычислительными устройствами?

Вполне возможно, что в следующий раз зайдя на защищенный сайт, вы полюбопытствуете, кто поручился за его надежность, и захотите узнать, какую цепочку сертификатов проверил ваш браузер (глава 9). А быть может, когда в следующий раз покупка через Интернет по непонятной причине сорвется, вы не расстроитесь, а с благодарностью вспомните, что база данных обеспечивает непротиворечивость, и потому никто не возьмет с вас денег за то, что вы так и не заказали (главу 8). Или однажды, подумав «А хорошо бы, если бы компьютер мог сделать *такую* штуку?», вы поймете, что это невозможно, потому что вожделенная задача неразрешима, и это можно доказать тем же способом, каким мы доказали неразрешимость обнаружения сбоев (глава 10).

Уверен, вы обнаружите *множество* примеров, когда знание о великих алгоритмах помогает изменить способ взаимодействия с ком-

пьютером. Однако, как я предупредил во введении, не это основная цель книги. А главная *моя* цель заключалась в другом – сообщить читателю достаточно знаний о великих алгоритмах, чтобы он проникся ощущением чуда, лежащего в основе некоторых ставших уже рутинными вычислительных задач, – как астроном-любитель замирает от восхищения, глядя в ночное небо.

Только ты, любезный читатель, знаешь, достиг я этой цели или нет. Но в одном я уверен: твой личный джинн всегда готов подчиниться мановению твоих пальцев. Пользуйся этим.

БЛАГОДАРНОСТИ

Ты, дорога, иду по тебе и гляжу, но мне думается, я вижу не все,
Мне думается, в тебе много такого, чего не увидишь глазами.

— Уолт Уитмен «*Песня большой дороги*»[1]

Многие мои друзья, коллеги и члены семьи прочитали рукопись полностью или частично. Среди них: Алекс Бэйтс (Alex Bates), Уилсон Белл (Wilson Bell), Майк Бэрроуз (Mike Burrows), Уолт Хромяк (Walt Chromiak), Майкл Айзерд (Michael Isard), Алистер Маккормик (Alastair MacCormick), Райан Маккормик (Raewyn MacCormick), Николетта Марини-Майо (Nicoletta Marini-Maio), Фрэнк Макшерри (Frank McSherry), Кристин Митчелл (Kristine Mitchell), Илья Миронов (Ilya Mironov), Вэнди Поллак (Wendy Pollack), Джудит Поттер (Judith Potter), Коттен Сейлер (Cotten Seiler), Элен Такакс (Helen Takacs), Кунал Талвар (Kunal Talwar), Тим Валс (Tim Wahls), Джонатан Уоллер (Jonathan Waller), Юди Видер (Udi Wieder) и Олли Уильямс (Ollie Williams). Отзывы этих читателей позволили существенно улучшить рукопись. Как и замечания двух рецензентов, пожелавших остаться анонимными.

Крис Бишоп помогал ободрением и советом. Том Митчелл разрешил использовать его рисунки и исходный код в главе 6.

Викки Кирн (редактор книги) и ее коллеги по издательству Princeton University Press проделали блестящую работу по вынашиванию проекта и доведению его до успешного завершения.

Мои коллеги по факультету математики и информатики в колледже Дикинсон всегда были готовы оказать мне поддержку и по-дружески подбодрить.

Майкл Айзерд и Майк Бэрроуз показали мне красоту и радость информатики. Эндрю Блейк способствовал моему становлению как ученого.

Моя жена Кристин всегда была и остается рядом; *в тебе много такого, чего не увидишь глазами*.

Всем им я выражаю огромную благодарность. И посвящаю книгу Кристине – с любовью.

[1] Перевод К. И. Чуковского. *Прим. перев.*

ИСТОЧНИКИ И ЛИТЕРАТУРА ДЛЯ ДАЛЬНЕЙШЕГО ЧТЕНИЯ

Как было сказано в начале книги, в тексте нет цитат. Но все источники перечислены ниже, вместе с моими рекомендациями по дополнительному чтению для тех, кто захочет узнать больше о великих алгоритмах информатики.

