ВЛАДСТОН ФЕРРЕЙРА ФИЛО

ТЕОРЕТИЧЕСКИЙ МИНИМУМ ПО COMPUTER SCIENCE

ВСЕ, ЧТО НУЖНО ПРОГРАММИСТУ И РАЗРАБОТЧИКУ

Wladston Ferreira Filho

COMPUTER SCIENCE DISTILLED

LEARN THE ART OF SOLVING COMPUTATIONAL PROBLEMS

code energy

Владстон Феррейра Фило

ТЕОРЕТИЧЕСКИЙ МИНИМУМ ПО COMPUTER SCIENCE

ВСЕ, ЧТО НУЖНО ПРОГРАММИСТУ И РАЗРАБОТЧИКУ

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2018

ББК 32.973.23-018
УДК 004.3
Ф54

Феррейра Фило Владстон

Ф54 Теоретический минимум по Computer Science. Все, что нужно программисту и разработчику. — СПб.: Питер, 2018. — 224 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0587-8

Хватит тратить время на скучные академические фолианты! Изучение Computer Science может быть веселым и увлекательным занятием.

Владстон Феррейра Фило знакомит нас с вычислительным мышлением, позволяющим решать любые сложные задачи. Научиться писать код просто — пара недель на курсах, и вы «программист», но чтобы стать профи, который будет востребован всегда и везде, нужны фундаментальные знания. Здесь вы найдете только самую важную информацию, которая необходима каждому разработчику и программисту каждый день.

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

ББК 32.973.23-018
УДК 004.3

ISBN 978-0997316001 англ.
ISBN 978-5-4461-0587-8

Оглавление

Друзья — это семья, которую мы сами себе выбираем.
Я посвящаю книгу моим друзьям Ромуло, Лео,
Мото и Крису, которые постоянно меня торопили,
чтобы я ее, наконец, закончил.

Я знаю, что дважды два — четыре, и был бы рад доказать это, если б мог, — хотя должен заметить, если бы мне удалось дважды два превратить в пять, это доставило бы мне гораздо больше удовольствия.

Лорд Байрон,
из письма будущей жене Аннабелле (1813 год).
Их дочь Ада Лавлейс стала первым программистом

Предисловие

> Каждый в нашей стране должен научиться программировать, потому что это учит думать.
>
> *Стив Джобс*

Когда компьютеры начали менять мир, открывая перед людьми беспрецедентные возможности, расцвела новая наука — *computer science*. Она показала, как использовать компьютеры для решения задач. Это позволило нам использовать весь потенциал вычислительных машин. И мы достигли удивительных, просто сумасшедших результатов.

Computer science повсюду, но эта наука по-прежнему преподается как скучная теория. Многие программисты даже не изучали ее! Однако она крайне важна для эффективного программирования. Некоторые мои друзья не могут найти хорошего программиста, чтобы взять его на работу. Вычислительные мощности сегодня в изобилии, а вот людей, способных ими пользоваться, не хватает.

Рис. 1. Компьютерные задачи[1]

Эта книга — моя скромная попытка помочь миру, а также подтолкнуть *вас* к эффективному использованию компьютеров. В ней понятия computer science представлены в простой форме. Я свел научные подробности к минимуму. Хочется надеяться, что computer science произведет на вас впечатление, и ваш программный код станет лучше.

Эта книга для меня?

Если вы хотите щелкать задачи как орешки, находя эффективные решения, то эта книга для вас. От вас потребуется только чуть-чуть опыта в написании программного кода. Если вам приходилось этим заниматься и вы различаете элементарные операторы вроде `for` и `while`, то все в порядке. В противном случае вы найдете все необходимое (и даже больше) на каких-нибудь онлайновых курсах программирования[2]. Вы можете пройти такой курс всего за неделю, и притом бесплатно. Для тех же, кто уже знаком с информатикой, эта книга станет превосходным повторением пройденного и поможет укрепить знания.

---

[1] Рисунок использован с согласия http://xkcd.com.

[2] См., например, http://code.energy/coding-courses.

Но разве computer science не только для ученых?

Эта книга — для всех. Она о вычислительном мышлении. Вы научитесь превращать задачи в вычислимые системы. Вы также будете использовать вычислительное мышление в повседневных задачах. Упреждающая выборка и кэширование помогут вам упростить процесс упаковки вещей. Освоив параллелизм, вы станете эффективнее управляться на кухне. Ну и, разумеется, ваш программный код будет просто потрясающим.

Да пребудет с вами сила! 

Влад

Глава 1

ОСНОВЫ

> Информатика не более наука о компьютерах, чем астрономия — наука о телескопах. Информатика неразрывно связана с математикой.
>
> *Эдсгер Дейкстра*

Компьютерам нужно, чтобы мы разбивали задачи на посильные для них части. Тут нам понадобится немного математики. Не паникуйте, это не высшая математика — написание хорошего программного кода редко требует знания сложных уравнений. В главе 1 вы найдете набор инструментов для решения разных задач. Вы научитесь:

моделировать *идеи* в блок-схемах и псевдокоде;


отличать правильное от неправильного при помощи *логики*;


выполнять *расчеты*;


уверенно вычислять *вероятности*.

Этого достаточно, чтобы переводить мысли в вычислимые решения.

1.1. Идеи

Оказавшись перед сложной задачей, поднимитесь над ее хитросплетениями и изложите все самое важное на бумаге. Оперативная память человеческого мозга легко переполняется фактами и идеями. Многие подходы к организации работы предполагают изложение мыслей в письменной форме. Есть несколько способов это сделать. Сначала мы посмотрим, как пользоваться блок-схемами для представления процессов. Затем узнаем, как конструировать программируемые процессы на псевдокоде. Мы также попробуем смоделировать простую задачу при помощи математических формул.

Блок-схемы

Когда разработчики «Википедии» обсуждали организацию коллективной работы, они создали блок-схему дискуссии. Договариваться проще, если все инициативы перед глазами и объединены в общую картину (рис. 1.1).

Компьютерный код, как и изображенный выше процесс редактирования вики-страницы, по существу является процессом. Программисты часто пользуются блок-схемами для изображения вычислительных процессов на бумаге. Чтобы другие могли понимать ваши блок-схемы, вы должны соблюдать следующие рекомендации[1]:

- записывайте состояния и инструкции внутри прямоугольников;
- записывайте принятие решений, когда процесс может пойти различными путями, внутри ромбов;
- никогда не объединяйте инструкции с принятием решений;

[1] Адаптация схемы с сайта http://wikipedia.org.

- соединяйте стрелкой каждый последующий шаг с предыдущим;
- отмечайте начало и конец процесса.

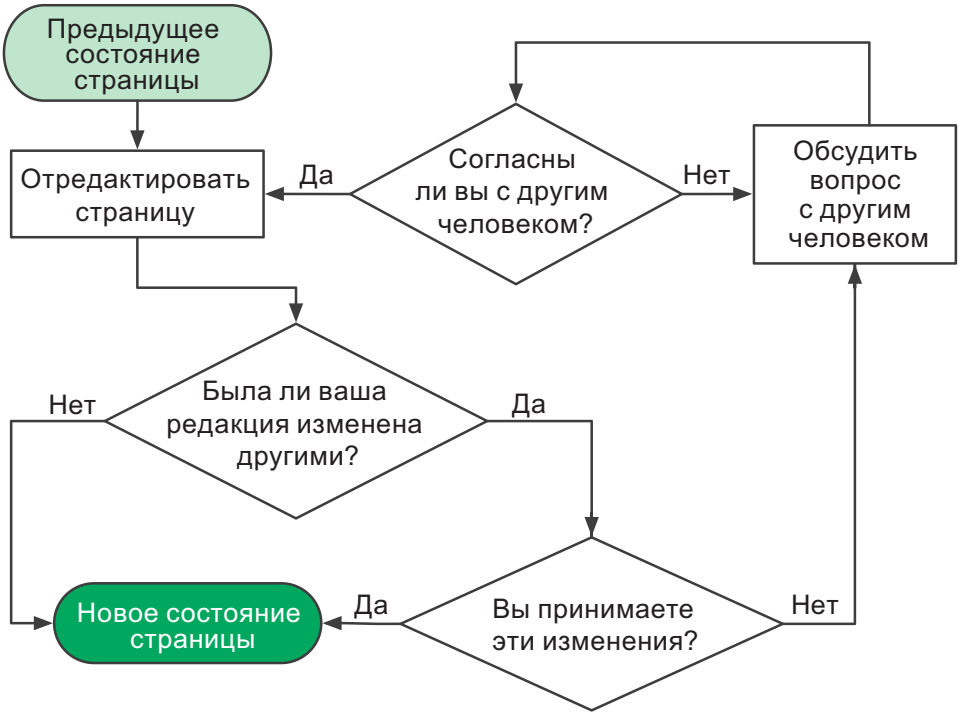


Рис. 1.1. Редакционный процесс в «Википедии»[1]

[1] См., например, http://code.energy/coding-courses.

Рассмотрим составление блок-схемы на примере задачи поиска наибольшего из трех чисел (рис. 1.2).

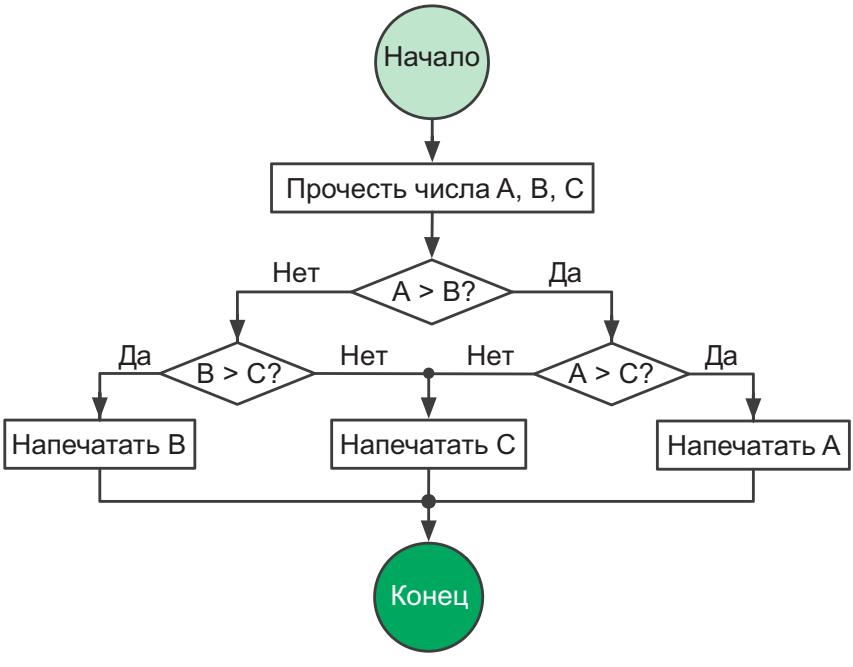


Рис. 1.2. Поиск наибольшего из трех чисел

Псевдокод

Так же, как блок-схемы, *псевдокод* выражает вычислительные процессы. Псевдокод — это код, удобный для нашего восприятия, но непонятный для машины. Следующий пример передает тот же процесс, что был изображен на рис. 1.2. Задержитесь на минуту и проверьте, как он работает с разными значениями A, B и C[1].

[1] Здесь ← означает оператор присваивания: $x \leftarrow 1$ следует читать как *«Присвоить x значение 1»*.

```
function maximum(A, B, C)
    if A > B
         if A > C
              max ← A
         else
              max ← C
    else
         if B > C
              max ← B
         else
              max ← C
    print max
```

Заметили, что этот пример полностью игнорирует синтаксические правила языков программирования? В псевдокод можно вставлять даже разговорные фразы! Когда вы пишете псевдокод, дайте своей творческой мысли течь свободно — как при составлении блок-схем (рис. 1.3 😄).

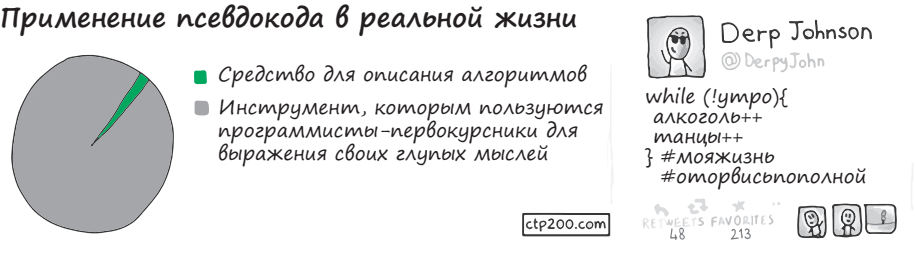


Рис. 1.3. Псевдокод в реальной жизни[1]

Математические модели

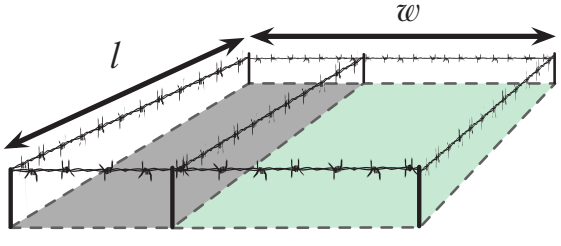
Модель — это набор идей, которые описывают задачу и ее свойства. Модель помогает рассуждать и принимать решения относительно задачи. Создание моделей настолько важно, что их преподают в школе — ведь в математике нужно иметь представление, как последовательно решать уравнения и совершать другие операции с числами и переменными.

[1] Любезно предоставлено http://ctp200.com.

Математические модели имеют большое преимущество: их можно приспособить для компьютеров при помощи четко сформулированных математических методов. Если ваша модель основана на графах, используйте теорию графов. Если она задействует уравнения, используйте алгебру. *Встаньте на плечи гигантов*, которые создали эти инструменты, и вы достигнете цели. Давайте посмотрим, как они работают, на примере типичной задачи из средней школы.

> **Загон для скота**  На ферме содержат два вида домашних животных. У вас есть 100 мотков проволоки для сооружения прямоугольного загона и перегородки внутри него, отделяющей одних животных от других. Как поставить забор, чтобы площадь пастбища была максимальной?

Начнем с того, что именно требуется определить; w и l — это размеры пастбища; $w \times l$ — его площадь. Сделать площадь максимальной означает использовать всю проволоку, потому мы устанавливаем связь между w и l, с одной стороны, и 100 мотками, с другой:



$$A = w \times l$$

$$100 = 2w + 3l$$

Подберем w и l, при которых площадь A будет максимальной.

Подставив l из второго уравнения $\left(l = \frac{100 - 20w}{3} \right)$ в первое, получаем:

$$A = \frac{100}{3} w - \frac{2}{3} w^2.$$

Да это же квадратное уравнение! Его максимум легко найти при помощи *формулы корней квадратного уравнения*, которую проходят в средней школе. Квадратные уравнения так же важны для программиста, как мультиварка — для повара. Они экономят время. Квадратные уравнения помогают быстрее решать множество задач, а это для вас самое главное. Повар знает свои инструменты, вы должны знать свои. Математическое моделирование вам просто необходимо. А еще вам потребуется логика.

1.2. Логика

Программистам приходится иметь дело с логическими задачами так часто, что у них от этого ум за разум заходит. Однако на самом деле многие из них логику не изучали и пользуются ею бессознательно. Освоив формальную логику, мы сможем осознанно использовать ее для решения задач.

Рис. 1.4. Логика программиста[1]

[1] Любезно предоставлено http://programmers.life.

Для начала мы поэкспериментируем с логическими высказываниями и операторами. Затем научимся решать задачи с таблицами истинности и увидим, как компьютеры опираются на логику.

Операторы

В математике переменные и операторы (+, ×, −, ...) используются для моделирования числовых задач. В математической логике переменные и операторы указывают на достоверность. Они выражают не числа, а истинность (true) или ложность (false). Например, достоверность выражения «Если вода в бассейне теплая, то я буду плавать» основывается на достоверности двух вещей, которые можно преобразовать в логические переменные A и B:

A : Вода в бассейне теплая.

B : Я плаваю.

Они либо истинны (true), либо ложны (false)[1]. A = `True` обозначает теплую воду в бассейне, B = `False` обозначает «Я не плаваю». Переменная B не может быть *наполовину истинной*, потому что я не способен плавать лишь отчасти. Зависимость между переменными обозначается символом →, *условным оператором*. $A \rightarrow B$ выражает идею, что A = `True` влечет за собой B = `True`:

$A \rightarrow B$: если вода в бассейне теплая, то я буду плавать.

При помощи других операторов можно выражать другие идеи. Для отрицания идеи используется знак !, *оператор отрицания*. $!A$ противоположно A:

$!A$: Вода в бассейне холодная.

$!B$: Я не плаваю.

[1] В нечеткой логике значения могут быть промежуточными, но в этой книге она рассматриваться не будет.

Противопоставление. Если дано $A \rightarrow B$, и я при этом не плаваю, что можно сказать о воде в бассейне? Теплая вода *влечет за собой* плавание, потому, если его нет, вода в бассейне не может быть теплой. Каждое условное выражение имеет *противопоставленный* ему эквивалент:

Для любых двух переменных A и B

$$A \rightarrow B \text{ тождественно } !B \rightarrow !A.$$

Еще пример: если вы не умеете писать хороший код, значит, вы не прочли эту книгу. *Противопоставлением данному суждению является такое:* если вы прочли эту книгу, значит, вы умеете писать хороший код. *Оба предложения сообщают одно и то же, но по-разному*[1].

Двусторонняя условная зависимость. Обратите внимание, что высказывание «*Если вода в бассейне теплая, то я буду плавать*» не означает: «Я буду плавать только в теплой воде». Данное высказывание ничего не говорит насчет холодных бассейнов. Другими словами, $A \rightarrow B$ не означает $B \rightarrow A$. Чтобы выразить оба условных суждения, используйте *двустороннюю условную зависимость*:

$A \leftrightarrow B$: Я буду плавать, если и только если вода в бассейне теплая.

Здесь теплая вода в бассейне равнозначна тому, что я буду плавать: знание о воде в бассейне означает знание о том, что я буду плавать, *и наоборот*. Опять же, остерегайтесь *обратной ошибки*: никогда не предполагайте, что $B \rightarrow A$ следует из $A \rightarrow B$.

AND, OR и XOR. Эти логические операторы — самые известные, поскольку они часто записываются в исходном коде в явном виде — `AND` (И), `OR` (ИЛИ) и `XOR` (исключающее ИЛИ). `AND` возвращает `True`, если все идеи истинны; `OR` возвращает `True`, если любая идея истинна; `XOR` возвращает `True`, если идеи взаимоисключающие. Представим вечеринку, где подают водку и вино:

[1] И, между прочим  , они оба *фактически* истинные.

A : Вы пили вино.

B : Вы пили водку.

A OR B : Вы пили.

A AND B : Вы пили и то и другое.

A XOR B : Вы пили, не смешивая.

Проверьте, правильно ли вы понимаете, как работают эти операторы. В табл. 1.1 перечислены все возможные комбинации двух переменных. Обратите внимание, что $A \rightarrow B$ тождественно $!A$ OR B, а A XOR B тождественно $!(A \leftrightarrow B)$.

A	B	! A	A → B	A ↔ B	A AND B	A OR B	A XOR B
✓	✓	✗	✓	✓	✓	✓	✗
✓	✗	✗	✗	✗	✗	✓	✓
✗	✓	✓	✓	✗	✗	✓	✓
✗	✗	✓	✓	✓	✗	✗	✗

Таблица 1.1. Логические операции для четырех возможных комбинаций A и B

Булева алгебра

Булева алгебра[1] позволяет упрощать логические выражения точно так же, как элементарная алгебра упрощает числовые.

Ассоциативность. Для последовательностей, состоящих только из операций AND или OR, круглые скобки не имеют значения. Так же, как последовательности только из операций сложения или умножения в элементарной алгебре, эти операции могут вычисляться в любом порядке.

[1] Названа так в честь Джорджа Буля (1815–1864). Его публикации положили начало математической логике.

$$A \text{ AND } (B \text{ AND } C) = (A \text{ AND } B) \text{ AND } C;$$

$$A \text{ OR } (B \text{ OR } C) = (A \text{ OR } B) \text{ OR } C.$$

Дистрибутивность. В элементарной алгебре мы раскрываем скобки: a × (b + c) = (a × b) + (a × c). Точно так же и в логике выполнение операции AND после OR эквивалентно выполнению операции OR над результатами операций AND и наоборот:

$$A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C);$$

$$A \text{ OR } (B \text{ AND } C) = (A \text{ OR } B) \text{ AND } (A \text{ OR } C).$$

Правило де Моргана[1]**.** Одновременно лета *и* зимы не бывает, поэтому у нас *либо не* лето, *либо не* зима. С другой стороны, оба выражения «не лето» и «не зима» истинны, *если (и только)* у нас не тот случай, когда *либо* лето, *либо* зима. Согласно этой логике, выполнение операций AND может быть сведено к операциям OR и наоборот:

$$!(A \text{ AND } B) = !A \text{ OR } !\, B;$$

$$!A \text{ AND } !B = !(A \text{ OR } B).$$

Эти правила позволяют преобразовывать логические модели, раскрывать их свойства и упрощать выражения. Давайте решим задачу.

> **Перегрев сервера**  Сервер выходит из строя из-за перегрева, когда кондиционирование воздуха выключено. Он также выходит из строя из-за перегрева, если барахлит кулер. При каких условиях сервер работает?

Моделируя эту задачу в логических переменных, можно в одном выражении сформулировать условия, когда сервер выходит из строя:

---

[1] Огастес де Морган дружил с Джорджем Булем. Кроме того, он обучал молодую Аду Лавлейс, ставшую первым программистом за век до того, как был создан первый компьютер.

A: Сервер перегревается.

B: Кондиционирование отключено.

C: Не работает кулер.

D: Сервер вышел из строя.

(*A* AND *B*) OR (*A* AND *C*) → *D*.

Используя правило дистрибутивности, выведем за скобки *A*:

A AND (*B* OR *C*) → *D*.

Сервер работает, когда !*D*. Противопоставление записывается так:

!*D* → !(*A* AND (*B* OR *C*)).

Применим правило де Моргана и раскроем скобки:

!*D* → !*A* OR !(*B* OR *C*).

Воспользуемся правилом де Моргана еще раз:

!*D* → !*A* OR (!*B* AND !*C*).

Данное выражение нам говорит, что когда сервер работает, мы имеем либо !*A* (он не перегревается), либо !*B* AND !*C* (все в порядке *и* с кондиционированием воздуха, *и* с кулером).

Таблицы истинности

Еще один способ анализа логических моделей состоит в сверке данных со всевозможными сочетаниями ее переменных. Каждой переменной в *таблице истинности* соответствует свой столбец. Строки представляют комбинации состояний переменных.

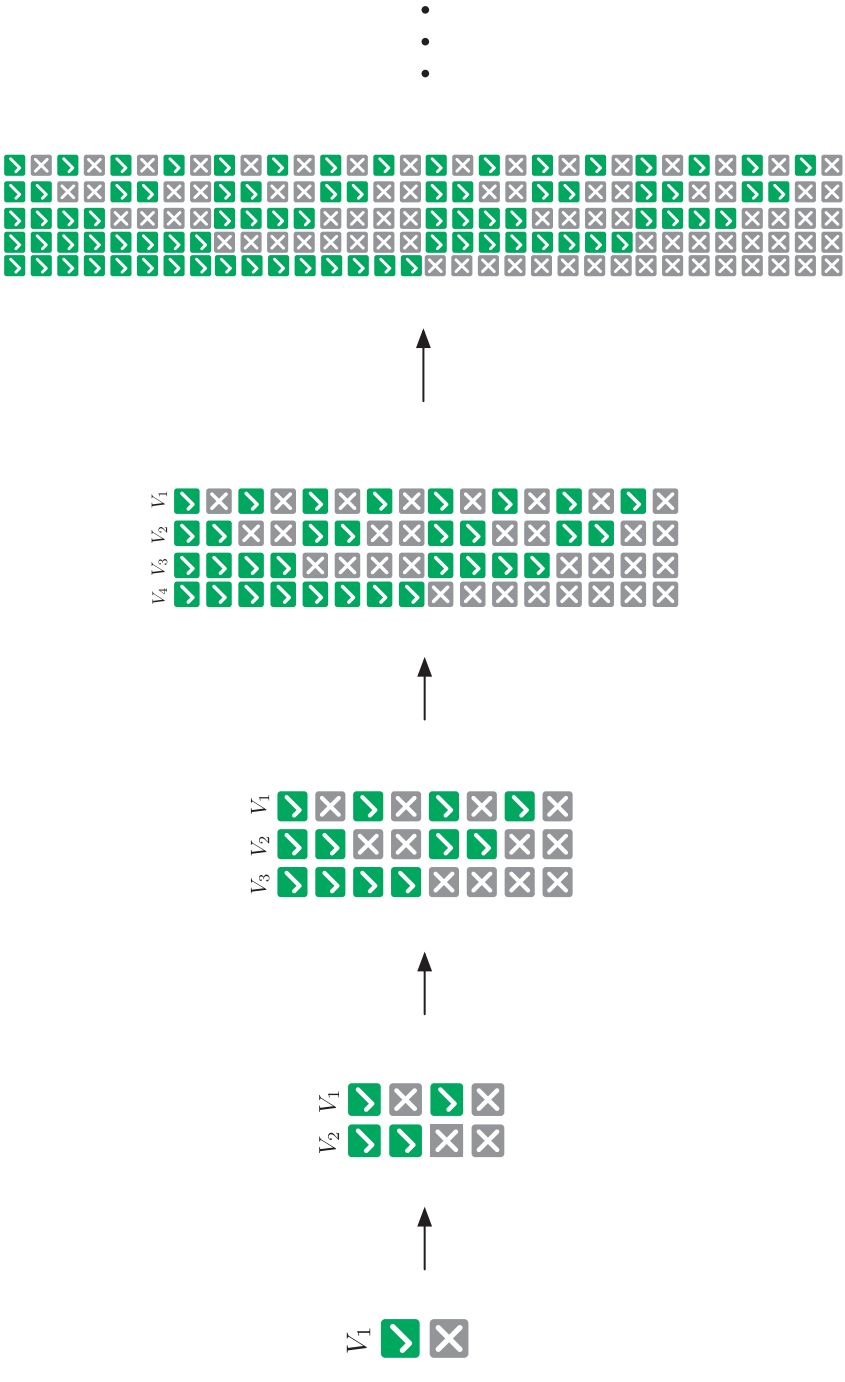


Рис. 1.5. Таблицы со всеми возможными сочетаниями от одной до пяти логических переменных

Одна переменная требует двух строк: в одной она имеет значение `True`, в другой — `False`. Чтобы добавить переменную, нужно удвоить число строк. Новой переменной задается `True` в исходных строках и `False` — в добавленных (рис. 1.5). Размер таблицы истинности увеличивается вдвое с каждым добавлением переменной, поэтому такую таблицу оправданно использовать лишь в случаях, когда переменных немного[1].

Давайте посмотрим, как можно использовать таблицу истинности для анализа задачи.

Хрупкая система Предположим, что мы должны создать систему управления базами данных с соблюдением следующих технических требований:

1) если база данных заблокирована, то мы можем сохранить данные;
2) база данных не должна блокироваться при заполненной очереди запросов на запись;
3) либо очередь запросов на запись полна, либо полон кэш;
4) если кэш полон, то база данных не может быть заблокирована.

Возможно ли это? При каких условиях станет работать такая система?

Сначала преобразуем каждое техническое требование в логическое выражение. Такую систему управления базами данных можно смоделировать при помощи четырех переменных.

A: База данных заблокирована	1: $A \rightarrow B$
B: Есть возможность сохранить данные	2: !(*A* AND *C*).
C: Очередь запросов на запись полна	3: *C* OR *D*.
D: Кэш полон	4: $D \rightarrow$!*A*.

[1] Например, таблица истинности для 30 переменных будет иметь более миллиарда строк.

Далее создадим таблицу истинности со всеми возможными сочетаниями переменных (табл. 1.2). Дополнительные столбцы добавлены для проверки соблюдения технических требований.

Таблица 1.2. Таблица истинности для проверки четырех выражений

Состояние #	A	B	C	D	1	2	3	4	Все четыре
1	✓	✓	✓	✓	✓	✗	✓	✗	✗
2	✓	✓	✓	✗	✓	✗	✓	✓	✗
3	✓	✓	✗	✓	✓	✓	✓	✗	✗
4	✓	✓	✗	✗	✓	✓	✗	✓	✗
5	✓	✗	✓	✓	✗	✗	✓	✗	✗
6	✓	✗	✓	✗	✗	✗	✓	✓	✗
7	✓	✗	✗	✓	✗	✓	✓	✗	✗
8	✓	✗	✗	✗	✗	✓	✗	✓	✗
9	✗	✓	✓	✓	✓	✓	✓	✓	✓
10	✗	✓	✓	✗	✓	✓	✓	✓	✓
11	✗	✓	✗	✓	✓	✓	✓	✓	✓
12	✗	✓	✗	✗	✓	✓	✗	✓	✗
13	✗	✗	✓	✓	✓	✓	✓	✓	✓
14	✗	✗	✓	✗	✓	✓	✓	✓	✓
15	✗	✗	✗	✓	✓	✓	✓	✓	✓
16	✗	✗	✗	✗	✓	✓	✗	✓	✗

Все технические требования удовлетворяются в состояниях с 9-го по 11-е и с 13-го по 15-е. В этих состояниях A = `False`, а значит, база данных не может быть заблокирована никогда. Обратите внимание, что кэш не заполнен лишь в состояниях 10 и 14.

Чтобы проверить, чему вы научились, попробуйте разгадать загадку «Кто держит зебру?»[1]. Это известная логическая задача, ошибочно

[1] См. http://code.energy/zebra-puzzle.

приписываемая Альберту Эйнштейну. Говорят, что только 2 % людей могут ее решить, но я сильно сомневаюсь. Используя большую таблицу истинности и правильно упрощая и объединяя логические высказывания, вы ее разгадаете, я уверен в этом.

Всегда, имея дело с ситуациями, допускающими один из двух вариантов, помните: их можно смоделировать с помощью логических переменных. Благодаря этому очень легко получать выражения, упрощать их и делать выводы.

А теперь давайте взглянем на самое впечатляющее применение логики: проектирование электронно-вычислительных машин.

Логика в вычислениях

Группы логических переменных могут представлять числа в двоичной форме[1]. Логические операции в случае с двоичными числами могут объединяться для расчетов. *Логические вентили* выполняют логические операции с электрическим током. Они используются в электрических схемах, выполняющих вычисления на сверхвысоких скоростях.

Логический вентиль получает значения через входные контакты, выполняет работу и передает результат через выходной контакт. Существуют логические вентили `AND`, `OR`, `XOR` и т. д. Значения `True` и `False` представлены электрическими сигналами с высоким и низким напряжением соответственно. Сложные логические выражения можно вычислять таким образом практически мгновенно. Например, электрическая схема на рис. 1.6 суммирует два числа.

Давайте посмотрим, как работает эта схема. Не поленитесь, проследите за ходом выполнения операций, чтобы понять, как устроена магия (рис. 1.7).

[1] `True` = 1, `False` = 0. Если вы не знаете, почему 101 — это 5 в двоичной системе счисления, загляните в приложение I.

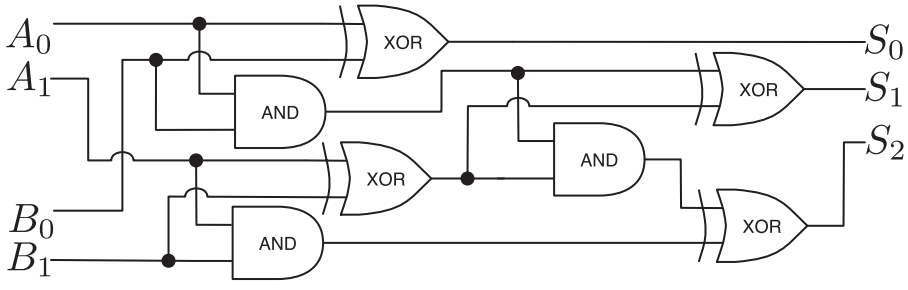


Рис. 1.6. Схема суммирования двухразрядных чисел, передаваемых парами логических переменных (A_1A_0 и B_1B_0) в трехразрядное число ($S_2S_1S_0$)

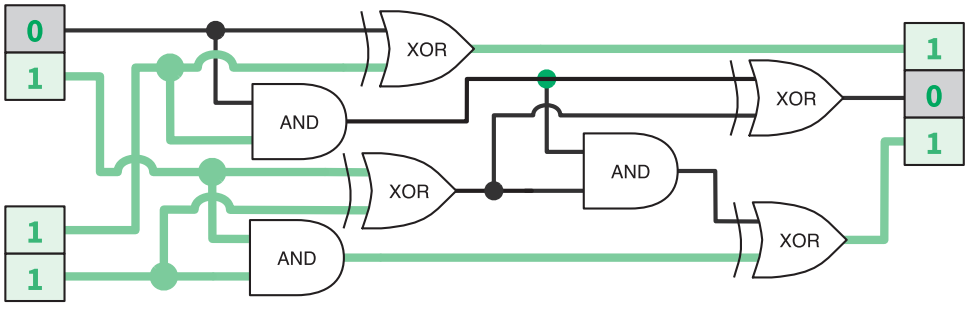


Рис. 1.7. Вычисление 2 + 3 = 5 (в двоичном формате это 10 + 11 = 101)

Чтобы воспользоваться преимуществом этого быстрого способа вычислений, мы преобразуем числовые задачи в двоичную (логическую) форму. Таблицы истинности помогают моделировать и проверять схемы. А булева алгебра — упрощать выражения и, следовательно, схемы.

Когда-то логические вентили изготавливали с использованием больших, неэффективных и дорогих электрических реле. Когда на смену реле пришли транзисторы, стало возможным массовое производство логических вентилей. Люди находили все новые и новые спо-

собы делать транзисторы меньше[1]. Принципы работы современного центрального процессора (ЦП) по-прежнему построены на булевой алгебре. Современный ЦП — это просто схема, которая состоит из миллионов микроскопических контактов и логических вентилей, управляющих электрическими потоками информации.

1.3. Комбинаторика

Важно уметь считать вещи правильно, ведь в случае с вычислительными задачами вам придется делать это много раз[2]. Математика далее будет еще более сложной, чем раньше, но не пугайтесь. Кое-кто полагает, что ему не стать хорошим программистом только потому, что, как ему кажется, математик он так себе. Если хотите знать, лично я завалил школьный экзамен по математике  , и все же я стал тем, кем хотел  . В школе дают совсем не ту математику, которая делает людей хорошими программистами.

Никто не захочет зубрить формулы и пошаговые процедуры, если он уже сдал выпускные экзамены. Если такая информация вдруг понадобится — ее легко отыскать в Интернете. Расчеты не обязательно делать от руки на бумаге. От программиста в первую очередь требуется интуиция. Познания в комбинаторике и умение решать комбинаторные задачи развивает эту интуицию. Так что давайте поработаем с несколькими инструментами по порядку: с умножением, перестановками, сочетаниями и суммами.

Правило умножения

Если некоторое событие происходит n разными способами, а другое событие — m разными способами, то число разных способов, которыми могут произойти оба события, равно $n \times m$. Вот пара примеров.

---

[1] В 2016 году был создан действующий транзистор с размером 1 нм. Для справки: атом золота имеет размер 0,15 нм.

[2] Комбинаторика и логика относятся к одной из важнейших областей информатики, которая называется дискретной математикой.

Взлом кода Предположим, что PIN-код состоит из двух цифр и латинской буквы. На то, чтобы ввести код один раз, уходит в среднем одна секунда. Какое максимальное время потребуется, чтобы подобрать правильный PIN-код?

Две цифры можно набрать 100 способами (00–99), букву — 26 способами (A–Z). Следовательно, всего существует 100 × 26 = 2600 PIN-кодов. В худшем случае, чтобы подобрать правильный, нам придется перепробовать их все. Через 2600 секунд (то есть через 43 минуты) мы его точно взломаем.

Формирование команды Допустим, 23 человека хотят вступить в вашу команду. В отношении каждого кандидата вы подбрасываете монету и принимаете его, только если выпадет «орел». Сколько всего может быть вариантов состава команды?

До начала набора есть всего один вариант состава — вы сами. Далее каждый бросок монеты удваивает число возможных вариантов. Это должно быть сделано 23 раза, таким образом, вам нужно посчитать, чему равно 2 в степени:

$$\underbrace{2\times 2\times \ldots 2}_{23\,раза} = 2^{23} = 8388608 \text{ вариантов команды.}$$

Обратите внимание, что один из этого множества вариантов — когда в команде состоите только вы.

Перестановки

Если у нас n элементов, то мы можем упорядочить их $n!$ разными способами. Факториал числа имеет взрывной характер, даже с малыми значениями n он дает огромные числа. На случай, если вы с ним не знакомы:

$$n! = n \times (n-1) \times (n-2) \ldots \times 2 \times 1.$$

Легко заметить, что $n!$ — это общее количество способов упорядочивания n элементов. Сколькими способами можно выбрать первый элемент из n? После того как он будет выбран, сколькими способами можно выбрать второй? Сколько вариантов останется для третьего? Подумайте об этом некоторое время, а потом переходите к примерам[1].

> **Коммивояжер** Ваша транспортная компания осуществляет поставки в 15 городов. Вы хотите знать, в каком порядке лучше объезжать эти города, чтобы уменьшить расход топлива. Если на вычисление длины одного маршрута требуется микросекунда, то сколько времени займет вычисление длины всех возможных маршрутов?

Любая перестановка 15 городов дает новый маршрут. Факториал — это количество различных комбинаций, так что всего существует $15! = 15 \times 14 \times ... \times 1 \approx 1{,}3$ трлн маршрутов. Число микросекунд, которые уйдут на их вычисление, примерно эквивалентно 15 дням. Будь у вас не 15, а 20 городов, вам бы понадобилось *77 тысяч лет*.

> **Совершенная мелодия** Девушка разучивает гамму из 13 нот. Она хочет, чтобы вы показали все возможные мелодии, в которых используется 6 нот. Каждая нота должна встречаться один раз на мелодию, а каждая такая мелодия должна звучать в течение одной секунды. О какой продолжительности звучания идет речь?

Мы должны подсчитать количество комбинаций по 6 нот из 13. Чтобы исключить неиспользуемые ноты, нужно остановить вычисление факториала после шестого множителя. Формально $\frac{n!}{(n-m)!}$ — это количество возможных комбинаций m из n возможных элементов. В нашем случае получится:

---

[1] По определению $0! = 1$. Мы говорим, что ноль элементов, то есть пустое множество, можно упорядочить единственным способом.

$$\frac{13!}{(13-6)!} = \frac{13\times12\times11\times10\times9\times8\times7}{7!}$$

$$= \underbrace{13\times12\times11\times10\times9\times8}_{6\ \text{множителей}}$$

$$= 1\ 235\ 520 \text{ мелодий.}$$

Чтобы их все прослушать, потребуется 343 часа, так что вам лучше убедить девушку найти идеальную мелодию каким-нибудь другим путем.

Перестановки без повторений

Факториал $n!$ дает завышенное число способов упорядочивания n элементов, если некоторые из них одинаковые. Лишние комбинации, где такие элементы просто оказываются на других позициях, не должны учитываться.

Если в последовательности из n элементов r идентичны, существуют $r!$ способов переупорядочить их. То есть $n!$ включает $r!$ таких комбинаций. Чтобы получить число уникальных комбинаций, нужно разделить $n!$ на этот излишек. Например, число различных сочетаний букв E в CODE ENERGY равняется $\frac{10}{3!}$.

> **Игры с ДНК** Биолог изучает сегмент ДНК, связанный с генетическим заболеванием. Тот состоит из 23 пар нуклеотидов, где 9 должны быть A–T, а 14 — G–C.

Ученый хочет выполнить моделирование на всех возможных сегментах ДНК, где есть такое количество пар нуклеотидов. Сколько задач ему предстоит выполнить?

Сначала вычислим все возможные комбинации этих 23 пар нуклеотидов. Затем, чтобы учесть повторяющиеся пары нуклеотидов A-T и G-C, разделим результат на 9! и на 14! и получим:

$$\frac{23!}{(9!\times 14!)} = 817\ 190 \text{ вариантов.}$$

Но задача еще не решена. Нужно учесть ориентацию пар нуклеотидов.

Следующие два примера не тождественны:

Для каждой последовательности из 23 пар нуклеотидов существует 2^{23} различных сочетаний ориентации. Потому общее количество комбинаций равно:

$$817\ 190 \times 2^{23} \approx 7 \text{ трлн.}$$

И это только для крошечной последовательности всего из 23 пар нуклеотидов, где мы знаем распределение! Наименьшая воспроизводимая ДНК, которая известна на сегодняшний день, — это ДНК крохотного *цирковируса* свиней, и в ней 1800 пар нуклеотидов. Код ДНК и жизнь в целом с технологической точки зрения по-настоящему удивительны. Просто с ума можно сойти: ДНК человека имеет около 3 млрд пар нуклеотидов, продублированных в каждой из 3 трлн клеток тела.

Комбинации

Представьте колоду из 13 игральных карт только пиковой масти ♠. Сколькими способами вы сможете раздать шесть карт своему сопернику? Мы уже видели, что $\frac{13!}{(13-6)!}$ — это количество перестановок

6 карт из 13. Поскольку порядок их следования не имеет значения, нужно разделить это число на 6!, чтобы получить

$$\frac{13!}{6!(13-6)!} = 1716 \text{ комбинаций.}$$

Бином $\binom{n}{m}$ — это количество способов, которыми можно извлечь m элементов из ряда, состоящего из n элементов, независимо от порядка их следования:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

Конструкция в левой части (запись бинома) читается как «из n по m»[1].

> **Шахматные ферзи** ♕ У вас есть пустая шахматная доска и 8 ферзей, которые допускается ставить на доске где угодно. Сколькими разными способами можно разместить фигуры?

Шахматная доска поделена на 64 клетки, 8 × 8. Число способов выбрать 8 клеток из 64 составляет $\binom{64}{8} \approx 4{,}4$ млрд[2].

Правило суммирования

Подсчет сумм последовательностей часто встречается при решении комбинаторных задач. Суммы последовательных чисел обозначаются *прописной буквой «сигма»* (Σ). Такая форма записи показывает, как выражение будет суммироваться для каждого значения i:

$$\sum_{\text{начал.}\,i}^{\text{конеч.}\,i} \text{выражение с участием } i.$$

---

[1] В литературе принято обозначение $C\binom{n}{m}$ (m — нижний индекс, n — верхний), которое читается как «сочетания m из n». — *Примеч. пер.*

[2] Профессиональная подсказка: поищите в Интернете по запросу *«из 64 по 8»*, чтобы узнать результат.

Например, суммирование первых пяти нечетных чисел записывается так:

$$\sum_{i=0}^{4}(2i+1)=1+3+5+7+9.$$

Обратите внимание: чтобы получить слагаемые 1, 3, 5, 7 и 9, вместо i последовательно используются числа от 0 до 4 включительно. Следовательно, сумма первых n натуральных чисел составляет:

$$\sum_{i=1}^{n} i=1+2+\ldots+(n-1)+n.$$

Когда гениальному математику Гауссу было 10 лет, он устал от суммирования натуральных чисел одного за другим по порядку и нашел такой ловкий прием:

$$\sum_{i=1}^{n} i=\frac{n(n+1)}{2}.$$

Догадаетесь, каким образом Гаусс это обнаружил? Объяснение приема приведено в приложении II. Давайте посмотрим, как можно его использовать для решения следующей задачи.

> **Недорогой перелет** ✈ Вы должны слетать в Нью-Йорк в любое время в течение следующих 30 дней. Цены на авиабилеты изменяются непредсказуемо в соответствии с датами отъезда и возвращения. Сколько пар дней необходимо проверить, чтобы отыскать самые дешевые билеты для полета в Нью-Йорк и обратно на ближайшие 30 дней?

Любая пара дней между сегодняшним (день 1) и последним (день 30) допустима при условии, что возвращение будет в тот же день или позже, чем отъезд. Следовательно, 30 пар начинаются с 1-го дня, 29 пар начинаются со 2-го дня, 28 — с 3-го и т. д. И есть всего одна пара, приходящаяся на последний день. Таким образом, 30 + 29 + ... + 2 + 1 — общее количество пар, которое нужно рассмотреть. Мы можем записать это как $\sum_{i=1}^{30} i$ и использовать удобную формулу Гаусса:

$$\sum_{i=1}^{30} i=\frac{30(30+1)}{2}=465 \text{ пар.}$$

Кроме того, мы можем решить эту задачу при помощи комбинаций, выбрав 2 дня из 30. Порядок не имеет значения: на более ранний день придется отъезд, на более поздний — возвращение. Таким образом, мы получим $\binom{30}{2} = 435$. Что-то не то... Дело в том, что мы должны учесть еще и случаи, когда прибытие и отъезд приходятся на одну дату. Так как дней всего 30, следовательно, $\binom{30}{2} + 30 = 465$.

1.4. Вероятность

Принципы случайности помогут вам разобраться в азартных играх, предсказании погоды или проектировании системы резервного хранения данных с низким риском отказа. Принципы эти просты, и все же большинство людей понимают их неправильно.

```
int getRandomNamber()
{
        return 4;  // выбрано честным броском кубика
                   // случайность гарантируется
}
```

Рис. 1.8. Случайное число[1]

Сейчас мы применим наши навыки решения комбинаторных задач к вычислению вероятностей. Затем мы узнаем, каким образом различные типы событий используются для решения задач. Наконец, мы увидим, почему азартные игроки проигрываются в пух и прах.

Подсчет количества возможных вариантов

Бросок кубика имеет шесть возможных результатов: 1, 2, 3, 4, 5 и 6. Шансы получить 4, следовательно, составляют $\frac{1}{6}$. А какова вероятность выпадения нечетного числа? Это может произойти в трех слу-

[1] Любезно предоставлено http://xkcd.com.

чаях (когда на кубике будет 1, 3 или 5), потому шансы составляют $\frac{3}{6}=\frac{1}{2}$. *Вероятность* того, что некое событие произойдет, выражается такой формулой:

$$P(\text{событие})=\frac{\text{Число способов наступления события}}{\text{Число возможных исходов}}.$$

Она работает, потому что каждый возможный исход одинаково вероятен. Кубик имеет ровные грани, и человек, бросающий его, нас не обманывает.

> **Еще одно формирование команды** Снова 23 человека хотят вступить в вашу команду. В отношении каждого кандидата вы подбрасываете монету и принимаете его, только если она падает «орлом». Какова вероятность, что вы никого не возьмете?

Мы уже убедились, что существует 2^{23} = 8 388 608 возможных вариантов состава команды. Вам придется рассчитывать только на себя в одном-единственном случае: если в результате подбрасывания монеты выпадут 23 «решки» подряд. Вероятность такого события равна $P(\text{никто}) = \frac{1}{8}388\ 608$. Если посмотреть на это с высоты птичьего полета, то вероятность того, что конкретный рейс коммерческой авиакомпании потерпит крушение, составляет порядка 1 из 5 млн.

Независимые (совместные) события

Если вы одновременно бросаете монету и кубик, то шанс получить «орел» и 6 равняются $\frac{1}{2}\times\frac{1}{6}=\frac{1}{12}\approx 0{,}08$, или 8 %. Когда исход одного события не влияет на исход другого, их называют *независимыми*. Вероятность получить сочетание конкретных результатов двух независимых событий равна произведению вероятностей каждого из них.

> **Резервное хранение** Вам нужно организовать хранение данных в течение года. Один диск имеет вероятность сбоя 1 на

1 млрд. Другой стоит 20 % от цены первого, но в его случае вероятность сбоя — 1 на 2000. Какой диск вам следует купить?

Если вы решите использовать три дешевых диска, то потеряете данные, только если все три выйдут из строя. Вероятность того, что это произойдет, равняется $\left(\frac{1}{2000}\right)^3 = \frac{1}{8\,000\,000\,000}$. Риск потери данных оказывается гораздо ниже, чем в случае с дорогим диском, а заплатите вы всего 60 % от его стоимости.

Несовместные события

Бросок кубика не может одновременно дать 4 и нечетное число. Вероятность получить либо 4, либо нечетное число, следовательно, равняется $\frac{1}{6}+\frac{1}{2}=\frac{2}{3}$. Когда два события не могут произойти одновременно, они *несовместные*, или взаимоисключающие. Если вам нужно подсчитать вероятность любого из нескольких несовместных событий, просто просуммируйте их индивидуальные вероятности.

Выбор подписки Ваш интернет-сервис предлагает три тарифа: бесплатный, основной и профессиональный. Вы знаете, что случайный посетитель выберет бесплатный тариф с вероятностью 70 %, основной — с вероятностью 20 % и профессиональный — с вероятностью 10 %. Каковы шансы, что человек подпишется на платный тариф?

Перечисленные события несовместны: нельзя выбрать и основной, и профессиональный тарифы *одновременно*. Вероятность, что пользователь подпишется на платный тариф, равняется 0,2 + 0,1 = 0,3.

Взаимодополняющие события

Выпавшее на кубике количество очков не может одновременно оказаться кратным трем (3, 6) и *не* делящимся на три, но оно определенно будет относиться к одной из этих категорий чисел. Вероятность

получить результат, кратный трем, равняется $\frac{2}{6}=\frac{1}{3}$, следовательно, вероятность получить число, которое не делится на три, равняется $1-\frac{1}{3}=\frac{2}{3}$. Когда два несовместных события охватывают все возможные варианты, их называют *взаимодополняющими*, или соподчиненными. Сумма вероятностей взаимодополняющих событий равна 100 %.

> **Игра «Защита башни»** Ваш замок защищен пятью башнями. Каждая имеет 20 %-ную вероятность поразить захватчика, прежде чем он достигнет ворот. Каковы шансы остановить его?

Вероятность поразить врага равна 0,2 + 0,2 + 0,2 + 0,2 + 0,2 = 1, или 100 %, верно? *Неверно!* Никогда не суммируйте вероятности независимых событий, не совершайте распространенной ошибки. Вместо этого используйте взаимодополняющие события дважды следующим образом.

- 20 %-ный шанс поразить врага — взаимодополняющий для 80 %-го шанса промахнуться. Вероятность того, что не попадут все башни, составляет $0{,}8^5 \approx 0{,}33$.
- Событие «все башни промахнулись» — взаимодополняющее для события «по крайней мере одна башня попала». Значит, вероятность остановить врага равна 1 – 0,33 = 0,67.

«Заблуждение игрока»

Если вы подбросили монету 10 раз и получили 10 «орлов», увеличилась ли от этого вероятность, что на 11-м броске выпадет «решка»? Или будет ли вероятность выигрыша в лотерею комбинации из шести последовательных чисел от 1 до 6 ниже, чем любой другой комбинации?

Не становитесь жертвой «заблуждения игрока»! Уже случившееся никак не влияет на результат независимого события. Никак. *Никог-*

да. В по-настоящему случайно разыгрываемой лотерее вероятность выпадения любого конкретного числа точно такая же, как любого другого. Нет никакой закономерности, согласно которой числа, редко выпадавшие в прошлом, должны чаще выпадать в будущем.

Более сложные вероятности

Можно было бы и дальше рассказывать о вероятности, но рамки раздела не позволяют этого. Всегда, занимаясь решением сложных задач, подыскивайте дополнительные инструменты. Вот пример.

> **И еще одно формирование команды** 23 человека хотят в вашу команду. В отношении каждого вы подбрасываете монету и принимаете его, только если выпадает «орел». Каковы шансы, что вы возьмете семь человек или меньше?

Да, это трудно посчитать. Если вы будете долго искать в Интернете, то в конечном счете придете к биномиальному распределению. Вы можете визуализировать его в Wolfram Alpha[1], набрав: `B(23,1/2) <= 7`.

Подведем итоги

В этой главе мы увидели приемы решения задач, не связанные с программированием непосредственно.

Раздел 1.1 объяснил, почему и как мы должны излагать мысли в письменной форме. Для наших задач мы создаем модели и применяем к ним концептуальные инструменты.

Раздел 1.2 познакомил с инструментами из булевой алгебры для работы с формальной логикой и таблицами истинности.

Раздел 1.3 показал важность теории вероятности и комбинаторики для решения задач разного рода. Быстрый приблизительный под-

[1] См. http://wolframalpha.com.

счет может показать вам, стоит ли браться за дальнейшие вычисления. Программисты-новички часто теряют время, анализируя слишком много сценариев.

Наконец, раздел 1.4 объяснил основные правила, позволяющие подсчитать вероятность чего-либо. Это бывает очень полезно при разработке решений, которые должны взаимодействовать с нашим дивным, но неопределенным миром.

Таким образом, мы в общих чертах обрисовали множество важных аспектов того, что ученые называют *дискретной математикой*. Еще больше интересного можно почерпнуть из приведенных ниже материалов или просто найти в «Википедии». Например, вы можете воспользоваться принципом Дирихле, чтобы доказать, что в Нью-Йорке по крайней мере у двух человек одинаковое число волос в шевелюре!

Часть из того, что мы здесь узнали, пригодится в следующей главе, где мы откроем для себя, возможно, самый важный аспект информатики.

Полезные материалы

- Дискретная математика и ее применения, 7-е издание (Discrete Mathematics and Its Applications, см. https://code.energy/rosen).
- Слайды профессора Жаннет Уинг, иллюстрирующие вычислительное мышление, см. https://code.energy/wing.

Глава 2

ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ

> Практически любой расчет можно выполнить несколькими способами. Из них следует выбирать такие, которые позволяют выполнить вычисления за наименьшее время.
>
> *Ада Лавлейс*

Сколько времени потребуется, чтобы разложить по порядку 26 перетасованных карт? А если у вас будет 52 карты, уйдет ли на эту же операцию вдвое больше времени? И насколько больше его потребуется на тысячу карточных колод? Ответ неразрывно связан с методом, который используется для сортировки карт.

Метод — это список однозначных команд, служащих для достижения цели. Метод, который всегда требует конечной серии операций, называется *алгоритмом*. Например, алгоритм сортировки карт представляет собой метод, где определены некие операции для сортировки колоды из 26 карт по масти и достоинству.

На меньшее количество операций нужно меньше вычислительной мощности. Нам нравятся быстрые решения, поэтому мы следим за числом операций в наших алгоритмах. В случае со многими алгоритмами необходимое число операций быстро растет с увеличением

объема входных данных. В нашем случае может потребоваться всего несколько операций для сортировки 26 карт, но в четыре раза больше операций для сортировки 52 карт!

Чтобы избежать непредвиденных сложностей, связанных с раздуванием задачи, нужно узнать *временну́ю сложность* алгоритма. В этой главе пойдет речь о том, как:


рассчитывать и интерпретировать *временные сложности*;


выражать их рост при помощи необычной нотации «О большое»;


избегать *экспоненциальных* алгоритмов;


убедиться, что у вашего компьютера достаточно *памяти*.

Но прежде нам предстоит узнать, как определяется временная сложность алгоритма.

Временная сложность записывается как: $T(n)$. Она показывает количество операций, которые алгоритм выполняет при обработке входящих данных объема n. Также $T(n)$ называют *стоимостью выполнения* алгоритма. Если наш алгоритм сортировки игральных карт подчиняется $T(n) = n^2$, то мы можем предсказать, насколько больше потребуется времени, чтобы отсортировать колоду двойного размера: $\frac{T(2n)}{T(n)} = 4$.

Надейтесь на лучшее, но готовьтесь к худшему

Будет ли быстрее отсортирована колода карт, которая уже почти упорядочена? Объем входящих данных — не единственная характеристика, влияющая на количество требуемых алгоритмом операций. Когда алгоритм может иметь разные значения $T(n)$ для одного n, мы обращаемся к *случаям*, или, говоря по-другому, вариантам развития событий.

- **Лучший случай** — это ситуация, когда для любых входящих данных установленного объема требуется минимальное количество операций. В сортировке такое происходит, когда входящие данные уже упорядочены.
- **Худший случай** — когда для любых входящих данных данного объема требуется максимальное количество операций. Во многих алгоритмах сортировки такое случается, когда данные на входе передаются в обратном порядке.
- **Средний случай** предполагает среднее количество операций, обычно нужных для обработки входящих данных этого объема. Для сортировки средним считается случай, когда входящие данные поступают в произвольном порядке.

Худший случай — самый важный из всех. Ориентируясь на него, вы обеспечиваете себе гарантию. Когда ничего не говорится о сценарии, ориентируйтесь на худший случай. Далее мы узнаем, как на практике анализировать события с учетом худшего варианта их развития.

Рис. 2.1. Оценка времени[1]

[1] Любезно предоставлено http://xkcd.com.

2.1. Оценка затрат времени

Временную сложность алгоритма определяют, подсчитывая основные операции, которые ему требуются для гипотетического набора входных данных объема n. Мы продемонстрируем это на примере *сортировки выбором*, алгоритма сортировки с вложенным циклом. Внешний цикл `for` обновляет текущую позицию, с которой ведется работа, внутренний цикл `for` выбирает элемент, который затем подставляется в текущую позицию[1]:

```
function selection_sort(list)
    for current ← 1 … list.length - 1
        smallest ← current
        for i ← current + 1 … list.length
            if list[i] < list[smallest]
                smallest ← i
       list.swap_items(current, smallest)
```

Давайте посмотрим, что произойдет со списком из n элементов в худшем случае. Внешний цикл совершит $n - 1$ итераций и в каждой из них выполнит две операции (одно присвоение и один обмен значениями), всего $2n - 2$ операций. Внутренний цикл сначала выполнится $n - 1$ раз, затем $n - 2$ раза, $n - 3$ раза и т. д. Мы уже знаем, как суммировать эти типы последовательностей[2]:

$$\text{количество запусков внутреннего цикла} = \overbrace{\underbrace{n-1}_{\text{первый пропуск внешнего цикла}} + \underbrace{n-2}_{\text{второй пропуск внешнего цикла}} + \ldots + 2 + 1}^{n-1 \text{ общее число запусков внешнего цикла}}$$

$$= \sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2}.$$

В худшем случае условие `if` всегда соблюдается. Это означает, что внутренний цикл выполнит одно сравнение и одно присвоение $\frac{(n^2 - n)}{2}$ раз, отсюда $n^2 - n$ операций. В целом стоимость алгоритма

[1] Чтобы разобраться в алгоритме, выполните его на бумаге с небольшим объемом входящих данных.

[2] $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ (см. раздел «Комбинаторика»).

$2n - n$ складывается из операций внешнего цикла и n^2 операций внутреннего цикла. Мы получаем временную сложность:

$$T(n) = n^2 + n - 2.$$

Что дальше? Если размер нашего списка был $n = 8$, а затем мы его удвоили, то время сортировки увеличится в 3,86 раза:

$$\frac{T(16)}{T(8)} = \frac{16^2 + 16 - 2}{8^2 + 8 - 2} \approx 3{,}86.$$

Если мы снова удвоим размер списка, нам придется умножить время на 3,9. Дальнейшие удвоения дадут коэффициенты 3,94, 3,97, 3,98. Заметили, как результат становится все ближе и ближе к 4? Это значит, что сортировка 2 млн элементов потребует в четыре раза больше времени, чем сортировка 1 млн элементов.

Понимание роста затрат

Допустим, объем входящих данных алгоритма очень велик, и мы его еще увеличиваем. Чтобы предсказать, как изменится время выполнения, нам не нужно знать все члены функции $T(n)$. Мы можем аппроксимировать $T(n)$ по ее наиболее быстрорастущему члену, который называется *доминантным членом*.

> **Учетные карточки** Вчера вы опрокинули коробку с учетными карточками, и вам пришлось потратить два часа на сортировку выбором, чтобы все исправить. Сегодня вы рассыпали 10 коробок. Сколько времени вам потребуется, чтобы расставить карточки в исходном порядке?

Мы уже убедились, что сортировка выбором подчиняется $T(n) = n^2 + n - 2$. Наиболее быстро растущим членом является n^2, поэтому мы можем записать $T(n) \approx n^2$. Приняв, что в одной коробке лежит n карточек, мы находим:

$$\frac{T(10n)}{T(n)} \approx \frac{(10n)^2}{n^2} = 100.$$

Вам потребуется приблизительно 100 × (2 часа) = 200 часов! А что, если сортировать по другому принципу? Например, есть метод под названием «сортировка пузырьком», временная сложность которого определяется формулой $T(n) = 0{,}5n^2 + 0{,}5n$. Наиболее быстро растущий член тогда даст $T(n) \approx 0{,}5n^2$, следовательно:

$$\frac{T(10n)}{T(n)} = \frac{0{,}5\times(10n)^2}{0{,}5\times n^2} = 100.$$

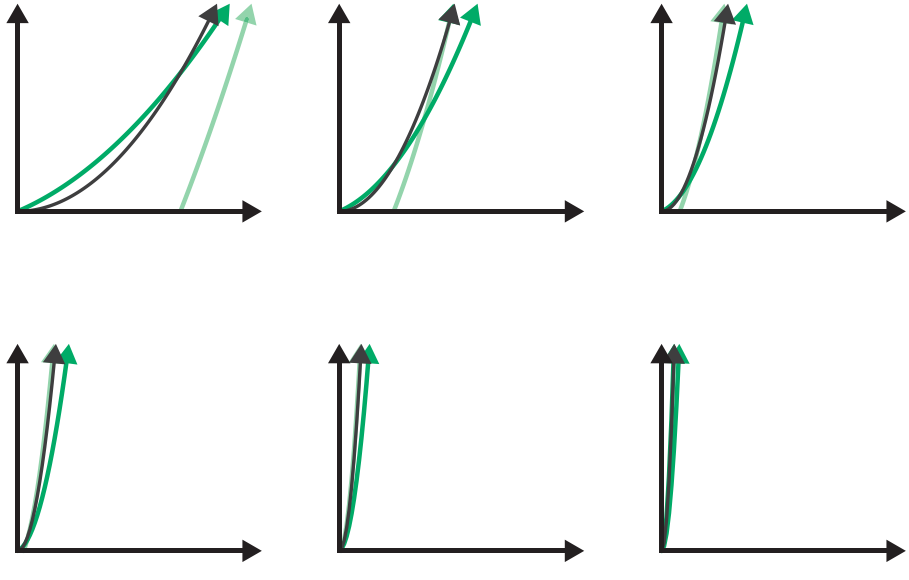


Рис. 2.2. Уменьшение масштаба n^2, $n^2 + n - 2$ и $0{,}5n^2 + 0{,}5n$ с увеличением n

Коэффициент 0,5 сам себя аннулирует! Понять мысль, что оба выражения — $n^2 - n - 2$ и $0{,}5n^2 + 0{,}5n$ — растут как n^2, не так-то просто. Почему доминантный член функции игнорирует все другие числа и оказывает наибольшее влияние на рост? Давайте попытаемся эту концепцию представить визуально.

На рис. 2.2 изображены графики двух временных сложностей, которые мы рассмотрели, рядом с графиком n^2 на разных уровнях мас-

штабирования. По мере увеличения значения n графики, судя по всему, становятся все ближе и ближе. На самом деле вместо точек в функции $T(n) = \cdot n^2 + \cdot n + \cdot$ вы можете подставить любые числа, и она по-прежнему будет расти как n^2.

Запомните, такой эффект сближения кривых работает, если наиболее быстро растущий член — одинаковый. График функции с линейным ростом (n) никогда не будет сближаться с графиком квадратичного роста (n^2), который, в свою очередь, никогда не догонит график, имеющий кубический рост (n^3).

Вот почему в случаях с очень большими объемами входных данных алгоритмы с квадратично растущей стоимостью показывают худшую производительность, чем алгоритмы с линейной стоимостью, но все же намного лучшую, чем алгоритмы с кубической стоимостью. Если вы все поняли, то следующий раздел будет для вас простым: мы всего лишь познакомимся с необычной формой записи, которую программисты используют для выражения этих идей.

2.2. Нотация «О большое»

Существует специальная форма записи, которая обозначает классы роста временных затрат: *нотация «О большое»*. Функция с членом, растущим *не быстрее* 2^n, обозначается как $O(2^n)$; функция, растущая *не быстрее* квадратичной, — как $O(n^2)$; функция с линейным *или более пологим* ростом — как $O(n)$ и т. д. Данная форма записи используется для выражения доминантного члена функций стоимости алгоритмов в худшем случае — это общепринятый способ выражения временной сложности[1].

[1] Читаются такие конструкции следующим образом: «алгоритм сортировки имеет временную сложность *о-n-квадрат*».

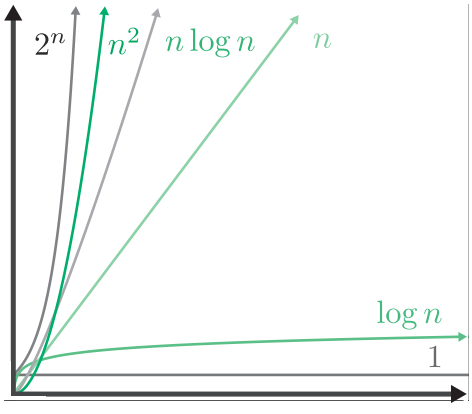


Рис. 2.3. Различные обозначения роста сложности, которые часто можно увидеть внутри *O*

Сортировка выбором и сортировка пузырьком имеют сложность $O(n^2)$, но мы вскоре встретим алгоритмы со сложностью $O(n \log n)$, которые выполняют ту же работу. В случае с $O(n^2)$ десятикратное увеличение объема входных данных привело к росту стоимости выполнения в 100 раз. Если использовать алгоритм $O(n \log n)$, то при увеличении объема входных данных в 10 раз стоимость возрастет всего в $10 \log 10 \approx 34$ раза.

Когда n равняется миллиону, n^2 составит триллион, тогда как $n \log n$ — всего лишь несколько миллионов. Работа, на которую алгоритму с квадратично растущей стоимостью потребуются годы, может быть выполнена за минуты алгоритмом со сложностью $O(n \log n)$. Вот почему при создании систем, обрабатывающих очень большие объемы данных, необходимо делать анализ временной сложности.

При разработке вычислительной системы важно заранее выявить самые частые операции. Затем нужно сравнить «О большое» разных алгоритмов, которые выполняют эти операции[1]. Кроме того, многие алгоритмы работают только с определенными структурами входных

[1] По поводу сложностей большого «О» большинства алгоритмов, которые выполняют типовые задачи, смотрите http://code.energy/bigo.

данных. Если выбрать алгоритм заранее, можно соответствующим образом структурировать данные.

Есть алгоритмы, которые всегда работают с постоянной продолжительностью, независимо от объема входных данных, — они имеют сложность $O(1)$. Например, проверяя четность/нечетность, мы смотрим, является ли последняя цифра нечетной, — и вуаля! Проблема решена. Скорость решения задачи не зависит от величины числа. С алгоритмами $O(1)$ мы познакомимся подробнее в следующих главах. Они превосходны... впрочем, давайте посмотрим, какие алгоритмы никак *нельзя* назвать «превосходными».

2.3. Экспоненциальное время

Мы говорим, что алгоритмы со сложностью $O(2^n)$ имеют *экспоненциальное время*. Из графика порядков роста (см. рис. 2.3) не похоже, что квадратичные n^2 и экспоненциальные сильно отличаются. Если уменьшить масштаб (рис. 2.4), то станет очевидно, что экспоненциальный рост явно доминирует над квадратичным.

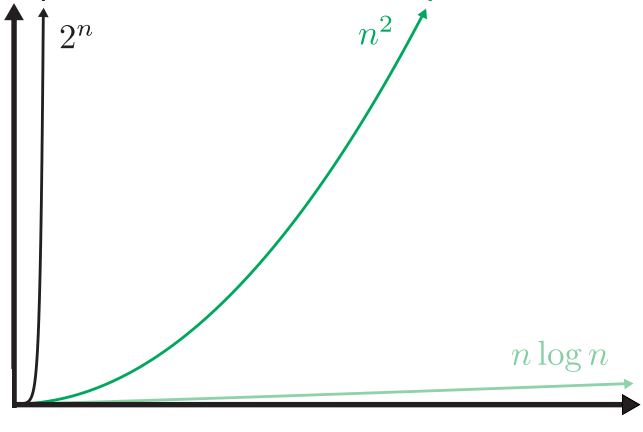


Рис. 2.4. Разные порядки роста в уменьшенном масштабе. Линейные и логарифмические кривые растут так медленно, что их уже не видно на графике

Экспоненциальное время растет *так быстро,* что мы рассматриваем эти алгоритмы как невыполнимые. Они пригодны для очень немногих типов входных данных и требуют значительной вычислительной мощности, если только объем данных не до смешного мал. Не помогут ни оптимизация каждого аспекта программного кода, ни использование суперкомпьютеров. Сокрушительное экспоненциальное время делает эти алгоритмы бесперспективными.

Чтобы наглядно представить взрывной экспоненциальный рост, уменьшим еще масштаб графика и изменим числа (рис. 2.5). Для экспоненциальной функции основание уменьшено с 2 до 1,5 и добавлен делитель 1000. Степенной же показатель увеличен с 2 до 3 и добавлен множитель 1000.

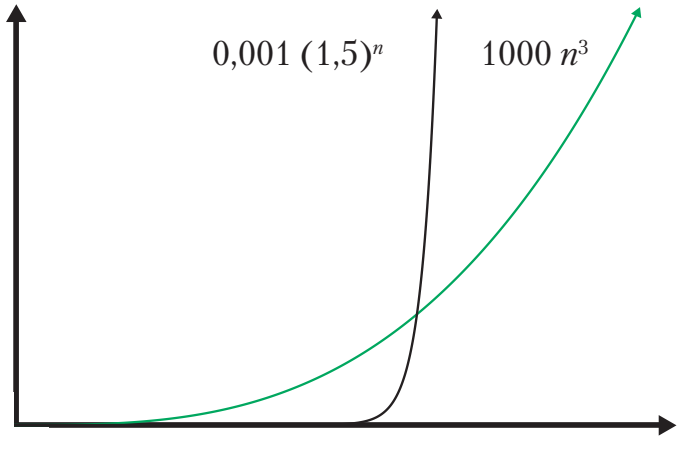


Рис. 2.5. Никакая степенная функция не превзойдет экспоненциальную. На этом графике выбран такой масштаб, что кривой $n \log n$ даже не видно из-за ее слишком медленного роста

Есть еще более бесполезные алгоритмы. Речь идет об алгоритмах с *факториальным временем*, сложность которых составляет $O(n!)$. Алгоритмы с экспоненциальным и факториальным временем ужасны, но они нужны для выполнения самых трудных вычислительных задач — знаменитых недетерминированных полиномиальных (*NP-полных*) задач. Мы увидим примеры NP-полных задач в следующей

главе. А пока запомните вот что: первый человек, который найдет неэкспоненциальный алгоритм для NP-полной задачи, получит миллион долларов [1] от Математического института Клэя, частной некоммерческой организация, расположенной в Кембридже.

Очень важно распознать класс задачи, с которой вы имеете дело. Если она является NP-полной, то пытаться найти ее оптимальное решение — это все равно что сражаться с ветряными мельницами (если только вы не решили получить тот миллион долларов).

2.4. Оценка затрат памяти

Даже если бы мы могли выполнять операции бесконечно быстро, мы все равно столкнулись бы с ограничениями. Алгоритмам во время их исполнения нужна рабочая область для хранения промежуточных результатов. Эта область занимает *память компьютера*, отнюдь не бесконечную.

Мера рабочей области хранения, в которой нуждается алгоритм, называется *пространственной сложностью*. Анализ пространственной сложности выполняется аналогично анализу временной сложности. Разница лишь в том, что мы ведем учет не вычислительных операций, а памяти компьютера. Мы наблюдаем за тем, как эволюционирует пространственная сложность с ростом объема входных данных, точно так же, как делаем это в случае временной сложности.

Например, для сортировки выбором (см. раздел «Оценка затрат времени») нужна рабочая область хранения для фиксированного набора переменных. Число переменных не зависит от объема входных данных. Поэтому мы говорим, что пространственная сложность сортировки выбором составляет $O(1)$ — независимо от объема входных

[1] Было доказано, что неэкспоненциальный алгоритм для *любой* NP-полной задачи может быть обобщен для *всех* NP-полных задач. Поскольку мы не знаем, существует ли такой алгоритм, вы также получите миллион долларов, если докажете, что NP-полная задача не может быть решена с использованием неэкспоненциальных алгоритмов.

данных она требует одного объема памяти компьютера для рабочей области хранения.

Однако многие другие алгоритмы нуждаются в такой рабочей области хранения, которая растет вместе с объемом входных данных. Иногда бывает невозможно удовлетворить потребности алгоритма в памяти. Вы не найдете подходящий алгоритм сортировки с временной сложностью $O(n \log n)$ и пространственной сложностью $O(1)$. Ограниченность памяти компьютера иногда вынуждает искать компромисс. В случае если доступно мало памяти, вам, вероятно, потребуется медленный алгоритм с временной сложностью, потому что он имеет пространственную сложность $O(1)$. В последующих главах мы увидим, как разумно выстроенная обработка данных способна улучшить пространственную сложность.

Подведем итоги

Из этой главы нам стало известно, что алгоритмы могут проявлять различный уровень «жадности» по отношению к потреблению вычислительного времени и памяти компьютера. Мы узнали, каким образом это можно диагностировать при помощи анализа временной и пространственной сложности, и научились вычислять временную сложность путем нахождения *точной* функции $T(n)$, то есть количества выполняемых алгоритмом операций.

Мы увидели, как можно выражать временную сложность с помощью нотации «О большое» (O). На протяжении всей книги мы будем использовать ее, выполняя простой анализ временной сложности алгоритмов. Во многих случаях нет необходимости вычислять $T(n)$, чтобы определить сложность алгоритма по «О большому». В следующей главе мы увидим более простые способы расчета сложности.

Еще мы увидели, что стоимость выполнения экспоненциальных алгоритмов имеет взрывной рост и делает их непригодными для входных данных большого объема. И мы узнали, как отвечать на вопросы:

- Насколько отличаются алгоритмы по числу требуемых для их выполнения операций?
- Как меняется время, необходимое алгоритму, при умножении объема входных данных на некую константу?
- Будет ли алгоритм по-прежнему выполнять приемлемое количество операций в случае, если вырастет объем входных данных?
- Если алгоритм выполняется слишком медленно для входных данных определенного объема, поможет ли его оптимизация или использование суперкомпьютера?

В следующей главе мы сосредоточимся на том, как связаны стратегии, лежащие в основе дизайна алгоритмов, с их временной сложностью.

Полезные материалы

- *Кнут Д.* Искусство программирования. Т. 1 (The Art of Computer Programming, см. https://code.energy/knuth).
- Зоопарк вычислительной сложности (The Computational Complexity Zoo, hackerdashery, см. https://code.energy/pnp).

Глава 3

СТРАТЕГИЯ

Если видишь хороший ход — ищи ход получше.

Эмануэль Ласкер

В историю входят те полководцы, что достигали выдающихся результатов с помощью надежной стратегии. Чтобы успешно решать задачи, необходимо быть хорошим стратегом. Эта глава посвящена основным стратегиям, использующимся при проектировании алгоритмов. Вы узнаете:


как справляться с повторяющимися задачами посредством *итераций*;


как изящно выполнять итерации при помощи *рекурсии*;


как использовать *полный перебор*;


как выполнять проверку неподходящих вариантов и *возвращаться на шаг назад*;


как экономить время при помощи *эвристических алгоритмов,* помогающих найти разумный выход;


как применять *принцип «Разделяй и властвуй»* к самым неподатливыми противникам;


как *динамически* идентифицировать уже решенные задачи, чтобы снова не тратить на них энергию;


как *ограничивать* рамки задачи.

Вам предстоит познакомиться с множеством инструментов, но не переживайте — мы начнем с простых задач, а затем по мере изучения новых методов постепенно будем находить все лучшие решения. Достаточно скоро вы научитесь просто и изящно справляться с вычислительными задачами.

3.1. Итерация

Итеративная стратегия состоит в использовании циклов (например, `for` и `while`) для повторения процесса до тех пор, пока не окажется соблюдено некое условие. Каждый шаг в цикле называется *итерацией*. Итерации очень полезны для пошагового просмотра входных данных и применения одних и тех же операций к каждой их порции. Вот пример.

> **Объединение списков рыб**  У вас есть списки морских и пресноводных рыб, оба упорядочены в алфавитном порядке. Как создать из них один общий список, тоже отсортированный по алфавиту?

Мы можем сравнивать в цикле верхние элементы двух списков (рис. 3.1).

Данный процесс можно записать в виде одного цикла с условием продолжения `while loop`:

```
function merge(sea, fresh)
    result ← List.new

    while not (sea.empty and fresh.empty)
        if sea.top_item > fresh.top_item
            fish ← sea.remove_top_item
       else
           fish ← fresh.remove_top_item
     result.append(fish)

return result
```

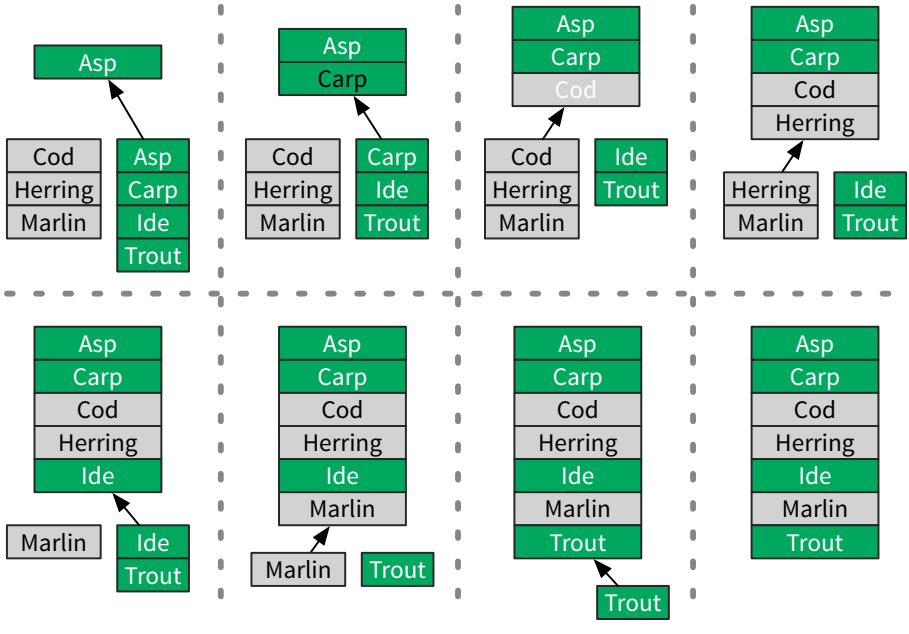


Рис. 3.1. Объединение двух отсортированных списков в третий, тоже отсортированный

Он выполняет обход всех названий рыб из входных списков, совершая фиксированное число операций для каждого элемента[1]. Следовательно, алгоритм слияния `merge` имеет сложность $O(n)$.

Вложенные циклы и степенные множества

В предыдущей главе мы увидели, как функция сортировки выбором `selection_sort` использует один цикл, вложенный в другой. Сейчас мы научимся использовать вложенный цикл для вычисления *степенного множества*. Если дана коллекция объектов S, то степенное множество S есть множество, содержащее все подмножества S[2].

[1] Объем входных данных (так называемый размер входа) — это число элементов в обоих входных списках, взятых вместе. Цикл `while` выполняет три операции для каждого из этих элементов, следовательно, $T(n) = 3n$.

[2] Если вам нужно больше узнать о множествах, см. приложение III.

> **Исследование запахов**  В парфюмерии цветочные ароматы изготавливают путем комбинирования запахов различных цветов. Если дано множество цветов *F*, то как посчитать все ароматы, которые можно изготовить из них?

Любой аромат состоит из подмножества *F*, потому его степенное множество содержит все возможные ароматы. Это степенное множество вычисляется итеративно. Для нулевого множества цветов есть всего один вариант — без запаха. В случае, когда мы берем очередной цветок, мы дублируем уже имеющиеся ароматы и добавляем его к ним (рис. 3.2).

Этот процесс можно описать при помощи циклов. Во внешнем цикле мы принимаем решение, какой цветок будем рассматривать следующим. Внутренний цикл дублирует ароматы и добавляет новый цветок к этим копиям.

```
function power_set(flowers)
    fragrances ← Set.new
    fragrances.add(Set.new)
    for each flower in flowers
        new_fragrances ← copy(fragrances)
        for each fragrance in new_fragrances
            fragrance.add(flower)
        fragrances ← fragrances + new_fragrances
    return fragrances
```

Добавление каждого нового цветка приводит к удвоению количества ароматов в множестве `fragrances`, что говорит об экспоненциальном росте ($2^{k+1} = 2 \times 2^k$). Алгоритмы, которые удваивают число операций, если объем входных данных увеличивается на один элемент, — экспоненциальные, их временная сложность — $O(2^n)$.

Генерирование степенных множеств эквивалентно генерированию таблиц истинности (см. раздел «Логика» в главе 1). Если обозначить каждый цветок логической переменной, то любой аромат легко представить в виде значений `True/False` этих переменных. В таблице истинности каждая строка будет возможной формулой аромата.

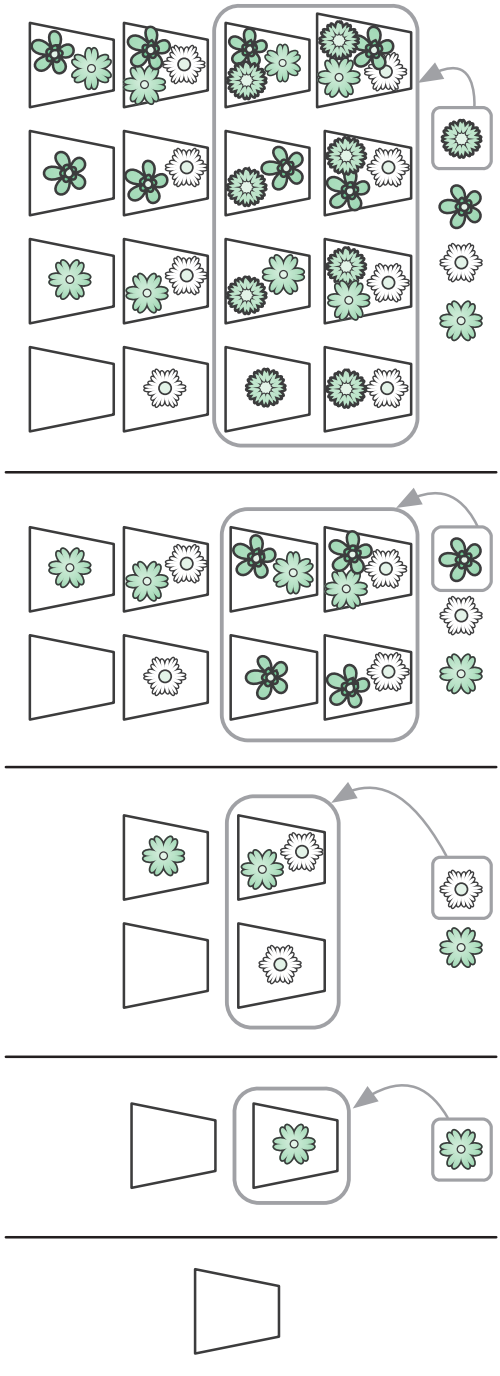


Рис. 3.2. Итеративное перечисление всех ароматов с использованием четырех цветков

3.2. Рекурсия

Мы говорим о *рекурсии*, когда функция делегирует работу своим клонам. Рекурсивный алгоритм естественным образом приходит на ум, когда нужно решить задачу, сформулированную с точки зрения *самой себя*. Например, возьмем известную последовательность Фибоначчи. Она начинается с двух единиц, и каждое последующее число является суммой двух предыдущих: 1, 1, 2, 3, 5, 8, 13, 21. Как создать функцию, возвращающую *n*-е число Фибоначчи (рис. 3.3)?

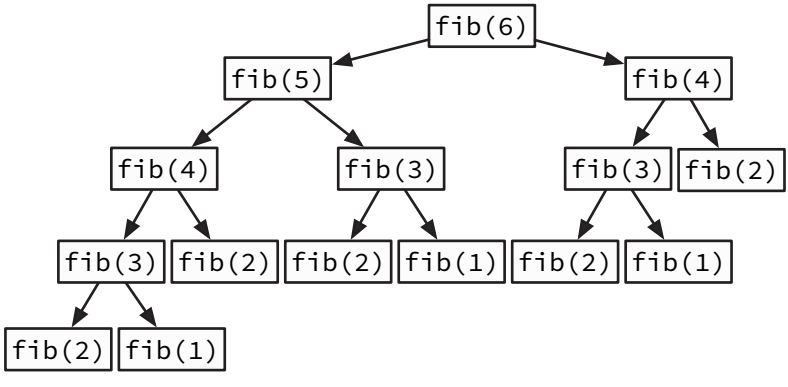


Рис. 3.3. Рекурсивное вычисление шестого числа Фибоначчи

```
function fib(n)
    if n ≤ 2
        return 1
    return fib(n - 1) + fib(n - 2)
```

При использовании рекурсии требуется творческий подход, чтобы понять, каким образом задача может быть поставлена с точки зрения самой себя. Чтобы проверить, является ли слово палиндромом[1], нужно посмотреть, изменится ли оно, если его перевернуть. Это можно сделать, проверив, одинаковы ли первая и последняя буквы

[1] Палиндромы — это слова и фразы, которые читаются одинаково в обе стороны, например «Ада», «топот», «ротатор».

слова и не является ли палиндромом заключенная между ними часть слова (рис. 3.4).

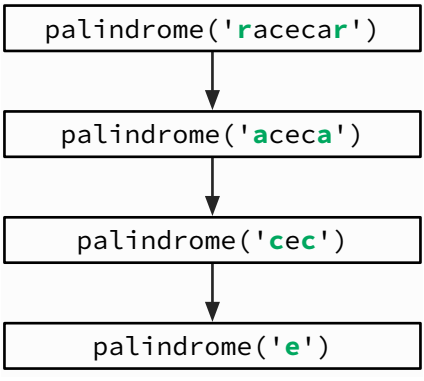


Рис. 3.4. Рекурсивная проверка, является ли слово racecar палиндромом

```
function palindrome(word)
    if word.length ≤ 1
        return True
    if word.first_char ≠ word.last_char
        return False
    w ← word.remove_first_and_last_chars
    return palindrome(w)
```

Рекурсивные алгоритмы имеют *базовые случаи*, когда объем входных данных слишком мал, чтобы его можно было продолжать сокращать. Базовые случаи для функции `fib` — числа 1 и 2; для функции `palindrome` это слова, состоящие из единственной буквы или не имеющие ни одной буквы.

Рекурсия против итераций

Рекурсивные алгоритмы обычно проще и короче итеративных. Сравните эту рекурсивную функцию с `power_set` из предыдущего раздела, которая не использует рекурсию:

```
function recursive_power_set(items)
    ps ← copy(items)
    for each e in items
       ps ← ps.remove(e)
       ps ← ps + recursive_power_set(ps)
       ps ← ps.add(e)
    return ps
```

Эта простота имеет свою цену. Рекурсивные алгоритмы при выполнении порождают многочисленные копии самих себя, создавая дополнительные вычислительные издержки. Компьютер должен отслеживать незаконченные рекурсивные вызовы и их частичные вычисления, что требует большего объема памяти. При этом дополнительные такты центрального процессора расходуются на переключение с одного рекурсивного вызова на следующий и назад.

Эту проблему можно наглядно увидеть на *деревьях рекурсивных вызовов* — диаграммах, показывающих, каким образом алгоритм порождает новые вызовы, углубляясь в вычисления. Мы уже видели деревья рекурсивных вызовов для поиска чисел Фибоначчи (см. рис. 3.3) и для проверки слов-перевертышей (см. рис. 3.4).

Если требуется максимальная производительность, то можно избежать этих дополнительных издержек, переписав рекурсивный алгоритм в чисто итеративной форме. Такая возможность есть всегда. Это компромисс: итеративный программный код обычно выполняется быстрее, но вместе с тем он более громоздкий и его труднее понять.

3.3. Полный перебор

Полный перебор, он же метод «грубой силы», предполагает перебор *всех* случаев, которые могут быть решением задачи. Эта стратегия также называется исчерпывающим поиском. Она обычно прямолинейна и незамысловата: даже в том случае, когда вариантов миллиарды, она все равно опирается исключительно на *силу*, то есть на способность компьютера проверить их все.

Рис. 3.5. Простое объяснение: полный перебор[1]

Давайте посмотрим, как ее можно использовать, чтобы решить следующую задачу.

> **Лучшая сделка** $ У вас есть список цен на золото по дням за какой-то интервал времени. В этом интервале вы хотите найти такие два дня, чтобы, купив золото, а затем продав его, вы получили бы максимально возможную прибыль.

Не всегда у вас получится сделать покупку по самой низкой цене, а продать по самой высокой: первая может случиться позже второй, а перемещаться во времени вы не умеете. Алгоритм полного перебора позволяет просмотреть *все пары дней*. По каждой паре он находит прибыль и сравнивает ее с наибольшей, найденной к этому моменту. Мы знаем, что число пар дней в интервале растет квадратично по

[1] Любезно предоставлено http://geek-and-poke.com.

мере его увеличения[1]. Еще не приступив к написанию кода, мы уже уверены, что он будет иметь $O(n^2)$.

Задача о лучшей сделке решается и с помощью других стратегий с меньшей временной сложностью — мы вскоре их рассмотрим. Но в некоторых случаях наилучшую временную сложность дает подход на основе полного перебора. Это имеет место в следующей задаче.

> **Рюкзак**  У вас есть рюкзак, вы носите в нем предметы, которыми торгуете. Его вместимость ограничена определенным весом, так что вы не можете сложить в него весь свой товар. Вы должны выбрать, что взять. Цена и вес каждого предмета известны, вам нужно посчитать, какое их сочетание дает самый высокий доход.

Степенное множество ваших предметов[2] содержит все возможные их сочетания. Алгоритм полного перебора просто проверяет эти варианты. Поскольку вы уже знаете, как вычислять степенные множества, алгоритм не должен вызвать у вас затруднений:

```
function knapsack(items, max_weight)
    best_value ← 0
    for each candidate in power_set(items)
        if total_weight(candidate) ≤ max_weight
            if sales_value(candidate) > best_value
                best_value ← sales_value(candidate)
                best_candidate ← candidate
    return best_candidate
```

Для n предметов существует $2n$ подборок. В случае каждой из них мы проверяем, не превышает ли ее общий вес вместимости рюкзака и не оказывается ли общая стоимость подборки выше, чем у лучшей, найденной к этому времени. Иными словами, для каждой подборки выполняется постоянное число операций, а значит, алгоритм имеет сложность $O(2^n)$.

---

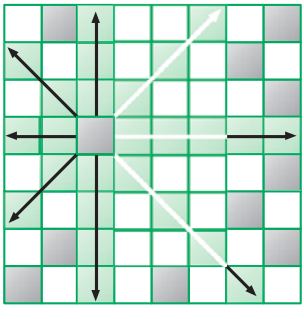
[1] В интервале n дней имеется $n(n + 1)/2$ пар дней (см. раздел «Комбинаторика» главы 1).

[2] Подробнее о степенных множествах см. в приложении III.

Однако проверять следует не каждую подборку предметов. Многие из них оставляют рюкзак полупустым, а это указывает на то, что существуют более удачные варианты[1]. Далее мы узнаем стратегии, которые помогут оптимизировать поиск решения, эффективным образом отбраковывая неподходящие варианты.

3.4. Поиск (перебор) с возвратом

Вы играете в шахматы? Фигуры перемещаются на доске 8×8 клеток и поражают фигуры соперника. Ферзь — это самая сильная фигура: она поражает клетки по горизонтали, по вертикали и по двум диагоналям. Следующая стратегия будет объяснена в контексте известной шахматной задачи.



Задача о восьми ферзях Как разместить восемь ферзей на доске так, чтобы ни один из них не оказался под ударом других?

Попробуйте найти решение вручную, и вы увидите, что оно далеко не тривиальное. Рис. 3.6 показывает один из способов расположения мирно сосуществующих ферзей.

В разделе 1.3 мы видели, что восемь ферзей можно разместить на шахматной доске *более чем 4 млрд* способами. Решение искать ответ полным перебором, проверяя все варианты, я бы назвал неосмотрительным. Предположим, что первые два ферзя помещены на доску таким образом, что представляют угрозу друг для друга. Тогда независимо от того, где окажутся следующие ферзи, решение найти не удастся. Подход на основе полного перебора не учитывает этого

[1] Задача о рюкзаке является частью класса NP-полных задач, который мы обсудили в разделе 2.3. Вне зависимости от стратегии ее решают только экспоненциальные алгоритмы.

и будет впустую тратить время, пытаясь разместить всех обреченных ферзей.

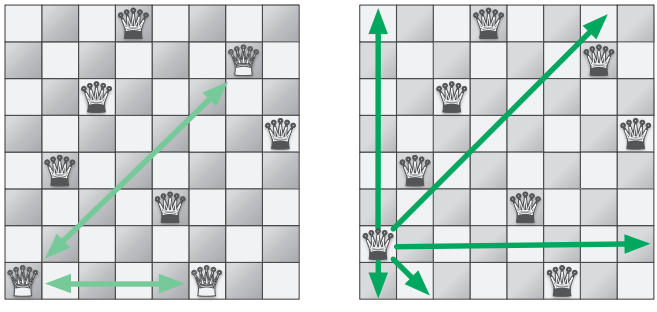


Рис. 3.6. Крайний левый ферзь может бить двух других. Если переместить его на одну клетку вверх, то он не будет никому угрожать

Более эффективный поход состоит в поиске только *приемлемых* позиций для фигур. Первого ферзя можно поместить куда угодно. Приемлемые позиции для каждого следующего будут ограничены уже размещенными фигурами: нельзя ставить ферзя на клетку, находящуюся под ударом другого ферзя. Если мы начнем руководствоваться этим правилом, мы, вероятно, получим доску, где невозможно разместить дополнительного ферзя, еще до того, как число фигур дойдет до восьми (рис. 3.7).

Это будет означать, что последнего ферзя мы разместили неправильно. Потому нам придется *отойти назад* — вернуться к предыдущей позиции и продолжить поиск. В этом заключается суть стратегии поиска с возвратом: продолжать размещать ферзей в допустимые позиции. Как только мы окажемся в тупике, мы отойдем назад, к моменту, предшествовавшему размещению последнего ферзя. Этот процесс можно оптимизировать при помощи рекурсии:

```
function queens(board)
    if board.has_8_queens
        return board
    for each position in board.unattacked_positions
        board.place_queen(position)
        solution ← queens(board)
```

```
        if solution
            return solution
        board.remove_queen(position)
    return False
```

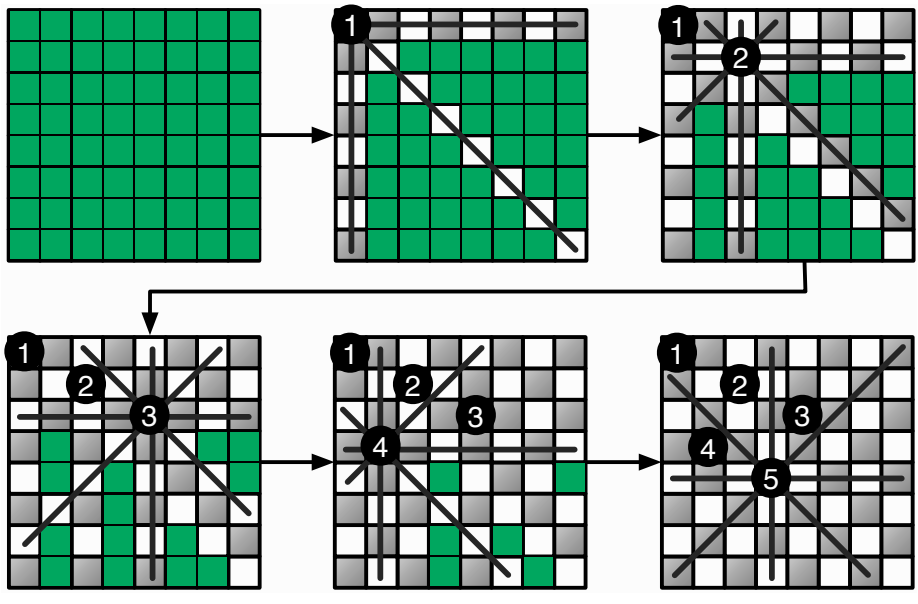


Рис. 3.7. Размещение ферзей ограничивает число приемлемых клеток для следующих фигур

Если требуемое по условию сочетание позиций на доске еще не найдено, функция обходит все приемлемые позиции для следующего ферзя. Она использует рекурсию, чтобы проверить, даст ли размещение ферзя в каждой из этих позиций решение. Как работает процесс, показано на рис. 3.8.

Поиск с возвратом лучше всего подходит для задач, где решением является последовательность вариантов, и выбор одного из них ограничивает выбор последующих. Этот подход позволяет выявлять варианты, которые не дают желаемого решения, так что вы можете отступить и попробовать что-то еще. *Ошибитесь как можно раньше, чтобы двигаться дальше.*

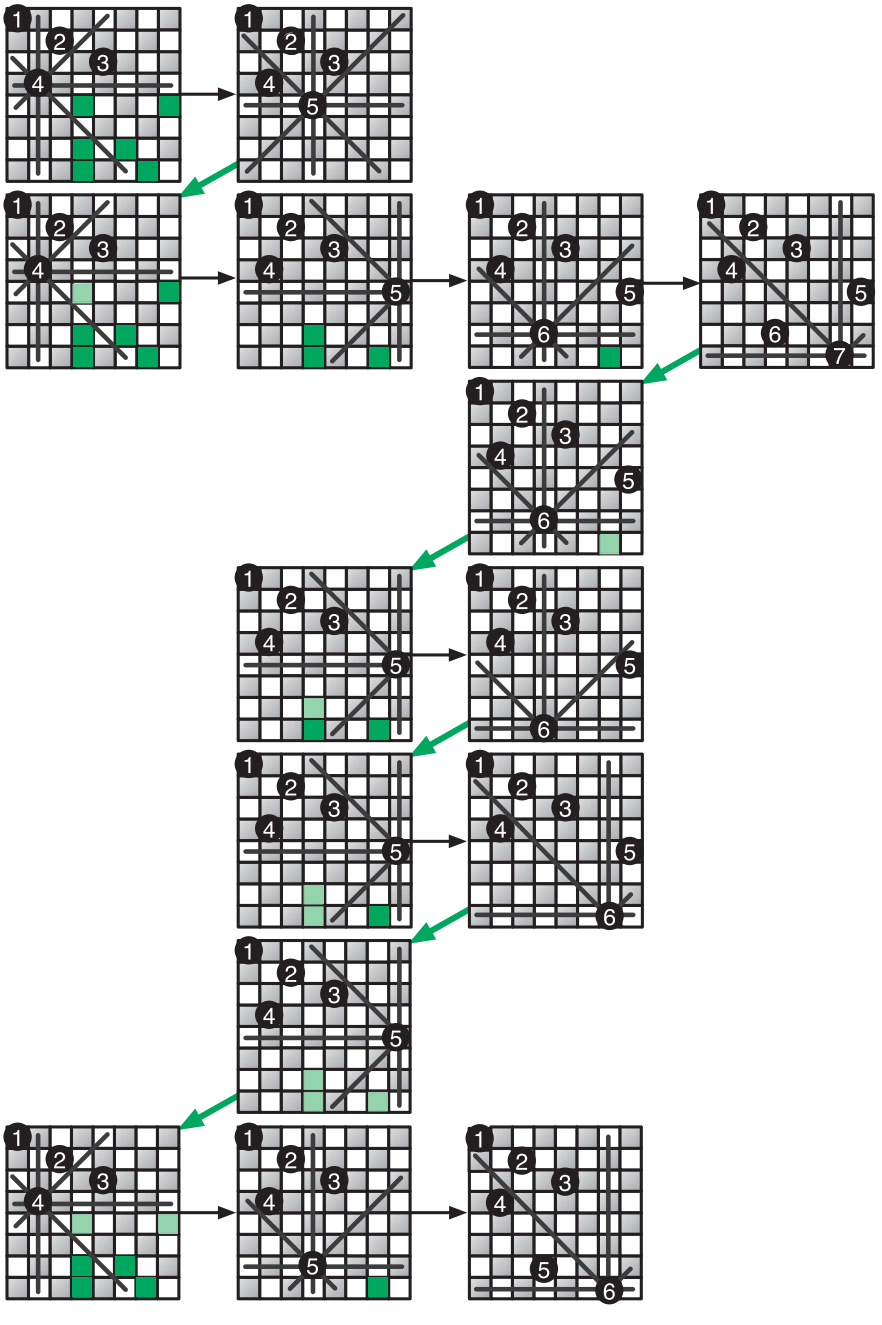


Рис. 3.8. Поиск с возвратом в «Задаче о восьми ферзях»

3.5. Эвристические алгоритмы

В обычных шахматах — 32 фигуры шести типов и 64 клетки, по которым они ходят. После каких-то четырех первых ходов число возможных дальнейших позиций достигает 288 млрд. Даже самые сильные игроки в мире не в состоянии найти идеальный ход. Они полагаются на интуицию, чтобы найти тот, который окажется *достаточно хорошим*. Мы можем делать то же самое при помощи алгоритмов. *Эвристический метод*, или просто *эвристика*, — это метод, который приводит к решению, не гарантируя, что оно — лучшее или оптимальное. Эвристические алгоритмы помогут, когда методы вроде полного перебора или поиска с возвратом оказываются слишком медленными. Существует много отличных эвристических подходов, но мы сосредоточимся на самом простом: на поиске *без* возврата.

«Жадные» алгоритмы

Очень распространенный эвристический подход к решению задач — использование так называемых *«жадных» алгоритмов*. Основная их идея состоит в том, чтобы никогда не откатываться к предыдущим вариантам. Это полная противоположность поиску с возвратом. Иными словами, на каждом шаге мы пытаемся сделать самый лучший выбор, а потом уже не подвергаем его сомнению. Давайте испытаем эту стратегию, чтобы по-новому решить задачу о рюкзаке (из раздела «Полный перебор» ).

> **Жадный грабитель и рюкзак**  Грабитель пробирается в ваш дом, чтобы украсть предметы, которые вы хотели продать. Он решает использовать ваш рюкзак, чтобы унести в нем украденное. Что он возьмет? Имейте в виду, что чем быстрее он уйдет, тем меньше вероятность, что его поймают с поличным.

В сущности, оптимальное решение здесь должно быть ровно таким же, что и в задаче о рюкзаке. Однако у грабителя нет времени для перебора всех комбинаций упаковки рюкзака, ему некогда постоян-

но откатываться назад и вынимать уже уложенные в рюкзак вещи! Жадина будет совать в рюкзак самые дорогие предметы, пока не заполнит его:

```
function greedy_knapsack(items, max_weight)
    bag_weight ← 0
    bag_items ← List.new
    for each item in sort_by_value(items)
        if max_weight ≤ bag_weight + item.weight
            bag_weight ← bag_weight + item.weight
            bag_items.append(item)
    return bag_items
```

Здесь мы не принимаем во внимание то, как наше текущее действие повлияет на будущие варианты выбора. Такой «жадный» подход позволяет отыскать подборку предметов намного быстрее, чем метод полного перебора. Однако он не дает никакой гарантии, что общая стоимость подборки окажется максимальной.

В вычислительном мышлении жадность — это не только смертный грех. Будучи добропорядочным торговцем, вы, возможно, тоже испытываете желание напихать в рюкзак всего побольше или очертя голову отправиться в поездку.

> **Снова коммивояжер**  Коммивояжер должен посетить n заданных городов и закончить маршрут в той точке, откуда он его начинал. Какой план поездки позволит минимизировать общее пройденное расстояние?

Как мы убедились в разделе «Комбинаторика» (см. главу 1), число возможных комбинаций в этой задаче демонстрирует взрывной рост и достигает неприлично больших величин, даже если городов всего несколько. Найти оптимальное решение задачи коммивояжера с тысячами городов — чрезвычайно дорого (а то и вовсе невозможно)[1].

[1] Задача коммивояжера относится к классу NP-полных задач, который мы обсудили в разделе «Экспоненциальное время» (см. главу 2). Пока не удалось найти оптимальное решение, которое было бы лучше экспоненциального алгоритма.

И тем не менее вам нужен маршрут. Вот простой «жадный» алгоритм для этой задачи:

1) посетить ближайший город, где вы еще не были;
2) повторять, пока не объедете все города.

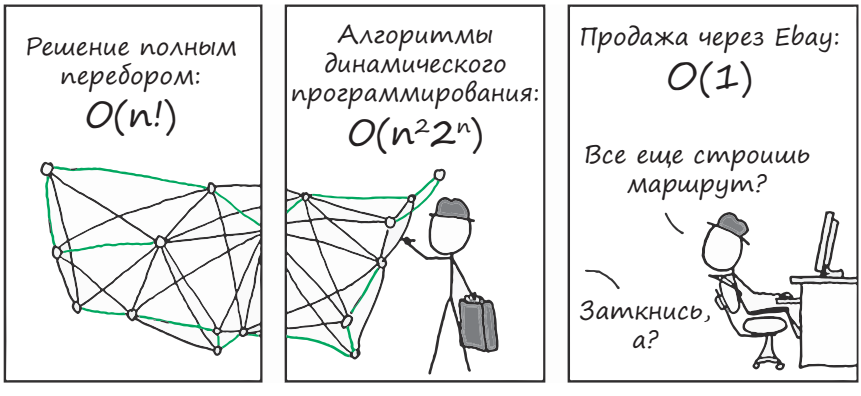


Рис. 3.9. Задача коммивояжера[1]

Можете ли вы придумать более хороший эвристический алгоритм, чем тот, что использует «жадный» подход? Специалисты по информатике вовсю ломают голову над этим вопросом.

Когда жадность побеждает силу

Выбирая эвристический алгоритм вместо классического, вы идете на компромисс. Насколько далеко от идеального решения вы можете отойти, чтобы результат все еще удовлетворял вас? Это зависит от конкретной ситуации.

Впрочем, даже если вам непременно требуется найти идеальный вариант, не стоит сбрасывать эвристику со счетов. Эвристический подход иногда приводит к самому лучшему решению. Например, вы

[1] Любезно предоставлено http://xkcd.com.

можете разработать «жадный» алгоритм, способный найти такое же решение, что и алгоритм полного перебора. Давайте посмотрим, как такое осуществляется.

> **Электрическая сеть** Поселки в удаленном районе не были электрифицированы, но вот в одном из них начали строить электростанции. Энергия пойдет от поселка к поселку по линиям электропередач. Как включить все поселки в сеть, используя минимум проводов?

Данная задача может быть решена очень просто.

1. Среди поселков, еще не подключенных к сети, выбрать тот, который находится ближе всех к электрифицированному поселку, и соединить их.
2. Повторять, пока все поселки не будут подключены.

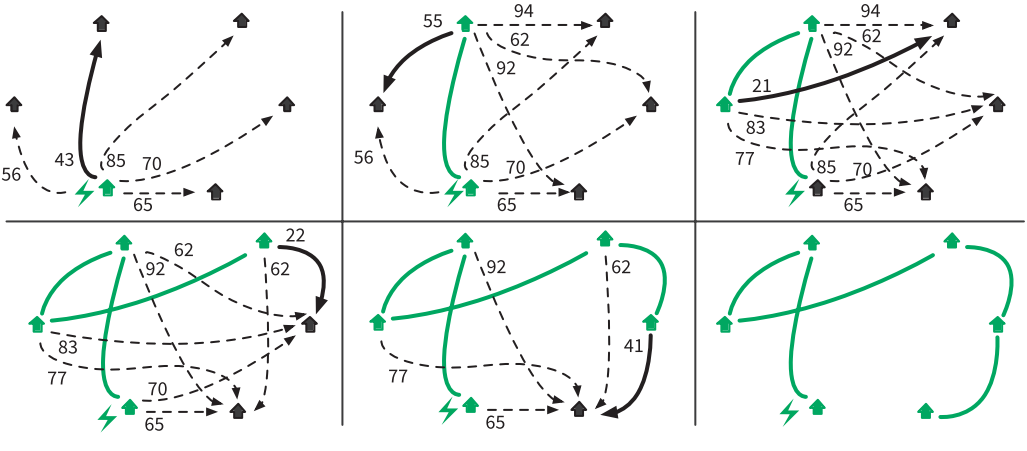


Рис. 3.10. Решение задачи об электрической сети с «жадными» вариантами выбора

На каждом шаге мы выбираем для соединения пару поселков, которая на текущий момент выглядит самой лучшей. Несмотря на то

что мы не анализируем, как этот вариант влияет на будущие возможности выбора, присоединение самого близкого поселка без электричества — всегда правильный выбор. Здесь нам повезло: структура задачи идеально подходит для решения «жадным» алгоритмом. В следующем разделе мы увидим структуры задач, для решения которых нужна стратегия великих полководцев.

3.6. Разделяй и властвуй

Когда силы врага раздроблены на небольшие группы, его проще победить. Цезарь и Наполеон управляли Европой, разделяя и завоевывая своих врагов. При помощи той же стратегии вы можете решать задачи — в особенности задачи с *оптимальной подструктурой*, то есть такие, которые легко делятся на подобные, но меньшие подзадачи. Их можно дробить снова и снова, пока подзадачи не станут простыми. Затем их решения объединяются — так вы получаете решение исходной задачи.

Разделить и отсортировать

Если у нас есть большой список, который нужно отсортировать, мы можем разделить его пополам: каждая половина становится подзадачей сортировки. Затем решения подзадач (то есть отсортированные половины списка) можно объединить в конечное решение при помощи алгоритма слияния[1]. Но как отсортировать эти две половины? Их тоже можно разбить на подзадачи, отсортировать и объединить.

Новые подзадачи будут также разбиты, отсортированы и объединены. Процесс разделения продолжаем, пока не достигнем базового случая: списка из одного элемента. Такой список уже отсортирован!

[1] Это самый первый алгоритм, который вы увидели в главе 3.

Этот изящный рекурсивный алгоритм называется *сортировкой слиянием*. Как и для последовательности Фибоначчи (см. раздел «Рекурсия»), дерево рекурсивных вызовов помогает увидеть, сколько раз функция `merge_sort` вызывает саму себя (рис. 3.11).

```
function merge_sort(list)
    if list.length = 1
        return list
    left ← list.first_half
    right ← list.last_half
    return merge(merge_sort(left),
                 merge_sort(right))
```

Теперь давайте найдем временную сложность сортировки слиянием. Для этого сначала подсчитаем операции, выполняемые на каждом отдельном шаге разбиения, а затем — общее количество шагов.

Подсчет операций. Допустим, у нас есть большой список размером n. При вызове функция `merge_sort` выполняет следующие операции:

- разбивает список на половины, что не зависит от размера списка $O(1)$;
- вызывает функцию `merge` (из раздела «Итерация» мы знаем, что `merge` имеет сложность $O(n)$;
- делает два рекурсивных вызова `merge_sort`, которые не учитываются[1].

Поскольку мы оставляем только доминирующий член и не учитываем рекурсивные вызовы, временная сложность функции составляет $O(n)$. Теперь подсчитаем временную сложность каждого шага разбиения.

Шаг разбиения 1. Функция `merge_sort` вызывается для списка из n элементов. Временная сложность этого шага составляет $O(n)$.

[1] Операции, которые выполняются рекурсивными вызовами, подсчитываются на следующем шаге разбиения.

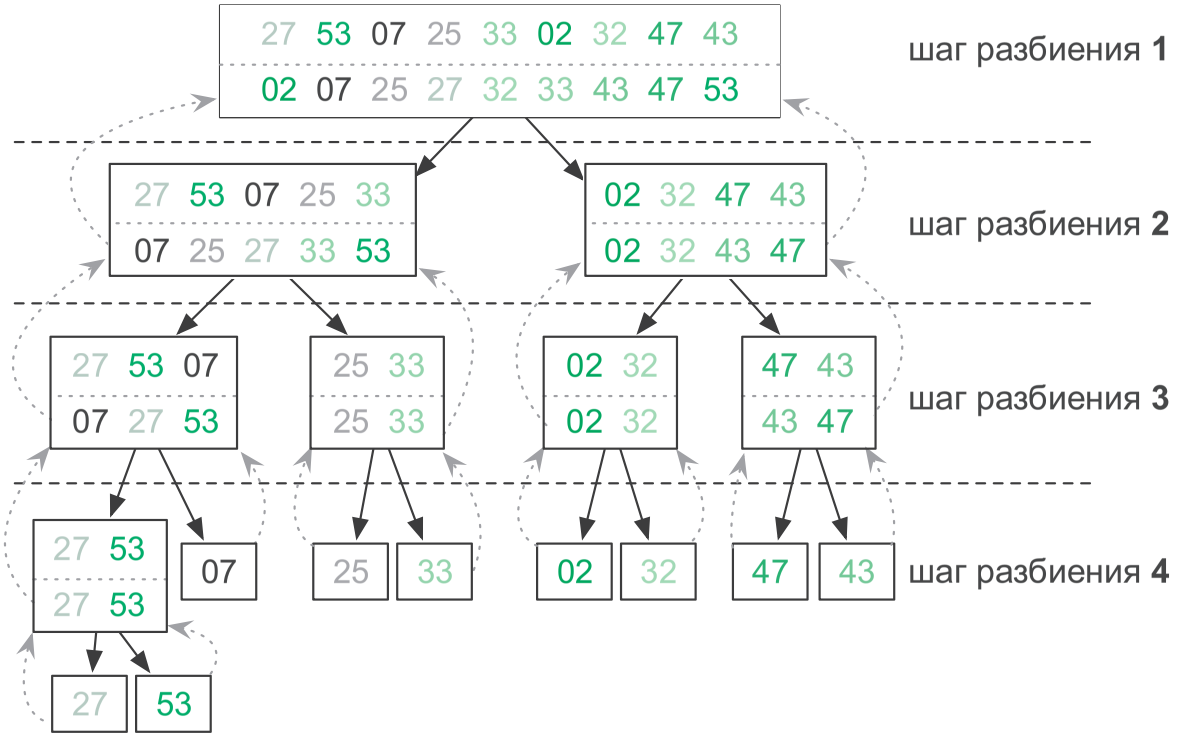


Рис. 3.11. Демонстрация сортировки слиянием. Прямоугольники показывают отдельные вызовы merge_sort, при этом входные данные находятся вверху, а выходные — внизу

Шаг разбиения 2. Функция `merge_sort` вызывается дважды, каждый раз для $\frac{n}{2}$ элементов. Мы получаем $2 \times O\left(\frac{n}{2}\right) = O(n)$.

Шаг разбиения 3. Функция `merge_sort` вызывается четыре раза, каждый раз для $\frac{n}{4}$ элементов: $4 \times O\left(\frac{n}{4}\right) = O(n)$.

⋮

Шаг разбиения *x*. Функция `merge_sort` вызывается 2^x раз, каждый для списка из $\frac{n}{2^x}$ элементов: $2^x \times O\left(\frac{n}{2^x}\right) = O(n)$.

Все шаги разбиения имеют одинаковую сложность $O(n)$. Временная сложность сортировки слиянием, следовательно, составляет $x \times O(n)$, где x — это количество шагов разбиения, необходимых для полного выполнения алгоритма[1].

Подсчет шагов. Как вычислить x? Мы знаем, что рекурсивные функции заканчивают вызывать себя, как только достигают своего базового случая. Наш базовый случай — это одноэлементный список. Мы также увидели, что шаг разбиения x работает на списках из $\frac{n}{2^x}$ элементов. Потому:

$$\frac{n}{2^x} = 1 \rightarrow 2^x = n \rightarrow x = \log_2 n.$$

Если вы не знакомы с функцией $\log_2$, то не робейте! $x = \log_2 n$ — это просто еще один способ написать $2^x = n$. Программисты любят логарифмический рост.

[1] Мы не можем проигнорировать x, потому что это не константа. Если размер списка n удвоится, то нам потребуется еще один шаг разбиения. Если n увеличится в четыре раза, тогда нужны будут два дополнительных шага разбиения.

Посмотрите, как медленно растет количество требуемых шагов разбиения[1] с увеличением общего числа сортируемых элементов (табл. 3.1).

Таблица 3.1. Количество шагов разбиения, требуемых для списков разных размеров

Размер списка (*n*)	log2 *n*	Требуемое количество шагов разбиения
10	3,32	4
100	6,64	7
1024	10,00	10
1 000 000	19,93	20
1 000 000 000	29,89	30

Временная сложность сортировки слиянием, следовательно, составляет $\log_2 n \times O(n) = O(n \log n)$. Это *колоссальное* улучшение по сравнению с сортировкой выбором $O(n^2)$. Помните разницу в производительности между линейно-логарифмическими и квадратичными алгоритмами, которые мы видели в предыдущей главе на рис. 2.4? Даже если предположить, что алгоритм $O(n^2)$ будет обрабатываться быстрым компьютером, в конечном счете он все равно окажется медленнее, чем алгоритм $O(n \log n)$ на слабой машине (табл. 3.2).

Убедитесь сами: напишите алгоритмы сортировки с линейно-логарифмической и квадратичной сложностью, а затем сравните их эффективность на примере случайных списков разного размера. Когда объемы входных данных огромны, такие улучшения часто оказываются необходимы.

[1] Любой процесс, постепенно сокращающий объем входных данных на каждом шаге, деля его на постоянный делитель, требует логарифмического количества шагов до полного сокращения входных данных.

А теперь давайте разделим и осилим задачи, в отношении которых мы раньше применяли полный перебор.

Таблица 3.2. В случае больших объемов входных данных алгоритмы $O(n \log n)$ выполняются намного быстрее алгоритмов $O(n^2)$, запущенных на компьютерах, в 1000 раз более производительных

Объем данных	Квадратичный	Логлинейный
196 (число стран в мире)	38 мс	2 с
44 000 (число аэропортов в мире)	32 минуты	12 минут
171 000 (число слов в словаре английского языка)	8 часов	51 минута
1 млн (число жителей Гавайев)	12 дней	6 часов
19 млн (число жителей штата Флорида)	11 лет	6 дней
130 млн (число книг, опубликованных за все время)	500 лет	41 день
4,7 млрд (число страниц в Интернете)	700 000 лет	5 лет

Разделить и заключить сделку

Для задачи о самой лучшей сделке (см. раздел «Полный перебор» $) подход «Разделяй и властвуй» оказывается лучше, чем решение «в лоб». Разделение списка цен пополам приводит к двум подзадачам: нужно найти лучшую сделку в первой половине и лучшую сделку во второй. После этого мы получим один из трех вариантов:

1) лучшая сделка с покупкой и продажей в первой половине;
2) лучшая сделка с покупкой и продажей во второй половине;
3) лучшая сделка с покупкой в первой половине и продажей во второй.

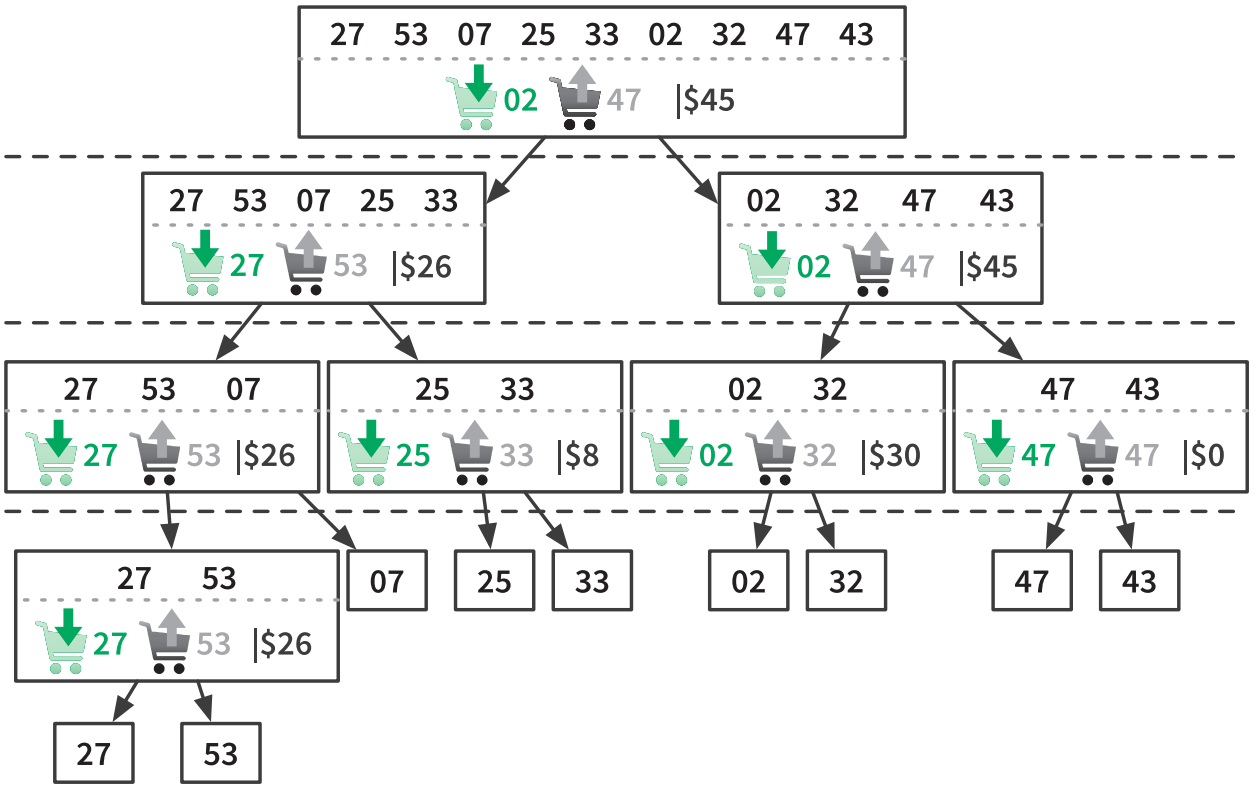


Рис. 3.12. Демонстрация выполнения функции trade.
Прямоугольники показывают отдельные вызовы trade с входными и выходными данными

Первые два случая — это решения подзадач. Третий легко находится: нужно найти самую низкую цену в первой половине списка и самую высокую во второй. Если на входе данные всего за один день, то единственным вариантом становится покупка и продажа в этот день, что приводит к нулевой прибыли.

```
function trade(prices)
    if prices.length = 1
        return 0
    former ← prices.first_half
    latter ← prices.last_half
    case3 ← max(latter) - min(former)
    return max(trade(former), trade(latter), case3)
```

Функция `trade` выполняет тривиальное сравнение, разбивает список пополам и находит максимум и минимум в его половинах. Поиск максимума или минимума в списке из n элементов требует просмотра всех n элементов, таким образом, отдельный вызов `trade` стоит $O(n)$.

Вы наверняка заметите, что дерево рекурсивных вызовов функции `trade` (рис. 3.12) очень похоже на такое же для сортировки слиянием (рис. 3.11). Оно тоже имеет $\log_2 n$ шагов разбиения, каждый стоимостью $O(n)$. Следовательно, функция `trade` тоже имеет сложность $O(n \log n)$ — это огромный шаг вперед по сравнению со сложностью $O(n^2)$ предыдущего подхода, основанного на полном переборе.

Разделить и упаковать

Задачу о рюкзаке (см. раздел «Полный перебор» ) тоже можно разделить и тем самым решить. Если вы не забыли, у нас n предметов на выбор. Мы обозначим свойство каждого из них следующим образом:

- w_i — это вес i-го предмета;
- v_i — это стоимость i-го предмета.

Индекс i предмета может быть любым числом от 1 до n. Максимальный доход для вместимости c рюкзака с уже выбранными n предметами со-

ставляет $K(n, c)$. Если рассматривается дополнительный предмет $i = n + 1$, то он либо повысит, либо не повысит максимально возможный доход, который становится равным большему из двух значений.

1. $K(n, c)$ — если дополнительный предмет не выбран.
2. $K(n, c - w_{n+1}) + v_{n+1}$ — если дополнительный предмет выбран.

Случай 1 предполагает отбраковку нового предмета, случай 2 — включение его в набор и размещение среди выбранных ранее вещей, обеспечивая для него достаточное пространство. Это значит, что мы можем определить решение для n предметов как максимум частных решений для $n - 1$ предметов:

$$K(n, c) = \max (K(n - 1, c), K(n - 1, c - w_n) + v_n).$$

Вы уже достаточно знаете и должны легко преобразовать эту рекурсивную формулу в рекурсивный алгоритм. Рисунок 3.13 иллюстрирует, как рекурсивный процесс решает задачу. На схеме выделены одинаковые варианты — они представляют идентичные подзадачи, вычисляемые более одного раза. Далее мы узнаем, как предотвратить такие повторные вычисления и повысить производительность.

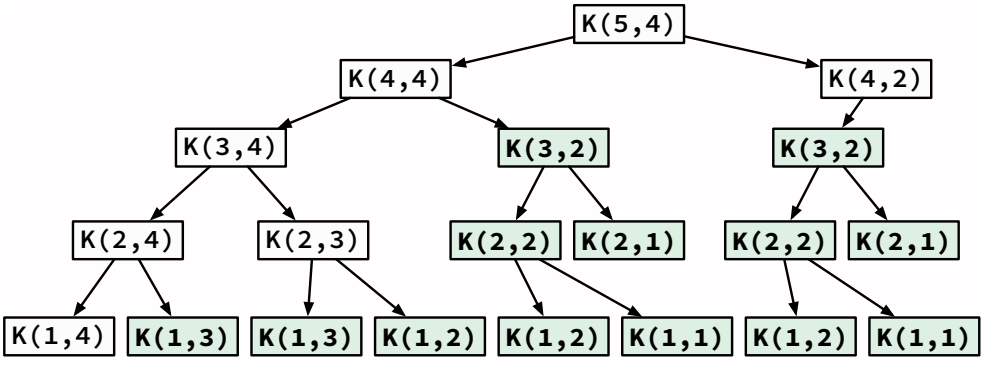


Рис. 3.13. Решение задачи о рюкзаке с 5 предметами и вместимостью рюкзака 4. Предметы под номерами 5 и 4 весят две единицы, остальные — одну единицу

3.7. Динамическое программирование

Во время решения задачи иногда приходится выполнять одни и те же вычисления многократно[1]. *Динамическое программирование* позволяет идентифицировать повторяющиеся подзадачи, чтобы можно было выполнить каждую всего один раз. Общепринятый метод, предназначенный для этого, основан на запоминании и имеет «говорящее» название.

Мемоизация Фибоначчи

Помните алгоритм вычисления чисел Фибоначчи? Его дерево рекурсивных вызовов (см. рис. 3.3) показывает, что `fib(3)` вычисляется многократно. Мы можем это исправить, сохраняя результаты по мере их вычисления и делая новые вызовы `fib` только для тех вычислений, результатов которых еще нет в памяти (рис. 3.14). Этот прием

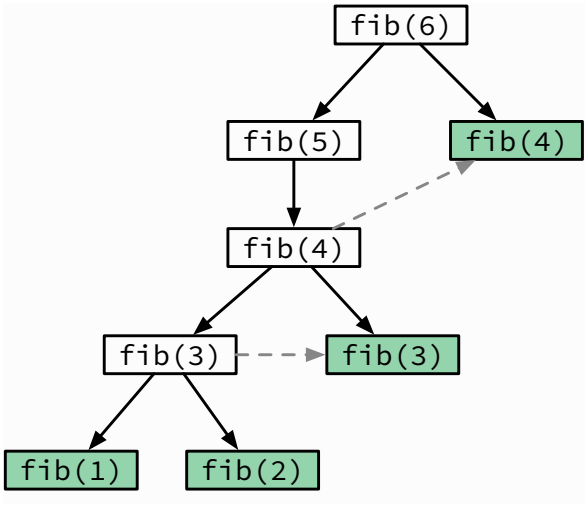


Рис. 3.14. Дерево рекурсивных вызовов для dfib. Зеленые прямоугольники обозначают вызовы, не выполняемые повторно

[1] В таком случае говорят, что задачи имеют перекрывающиеся подзадачи.

многократного использования промежуточных результатов называется *мемоизацией*. Он повышает производительность функции `fib`:

```
M ← [0 ⇨ 0; 2 ⇨ 2]
function dfib(n)
    if n not in M
        M[n] ← dfib(n-1) + dfib(n-2)
    return M[n]
```

Мемоизация предметов в рюкзаке

Очевидно, что в дереве рекурсивных вызовов для задачи о рюкзаке (см. рис. 3.13) имеются многократно повторяемые вызовы. Применение того же самого приема, который мы использовали для функции Фибоначчи, позволяет избежать этих повторных вызовов и в итоге уменьшить объем вычислений (рис. 3.15).

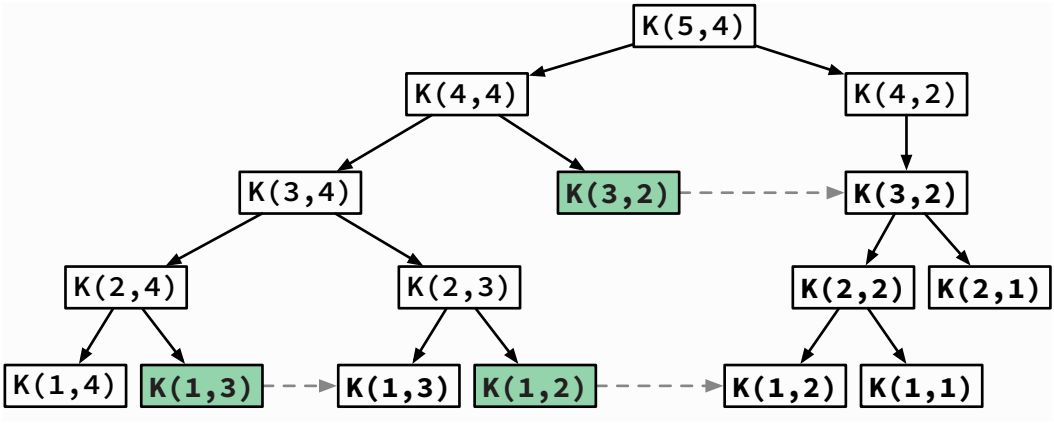


Рис. 3.15. Рекурсивное решение задачи о рюкзаке при помощи мемоизации

Динамическое программирование позволяет добиться от чрезвычайно медленного программного кода приемлемого быстродействия.

Тщательно анализируйте свои алгоритмы, чтобы убедиться, что в них нет повторных вычислений. Как мы увидим далее, иногда перекрывающиеся подзадачи могут порождать проблемы.

Лучшая сделка снизу вверх

Дерево рекурсии для функции `trade` (см. рис. 3.12) не имеет повторных вызовов, и все равно делает повторные вычисления. Он просматривает вход, чтобы найти максимальное и минимальное значения. Затем входные данные разбиваются на две части, и рекурсивные вызовы анализируют их снова, чтобы найти максимум и минимум в каждой половине[1]. Нам нужен другой принцип, для того чтобы избежать этих повторных проходов.

До сих пор мы использовали *нисходящий* подход, где объем входных данных постепенно уменьшается, пока не будут достигнуты базовые случаи. Но мы также можем пойти *снизу вверх*: сначала вычислить базовые случаи, а затем раз за разом собирать их, пока не получим общий результат. Давайте решим задачу о лучшей сделке (см. раздел «Полный перебор» $) таким способом.

Пусть $P(n)$ — это цена в n-й день, а $B(n)$ — лучший день для покупки при продаже в n-й день. Если мы продаем в первый день, то купить у нас получится только тогда же, других вариантов нет, поэтому $B(1) = 1$. Но если мы продаем во второй день, $B(2)$ может равняться 1 либо 2:

- $P(2) < P(1)$! $B(2) = 2$ (купить и продать в день 2);
- $P(2) \geq P(1)$! $B(2) = 1$ (купить в день 1, продать в день 2).

[1] Вам нужно найти самого высокого мужчину, самую высокую женщину и самого высокого человека в комнате. Будете ли вы измерять рост каждого присутствующего с целью найти самого высокого человека, а затем делать это еще и еще раз применительно к женщинам и мужчинам по отдельности?

День с самой низкой ценой *перед* днем 3, но *не в* день 3 — это $B(2)$. Потому для $B(3)$:

- $P(3) <$ цена в день $B(2) \rightarrow B(3) = 3$.
- $P(3) \geq$ цена в день $B(2) \rightarrow B(3) = B(2)$.

Обратите внимание, что день с самой низкой ценой *перед* днем 4 будет $B(3)$. Фактически для каждого n день с самой низкой ценой перед днем n — $B(n-1)$. Мы можем это использовать, чтобы выразить $B(n)$ через $B(n-1)$:

$$B(n) = \begin{cases} n & \text{если } P(n) < P(B(n-1)) \\ B(n-1) & \text{иначе.} \end{cases}$$

Когда у нас есть все пары $[n, B(n)]$ для для каждого дня n, решением является пара, которая дает самую высокую прибыль. Следующий алгоритм решает задачу, вычисляя все значения B снизу вверх:

```
function trade_dp(P)
    B[1] ← 1
    sell_day ← 1
    best_profit ← 0

    for each n from 2 to P.length
        if P[n] < P[B[n-1]]
            B[n] ← n
        else
            B[n] ← B[n-1]

        profit ← P[n] - P[B[n]]
        if profit > best_profit
            sell_day ← n
            best_profit ← profit

    return (sell_day, B[sell_day])
```

Алгоритм выполняет фиксированное число простых операций для каждого элемента входного списка, следовательно, он имеет сложность $O(n)$. Это огромный рывок в производительности по сравнению со сложностью предыдущего алгоритма $O(n \log n)$ — и совер-

шенно несравнимо со сложностью $O(n^2)$ метода полного перебора. Этот алгоритм также имеет пространственную сложность $O(n)$, поскольку вспомогательный вектор B содержит столько же элементов, что и входные данные. Из приложения IV вы узнаете, как сэкономить память за счет создания алгоритма с пространственной сложностью $O(1)$.

3.8. Ветви и границы

Многие задачи связаны с минимизацией или максимизацией целевого значения: найти кратчайший путь, получить наибольшую прибыль и т. д. Такие задачи называются *задачами оптимизации*. Когда решением является последовательность вариантов, мы часто используем стратегию *ветвей и границ*. Ее цель состоит в том, чтобы выиграть время за счет быстрого обнаружения и отбрасывания плохих вариантов. Чтобы понять, каким образом они ищутся, мы сначала должны разобраться в понятиях «верхняя граница» и «нижняя граница».

Верхние и нижние границы

Границы обозначают диапазон значения. *Верхняя граница* устанавливает предел того, каким высоким оно может быть. *Нижняя граница* — это наименьшее значение, на которое стоит надеяться; она гарантирует, что любое значение либо равно ей, либо ее превышает.

Мы порой легко находим решения, близкие к оптимальным: короткий путь — но, возможно, не самый короткий; большая прибыль — но, возможно, не максимальная. Они дают границы оптимального решения. К примеру, любой короткий маршрут из одной точки в другую никогда не будет короче расстояния между ними по прямой. Следовательно, расстояние по прямой является нижней границей самого короткого пути.

В задаче о жадном грабителе и рюкзаке (см. раздел «Эвристические алгоритмы» ) прибыль, полученная посредством `greedy_knapsack`, является нижней границей оптимальной прибыли (она может быть или не быть близкой к оптимальной прибыли). Теперь представим версию задачи о рюкзаке, в которой вместо предметов у нас сыпучие материалы, и мы можем насыпать их в рюкзак, сколько поместится. Эта версия задачи решается «жадным» способом: просто продолжайте насыпать материалы с самым высоким соотношением стоимости и веса:

```
function powdered_knapsack(items, max_weight)
    bag_weight ← 0
    bag_items ← List.new
    items ← sort_by_value_weight_ratio(items)
    for each i in items
        weight ← min(max_weight - bag_weight,
                     i.weight)
        bag_weight ← bag_weight + weight
        value ← weight * i.value_weight_ratio
        bagged_value ← bagged_value + value
        bag_items.append(item, weight)
    return bag_items, bag_value
```

Добавление ограничения неделимости предметов только уменьшит максимально возможную прибыль, потому что нам придется менять последнюю уложенную в рюкзак вещь на что-то подешевле. Это означает, что `powdered_knapsack` дает верхнюю границу оптимальной прибыли с неделимыми предметами[1].

Ветви и границы в задаче о рюкзаке

Мы уже убедились, что поиск оптимальной прибыли в задаче о рюкзаке требует дорогих вычислений $O(n^2)$. Однако мы можем быстро получить верхние и нижние границы оптимальной прибыли при помощи функций `powdered_knapsack` и `greedy_knapsack`. Давайте попробуем это сделать на примере задачи о рюкзаке (табл. 3.3).

[1] Метод удаления ограничений из задач называется *ослаблением*. Он часто используется для вычисления ограничений в задачах оптимизации.

Таблица 3.3. Верхняя и нижняя границы в задаче о рюкзаке

Предмет	Стоимость	Вес	Соотношение стоимости и веса	Макс. вместимость
A	20	5	4,00	
B	19	4	4,75	
C	16	2	8,00	10
D	14	5	2,80	
E	13	3	4,33	
F	9	2	4,50	

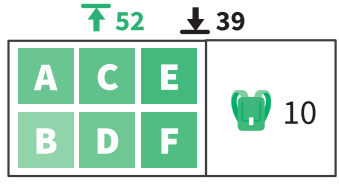
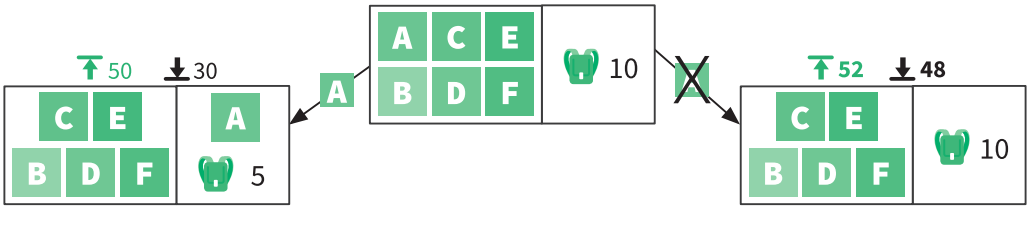
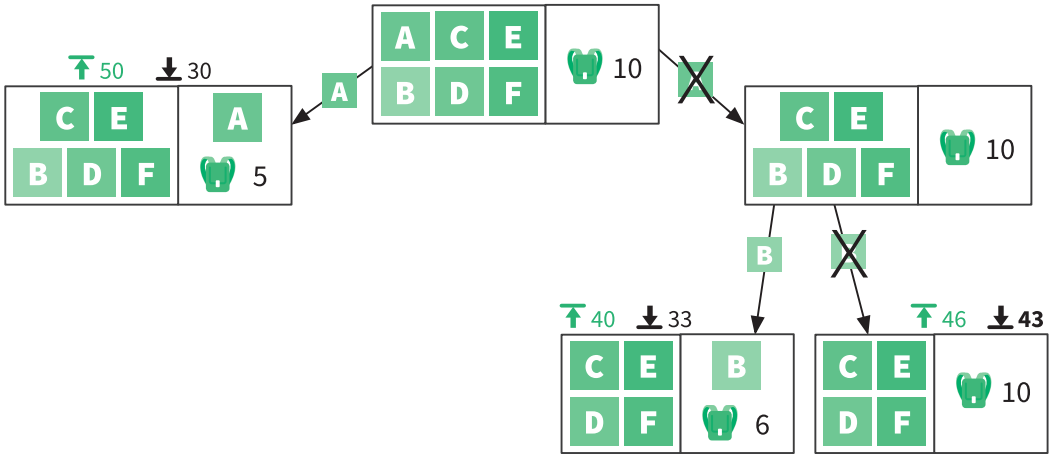


Рисунок справа иллюстрирует ситуацию перед началом заполнения рюкзака. В первом поле находятся неупакованные предметы, которые нам предстоит рассмотреть. Второе поле представляет свободное место в рюкзаке и предметы, которые уже уложены. Выполнение функции `greedy_knapsack` дает прибыль 39, а `powdered_knapsack` — 52,66. Это означает, что оптимальная прибыль находится где-то посередине. Как мы знаем из раздела «Разделяй и властвуй», эта задача с n предметами делится на две подзадачи с $n - 1$ предметами. Первая подзадача подразумевает, что предмет `A` был взят, вторая — что он *не был* взят:

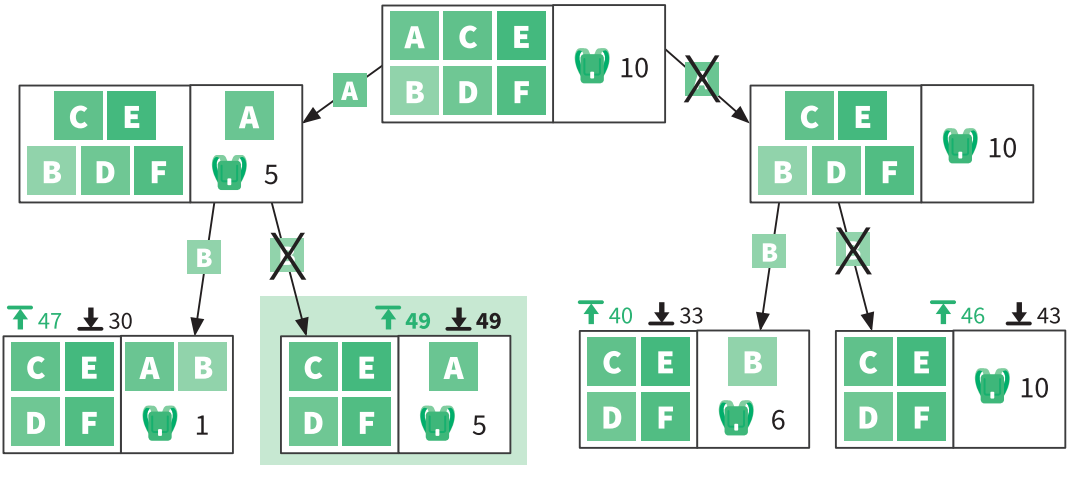


Мы вычисляем верхнюю и нижнюю границы для этих двух подзадач. Каждая имеет нижнюю границу, равную 48: теперь мы знаем, что

оптимальное решение находится между 48 и 52. Давайте рассмотрим подзадачу справа, поскольку у нее более интересные границы:



Крайняя левая подзадача имеет самую многообещающую верхнюю границу. Давайте продолжим наш анализ и выполним разбиение этой подзадачи:



Теперь мы можем сделать важные выводы. Выделенная цветом подзадача имеет нижнюю границу 49, которая равна ее верхней границе. Это означает, что оптимальная прибыль здесь должна равняться *строго* 49. Кроме того, обратите внимание, что 49 больше верхних границ во всех других ветвях, которые были проанализированы. Никакая другая ветвь не даст большую прибыль, чем 49, а значит, мы можем исключить их все из дальнейшего поиска.

Рациональное использование верхних и нижних границ позволило нам найти оптимальную прибыль, выполнив совсем немного вычислений. Мы динамически адаптировали наше пространство поиска по мере анализа возможностей.

Вот общие принципы работы метода ветвей и границ:

1) разделить задачу на подзадачи;
2) найти верхние и нижние границы каждой подзадачи;
3) сравнить границы подзадач всех ветвей;
4) выбрать самую многообещающую задачу и вернуться к шагу 1.

Если вы помните, стратегия поиска с возвратом (см. соответствующий раздел) тоже позволяет найти решение без обследования каждого возможного варианта. В случае поиска с возвратом мы исключаем пути, изучив каждый из них так далеко, как это возможно, и останавливаемся, когда нас устраивает решение. В случае же с методом ветвей и границ мы заранее определяем бесперспективные пути и не тратим впустую энергию на их обследование.

Подведем итоги

Решение задач, в сущности, представляет собой перемещение по пространству возможностей с целью найти правильный вариант. Мы узнали несколько способов, как это делается. Самый простой — полный перебор, то есть последовательная проверка каждого элемента в пространстве поиска.

Мы научились систематически делить задачи на меньшие, получая большое увеличение производительности. Многократное деление задач часто бывает сопряжено с решением проблем, вызванных одинаковыми подзадачами. В этих случаях важно использовать динамическое программирование, чтобы избежать повторных вычислений.

Мы убедились, что поиск с возвратом позволяет оптимизировать некоторые алгоритмы, основанные на полном переборе. Значения верхних и нижних границ (там, где их можно получить) позволяют ускорить поиск решения, для этого используется метод ветвей и границ. А когда стоимость вычисления оптимального решения оказывается неприемлемой, следует использовать эвристический алгоритм.

Все стратегии, с которыми мы познакомились, предназначены для работы с данными. Далее мы узнаем самые распространенные способы организации данных в памяти компьютера и как они влияют на производительность операций.

Полезные материалы

- *Клейнберг Дж., Традос Е.* Алгоритмы: разработка и применение. СПб.: Питер, 2017.
- Выбор стратегии проектирования алгоритмов (Choosing Algorithm Design Strategy, Shailendra Nigam, см. https://code.energy/nigam).
- Динамическое программирование (Dynamic programming, by Umesh V. Vazirani, см. https://code.energy/vazirani).

Глава 4

ДАННЫЕ

> Хорошие программисты беспокоятся о структурах данных и их отношениях.
>
> *Линус Торвальдс*

Контроль над данными в computer science имеет принципиальное значение: вычислительные процессы состоят из операций над данными, которые преобразуют вход в выход. Но алгоритмы обычно не конкретизируют, *как* они выполняются. К примеру, алгоритм `merge` (см. раздел «Итерация» главы 3) опирается на неустановленный внешний исходный код, который создает списки чисел, проверяет наличие в них элементов и добавляет эти элементы в списки. Алгоритм `queens` (раздел «Поиск (перебор) с возвратом») делает то же самое: он не заботится о том, как выполняются операции на шахматной доске или как позиции хранятся в памяти. Эти детали скрыты позади так называемых *абстракций*. В главе 4 мы узнаем:


как *абстрактные типы данных* делают код чистым;


какие *общие абстракции* желательно знать и уметь ими пользоваться;


какие существуют способы *структурирования данных* в памяти.

Но прежде чем мы углубимся в эту тему, давайте разберемся, что означают термины «абстракция» и «тип данных».

Абстракции

Абстракции позволяют нам опускать детали; они представляют простой интерфейс для доступа к функциональности сложных объектов. Например, автомобиль скрывает сложный механизм за панелью управления, причем таким образом, что любой человек может легко научиться водить без необходимости разбираться в машиностроении.

В программном обеспечении *процедурные абстракции* скрывают за вызовом процедур сложности реализации процесса. В алгоритме `trade` (см. раздел «Разделяй и властвуй» главы 3) процедуры `min` и `max` скрывают механику поиска минимальных и максимальных чисел и тем самым упрощают алгоритм. При помощи абстракций можно создавать модули[1], которые позволяют выполнять сложные операции вызовом одной единственной процедуры, вроде этой:

```
html ← fetch_source("https://code.energy")
```

Всего одной строкой кода мы получили страницу сайта, несмотря на то что внутренние операции для этой задачи чрезвычайно сложны[2].

Абстракции данных будут центральной темой главы. Они скрывают детали процессов обработки данных. Но прежде чем мы сможем понять, как работает абстракция, нам необходимо освежить наше понимание типов данных.

---

[1] *Модуль*, или *библиотека*, — это структурная часть программного обеспечения, которая предлагает универсальные вычислительные процедуры. Их можно включать при необходимости в другие части программного обеспечения.

[2] Они сопряжены с разрешением доменного имени, созданием сетевого сокета, установлением шифрованного SSL-соединения и многим другим.

Тип данных

Мы различаем разные типы крепежных изделий (как, например, винты, гайки и гвозди) согласно операциям, которые можем с ними выполнить (к примеру, используя отвертку, гаечный ключ или молоток). Точно так же мы различаем разные типы *данных* согласно операциям, которые могут быть выполнены с ними.

Например, переменная, содержащая последовательность символов, которые можно преобразовать в верхний или нижний регистр, и допускающая добавление новых символов, имеет тип String (строка). Строки представляют текстовые данные. Переменная, которую можно инвертировать и которая допускает операции `XOR`, `OR` и `AND`, имеет тип Boolean (логический). Такие булевы переменные принимают одно из двух значений: `True` или `False`. Переменные, которые можно складывать, делить и вычитать, имеют тип Number (численный).

Каждый тип данных связан с конкретным набором процедур. Процедуры, которые предназначены для работы с переменными, хранящими списки, отличаются от процедур, которые предназначены для работы с переменными, хранящими множества, а те, в свою очередь, отличаются от процедур, которые предназначены для работы с числами.

4.1. Абстрактные типы данных

Абстрактный тип данных (АТД) — это подробное описание группы операций, применимых к конкретному типу данных. Они определяют интерфейс для работы с переменными, содержащими данные конкретного типа, и скрывают все подробности хранения данных в памяти и управления ими.

Когда нашим алгоритмам приходится оперировать данными, мы не включаем в них команды чтения и записи в память компьютера. Мы используем внешние модули обработки данных, которые предоставляют процедуры, определенные в АТД.

Например, для работы с переменными, хранящими списки, нам нужны: процедуры для создания и удаления списков; процедуры для доступа к *n*-му элементу списка и его удаления; процедура для добавления нового элемента в список. Определения этих процедур (их имена и что они делают) содержатся в АТД «Список». Мы можем работать со списками, руководствуясь исключительно этими процедурами. Так что мы никогда не управляем памятью компьютера непосредственно.

Преимущества использования АТД

Простота. АТД делает наш код доступнее для понимания и изменения. Опустив детали в процедурах обработки данных, вы сможете сосредоточиться на самом главном — на алгоритмическом процессе решения задачи.

Гибкость. Существуют разные способы структурирования данных в памяти и, как следствие, разные модули обработки одного и того же типа данных. Мы должны выбрать тот, что лучше соответствует текущей ситуации. Модули, которые реализуют тот же АТД, предлагают одинаковые процедуры. Это означает, что мы можем изменить способ хранения данных и выполнения операций, просто применив другой модуль обработки данных. Это как с автомобилями: все автомобили на электрической и бензиновой тяге имеют одинаковый интерфейс. Если вы умеете управлять одним автомобилем, вы сумеете управлять и любыми другими.

Повторное использование. Мы задействуем одни и те же модули в проектах, где обрабатываются данные одинакового типа. Например, процедуры `power_set` и `recursive_power_set` из предыдущей главы работают с переменными, представляющими множества, `Set`. Это означает, что мы можем использовать один и тот же модуль `Set` в обоих алгоритмах.

Организация. Нам, как правило, приходится оперировать несколькими типами данных: числами, текстом, географическими координатами, изображениями и пр. Чтобы лучше организовать нашу

программу, мы создаем отдельные модули, каждый из которых содержит код, работающий исключительно с конкретным типом данных. Это называется *разделением функциональности*: части кода, которые имеют дело с одним и тем же логическим аспектом, должны быть сгруппированы в собственном, отдельном модуле. Когда они перемешаны с чем-то посторонним, это называется *запутанным кодом*.

Удобство. Мы берем модуль обработки данных, написанный кем-то другим, разбираемся с использованием определенных в его АТД процедур, и сразу после этого можем их использовать, чтобы оперировать данными нового типа. Нам не нужно понимать, как функционирует этот модуль.

Устранение программных ошибок. Если вы используете модуль обработки данных, то в вашем программном коде не будет ошибок обработки данных. Если же вы найдете ошибку в модуле обработки данных, то, устранив ее один раз, вы немедленно исправите все части своего кода, которые она затрагивала.

4.2. Общие абстракции

Чтобы решить вычислительную задачу, крайне важно знать тип обрабатываемых данных и операции, которые вам предстоит выполнять с этими данными. Не менее важно принять решение, какой АТД вы будете использовать.

Далее мы представим хорошо известные абстрактные типы данных, которые вы должны знать. Они встречаются во множестве алгоритмов. Они даже поставляются в комплекте вместе со многими языками программирования.

Примитивные типы данных

Примитивные типы данных — это типы данных со встроенной поддержкой в языке программирования, который вы используете. Для

работы с ними на нужны внешние модули. Сюда относятся целые числа, числа с плавающей точкой[1] и универсальные операции с ними (сложение, вычитание, деление). Большинство языков также по умолчанию поддерживают хранение в своих переменных текста, логических значений и других простых данных.

Стек

Представьте стопку бумаги. Вы можете положить на нее еще один лист либо взять верхний. Лист, который добавили в стопку первым, всегда будет удален из стопки в последнюю очередь. *Стек* (stack) представляет такую стопку и позволяет работать только с ее верхним элементом. Элемент на вершине стека — это *всегда* элемент, который был добавлен последним. Реализация стека должна обеспечивать по крайней мере две операции:

- `push(e)` — добавить элемент `e` на вершину стека;
- `pop()` — получить и удалить элемент с вершины стека.

Более совершенные разновидности стеков могут поддерживать дополнительные операции: проверку наличия в стеке элементов или получение их текущего количества.

Такая обработка данных известна под названием **LIFO** (Last-In, First-Out, «последним пришел, первым вышел»); мы можем удалить только верхний элемент, который был добавлен последним. Стек — это важный тип данных, он встречается во многих алгоритмах. Для реализации функции «Отменить ввод» в текстовом редакторе каждая вносимая вами правка помещается в стек. Если вы хотите ее отменить, то текстовый редактор выталкивает правку из стека и возвращается к предыдущему состоянию.

---

[1] Числа с плавающей точкой — это общепринятый способ представления чисел, имеющих десятичный разделитель.
В англоязычных странах в качестве десятичного разделителя используется точка, в большинстве остальных — запятая. — *Примеч. пер.*

Чтобы реализовать поиск с возвратом (см. соответствующий раздел главы 3) без рекурсии, вы должны запоминать в стеке последовательность вариантов, которые привели вас к текущей точке. Обследуя новый узел, мы помещаем ссылку на него в стек. Чтобы вернуться на шаг назад, мы просто выталкиваем (`pop()`) последний элемент из стека, заодно получая ссылку на предыдущее состояние.

Очередь

Очередь (queue) — это полная противоположность стека. Она тоже позволяет сохранять и извлекать данные, но элементы всегда берутся из начала очереди — тот, который находился в очереди дольше всего. Звучит пугающе? В действительности это то же самое, что и реальная очередь из людей, стоящих у раздачи в столовой! Вот основные операции с очередями:

- **`enqueue(e)`** — добавить элемент `e` в конец очереди;
- **`dequeue()`** — удалить элемент из начала очереди.

Очередь работает по принципу организации данных **FIFO** (First-In, FirstOut, «первый пришел, первый вышел»), потому что первый помещенный в очередь элемент всегда покидает ее первым.

Очереди используются во многих вычислительных сценариях. Если вы реализуете онлайновую службу доставки пиццы, то вы, скорее всего, будете хранить заказы в очереди. В качестве мысленного эксперимента подумайте, что вышло бы, если б ваша пиццерия обслуживала заказы с использованием стека вместо очереди. 

Очередь с приоритетом

Очередь с приоритетом (priority queue) аналогична обычной очереди с той лишь разницей, что помещенным в нее элементам присваивается *приоритет*. Люди, ожидающие медицинской помощи в больнице, — вот реальный пример очереди с приоритетом. Экстренные

случаи получают высший приоритет и переходят непосредственно в начало очереди, тогда как незначительные добавляются в ее конец. Основные операции, реализуемые очередью с приоритетом, таковы:

- **`enqueue(e, p)`** — добавить элемент `e` в очередь согласно уровню приоритетности `p`;
- **`dequeue()`** — вернуть элемент, расположенный в начале очереди, и удалить его.

В компьютере, как правило, много рабочих процессов — и всего один или несколько ЦП, предназначенных для их выполнения. Операционная система ставит все процессы, ожидающие выполнения, в очередь с приоритетом. Каждый процесс получает свой уровень приоритетности. Операционная система исключает процесс из очереди и позволяет ему некоторое время поработать. Позднее, если процесс не завершился, он снова ставится в очередь. Операционная система раз за разом повторяет эту процедуру.

Некоторые процессы более важны и безотлагательно получают процессорное время, другие ожидают в очереди дольше. Процесс, который получает ввод с клавиатуры, как правило, получает самый высокий приоритет — ведь если компьютер не реагирует на нажатия клавиш, то пользователь может подумать, что он завис, и попробует сделать «холодный» перезапуск, что *всегда* вредно.

Список

При работе с группами элементов иногда требуется гибкость. Например, может понадобиться переупорядочить элементы или извлекать, вставлять и удалять их в произвольном порядке. В этих случаях удобно использовать *список* (list). Чаще всего АТД «Список» поддерживает следующие операции:

- **`insert(n, e)`** — вставить элемент `e` в позицию `n`;
- **`remove(n)`** — удалить элемент, находящийся в позиции `n`;

- **`get(n)`** — получить элемент, находящийся в позиции `n`;
- **`sort()`** — отсортировать элементы;
- **`slice(start, end)`** — вернуть фрагмент списка, начинающийся с позиции `start` и заканчивающийся в позиции `end`;
- **`reverse()`** — изменить порядок следования элементов на обратный.

Список — один из наиболее используемых АТД. Например, если вам нужно хранить ссылки на часто запрашиваемые файлы в системе, то список — идеальное решение: вы можете сортировать ссылки для отображения и удалять их, если к соответствующим файлам стали обращаться реже.

Стеку и очереди следует отдавать предпочтение, когда гибкость, предоставляемая списком, не нужна. Использование более простого АТД гарантирует, что данные будут обрабатываться строгим и предсказуемым образом (по принципу FIFO или LIFO). Это также делает код яснее: зная, что переменная представляет собой стек, легче понять характер потоков данных на входе и выходе.

Сортированный список

Сортированный список (sorted list) бывает полезен, когда нужна *постоянная упорядоченность* элементов. В этих случаях вместо поиска правильной позиции перед каждой вставкой в список (и периодической сортировки его вручную) мы используем сортированный список. Что бы в него ни помещали, элементы всегда будут стоять по порядку. Ни одна из операций этого АТД не позволяет переупорядочить его элементы. Сортированный список поддерживает меньше операций, чем обычный:

- **`insert(e)`** — вставить элемент `e` в автоматически определяемую позицию в списке;

- **`delete(n)`** — удалить элемент, находящийся в позиции `n`;
- **`get(n):`** получить элемент, находящийся в позиции `n`.

Словарь (map) используется для хранения соответствий между двумя объектами: ключом `key` и значением `value`. Вы можете осуществить поиск по ключу и получить связанное с ним значение. Словарь хорошо подходит, например, для хранения идентификационных номеров пользователей в качестве ключей и полных имен в качестве значений. Такой словарь по заданному идентификационному номеру вернет связанное с ним имя. Существуют следующие операции для словарей:

- **`set(key, value)`** — добавить элемент с заданным ключом и значением;
- **`delete(key)`** — удалить ключ `key` и связанное с ним значение;
- **`get(key)`** — получить значение, связанное с ключом `key`.

Множество

Множество (set) представляет неупорядоченные группы *уникальных* элементов, подобные математическим множествам, которые описаны в приложении III. Этот АТД используется, когда неважен порядок следования элементов либо когда нужно обеспечить уникальность элементов в группе. Стандартный набор операций для множества включает в себя:

- **`add(e)`** — добавить элемент в множество или вернуть ошибку, если элемент уже присутствует в множестве;
- **`list()`** — перечислить все элементы, присутствующие в множестве;
- **`delete(e)`** — удалить элемент из множества.

АТД для программиста — как приборная панель для водителя. Но давайте все-таки попробуем понять, как проложены провода *за* этой приборной панелью.

4.3. Структуры

Абстрактный тип данных лишь описывает, какие действия можно совершать с переменными конкретного типа. Он определяет список операций, но не объясняет, *как* они выполняются. Со *структурами данных* — обратная ситуация: они описывают, как данные организованы и как к ним получить доступ в памяти компьютера. Структуры данных обеспечивают реализацию АТД в модулях обработки данных.

Есть множество разных способов реализации АТД, потому что существуют самые разные структуры данных. Выбор реализации АТД, которая использует структуру данных, лучше соответствующую вашим потребностям, имеет существенное значение для создания эффективных компьютерных программ. Далее мы рассмотрим наиболее распространенные структуры данных и узнаем об их сильных и слабых сторонах.

Массив

Массив (array) — это самый простой способ хранения набора элементов в памяти компьютера. Он заключается в выделении единого пространства в памяти и последовательной записи в него ваших элементов. Конец последовательности отмечается специальным маркером `NULL` (рис. 4.1).

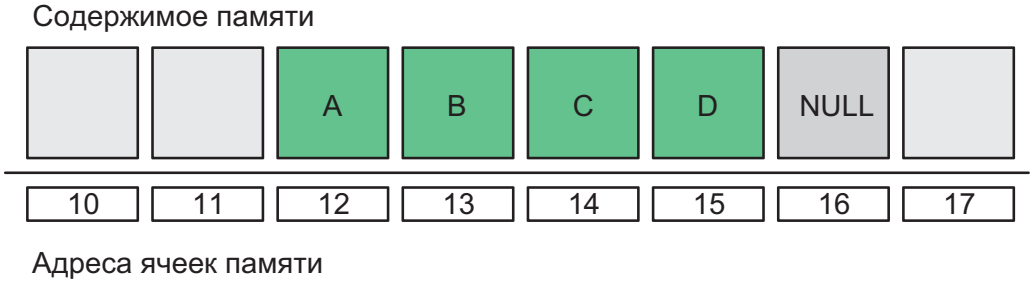


Рис. 4.1. Массив в памяти компьютера

Каждый объект в массиве занимает такой же объем памяти, что и любой другой. Представим массив, начинающийся с адреса ячейки памяти *s,* где каждый элемент занимает *b* байт. Чтобы получить *n*-й элемент, нужно извлечь *b* байт, начиная с позиции в памяти $s + (b \times n)$.

Это позволяет *напрямую* обращаться к любому элементу массива. Массив наиболее полезен для реализации стека, однако он может также использоваться для списков и очередей. Массивы легко программируются и имеют преимущество, позволяя мгновенно обращаться к любым элементам. Но есть у них и недостатки.

Может оказаться практически нецелесообразным выделять большие непрерывные блоки памяти. Если вам нужно наращивать массив, то в памяти может не оказаться смежного с ним достаточного большого пространства. Удаление элемента из середины массива сопряжено с определенными проблемами: вам придется сдвинуть *все* последующие элементы на одну позицию к началу либо отметить пространство памяти удаленного элемента как свободное. Ни один из этих вариантов не желателен. Аналогично вставка элемента внутрь массива вынудит вас сдвинуть *все* последующие элементы на одну позицию к концу.

Связный список

Связный список (linked list) позволяет хранить элементы в цепи ячеек, которые не обязательно должны находиться в последовательных адресах памяти. Память для ячеек выделяется по мере необходимости. Каждая ячейка имеет указатель, сообщающей об адресе следующей в цепи. Ячейка с пустым указателем (`NULL`) отмечает конец цепи (рис. 4.2).

Связные списки используются для реализации стеков, списков и очередей. При наращивании связного списка не возникает никаких проблем: любая ячейка может храниться в любой части памяти. Таким образом, размер списка ограничен только объемом имеющейся

свободной памяти. Также не составит труда вставить элементы в середину списка или удалить их — достаточно просто изменить указатели ячеек (рис. 4.3).

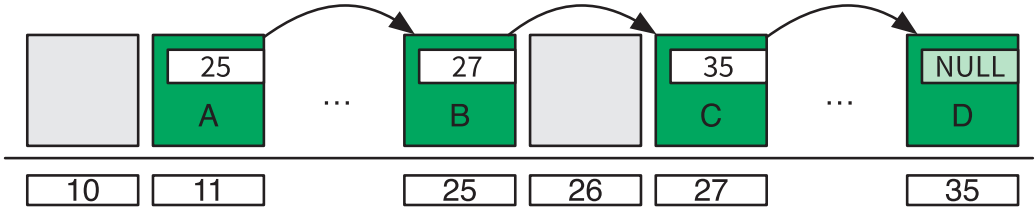


Рис. 4.2. Связный список в памяти компьютера

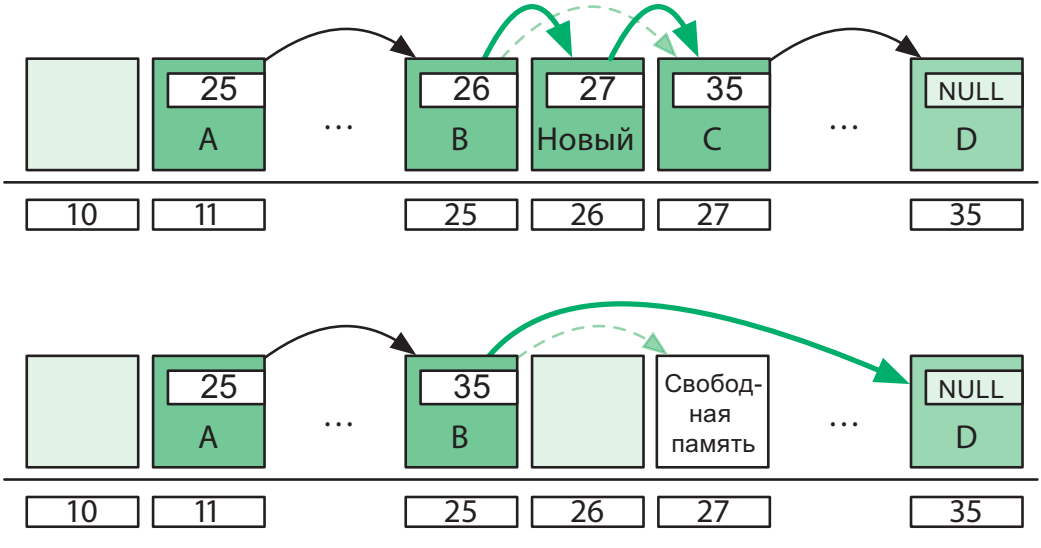


Рис. 4.3. Добавление элемента между B и C. Удаление C

Связный список тоже имеет свои недостатки: мы не можем сразу получить *n*-й элемент. Сначала придется прочитать первую ячейку, извлечь из нее адрес второй ячейки, затем прочитать вторую ячейку,

извлечь из нее указатель на следующую ячейку и т. д., пока мы не доберемся до *n*-й ячейки.

Кроме того, когда известен адрес всего одной ячейки, не так просто ее удалить или переместиться по списку назад. Не имея другой информации, нельзя узнать адрес предыдущей ячейки в цепи.

Двусвязный список

Двусвязный список (double linked list) — это связный список, где ячейки имеют два указателя: один на предыдущую ячейку, другой — на следующую (рис. 4.4).

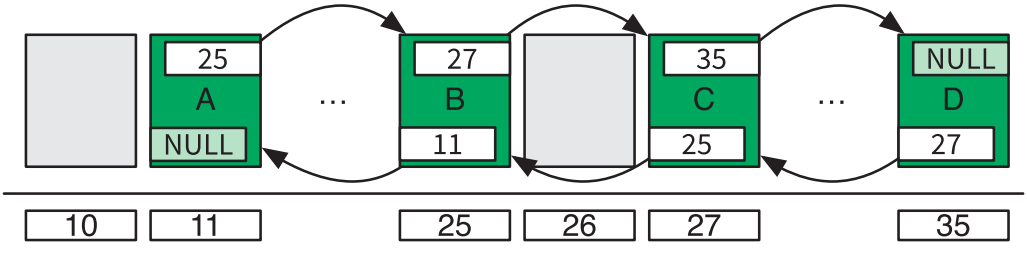


Рис. 4.4. Двусвязный список в памяти компьютера

Он обладает тем же преимуществом, что и связный список: не требует предварительного выделения большого блока памяти, потому что пространство для новых ячеек может выделяться по мере необходимости. При этом дополнительные указатели позволяют двигаться по цепи ячеек вперед *и* назад. В таком случае, если известен адрес всего одной ячейки, мы сможем быстро ее удалить.

И тем не менее мы по-прежнему не имеем прямого доступа к *n*-му элементу. Кроме того, для поддержки двух указателей в каждой ячейке требуется более сложный код и больше памяти.

Массивы против связных списков

Языки программирования с богатым набором средств обычно включают встроенные реализации списка, очереди, стека и других АТД. Эти реализации часто основаны на некоторой стандартной структуре данных. Некоторые из них могут даже автоматически переключаться с одной структуры данных на другую во время выполнения программ, в зависимости от способа доступа к данным.

Когда производительность не является проблемой, мы можем опереться на эти универсальные реализации АТД и не переживать по поводу структур данных. Но когда производительность должна быть оптимальной либо когда вы имеете дело с низкоуровневым языком, не имеющим таких встроенных средств, вы *сами* должны решать, какие структуры данных использовать. Проанализируйте операции, посредством которых вы будете обрабатывать информацию, и выберите реализацию с надлежащей структурой данных. Связные списки предпочтительнее массивов, когда:

- нужно, чтобы операции вставки и удаления выполнялись чрезвычайно быстро;
- не требуется произвольный доступ к данным;
- приходится вставлять или удалять элементы между других элементов;
- заранее не известно количество элементов (оно будет расти или уменьшится по ходу выполнения программы).

Массивы предпочтительнее связных списков, когда:

- нужен произвольный доступ к данным;
- нужен очень быстрый доступ к элементам;
- число элементов не изменяется во время выполнения программы, благодаря чему легко выделить непрерывное пространство памяти.

Дерево

Как и связный список, *дерево* (tree) использует элементы, которым для хранения объектов не нужно располагаться в физической памяти непрерывно. Ячейки здесь тоже имеют указатели на другие ячейки, однако, в отличие от связных списков, они располагаются не линейно, а в виде ветвящейся структуры. Деревья особенно удобны для иерархических данных, таких как каталоги с файлами или система субординации (рис. 4.5).

В терминологии деревьев ячейка называется *узлом*, а указатель из одной ячейки на другую — *ребром*. Самая первая ячейка — это *корневой узел*, он единственный не имеет родителя. Все остальные узлы в деревьях должны иметь строго *одного* родителя[1].

Два узла с общим родителем называются братскими. Родитель узла, прародитель, прапрародитель (и т. д. вплоть до корневого узла) — это предки. Аналогично дочерние узлы, внуки, правнуки (и т. д. вплоть до нижней части дерева) называются потомками.

Узлы, не имеющие дочерних узлов, — это *листья* (по аналогии с листьями настоящего дерева ). *Путь* между двумя узлами определяется множеством узлов и ребер.

Уровень узла — это длина пути от него до корневого узла, *высота* дерева — уровень самого глубокого узла в дереве (рис. 4.6). И, наконец, множество деревьев называется *лесом*.

[1] Если узел нарушает это правило, многие алгоритмы поиска по дереву не будут работать.

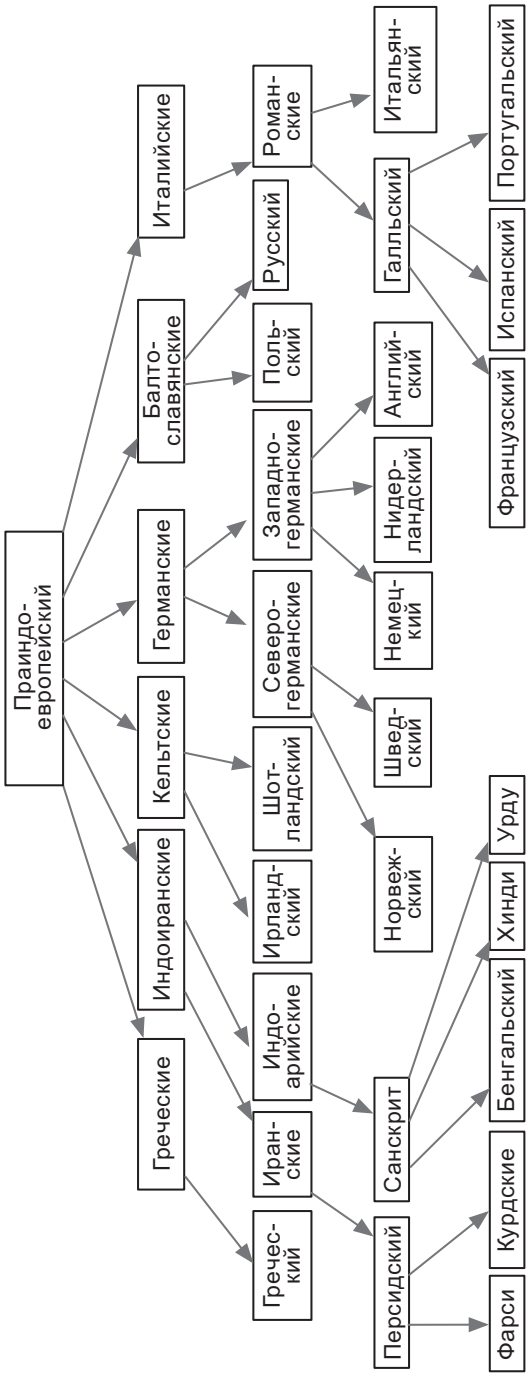


Рис. 4.5. Дерево происхождения индоевропейских языков

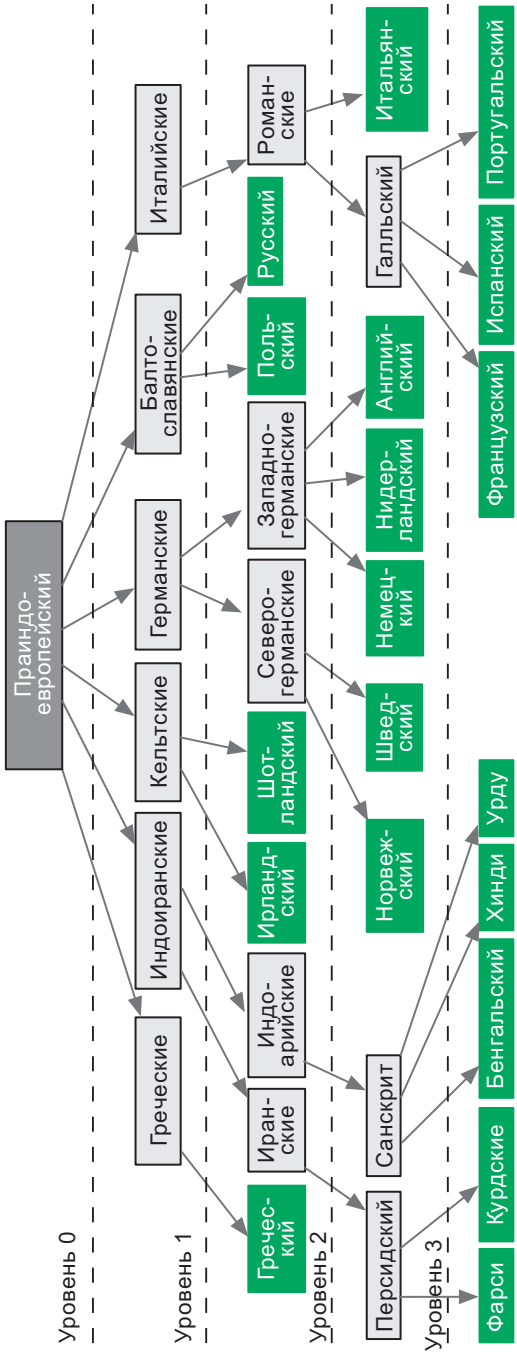


Рис. 4.6. Листья этого дерева представляют современные языки

Двоичное дерево поиска

Двоичное дерево поиска (binary search tree) — это особый тип дерева, поиск в котором выполняется особенно эффективно. Узлы в двоичном дереве поиска могут иметь не более двух дочерних узлов. Кроме того, узлы располагаются согласно их значению/ключу. Дочерние узлы слева от родителя должны быть меньше него, а справа — больше (рис. 4.7).

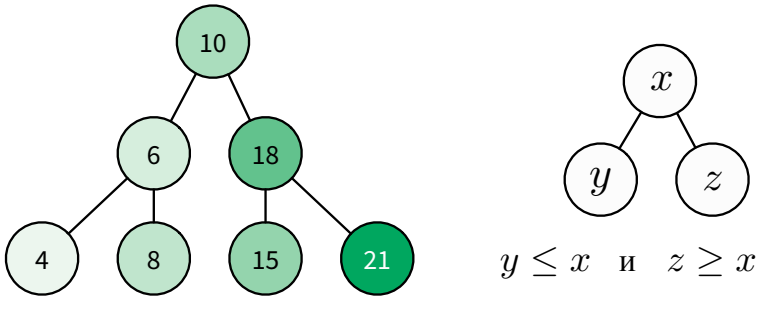


Рис. 4.7. Пример двоичного дерева поиска

Если дерево соблюдает это свойство, в нем легко отыскать узел с заданным ключом/значением:

```
function find_node(binary_tree, value)
node ← binary_tree.root_node
    while node
        if node.value = value
            return node
        if value > node.value
            node ← node.right
        else
            node ← node.left
    return "NOT FOUND"
```

Чтобы вставить элемент, находим последний узел, следуя правилам построения дерева поиска, и подключаем к нему новый узел справа или слева:

```
function insert_node(binary_tree, new_node)
    node ← binary_tree.root_node
    while node
        last_node ← node
        if new_node.value > node.value
            node ← node.right
        else
            node ← node.left
    if new_node.value > last_node.value
        last_node.right ← new_node
    else
        last_node.left ← new_node
```

Балансировка дерева. Если вставить в двоичное дерево поиска слишком много узлов, в итоге получится очень высокое дерево, где большинство узлов имеют всего один дочерний узел. Например, если последовательно вставлять узлы с ключами/значениями, которые всегда больше предыдущих, в итоге получится нечто, похожее на связный список. Однако мы можем перестроить узлы в дереве так, что его высота уменьшится. Эта процедура вызывается *балансировкой дерева*. Идеально сбалансированное дерево имеет минимальную высоту (рис. 4.8).

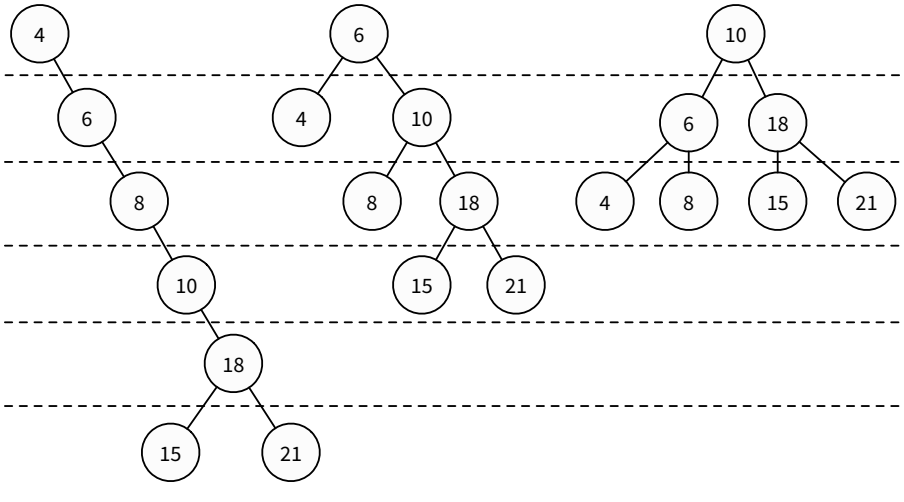


Рис. 4.8. Одно и то же двоичное дерево поиска с разной балансировкой: сбалансированное плохо, средне и идеально

Большинство операций с деревом требует обхода узлов по ссылкам, пока не будет найден конкретный узел. Чем больше высота дерева, тем длиннее средний путь между узлами и тем чаще приходится обращаться к памяти. Поэтому важно уменьшать высоту деревьев. Идеально сбалансированное двоичное дерево поиска можно создать из сортированного списка узлов следующим образом:

```
function build_balanced(nodes)
    if nodes is empty
        return NULL
    middle ← nodes.length/2
    left ← nodes.slice(0, middle - 1)
    right ← nodes.slice(middle + 1, nodes.length)
    balanced ← BinaryTree.new(root=nodes[middle])
    balanced.left ← build_balanced(left)
    balanced.right ← build_balanced(right)
    return balanced
```

Рассмотрим двоичное дерево поиска с n узлами и с максимально возможной высотой n. В этом случае оно похоже на связный список. Минимальная высота идеально сбалансированного дерева равняется $\log_2 n$. Сложность поиска элемента в дереве пропорциональна его высоте. В худшем случае, чтобы найти элемент, придется опускаться до самого нижнего уровня листьев. Поиск в сбалансированном дереве с n элементами, следовательно, имеет $O(\log n)$. Вот почему эта структура данных часто выбирается для реализации множеств (где предполагается проверка присутствия элементов) и словарей (где нужно искать пары «ключ — значение»).

Однако балансировка дерева — дорогостоящая операция, поскольку требует сортировки всех узлов. Если делать балансировку после каждой вставки или удаления, операции станут значительно медленнее. Обычно деревья подвергаются этой процедуре после нескольких вставок и удалений. Но балансировка от случая к случаю является разумной стратегией только в отношении редко изменяемых деревьев.

Для эффективной обработки двоичных деревьев, которые изменяются часто, были придуманы *сбалансированные двоичные деревья*

(self-balancing binary tree)[1]. Их процедуры вставки или удаления элементов гарантируют, что дерево остается сбалансированным. *Красно-черное дерево* (red-black tree) — это хорошо известный пример сбалансированного дерева, которое окрашивает узлы красным либо черным цветом в зависимости от стратегии балансировки[2]. Красно-черные деревья часто используются для реализации словарей: словарь может подвергаться интенсивной правке, но конкретные ключи в нем по-прежнему будут находиться быстро вследствие балансировки.

AVL-дерево (AVL tree) — это еще один подвид сбалансированных деревьев. Оно требует немного большего времени для вставки и удаления элементов, чем красно-черное дерево, но, как правило, обладает лучшим балансом. Это означает, что оно позволяет получать элементы быстрее, чем красно-черное дерево. AVL-деревья часто используются для оптимизации производительности в сценариях, для которых характерна высокая интенсивность чтения.

Данные часто хранятся на магнитных дисках, которые считывают их большими блоками. В этих случаях используется обобщенное двоичное *B-дерево* (B-tree). В таких деревьях узлы могут хранить более одного элемента и иметь более двух дочерних узлов, что позволяет им эффективно оперировать данными в больших блоках. Как мы вскоре увидим, B-деревья обычно используются в системах управления базами данных.

Двоичная куча

Двоичная куча (binary heap) — особый тип двоичного дерева поиска, в котором можно мгновенно найти самый маленький (или самый

[1] Иногда их называют двоичными деревьями с автоматической балансировкой, или самоуравновешивающимися двоичными деревьями. — *Примеч. пер.*

[2] Стратегии автоматической балансировки выходят за рамки этой книги. Если вам любопытно, то в Интернете имеются видеоматериалы, которые показывают, как работают эти алгоритмы.

большой) элемент. Эта структура данных особенно полезна для реализации очередей с приоритетом. Операция получения минимального (или максимального) элемента имеет сложность $O(1)$, потому что он всегда является корневым узлом дерева. Поиск и вставка узлов здесь по-прежнему стоят $O(\log n)$. Кучи подчиняются тем же правилам размещения узлов, что и двоичные деревья поиска, но есть одно ограничение: родительский узел должен быть больше (либо меньше) *обоих* своих дочерних узлов (рис. 4.9).

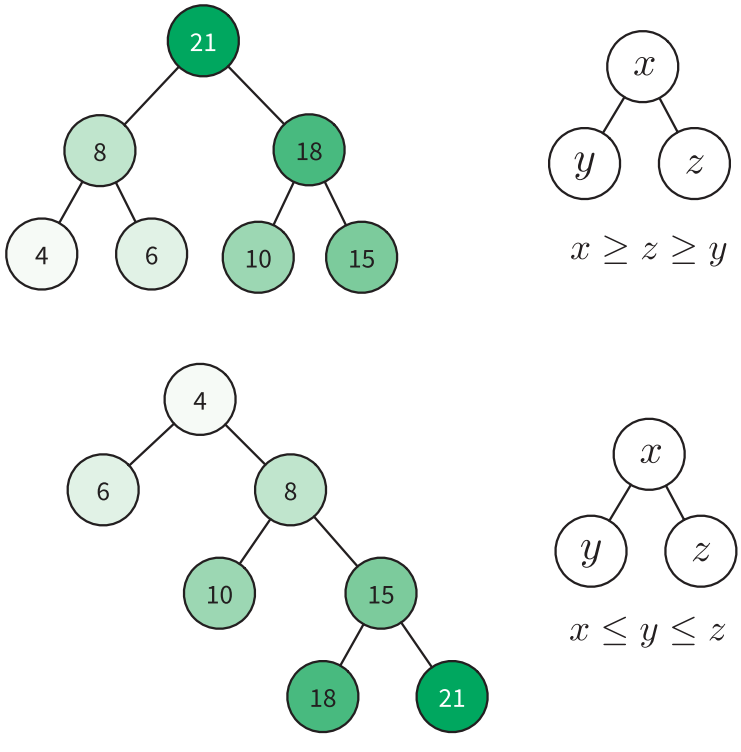


Рис. 4.9. Узлы, организованные как двоичная куча max-heap (вверху) и двоичная куча min-heap (внизу)[1]

[1] Двоичная куча min-heap характерна тем, что значение в любой ее вершине *не больше*, чем значения ее потомков; в двоичной куче max-heap, наоборот, значение в любой вершине *не меньше*, чем значения ее потомков. — *Примеч. пер.*

Обращайтесь к двоичной куче всегда, когда планируете часто иметь дело с максимальным (либо минимальным) элементом множества.

Граф

Граф (graph) аналогичен дереву. Разница состоит в том, что у него нет ни дочерних, ни родительских узлов (вершин) и, следовательно, нет корневого узла. Данные свободно организованы в виде узлов (вершин) и дуг (ребер) так, что любой узел может иметь произвольное число входящих и исходящих ребер.

Это самая гибкая из всех структур, она используется для представления почти всех типов данных. Например, графы идеальны для социальной сети, где узлы — это люди, а ребра — дружеские связи.

Хеш-таблица

Хеш-таблица (hash table) — это структура данных, которая позволяет находить элементы за $O(1)$. Поиск занимает постоянное время вне зависимости от того, ищете вы среди 10 млн элементов или всего среди 10.

Так же, как массив, хеш для хранения данных требует предварительного выделения большого блока последовательной памяти. Но, в отличие от массива, его элементы хранятся не в упорядоченной последовательности. Позиция, занимаемая элементом, «волшебным образом» задается *хеш-функцией*. Это специальная функция, которая на входе получает данные, предназначенные для хранения, и возвращает число, кажущееся случайным. Оно интерпретируется как позиция в памяти, куда будет помещен элемент.

Это позволяет нам получать элементы немедленно. Заданное значение сначала пропускается через хеш-функцию. Она выдает точную позицию, где элемент должен находиться в памяти. Если элемент был сохранен, то вы найдете его в этой позиции.

Однако с хеш-таблицами возникают проблемы: иногда хеш-функция возвращает одинаковую позицию для разных входных данных. Такая ситуация называется *хеш-коллизией*. Когда она происходит, все такие элементы должны быть сохранены в одной позиции в памяти (эта проблема решается, например, посредством связного списка, который начинается с заданного адреса). Хеш-коллизии влекут за собой издержки процессорного времени и памяти, поэтому их желательно избегать.

Хорошая хеш-функция должна возвращать разные значения для разных входных данных. Следовательно, чем шире диапазон значений, которые может вернуть хеш-функция, тем больше будет доступно позиций для данных и меньше вероятность возникновения хеш-коллизии. Поэтому нужно гарантировать, чтобы в хеш-таблице оставалось незанятым по крайней мере 50 % пространства. В противном случае коллизии начнут происходить слишком часто и производительность хеш-таблицы значительно упадет.

Хеш-таблицы часто используются для реализации словарей и множеств. Они позволяют выполнять операции вставки и удаления быстрее, чем структуры данных, основанные на деревьях.

С другой стороны, для корректной работы хеш-таблицы требуют выделения очень большого блока непрерывной памяти.

Подведем итоги

Мы узнали, что структуры данных определяют конкретные способы организации элементов в памяти компьютера. Разные структуры данных требуют разных операций для хранения, удаления, поиска и обхода хранящихся данных. Чудодейственного средства не существует: всякий раз нужно выбирать, какую структуру данных использовать в соответствии с текущей ситуацией.

Еще мы узнали, что вместо структур данных лучше иметь дело с АТД. Они освобождают код от деталей, связанных с обработкой данных,

и позволяют легко переключаться с одной структуры на другую без каких-либо изменений в коде.

Не изобретайте колесо заново, пытаясь с нуля создавать базовые структуры данных и абстрактные типы данных (если только вы не делаете это ради забавы, обучения или исследования). Пользуйтесь проверенными временем сторонними библиотеками обработки данных. Большинство языков имеет встроенную поддержку этих структур.

Полезные материалы

- Балансировка двоичного дерева поиска (Balancing a Binary Search Tree, Stoimen, см. https://code.energy/stoimen).
- Лекция Корнелльского университета по абстрактным типам данных и структурам данных (Cornell Lecture on Abstract Data Types and Data Structures, см. https://code.energy/cornell-adt).
- Конспекты IITKGP по абстрактным типам данных (IITKGP notes on Abstract Data Types, см. https://code.energy/iitkgp).
- Реализация дерева поиска на «интерактивном Python» (Search Tree Implementation by “Interactive Python”, см. https://code.energy/python-tree).

Глава 5

АЛГОРИТМЫ

> [Программирование является] привлекательным занятием не только потому, что оно перспективно с экономической и научной точек зрения, но и потому, что оно во многом может стать эстетическим опытом, как сочинение стихов или музыки.
>
> *Дональд Кнут*

Человечество изыскивает решения все более и более трудных задач. В большинстве случаев вам приходится иметь дело с задачами, над примерными аналогами которых уже потрудились многие разработчики. Вполне вероятно, что они придумали эффективные алгоритмы, которые можно брать и использовать. Когда вы решаете задачи, вашим первым шагом всегда должен быть поиск существующих алгоритмов[1]. В этой главе мы займемся исследованием хорошо известных алгоритмов, которые:


эффективно *сортируют* очень длинные списки;


быстро *отыскивают* нужный элемент;

[1] Ситуация, когда найдена новая, неисследованная задача, — это редкость. Когда исследователи находят новую задачу, они пишут об этом научную работу.


оперируют и управляют *графами*;


используют *исследование операций* для оптимизации процессов.

Вы научитесь распознавать проблемы, к которым можно применить известные решения. Существует много различных типов задач: сортировка данных, поиск закономерностей (образов, шаблонов), прокладывание маршрута и др. И многие типы алгоритмов имеют непосредственное отношение к областям научно-практических исследований: обработке изображений, криптографии, искусственному интеллекту... В этой книге мы не сможем охватить их все[1]. Однако мы изучим самые важные алгоритмы, с которыми должен быть знаком любой хороший программист.

5.1. Сортировка

До появления компьютеров сортировка данных была известной проблемой, ее ручное выполнение занимало колоссальное количество времени. Когда в 1890-х годах компания Tabulating Machine Company (которая позже стала называться IBM) автоматизировала операции сортировки, это позволило на несколько лет быстрее обработать данные переписи населения США.

Существует много алгоритмов сортировки. Более простые имеют временную сложность $O(n^2)$. *Сортировка выбором* (см. раздел «Оценка затрат времени» главы 2) — один из таких алгоритмов. Именно его люди предпочитают использовать для сортировки физической колоды карт. Сортировка выбором принадлежит многочисленной группе алгоритмов с квадратичной стоимостью. Мы, как правило, используем их для упорядочивания наборов данных, состоящих меньше чем из 1000 элементов. Одним из известных алгоритмов является *сортировка вставками*. Он показывает очень хорошую эффективность

[1] Вот более полный список: https://code.energy/algo-list.

в сортировке уже почти упорядоченных наборов данных даже очень большого объема:

```
function insertion_sort(list)
    for i ← 2 … list.length
        j ← i
        while j and list[j-1] > list[j]
            list.swap_items(j, j-1)
            j ← j - 1
```

Выполните этот алгоритм на бумаге, с использованием большей частью отсортированного списка чисел. Для массивов, где не упорядочено незначительное число элементов, `insertion_sort` имеет сложность $O(n)$. В этом случае он выполняет меньше операций, чем какой-либо другой алгоритм сортировки.

В отношении крупных наборов данных, о которых нельзя сказать, что они отсортированы большей частью, алгоритмы с временной сложностью $O(n^2)$ оказываются слишком медленными (см. табл. 3.2). Здесь нам нужны более эффективные подходы. Самыми известными высокоскоростными алгоритмами сортировки являются *сортировка слиянием* (см. раздел «Разделяй и властвуй» главы 3) и так называемая *быстрая сортировка*, оба имеют сложность $O(n \log n)$. Вот как алгоритм быстрой сортировки раскладывает по порядку колоду карт.

1. Если в колоде менее четырех карт, то упорядочить их — и работа завершена. В противном случае перейти к шагу 2.
2. Вынуть из колоды наугад любую карту, которая становится *опорной*.
3. Карты со значением *больше*, чем у опорной, кладутся в новую колоду *справа*; карты с *меньшим* значением кладутся в новую колоду *слева*.
4. Проделать эту процедуру для каждой из двух только что созданных колод.
5. Объединить левую колоду, опорную карту и правую колоду, чтобы получить отсортированную колоду (рис. 5.1).

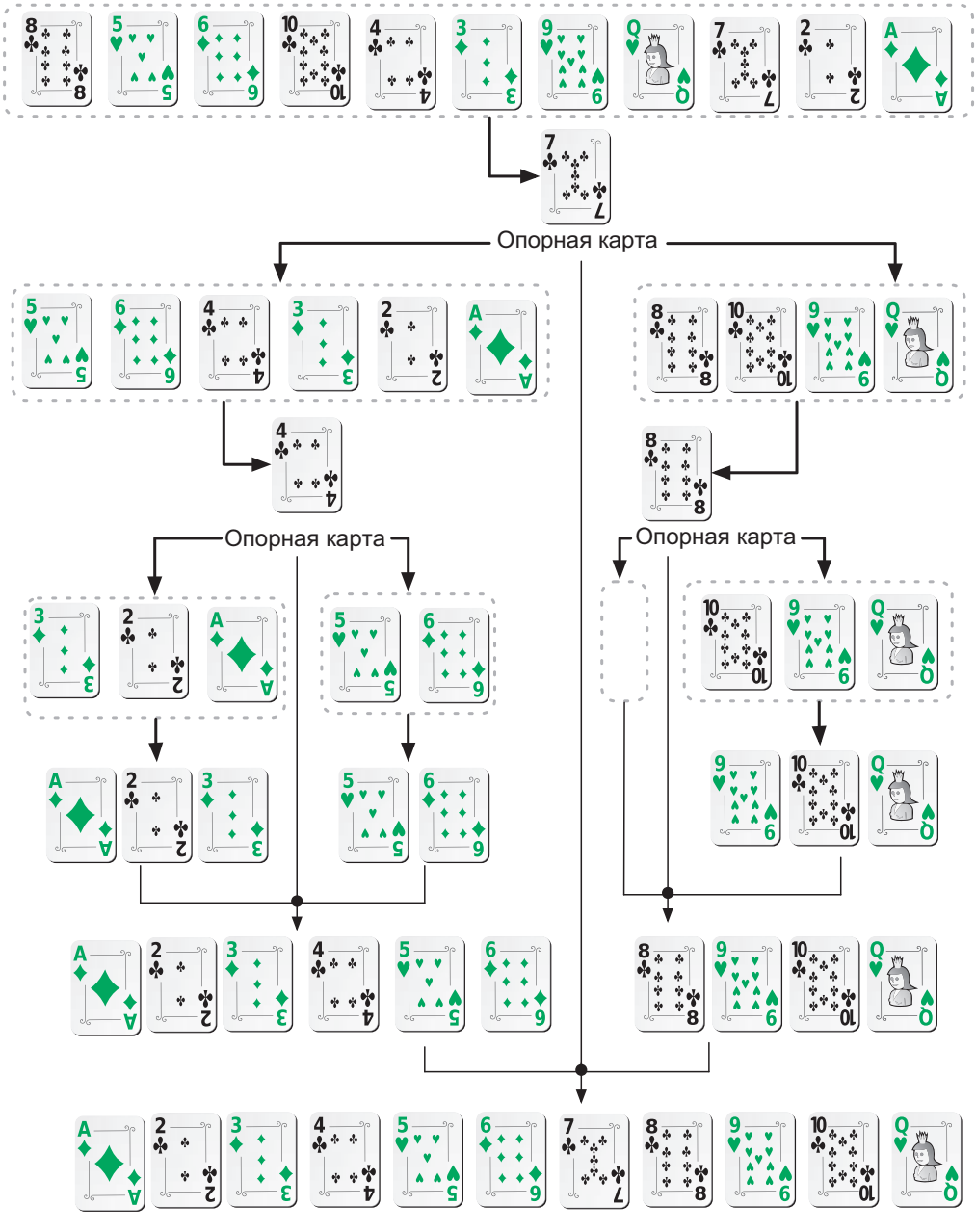


Рис. 5.1. Пример выполнения быстрой сортировки

Перетасуйте колоду карт и проделайте описанные шаги. Это поможет вам опробовать на практике алгоритм быстрой сортировки, а заодно укрепит ваше понимание рекурсии.

Теперь вы готовы решать большинство задач, связанных с сортировкой. Здесь мы осветили не все алгоритмы сортировки, так что просто помните, что их гораздо больше и каждый из них соответствует конкретным задачам.

5.2. Поиск

Поиск определенной информации в памяти является ключевой операцией в вычислениях. Программисту очень важно владеть алгоритмами поиска. Самый простой из них — *последовательный поиск*: вы просматриваете все элементы один за другим, пока не будет найден нужный; как вариант, вы можете проверить *все* элементы, чтобы понять, что искомого среди них нет.

Легко заметить, что последовательный поиск имеет сложность $O(n)$, где n — это общее количество элементов в пространстве поиска. Но на случай, когда элементы хорошо структурированы, есть более эффективные алгоритмы. В разделе «Структуры» предыдущей главы мы убедились, что извлечение данных, представленных в формате сбалансированного двоичного дерева поиска, стоит всего $O(\log n)$.

Если ваши элементы хранятся в сортированном массиве, то их можно отыскать за такое же время, $O(\log n)$, посредством *двоичного поиска*. Этот алгоритм на каждом шаге отбрасывает половину пространства поиска:

```
function binary_search(items, key)
    if not items
        return NULL
    i ← items.length / 2
```

```
if key = items[i]
    return items[i]
if key > items[i]
    sliced ← items.slice(i+1, items.length)
else
    sliced ← items.slice(0, i-1)
return binary_search(sliced, key)
```

На каждом шаге алгоритм `binary_search` выполняет постоянное число операций и отбрасывает *половину* входных данных. Это означает, что для n элементов пространство поиска сведется к нулю за $\log_2 n$ шагов. Поскольку на каждом шаге выполняется постоянное количество операций, алгоритм имеет сложность $O(\log n)$. Вы можете выполнять поиск среди миллиона или триллиона элементов, и этот алгоритм по-прежнему будет показывать хорошую производительность.

Впрочем, существуют еще более эффективные алгоритмы. Если элементы хранятся в хеш-таблице (см. раздел «Структуры» предыдущей главы), достаточно вычислить хеш-ключ искомого элемента. Этот хеш даст его адрес! Время, необходимое для нахождения элемента, *не меняется* с увеличением пространства поиска. Не имеет значения, ищете вы среди миллионов, миллиардов или триллионов элементов, — количество операций останется постоянным, а значит, процесс имеет временную сложность $O(1)$, он действует почти мгновенно.

5.3. Графы

Мы уже знаем, что графы — гибкая структура данных, которая для хранения информации использует вершины и ребра. Графы широко используются для представления таких данных, как социальные сети (вершины — люди, ребра — дружеские связи), телефонные сети (вершины — телефоны и станции, ребра — линии связи) и многих других.

Поиск в графах

Как найти узел в графе? Если структура графа не предоставляет никакой помощи в навигации, вам придется посетить каждую вершину, пока не обнаружится нужная. Есть два способа сделать это: выполнить обход графа в глубину и в ширину (рис. 5.2).

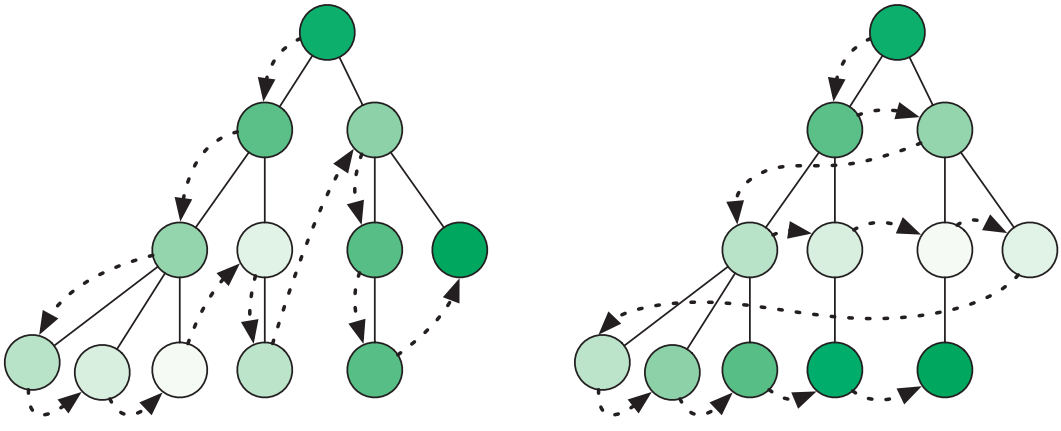


Рис. 5.2. Обход графа в глубину против обхода в ширину

Выполняя поиск в графе в глубину (DFS, depth first search), мы продвигаемся вдоль ребер, уходя все глубже и глубже в граф. Достигнув вершины без ребер, ведущих к каким-либо новым вершинам, мы возвращаемся к предыдущей и продолжаем процесс. Мы используем стек, чтобы запомнить путь обхода графа, помещая туда вершину на время ее исследования и удаляя ее, когда нужно вернуться. Стратегия поиска с возвратом (см. соответствующий раздел главы 3) выполняет обход решений точно так же.

```
function DFS(start_node, key)
    next_nodes ← Stack.new()
    seen_nodes ← Set.new()

    next_nodes.push(start_node)
    seen_nodes.add(start_node)
```

```
    while not next_nodes.empty
        node ← next_nodes.pop()
        if node.key = key
            return node
        for n in node.connected_nodes
            if not n in seen_nodes
                next_nodes.push(n)
                seen_nodes.add(n)
    return NULL
```

Если обход графа вглубь не кажется приемлемым решением, можно попробовать обход в ширину (BFS, breadth first search). В этом случае обход графа выполняется по уровням: сначала соседей начальной вершины, затем соседей его соседей и т. д. Вершины для посещения запоминаются в очереди. Исследуя вершину, мы ставим в очередь ее дочерние вершины, затем определяем следующую исследуемую вершину, извлекая ее из очереди.

```
function BFS(start_node, key)
    next_nodes ← Queue.new()
    seen_nodes ← Set.new()

    next_nodes.enqueue(start_node)
    seen_nodes.add(start_node)

    while not next_nodes.empty
        node ← next_nodes.dequeue()
        if node.key = key
            return node
        for n in node.connected_nodes
            if not n in seen_nodes
                next_nodes.enqueue(n)
                seen_nodes.add(n)
    return NULL
```

Обратите внимание, что алгоритмы `DFS` и `BFS` отличаются только способом хранения следующих исследуемых вершин: в одном случае это очередь, в другом — стек.

Итак, какой подход нам следует использовать? Алгоритм `DFS` более прост в реализации и использует меньше памяти: достаточно хра-

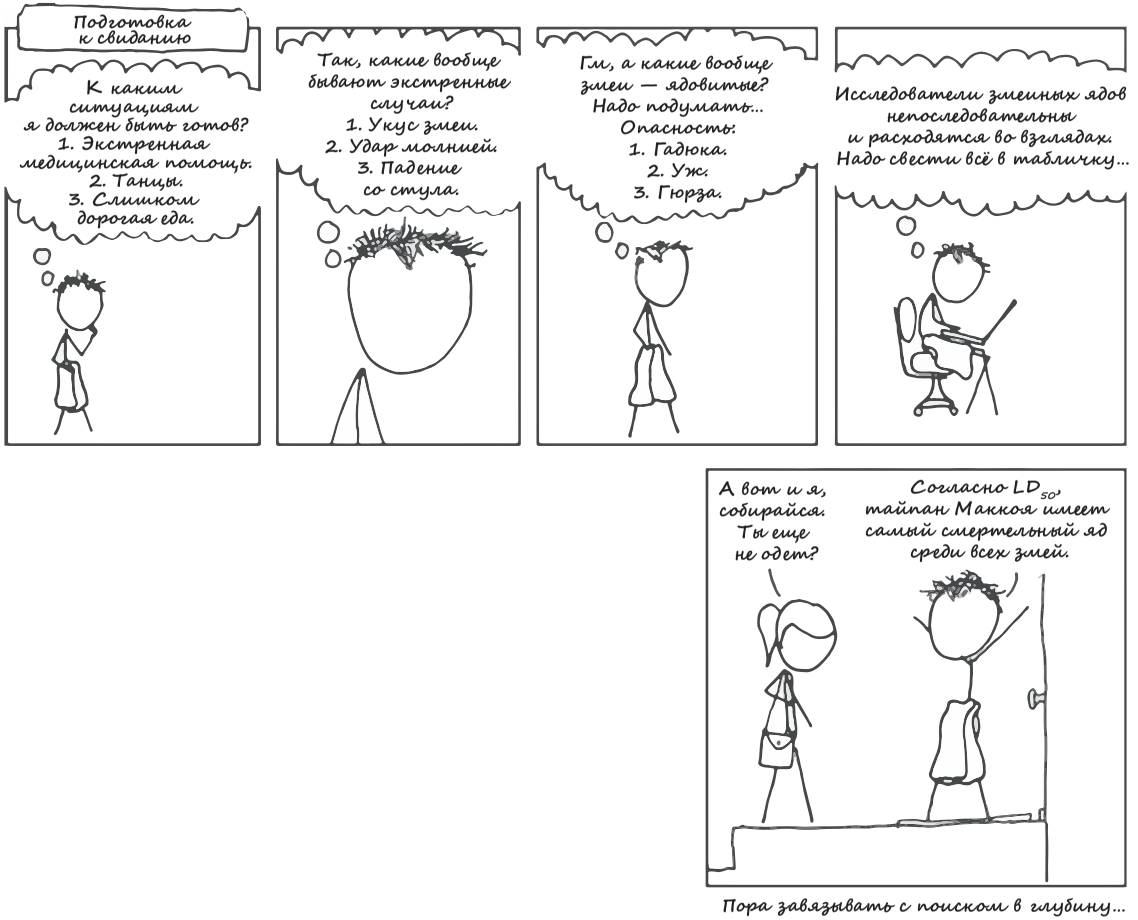


Рис. 5.3. Поиск в графе в глубину[1]

[1] Любезно предоставлено http://xkcd.com.

нить родительские вершины, ведущие к текущей исследуемой вершине. В BFS придется хранить всю границу процесса поиска. Если граф состоит из миллиона вершин, это может оказаться непрактичным.

Когда есть основания предполагать, что искомая вершина не находится многими уровнями ниже начальной, обычно имеет смысл заплатить более высокую стоимость BFS, потому что так вы, скорее всего, закончите поиск быстрее. Если нужно исследовать абсолютно все вершины графа, лучше придерживаться алгоритма DFS из-за его простой реализации и меньшего объема потребляемой памяти.

Рис. 5.3 показывает, что выбор неправильного метода обхода может иметь страшные последствия.

Раскраска графов

Задачи раскраски графов возникают, когда есть фиксированное число «красок» (либо любой другой набор меток) и вы должны назначить «цвет» каждой вершине в графе. Вершины, которые соединены ребром, не могут иметь одинаковый «цвет». В качестве примера давайте рассмотрим следующую задачу.

> **Помехи**  Дана карта вышек сотовой связи и районов обслуживания. Вышки в смежных районах должны работать на разных частотах для предотвращения помех. Имеется четыре частоты на выбор. Какую частоту вы назначите каждой вышке?

Первый шаг состоит в моделировании задачи при помощи графа. Вышки являются вершинами в графе. Если две из них расположены настолько близко, что вызывают помехи, соединяем их ребром. Каждая частота имеет свой цвет.

Как назначить частоты приемлемым способом? Можно ли найти решение, которое использует всего три цвета? Или два? Определение

минимально возможного количества цветов на самом деле является NP-полной задачей — для этого подходят только экспоненциальные алгоритмы.

Мы не покажем алгоритм для решения данной задачи. Используйте то, чему вы научились к настоящему моменту, и попробуйте решить задачу самостоятельно. Это можно сделать на сайте UVA[1] с онлайн-экспертом, который протестирует предложенное вами решение, выполнит ваш программный код и сообщит, работоспособен ли он. Если с кодом окажется все в порядке, эксперт также оценит время выполнения вашего кода в сравнении с тем, что написали другие люди. Дерзайте! Продумайте алгоритмы и стратегии решения данной задачи и испытайте их. Чтение книги может лишь подвести вас к решению. Взаимодействие с онлайновым экспертом даст вам практический опыт, необходимый для того, чтобы стать отличным программистом.

Поиск путей в графе

Поиск кратчайшего пути между узлами является самой известной графовой задачей. Системы навигации GPS проводят поиск в графе улиц и перекрестков для вычисления маршрута. Некоторые из них даже используют данные дорожного движения с целью увеличения веса ребер, представляющих улицы, где образовался затор.

Для поиска кратчайшего пути вполне можно использовать стратегии BFS и DFS, но это плохая идея. Одним из хорошо известных и очень эффективных способов поиска кратчайшего пути является *алгоритм Дейкстры*. В отличие от BFS, для запоминания просматриваемых вершин алгоритм Дейкстры использует очередь с приоритетом. Когда исследуются новые вершины, их связи добавляются в эту очередь. Приоритетом вершины является вес ребер, которые приводят ее в стартовую вершину. Благодаря этому следующая ис-

[1] Задача раскраски графа на сайте онлайновых экспертов UVA: https://code.energy/uva-graph-coloring.

следуемая вершина всегда оказывается самой близкой к месту, откуда мы начали.

Известны случаи, когда алгоритм Дейкстры зацикливается, не в силах найти конечную вершину. Процесс поиска может быть обманут *отрицательным циклом*, который приводит к бесконечному исследованию вершин. Отрицательный цикл — это путь в графе, чье начало и конец приходятся на одну вершину с весом ребер на пути, в сумме дающим отрицательное значение. Если вы ищете кратчайший путь в графе, где ребра могут иметь отрицательный вес, будьте начеку.

А что, если граф, в котором вы ищете, огромен? Для ускорения можно использовать *двунаправленный поиск*. Два процесса поиска выполняются одновременно: один начинает со стартовой вершины, другой — с конечной. Когда оказывается, что вершина, обнаруженная в одном пространстве поиска, также присутствует в другом, это значит, что у нас есть путь. Пространство поиска в таком случае вдвое меньше, чем в однонаправленном поиске. Посмотрите на рис. 5.4: серое пространство меньше белой области.

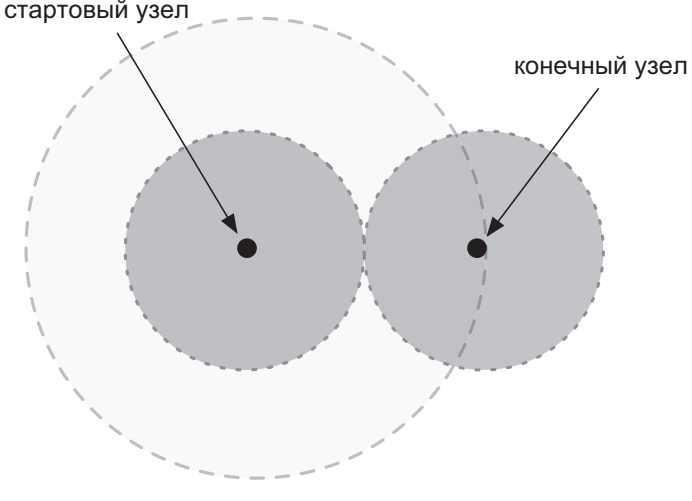


Рис. 5.4. Пространства однонаправленного и двунаправленного поиска

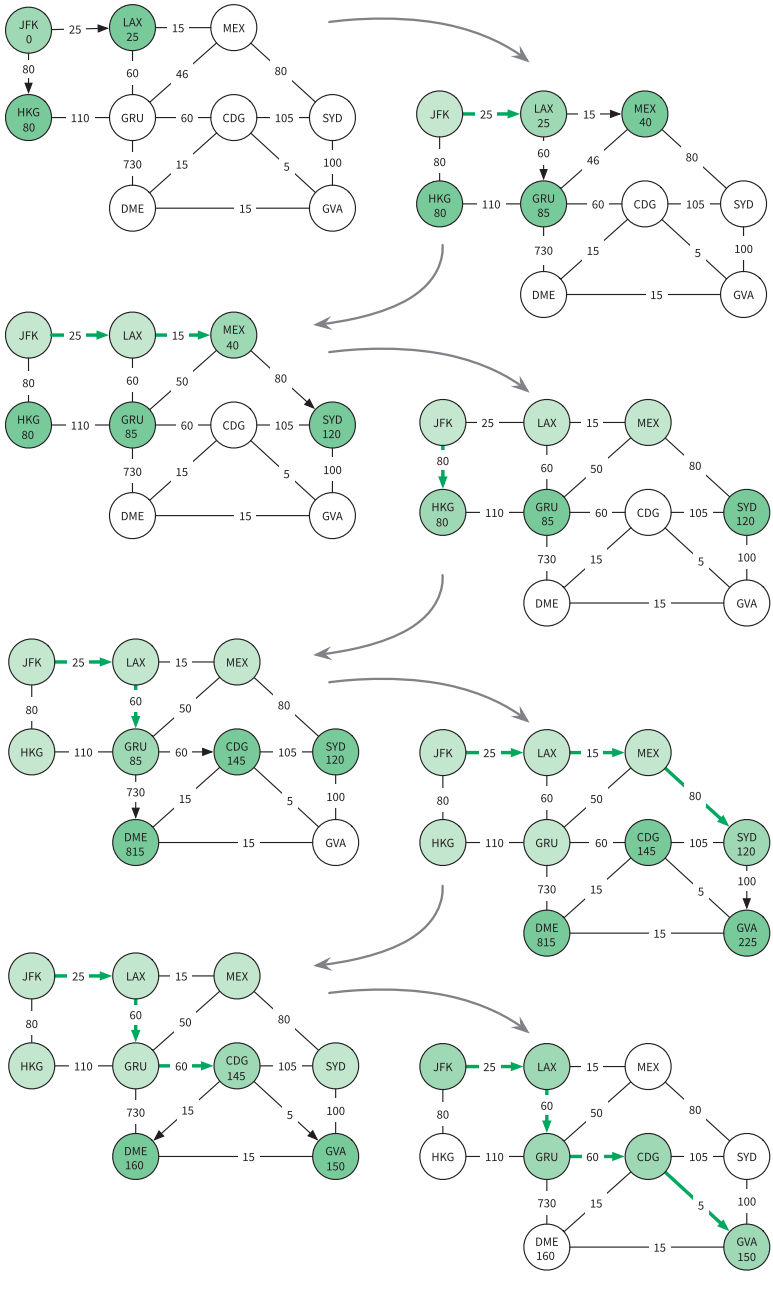


Рис. 5.5. Поиск кратчайшего маршрута от аэропорта JFK до аэропорта GVA при помощи алгоритма Дейкстры

PageRank

Вы когда-нибудь задавались вопросом, как поисковой системе Google удается анализировать миллиарды веб-страниц и показывать вам самые подходящие? В этом процессе задействовано множество алгоритмов, но самым важным является *алгоритм PageRank*.

Прежде чем основать компанию Google, Сергей Брин и Ларри Пейдж работали научными сотрудниками в области computer science в Стэнфордском университете и занимались исследованием графовых алгоритмов. Они смоделировали Всемирную паутину в виде графа: веб-страницы — это вершины, и связи между ними — ребра.

Они решили, что если веб-страница получает много связей от других важных страниц, то она тоже должна быть важной. Опираясь на эту идею, они создали алгоритм PageRank. Он выполняется в несколько заходов. Вначале каждая веб-страница в графе имеет то же количество единиц значимости, что и остальные. После каждого захода она распределяет свои единицы среди страниц, ссылки на которые на ней размещены. Этот процесс повторяется до тех пор, пока все значения не стабилизируются. Стабилизированная оценка каждой страницы называется ее рангом, отсюда и название — PageRank (англ. «ранг страницы»). Используя этот алгоритм для определения важности веб-страниц, поисковая система Google быстро заняла доминирующую позицию среди других аналогичных сервисов.

Алгоритм PageRank применим и к другим типам графов. Например, мы можем смоделировать пользователей сети Twitter на графе, а затем вычислить ранг каждого. Как вы считаете, будут ли пользователи с более высоким рангом известными людьми?

5.4. Исследование операций

Во время Второй мировой войны британская армия столкнулась с необходимостью оптимизировать принятие стратегических решений, чтобы повысить действенность операций. Было разработано

большое количество аналитических инструментов для выявления наилучшего способа координации военных действий.

Эта практическая дисциплина получила название *исследование операций*. Она позволила усовершенствовать британскую систему радаров дальнего обнаружения и помогла Соединенному Королевству лучше управлять людскими и материальными ресурсами. Во время войны сотни британцев участвовали в исследовании операций. В дальнейшем для оптимизации процессов в торгово-промышленной деятельности были применены новые идеи. Исследование операций включает в себя определение целевого показателя, который подлежит оптимизации, то есть *максимизации* или *минимизации*. Эта дисциплина позволяет максимизировать такие целевые показатели, как урожай, прибыль или производительность, и минимизировать убытки, риск или стоимость.

Например, исследование операций используется авиакомпаниями для оптимизации графиков полетов. Точные корректировки в планировании распределения трудовых ресурсов и оборудования могут сэкономить миллионы долларов. Еще один пример касается нефтеперерабатывающих заводов, где определение оптимальных пропорций сырья в смеси может рассматриваться как задача исследования операций.

Задачи линейной оптимизации

Задачи, где целевой показатель и ограничения можно смоделировать с использованием линейных уравнений[1], называются задачами линейной оптимизации. Давайте посмотрим, как решаются эти задачи.

> **Умная меблировка**  В вашем офисе не хватает каталожных шкафов. Шкаф X стоит 10 долларов, занимает 6 квадратных футов и содержит 8 кубических футов папок. Шкаф Y стоит

[1] Формально полиномов 1-й степени. Они не имеют квадратов (впрочем, и любых других степеней), а их переменные могут умножаться лишь на константы.

20 долларов, занимает 8 квадратных футов и содержит 12 кубических футов папок. У вас есть 140 долларов, и вы можете использовать под шкафы до 72 квадратных футов площади офиса. Какие шкафы следует приобрести, чтобы максимизировать емкость хранения?

Прежде всего определим переменные нашей задачи. Мы хотим найти количество шкафов каждого типа, которые следует приобрести, поэтому:

- x — количество шкафов модели X;
- y — количество шкафов модели Y.

Мы хотим максимизировать емкость хранения. Дадим емкости хранения имя z и смоделируем это значение как функцию от x и y:

- $z = 8x + 12y$.

Теперь выберем значения x и y, которые дадут максимальное значение z. При этом мы должны соблюсти ограничение по бюджету (то есть уложиться в 140 долларов) и по площади (она должна быть меньше 72 квадратных футов). Смоделируем эти ограничения:

- $10x + 20y \leq 140$ (ограничение по бюджету);
- $6x + 8y \leq 72$ (ограничение по площади);
- $x \geq 0, y \geq 0$ (нельзя купить отрицательное количество шкафов).

Как бы вы решили эту задачу? Покупка максимального количества шкафов с наилучшим соотношением хранение/площадь не является правильным решением, потому что пространство под установку шкафов ограниченно. Можно пойти по пути полного перебора: написать программу, вычисляющую z для всех возможных x и y, и получить пару, дающую оптимальное z. Это решение годится для простых задач, но оно невыполнимо при большом количестве переменных.

Оказывается, что решать задачи линейной оптимизации вроде этой можно и без программирования. Нужно просто использовать пра-

вильный инструмент для работы: *симплекс-метод*. Он очень эффективно справляется с задачами линейной оптимизации. Симплекс-метод помогает целым отраслям решать сложные проблемы, начиная с 1960-х годов. Когда перед вами встанет такая задача, не изобретайте колесо, просто возьмите готовый симплексный решатель.

Симплексные решатели требуют указать функцию для максимизации (или минимизации) и уравнения, моделирующие ограничения. Решатель сделает все остальное. В данной задаче максимальное значение z достигается при $x = 8$ и $y = 3$.

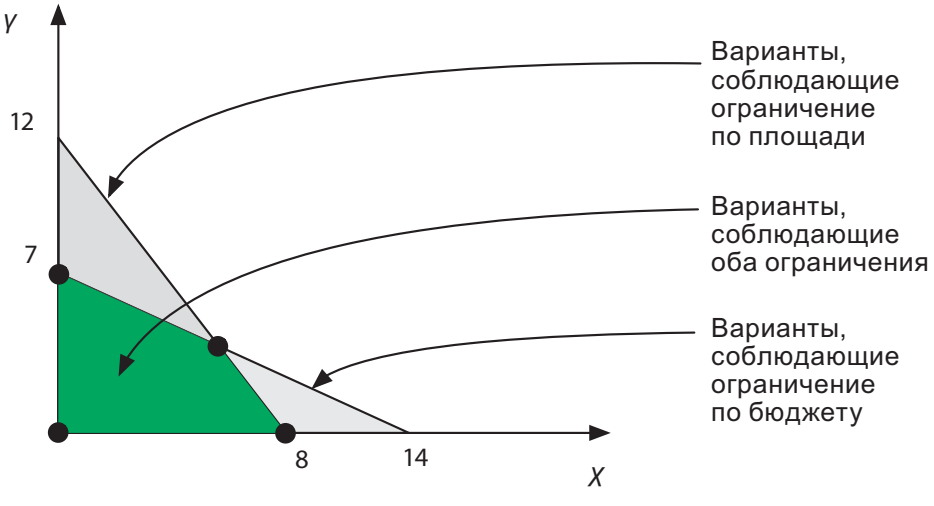


Рис. 5.6. Значения x и y, удовлетворяющие ограничениям задачи

Симплекс-метод отыскивает оптимальное значение в пространстве приемлемых решений. Чтобы понять механику его работы, представим все возможные значения x и y на двумерной плоскости (рис. 5.6). Ограничения по бюджету и площади представлены на графике линиями.

Обратите внимание, что пространство всех возможных решений является замкнутой областью на графике. Доказано, что оптимальным решением линейной задачи должна быть угловая точка замкнутой области — та, где пересекаются линии, представляющие ограниче-

ния. Симплекс проверяет угловые точки и вычисляет, которая из них оптимизирует *z*. Отнюдь не просто визуализировать этот процесс в задачах линейной оптимизации, имеющих более двух переменных, но математический принцип везде работает одинаково.

Задачи о максимальном потоке в Сети

Многие задачи, касающиеся сетей и потоков, можно сформулировать с точки зрения линейных уравнений и, следовательно, решить при помощи симплекс-метода. Например, во время холодной войны армия США вычисляла маршруты пополнения материально-технических запасов, которые Советский Союз мог использовать в Восточной Европе (рис. 5.7).

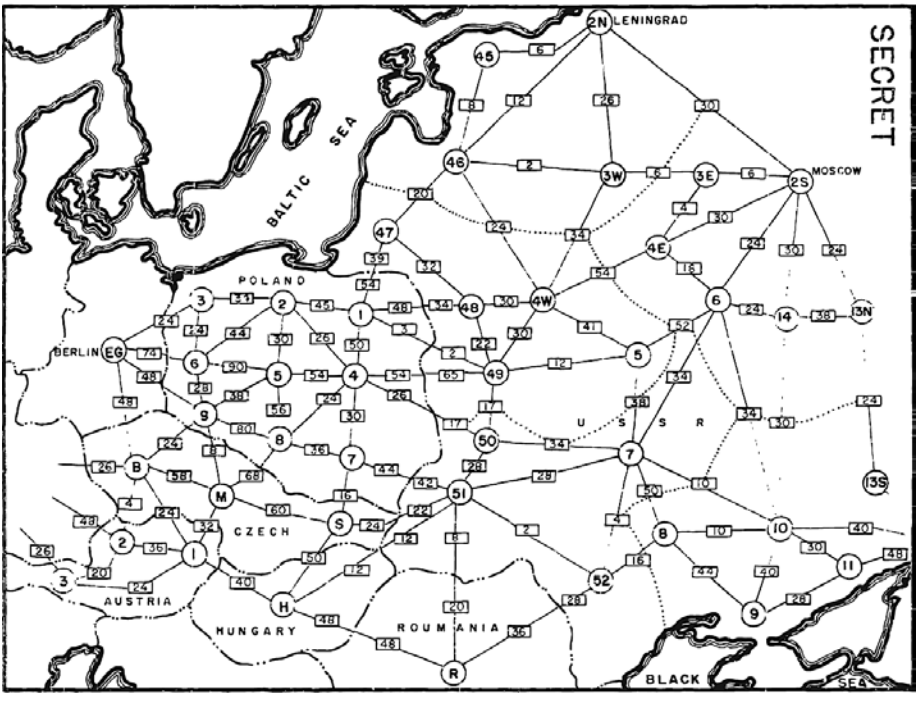


Рис. 5.7. Рассекреченный военный отчет 1955 г., показывающий пропускную способность советской сети железных дорог

> **Сеть снабжения**  Сеть железных дорог представлена линиями, которые соединяют города. Каждая имеет максимальную пропускную способность — самый большой ежедневный поток грузов. Какой объем можно перевезти из заданного производящего города в заданный потребляющий город?

Чтобы смоделировать задачу с линейными уравнениями, каждой железной дороге нужно назначить переменную, представляющую объем грузов, который она сможет перевезти. Ограничения следующие: ни одна железная дорога не может перевезти больше своей пропускной способности; входящий поток грузов должен быть эквивалентен исходящему во всех населенных пунктах, кроме производящего и потребляющего городов. Затем нужно подобрать такие значения для переменных, которые позволят доставить в получающий город максимум грузов.

Мы не будем подробно расписывать, как отобразить эту задачу в линейной форме. Наша цель здесь состоит только в том, чтобы донести мысль, что многие задачи оптимизации с привлечением графов, стоимости и потоков можно легко решить существующими реализациями симплекс-метода. В Сети есть вся необходимая документация. Смотрите в оба и не изобретайте колеса.

Подведем итоги

Мы показали несколько хорошо известных алгоритмов и методов решения самых разнообразных задач. Первым делом, приступая к решению новой задачи, всегда старайтесь найти готовый алгоритм или метод.

Существует большое количество важных алгоритмов, которые мы не смогли включить в эту главу. Например, имеются поисковые алгоритмы, более продвинутые, чем алгоритм Дейкстры (такие как, `A*`[1]), алгоритмы, оценивающие подобие двух слов (расстояние редакти-

[1] Читается как «А-звезда» или «А-стар». — *Примеч. ред.*

рования Левенштейна), алгоритмы машинного обучения и многие другие...

Полезные материалы

- *Комен Т., Лейзерсон Ч. И., Ривест Р. Л., Штайн К.* Алгоритмы. Построение и анализ. — М.: «Вильямс», 2016.
- Алгоритмы (Algorithms, Sedgewick, см. https://code.energy/sedgewick).
- Простая модель линейного программирования (Simple Linear Programming Model, Katie Pease, см. https://code.energy/katie).

Глава 6

БАЗЫ ДАННЫХ

> Хотя я известен прежде всего своими работами в области баз данных, мои фундаментальные умения лежат в области архитектуры: это анализ технических требований и построение простых, но изящных решений.
>
> *Чарльз Бэкмен*

Управлять колоссальными объемами данных в компьютерных системах очень сложно, но часто жизненно необходимо. Биологи хранят и получают последовательности ДНК и связанные с ними структуры белка. Facebook управляет контентом, созданным миллиардами людей. Amazon отслеживает свои продажи, запасы товаров и логистику.

Как хранить все эти большие, постоянно изменяющиеся массивы данных на дисках? Как дать разным агентам возможность одновременно получать, редактировать и добавлять данные? Вместо того чтобы самостоятельно решать эти задачи, мы используем **с**истему **у**правления **б**азами **д**анных (СУБД) — специальный компонент программного обеспечения для управления базами данных. СУБД организует и хранит информацию, она обеспечивает возможность

доступа и изменения этой информации. Прочитав главу 6, вы научитесь:


понимать *реляционную* модель большинства баз данных;


использовать гибкость *нереляционных* баз данных;


координировать работу компьютеров и *распределять* между ними ваши данные;


лучше соотносить информацию с картами при помощи *географических* баз данных;


обмениваться данными с различными системами, используя прием *сериализации* данных.

Реляционные базы данных распространены шире, но нереляционные нередко оказываются проще и эффективнее. Базы данных очень разнообразны, и сделать выбор между ними бывает непросто. В этой главе сделан общий обзор различных типов современных СУБД.

Упростив доступ к данным с помощью СУБД, им можно найти хорошее применение. Из невзрачного каменистого клочка земли шахтер способен добыть ценные минералы и металлы. Аналогично мы нередко можем извлечь ценную информацию из имеющихся у нас массивов информации. Этот процесс называется *глубинным анализом данных*.

Например, большая сеть бакалейных магазинов проанализировала свои данные о продажах и обнаружила, что ее самые склонные к расходам покупатели часто берут сорт сыра с уровнем продаж ниже 200 пунктов. Обычно продукты, продающиеся настолько плохо, снимают с продаж. Анализ данных побудил менеджеров не только оставить сыр, но и выставить его на виду. Это понравилось лучшим покупателям, и они стали возвращаться чаще. Чтобы суметь сделать такой умный ход, сети бакалейных магазинов потребовалось хорошо организовать свои данные в СУБД.

6.1. Реляционная модель

Появление *реляционной модели* в конце 1960-х стало огромным рывком в управлении информацией. Реляционные базы данных помогают избегать дублирования информации и противоречий. Большинство СУБД, которые используются сегодня, являются реляционными.

В реляционной модели данные разделены на *таблицы*. Таблица — это нечто вроде матрицы или листа Excel. Каждая запись в ней является *строкой*. *Столбцы* — различные свойства записей. Обычно столбцы определяют типы хранимых в них данных. Столбцы могут также определять другие ограничения: обязательно ли строка должна иметь в этом столбце значение, должно ли оно быть уникальным по всем строкам в таблице и т. д.

Столбцы обычно называются *полями*. Если столбец допускает только целые числа, то мы говорим, что это *целочисленное поле*. Разные таблицы используют разные типы полей. Организация таблицы базы данных задается ее полями и ограничениями, которые те налагают. Такая комбинация полей и ограничений называется *схемой* таблицы.

Все записи — строки, и СУБД не примет новую строку, если та нарушает схему таблицы. В этом состоит большой недостаток реляционной модели. Когда характеристики данных значительно варьируются, подгонка их к фиксированной схеме может создать много хлопот. Но если вы работаете с данными однородной структуры, то фиксированная схема гарантирует, что все они будут допустимыми.

Отношения

Представим базу данных счетов-фактур, которая содержится в единственной таблице. По каждому счету мы должны хранить информацию о заказе и клиенте. Когда в базе данных хранится несколько счетов, относящихся к одному клиенту, информация повторяется (рис. 6.1).

Дата	Имя клиента	Телефон	Итого
17.02.2017	Бобби Тейблс	997-10-09	$93.37
18.02.2017	Элен Робертс	101-99-73	$77.57
20.02.2017	Бобби Тейблс	997-10-09	$99.73
22.02.2017	Бобби Тейблс	991-10-09	$12.01

Рис. 6.1. Данные о счетах-фактурах, хранящиеся в единственной таблице

Повторяющейся информацией трудно управлять, и ее сложно обновлять. Чтобы избежать таких сложностей, реляционная модель разбивает связанную информацию на разные таблицы. Например, разделим наши данные о счетах-фактурах на две таблицы — «Заказы» и «Клиенты» — и сделаем так, чтобы каждая строка в первой таблице ссылалась на строку во второй (рис. 6.2).

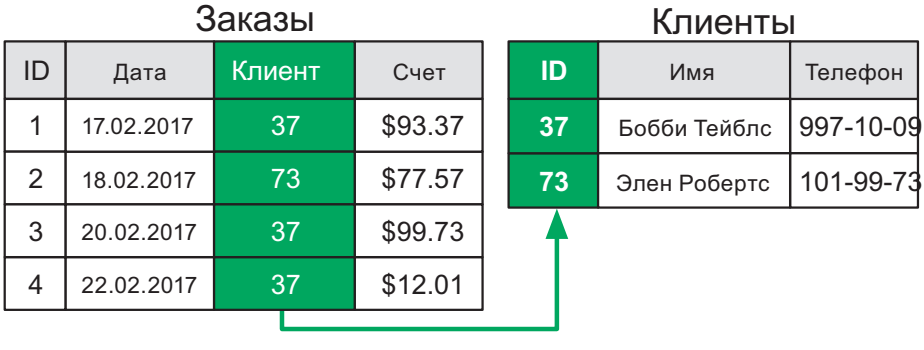


Рис. 6.2. Связи между строками позволяют избежать дублирования данных

За счет связывания между собой данных из различных таблиц один клиент может быть частью многих заказов, а дублирования данных при этом не случится. Для поддержки связей каждая таблица имеет специальное идентификационное поле, или `ID`. Мы используем зна-

чения `ID` для ссылки на конкретную строку в таблице. Эти значения должны быть уникальными. Поле `ID` таблицы также называется ее *первичным ключом*. Поле со ссылками на `ID` других строк называется *внешним ключом*.

Рис. 6.3. Специалисты, награжденные премией Тьюринга

При помощи первичных и внешних ключей мы можем создать сложные отношения между отдельными наборами данных. Например,

таблицы на рис. 6.3 хранят информацию о лауреатах премии Тьюринга[1].

Связь между специалистами в области computer science и премиями не настолько проста, как между клиентами и заказами в предыдущем примере. Премия может быть поделена между двумя специалистами, и нигде не сказано, что ученый имеет право получить ее всего один раз. Поэтому мы используем таблицу «Лауреаты», только чтобы хранить связи между специалистами и премиями.

Когда база данных организована таким образом, что не содержит повторяющейся информации, говорят, что она *нормализована*. Процесс преобразования базы данных с дубликатами в базу данных без таковых называется *нормализацией*.

Миграция схемы

Когда приложение растет и добавляются новые свойства, маловероятно, что его структура базы данных (схема всех его таблиц) останется прежней. В этом случае приходится изменять структуру, и тогда создают сценарий, или скрипт, *миграции схемы*. Он автоматически обновляет схему и преобразует существующие данные. Как правило, такие сценарии могут также отменять производимые ими изменения. Это позволяет легко восстановить структуру базы данных, соответствующую предыдущей рабочей версии программы.

В большинстве СУБД существуют готовые инструменты, которые помогают создавать и применять сценарии миграции схемы, а также возвращать базу данных к прежнему состоянию. Некоторые большие системы претерпевают сотни миграций за год, так что эти инструменты играют незаменимую роль. Если не делать миграцию схемы, а вручную вносить изменения в базу данных, ее потом будет трудно вернуть к конкретной рабочей версии. Такой «ручной» подход не гарантирует совместимости между локальными базами данных

[1] Премия Тьюринга — самая престижная премия в области информатики. Ее премиальный фонд составляет один миллион долларов.

различных разработчиков ПО. Подобные проблемы часто случаются в больших программных проектах, где наплевательски относятся к работе с базами данных.

SQL

Почти каждая реляционная СУБД поддерживает язык запросов под названием SQL[1]. Мы не ставим перед собой задачи дать вам всесторонний курс по SQL, но в этой книге вы получите общее представление о том, как он работает. Важно разбираться в SQL хотя бы поверхностно, даже если вы непосредственно с ним не работаете. SQL-запрос — это команда, сообщающая, какие данные должны быть получены:

```
SELECT <field name> [, <field name>, <field name>,…]
FROM <table name>
WHERE <condition>;
```

Элементы, идущие после `SELECT`, — это поля, которые нужно получить. Чтобы получить все поля в таблице, можно написать: `SELECT *`. В базе данных может быть несколько таблиц, поэтому `FROM` уточняет, какую таблицу вы запрашиваете. После команды `WHERE` вы устанавливаете критерии отбора строк. Для перечисления многочисленных условий можно использовать булеву логику. Следующий запрос получает все поля из таблицы customers («клиенты»), фильтруя строки по полям name («имя») и age («возраст»):

```
SELECT * FROM customers
WHERE age > 21 AND name = "John";
```

Вы можете послать запрос: `SELECT * FROM customers`, без оператора `WHERE`. СУБД выдаст вам список всех клиентов. Помимо этого, имеются другие операторы запросов, о которых вам следует знать: оператор `ORDER BY` сортирует результаты по указанному полю (по-

[1] В английском языке аббревиатура SQL чаще произносится, как «*сиквел*», а устная форма «*эс-кью-эл*» не считается неправильной.

лям), а GROUP BY поможет выполнить группировку и получить агрегированные результаты для групп. Например, при наличии таблицы customers («клиенты») с полями country («страна») и age («возраст»), вы можете выполнить такой запрос:

```
SELECT country, AVG(age)
FROM customers
GROUP BY country
ORDER BY country;
```

Он вернет сортированный список стран, где проживают ваши клиенты, вместе со средним возрастом клиентов по каждой стране. SQL предоставляет и другие агрегатные функции. Например, замените AVG(age) на MAX(age), и вы получите возраст самого старого клиента в каждой стране.

Иногда бывает нужно изучить информацию из строки и строк, с которыми она связана. Представьте, что у вас есть таблица с заказами и таблица с клиентами. Таблица orders имеет внешний ключ для ссылки на клиентов. Чтобы найти информацию о клиентах, сделавших дорогостоящие заказы, придется выбрать данные из обеих таблиц. Но вам не нужно запрашивать их по отдельности и сопоставлять записи самостоятельно. Для этого в языке SQL имеется специальная команда:

```
SELECT DISTINCT customers.name, customers.phone
FROM customers
JOIN orders ON orders.customer = customers.id
WHERE orders.amount > 100.00;
```

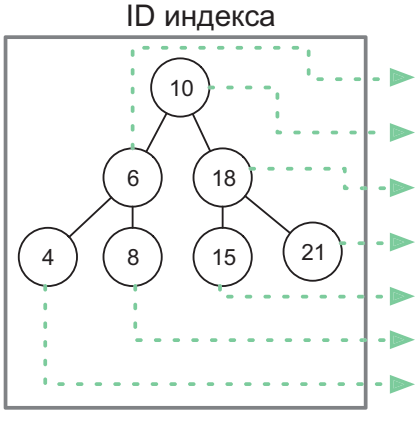
Этот запрос вернет имена и телефонные номера клиентов, сделавших заказы на сумму более 100 долларов. Команда SELECT DISTINCT заставляет СУБД вернуть каждого клиента только один раз. JOIN позволяет делать очень гибкие запросы[1], но эта гибкость имеет свою цену. Соединения обходятся дорого. Базе данных придется рассмотреть все сочетания строк из таблиц, которые вы объединяете в сво-

[1] Существует несколько способов выполнить операцию JOIN — см. https://code.energy/joins.

ем запросе. Администратор базы данных должен всегда принимать во внимание произведение числа строк объединяемых таблиц. Для очень больших таблиц соединения становятся невыполнимыми. Оператор `JOIN` — это самый мощный инструмент и одновременно главная слабость реляционных баз данных.

Индексация

Чтобы от первичного ключа таблицы была польза, необходимо иметь возможность быстро получить запись по `ID`. Для этого СУБД строит вспомогательный *индекс*, содержащий `ID` строк, и соответствующие им адреса в памяти (рис. 6.4). По сути, индекс — это сбалансированное двоичное дерево поиска (см. раздел «Структуры» предыдущей главы). Каждая строка в таблице соответствует узлу в дереве.



ID	Имя	Дата рождения	Нацио-нальность
06	Барбара Лисков	07.11.1939	US
10	Петер Наур	25.10.1928	US
18	Ади Шамир	06.07.1952	IL
21	Ада Лавлейс	10.12.1815	UK
15	Бил Гейтс	28.10.1955	US
08	Алан Тьюринг	23.06.1912	UK
04	Деннис Ритчи	09.09.1941	US

Рис. 6.4. Индекс, отображающий значения ID и расположение соответствующих строк

Ключи узлов — это значения в индексируемом поле. Чтобы найти запись с заданным значением, мы ищем его в дереве. Найдя узел, мы получаем адрес, который он хранит, и используем его для выборки

записи. Поиск по двоичному дереву имеет сложность $O(\log n)$, поэтому нахождение записей в больших таблицах выполняется быстро.

Обычно СУБД создает индекс для каждого первичного ключа в базе данных. Но если часто приходится искать записи по другим полям (например, искать клиентов по именам), можно поручить СУБД создать для них дополнительные индексы.

Ограничения уникальности. Индексы часто создаются автоматически для полей, которые имеют ограничение уникальности. При вставке новой строки СУБД должна обследовать всю таблицу, чтобы удостовериться, что ни одно ограничение уникальности не нарушено. Не будь индекса, такая проверка означала бы, что нужно свериться со всеми строками в таблице. При помощи индекса мы можем быстро выполнить поиск и, например, обнаружить, что значение, которое мы пытаемся вставить, уже присутствует. Индексация полей, имеющих ограничение уникальности, необходима для быстрой вставки элементов.

Сортировка. Индексы помогают выбирать строки в порядке сортировки по индексированным полям. Например, если имеется индекс для поля name («имя»), мы можем получить строки, отсортированные по имени, без дополнительных вычислений. Если применить команду `ORDER BY` к полю без индекса, СУБД придется отсортировать данные в памяти, прежде чем выполнить запрос. Многие СУБД могут даже отказаться выполнять запрос, требующий произвести сортировку по неиндексированному полю, если в работу будет вовлечено слишком много строк.

Если вы должны отсортировать строки сначала по стране, а затем по возрасту, наличие индекса в поле age («возраст») или в поле country («страна») не сильно вам поможет. Индекс в country позволяет выбирать строки, отсортированные по стране, но затем вам потребуется вручную сортировать по возрасту элементы, которые имеют одинаковую страну. Когда требуется сортировка по двум полям, используются комбинированные, или *объединенные*, *индексы*. Они индексируют многочисленные поля и не способны помочь искать элементы

быстрее, зато позволяют легко получать данные, отсортированные по нескольким полям.

Производительность. Итак, индексы — это круто: они позволяют делать сверхбыстрые запросы и мгновенно получать доступ к отсортированным данным. Тогда почему у нас нет индексов для *всех* полей в каждой таблице? Проблема в том, что, когда новая запись вставляется в таблицу или удаляется из нее, приходится обновлять все индексы, чтобы отразить это изменение. Если индексов много, то обновление, вставка или удаление строк могут стать в вычислительном плане дорогостоящими операциями (вспомним про балансировку дерева). Более того, индексы занимают ограниченное дисковое пространство.

Вы должны следить за тем, как ваше приложение использует базу данных. СУБД обычно поставляются вместе с инструментами, которые помогают это делать. Они могут «объяснять» запросы, сообщая, какие индексы использовались, а также сколько строк необходимо было последовательно просканировать, чтобы выполнить запрос. Если ваши запросы тратят впустую слишком много времени, последовательно сканируя данные в некоем поле, то добавьте для этого поля индекс и посмотрите, будет ли польза. Например, если вы часто ищете в базе данных людей конкретного возраста, то определение индекса для поля age позволит СУБД сразу отбирать строки, соответствующие конкретному возрасту. Вы сэкономите время, избежав последовательного просмотра базы данных с дальнейшей фильтрацией строк, не соответствующих требуемому возрасту.

Если вы хотите настроить базу данных, чтобы повысить ее производительность, чрезвычайно важно знать, какие индексы стоит сохранять, а какие — отбрасывать. Если доступ к БД главным образом осуществляется в режиме чтения, а обновляется она редко, может иметь смысл создать больше индексов. Плохая индексация — главная причина замедлений в коммерческих системах. Небрежные системные администраторы зачастую не задаются вопросом, как выполняются типичные запросы, — они просто индексируют произвольные поля, которые, по их мнению, будут способствовать производительности. Этого не стоит делать! Воспользуйтесь «объясняющими» инстру-

ментами, чтобы проверить свои запросы и создать индексы только там, где они нужны.

Транзакции

Представим, что скрытный швейцарский банк не ведет учета денежных переводов: его база данных просто хранит баланс счетов. Предположим, что кто-то хочет перечислить деньги со своего счета на счет друга в том же банке. Две операции должны быть выполнены в базе данных банка — денежную сумму нужно вычесть из одного баланса и прибавить к другому.

Сервер БД обычно позволяет многочисленным клиентам читать и записывать данные одновременно — исполнение операций в последовательном режиме сделало бы любую СУБД слишком медленной. Но вот подвох: если кто-то запросит общий баланс всех счетов *после* регистрации вычитания, но *до* соответствующего добавления, то какая-то сумма будет отсутствовать. Или вот вариант похуже: а что, если система окажется обесточена между этими двумя операциями? Когда сервер снова заработает, будет трудно выяснить причину расхождения в данных.

Нам нужны способы, которыми СУБД выполняла бы либо *все* изменения, входящие в многосоставную операцию, либо сохраняла данные неизменными. С этой целью системы баз данных поддерживают *транзакции*. Транзакция — список операций, которые должны быть выполнены *атомарно*[1]. Транзакции упрощают жизнь программиста: вместо него за обеспечение непротиворечивости данных отвечает СУБД. От программиста только требуется обертывать зависимые операции в соответствующие команды:

```
START TRANSACTION;
UPDATE vault SET balance = balance + 50 WHERE id=2;
UPDATE vault SET balance = balance – 50 WHERE id=1;
COMMIT;
```

[1] Атомарные операции выполняются одноэтапно: они не могут быть выполнены наполовину.

Запомните: выполнение многосоставных обновлений без транзакций рано или поздно создаст беспорядочные, непредсказуемые и трудные в обнаружении противоречия в ваших данных.

6.2. Нереляционная модель

Реляционные базы данных замечательны, однако у них есть некоторые недостатки. По мере усложнения приложения в его реляционную базу данных приходится добавлять все больше таблиц. Запросы становятся все менее понятными. И, главное, все чаще приходится прибегать к соединениям (`JOIN`), требующим большого объема вычислений и создающим в системе узкие места.

Рис. 6.5[1]

Нереляционная модель не использует табличные связи. Она почти никогда не требует объединять информацию из нескольких записей. Поскольку нереляционные СУБД используют языки запросов, отличные от SQL, они также называются *базами данных NoSQL*.

Документные хранилища

Наиболее известным типом баз данных NoSQL являются *документные хранилища*. В них записи хранятся в том виде, в котором они

[1] Любезно предоставлено http://geek-and-poke.com.

необходимы приложению. Рис. 6.6, приведенный ниже, сравнивает табличный и документный способы хранения постов в блоге.

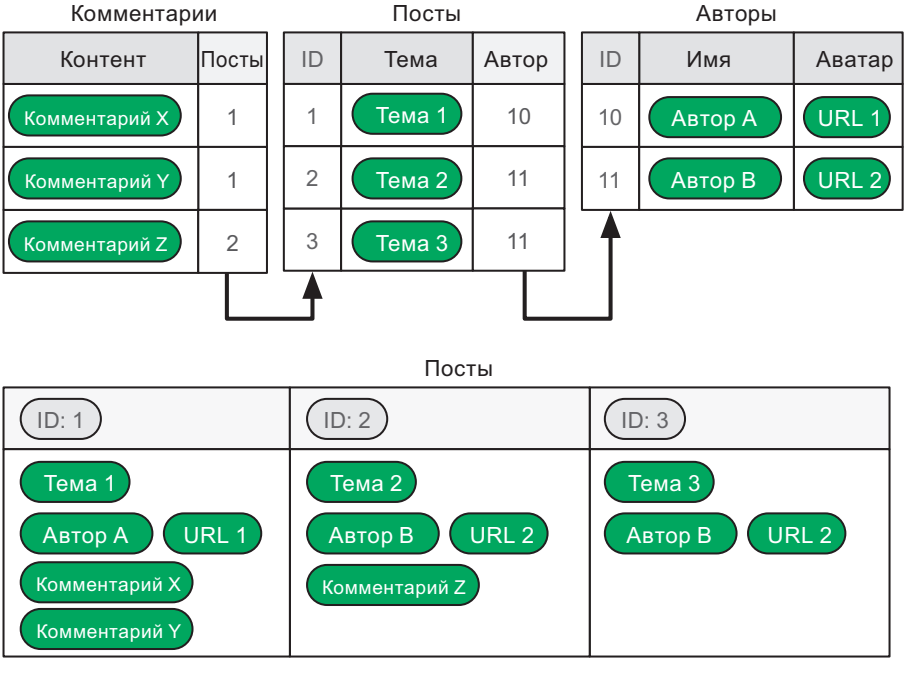


Рис. 6.6. Данные в реляционной модели (вверху) и данные в NoSQL (внизу)

Заметили, что все данные о сообщении копируются в соответствующую ему запись? Нереляционная модель *предполагает* возможность дублирования информации при необходимости. Однако дублированные данные сложно своевременно обновлять и поддерживать их непротиворечивость. С другой стороны, группируя соответствующие данные, документное хранилище может предложить бóльшую гибкость:

- вам не нужно соединять строки;
- можно обойтись без фиксированных схем;
- каждая запись может иметь собственное сочетание полей.

В документных хранилищах вообще нет таблиц и строк. Вместо них есть записи, называемые *документами*. Связанные между собой документы группируются в *коллекцию*.

Документы имеют поле первичного ключа, поэтому их можно связывать друг с другом. Но операции `JOIN` в документных хранилищах неэффективны. Иногда они даже невозможны, в этом случае вам придется следить за связями между документами самостоятельно. И то и другое плохо — если документы имеют общие данные, их приходится дублировать.

Как и реляционные базы данных, базы данных NoSQL создают индексы для полей с первичным ключом. Также можно определять дополнительные индексы для полей, которые часто запрашиваются или сортируются.

Хранилища «ключ — значение»

Хранилище «ключ — значение» — это простейшая форма организованного хранения данных. В основном используется для кэширования. Например, когда некто запрашивает определенную веб-страницу на сервере, тот должен выбрать соответствующие ей данные из БД и использовать их для конструирования HTML-разметки, которую увидит пользователь. В сайтах с высокой посещаемостью, где случаются тысячи параллельных доступов, делать это становится невозможным.

Для решения проблемы мы используем хранилище «ключ — значение» как механизм кэширования. Ключом является требуемый URL-адрес, значением — HTML-разметка соответствующей веб-страницы. В следующий раз, когда кто-то запросит тот же URL-адрес, готовый код HTML просто будет извлечен из хранилища «ключ — значение» через ключ-адрес.

Если вам приходится раз за разом выполнять медленную операцию, всегда приводящую к одному и тому же результату, рассмотрите возможность его кэширования. Вам не обязательно использовать

хранилище «ключ — значение», кэш может содержаться и в базах данных другого типа. Однако когда кэш запрашивается очень часто, система хранилищ данных типа «ключ — значение» — наилучший вариант.

Графовые базы данных

В *графовой базе данных* записи хранятся в виде вершин, а связи — в виде ребер. Вершины не привязаны к фиксированной схеме и могут содержать данные в разном формате. Графовая структура делает эффективной работу с записями в соответствии с их связями. Вот как информация из рис. 6.6 будет выглядеть в форме графа:

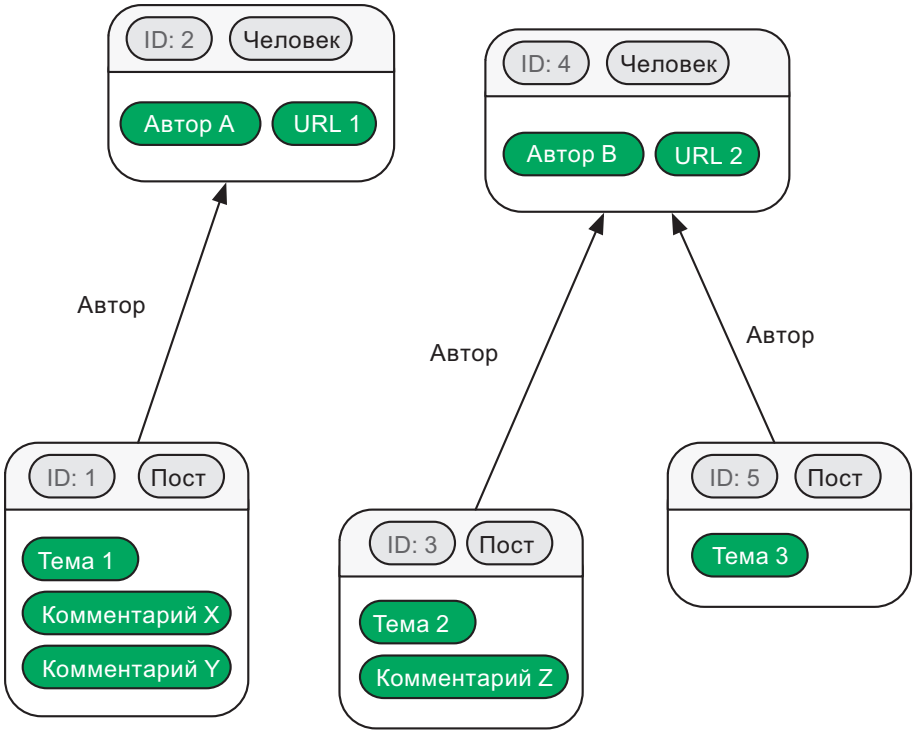


Рис. 6.7. Информация блога, хранящаяся в графовой базе данных

Это самый гибкий тип баз данных. Избавившись от таблиц и коллекций, вы можете хранить сетевые данные интуитивно понятным способом. Если бы вы решили нарисовать станции метро и остановки наземного общественного транспорта на доске, вы не стали бы изображать их в табличной форме. Вы бы использовали круги, прямоугольники и стрелки. Графовые БД позволяют хранить информацию именно таким образом.

Если ваши данные похожи на сеть, подумайте об использовании графовой базы данных. Этот тип БД особенно полезен, когда между компонентами данных много важных связей. Графовые базы данных также позволяют выполнять различные типы граф-ориентированных запросов. Например, если вы храните данные об общественном транспорте в графе, можете прямо запросить лучший маршрут между двумя остановками в одну сторону или туда и обратно.

Большие данные

Популярный в последнее время термин *«большие данные»* (big data) описывает ситуации обработки данных, которые чрезвычайно сложны с точки зрения объема, скорости или разнообразия[1]. «Объем» больших данных — это, например, обработка тысяч терабайт информации в случае с БАК[2]. «Скорость» применительно к большим данным означает, что вы должны сохранять миллион записей в секунду без задержек или быстро выполнять миллиарды запросов на чтение. «Разнообразие» означает, что данные не имеют строгой структуры,

---

[1] В профессиональных кругах это называется «три V»: volume (объем), velocity (скорость) и variety (разнообразие). Некоторые добавляют еще два аспекта: variability (переменчивость) и veracity (достоверность), превращая термин в «пять V».

[2] Большой адронный коллайдер, или БАК, — это самый большой ускоритель частиц в мире. Во время эксперимента его датчики генерируют 1000 терабайт данных в секунду.

и потому становится очень трудно с ними справляться, используя традиционные реляционные базы данных.

Каждый раз, когда вам требуется искать нестандартный подход к управлению данными по причине их объема, скорости или разнообразия, вы можете смело сказать, что имеете дело с большими данными. Для выполнения некоторых современных научных экспериментов (к примеру, связанных с БАК или SKA[1]) специалисты уже проводят исследования в области *мегаданных*, предполагающей хранение и анализ миллионов терабайт информации.

Большие данные часто связаны с нереляционными базами данных из-за их повышенной гибкости. Многие типы приложений, работающих с большими данными, практически невозможно реализовать при помощи реляционных баз данных.

SQL против NoSQL

Реляционные БД ориентированы на данные: они максимизируют структурирование данных и устраняют их дублирование независимо от того, в каком виде те требуются. Нереляционные БД, напротив, ориентированы на применение: они облегчают доступ к данным и их использование в соответствии с вашими потребностями.

Мы видели, что базы данных NoSQL позволяют быстро и эффективно сохранять крупные, изменчивые и неструктурированные данные. Не беспокоясь о фиксированных схемах и миграциях схемы, вы можете разрабатывать свои решения гораздо быстрее. Нереляционные базы данных для многих программистов естественней и проще.

---

[1] Антенная решетка площадью в квадратный километр, или SKA (от англ. Square Kilometer Array), — это группа телескопов, которые планируется ввести в строй в 2020 году. Они будут генерировать 1 млн терабайт данных каждый день.

Однако нужно помнить: какой бы крутой ни была ваша нереляционная база данных, ответственность за обновление дублированной информации по всем документам и коллекциям лежит *только на вас*. Только вы должны принимать меры для поддержания информации в непротиворечивом состоянии. Запомните: большая мощь этих БД идет рука об руку с большой ответственностью.

6.3. Распределенная модель

Существует несколько ситуаций, в которых для поддержания базы данных должен работать не один компьютер, а несколько, действующих координированно.

- Базы данных объемом в нескольких сотен терабайт. Найти одиночный компьютер с таким большим пространством хранения нереально.
- СУБД, обрабатывающие несколько тысяч одновременных запросов в секунду[1]. Никакой одиночный компьютер не имеет достаточных возможностей по передаче данных по сети или по их обработке, чтобы справиться с такой нагрузкой.
- Жизненно важные базы данных, как, например, те, что регистрируют высоту и скорость самолета, находящегося в конкретном воздушном пространстве. Полагаться на одиночный компьютер в этом случае слишком рискованно — если он выйдет из строя, база данных станет недоступной.

Для таких ситуаций существуют СУБД, способные работать на нескольких скоординированных компьютерах, образующие *распределенные базы данных*. Давайте рассмотрим наиболее распространенные способы организации таких БД.

---

[1] Сразу после финального матча на чемпионате мира 2014 года по футболу Twitter испытал пиковую нагрузку в более чем 10 000 твитов в секунду.

Репликация с одним ведущим

Один компьютер является *ведущим* и получает все запросы к базе данных. Он подключен к нескольким другим, *ведомым* компьютерам. Каждый из них содержит реплику, или копию, базы данных. Когда ведущий компьютер получает запросы на запись, он направляет их ведомым, обеспечивая их синхронизацию (рис. 6.8).

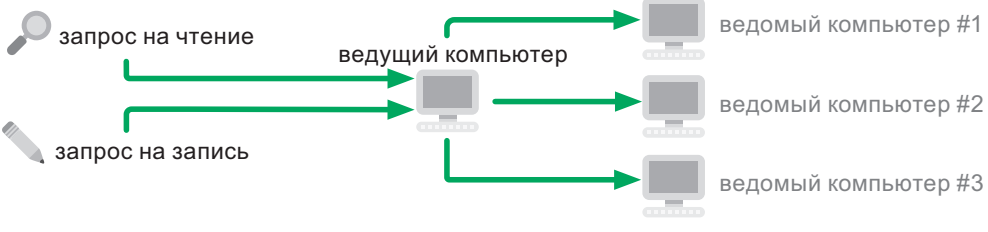


Рис. 6.8. Распределенная база данных с одним ведущим компьютером

При такой организации ведущий компьютер способен обслужить больше запросов на чтение, потому что может делегировать их ведомым компьютерам. Система становится надежнее: если основной компьютер выключается, ведомые машины автоматически координируются и выбирают новый ведущий компьютер. Благодаря этому система не прекращает свою работу.

Репликация с многочисленными ведущими

Если ваша СУБД должна обрабатывать большое количество одновременных запросов на запись, то один-единственный ведущий компьютер не справится с этой задачей. В таком случае все компьютеры в кластере становятся ведущими. Для равного распределения входящих запросов на чтение и запись между машинами используется балансировщик нагрузки (рис. 6.9).

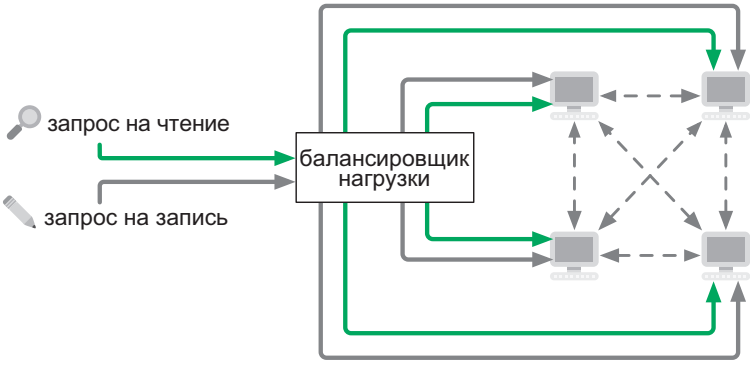


Рис. 6.9. Распределенная база данных с многочисленными ведущими машинами

Каждый компьютер подключен ко всем остальным, находящимся в кластере. Они делят запросы на запись между собой, в результате чего все остаются синхронизованными. Каждый из них имеет копию всей базы данных.

Фрагментирование

Если БД получает много запросов на запись с большими объемами данных, бывает чрезвычайно трудно синхронизировать ее везде в кластере. Некоторые компьютеры могут не иметь достаточного пространства для размещения всех данных полностью. Одно из решений состоит в том, чтобы поделить базу данных между компьютерами. Поскольку каждый из них владеет лишь ее частью, маршрутизатор направляет запросы соответствующей машине (рис. 6.10).

Такая конфигурация способна обрабатывать многочисленные запросы на чтение и запись в случае с очень большими базами данных. Но с ней возможна проблема: если машина в кластере выходит из строя, фрагмент данных, за который она отвечала, становится недоступен. Для снижения риска фрагментирование можно использовать в сочетании с репликацией (рис. 6.11).

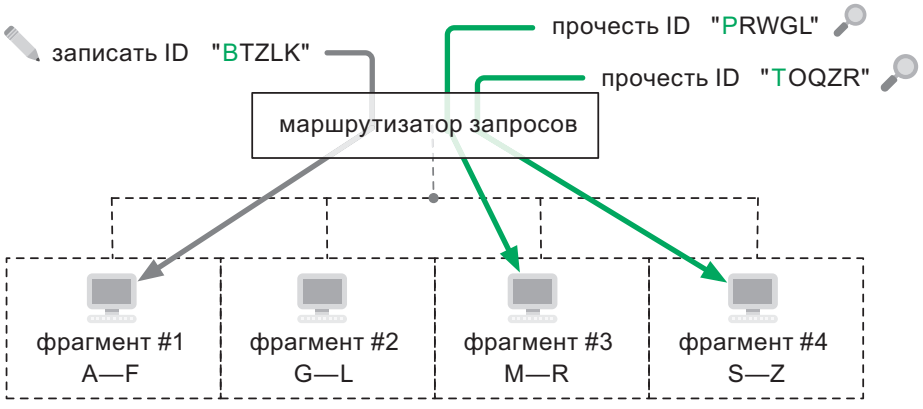


Рис. 6.10. Пример фрагментации базы данных. Запросы направляются согласно первой букве в запрашиваемом ID

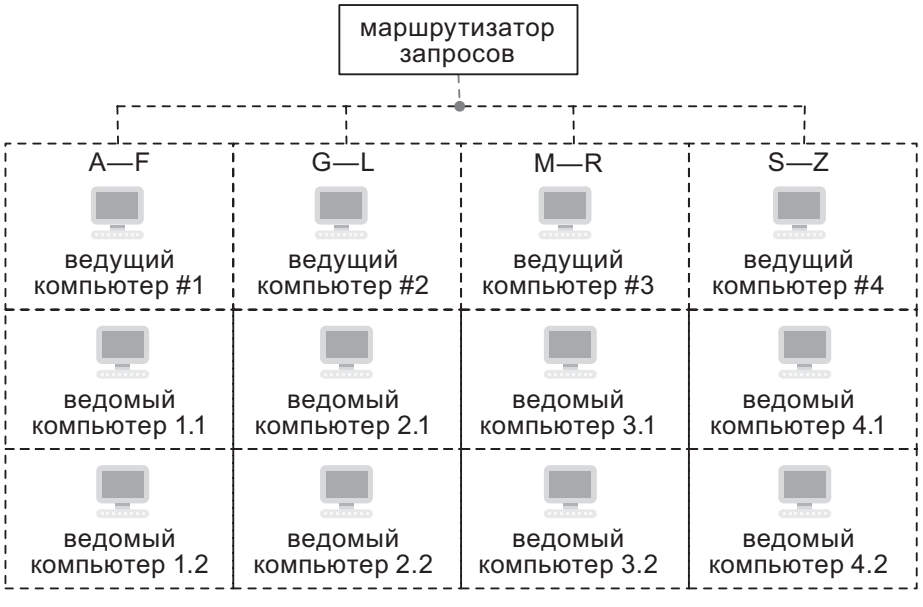


Рис. 6.11. Фрагментированная база данных с тремя репликами на фрагмент

В таком случае каждый фрагмент выполняется кластером «ведущий — ведомый». Это еще более увеличивает возможность СУБД обслуживать запросы на чтение. И если один из главных серверов во фрагменте отключается от сети, ведомое устройство автоматически занимает его место — это гарантирует, что система не развалится и не потеряет данные.

Непротиворечивость данных

Обновления в распределенных базах данных с репликацией, выполняемые на одной машине, не распространяются немедленно по всем копиям. Проходит некоторое время, пока все компьютеры в кластере синхронизируются. Это может нарушить непротиворечивость ваших данных.

Предположим, вы продаете на сайте билеты в кино. Трафик слишком большой, поэтому база данных распределена на два сервера. Элис приобретает билет на сервере А. Боб обслуживается сервером Б и видит тот же самый свободный билет. Прежде чем информация о покупке Элис дойдет до сервера Б, Боб тоже заплатит за этот билет. Теперь два сервера имеют *противоречивость* в данных. Чтобы исправить ситуацию, вам придется отменить одну из продаж и принести извинения либо недовольной Элис, либо недовольному Бобу.

Системы баз данных часто содержат инструменты для снижения противоречивости данных. Например, где-то вам позволяют делать запросы, которые обеспечивают соблюдение непротиворечивости данных по всему кластеру. Однако это уменьшает производительность СУБД. В особенности сказанное касается транзакций: они могут вызывать серьезные проблемы производительности в распределенных базах данных, поскольку вынуждают выполнять координацию всех машин в кластере с блокировкой потенциально больших объемов информации.

Есть компромиссное решение между непротиворечивостью и производительностью. Если ваши запросы к БД не требуют соблюдения

строгой непротиворечивости данных, то говорят, что они работают в условиях *потенциальной непротиворечивости*. Данные гарантированно будут непротиворечивыми *в конечном счете* — то есть через какое-то время. Это означает, что некоторые запросы на запись, вероятно, будут отклонены, а некоторые запросы на чтение могут вернуть устаревшую информацию.

Во многих случаях работа с потенциальной непротиворечивостью не вызывает особых проблем. Например, ничего страшного, если на странице вашего продукта отображается 284 отзыва вместо 285, потому что один из них был сделан только что.

6.4. Географическая модель

Многие базы данных хранят географическую информацию, такую как расположение городов или многоугольников, описывающих государственные границы. Транспортным приложениям, возможно, потребуется схема соединений автотрасс, железных дорог и станций. Бюро переписи населения нужно хранить в картографической форме тысячи переписных районов вместе с данными переписи по каждому из них.

Что интересно в таких базах данных, так это выполнение запросов в отношении пространственной информации. Например, если вы руководите службой скорой помощи, то вам нужна база данных с расположением больниц в районе. Ваша СУБД должна уметь быстро выдавать ближайшую больницу относительно любого заданного местоположения.

Такие приложения ознаменовали разработку специальных СУБД, которые называются географическими информационными системами (ГИС). Они содержат поля, специально предназначенные для географических данных: PointField (точка), LineField (прямая), PolygonField (многоугольник) и т. д. И они способны выполнять пространственные запросы в этих полях. По ГИС рек и городов вы

можете непосредственно делать запросы такого рода: *«Города в пределах 10 миль от реки Миссисипи, упорядоченные по численности населения»*. ГИС использует пространственные индексы, и поэтому поисковые запросы по пространственной близости осуществляются очень эффективно.

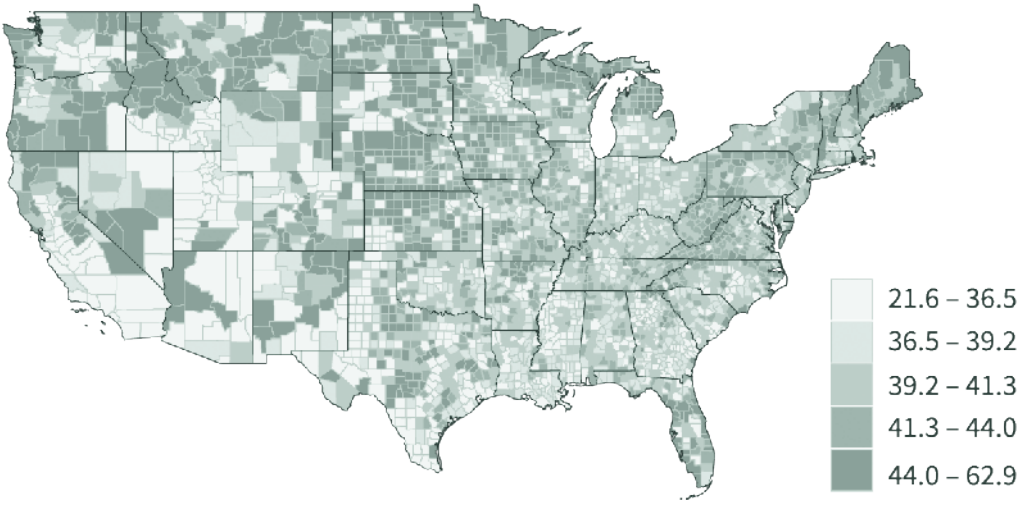


Рис. 6.12. Медианный возраст в США[1]

Эти системы даже позволяют определять пространственные ограничения. Например, в таблице, хранящей информацию по земельным участкам, можно установить ограничение, что никакие два участка не должны перекрываться. Это оградит агентства по земельному кадастру от колоссального количества проблем.

Многие общецелевые СУБД предоставляют ГИС-расширения. Всегда, когда вам приходится иметь дело с географическими данными, убедитесь, что вы используете ядро базы данных с поддержкой ГИС, и применяйте ее функционал для создания более умных запросов. ГИС-приложения часто используются в ежедневной жизни, например, в GPS-навигаторах вроде Google Maps или Waze.

[1] По данным https://census.gov.

6.5. Форматы сериализации

Как хранить данные за пределами БД в формате, совместимом с разными системами? Например, мы можем захотеть продублировать данные либо экспортировать их в другую систему. С этой целью данные должны пройти процесс *сериализации*, в ходе которого они будут преобразованы согласно формату кодирования. Получившийся файл прочитает любая система, поддерживающая этот формат. Давайте кратко пройдемся по нескольким форматам, которые широко используются для сериализации данных.

SQL — наиболее распространенный формат сериализации реляционных баз данных. Мы пишем серию команд SQL, которые воспроизводят базу данных и все ее детали. Большинство реляционных систем баз данных содержат команду DUMP для создания SQL-сериализированного дампа базы данных. Они также содержат команду RESTORE для загрузки такого файла дампа назад в СУБД.

XML — это еще один способ представить структурированные данные, но он не зависит от реляционной модели или реализации СУБД. Формат XML создавался для совместимости с разнообразными вычислительными системами и описания структуры и сложности данных. Некоторые говорят, что XML был разработан учеными, не понимавшими, что их творение не очень практично.

JSON — это формат сериализации, к которому все больше сходятся разработчики во всем мире. Он может представлять реляционные и нереляционные данные интуитивно понятным для программистов образом. Существует много расширений JSON: BSON (двоичный JSON) дает максимальную эффективность обработки данных; JSON-LD привносит в JSON мощь XML-структуры.

CSV, или файл с разделением значений запятыми, — это, возможно, самый простой формат обмена данными. Данные здесь хранятся в виде текста, по одной записи на строку. Поля в записи разделяются запятой или каким-либо другим символом, не встречающимся в данных. Формат CSV полезен для создания дампов простых БД, но он не годится для представления сложных данных.

Подведем итоги

В этой главе мы узнали, что структурирование информации в базе данных имеет чрезвычайно важное значение для того, чтобы сделать данные полезными. Мы изучили различные способы того, как это делается. Мы увидели, как реляционная модель разделяет данные на таблицы и как они связываются вместе при помощи отношений.

Большинство программистов учатся работать только с реляционной моделью, но мы вышли за эти рамки. Мы увидели альтернативные, нереляционные способы структурирования данных. Мы обсудили проблемы непротиворечивости данных и как их смягчить при помощи транзакций. Мы рассмотрели способы масштабирования СУБД для обработки интенсивных нагрузок при помощи распределенной базы данных. Мы также узнали о ГИС и о функционале, который они предлагают для работы с географическими данными. А еще мы узнали распространенные способы обмена данными между различными приложениями.

И, наконец (если только вы не экспериментируете), остановите свой выбор на широко используемой СУБД. Она производительнее и содержит меньше ошибок. Идеальной системы управления базами данных не существует. Ни одна СУБД не подходит для любых, без исключения, сценариев. Прочитав эту главу, вы теперь лучше разбираетесь в различных типах СУБД и их особенностях, и потому сможете сделать обоснованный выбор, какую из них использовать.

Полезные материалы

- Концепции систем баз данных (Database System Concepts, Silberschatz, см. https://code.energy/silber).
- *Садаладж П. Дж., Фаулер М.* NoSQL. Новая методология разработки нереляционных баз данных.
- Принципы систем распределенных баз данных (Principles of Distributed Database Systems, Özsu, см. https://code.energy/ozsu)

Глава 7

КОМПЬЮТЕРЫ

> Любая достаточно развитая технология неотличима от магии.
>
> *Артур Кларк*

Бесчисленные и разнообразные машины были изобретены для решения задач. Существует много типов компьютеров: от встроенных в роботов, которые бродят по Марсу, до тех, что управляют навигационными системами атомных подводных лодок. Почти все компьютеры, включая наши ноутбуки и телефоны, имеют тот же самый принцип работы, что и первая вычислительная машина, изобретенная фон Нейманом в 1945 году. А вы знаете, как устроены компьютеры? В этой главе вы научитесь:

понимать основы компьютерной *архитектуры*;

выбирать *компилятор* для трансляции вашего исходного кода на язык компьютеров;

разменивать память на быстродействие при помощи *иерархии памяти*.

В конце концов, программирование должно выглядеть как волшебство только для непрограммистов — но не для нас с вами.

7.1. Архитектура

Компьютер — машина, которая подчиняется командам, управляющим данными. Он имеет два главных компонента: процессор и память. Память, она же *ОЗУ*[1], — это то место, где мы пишем команды. Она также хранит данные, которыми компьютер оперирует. Процессор, или *ЦП*[2], получает команды и данные из памяти и выполняет соответствующие вычисления. Давайте разберемся, как работают эти два компонента.

Память

Память поделена на множество ячеек. Каждая хранит крошечный объем данных и имеет числовой адрес. Чтение или запись данных в памяти выполняется посредством операций, которые воздействуют на одну ячейку за раз. Чтобы прочитать ячейку памяти или произвести запись в нее, мы должны передать ее числовой адрес (рис. 7.1).

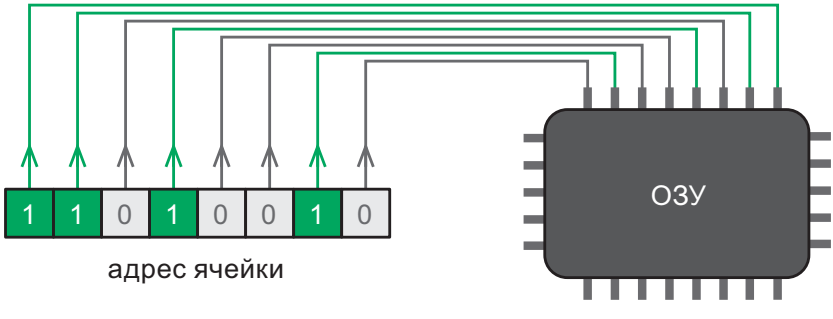


Рис. 7.1. Сообщение для ОЗУ выполнить операцию в ячейке № 210 (11010010)

[1] Оперативное запоминающее устройство (англ. RAM, random access memory). Более точное наименование на русском — запоминающее устройство (память) с произвольным доступом, сокращенно ЗУПД (ППД).

[2] Центральный процессор (англ. CPU, central processing unit).

Поскольку память является электрической схемой, мы передаем адреса ячеек по проводам в виде двоичных чисел[1]. Каждый провод передает двоичную цифру. Высокое напряжение соответствует сигналу «единица», низкое — сигналу «ноль».

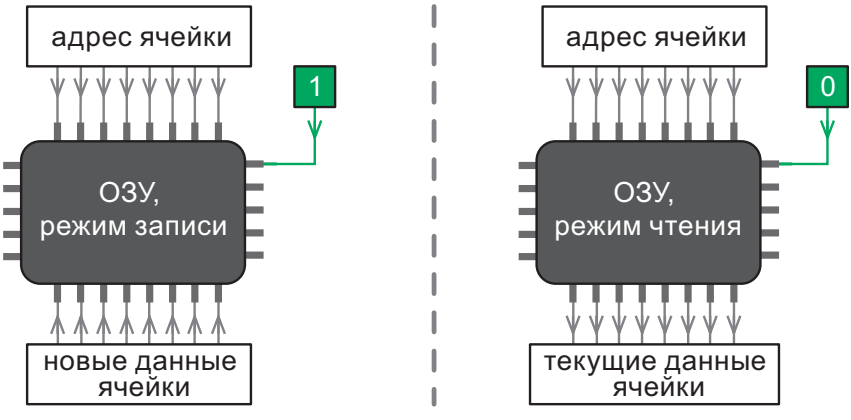


Рис. 7.2. Память может работать в режиме чтения или записи

Память способна выполнить с адресом ячейки две операции: получить хранящееся в ней значение или записать новое. Память имеет специальный входной контакт для установки ее рабочего режима (рис. 7.2).

Каждая ячейка памяти хранит 8-разрядное двоичное число, которое называется *байтом*. В режиме чтения память получает хранящийся в ячейке байт и выводит его по восьми проводам, которые передают данные (рис. 7.3).

Когда память находится в режиме записи, она *получает* байт по этим проводам и записывает его в указанную ячейку (рис. 7.4).

Группа проводов, используемых для передачи одинаковых данных, называется *шиной*. Восемь проводов для передачи адресов форми-

[1] Двоичные числа выражены в системе счисления с основанием 2. Приложение I объясняет, как это следует понимать.

руют *адресную шину*. Другие восемь, используемых для передачи информации в ячейки памяти и обратно, формируют *шину данных*. Адресная шина является однонаправленной (используется только для получения данных), а шина данных – двунаправленной (используется для отправки и для получения данных).

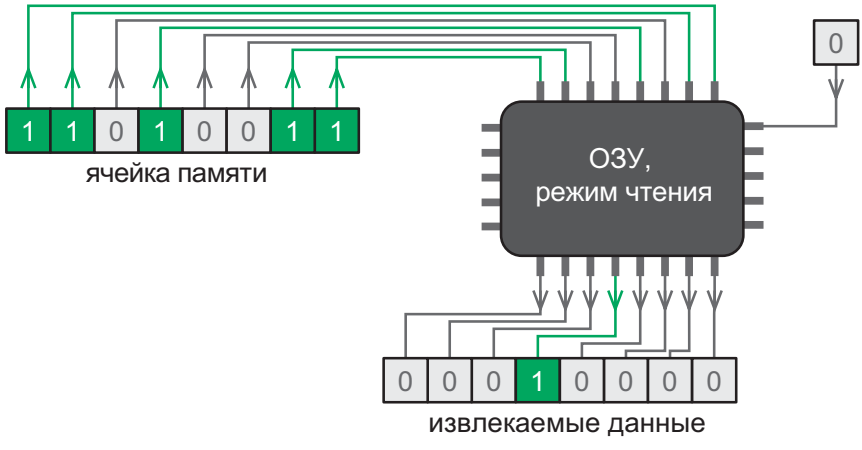


Рис. 7.3. Чтение числа 32 из ячейки с адресом 211

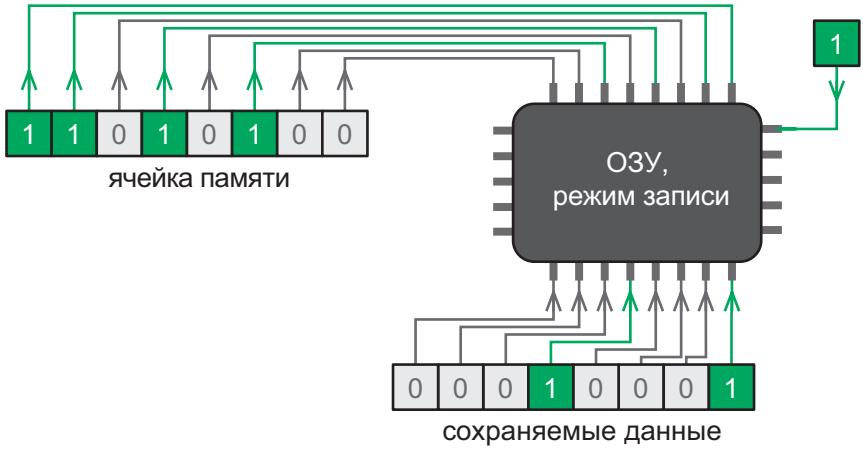


Рис. 7.4. Запись числа 33 в ячейку с адресом 212

В любом компьютере ЦП и ОЗУ постоянно обмениваются данными: процессор выбирает команды и данные из памяти и иногда сохраняет туда данные для вывода и промежуточные результаты вычислений (рис. 7.5).

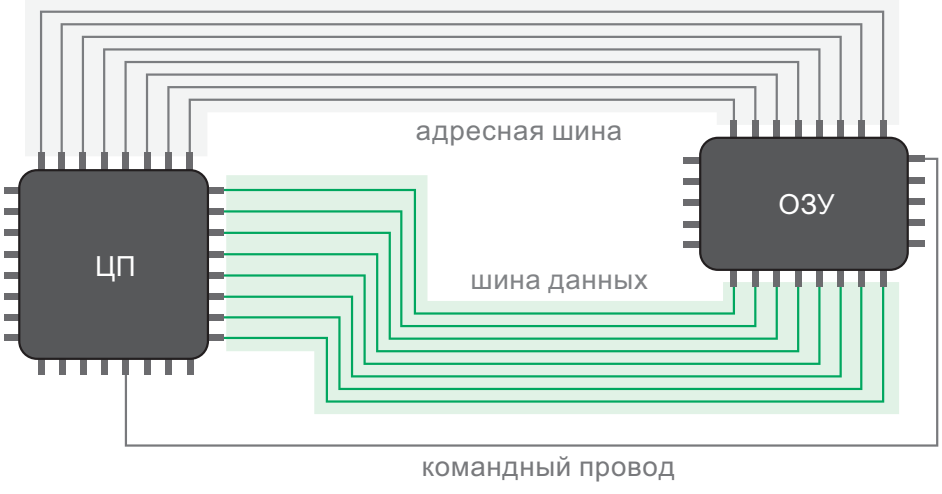


Рис. 7.5. ЦП подключен проводами к ОЗУ

Процессор

Центральный процессор имеет несколько ячеек внутренней памяти, которые называются *регистрами*. Он может выполнять простые математические операции с числами, хранящимися в этих регистрах. Он также может перемещать данные между регистрами и ОЗУ. Вот примеры типичных операций, которые приходится исполнять центральному процессору:

- скопировать данные из ячейки памяти № 220 в регистр № 3;
- сложить число в регистре № 3 с числом в регистре № 1.

Набор всех операций, которые может выполнять ЦП, называется его *набором команд*. Каждой операции в наборе команд присвоено число. Машинный код по существу — последовательность чисел, представ-

ляющих операции центрального процессора. Они хранятся в виде чисел в ОЗУ. Мы сохраняем входные/выходные данные, промежуточные результаты и машинный код — все вперемешку — в ОЗУ[1].

Рис. 7.6 показывает, как некоторым процессорным командам ставятся в соответствие числа в том виде, в котором они приводятся в руководствах по ЦП. По мере совершенствования технологии производства процессоры стали поддерживать дополнительные операции. Набор команд современных ЦП огромен. Однако самые важные операции существовали уже несколько десятилетий назад.

4004 Instruction Set

BASIC INSTRUCTIONS

MNEMONIC	OPR D_3 D_2 D_1 D_0	OPA D_3 D_2 D_1 D_0	DESCRIPTION OF OPERATION
NOP	0 0 0 0	0 0 0 0	No operation.
INC	0 1 1 0	R R R R	Increment contents of register RRRR.
ADD	1 0 0 0	R R R R	Add contents of register RRRR to accumulator with carry.
LD	1 0 1 0	R R R R	Load contents of register RRRR to accumulator.
LDM	1 1 0 1	D D D D	Load data DDDD to accumulator.
CLC	1 1 1 1	0 0 0 1	Clear carry.
IAC	1 1 1 1	0 0 1 0	Increment accumulator.
DAC	1 1 1 1	1 0 0 0	Decrement accumulator.

Рис. 7.6. Часть технического описания Intel 4004, показывающая, как операциям ставятся в соответствие числа. Это был первый в мире ЦП, выпущенный в 1971 году

ЦП работает в бесконечном цикле, постоянно выбирая и исполняя команды из памяти. В ядре цикла находится регистр PC, или счетчик

[1] Программный код даже может изменять сам себя за счет включения команд, которые переписывают части собственного кода в ОЗУ. Нередко компьютерные вирусы поступают именно так, чтобы затруднить их обнаружение антивирусным ПО. Здесь можно провести удивительную параллель с биологическими вирусами, изменяющими свои ДНК, чтобы спрятаться от иммунной системы носителей.

команд[1]. Это специальный регистр, который хранит адрес памяти следующей исполняемой команды. Вот что делает ЦП:

1) выбирает команду в адресе памяти, заданном регистром PC;
2) увеличивает PC на 1;
3) выполняет команду;
4) возвращается к шагу 1.

Когда ЦП включается, PC присваивается значение по умолчанию, то есть адрес первой команды, выполняемой машиной. Это обычно неизменяемая встроенная программа, ответственная за загрузку основных функций компьютера[2].

После включения ЦП начинает выполнять этот бесконечный цикл выборки и исполнения, пока вы не выключите компьютер. Однако если бы ЦП мог выполнять только упорядоченный, последовательный список операций, то компьютер был бы не более чем продвинутым калькулятором. ЦП удивителен, потому что ему можно поручить записать новое значение в регистр PC, заставив процесс исполнения команд выполнить переход — «перепрыгнуть» куда-то в другое место в памяти. Такое ветвление может быть условным выражением. Например, команда ЦП может сообщить: *«Записать в PC адрес № 200, если регистр № 1 хранит ноль»*. Это позволяет компьютерам выполнять операторы, подобные следующему:

```
if x = 0
    compute_this()    # вычислить это
else
    compute_that()    # вычислить то
```

Вот и все, что вам надо знать. Открываете ли вы сайт, играете ли в компьютерную игру или редактируете электронную таблицу, вы-

[1] Не путайте эту аббревиатуру с общепринятым акронимом для Personal Computer (PC) — «персональный компьютер».

[2] Во многих персональных компьютерах эта программа называется BIOS (англ. basic input/output system, «базовая система ввода-вывода»).

числения всегда одинаковы: это серия простых операций, которые могут лишь суммировать, сравнивать или перемещать данные в памяти.

При помощи множества этих простых операций можно выражать запутанные процедуры. Например, код классической игры Space Invaders (рис. 7.7) включает порядка 3000 машинных команд.

Рис. 7.7. Игру Space Invaders, выпущенную в 1978 году, многие называют самой влиятельной за всю историю

Тактовая частота ЦП. В 1980-х годах чрезвычайно популярной стала игра Space Invaders. Люди играли в нее на игровых автоматах, оборудованных процессорами с тактовой частотой 2 МГц. Этот показатель — число базовых операций, которые процессор выполняет в секунду. Процессор с тактовой частотой 2 МГц выполняет примерно 2 млн базовых операций в секунду. Для выполнения машинной команды требуется от пяти до десяти базовых операций. Следовательно, винтажные игровые автоматы выполняли *сотни тысяч машинных команд каждую секунду.*

В условиях современного технологического прогресса обычные настольные компьютеры и смартфоны обычно имеют процессоры с тактовой частотой 2 ГГц. Они способны выполнять сотни миллионов машинных команд каждую секунду. А с недавних пор массовое применение получили многоядерные ЦП. Четырехъядерный процессор с тактовой частотой 2 ГГц может выполнять почти миллиард машинных команд в секунду. И, похоже, в перспективе у наших процессоров будет все больше ядер[1].

Архитектуры ЦП. Вы когда-нибудь задавались вопросом, почему нельзя вставить компакт-диск для Sony PlayStation в настольный компьютер и начать играть? Или почему приложения для iPhone не запускаются на Mac? Причина проста: разные архитектуры ЦП.

В наше время архитектура x86 является довольно стандартной, и потому одинаковый код может выполняться на большинстве персональных компьютеров. Однако сотовые телефоны, например, имеют процессоры с другой, более энергоэффективной архитектурой. Разные архитектуры означают разные наборы процессорных команд и, следовательно, разные способы их кодирования числами. Числа, которые транслируются как команды для ЦП вашего настольного компьютера, не являются допустимыми командами для ЦП в вашем сотовом телефоне, и наоборот.

32-разрядная архитектура против 64-разрядной. Первый ЦП под названием Intel 4004 был основан на 4-разрядной архитектуре. Это означает, что он мог оперировать двоичными числами (суммировать, сравнивать, перемещать их) до 4 разрядов в одной машинной команде. Шина данных и шина адресов на Intel 4004 состояли всего из четырех проводов каждая.

Вскоре после этого широкое распространение получили 8-разрядные ЦП. Они использовались в ранних персональных компьютерах, работавших под DOS[2]. Game Boy, популярный в 1980–1990-х годах

[1] О процессоре с 1000 ядер исследователи объявили еще в 2016 году.

[2] Дисковая операционная система (Disk Operating System). Об операционных системах мы вскоре расскажем подробнее.

переносной игровой компьютер, тоже имел 8-разрядный процессор. Одиночная команда в таких ЦП может оперировать 8-разрядными двоичными числами.

Быстрый технологический прогресс позволил занять доминирующее положение 16-разрядной, а затем — 32-разрядной архитектуре. Емкость регистров ЦП была увеличена до 32 разрядов. Для более емких регистров естественно потребовалось расширить шины данных и адресов. Адресная шина с 32 проводами позволяет адресовать 2^{32} байт (4 Гб) памяти.

А затем наша жажда вычислительной мощи стала просто неудержимой. Компьютерные программы быстро усложнялись и использовали все больше памяти. 4Гб ОЗУ оказалось слишком мало. И обращение к памяти большего объема с числовыми адресами, которые укладываются в 32-разрядные регистры, превратилась в непростой процесс. Это ознаменовало появление доминирующей сегодня 64-разрядной архитектуры. 64-разрядные процессоры могут оперировать в одной команде чрезвычайно большими числами. При этом 64-разрядные регистры хранят адреса в огромном пространстве памяти — 2^{64} байт, что составляет более 17 млрд гигабайт.

Прямой порядок байтов против обратного. Некоторые разработчики компьютеров посчитали, что в ОЗУ и ЦП целесообразно хранить числа слева направо (от младших разрядов к старшим), способом, известным как *обратный порядок байтов*. Другие предпочли записывать данные справа налево, способом, который называется *прямым порядком байтов*. Двоичная последовательность `1-0-0-0-0-0-1-1` может представлять разные числа в зависимости от порядка байтов:

- прямой порядок байтов: $2^7 + 2^1 + 2^0 = 131$;
- обратный порядок байтов: $2^0 + 2^6 + 2^7 = 193$.

Большинство ЦП сегодня имеют обратный порядок байтов, вместе с тем существует много компьютеров с прямым порядком. Если данные, сгенерированные ЦП с обратным порядком байтов, должны интерпретироваться процессором с прямым порядком, то не-

обходимо принять меры, чтобы избежать *несоответствия порядка байтов*. Программисты, манипулирующие двоичными числами напрямую, в особенности во время разбора данных, выходящих из сетевых коммутаторов, должны об этом помнить. Несмотря на то что большинство компьютеров сегодня имеет обратный порядок байтов, интернет-трафик стандартизировал прямой порядок, потому что большинство ранних сетевых маршрутизаторов имели соответствующие ЦП. Данные с прямым порядком окажутся искажены, если их прочитать так, как если бы порядок в них был обратным, и наоборот.

Эмуляторы. Иногда бывает полезно на своем компьютере выполнить некоторый программный код, разработанный для другого ЦП. Это позволяет протестировать приложение для iPhone без iPhone или сыграть в вашу любимую старинную игру для Super Nintendo. Для этих задач существуют компоненты программного обеспечения, которые называются эмуляторами.

Эмулятор имитирует целевую машину: компьютер притворяется, что имеет тот же ЦП, ОЗУ и другие аппаратные средства. Команды декодируются программой эмулятора и выполняются в эмулированной машине. Как вы понимаете, очень сложно эмулировать одну машину внутри другой, когда у них разная архитектура. Но поскольку наши компьютеры намного быстрее старых, это стало возможным. Если вы раздобудете эмулятор Game Boy и позволите своему компьютеру создать виртуальную игровую приставку, то сможете играть в игры точно так же, как если бы вы играли на настоящем Game Boy.

7.2. Компиляторы

Мы программируем компьютеры, чтобы они могли делать МРТ, распознавать речь, исследовать далекие планеты и выполнять много других сложных задач. Удивительно, но все, на что способен компьютер, в конечном счете осуществляется посредством простых команд ЦП, которые просто суммируют и сравнивают числа. Сложные приложения, например интернет-браузер, требуют миллионов или миллиардов таких машинных команд.

Но мы редко пишем программы непосредственно как команды ЦП. Человеку не под силу написать реалистичную трехмерную компьютерную игру подобным образом. Чтобы выражать свои предписания более естественным и компактным образом, люди создали *языки программирования*. Мы пишем программный код на этих языках[1], а затем используем программу, которая называется *компилятором*, для перевода наших предписаний в машинные команды, понятные процессору.

Чтобы объяснить, что делает компилятор, давайте представим простую математическую аналогию. Если мы хотим попросить кого-то вычислить факториал числа 5, мы можем задать вопрос:

$$5! = ?$$

Однако если человек, которого мы спрашиваем, не знает, что такое факториал, то вопрос не будет иметь смысла. Нам придется его перефразировать, используя более простые операции:

$$5 \times 4 \times 3 \times 2 \times 1 = ?$$

А вдруг человек, которого мы спрашиваем, умеет только суммировать? Нам придется упростить наше выражение еще больше:

$$5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 +$$

$$5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 + 5 = ?$$

По мере того как мы переписываем наше вычисление во все более простой форме, требуется все больше операций. Так же обстоит дело и с машинным кодом. Компилятор переводит сложные предписания на языке программирования в эквивалентные команды ЦП. Задействуя мощные возможности внешних библиотек, мы выражаем сложные программы, состоящие из миллиардов команд ЦП, посред-

[1] О языках программирования подробнее будет рассказано в следующей главе.

ством относительно небольшого числа строк программного кода, которые понятны и легко изменяемы.

Алан Тьюринг, основоположник компьютерных вычислений, обнаружил, что простые машины способны вычислить *все*, что в принципе поддается вычислению. Чтобы обладать универсальными вычислительными возможностями, машина должна уметь выполнять программу, которая содержит команды:

- чтения и записи данных в памяти;
- условного ветвления (если адрес памяти имеет заданное значение, то перейти к другой точке в программе).

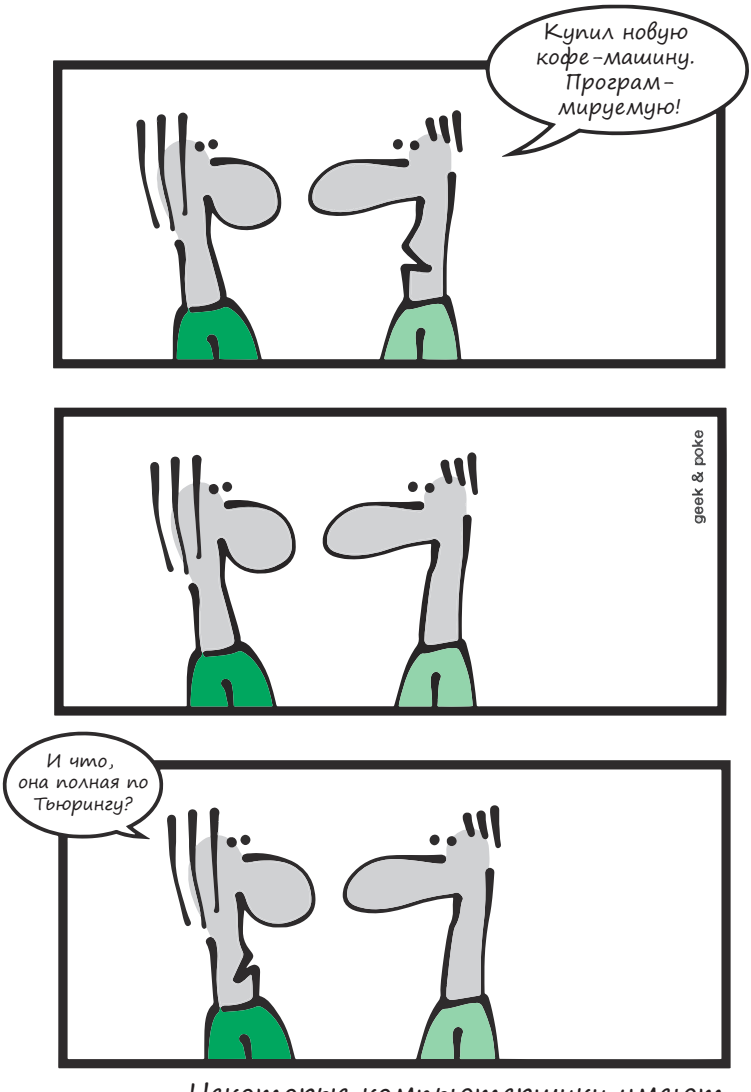
Машины, обладающие универсальными вычислительными возможностями, называются *полными по Тьюрингу*. Не имеет значения, насколько длинным или запутанным является вычисление, оно всегда может быть выражено с точки зрения простых команд чтения/записи и перехода. При достаточном количестве времени и памяти эти команды способны вычислять что угодно.

Недавно было показано, что команда ЦП под названием `MOV` («перемещение») является полной по Тьюрингу. Это значит, что ЦП, который выполняет только команду `MOV`, способен делать все то, что может полноценный ЦП. Другими словами, любой тип программного кода вполне реально выразить исключительно с помощью команды `MOV`[1].

Важный вывод из этой новости состоит в том, что если программу можно записать на языке программирования, то ее можно переписать для выполнения на любой полной по Тьюрингу машине, какой бы простой та ни была. Компилятор — это волшебная программа, которая автоматически транслирует код из сложного языка в более простой.

---

[1] Можете взглянуть на компилятор, который превращает любой код на C в двоичный код лишь с одной машинной командой `MOV`: https://code.energy/mov.



Некоторые компьютерщики умеют испортить удовольствие

Рис. 7.8[1]

[1] Любезно предоставлено http://geek-and-poke.com.

Операционные системы

Скомпилированные компьютерные программы по существу являются последовательностями команд ЦП. Как мы выяснили, код, скомпилированный для настольного компьютера, не станет работать на смартфоне, потому что эти машины имеют процессоры различной архитектуры. Но скомпилированная программа может не работать и на одном из двух компьютеров, имеющих одинаковую архитектуру ЦП. Дело в том, что программы, чтобы запускаться без проблем, должны взаимодействовать с *операционной системой* (ОС) компьютера.

Чтобы осуществлять контакты с внешним миром, программе нужно вводить и выводить информацию: открывать файлы, писать сообщения на экране, устанавливать сетевое соединение и т. д. Но разные компьютеры имеют разные аппаратные средства. Программа сама по себе не способна поддерживать все существующие типы экранов, звуковых карт или сетевых плат.

Вот почему в своей работе программы опираются на операционную систему. Благодаря ее помощи они легко работают с различными аппаратными средствами. Программы совершают специальные *системные вызовы*, чтобы ОС выполнила необходимые операции ввода-вывода. Компиляторы переводят команды ввода-вывода в надлежащие системные вызовы.

Однако разные ОС часто используют несовместимые системные вызовы. Системный вызов печати чего-либо на экране в Windows отличается от такового в Mac OS или Linux.

Вот почему, если вы компилируете программу для выполнения в Windows с процессором x86, она не будет работать в Mac с таким же процессором. Скомпилированный программный код должен быть ориентирован не только на конкретную архитектуру процессора, но и на конкретную операционную систему.

Оптимизация при компиляции

Хорошие компиляторы стараются оптимизировать машинный код, который они генерируют. Если они видят, что части вашего кода можно заменить более эффективными эквивалентами, они это сделают. Компиляторы порой применяют сотни правил оптимизации, прежде чем произвести двоичный код.

Именно поэтому вам не следует жертвовать простотой чтения кода в пользу его микрооптимизации. Компилятор так или иначе применит все тривиальные оптимизации. Посмотрите на этот фрагмент кода:

```
function factorial(n)
    if n > 1
        return factorial(n - 1) * n
    else
        return 1
```

Кто-то скажет, что его лучше заменить на этот эквивалент:

```
function factorial(n)
    result ← 1
    while n > 1
        result ← result * n
        n ← n - 1
    return result
```

Да, выполнение процедуры `factorial` без рекурсии использует меньше вычислительных ресурсов. Но это еще не повод менять программный код. Современные компиляторы автоматически перепишут простые рекурсивные функции. Вот еще один пример:

```
i ← x + y + 1
j ← x + y
```

Компиляторы избавятся от повторного вычисления `x + y` и сделают вот такое преобразование:

```
t1 ← x + y
i ← t1 + 1
j ← t1
```

Сосредоточьтесь на написании чистого и ясного программного кода. Если у вас есть проблемы, связанные с производительностью, используйте инструменты профилирования для обнаружения узких мест в вашем программном коде и пробуйте реализовать эти части более умными способами. Не стоит напрасно тратить время на ненужную микрооптимизацию.

Впрочем, иногда этап компиляции просто отсутствует. Давайте посмотрим, что это за ситуации.

Языки сценариев

Некоторые языки программирования, так называемые языки сценариев, выполняются без прямой компиляции в машинный код. К ним относятся JavaScript, Python и Ruby. Код на этих языках выполняет не центральный процессор непосредственно, а *интерпретатор* — программа, которая должна быть установлена на компьютере.

Так как интерпретатор переводит программный код в машинный в режиме реального времени, тот обычно работает *намного* медленнее скомпилированного кода. С другой стороны, программист может выполнить код, не ожидая окончания компиляции. Когда проект очень большой, компиляция иногда занимает несколько часов.

Инженерам Google приходилось постоянно компилировать большие пакеты кода. Это заставляло разработчиков терять (рис. 7.9) много времени. В Google не могли переключиться на языки сценариев — нужна была максимальная производительность скомпилированного двоичного файла. Поэтому они разработали Go — язык, который компилируется невероятно быстро и имеет очень высокую производительность.

Рис. 7.9. Компиляция[1]

Дизассемблирование и обратный инженерный анализ

Восстановить исходный код скомпилированной программы — тот, что был до момента компиляции, — нельзя[2]. Но можно декодировать бинарную программу, трансформировав числа, в которых закодированы команды ЦП, в последовательность команд, более-менее понятную для человека. Этот процесс называется *дизассемблированием*.

Затем можно рассмотреть команды ЦП и попытаться выяснить, что они делают, — такой процесс называется *обратным инженерным анализом*. Некоторые программы дизассемблирования значительно помогают в этом, автоматически обнаруживая и аннотируя системные

[1] Любезно предоставлено http://xkcd.com.

[2] По крайней мере, на данный момент. С развитием искусственного интеллекта это когда-нибудь окажется возможным.

вызовы и часто используемые функции. Благодаря инструментам дизассемблирования хакер может разобраться в любом аспекте работы двоичного кода. Я уверен, что многие лучшие компании в области ИТ имеют секретные лаборатории обратного инженерного анализа, где изучают программное обеспечение конкурентов.

Хакеры часто анализируют двоичный код лицензируемых программ, таких как Microsoft Windows, Adobe Photoshop и Grand Theft Auto, чтобы определить, какая часть кода проверяет лицензию. Они модифицируют двоичный код, помещая команду `JUMP` для прямого перехода в ту часть кода, которая выполнятся после проверки лицензии. Когда модифицированный двоичный файл выполняется, он добирается до введенной команды `JUMP` прежде, чем будет сделана проверка достоверности лицензии. Таким образом люди запускают незаконные пиратские копии программы, не платя за лицензию.

Исследователи и инженеры по безопасности, работающие на секретные правительственные структуры, также имеют лаборатории для изучения популярного потребительского программного обеспечения, такого как iOS, Microsoft Windows или Internet Explorer. Они идентифицируют потенциальные нарушения защиты в этих программах, чтобы обезопасить людей от кибератак или предотвратить взлом целей, имеющих большую ценность. Самой известной атакой такого рода был Stuxnet — кибероружие, созданное агентствами из США и Израиля. Оно замедлило ядерную программу Ирана, инфицировав компьютеры, которые управляли подземными термоядерными реакторами.

Программное обеспечение с открытым исходным кодом

Как мы уже объясняли, вы можете проанализировать команды дизассемблированной программы, но вам не удастся восстановить исходный код, который использовался для генерирования двоичного кода, так называемого бинарника.

Не имея исходного кода, вы можете лишь слегка изменить двоичный код, но у вас не выйдет внести в программу какое-либо существенное изменение, например добавить новый функционал. Некоторые люди считают, что намного лучше разрабатывать программы сообща. Они оставляют свой код открытым для других людей, чтобы те могли вносить свои изменения. Главная идея здесь — создавать программное обеспечение, которое всякий может свободно использовать и модифицировать. Основанные на Linux операционные системы (такие как Ubuntu, Fedora и Debian) являются открытыми, тогда как Windows и Mac OS — закрытыми.

Интересное преимущество операционных систем с открытым исходным кодом состоит в том, что любой может проинспектировать исходный код в поисках уязвимостей. Уже не раз было подтверждено, что государственные учреждения шпионят за миллионами граждан, используя неисправленные уязвимости защиты в повседневном потребительском программном обеспечении.

Надзор за ПО с открытым исходным кодом осуществляет куда больше глаз, поэтому лицам с дурными намерениями и правительственным учреждениям становится все труднее находить лазейки для слежки. Когда вы используете Mac OS или Windows, вам приходится доверять Microsoft или Apple, что они не поставят под угрозу вашу безопасность и приложат все усилия для предотвращения любого серьезного дефекта. А вот системы с открытым исходным кодом открыты для общественного контроля, потому в случае с ними меньше вероятность, что брешь в системе безопасности останется незамеченной.

7.3. Иерархия памяти

Мы знаем, что компьютер работает за счет ЦП, который исполняет простые команды. Мы знаем также, что эти команды могут оперировать только данными, хранящимися в регистрах ЦП. Однако их емкость обычно намного меньше тысячи байтов. Это означает, что

регистрам ЦП постоянно приходится перемещать данные в ОЗУ и обратно.

Если доступ к памяти медленный, то ЦП приходится простаивать, ожидая, пока ОЗУ выполнит свою работу. Время, которое требуется, чтобы прочитать и записать данные в память, непосредственно отражается на производительности компьютера. Увеличение скорости памяти может разогнать ваш компьютер так же, как увеличение скорости ЦП. Данные в регистрах ЦП выбираются почти моментально самим процессором всего в *одном* цикле[1]. А вот ОЗУ *гораздо* медленнее.

Разрыв между памятью и процессором

Недавние технические разработки позволили экспоненциально увеличивать скорость ЦП. Быстродействие памяти тоже растет, но гораздо медленнее. Эта разница в производительности между ЦП и ОЗУ называется *разрывом между памятью и процессором*: команды ЦП «дешевы» — мы можем выполнять их в огромном количестве, тогда как получение данных из ОЗУ занимает намного больше времени и потому обходится «дорого». По мере увеличения этого разрыва возрастала важность эффективного доступа к памяти (рис. 7.10).

В современных компьютерах требуется приблизительно *тысяча* циклов ЦП, чтобы получить данные из ОЗУ, — около 1 микросекунды[2]. Это невероятно быстро, но составляет целую вечность по сравнению со временем доступа к регистрам ЦП. Программистам приходится искать способы сократить количество операций с ОЗУ.

---

[1] В ЦП с таковой частотой 1 ГГц один цикл длится порядка одной миллиардной секунды — время, которое необходимо, чтобы свет прошел расстояние от страницы этой книги до ваших глаз.

[2] Нужно где-то 10 микросекунд, чтобы звуковые волны вашего голоса достигли человека, который стоит перед вами.

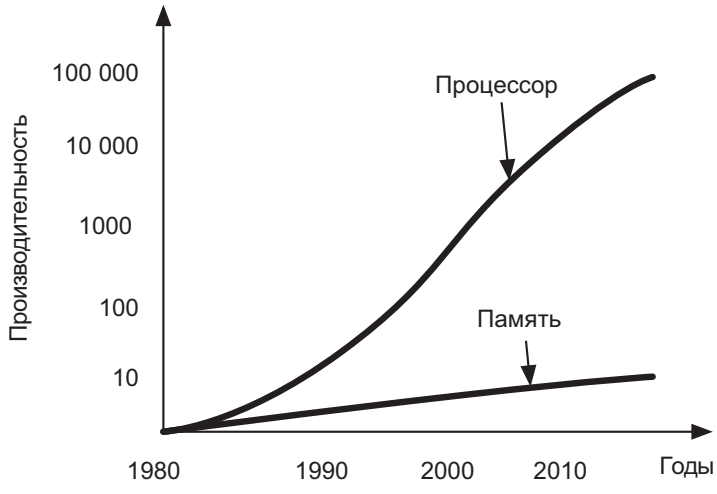


Рис. 7.10. Разрыв в быстродействии между памятью и процессором в последние десятилетия

Временная и пространственная локальность

Пытаясь свести к минимуму количество обращений к ОЗУ, специалисты в области computer science стали замечать две закономерности, получившие следующие названия:

- **временна́я локальность** — если выполняется доступ к некоему адресу памяти, то вполне вероятно, что к нему вскоре обратятся снова;
- **пространственная локальность** — если выполняется доступ к адресу памяти, то вполне вероятно, что к смежным с ним адресам вскоре обратятся тоже.

Если исходить из вышеприведенных наблюдений, можно подумать, что хранить такие адреса памяти в регистрах ЦП — отличная идея. Это позволило бы избежать большинства дорогостоящих операций с ОЗУ. Однако отраслевые инженеры не нашли надежного способа разработать микросхемы ЦП с достаточным количеством внутрен-

них регистров. И тем не менее они обнаружили отличный способ опереться на временную и пространственную локальность. Давайте посмотрим, как он работает.

Кэш L1

Есть возможность сформировать чрезвычайно быструю вспомогательную память, интегрированную в ЦП. Мы называем ее кэшем первого уровня, или *кэшем L1*. Получение данных из этой памяти в регистры осуществляется чуть-чуть медленнее, чем получение данных из самих регистров.

При помощи кэша L1 мы можем копировать содержимое адресов памяти, причем скорость доступа с высокой вероятностью будет близка к скорости регистров ЦП. Данные очень быстро загружаются в регистры ЦП. Требуется где-то 10 циклов ЦП, чтобы переместить данные из кэша L1 в регистры. Это в сто раз быстрее ситуации, когда их приходится брать из ОЗУ.

Более половины обращений к ОЗУ можно выполнить за счет кэша L1 объемом 10 Кб и умного использования временной и пространственной локальности. Это новаторское решение привело к коренному изменению вычислительных технологий. Оборудование ЦП кэшем L1 позволило кардинально сократить время, которое процессору прежде приходилось тратить на ожидание данных. Вместо простоев он теперь занят выполнением фактических вычислений.

Кэш L2

Можно было бы дальше увеличивать размер кэша L1 — выборка данных из ОЗУ тогда стала бы еще более редкой операцией, а время ожидания ЦП еще больше сократилось бы. Однако такое усовершенствование сопряжено с трудностями. Когда кэш L1 достигает размера около 50 Кб, его дальнейшее увеличение становится очень дорогостоящим. Куда лучшее решение состоит в том, чтобы сформировать

дополнительный кэш памяти — кэш второго уровня, или *кэш L2*. Он будет медленнее, зато намного больше кэша L1 по объему. Современный ЦП имеет кэш L2 объемом около 200 Кб. Чтобы переместить данные из кэша L2 в регистры, требуется примерно 100 циклов процессора.

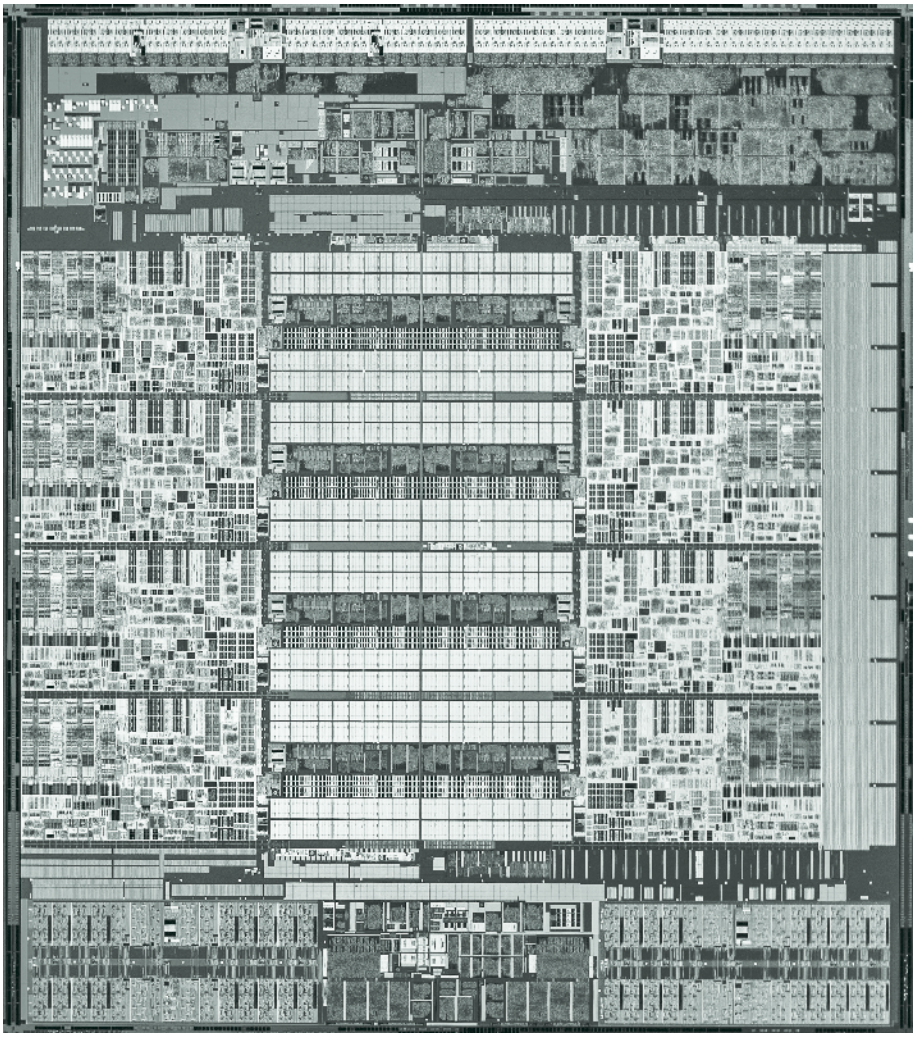


Рис. 7.11. Микрофотография процессора Intel Haskell-E. Квадратные структуры в центре — это кэш L3 объемом 20 Мб

Адреса, имеющие очень высокую вероятность доступа, мы копируем в кэш L1. Адреса с относительно высокой вероятностью — в кэш L2. Если адреса нет в кэше L1, процессор может попытаться найти его в кэше L2, и только если этот поиск закончится неудачей, ему придется обращаться к ОЗУ.

Многие производители теперь поставляют процессоры с кэшем L3: он больше и медленнее, чем L2, но по-прежнему быстрее ОЗУ. Кэши L1/L2/L3 имеют настолько важное значение, что занимают бóльшую часть кремниевого пространства внутри микросхемы ЦП (рис. 7.11).

Использование кэшей L1/L2/L3 существенно увеличивает производительность компьютеров. Благодаря кэшу L2 емкостью 200 Кб менее 10 % запросов к памяти, которые делает ЦП, приходятся на выборку непосредственно из ОЗУ.

В следующий раз, когда вы пойдете покупать компьютер, не забудьте сравнить размеры кэшей L1/L2/L3 процессоров. Более хорошие ЦП будут иметь кэш большей емкости. Лучше взять ЦП с меньшей тактовой частотой, но с более объемным кэшем.

Первичная память против вторичной

Как показано на рис. 7.12, компьютер имеет разные типы памяти, организованные иерархически. Наиболее эффективные типы имеют ограниченную емкость и очень дорого стоят. Спускаясь по иерархии вниз, мы получаем больше памяти, но скорость доступа к ней становится все меньше.

После регистров ЦП и кэшей в иерархии памяти находится *ОЗУ*. Она отвечает за хранение данных и кода всех выполняющихся процессов. По состоянию на 2017 год компьютер обычно имеет ОЗУ емкостью от 1 до 10 Гб. Во многих случаях этого недостаточно, чтобы разместить операционную систему со всеми другими выполняющимися программами.

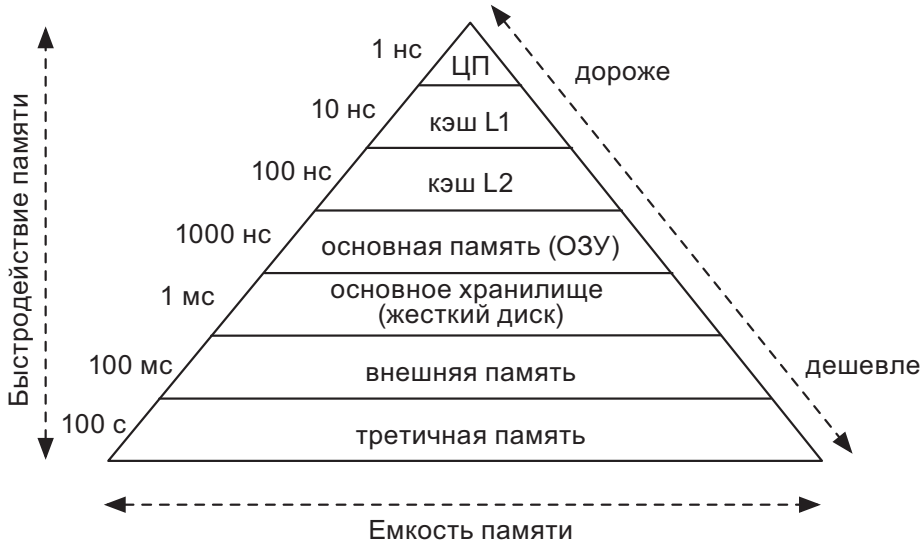


Рис. 7.12. Диаграмма иерархии памяти

В этих случаях приходится спускаться ниже по иерархической лестнице и использовать *жесткий диск*. По состоянию на 2017 год компьютеры обычно имеют жесткие диски емкостью в сотни гигабайт — этого более чем достаточно, чтобы уместить данные из всех выполняющихся программ. Когда ОЗУ заполнено, мы перемещаем временно не используемые данные на жесткий диск, чтобы высвободить немного оперативной памяти.

Но проблема в том, что жесткие диски работают *чрезвычайно* медленно. Как правило, на перемещение данных между диском и ОЗУ требуется *миллион* циклов ЦП — целая миллисекунда[1]. Может показаться, что это все равно быстро, но не забывайте: в то время как доступ к ОЗУ занимает всего 1000 циклов, на доступ к диску их уходит миллион. ОЗУ нередко носит название *первичной памяти*, а программы и данные, хранящиеся на жестком диске, являются *вторичной памятью*.

[1] Стандартная фотографическая съемка фиксирует свет в течение примерно четырех миллисекунд.

ЦП не может обращаться к вторичной памяти напрямую. Программы, которые хранятся во вторичной памяти, нужно скопировать в первичную — только тогда они будут исполнены. В действительности всякий раз, когда вы загружаете компьютер, даже операционную систему приходится копировать с диска в ОЗУ, прежде чем ЦП сможет ее выполнить.

Никогда не истощайте ОЗУ! Очень важно, чтобы все данные и программы, которыми компьютер управляет во время обычной работы, могли уместиться в его ОЗУ. В противном случае он будет постоянно перемещать их между диском и ОЗУ. Поскольку этот процесс *очень медленный*, производительность компьютера сильно падает, и он становится бесполезным. В таком случае он большую часть времени ждет, пока данные будут перемещены, вместо того чтобы выполнять фактические вычисления.

Когда компьютер постоянно перемещает данные с диска в ОЗУ, мы говорим, что он *вошел в режим интенсивной подкачки*. Эту ситуацию необходимо постоянно отслеживать — особенно в случае с серверами: если они начинают обрабатывать данные, не умещающиеся в ОЗУ, такой режим работы может вывести их из строя. Следствием этого, например, становится длинная очередь в банке или у кассового аппарата, — и оператору ничего не останется, кроме как валить все на сбой компьютерной системы. Недостаточный объем ОЗУ, возможно, является одной из главных причин отказа серверов.

Внешняя и третичная память

Спустимся еще ниже по иерархической лестнице памяти. Если компьютер подключить к локальной или Глобальной сети, он может получить доступ к памяти, управляемой другими компьютерами. Но этот процесс требует еще больше времени: если на чтение локального диска уходит миллисекунда, то получение данных из сети может занимать сотни миллисекунд. Только на то, чтобы сетевой пакет переместился с одного компьютера на другой, требуется порядка десяти миллисекунд. Если сетевой пакет проходит через Интернет, то

он часто движется намного дольше, от двухсот до трехсот миллисекунд — столько времени у нас уходит, чтобы моргнуть 😉.

В самом низу иерархии находится *третичная память* — устройства хранения, которые не всегда подключены к сети и доступны. Мы можем хранить десятки миллионов гигабайт данных на магнитном носителе или компакт-дисках. Доступ к таким данным, однако, требует, чтобы кто-то взял носитель и вставил его в считывающее устройство. Это может занимать минуты, а может и дни[1]. Третичная память подходит только для архивации данных, к которым редко обращаются.

Тенденции в технологии памяти

Технология, используемая в быстродействующих устройствах памяти (которые находятся в верхней части иерархической лестницы), с большим трудом поддается качественному улучшению. С другой стороны, «медленные» устройства памяти становятся быстрее и дешевле. Стоимость хранения данных на жестком диске десятилетиями падала, и, по всей видимости, эта тенденция будет продолжаться (рис. 7.13).

Кроме того, новые технологии приводят к ускорению дисков. Мы переключаемся с магнитных вращающихся дисков на твердотельные (SSD). Отсутствие подвижных частей делает их быстрее, надежнее и менее затратными в энергетическом плане.

SSD-диски становятся дешевле и быстрее каждый день, но они попрежнему стоят дорого. Некоторые производители выпускают гибридные диски, сочетающие технологии SSD и магнитных дисков. Данные, к которым приходится обращаться часто, хранятся в SSD, а те, что бывают нужны реже, — в более медленном магнитном разделе. Когда последние начинают запрашиваться часто, они копируются в SSD-раздел гибридного диска — точно так же, как в случае с ОЗУ и кэшами в процессоре.

---

[1] Не верите? Тогда ближе к вечеру пятницы попросите ИТ-отдел сделать резервную копию магнитных лент.

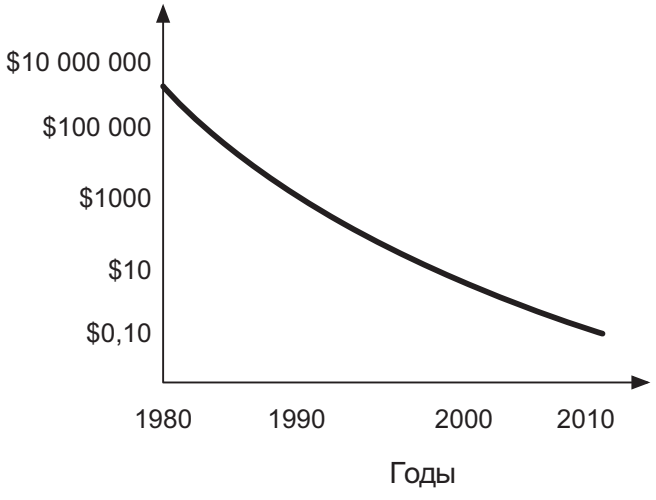


Рис. 7.13. Стоимость дисковой памяти в расчете на гигабайт объема

Подведем итоги

В этой главе мы разобрались с несколькими простыми принципами устройства компьютеров. Мы увидели, что *всё*, поддающееся вычислению, можно выразить в виде простых команд. Мы также узнали, что существует программа, называемая компилятором, которая транслирует наши сложные вычислительные команды в простые, понятные для ЦП. Компьютеры способны делать сложные вычисления просто потому, что их процессоры выполняют огромное количество базовых операций.

Мы узнали, что наши компьютеры имеют быстродействующие процессоры, но относительно медленную память. Доступ к памяти осуществляется не наугад, а согласно пространственной и временной локальностям. Это позволяет использовать более быстрые типы памяти для кэширования тех данных, доступ к которым производится наиболее часто. Мы проследили применение этого принципа на нескольких уровнях кэширования: от кэша L1 вниз по иерархической лестнице вплоть до третичной памяти.

Принцип кэширования, речь о котором шла в этой главе, применим во многих сценариях. Идентификация частей данных, которые используются вашим приложением чаще, и ускорение доступа к ним — одна из наиболее широко используемых стратегий, предназначенных для ускорения компьютерных программ.

Полезные материалы

- *Таненбаум Э., Остин Т.* Архитектура компьютера. — СПб.: «Питер», 2017.
- Современная реализация компилятора на C (Modern Compiler Implementation in C, Appel, https://code.energy/appel).

Глава 8

ПРОГРАММИРОВАНИЕ

> Когда кто-то скажет: «Мне нужен язык программирования, в котором достаточно только сказать, что мне нужно сделать», — дайте ему леденец на палочке.
>
> *Алан Перлис*

Мы хотим, чтобы компьютеры нас понимали. Вот почему мы выражаем наши предписания на языке программирования: это язык, который машина поймет. Вы не можете просто взять и сказать на языке Шекспира или Пушкина, что компьютер должен сделать, — если только вы не взяли на работу программиста либо не попали в сюжет научно-фантастического фильма. Пока что только программисты обладают неограниченными полномочиями сообщать машине, что ей делать. По мере углубления ваших познаний в языках программирования ваши возможности как программиста будут расти. В этой главе вы научитесь:

определять *лингвистику*, которая управляет программным кодом;

хранить вашу драгоценную информацию внутри *переменных*;

обдумывать решения в условиях разных *парадигм*.

Мы не будем вдаваться в синтаксический и грамматический формализм. Расслабьтесь и продолжайте читать!

8.1. Лингвистика

Языки программирования очень сильно отличаются, но все они были созданы, чтобы делать одно: управлять информацией. С этой целью все они опираются на три основных структурных элемента. *Значение* представляет информацию. *Выражение* производит значение. *Инструкция* использует значение, чтобы дать команду компьютеру.

Значения

Вид информации, которую может содержать значение, варьируется от языка к языку. В самых элементарных языках значения содержат только очень простые данные, такие как целое число или число с плавающей точкой. Со временем языки становились сложнее: сперва они стали в качестве значений обрабатывать символы, потом — строки. В языке C, который по-прежнему остается очень низкоуровневым, можно задать структуру — способ определения значений, состоящих из групп других значений. Например, можно определить тип значения, именуемый координатой, которое будет состоять из двух чисел с плавающей точкой: широты и долготы.

Значения настолько важны, что их также называют объектами первого класса языка программирования. Языки допускают разнообразные виды операций со значениями: они могут создаваться во время выполнения функции, могут передаваться как параметры, возвращаться ею.

Выражения

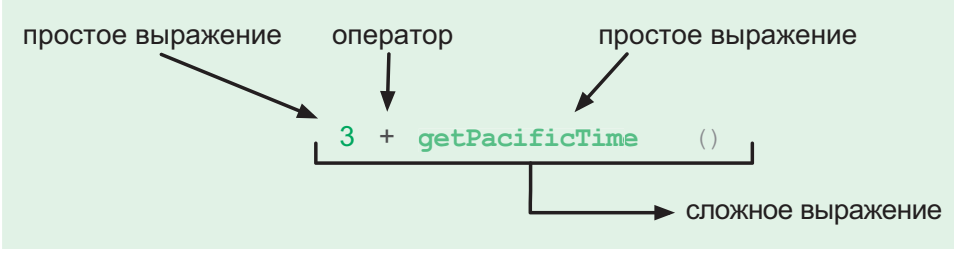
Вы можете создать значение двумя способами: написав *литерал* либо вызвав *функцию*. Вот пример выражения с литералом:

Бум! Мы буквально только что создали значение 3, написав: «3». Довольно прямолинейно. Как литералы можно создавать и другие типы значений. Большинство языков программирования позволит вам создать строковое значение *Привет мир*, набрав на клавиатуре «`Привет мир`». Функции же генерируют значение согласно методу или процедуре, которые запрограммированы в каком-то другом месте. Например:

```
getPacificTime()
```

Это выражение создало значение, равное текущему времени в Лос-Анджелесе. Если сейчас 4 часа утра, то метод вернет 4.

Еще одним базовым элементом любого языка программирования является *оператор*. Оператор может объединять простые выражения для формирования более сложных. Например, оператор `+` позволяет создать значение, равное времени в Нью-Йорке:



Когда в Лос-Анджелесе 4 часа утра, наше выражение сведется к 7. В действительности выражение — это любая запись, которую компьютер сможет свести к единственному значению. Большие выражения могут сочетаться с другими выражениями посредством операторов, формируя еще более крупные выражения. В конечном счете даже самое сложное выражение всегда будет вычислено и сведено к единственному значению.

Наряду с литералами, операторами и функциями выражения могут также содержать круглые скобки. Они позволяют управлять *порядком*

выполнения операторов: $(2 + 4)^2$ сводится к 6^2, которое, в свою очередь, сводится к 36. Выражение $2 + 4^2$ сводится к 2 + 16 , а затем к 18.

Инструкции

В то время как выражение представляет значение, инструкция используется, чтобы дать компьютеру команду *сделать* что-то. Например, эта инструкция заставит его показать сообщение: `print("привет мир")`.

Рис. 8.1.[1]

Более сложные примеры включают условная инструкция *if*, инструкции циклов *while* и *for*. Разные языки программирования поддерживают разные типы инструкций.

[1] Любезно предоставлено http://geek-and-poke.com.

Определения. Некоторые языки программирования имеют специальные инструкции, именуемые *определениями*. Они изменяют состояние программы, добавляя не существовавшие ранее объекты, такие как новые значения или функции[1]. Чтобы обратиться к объекту, который мы определили, мы должны назвать его. Этот процесс называется *привязкой имен*. Например, имя `getPacificTime` должно быть привязано к определению функции, заданному где-то в другом месте.

8.2. Переменные

Переменные — это самая важная привязка имен: она устанавливает отношения между именами и значениями. Переменная связывает имя с адресом памяти, где значение хранится, и, таким образом, выступает в качестве его псевдонима. Чаще всего переменная создается при помощи оператора присваивания. В псевдокоде этой книги присвоения обозначаются символом ←:

```
pi ← 3.142
```

В большинстве языков программирования присвоения записываются при помощи символа `=`. Некоторые языки даже требуют, чтобы вы *объявляли* имя как переменную, перед тем как она будет определена. В итоге у вас получится нечто вроде этого:

```
var pi
pi = 3.142
```

Эта инструкция резервирует блок памяти, записывает в него значение 3,142 и привязывает имя `"pi"` к адресу блока памяти.

---

[1] Иногда такие сущности могут быть импортированы из заранее созданных внешних библиотек.

Типизация переменных

В большинстве языков программирования переменные должны иметь присвоенный тип (например, целочисленный, с плавающей точкой либо строковый). Благодаря этому программа знает, как она должна интерпретировать единицы и нули, которые она читает из блока памяти, отведенной для переменной. Это помогает определять ошибки. Если одна переменная имеет строковый тип, а другая — целочисленный, то нет никакого смысла их складывать.

Существуют два способа проверки типа: статический и динамический. Статическая проверка требует, чтобы разработчик кода объявлял тип каждой переменной перед ее использованием. Например, языки программирования вроде C и C++ вынуждают нас писать:

```
float pi;
pi = 3.142;
```

Такое объявление сообщает, что переменная с именем `pi` может хранить только данные, представляющие числа с плавающей точкой. Статически типизированные языки могут применять дополнительную оптимизацию во время компиляции кода и обнаруживать потенциальные ошибки еще до первого запуска программы. Однако объявление типов для всех переменных может быстро наскучить.

Некоторые языки предпочитают проверять типы динамически. Благодаря такой проверке любая переменная может хранить любой тип значения, и потому объявление типа не требуется. Однако во время выполнения кода производится дополнительная проверка типов переменных, чтобы гарантировать, что все операции между ними имеют смысл.

Область видимости переменных

Если бы все привязки имен были доступны и допустимы во всех точках в коде, то программирование считалось бы чрезвычайно

трудным процессом. По мере того как программы становятся больше, одинаковые имена переменных (такие как `time`, `length` либо `speed`) все чаще начинают использоваться в разных частях программного кода.

Например, я могу определить переменную `length` в двух точках в моей программе, не заметив этого, и в итоге получу ошибку. Что еще хуже, я могу импортировать библиотеку, которая также использует переменную `length`, и тогда `length` из моего кода будет конфликтовать с `length` из библиотеки.

Если ограничить участки кода, где действует привязка имени, это позволит избежать такого рода конфликтов. *Область видимости* переменной определяет, где она действует и может использоваться. Большинство языков устроены таким образом, что переменная действует только внутри функции, где она была определена.

Текущий *контекст*, или *окружение*, — это набор всех привязок имен, которые имеются в программе в конкретной точке. Обычно переменные, определенные внутри контекста, немедленно удаляются и высвобождают память компьютера, как только поток выполнения покидает этот контекст. Хоть такое и не рекомендуется, вы можете обходить данное правило и создавать переменные, которые всегда доступны *где угодно* в вашей программе. Они называются *глобальными переменными*.

Коллекция всех имен, доступных глобально, составляет *пространство имен*. Вы должны внимательно следить за пространством имен своих программ. Оно должно быть как можно меньше. В больших пространствах выше вероятность появления конфликтов имен.

Добавляя новые имена в свое пространство, старайтесь минимизировать их число. Например, импортируя внешний модуль, добавляйте только имена функций, которые собираетесь использовать. Хорошие модули должны добавлять в пространство имен пользователя как можно меньше. Добавление ненужных элементов в этом случае вызывает проблему, известную как *загрязнение пространства имен*.

8.3. Парадигмы

Парадигма — это конкретный набор понятий и методов, обозначающий область науки. Парадигма сориентирует вас, каким образом подойти к задаче, какие приемы использовать, и подскажет структуру решения. Например, ньютоновская и релятивистская школы — это две разные парадигмы физики.

В программировании, как и в физике, подходы к решению задач полностью меняются в зависимости от парадигмы. *Парадигма программирования* — это определенная точка зрения на стиль программирования и методику.

В своем коде вы можете использовать одну или несколько парадигм. Лучше всего придерживаться тех парадигм, на которых основан используемый вами язык. В 1940-х годах первые компьютеры программировались вручную с помощью переключателей для вставки единиц и нулей в память компьютера. Программирование никогда не прекращало эволюционировать, и парадигмы возникли, чтобы расширить возможности людей в создании более эффективного, сложного и быстрого кода.

Существуют три основные парадигмы программирования: императивная, декларативная и логическая. К сожалению, большинство разработчиков учатся правильно работать только с первой. Очень важно знать обо всех трех, это позволит вам извлечь пользу из функциональных особенностей и перспектив, которые предлагает каждый язык программирования. Благодаря этому вы сможете программировать с максимальной эффективностью.

Императивное программирование

Парадигма императивного программирования подразумевает создание списка конкретных команд, описывающих, что именно должен делать компьютер на каждом шаге. Каждая команда изменяет состояние компьютера. Команды, составляющие программу, выполняются поочередно одна за другой.

Она была самой первой парадигмой программирования, поскольку является естественным отражением способа работы наших компьютеров. Вычисления всегда делаются при помощи команд ЦП, которые выполняются одна за другой. В конечном счете каждая компьютерная программа выполняется компьютерами в рамках данной парадигмы.

Рис. 8.2. Типичная задача[1]

Императивное программирование — это, безусловно, самая известная парадигма. По сути, многие программисты знакомы только с ней. Она также является естественным отражением способа, которым работают люди: мы используем эту парадигму для описания кулинарного рецепта, плана ремонта и других повседневных процедур. Когда нам лень выполнять скучную работу, мы переносим инструкции в программу, и компьютер выполняет их за нас. Программистская лень серьезно помогла прогрессу.

Программирование машинного кода. Первым программистам приходилось вводить код в компьютер вручную, используя единицы и нули, но они тоже были ленивы. Они решили, что будет намного лучше записывать последовательность команд ЦП при помощи мнемоник, таких как `CP` для команды «копировать», `MOV` для команды «переместить», `CMP` для команды «сравнить» и т. п. Затем они написали программу, преобразующую мнемонический код в соответству-

[1] Источник: http://xkcd.com.

ющие ему двоичные числа процессорного кода. Так родился язык *ассемблера* (или *ASM*).

Программа, написанная с использованием этого мнемокода, гораздо понятнее для человека, чем соответствующий набор единиц и нулей. Старинный мнемокод и стиль программирования по-прежнему широко используются. По мере того как более совершенные ЦП начинали поддерживать новые команды, создавался дополнительный мнемокод, но основной принцип оставался неизменным.

ASM используется, например, для программирования микроволновых печей или компьютерных систем в автомобиле. Этот язык также идеален для создания частей программы, где необходима предельная производительность и имеет значение экономия даже нескольких циклов ЦП.

Например, представьте, что вы занимаетесь оптимизацией высокопроизводительного веб-сервера и столкнулись с серьезным узким местом. Вы можете переписать его на ASM, проинспектировать, а затем раз за разом изменять, чтобы уменьшить число используемых команд. Некоторые языки поддерживают вставку в свой код фрагментов на машинном языке для такой тонкой оптимизации. Поддержка машинного кода дает вам неограниченный контроль над тем, что именно и как будет делать процессор.

Структурное программирование. Когда-то давно программисты использовали команду `GOTO` для управления потоком выполнения. Она заставляет процесс перепрыгивать в другую часть кода. По мере того как программы усложнялись, стало почти невозможно понимать, что она делает. Различные потоки выполнения переплетались с командами `GOTO` и `JUMP`, создавая то, что называется *запутанным кодом* или спагетти-кодом[1]. В 1968 году Дейкстра написал свой знаменитый манифест «О вреде оператора GOTO», и это вызвало революцию. Программный код стали разделять на логические части. Вместо ситуативных `GOTO` программисты начали использовать управляющие

[1] Если вы хотите обругать чей-то исходный код, скажите, что это спагетти  .

структуры (`if`, `else`, `while`, `for`). Это позволило намного упростить написание и отладку программ.

Процедурное программирование. Следующим шагом в искусстве программирования стало процедурное программирование. Оно позволяет организовать код в *процедуры*, избежать повторов и сделать более удобным его многократное использование. Например, вы можете создать функцию, преобразующую единицы метрической системы мер в единицы британской системы, принятые в США, а затем вызывать свою функцию, многократно используя один и тот же код, когда это потребуется. Процедуры усовершенствовали структурное программирование еще больше. Их использование сильно упростило разбиение связанных кусков программного кода на группы.

Декларативное программирование

Парадигма декларативного программирования позволяет объявить желаемый результат, не разбираясь с каждым отдельным шагом, ведущим к нему. Эта парадигма связана с объявлением того, *что* (а не *как*) вы хотите сделать. Во многих ситуациях она позволяет сильно сократить и упростить программы. Кроме того, нередко их бывает легче читать.

Функциональное программирование. В парадигме функционального программирования функции — это больше, чем просто процедуры. Они используются для объявления связи между двумя или более элементами, почти как математические уравнения. В функциональной парадигме первоклассными объектами являются функции. Они обрабатываются так же, как любой другой примитивный тип данных, например строки и числа.

Функции могут получать другие функции в аргументах и возвращать функции в виде результата. Функции, имеющие такие признаки, называются *функциями высшего порядка*. Многие основные языки программирования включают такие элементы из функциональной парадигмы. Вам следует непременно воспользоваться их выразительностью при первой возможности.

Например, большинство языков функционального программирования поставляются вместе с универсальной функцией `sort`. Она может сортировать любую последовательность элементов. Функция `sort` на входе принимает другую функцию, которая определяет, как элементы будут сравниваться в процессе сортировки. Например, переменная `coordinates` содержит список географических точек. При наличии двух точек функция `closer_to_home` сообщает, какая из них находится ближе к вашему дому. Вы можете отсортировать список точек по критерию близости к вашему дому, как это сделано тут:

```
sort(coordinates, closer_to_home)
```

Функции высшего порядка часто используются для фильтрации данных. Языки функционального программирования также предлагают универсальную функцию `filter`, получающую набор элементов, и функцию, которая указывает, следует ли отбросить заданный элемент или нет. Например, удаление четных чисел из списка можно записать так:

```
odd_numbers ← filter(numbers, number_is_odd)
```

`number_is_odd` — это функция, которая получает число и возвращает `True`, если число является нечетным, и `False` в противном случае.

Еще одна типичная задача, которая возникает во время программирования, — применение специальной функции ко всем элементам в списке. В функциональном программировании она называется *отображением*. Многие языки имеют встроенную функцию `map`, предназначенную для этой задачи. Например, вычисление квадрата каждого числа в списке можно организовать так:

```
squared_numbers ← map(numbers, square)
```

Функция `square` возвращает квадрат заданного числа. Операции отображения и фильтрации встречаются так часто, что многие языки программирования предлагают возможность записи этих выраже-

ний в более простой форме. Например, в языке программирования Python вычислить квадраты чисел в списке можно так:

```
squared_numbers = [x**2 for x in numbers]
```

Эта форма записи называется *«синтаксическим сахаром»*: дополнительной синтаксической конструкцией, позволяющей записывать выражения короче и понятнее. Многие языки программирования предоставляют несколько форм «синтаксического сахара». Применяйте их и злоупотребляйте ими.

Наконец, когда нужно обработать список значений так, чтобы свести процесс к единственному результату, вы можете воспользоваться функцией `reduce`. На входе она получает список, начальное значение и редуцирующую функцию. Начальное значение инициирует «аккумуляторную» переменную, которая будет обновляться редуцирующей функцией для каждого элемента в списке, а в конце — возвращена:

```
function reduce(list, initial_val, func)
    accumulator ← initial_val
    for item in list
        accumulator ← func(accumulator, item)
    return accumulator
```

Например, с помощью `reduce` можно просуммировать элементы в списке:

```
sum ← function(a, b): a + b
summed_numbers ← reduce(numbers, 0, sum)
```

Использование функции `reduce` упростит ваш программный код и сделает его более читаемым. Еще пример: если `sentences` — это просто список предложений, и вы хотите подсчитать общее количество слов в них, это можно реализовать так:

```
wsum ← function(a, b): a + length(split(b))
number_of_words ← reduce(sentences, 0, wsum)
```

Функция `split` разбивает строку на список слов, а функция `length` подсчитывает количество элементов в списке.

Функции высшего порядка могут не только принимать функции на входе, но также порождать и возвращать новые функции. Они даже в состоянии *замкнуть* ссылку на значение в сгенерированную функцию. Мы называем это *замыканием*. Функция, имеющая замыкание, «помнит» окружение, в котором была создана, и может обращаться к заключенным в нем значениям.

Используя замыкания, можно разбить исполнение функции, принимающей множество аргументов, на несколько шагов. Это называется *каррингом*. Предположим, что ваш программный код имеет такую функцию `sum`:

```
sum ← function(a, b): a + b
```

Функция `sum` ожидает два параметра, но ее можно вызвать с одним аргументом. Выражение `sum(3)` вернет не число, а новую *каррированную* функцию. При обращении к ней она вызовет `sum` и передаст ей `3` в первом аргументе. Ссылка на значение `3` замыкается в каррированной функции. Например:

```
sum_three ← sum(3)
print sum_three(1) # печатает "4".

special_sum ← sum(get_number())
print special_sum(1) # печатает "get_number() + 1".
```

Обратите внимание, что `get_number` не будет вызвана при создании функции `special_sum`. Вместо этого в определение `special_sum` будет заключена ссылка на `get_number`. Функция `get_number` вызывается только при вызове функции `special_sum`. Такой подход называется *ленивыми*, или *отложенными, вычислениями*, это очень важная особенность языков функционального программирования.

Замыкания также используются для генерации набора связанных функций, соответствующих шаблону. Использование шаблона функ-

ции поможет сделать код более читаемым и избежать дублирования. Давайте посмотрим на пример:

```
function power_generator(base)
    function power(x)
        return power(x, base)
    return power
```

Мы можем использовать `power_generator` для генерации разных функций, которые вычисляют степень:

```
square ← power_generator(2)
print square(2)    # печатает 4.

cube ← power_generator(3)
print cube(2)      # печатает 8.
```

Обратите внимание, что возвращаемые функции `square` и `cube` сохраняют значение переменной `base`. Она существовала только в среде `power_generator`, но несмотря на это возвращаемые функции абсолютно независимы от `power_generator`. Еще раз: замыкание — это функция, которая имеет доступ к некоторым переменным *за пределами* собственного контекста.

Еще замыкания можно использовать для управления внутренним состоянием функции. Давайте предположим, что вам нужна функция, которая накапливает сумму всех переданных ей чисел. Для этого можно использовать глобальную переменную:

```
GLOBAL_COUNT ← 0
function add(x)
    GLOBAL_COUNT ← GLOBAL_COUNT + x
    return GLOBAL_COUNT
```

Как вы уже знаете, глобальных переменных следует избегать, потому что они загрязняют пространство имен программы. Более чистый подход состоит в использовании замыкания, включающего ссылку на аккумуляторную переменную:

```
function make_adder()
    n ← 0
    function adder(x)
        n ← x + n
        return n
    return adder
```

Это позволит нам создать несколько сумматоров, не используя глобальные переменные:

```
my_adder ← make_adder()
print my_adder(5)     # печатает 5.
print my_adder(2)     # печатает 7 (5 + 2).
print my_adder(3)     # печатает 10 (5 + 2 + 3).
```

Сопоставление с шаблоном. Функциональное программирование позволяет рассматривать функции как математические. При помощи математики мы можем описывать поведение функций в зависимости от входных данных. Обратите внимание на входной шаблон функции факториала:

$0! = 1$

$n! = (n - 1)!$

Функциональное программирование допускает *сопоставление с шаблоном* — то есть процесс распознавания этого шаблона. Вы можете просто написать:

```
factorial(0): 1
factorial(n): n × factorial(n – 1)
```

А вот императивное программирование требует, чтобы вы написали:

```
function factorial(n)
    if n = 0
        return 1
    else
        return n × factorial(n - 1)
```

Какая версия выглядит яснее? Я бы сделал выбор в пользу функциональной версии везде, где это возможно. Некоторые языки программирования *строго* функциональны; весь код на них эквивалентен чистым математическим функциям. Такие языки заходят настолько далеко, что являются вневременными, причем порядок инструкций в коде не влияет на его поведение. В таких языках все присвоенные переменным значения являются неизменяемыми. Мы называем это *однократным присвоением*. Поскольку состояние программы отсутствует, то и нет момента времени, когда переменная может измениться. Вычисления в строгой функциональной парадигме просто сводятся к вычислению функций и сопоставлению с шаблоном.

Логическое программирование

Всегда, когда вашей задачей является решение ряда логических формул, вы можете воспользоваться логическим программированием. Разработчик перечисляет логические высказывания о ситуации, например такие, как в разделе «Логика» главы 1. Затем выполняются запросы, чтобы получить ответы из предоставленной модели. Компьютер отвечает за интерпретацию логических переменных и запросов. Он также создает пространство решений из высказываний и занимается поиском ответов на запросы, которые удовлетворяют всем этим высказываниям.

Самое большое преимущество парадигмы логического программирования состоит в том, что программирование как таковое здесь сведено к минимуму. Компьютеру даются только факты, инструкции и запросы, а он отвечает за определение лучшего способа поиска в пространстве решений и представление результатов.

Эта парадигма не очень широко используется в господствующей тенденции, но если вы работаете с искусственным интеллектом или занимаетесь обработкой естественного языка, то не забудьте обратить на нее внимание.

Подведем итоги

По мере эволюции методологии компьютерного программирования появлялись все новые парадигмы. Они придавали программному коду выразительность и элегантность. Чем больше вы узнаете о различных парадигмах, тем лучше будете владеть программированием.

В этой главе мы увидели, как программирование эволюционировало от непосредственного ввода единиц и нулей в память компьютера до написания ассемблерного кода. Затем с внедрением управляющих структур, таких как циклы и переменные, оно стало еще проще. Мы увидели, как использование функций позволило лучше организовать программный код.

Мы познакомились с элементами парадигмы декларативного программирования, которое становится популярным в массовых языках. И, наконец, мы упомянули логическое программирование, которое является предпочтительной парадигмой в некоторых очень специфических контекстах.

Хотелось бы надеяться, что у вас хватит смелости заняться каким-либо новым языком программирования. У них у всех есть что предложить вам. Так что закрывайте книгу — и начинайте программировать! 

Полезные материалы

- Основы языков программирования (Essentials of Programming Languages, Friedman, см. https://code.energy/friedman).
- *Макконнелл С.* Совершенный код. Мастер-класс.

Заключение

> Образование в области информатики никого не может сделать хорошим программистом, так же как изучение кистей и красок никого не может сделать хорошим живописцем.
>
> *Эрик Рэймонд*

В книге были представлены самые важные темы computer science в очень простой форме. Это абсолютный минимум, который хороший программист должен знать о computer science.

Я надеюсь, что это новое знание вдохновит вас углубиться в темы, которые вам по душе. Вот почему я по ходу изложения в конце каждой главы давал ссылки на лучшие справочные пособия.

Следует отметить, что в этой книге не затронуты некоторые важные темы. Как добиться, чтобы компьютеры в сети, покрывающей всю планету (Интернет), надежно взаимодействовали друг с другом? Как сделать так, чтобы несколько процессоров работали синхронно для ускоренного решения вычислительной задачи? Одна из самых важных парадигм программирования, объектно-ориентированная, также осталась за бортом. Я планирую обратиться к этим недостающим частям в следующей книге.

Кроме того, вам придется заняться написанием программ, чтобы полностью изучить то, с чем вы познакомились. И это хорошо. Разработка программного кода может казаться неблагодарным делом поначалу, когда вы совершаете первые шаги. Как только вы изучите основы, я обещаю, программирование начнет приносить вам огром-

ное удовлетворение. Давайте заканчивайте читать и начинайте программировать.

Напоследок я хотел бы заметить, что это мой первый опыт в написании книги. Я понятия не имею, насколько хорошо у меня получилось. Вот почему ваши отзывы о книге представляют для меня невероятную ценность. Что вам в ней понравилось? Какие части сбили с толку? Как, по вашему мнению, ее можно было бы улучшить? Пишите мне на hi@code.energy.

Приложения

I. Системы счисления

Вычисления сводятся к работе с числами, потому что информация выражается в числах. Если сопоставить числа с буквами, можно будет записывать текст в цифровой форме. Цвета являются комбинацией интенсивности световых потоков красного, синего и зеленого — эту интенсивность можно задать в числовых значениях. Изображения легко представить в виде мозаики из цветных квадратов, так что они тоже выражаются через числа.

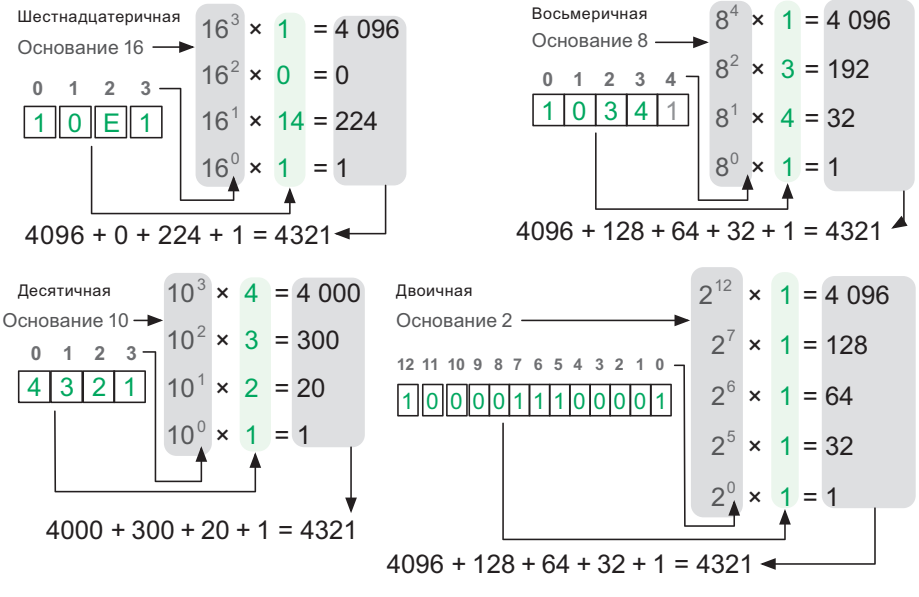


Рис. I.1. Число 4321 в разных системах счисления

Архаичные системы счисления (например, римские цифры — I, II, III и т. д.) составляют числа из сумм цифр. Система счисления, которая используется сегодня, тоже опирается на суммы, но значение каждой цифры в позиции i умножается на d в степени i, где d — это некое число. Его называют *основанием системы счисления*. Мы обычно используем $d = 10$, потому что у нас десять пальцев, но система работает для любого основания d.

II. Метод Гаусса

Рассказывают, что как-то раз учитель в начальной школе в качестве наказания поставил Гауссу задачу: просуммировать все числа от 1 до 100. К изумлению учителя, Гаусс нашел ответ (5050) в течение нескольких минут. Его прием состоял в том, чтобы манипулировать порядком элементов *удвоенной* суммы:

$$2\times\sum_{i=1}^{100} i = (1+2+\ldots+99+100)+(1+2+\ldots+99+100)$$

$$=\underbrace{(1+100)+(2+99)+\ldots+(99+2)+(100+1)}_{\text{100 удвоений}}$$

$$=\underbrace{101+101+\ldots+101+101}_{\text{100 раз}}$$

$$= 10100$$

Разделив это на 2, мы получим 5050. Мы можем записать так:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n} (n+1-i).$$

Следовательно:

$$2\times\sum_{i=1}^{n} i = \sum_{i=1}^{n} i + \sum_{i=1}^{n} (n+1-i)$$

$$=\sum_{i=1}^{n} (i+n+1-i)$$

$$=\sum_{i=1}^{n} (n+1).$$

В последней строке i отсутствует, поэтому $(n + 1)$ суммируется снова и снова n раз. Следовательно:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

III. Множества

Мы используем слово *множество* для описания группы объектов. Например, мы можем назвать S множеством обезьянок-эмодзи:

$S = \{$🐵, 🙈, 🙉, 🙊$\}$.

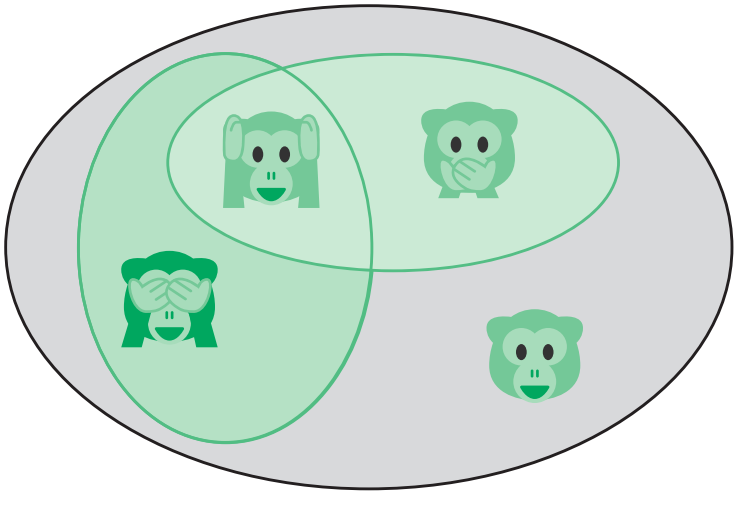


Рис. III.1. S_1 и S_2 есть подмножества S

Подмножества. Множество объектов, содержащихся в другом множестве, называется *подмножеством*. Например, обезьянки, показывающие лапы и глаза, составляют подмножество $S_1 = \{$🙈, 🙊$\}$. Все обезьянки в S_1 содержатся в S. Мы записываем это так: $S_1 \subset S$ Мы можем сгруппировать обезьянок с лапками и ртами в другом подмножестве: $S_2 = \{$🙉, 🐵$\}$.

Объединение. Какие обезьянки принадлежат либо S_1, либо S_2? Ответ: обезьянки в $S_3 = \{$🙈, 🙉, 🙊$\}$. Новое множество — объединение двух предыдущих. Мы записываем это так: $S_3 = S_1 \cup S_2$.

Пересечение. Какие обезьянки принадлежат и S_1, и S_2? Ответ: обезьянки в $S_4 = \{$🐵$\}$. Новое множество получается путем пересечения двух предыдущих. Мы записываем это так: $S_4 = S_1 \cap S_2$.

Степенные множества. Обратите внимание, что S_3 и S_4 одновременно являются подмножествами S. Мы также полагаем, что $S_5 = S$ и пустое множество $S_6 = \{\}$ являются подмножествами S. Если подсчитать все подмножества S, то вы найдете $2^4 = 16$ подмножеств. Если же рассматривать их все как объекты, то мы можем собрать их в множество. Множество всех подмножеств S называется его *степенным множеством*:

$$P_S = \{S_1, S_2, \ldots, S_{16}\}.$$

IV. Алгоритм Кэдейна

В разделе «Полный перебор» главы 3 мы представили задачу «Лучшая сделка».

> **Лучшая сделка** $ У вас есть список цен на золото по дням за какой-то интервал времени. В этом интервале вы хотите найти такие два дня, чтобы, купив золото, а затем продав его, вы получили бы максимально возможную прибыль.

В разделе «Динамическое программирование» той же главы мы показали алгоритм, который решил эту задачу с временной сложностью $O(n)$ и пространственной сложностью $O(n)$. Когда в 1984 году Джей Кэдейн обнаружил эту задачу, он нашел способ решить ее с $O(n)$ по времени и $O(1)$ по пространству:

```
function trade_kadane(prices):
    sell_day ← 1
    buy_day ← 1
    best_profit ← 0
    for each s from 2 to prices.length
        if prices[s] < prices[buy_day]
            b ← s
        else
            b ← buy_day
        profit ← prices[s] - prices[b]
        if profit > best_profit
            sell_day ← s
            buy_day ← b
            best_profit ← profit
    return (sell_day, buy_day)
```

Дело в том, что нам незачем хранить день лучшей покупки для каждого дня на входе. Достаточно сохранить день лучшей покупки относительно дня лучшей продажи, найденной к настоящему моменту.

Владстон Феррейра Фило

Теоретический минимум по Computer Science. Все, что нужно программисту и разработчику

Перевел с английского А. Логунов

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*А. Киселев*
Литературный редактор	*А. Петров*
Художественный редактор	*С. Петров*
Корректоры	*Н. Сидорова, Г. Шкатова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 191123, Россия, г. Санкт-Петербург,
ул. Радищева, д. 39, к. Д, офис 415. Тел.: +78127037373.

Дата изготовления: 03.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларуси: ООО «ПИТЕР М», РБ, 220020, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./ факс 208 80 01.

Подписано в печать 14.03.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 18,060. Тираж 1700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87