Эпиграф взят из эссе Ванневара Буша «Возможный способ нашего мышления», которое было опубликовано в июльском номере журнала «The Atlantic» в 1945 году.

Введение (глава 1). Желающим познакомиться с простыми и поучительными объяснениями алгоритмов и разных компьютерных технологий рекомендую рождественские лекции Криса Бишопа, прочитанные в 2008 году в Королевском институте. Их видеозаписи можно бесплатно просмотреть в сети. В этих лекциях не предполагается никакого предварительного знания информатики. Книга А. К. Дьюдени «New Turing Omnibus» удачно дополняет некоторые рассмотренные мной темы; здесь же вы найдете введение во многие другие интересные концепции информатики. Но чтобы оценить ее в полной мере, необходимы кое-какие навыки программирования. Книга Юрайи Хромковича (Juraj Hromkovič) «Algorithmic Adventures» – отличный выбор для читателей с небольшой математической подготовкой, но без знания информатики. Из многочисленных университетских учебников по алгоритмам наиболее доступны следующие: Dasgupta, Papadimitriou, Vazirani «Algorithms», Harel and Feldman «Algorithmics: The Spirit of Computing» и Cormen, Leiserson, Rivest, Stein «Introduction to Algorithms».

Индексирование в поисковых системах (глава 2). Трюк с метасловами впервые описан в патенте США 6105019 «Поиск с ограничениями по индексу», выданном AltaVista на имя Майка Бэрроуза (2000). Читатели, имеющие подготовку в области информатики, смогут много узнать об индексировании и других аспектах поисковых

систем из книги Croft, Metzler, Strohman «Search Engines: Information Retrieval in Practice».

PageRank (глава 3). Приведенные в эпиграфе слова Ларри Пейджа взяты из интервью с Беном Элгиным, опубликованным в номера журнала Businessweek от 3 мая 2004. Эссе Ванневара Буша «Возможный способ нашего мышления», как уже отмечалось, было впервые опубликовано в журнале «The Atlantic» (июль 1945). В лекциях Бишопа (см. выше) имеется элегантная демонстрация алгоритма PageRank с помощью системы водопроводных труб, моделирующих гиперссылки. Оригинальная статья с описанием архитектуры Google «The Anatomy of a Large-Scale Hypertextual Web Search Engine» написана сооснователями Google Сергеем Брином и Ларри Пейджем и представлена на конференции World Wide Web в 1998 году. В статье имеется краткое описание и анализ алгоритма PageRank. Гораздо более технический и развернутый анализ приведен в книге Langville, Meyer «Google's PageRank and Beyond», но для ее чтения необходимо знакомство с линейной алгеброй в объеме университетского курса. Книга John Battelle «The Search» начинается с доступно и интересно написанной истории индустрии поиска в веб, включая восхождение Google. Веб-спам, упоминаемый на стр. 49, обсуждается в работе Fetterly, Manasse, and Najork «Spam, Damn Spam, and Statistics: Using Statistical Analysis to Locate Spam Web Pages», опубликованной в трудах конференции WebDB 2004 года.

Криптография с открытым ключом (глава 4). В книге Simon Singh «The Code Book» великолепно и доступно описаны многие вопросы криптографии, в том числе криптография с открытым ключом. Там же приведена подробная история открытия криптографии с открытым ключом в секретной Британской лаборатории GCHQ. Практическая демонстрация аналогии открытого ключа со смешиванием красок имеется в лекциях Бишопа (см. выше).

Коды, исправляющие ошибки (глава 5). История о Хэмминге документально засвидетельствована в книге Thomas M. Thompson «From Error-Correcting Codes through Sphere Packings to Simple Groups». Цитата Хэмминга, приведенная на стр. 73, также имеется в этой книге и взята из интервью, которое Хэмминг дал Томпсону в 1977 году. Математикам восхитительная книга Томпсона, безусловно, понравится, но для ее чтения необходимо хорошее знакомство с математикой. В книге Дьюдени (см. выше) также есть две интересные главы о теории кодирования. Слова о Шенноне на стр. 91 взяты из его краткой биографии, написанной Н. Дж. А. Слоуном (N. J. A. Sloane)

и А. Д. Вайнером (A. D. Wyner) и опубликованной в книге «Claude Shannon: Collected Papers» под редакцией Слоуна и Вайнера (1993).

Распознавание образов (глава 6). В лекциях Бишопа (см. выше) имеется интересный материал, удачно дополняющий эту главу. Географические данные о распределении приверженцев политических партий взяты из проекта Fundrace, организованного газетой Huffington Post. Данные о рукописных цифрах взяты из набора данных, предоставленного Янном ЛеКуном (Yann LeCun) из Института Куранта при Нью-Йоркском университете и его сотрудниками. Набор данных MNIST подробно обсуждается в работе LeCun et al. «Gradient-Based Learning Applied to Document Recognition» 1998 года. Результаты по веб-спаму взяты из работы Ntoulas et al. «Detecting Spam Web Pages through Content Analysis», опубликованной в трудах конференции World Wide Web 2006 года. Данные о лицах были собраны в 1990-х годах ведущим специалистом по распознаванию образов Томом Митчеллом из Университета Карнеги-Меллона. Митчелл использовал эту базу данных на своих курсах в Университете Карнеги-Меллона и описал в оказавшей большое влияние книге «Machine Learning». На сопроводительном сайте этой книги Митчелл выложил программу для обучения нейронной сети и применения ее для классификации лиц. Результаты для задачи о солнечных очках были получены с помощью немного модифицированных версий этой программы. Даниэль Кревье (Daniel Crevier) приводит интересный отчет о Дартмутской конференции в книге «AI: The Tumultuous History of the Search for Artificial Intelligence». Выдержка из заявки на финансирование конференции (стр. 118) процитирована по книге Pamela McCorduck «Machines Who Think», вышедшей в 1979 году.

Сжатие (глава 7). Рассказ о Фано, Шенноне и открытии кодов Хаффмана взят из интервью, которое Фано дал Артуру Норбергу в 1989 году. Это интервью имеется в архиве устной истории Института Чарльза Бэббиджа. Лично мне больше всего нравится изложение теории сжатия данных в книге David MacKay «Information Theory, Inference and Learning Algorithms», но для ее чтения необходимо знание математики в объеме университетского курса. В книге Дьюдени (см. выше) эта тема освещается на более доступном уровне и гораздо короче.

Базы данных (глава 8). Чего-чего, а введений в базы данных для начинающих хватает, но обычно в них объясняется, как пользоваться базами данных, а не как они работают – а именно такую цель я ставил в этой главе. Даже университетские учебники, как правило, уделяют

основное внимание использованию баз данных. Исключение составляет вторая часть книги Garcia-Molina, Ullman and Widom «Database Systems», где имеется достаточно подробностей, относящихся к теме этой главы.

Цифровые подписи (глава 9). Книга Gail Grant «Understanding Digital Signatures» содержит массу информации по цифровым подписям и вполне доступна читателям, не имеющим подготовки в информатике.

Вычислимость (глава 10). Эпиграф к этой главе взят из лекции, прочитанной Ричардом Фейнманом в Калифорнийском технологическом институте 29 декабря 1959 года. Лекция называлась «There's Plenty of Room at the Bottom» (Там внизу много места) и позже была опубликована в журнале «Engineering & Science», издаваемом Калтехом (февраль 1960). Нетрадиционное, но очень интересное изложение вопросов вычислимости и неразрешимости в форме художественного произведения – роман Christos Papadimitriou «Turing (A Novel about Computation)».

Послесловие (глава 11). Лекция Стивена Хокинга «Будущее Вселенной» была прочитана в качестве Дарвиновской лекции 1991 года в Кэмбриджском университете, а затем перепечатана в книге Хокинга «Черные дыры и молодые вселенные». Телевизионный цикл лекций А. Дж. П. Тейлора назывался «Как начинаются войны» и также был опубликован в виде книге в 1977 году.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге«Планета Альянс»
наложенным платежом, выслав открытку или письмо по почтовому
адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги; фамилию, имя и отчество
получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(499) 782-38-89**

Электронный адрес: **books@alians-kniga.ru**.

Джон Маккормик

Девять алгоритмов, которые изменили мир

Остроумные идеи, лежащие в основе современных компьютеров

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 14,45.
Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф