Глава 1

Введение

> В основе мироздания лежат две идеи: исчисление и алгоритм. Исчисление, базирующееся на мощном аппарате математического анализа, является основой современной науки. Однако наука невозможна без алгоритма, лежащего в основе современного мира.
>
> — Дэвид Берлински (David Berlinski),
> "Появление алгоритма" ("The Advent of the Algorithm") (2000)

Зачем изучать алгоритмы? Для настоящего компьютерного профессионала существует две причины: практическая и теоретическая. С практической точки зрения он должен иметь представление о стандартном наборе основных алгоритмов, относящихся к разным областям вычислительной техники. Кроме того, он должен уметь разрабатывать новые алгоритмы и анализировать их эффективность. С теоретической точки зрения процесс изучения алгоритмов, который иногда называют ***алгоритмикой*** (algorithmics), считается краеугольным камнем информатики (computer science). Дэвид Харел (David Harel) в своей великолепной книге, остроумно озаглавленной *Algorithmics: the Spirit of Computing*[1], пишет следующее:

> Алгоритмика — это нечто большее, чем просто раздел информатики. Она является основой информатики и, положа руку на сердце, можно сказать, что она существенно повлияла на современную науку, технику и бизнес [[48], с. 6].

И даже если вы не являетесь студентом вуза, специализирующимся в компьютерных науках, существуют достаточно веские причины для того, чтобы заняться изучением алгоритмов. Попросту говоря, без алгоритмов невозможно создать ни одну компьютерную программу. Поэтому по мере того, как компьютерные программы становятся все более значимыми практически во всех сферах профессиональной деятельности (да и личной жизни!), изучение алгоритмов становится все более и более важной задачей для широкого круга людей.

[1] Это название можно перевести на русский язык как *Алгоритмика: дух вычислений*. — *Прим. ред.*

Еще одна причина для изучения алгоритмов заключается в том, что этот процесс развивает у учащихся умение аналитически мыслить. В конце концов, алгоритмы можно рассматривать как особый подход к решению задач, когда важны не столько сами ответы, сколько точные инструкции для их получения. Следовательно, специальные методы проектирования алгоритмов можно считать стратегическим планом решения задачи, который пригодится в любом случае, независимо от того, используется компьютер или нет. Само собой разумеется, что круг задач, которые могут быть решены с помощью какого-либо алгоритма, по существу зависит от точности алгоритмического мышления. Например, невозможно сформулировать алгоритм, который позволит прожить счастливую жизнь или стать богатым и знаменитым. С другой стороны, выдвинутое требование точности имеет важное преимущество с точки зрения обучения. Дональд Кнут, один из выдающихся ученых в области информатики и истории алгоритмики, пишет следующее:

> Хорошо обученный в области информатики специалист обязан знать, как работать с алгоритмами: как их создавать, изменять, понимать и анализировать. Эти знания позволят не только писать хорошие компьютерные программы, но и станут основой универсального мыслительного аппарата, который окажет неоценимую помощь при постижении других наук, будь то химия, лингвистика, музыка и т.д. Причину этого можно объяснить следующим образом: часто говорят, что человек ничего не понимает, пока не объяснит это кому-то другому. Я бы перефразировал это так: человек **глубоко** не понимает предмет до тех пор, пока не научит этому **компьютер**, т.е. выразит что-либо в виде алгоритма... Попытка формализовать нечто в виде набора алгоритмов приводит к более глубокому пониманию сути вещей, чем при их осмыслении традиционным способом [[64], с. 9].

О том, что такое алгоритм речь пойдет в разделе 1.1. Там в качестве примеров мы рассмотрим три алгоритма решения одной и той же задачи — поиска наибольшего общего делителя (НОД). Я выбрал эту задачу по трем причинам. Во-первых, она знакома практически каждому со времени обучения в средней школе. Во-вторых, она позволяет продемонстрировать важный принцип: то, что одну и ту же задачу можно решить с помощью нескольких алгоритмов. Вполне естественно, что эти алгоритмы отличаются по своей сути, уровню сложности и эффективности. В-третьих, один из этих алгоритмов вполне достоин того, чтобы быть представленным первым, — как по “возрасту” (впервые он был описан в известном трактате Евклида более двух тысяч лет тому назад), так и по неисчерпаемым возможностям и важности. Наконец, метод определения НОД, которому учат в школе, позволит продемонстрировать одно из важных условий, которому должен удовлетворять любой алгоритм.

В разделе 1.2 рассматривается решение алгоритмической задачи. Там мы обсудим несколько важных моментов, связанных с разработкой и анализом алго-

ритмов. К различным аспектам решения алгоритмической задачи относятся как анализ задачи и средства выражения алгоритма, так и методы оценки его правильности и эффективности. В этом разделе вы не найдете описания магического средства для создания алгоритма решения *любой* задачи. Должно быть уже понятно, что такого средства попросту не существует. Тем не менее материал раздела 1.2 поможет вам систематизировать проектирование и анализ алгоритмов.

Раздел 1.3 посвящен некоторым типам задач, считающимся особенно важными при изучении алгоритмов, и их применению. На самом деле существует ряд учебников, материал которых структурирован по типу решаемых задач. Однако мы придерживаемся иного мнения, с которым согласны многие преподаватели: структурирование материала на основе метода проектирования алгоритма — гораздо важнее. Тем не менее очень важно разбираться в основных типах задач и не только потому, что они находят широкое применение в реальной жизни. В этой книге они используются для демонстрации особенностей метода проектирования алгоритмов.

В разделе 1.4 приведен обзор основных структур данных. Однако его следует рассматривать скорее как справочник, а не подробное руководство. Если же вам нужна детальная информация по этой теме, обратитесь к дополнительной литературе. Хороших книг, посвященных этой теме, написано довольно много, однако в большинстве из них акцент сделан на том или ином языке программирования.

1.1 Понятие алгоритма

Что же такое алгоритм? Универсального определения этого понятия нет, однако существует общее мнение по поводу того, что оно должно означать:

> **Алгоритм** — это последовательность четко определенных инструкций, предназначенных для решения некоторой задачи. Другими словами, это последовательность команд, позволяющих получить из корректных входных данных требующиеся выходные данные за ограниченный промежуток времени.

Это определение можно проиллюстрировать с помощью простой схемы (рис. 1.1).

Упомянув в приведенном выше определении слово “инструкции”, мы подразумевали, что существует некоторое абстрактное устройство (или человек), способное распознать эти инструкции и выполнить предписываемые ими действия. На рис. 1.1 это устройство названо “вычислительным”. Однако следует иметь в виду, что до появления компьютеров под термином “вычислительное устройство” понимался человек, выполняющий числовые расчеты. Естественно, если речь идет о современности, под словом “вычислительное устройство” понимается компьютер, т.е. популярное электронное устройство, которое практически повсеместно

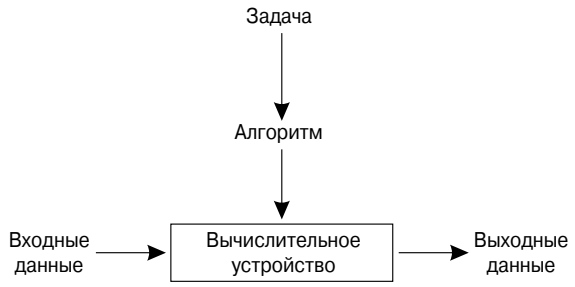


Рис. 1.1. Иллюстрация понятия алгоритма

вторглось в нашу жизнь. Тем не менее обратите внимание на то, что, хотя бóльшая часть алгоритмов в конечном счете предназначена для реализации на компьютере, само понятие алгоритма никак не связано с этим допущением.

В этом разделе в качестве примеров, иллюстрирующих понятие алгоритма, мы рассмотрим три способа решения одной и той же задачи — поиска НОД двух целых чисел. Эти примеры помогут нам проиллюстрировать перечисленные ниже важные моменты.

- Каждый шаг алгоритма должен быть четко и однозначно определен. Это требование является обязательным и не должно нарушаться ни при каких обстоятельствах.
- Должны быть точно указаны диапазоны допустимых значений входных данных, которые обрабатываются с помощью алгоритма.
- Один и тот же алгоритм можно представить несколькими разными способами.
- Для решения одной и той же задачи может существовать несколько разных алгоритмов.
- В основу алгоритмов для решения одной и той же задачи могут быть положены совершенно разные принципы, что может существенно повлиять на скорость решения этой задачи.

Обозначим функцию поиска НОД двух неотрицательных целых чисел m и n (причем m и n не могут одновременно равняться нулю) через $\gcd(m, n)$.[2] По определению эта функция должна найти наибольшее целое число, которое делится без остатка как на m, так и на n. Древнегреческий математик Евклид из Александрии (III в до н.э.), который прославился тем, что впервые систематически изложил курс геометрии, описал алгоритм решения этой задачи в одном из своих трудов под названием *Начала*. Выражаясь современным языком, ***алгоритм***

[2] От английского “greatest common divisor”. — *Прим. ред.*

Евклида основан на рекуррентном вычислении следующего равенства:

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Здесь выражение $(m \bmod n)$ является остатком от деления m на n. Выполнение алгоритма заканчивается, когда выражение $(m \bmod n)$ становится равным нулю. Поскольку $\gcd(m, 0) = m$ (понятно, почему?), последнее полученное значение m будет также являться НОД исходных чисел m и n.

Например, вычисление НОД пары чисел (60, 24) можно выполнить следующим образом:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

(Если этот алгоритм не произвел на вас впечатления, попытайтесь определить НОД двух бóльших чисел, например таких, которые используются в задаче 4 упражнения 1.1.)

Ниже приводится более структурированное описание рассматриваемого нами алгоритма.

Вычисление НОД чисел m и n при помощи алгоритма Евклида

Шаг 1 Если $n = 0$, вернуть m в качестве ответа и закончить работу; иначе перейти к шагу 2.

Шаг 2 Поделить нацело m на n и присвоить значение остатка переменной r.

Шаг 3 Присвоить значение n переменной m, а значение r — переменной n. Перейти к шагу 1.

В качестве альтернативы запишем тот же алгоритм в виде псевдокода.

Алгоритм $Euclid(m, n)$

```
// Алгоритм Евклида вычисляет значение функции gcd(m, n)
// Входные данные:  два неотрицательных целых числа m и n,
//                  которые не могут одновременно быть равны нулю
// Выходные данные: наибольший общий делитель чисел m и n
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

Можем ли мы убедиться, что в конечном счете выполнение алгоритма Евклида завершится? Это следует из констатации следующего факта: на каждом шаге итерации значение второго числа пары (n) будет уменьшаться, причем, по определению, оно не может быть меньше нуля. В самом деле, новое значение

числа n, получаемое на следующей итерации в результате вычисления выражения $(m \bmod n)$, будет всегда меньше, чем предыдущее значение числа n. Следовательно, рано или поздно значение второго числа пары станет равным 0, и выполнение алгоритма завершится.

Как и при решении большинства других задач, существует несколько алгоритмов вычисления НОД. Давайте рассмотрим два других способа решения этой задачи. Первый из них основан на подборе наибольшего целого числа — такого, чтобы числа m и n делились на него без остатка. Очевидно, что такой общий делитель не может быть больше наименьшего из чисел пары, которое можно записать как $t = \min\{m, n\}$. Поэтому выполнение алгоритма можно начать с проверки того, делятся ли оба числа, m и n, на t без остатка. Если это так, то число t является ответом; если нет, нужно уменьшить значение t на единицу и снова выполнить проверку. (Можем ли мы убедиться, что в конечном итоге этот процесс завершится?). Например, для рассмотренной выше пары чисел $(60, 24)$, выполнение алгоритма начинается с проверки числа 24, затем — 23 и т.д. до тех пор, пока значение числа t не станет равным 12, после чего алгоритм должен завершить свою работу.

Вычисление НОД чисел m и n методом последовательного перебора

Шаг 1 Присвоить значение функции $\min\{m, n\}$ переменной t.

Шаг 2 Разделить m на t. Если остаток равен нулю, перейти к шагу 3; иначе перейти к шагу 4.

Шаг 3 Разделить n на t. Если остаток равен нулю, вернуть t в качестве ответа и закончить работу; иначе перейти к шагу 4.

Шаг 4 Вычесть 1 из t. Перейти к шагу 2.

Обратите внимание, что в отличие от алгоритма Евклида, рассматриваемый нами алгоритм поиска НОД в описанной выше форме не будет корректно работать, если хотя бы один из его входных параметров равен нулю. Таким образом, этот пример позволяет показать, почему так важно явно определять диапазоны допустимых значений входных параметров алгоритма.

Третий способ поиска НОД должен быть вам знаком из курса средней школы.

Вычисление НОД чисел m и n "школьным" методом

Шаг 1 Разложить на простые множители число m.

Шаг 2 Разложить на простые множители число n.

Шаг 3 Для простых множителей чисел m и n, найденных на шаге 1 и 2, выделить их общие делители. (Если p является общим делителем чисел m и n и встречается в их разложении на простые множители, соответственно, p_m и p_n раз, то при выделении нужно повторить это $\min\{p_m, p_n\}$ раз.)

Шаг 4 Вычислить произведение всех выделенных общих делителей и вернуть его в качестве результата поиска НОД двух указанных чисел.

Таким образом, для рассмотренной выше пары чисел $(60, 24)$, получим:

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$
$$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12.$$

Ностальгия по школьным годам не должна помешать нам сделать вывод о том, что последний из рассмотренных способов поиска НОД гораздо сложнее и медленнее, чем алгоритм Евклида. (Методы определения и сравнения времени выполнения алгоритмов будут описаны в следующей главе.) Но даже не принимая во внимание его низкую эффективность, метод определения НОД, изучаемый в средней школе, нельзя считать законным алгоритмом (по крайней мере в представленном здесь виде). Вы спросите, почему? Все дело в том, что этапы разложения на простые множители не определены однозначно. Для их выполнения требуется иметь список простых чисел, а я почему-то уверен, что школьный учитель не объяснил вам, как составить такой список. Вы, конечно же, можете возразить, что это не повод для подобных придирок. Однако до тех пор, пока этот момент не будет прояснен, мы не сможем, например, написать на каком-либо из языков программирования программу, реализующую этот алгоритм. (Кстати, с этой точки зрения шаг 3 также недостаточно четко определен. Однако его неопределенность гораздо легче прояснить, чем этапы разложения на простые множители. И еще, как вы собираетесь выделять общие элементы из двух списков?)

Итак, подошло время рассмотреть несложный алгоритм генерации последовательности простых чисел, не превышающих произвольно заданного целого числа n. Вероятнее всего он был придуман в древней Греции и поэтому назван ***решетом Эратосфена*** (прим. 200 год до н.э.). Для начала составим список, содержащий последовательность целых чисел от 2 до n, из которого затем мы должны будем выбрать простые числа. Далее, на первом проходе алгоритма нужно удалить из списка все числа, которые делятся на 2, т.е. 4, 6 и т.д. Затем необходимо выбрать из списка следующий элемент (в данном случае это 3) и удалить из списка все числа, которые делятся на него. (В описываемой простой версии алгоритма существует небольшая накладка, поскольку некоторые из чисел, например 6, должны удаляться из списка более одного раза.) Для числа 4 не нужно выполнять специальный проход по списку, так как число 4 кратно 2, поэтому оно уже будет удалено из списка на первом проходе. (Точно так же в этом алгоритме не нужно выполнять проход и для всех других чисел, которые были удалены из списка на предыдущих проходах.) Следующим числом, которое осталось в списке и используется на третьем проходе, является 5. Работа алгоритма продолжается так до тех пор, пока

в списке существуют числа, которые можно удалить. Числа, оставшиеся в списке после выполнения алгоритма, являются простыми.

В качестве примера использования описанного выше алгоритма рассмотрим процесс поиска простых чисел, не превышающих $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

Для нашего примера не требуется большего количества проходов по списку, поскольку из него на рассмотренных проходах были удалены все составные числа. Таким образом, в списке остались только упорядоченные простые числа, меньшие или равные 25.

Возникает общий вопрос: при каком наибольшем значении числа p в списке еще остаются кратные ему числа? Прежде чем ответить на него, заметим, что, если p является числом, кратные которому числа должны быть удалены из списка на текущем проходе, то начать рассмотрение следует с числа $p \cdot p$, поскольку все меньшие кратные ему числа — $2p, \ldots, (p-1)\,p$ — уже были удалены из списка на предыдущих проходах. Это наблюдение позволит избежать удаления из списка одного и того же числа более одного раза. Очевидно, что $p \cdot p$ не должно быть больше, чем n, поэтому p не может превышать значения $\sqrt{n}$, округленного до целого числа в нижнюю сторону. На математическом языке это записывается как $\lfloor\sqrt{n}\rfloor$, т.е. с помощью так называемой функции “пол” (округления вниз, floor). В приведенном ниже псевдокоде алгоритма предполагается, что для вычисления используется готовая функция $\lfloor\sqrt{n}\rfloor$. Как альтернативный вариант можно в качестве условия продолжения вычисления цикла проверять значение неравенства $p \times p \leqslant n$.

Алгоритм $Sieve\,(n)$ (решето Эратосфена)

```
// Реализация решета Эратосфена
// Входные данные:   Положительное целое число n ⩾ 2
// Выходные данные: Массив L простых чисел, меньших или
//                   равных n
for p ← 2 to n do
   A[p] ← p
for p ← 2 to ⌊√n⌋ do        // См. абзац перед псевдокодом
   if A[p] ≠ 0              // Элементы, кратные p еще не были
      j ← p * p             // удалены на предыдущих проходах
      while j ⩽ n
         A[j] ← 0           // Пометим этот элемент и все
         j ← j + p          // последующие, кратные p, как удаленные
```

```
// Скопируем оставшиеся элементы массива A в массив
// простых чисел L
i ← 0
for p ← 2 to n do
  if A[p] ≠ 0
    L[i] ← A[p]
    i ← i + 1
return L
```

Итак, теперь можно объединить алгоритм решета Эратосфена с методом, который вы проходили в средней школе, получив совершенно законный алгоритм поиска НОД двух положительных чисел. Обратите внимание, что в этом алгоритме нужно предусмотреть частный случай, когда один или оба входных параметра равны 1. Поскольку математики не считают число 1 простым, то, строго говоря, этот метод и не должен работать в подобном случае.

Прежде чем закончить этот раздел, необходимо сделать еще одно важное замечание. Примеры, рассмотренные в этом разделе, не относятся к математическим задачам, несмотря на то, что большинство этих алгоритмов широко используются, а некоторые даже реализованы в виде компьютерных программ. Умение раскладывать задачи на алгоритмы пригодится в решении повседневных рутинных проблем, возникающих как на работе, так и дома. Вполне возможно, что, осознав важность алгоритмов в современном мире, вам захочется еще больше узнать об этом замечательном двигателе информационного века.

Упражнения 1.1

1. Ознакомьтесь с учениями Аль Хорезми — человека, от имени которого произошло слово “алгоритм”. В частности, узнайте, что общего в словах “алгоритм” и “алгебра”.

2. Известно, что основной целью деятельности патентной системы США является содействие “прикладным искусствам”. Как вы думаете, можно ли запатентовать алгоритмы в этой стране?

3. а) Составьте подробную инструкцию (как это делается при описании алгоритма) вашего возвращения из института домой.

б) Составьте подробный рецепт приготовления вашего любимого блюда (как это делается при описании алгоритма).

4. а) Найдите значение функции $\gcd(31415, 14142)$ при помощи алгоритма Евклида.

б) Оцените, во сколько раз быстрее вычисляется значение функции $\gcd(31415, 14142)$ при помощи алгоритма Евклида по сравнению

с алгоритмом последовательного перебора чисел сверху вниз от значения $\min\{m, n\}$ до значения $\gcd(m, n)$.

5. Докажите, что для любых двух положительных чисел m и n выполняется равенство $\gcd(m, n) = \gcd(n, m \bmod n)$.

6. Что произойдет, если при использовании алгоритма Евклида первое из чисел будет меньше второго? Каково будет максимальное время выполнения алгоритма при подстановке таких входных данных?

7. а) При каких значениях входных параметров m и n в алгоритме Евклида выполняется минимальное количество операций деления, при условии, что $1 \leqslant m, n \leqslant 10$?

б) При каких значениях входных параметров m и n в алгоритме Евклида выполняется максимальное количество операций деления, при условии, что $1 \leqslant m, n \leqslant 10$?

8. а) В своем трактате Евклид описал алгоритм поиска НОД, в котором вместо операций целочисленного деления используются операции вычитания. Запишите этот вариант алгоритма Евклида на псевдокоде.

б) *Игра Евклида* (см. [20]). Напишите на доске два неравных положительных числа. В игре участвуют два игрока. Игроки должны по очереди писать на доске положительное число, равное разности двух чисел, уже написанных на доске. Причем это число должно быть новым, т.е. оно не должно уже находиться на доске. Проигравшим считается тот игрок, кто не сможет написать новое число. Какой игрок, по вашему мнению, имеет преимущество в этой игре: первый или второй?

9. Придумайте алгоритм вычисления функции $\lfloor \sqrt{n} \rfloor$.

10. В ***усовершенствованном алгоритме Евклида*** определяется не только НОД двух положительных чисел m и n (обозначим его через d), но и два числа x и y (не обязательно положительные), такие, что выполняется равенство $mx + ny = d$.

а) Найдите описание усовершенствованного алгоритма Евклида (например, см. [[65], с. 28]) и реализуйте его в виде программы на своем любимом языке программирования.

б) Видоизмените программу так, чтобы она могла находить целочисленное решение *диофантова уравнения* $ax + by = c$ для любых целых коэффициентов a, b и c. (Учтите, что для некоторых комбинаций коэффициентов уравнение может не иметь решения.)

1.2 Основы решения алгоритмической задачи

Начнем этот раздел с повторения важного вывода, сделанного во введении к этой главе:

Алгоритмы можно считать процедурным решением задач.

Причем эти решения являются не столько ответами, сколько точно определенными инструкциями для получения ответов. Именно этот акцент на точности определения конструктивных процедур отличает информатику от других дисциплин, в частности от теоретической математики, в которой обычно ограничиваются доказательством существования решения и (иногда) исследованиями характеристик решения.

А теперь перечислим и кратко опишем последовательность этапов проектирования и анализа алгоритмов (рис. 1.2).

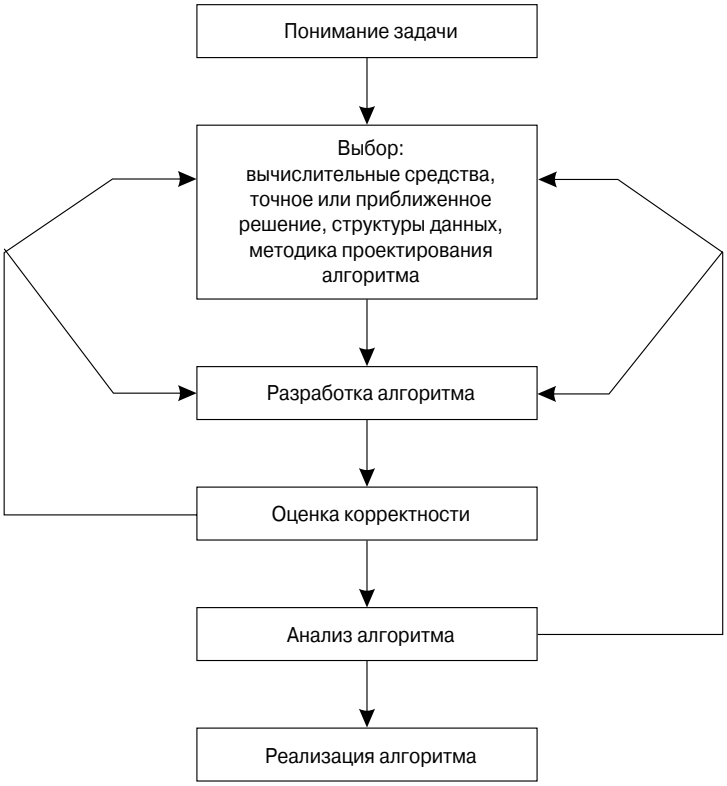


Рис. 1.2. Процесс проектирования и анализа алгоритмов

Понимание задачи

С практической точки зрения, прежде чем заняться проектированием алгоритма, необходимо полностью понять поставленную перед вами задачу. Прочтите внимательно условие задачи и задайте вопросы, если вы чего-то не поняли. Просчитайте вручную несколько небольших примеров, продумайте частные случаи и при необходимости снова задайте вопросы.

Существует несколько типов задач, которые при создании компьютерных приложений встречаются чаще всего. Их обзор будет сделан в следующем разделе. Если перед вами поставлена одна из этих задач, для ее решения можно воспользоваться известным алгоритмом. Конечно, в процессе решения необходимо понять, как работает алгоритм, и оценить все его достоинства и недостатки, особенно если существует возможность выбора среди нескольких алгоритмов. Однако чаще всего вы не сможете найти подходящий готовый алгоритм и должны будете создать собственный. Описанная в этом разделе последовательность этапов поможет вам в этом захватывающем, но не всегда легком деле.

Входные данные, обрабатываемые алгоритмом, определяют ***экземпляр задачи*** (problem's instance), решаемой с помощью выбранного алгоритма. При этом очень важно точно указать границы примеров, которые должны учитываться в алгоритме. (Вспомните, насколько отличаются границы примеров для трех алгоритмов поиска НОД, рассмотренных в предыдущем разделе.) Если этого не сделать, то алгоритм будет прекрасно работать для большинства значений входных параметров, однако при подстановке отдельных "граничных" значений вы получите некорректные результаты. Позвольте напомнить, что корректным считается такой алгоритм, который выдает абсолютно правильный результат для *всех* (а не только для большинства!) заранее оговоренных значений входных данных.

Не стоит недооценивать первый этап процесса решения алгоритмической задачи, поскольку в противном случае, как правило, приходится многое переделывать.

Определение возможностей вычислительного устройства

Полностью уяснив суть поставленной задачи, необходимо оценить возможности вычислительного устройства, для которого создается алгоритм. Подавляющее большинство современных алгоритмов предназначено для создания на их основе программы, работающей на компьютере, сильно напоминающем по структуре машину фон Неймана[3]. Суть этой архитектуры выражена в ее названии — ***машина с произвольным доступом*** (random-access machine, или ***RAM***). Предполагалось,

[3]Речь идет о структуре вычислительного устройства, описанного выдающимся американским математиком венгерского происхождения Джоном фон Нейманом (John von Neumann) (1903–1957) в сотрудничестве с А. Барксом (A. Burks) и Г. Голдстином (H. Goldstine) в 1946 году.

что команды должны последовательно выбираться из памяти и выполняться специальным устройством, называемым центральным процессором, причем в каждый момент времени в машине Неймана могла выполняться только одна команда. Поэтому алгоритмы, разработанные для выполнения на такой машине, назвали ***последовательными*** (sequential algorithms).

Появление машины Неймана не могло остановить прогресс в области вычислительной техники. Вскоре были придуманы компьютеры, которые могли одновременно (т.е. параллельно) выполнять несколько команд. Поэтому алгоритмы, разработанные для таких машин, назвали ***параллельными*** (parallel algorithms). Тем не менее изучение классических методов проектирования и анализа алгоритмов для машины Неймана еще долго будет оставаться краеугольным камнем алгоритмики.

Задумывались ли вы когда-нибудь о том, насколько быстро ваш компьютер выполняет команды и какой у него объем оперативной памяти? Наверняка вы ответите "нет", если до этого проектировали алгоритмы лишь с теоретической точки зрения. Как будет показано в разделе 2.1, большинство кибернетиков предпочитают изучать алгоритмы, не привязываясь к параметрам конкретной компьютерной системы. Ответ специалиста-практика наверняка будет зависеть от того, какую задачу перед ним поставили. Даже "медленные" по современным меркам компьютеры на деле оказываются невероятно быстродействующими. Следовательно, проблема "медленного компьютера" не должна возникать для большинства решаемых вами задач. Однако существует целый класс важных задач, которые по своей природе являются очень сложными, должны обрабатывать огромные массивы данных или работать в жестких временны́х рамках. В подобных ситуациях непременно следует учитывать быстродействие компьютерной системы, на которой будет работать реализация алгоритма, и доступный объем оперативной памяти.

Выбор между точным или приближенным методом решения задачи

Следующий принципиальный вопрос — выбор точного или приближенного метода решения задачи. В первом случае алгоритм называется ***точным*** (exact algorithm), а во втором — ***приближенным*** (approximation algorithm). Почему для решения задачи иногда выбираются приближенные алгоритмы? Во-первых, существуют задачи, которые нельзя решить точно. В качестве примера можно привести извлечение квадратного корня, решение нелинейных уравнений или вычисление определенных интегралов. Во-вторых, существующие алгоритмы для точного решения задачи могут быть недопустимо медленными, если ее сложность достаточно высока. Наиболее известной из таких задач является ***задача коммивояжера*** (traveling salesman problem), которая заключается в поиске кратчайшего маршрута

между n городами. В главах 3, 10 и 11 будут приведены и другие примеры подобных задач. В-третьих, приближенный алгоритм может являться частью другого, более сложного алгоритма, с помощью которого задача решается точно.

Выбор подходящих структур данных

В некоторых алгоритмах не требуется, чтобы входные данные были представлены в каком-то специфическом формате. Однако так бывает не всегда, более того, для работы многих алгоритмов требуются совершенно определенные структуры данных. Кроме того, некоторые из методов проектирования алгоритмов, о которых пойдет речь в главах 6 и 7, очень тесно связаны со структуризацией и реструктуризацией данных, определяющих экземпляры задачи. Много лет назад в уважаемой всеми книге провозглашалась чрезвычайная важность алгоритмов и структур данных и их влияние на процесс программирования. Причем об этой важности говорило само за себя название этой книги: *Algorithms + Data Structures = Programs* [123]. В современном мире, где властвует объектно-ориентированное программирование, структуры данных остаются чрезвычайно важным элементом процесса проектирования и анализа алгоритмов. Обзор основных структур данных будет сделан в разделе 1.4.

Методы проектирования алгоритмов

Теперь, когда все составляющие решения алгоритмической проблемы находятся на своих местах, остается выяснить, как нужно проектировать алгоритмы решения поставленных перед вами задач. Это основной вопрос данной книги. Для ответа на него вам предлагается изучить несколько общих методов проектирования алгоритмов. Что же такое метод проектирования алгоритма?

> **Метод проектирования алгоритма** (algorithm design technique) (или "стратегия", или "принцип") — это универсальный подход, применяемый для алгоритмического решения широкого круга задач, относящихся к различным областям вычислительной техники.

Взглянув на оглавление этой книги, вы поймете, что основная часть ее глав посвящена именно отдельным методам проектирования алгоритмов. В них излагается несколько проверенных временем ключевых идей, используемых при разработке алгоритмов. Изучение этих методов в высшей степени важно по следующим причинам.

Во-первых, они обеспечивают набор универсальных принципов, руководствуясь которыми можно разработать алгоритмы решения новых задач, т.е. таких задач, для решения которых еще не существует достаточно хороших алгоритмов. Поэтому, перефразируя известную поговорку, можно сказать, что изучение подобных методов сродни подаренной удочке, а не предложенной рыбе. Конечно, не

все из этих универсальных методов подойдут для решения практических задач, с которыми вам предстоит столкнуться. Однако они представляют собой мощный набор средств, которые пригодятся вам в учебе и работе.

Во-вторых, алгоритмы являются краеугольным камнем информатики. Классификация основополагающих понятий важна для любой науки, и информатика не является исключением. Изучение этих методов позволяет классифицировать алгоритмы согласно лежащему в их основе принципу проектирования. Поэтому они как нельзя лучше подходят и для классификации, и для изучения алгоритмов.

Методы представления алгоритмов

После того как алгоритм спроектирован, нужно представить его в каком-либо виде. В разделе 1.1 в качестве примера мы описали алгоритм Евклида — как словами (т.е. в свободной форме в виде последовательности выполняемых действий), так и в виде псевдокода. В наши дни для представления алгоритмов чаще всего используются эти две формы.

Безусловно, использование естественного языка для описания алгоритма имеет очевидное преимущество. Тем не менее присущая любому такому языку неопределенность иногда затрудняет лаконичное и понятное описание алгоритмов. Как бы то ни было, умение словесно описать алгоритм в процессе его изучения никогда не будет лишним.

Псевдокод (pseudocode) представляет собой смесь одного из естественных языков[4] и конструкций, характерных для языка программирования. Описание алгоритма на псевдокоде обычно является более точным по сравнению с естественным языком. Кроме того, в результате его использования часто получается более компактная запись алгоритма. Неожиданностью является то, что специалисты до сих пор так и не приняли какую-либо форму псевдокода в качестве стандарта. Поэтому в разных книгах можно встретить различные “диалекты” этого языка. К счастью, эти диалекты настолько близки, что любой человек, владеющий каким-либо современным языком программирования, сможет без особого труда в них разобраться.

Используемый в этой книге диалект псевдокода не должен представлять особых затруднений для читателя. Ради простоты изложения мы опустили операторы определения переменных, а для обозначения области действия таких операторов, как **for**, **if** и **while**, в тексте псевдокода использованы отступы. Как вы уже, наверное, заметили при чтении предыдущего раздела, для обозначения операции присваивания мы используем стрелку, ←, а комментариев — две косые черты подряд, `//`.

[4] Чаще всего английского, хотя встречаются случаи использования в псевдокоде русского языка. — *Прим. перев.*

На заре развития вычислительной техники основным способом представления алгоритмов были ***блок-схемы*** (flowchart), т.е. чертежи, состоящие из последовательности соединенных стрелками геометрических фигур, с помощью которых описывался каждый шаг выполнения алгоритма (примером может служить рис. 1.2). Однако подобный способ представления сочли неудобным, особенно в случае больших и сложных алгоритмов, поэтому в современной литературе он не используется.

Современные средства вычислительной техники еще не достигли такого уровня, чтобы описание алгоритма (будь-то его словесное описание или псевдокод) можно было непосредственно ввести в компьютер. Поэтому пока приходится вручную преобразовывать алгоритмы в компьютерные программы и записывать их на одном из доступных языков программирования. Таким образом, подобную программу можно считать еще одним средством представления алгоритма, хотя, строго говоря, программа является реализацией конкретного алгоритма.

Оценка корректности алгоритма

После представления алгоритма в какой-либо форме, необходимо оценить его ***корректность*** (correctness). Это означает, что вы должны доказать, что выбранный алгоритм за ограниченный промежуток времени выдает требуемый результат для любых корректных значений входных данных. Например, корректность алгоритма Евклида для вычисления НОД двух чисел, m и n, следует из правильности равенства $\gcd(m, n) = \gcd(n, m \bmod n)$, которое, в свою очередь, должно быть доказано (см. упражнение 1.1.5), а также из простого наблюдения, что второе число на каждой итерации будет все время уменьшаться, и по достижении им нуля, выполнение алгоритма прекращается.

Доказать корректность одних алгоритмов очень легко, других — невероятно сложно. Универсальным методом доказательства корректности алгоритма считается метод математической индукции. Дело в том, что алгоритмы по своей природе являются итеративными и описываются в виде последовательности пошаговых инструкций, которые как раз и используются при доказательстве методом индукции. Стоит отметить, что, хотя отслеживание быстродействия алгоритма с помощью нескольких наборов тщательно подобранных входных данных приводит к очень полезным результатам, подобную проверку нельзя считать убедительной при доказательстве корректности алгоритма. Однако доказательством некорректной работы алгоритма может служить лишь один набор входных данных, при обработке которых получается неправильный результат. Если некорректная работа алгоритма будет доказана, придется либо несколько видоизменить его, не выходя за рамки принятых структур данных, методов проектирования и т.п., либо решать проблему более кардинально, пересмотрев принятые ранее принципы и подходы (см. рис. 1.2).

Оценка корректности приближенных алгоритмов менее тривиальна, чем их точных аналогов. При этом нужно показать, что погрешность получаемых в результате работы алгоритма выходных данных не превышает заранее установленных пределов. Примеры подобных оценок приведены в главе 11.

Анализ алгоритма

Обычно создатели алгоритмов стараются, чтобы они удовлетворяли нескольким требованиям. После проверки корректности алгоритма одной из самых важных характеристик является оценка его эффективности. На практике существует два вида оценки эффективности алгоритма: временна́я и пространственная. ***Временная эффективность*** (time efficiency) является индикатором скорости работы алгоритма. Что касается ***пространственной эффективности*** (space efficiency), то эта оценка показывает количество дополнительной оперативной памяти, необходимой для работы алгоритма. Общая идея и конкретные методы анализа эффективности алгоритмов будут рассмотрены в главе 2.

Еще одной важной характеристикой алгоритма является его ***простота*** (simplicity). В отличие от эффективности, которую можно точно определить и оценить с помощью строгих математических выражений, простота алгоритма чем-то напоминает человеческую красоту, понятие которой настолько субъективно, что выработать объективные критерии ее оценки весьма непросто. Например, большинство людей считают, что алгоритм Евклида поиска НОД двух чисел проще, чем тот, которому нас учили в школе, но это вовсе не означает, что он понятнее. В то же время алгоритм Евклида проще алгоритма последовательного перебора целых чисел. Тем не менее простота является важной характеристикой алгоритма, и к ней нужно стремиться. Вы спросите, почему? Дело в том, что простые алгоритмы легче понять и запрограммировать. Следовательно, в полученной программе будет содержаться меньше ошибок. Кроме того, в простоте существует неоспоримая эстетическая привлекательность. Ну и наконец, простые алгоритмы зачастую более эффективны, чем их сложные аналоги. К сожалению, последнее утверждение не всегда выполняется. В подобных случаях необходимо руководствоваться здравым смыслом и принимать компромиссные решения.

Следующей важной характеристикой алгоритма является его ***общность***, или ***универсальность*** (generality). По сути, здесь можно выделить два момента: общность задачи, для решения которой разработан алгоритм, и диапазон допустимых значений его входных данных. По поводу первого замечания следует сказать, что иногда легче разработать алгоритм для решения общей задачи, чем заниматься поиском решения ее частного случая. В качестве примера рассмотрим такую задачу: определить, являются ли два целых числа взаимно простыми. Напомним, что два числа считаются взаимно простыми, если у них нет общих делителей, кроме 1. В данном случае легче разработать алгоритм для решения общей зада-

чи вычисления НОД двух целых чисел. Затем, чтобы решить исходную задачу, достаточно подставить заданные числа в функцию $\gcd$ и проверить, равно ли ее значение 1. Однако бывают случаи, когда разработка более общего алгоритма нежелательна, затруднена или даже невозможна. Например, излишне выполнять сортировку всего списка, содержащего n чисел, чтобы определить их медиану, поскольку достаточно просто найти значение m-го наименьшего элемента, где $m = \lceil n/2 \rceil$. Еще один пример: известно, что стандартную формулу для поиска корней квадратного уравнения нельзя обобщить для поиска корней полиномов более высоких степеней.

Что касается диапазона допустимых значений входных данных алгоритма, то здесь нужно отметить следующее. При разработке алгоритма нужно учитывать, что диапазон изменения значений входных параметров может колебаться в очень широких пределах и должен естественным образом соответствовать решаемой задаче. Например, неестественно в алгоритме поиска НОД исключать из рассмотрения значения входных параметров, равные 1. С другой стороны, хотя стандартную формулу корней квадратного уравнения можно использовать и в случае комплексных коэффициентов, при принятом уровне обобщения этот случай исключают из рассмотрения, поскольку он должен оговариваться отдельно.

Если вас не устраивает эффективность, простота или общность алгоритма, придется снова взять в руки карандаш и перепроектировать алгоритм. И даже если анализ алгоритма привел к положительным результатам, имеет смысл поискать другое алгоритмическое решение задачи. Достаточно вспомнить три алгоритма определения НОД, рассмотренных в предыдущем разделе. Вообще говоря, не стоит надеяться, что вы с первой попытки найдете оптимальное решение задачи. Самое меньшее, что можно сделать, попытаться оптимизировать созданный алгоритм. Например, мы внесли несколько улучшений в реализацию алгоритма решета Эратосфена, по сравнению с той, что была описана в разделе 1.1. (Можете ли вы назвать их не задумываясь?) Всегда помните высказывание Антуана де Сент–Экзюпери (Antoine de Saint–Exupery), известного французского писателя, летчика и авиаконструктора: “Конструктор достигает совершенства не тогда, когда ему больше нечего добавить к своему детищу, а тогда, когда больше ничего удалить”.[5]

Кодирование алгоритма

Большинство алгоритмов рассчитано на то, что в конечном итоге они будут превращены в компьютерную программу. В процессе написания программы на

[5] Я обнаружил это высказывание по поводу простоты конструкции в сборнике рассказов Джона Бентли (Jon Bentley) [15]. Этот сборник посвящен особенностям проектирования и реализации алгоритмов и имеет совершенно логичное название *Programming Pearls* [15]. Настоятельно рекомендую почитать рассказы Бентли и Сент–Экзюпери. Не пожалеете!

основе алгоритма может возникнуть множество подводных камней. Главная опасность заключается в том, что при переводе алгоритма на машинный язык могут быть внесены ошибки, либо окажется, что сама программа написана крайне неэффективно. Поэтому некоторые авторитетные специалисты считают, что до тех пор, пока корректность компьютерной программы не будет доказана с помощью строгих математических выкладок, ее нельзя считать правильной. Они даже разработали специальные методы для выполнения подобных оценок (см. [47]). Однако пока эти методы формальной верификации удается применить только к очень маленьким программам. На практике корректность компьютерных программ все еще проверяется с помощью тестирования. Процесс тестирования является скорее искусством, чем наукой, но это отнюдь не означает, что здесь нечему поучиться. Тестированию и отладке посвящено довольно много хороших книг, однако их основную мысль можно сформулировать так: в процессе реализации алгоритма программа должна быть тщательно протестирована и отлажена.

Следует также заметить, что в этой книге мы всегда предполагаем, что диапазоны значений входных данных алгоритмов не выходят за заранее оговоренные рамки, поэтому их не нужно проверять. Однако при реализации алгоритмов в виде реально работающих прикладных программ такая проверка просто необходима.

Само собой разумеется, что корректная реализация алгоритма в виде программы является необходимым, но недостаточным условием, поскольку мощь алгоритма можно свести на нет неэффективной реализацией. Современные компиляторы до некоторой степени позволяют застраховаться от этого, особенно при использовании режима оптимизации кода. Тем не менее следует знать о таких стандартных вещах, как вынесение операторов вычисления инвариантного выражения (т.е. такого выражения, которое не изменяется в цикле) за пределы цикла, выделение общих подвыражений, замена медленных операторов их более быстрыми аналогами и т.п. (Хорошее описание методов оптимизации кода программы и других вопросов, связанных с кодированием алгоритмов, можно найти в [15] и [61].) Как правило, применение подобных методов оптимизации может ускорить выполнение программы только на небольшой постоянный множитель, тогда как использование более эффективного алгоритма иногда ускоряет выполнение программы на несколько порядков. Когда алгоритм окончательно выбран, повышение производительности реализующей его программы на 10–50% считается хорошим результатом.

После создания работающей версии программы можно выполнить дополнительный эмпирический анализ лежащего в ее основе алгоритма. Для этого нужно зафиксировать время выполнения программы при разных значениях входных данных, а затем проанализировать полученный результат. Достоинства и недостатки данного метода анализа алгоритмов будут описаны в разделе 2.6.

И в заключение позвольте еще раз подчеркнуть основную идею процесса проектирования и анализа алгоритмов, показанного на рис. 1.2.

> Хороший алгоритм получается, как правило, в результате кропотливой циклической работы, связанной с возможными переделками.

Даже если вам крупно повезет и вы придумаете алгоритм, который на первый взгляд кажется идеальным, все равно попытайтесь проанализировать, нельзя ли его еще в чем-то улучшить. Как ни странно, эта мысль не такая уж и плохая, поскольку позволяет получить истинное удовольствие от конечного результата. (Да, да! Кстати я собирался назвать эту книгу как *The Joy of Algorithms*.[6]) С другой стороны, нужно уметь вовремя останавливаться. Что касается реальной жизни, то здесь обычно критерий остановки один — срок сдачи проекта или долготерпение вашего начальника, в зависимости от того, что быстрее закончится. В общем вывод такой: достижение идеала — слишком дорогое удовольствие, которое, к тому же, не всегда нужно. Проектирование алгоритма является достаточно сложной инженерной задачей, связанной с принятием компромиссных решений в условиях ограниченного доступа к ресурсам, одним из которых является время работы проектировщика.

С академической точки зрения затронутый в этом разделе вопрос связан с проведением интересного, но, как правило, очень сложного исследования ***оптимальности*** (optimality) алгоритма. Как ни странно, этот вопрос не относится к эффективности самого алгоритма, а связан со сложностью решаемой с его помощью задачи. То есть необходимо выяснить, какое минимальное количество усилий нужно затратить, чтобы решить стоящую перед вами задачу с помощью *произвольного* алгоритма? Для некоторых задач ответ на этот вопрос найден. Например, в любом алгоритме сортировки элементов массива методом сравнения необходимо выполнить порядка $n \log_2 n$ операций сравнения, где n — размерность массива (см. раздел 10.2). Однако для многих кажущихся простыми задач, типа перемножения матриц, ученым до сих пор так и не удалось найти окончательного ответа.

Еще один важный вопрос, возникающий при решении алгоритмической проблемы, заключается в том, можно ли решить задачу вообще с помощью какого-либо алгоритма? В этой книге мы не будем обсуждать задачи, не имеющие решения, наподобие поиска вещественных корней квадратного уравнения с отрицательным дискриминантом. В подобных случаях алгоритм должен возвращать специальное значение, являющееся признаком того, что задача не имеет решения. Мы не будем также рассматривать неоднозначно определенные задачи. Речь идет о таких задачах, решение которых нельзя найти с помощью любого алгоритма, хотя у них может быть простой ответ — да или нет. Пример такой задачи будет приведен в разделе 10.3. К счастью, подавляющее большинство подобных задач *может* быть решено с помощью некоторого алгоритма.

И в заключение этого раздела хотелось бы развеять ваши сомнения по поводу того, что проектирование алгоритмов — довольно скучное занятие. Отчасти они

[6] Это название можно перевести на русский язык как *Удовольствие от алгоритмов. — Прим. ред.*

могли появиться при взгляде на рис. 1.2, где все четко разложено по полочкам и на первый взгляд нет никакой свободы для творчества. Это абсолютно не соответствует действительности! Изобретение (или поиск?) алгоритмов — творческий и необычайно захватывающий процесс, и я надеюсь, что книга убедит вас в этом.

Упражнения 1.2

1. *Древняя народная головоломка*. На берегу реки находятся крестьянин, волк, коза и кочан капусты. Крестьянин должен в своей лодке перевезти их на другой берег. Однако в лодке есть только два места — для крестьянина и еще одного объекта (т.е. либо волка, либо козы, либо капусты). В отсутствие крестьянина волк может съесть козу, а коза — капусту. Помогите крестьянину решить эту задачу или докажите, что она не имеет решения. (*Примечание*. Для определенности будем считать, что крестьянин является вегетарианцем, но терпеть не может капусту, поэтому в лодке он не может съесть ни козу, ни капусту. Кроме того, крестьянин должен учитывать, что он столкнулся с редкой породой волка, которая занесена в Красную книгу.)

2. *Современная народная головоломка*. Предположим, что четыре человека движутся по дороге в одном направлении и хотят перейти через мост. Ваша задача помочь им переправиться на другой берег за 17 минут. На дворе ночь, и у них только один фонарик. По мосту одновременно могут следовать не более двух человек (т.е. либо один, либо два), причем у одного из них обязательно должен быть фонарик. Фонарик нельзя перебросить с одного берега реки на другой, его можно только перенести по мосту обратно. Каждый человек затрачивает разное время на прохождение моста: первый — 1 минуту, второй — 2 минуты, третий — 5 минут и четвертый — 10 минут. Если по мосту передвигается пара людей, то они идут со скоростью более медлительного из них. Например, если по мосту передвигается первый и четвертый человек, то они достигнут противоположного берега через 10 минут. Если четвертый человек будет возвращать фонарь на другой берег, то с момента начала задачи пройдет 20 минут, и вы не решите задачу. (*Примечание*. В Internet бродят слухи, что одна из известных компаний по производству программного обеспечения, расположенная вблизи Сиэтла, предлагает решить эту задачу претендентам на собеседовании.)

3. Какую из перечисленных ниже формул можно использовать в качестве алгоритма для вычисления площади треугольника, длина сторон которого выражена положительными числами a, b и c?

а) $S = \sqrt{p(p-a)(p-b)(p-c)}$, где $p = (a+b+c)/2$;

б) $S = \frac{1}{2}bc\sin A$, где A — угол между сторонами b и c;

в) $S = \frac{1}{2}ah_a$, где h_a — высота треугольника, опущенная на сторону a.

4. Запишите на псевдокоде алгоритм поиска вещественных корней квадратного уравнения $ax^2 + bx + c = 0$, где a, b и c — произвольные вещественные коэффициенты. (Подразумевается, что для поиска квадратного корня вы можете воспользоваться функцией $sqrt\,(x)$.)

5. Опишите стандартный алгоритм преобразования положительного десятичного числа в двоичное:

а) словами;

б) на псевдокоде.

6. Опишите алгоритм работы с банкоматом при получении денег с карточки (если, конечно, она у вас есть). (Опишите алгоритм либо словами, либо на псевдокоде — как вам больше нравится.)

7. а) Может ли быть точно решена задача вычисления числа π?

б) Сколько экземпляров имеет данная задача?

в) Поищите алгоритм решения этой задачи в World Wide Web.

8. Приведите пример задачи, отличной от поиска НОД, для решения которой существует несколько алгоритмов. Какой из этих алгоритмов проще? Какой эффективнее?

9. Проанализируйте приведенный ниже алгоритм поиска минимальной разницы между двумя элементами массива чисел.

Алгоритм $MinDistance\,(A\,[0..n-1])$

```
// Входные данные:  массив чисел A[0..n − 1]
// Выходные данные: минимальная разница между двумя
//                  элементами массива A
dmin ← ∞
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        if i ≠ j and |A[i] − A[j]| < dmin
            dmin ← |A[i] − A[j]|
return dmin
```

Можете ли вы усовершенствовать алгоритм решения этой задачи, и если да, то какое количество изменений вы можете в него внести? (При

необходимости можно выбрать другой алгоритм. Если вы не можете улучшить предложенный алгоритм, то можете ли вы усовершенствовать его реализацию?)

10. Одна из самых известных книг, посвященная алгоритмическому решению задач, озаглавленная *How to Solve It*, написана американским математиком венгерского происхождения Джорждем Пойа (George Polya) (1887–1985) [89]. Пойа обобщил выдвинутые им идеи в виде резюме, состоящего из четырех пунктов. Найдите это резюме в Web (или, что еще лучше, прочтите саму книгу) и сравните его с предложенным планом решения алгоритмической задачи, который мы рассматривали в разделе 1.2. Что у них общего? В чем различия?

1.3 Важные типы задач

Среди огромного количества задач, встречающихся в вычислительной технике, можно выделить несколько типов, которым ученые всегда уделяли особое внимание. Подобный интерес вызван либо практическим значением задачи, либо какими-то ценными ее свойствами, представляющими особый интерес для исследования. К счастью, в большинстве случаев эти причины взаимосвязаны, что только усиливает интерес ученых.

В этом разделе кратко рассматриваются наиболее важные типы задач:

- сортировка;
- поиск;
- обработка строк;
- задачи из теории графов;
- комбинаторные задачи;
- геометрические задачи;
- численные задачи.

Эти задачи будут использоваться в последующих главах книги для иллюстрации различных методов проектирования и анализа алгоритмов.

Сортировка

Задача сортировки (sorting problem) заключается в упорядочении заданного списка каких-либо элементов в возрастающем порядке[7]. Само собой разумеется, что для определенности задачи структура этих элементов списка должна позволять их упорядочить. (В подобных случаях математики говорят, что между

[7] Очевидно, что для сортировки в порядке убывания достаточно заменить вид сравнения элементов последовательности на обратный. — *Прим. ред.*

элементами должны существовать отношения, допускающие полное упорядочение.) На практике обычно требуется отсортировать по возрастанию список чисел, расположить символы и строки в алфавитном порядке или, что наиболее важно, упорядочить набор записей, содержащих различную информацию, наподобие той, что хранится в папках со сведениями об учащихся школы, в читательских формулярах библиотеки, в отделе кадров о сотрудниках организации. В случае набора записей необходимо выбрать фрагмент записи, содержащий данные, по которым будет осуществляться сортировка. Например, набор записей о студентах можно отсортировать по фамилии, идентификационному номеру или по среднему баллу. Специально отобранный фрагмент данных называется ***ключом*** (key). Специалисты по информатике часто говорят о сортировке списка ключей, даже если элементы этого списка являются не записями, а, скажем, целыми числами.

Зачем может понадобиться отсортированный список? Ну, скажем затем, чтобы облегчить поиск ответов на ряд вопросов, связанных со списком. Наибольшую важность при этом имеет быстрота поиска информации. Вот почему словари, телефонные справочники, списки учащихся и т.п. всегда упорядочиваются по алфавиту. В разделе 6.1 будут приведены другие примеры, в которых используется предварительно отсортированный список. Аналогично сортировка используется также как вспомогательный этап в некоторых важных алгоритмах, относящихся к другим предметным областям, например к геометрии.

К настоящему моменту специалистами по вычислительной технике разработаны десятки алгоритмов сортировки. По сути, выдумывание нового алгоритма сортировки можно сравнить с изобретением пресловутого велосипеда. Тем не менее мы с гордостью хотим заявить, что поиск лучших вариантов велосипеда (т.е. алгоритмов сортировки) продолжается. Подобная настойчивость может быть оправданна, если принять во внимание следующие факты. С одной стороны, существует несколько хороших алгоритмов сортировки, в которых для сортировки произвольного массива из n элементов используется $n \log_2 n$ операций сравнения. С другой стороны, нет алгоритма, который бы выполнял сортировку методом сравнения всего значения ключа (а не, скажем, небольшой части ключа) за существенно меньшее количество операций, чем было указано выше.

Существует причина, сдерживающая разработку новых алгоритмов сортировки. Несмотря на то что часть алгоритмов оказывается лучше остальных, пока еще не придуман универсальный алгоритм сортировки, который бы наилучшим образом подходил для всех случаев. Одни из существующих алгоритмов являются простыми, но сравнительно медленными, другие работают быстрее, но более сложны. Некоторые из алгоритмов хорошо работают с неупорядоченными данными, тогда как для других нужно, чтобы списки были частично отсортированы. Часть алгоритмов пригодна только для сортировки списков данных, расположенных в оперативной памяти (т.е. в памяти с быстрым доступом), тогда как другие

можно применить для сортировки больших массивов данных, расположенных на внешних носителях данных (магнитном диске, ленте и т.д.).

Два свойства алгоритмов сортировки заслуживают особого внимания. Алгоритм сортировки называется ***устойчивым*** (stable), если в нем сохраняется относительный порядок любых двух равных элементов, находящихся во входном списке. Другими словами, если во входном списке есть два одинаковых элемента, номера которых равны i и j, причем $i < j$, то в отсортированном списке, где их номера будут, соответственно, равны i' и j', будет выполняться отношение $i' < j'$. Это свойство может пригодиться в случае, если, например, требуется отсортировать упорядоченный в алфавитном порядке список учащихся согласно их успеваемости (среднему баллу). В случае применения устойчивого алгоритма будет получен список, в котором учащиеся с одинаковым средним баллом будут упорядочены по алфавиту. Вообще говоря, алгоритмы сортировки методом обмена значениями ключей, расположенными на значительном расстоянии друг от друга, не являются устойчивыми, однако обычно они работают быстрее. Ниже в этой книге мы покажем, как это общее утверждение применяется к основным алгоритмам сортировки.

Второе важное свойство алгоритмов сортировки связано с количеством дополнительной оперативной памяти, необходимой для работы алгоритма. Алгоритм называют ***обменным*** (in place), если для его работы не требуется дополнительная оперативная память, кроме случаев возможного использования нескольких дополнительных ячеек памяти. Многие важные алгоритмы сортировки относятся к классу обменных, а другие, не менее важные — нет.

Поиск

Задача ***поиска*** связана с нахождением заданного значения, называемого ***ключом поиска*** (search key), среди заданного множества (или мультимножества[8]). Существует огромное количество алгоритмов поиска, так что есть из чего выбирать. Их сложность варьируется от самых простых алгоритмов поиска методом последовательного сравнения, до чрезвычайно эффективных, но ограниченных алгоритмов бинарного поиска, а также алгоритмов, основанных на представлении базового множества в иной, более подходящей для выполнения поиска форме. Последние из упомянутых здесь алгоритмов имеют особое практическое значение, поскольку применяются в реально действующих приложениях, выполняющих выборку и хранение массивов информации в огромных базах данных.

Для решения задачи поиска также не существует единого алгоритма, который бы наилучшим образом подходил для всех случаев. Некоторые из алгоритмов выполняются быстрее остальных, но для их работы требуется дополнительная

[8] ***Мультимножество*** — это множество, в котором несколько принадлежащих ему элементов могут иметь одинаковые значения.

оперативная память. Другие выполняются очень быстро, но их можно применять только для предварительно отсортированных массивов, и т.п. В отличие от алгоритмов сортировки в алгоритмах поиска нет проблемы устойчивости, но при их использовании могут возникать другие сложности. В частности, в тех приложениях, где обрабатываемые данные могут часто изменяться, причем количество изменений сравнимо с количеством операций поиска, поиск следует рассматривать в неразрывной связи с двумя другими операциями — добавления элемента в набор данных и удаления из него. В подобных ситуациях необходимо видоизменить структуры данных и алгоритмы так, чтобы достигалось равновесие между требованиями, выдвигаемыми к каждой операции. Кроме того, организация очень больших наборов данных с целью выполнения в них эффективного поиска (а также добавления и удаления элементов) представляет собой чрезвычайно сложную задачу, решение которой особенно важно с точки зрения практического применения.

Обработка строк

В связи с быстрым увеличением в последнее время количества приложений, связанных с обработкой нечисловых данных, интерес ученых и специалистов-практиков все больше привлекают алгоритмы обработки строк. ***Строкой*** (string) называется последовательность символов, взятых из заранее определенного алфавита. Практический интерес представляют, например, текстовые строки, состоящие из букв, цифр и специальных символов; битовые строки, состоящие из нулей и единиц; последовательности генов, которые могут быть смоделированы с помощью строк символов, взятых из четырехсимвольного алфавита {A, C, G, T}. Тем не менее стоит отметить, что алгоритмы обработки строк стали важны для вычислительной техники очень давно — с тех пор, как появились первые языки программирования и соответствующие им программы — компиляторы.

Существует одна специфическая задача, которая привлекла особое внимание специалистов по информатике. Речь идет о поиске заданного слова в строке текста. Ее назвали ***поиском подстрок*** (string matching). Для выполнения такого специфического поиска разработано несколько алгоритмов. В главе 3 мы опишем один очень простой алгоритм, а в главе 7 обсудим два алгоритма, созданных на основе замечательной идеи Р. Бойера (R. Boyer) и Дж. Мура (J. Moore).

Задачи из теории графов

Одной из самых старых и, пожалуй, наиболее интересных областей алгоритмики является обработка графов. Нестрого ***граф*** можно определить как набор точек, называемых вершинами, часть из которых соединена отрезками, называемыми ребрами. (Более строгое определение графа будет дано в следующем разделе.)

Графы являются довольно интересным объектом для изучения как с теоретической, так и с практической точек зрения. С помощью графов можно смоделировать довольно большое количество процессов, происходящих в реальной жизни, например функционирование транспортных и коммуникационных сетей, календарное планирование проекта и т.д. Одна из последних интересных задач — оценка "диаметра" Web, заключающаяся в определении максимального количества ссылок, которые нужно пройти от одной Web-страницы до другой по оптимальному маршруту[9].

К числу основных алгоритмов теории графов относят следующие:

- алгоритмы обхода графа (этот класс алгоритмов позволяет ответить на такие вопросы, как каким образом можно объехать все узлы железнодорожной сети);
- алгоритмы определения кратчайшего пути (этот класс алгоритмов позволяет ответить на вопросы наподобие следующего: каков кратчайший путь между двумя городами);
- алгоритмы топологической сортировки для ориентированных графов ребрами (этот класс алгоритмов позволяет выяснить, не является ли множество читаемых студентам курсов внутренне противоречивым).

К счастью, все перечисленные алгоритмы можно рассматривать как пример общих методов проектирования. Поэтому вы сможете найти их описание в соответствующих главах книги.

Некоторые из задач теории графов с вычислительной точки зрения очень трудны. Это означает, что только небольшое количество экземпляров подобных задач можно решить за приемлемое время с помощью самого быстродействующего компьютера, который только можно себе представить. К наиболее известным задачам теории графов этого типа вероятнее всего относятся задачи коммивояжера и раскраски графа. Напомним, что задача коммивояжера заключается в нахождении кратчайшего пути между n городами, каждый из которых он должен посетить только один раз. Задача ***раскраски графа*** (graph-coloring problem) заключается в том, что нужно с помощью минимального количества красок раскрасить вершины графа так, чтобы не было двух смежных вершин одинакового цвета. Эта задача появляется в процессе решения важных практических задач, таких как, например, календарное планирование. Если представить работы в виде вершин графа и соединить их между собой ребрами тогда и только тогда, когда соответствующие им работы не могут выполняться в одно и то же время, решение задачи раскраски такого графа даст оптимальный график выполнения работ.

[9] Согласно оценке группы исследователей из университета Нотр–Дама (University of Notre Dame), это количество ссылок составляет всего 19 [6].

Комбинаторные задачи

С точки зрения обобщения задачи коммивояжера и раскраски графа являются примерами ***комбинаторных задач***. Суть этих задач в конечном итоге сводится к нахождению такого комбинаторного объекта, как перестановка, сочетание или подмножество, который бы удовлетворял определенным ограничениям и обладал заданными свойствами (например, позволял максимизировать прибыль или минимизировать затраты.)

Вообще говоря, комбинаторные задачи относятся к классу самых сложных как с теоретической, так и с практической точек зрения. Их сложность обусловлена следующим. Во-первых, количество комбинаторных объектов обычно очень быстро растет при увеличении масштаба задачи и достигает невообразимых значений уже при весьма скромных ее масштабах. Во-вторых, пока не существует алгоритмов для поиска точного решения подобных задач за приемлемое время. Более того, большинство специалистов в области информатики считают, что таких алгоритмов попросту не существует. Это предположение не было ни доказано, ни опровергнуто. Оно до сих пор остается наиболее важным из нерешенных вопросов теоретической информатики. Более подробно этот вопрос мы обсудим в разделе 10.3.

Некоторые комбинаторные задачи можно решить с помощью эффективных алгоритмов, но это скорее счастливое исключение, чем правило.

Геометрические задачи

Геометрические алгоритмы связаны с такими геометрическими объектами, как точки, линии, многоугольники. Древние греки проявляли большой интерес к разработке процедур (естественно, они их еще не называли алгоритмами) решения различных геометрических задач, среди которых можно назвать построение простых геометрических форм (треугольников, окружностей и т.п.) с помощью неградуированной линейки и циркуля. С тех пор прошло более 2000 лет, и в век компьютеров интерес к геометрическим алгоритмам вспыхнул с новой силой. Для решения подобных задач линейки и циркули уже не нужны — только биты, байты и накопленный годами человеческий опыт. Естественно, сейчас человечество проявляет совершенно другой интерес к геометрическим алгоритмам. Они нужны для решения современных задач компьютерной графики, робототехники и томографии.

В этой книге мы рассмотрим алгоритмы решения только двух классических задач вычислительной геометрии: поиска пары ближайших точек и определения выпуклой оболочки. Название задачи поиска ***пары ближайших точек*** говорит само за себя: среди n точек на плоскости необходимо выбрать пару точек, расположенных наиболее близко друг к другу. При решении задачи построения ***выпуклой оболочки*** необходимо найти наименьший выпуклый многоугольник, который бы

охватывал все точки некоторого множества на плоскости. Чтобы больше узнать об этих и других геометрических задачах, обратитесь к специализированным монографиям (например, [90]) или к соответствующим главам учебников, построенных по принципу описания конкретных типов задач (например, [102]).

Численные задачи

Численные задачи относятся к еще одной довольно обширной области практического использования алгоритмов. Они имеют дело с математическими объектами, которые по своей сути являются непрерывными. Вот примеры типичных численных задач: решение уравнений и систем уравнений, вычисление определенных интегралов и значений функций и т.д. Подавляющее большинство таких задач может быть решено только приблизительно. Еще одна принципиальная трудность заключается в том, что при решении подобных задач обычно нужно выполнять операции с вещественными числами, которые в компьютере могут быть представлены только с определенной погрешностью. Более того, выполнение большого количества арифметических операций над представленными приближенно числами может привести к накоплению ошибок округления, что в свою очередь может кардинально повлиять на точность получаемых результатов.

За прошедшие годы было придумано большое количество довольно сложных алгоритмов решения численных задач, причем они продолжают играть ключевую роль при выполнении научных и инженерных расчетов. Однако в течение последних 25 лет интересы компьютерной индустрии сместились в сторону создания прикладных программ для деловой сферы. Для создания приложений этой категории потребовалось разработать базовые алгоритмы хранения и выборки данных, передачи их по сетям и отображения в удобном для пользователя виде. В результате таких революционных изменений численный анализ утратил былые позиции как в области промышленного, так и научного использования программ. Тем не менее для любого человека, изучающего компьютерную литературу, важно иметь хотя бы элементарные понятия в области численных алгоритмов. Несколько таких классических алгоритмов мы опишем в разделах 6.2, 10.4 и 11.4.

Упражнения 1.3

1. Проанализируйте приведенный ниже алгоритм сортировки массива методом подсчета. Вначале для каждого элемента массива подсчитывается количество элементов, меньших, чем он, и на основе этой информации текущий элемент помещается в соответствующее место отсортированного массива.

Алгоритм $ComparisonCountingSort\,(A\,[0..n-1])$

```
// Сортировка массива методом подсчета сравнений
// Входные данные:  массив чисел A[0..n − 1], который нужно
//                  отсортировать
// Выходные данные: массив чисел S[0..n − 1], состоящий из
//                  элементов массива A, отсортированных
//                  в неубывающем порядке
for i ← 0 to n − 1 do
  Count[i] ← 0
for i ← 0 to n − 2 do
  for j ← i + 1 to n − 1 do
    if A[i] < A[j]
      Count[j] ← Count[j] + 1
    else
      Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do
  S[Count[i]] ← A[i]
return S
```

а) Попробуйте с помощью этого алгоритма отсортировать числа: 60, 35, 81, 98,14, 47.

б) Является ли этот алгоритм устойчивым?

в) Относится ли он к обменным алгоритмам?

2. Назовите известные вам алгоритмы поиска. Кратко опишите словами каждый алгоритм. (Если вам еще не знаком ни один подобный алгоритм, воспользуйтесь удобной возможностью и разработайте его самостоятельно.)

3. Придумайте простой алгоритм поиска строк.

4. *Мосты Кенигсберга*. Считается, что решение этой головоломки дало толчок развитию теории графов. Задача была решена выдающимся российским математиком швейцарского происхождения Леонардом Эйлером (Leonard Euler) (1707–1783). В головоломке предлагается ответить, можно ли одну прогулку обойти все семь мостов Кенигсберга и вернуться в отправную точку? На рис. 1.3 изображен схематический план реки, посередине которой расположены два острова, к которым ведут семь мостов.

а) Сформулируйте эту задачу в терминах теории графов.

б) Имеет ли задача решение? Если вы уверены в том, что имеет, начертите схему обхода мостов. Если нет, объясните, почему, и оцени-

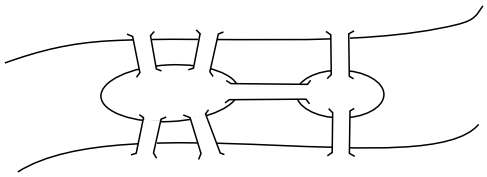


Рис. 1.3. Схематический план мостов Кенигсберга

те, какое минимальное количество новых мостов нужно построить, чтобы задача имела решение.

5. *Игра в икосаэдр*. Через сто лет после исследований Эйлера (см. задачу 1.3.4), известный ирландский математик сэр Вильям Гамильтон (Sir William Hamilton) (1805–1865) придумал еще одну головоломку — *Icosian Game*. На круглой деревянной доске вырезался граф, представленный на рис. 1.4.

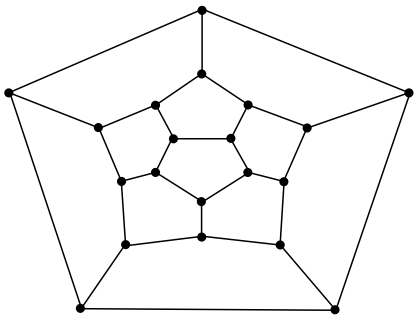


Рис. 1.4. Граф Гамильтона

Начертите ***гамильтонов цикл*** (Hamiltonian circuit) этого графа, т.е. путь однократного обхода всех вершин графа с возвратом в исходную точку.

6. Разработайте алгоритм поиска наилучшего маршрута для пассажира метрополитена, который бы позволял ему перемещаться от одной станции до другой по развитой системе подземных коммуникаций, наподобие вашингтонской или лондонской.

а) В условии этой задачи имеется небольшая неопределенность, что в общем-то типично для всех задач, находящих практическое применение. В частности, неясно, какой должен быть критерий оценки “наилучшего” маршрута? Можете ли вы его определить?

б) Можете ли вы смоделировать решение этой задачи с помощью графа?

7. а) Сформулируйте задачу коммивояжера в терминах комбинаторики.
б) Сформулируйте задачу раскраски графа в терминах комбинаторики.

8. Проанализируйте карту, изображенную на рис. 1.5.

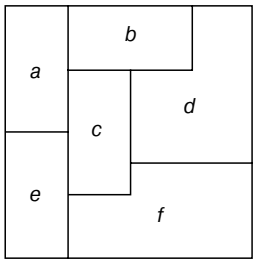


Рис. 1.5. Учебная контурная карта

а) Поясните, как можно использовать решение задачи раскраски графа для окрашивания этой карты так, чтобы цвета двух соседних областей не совпадали.
б) Воспользуйтесь решением пункта a упражнения и раскрасьте карту минимально возможным количеством цветов

9. Разработайте алгоритм решения следующей задачи. Предположим, на плоскости находится n точек, заданных их координатами (x, y). Определите, лежат ли все точки на одной и той же окружности?

10. Напишите программу, которая бы считывала из входного потока данных координаты (x, y) конечных точек двух отрезков P_1Q_1 и P_2Q_2 и определяла, пересекаются ли эти отрезки (т.е. программа должна найти координаты общей точки двух отрезков, если таковая существует).

1.4 Базовые структуры данных

Поскольку большинство рассматриваемых в этой книге алгоритмов выполняют обработку данных, то на процесс их проектирования и анализа существенно влияют конкретные способы организации данных. Понятие ***структуры данных*** (data structure) можно определить как особую систему организации взаимосвязанных элементов данных. Структура самих элементов данных зависит от задачи, в которой они используются. Они могут быть как очень простыми (например, представлять собой целые числа или строки символов), так и сложными структурами данных (например, для представления матриц часто используют одномерный массив, каждый элемент которого, в свою очередь, также является одномерным массивом). Тем не менее существует несколько чрезвычайно важных при проекти-

ровании алгоритмов для компьютеров структур данных. Поскольку вы наверняка знакомы с большинством из них (если не со всеми), ниже представлен лишь их краткий обзор.

Линейные структуры данных

Среди простых структур данных можно выделить две самых важных — одномерный массив и связанный список. ***Одномерный массив*** (array) — это последовательность из n однотипных элементов, расположенных подряд в оперативной памяти компьютера, доступ к которым выполняется по значению так называемого ***индекса*** массива (см. рис. 1.6).

Рис. 1.6. Одномерный массив из n элементов

В большинстве случаев индекс массива, состоящего из n элементов, является целым числом и принимает значения от 0 до $n-1$ (как показано на рис. 1.6) или от 1 до n. В некоторых языках программирования допускается, чтобы индекс массива принимал любые целочисленные значения (в том числе и отрицательные). Главное, чтобы он не выходил за пределы заранее установленного диапазона, определяющего *нижнюю* и *верхнюю* границы изменения индекса. Иногда в языках программирования для доступа к элементам массива могут использоваться нечисловые индексы. Подобные массивы называются *ассоциативными*. Например, можно создать ассоциативный массив из 12 элементов, соответствующий месяцам года, доступ к котором осуществляется по названиям месяцев.

Время обращения к любому элементу массива одинаково и не зависит от его положения в массиве. Эта особенность выгодно отличает массивы от связанных списков, о которых речь пойдет ниже. Однако не стоит забывать, что каждый элемент массива занимает одинаковое количество ячеек памяти компьютера.

На основе одномерных массивов создаются различные структуры данных. Среди них выделяются ***строки*** (string), т.е. последовательности символов заранее определенного алфавита, заканчивающиеся особым символом. Этот символ служит признаком конца строки. Строки, состоящие только из нулей или единиц, называются ***двоичными*** (***бинарными***) (binary strings) или ***битовыми*** (bit strings). Строки являются основным элементом, используемым при обработке текстовых данных, определении синтаксиса языка программирования, компилировании написанных на нем программ, а также при изучении абстрактных вычислительных моделей. Выполняемые над массивом операции в первую очередь зависят от его типа. Так, операции обычно выполняемые со строками, отличаются от операций, выполняемых над массивом чисел. В этот перечень входит:

- определение длины строки;
- сравнение двух строк, позволяющее определить, какая из них располагается раньше согласно так называемому лексикографическому порядку, т.е. как в словаре;
- объединение (конкатенация) двух строк, позволяющее сформировать из них одну строку, поместив вторую строку в конец первой.

Связанный список — это последовательность (которая может быть пустой) нескольких элементов данных, называемых ***узлами*** (nodes). В каждом узле хранится информация двух видов: собственно данные узла и одна или несколько ссылок на другие узлы связанного списка, называемых ***указателями*** (pointers). Если текущий узел связанного списка является последним, в него помещается т.н. нулевой указатель со специальным значением NULL; это говорит о том, что указатель ни на что не указывает. В ***однонаправленном связанном списке*** (singly linked list) каждый узел содержит только один указатель, в который помещается ссылка на следующий элемент списка, либо "ноль", если текущий элемент является последним (рис. 1.7).

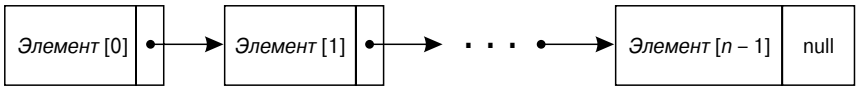


Рис. 1.7. Однонаправленный связанный список, состоящий из n элементов

Для обращения к заданному элементу связанного списка необходимо выбрать первый узел списка, а затем перемещаться по цепочке элементов до достижения нужного узла. Поэтому, в отличие от одномерных массивов, время, затрачиваемое на доступ к произвольному элементу однонаправленного списка зависит от его положения в списке. Это является существенным недостатком связанного списка. Однако у него есть и преимущество. Оно заключается в том, что для элементов связанного списка не требуется предварительное резервирование оперативной памяти компьютера. Кроме того, вставка и удаление элементов списка не представляют особого труда и выполняются достаточно быстро изменением значений нескольких указателей.

Гибкость структуры связанного списка позволяет использовать его различными способами во множестве приложений. Например, часто бывает очень удобно, чтобы в первом элементе связанного списка находился специальный узел, называемый ***заголовком*** (header). В него обычно помещают информацию о самом списке, такую как текущее количество элементов в списке и указатель на последний элемент списка.

Существует модификация структуры однонаправленного связанного списка, которая называется ***двунаправленным связанным списком*** (doubly linked list).

Отличие между ними в том, что каждый узел двунаправленного списка (кроме первого и последнего) содержит указатели на предыдущий и последующий узлы (рис. 1.8).

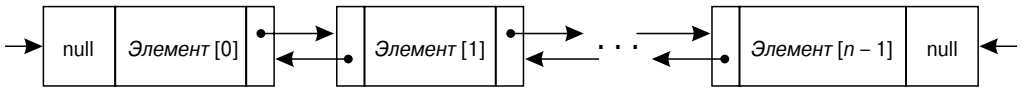


Рис. 1.8. Двунаправленный связанный список, состоящий из n элементов

Массив и связанный список чаще всего используются для представления еще одной абстрактной структуры данных, называемой линейным списком или просто списком. ***Список*** (list) представляет собой конечную последовательность элементов данных, т.е. набор элементов данных, организованных в заданном линейном порядке. К этой структуре данных могут применяться такие базовые операции, как поиск, добавление и удаление элемента.

Среди всего множества списков можно выделить два типа: стеки и очереди. ***Стеком*** (stack) называют такой список, в котором можно удалить только последний элемент, а новый элемент добавить только в его конец. Последний часто называют ***вершиной*** (top) стека, поскольку стеки обычно изображают на рисунках в виде вертикального прямоугольника (по аналогии со стопкой тарелок, которую он напоминает). Кратко сформулировать принцип работы стека можно так: "последним вошел, первым вышел" ("last-in-first-out", или LIFO), поскольку первым из стека всегда удаляется элемент, добавленный в него последним. Этим стек напоминает стопку тарелок, поскольку мы можем взять из нее только верхнюю тарелку, а новую тарелку — положить на верх стопки. Стеки широко применяются на практике, в частности, без них не обойтись при реализации рекурсивных алгоритмов.

Под ***очередью*** (queue) понимается такой список, элементы которого удаляются с одной стороны структуры (она называется ***головой*** (front)), а добавляются с другой (она называется ***хвостом*** (rear)). Первая операция называется ***удалением*** элемента из очереди (dequeue), а вторая — ***постановкой*** в очередь (enqueue). Таким образом, принцип работы очереди можно сформулировать так: "первым вошел, первым вышел" ("first-in-first-out", или FIFO). Здесь прослеживается полная аналогия с очередью в магазине, которая образуется, если продавец медленно отпускает товар или наплыв покупателей слишком велик. Очереди также находят широкое практическое применение, в частности, в некоторых алгоритмах решения задач теории графов.

Во многих важных приложениях требуется выбрать среди динамически изменяемого множества элемент с наивысшим приоритетом. Часто для решения подобной задачи применяют структуру данных, которая называется ***очередью с приоритетами***, или ***приоритетной очередью*** (priority queue). Эта очередь представ-

ляет собой совокупность элементов данных, относящихся к вполне упорядоченному универсуму[10] (чаще всего целых или вещественных чисел). К основным операциям над очередью с приоритетами относятся: поиск наибольшего элемента, извлечение наибольшего элемента и добавление нового элемента. Само собой разумеется, что последние две операции должны быть реализованы так, чтобы в результате их выполнения получалась новая очередь с приоритетами (либо переупорядочивалась старая). Проще всего реализовать рассматриваемую структуру данных на основе массива или упорядоченного массива, однако ни одно из этих решений не является оптимальным в плане достижения максимально возможной эффективности. В более удачных реализациях используется оригинальная структура данных, называемая ***пирамидой*** (heap)[11]. Эту структуру данных и основные алгоритмы сортировки на ее основе мы обсудим в разделе 6.4.

Графы

Как уже упоминалось, граф нестрого можно определить как совокупность точек на плоскости, называемых *вершинами*, или *узлами*, часть из которых соединена отрезками, называемыми *ребрами*, или *дугами*. Строго ***граф*** $G = \langle V, E \rangle$ определяется парой множеств: конечного множества элементов V, называемых ***вершинами***, и множества E, содержащего пары вершин, называемые ***ребрами***. Если пары вершин неупорядочены, т.е. пара вершин (u, v) означает то же самое, что и пара (v, u), то такой граф называют неориентированным. Если же пары упорядочены, то говорят, что ребро (u, v) ориентировано из вершины u к вершине v, при этом сам граф G называют ***ориентированным*** (directed). Ориентированные графы называют также ***диграфами*** (digraph).

Для удобства вершины графа обозначают латинскими буквами, целыми числами или даже символьными строками, если этого требуют условия задачи (рис. 1.9). Граф, показанный на рис. 1.9*а*, имеет 6 вершин и семь ребер:

$$V = \{a, b, c, d, e, f\}, \quad E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}.$$

Ориентированный граф, изображенный на рис. 1.9*б*, имеет шесть вершин и восемь ориентированных ребер (дуг):

$$V = \{a, b, c, d, e, f\}, \quad E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}.$$

[10] Математический термин, означающий некоторое множество, фиксированное в рамках данной математической теории и содержащее в качестве элементов все объекты, рассматриваемые в этой теории. Употребляется как синоним термина ***универсальное множество***. *Математическая энциклопедия*, т. 5, М.: Сов. энц., 1985. — *Прим. ред.*

[11] Изначально термин “heap” использовался в контексте пирамидальной сортировки (heapsort), но в последнее время его основной смысл изменился, и он стал обозначать память со сборкой мусора (в частности, в языках программирования Lisp и Java) и переводиться как “куча”. Однако в данной книге термину heap (который здесь переводится как “пирамида”) возвращен его первоначальный смысл. — *Прим. ред.*

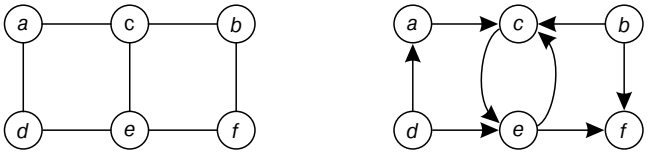


Рис. 1.9. а) Неориентированный граф; б) ориентированный граф

В приведенном выше определении графа не исключены ***петли*** (loops), или ребра, начинающиеся и заканчивающиеся в одной вершине. В книге мы будем считать, что у графа не может быть петель, если явно не указано иное. Поскольку по определению не разрешается, чтобы между двумя вершинами неориентированного графа существовало несколько ребер, то для такого графа, имеющего $|V|$ вершин и не содержащего петель, можно оценить количество возможных ребер $|E|$ с помощью следующего неравенства:

$$0 \leqslant |E| \leqslant |V|\,(|V|-1)/2.$$

Если каждая из $|V|$ вершин графа соединяется ребрами с остальными $|V|-1$ вершинами, то количество ребер в нем будет максимальным. Поскольку граф неориентированный, произведение $|V|\,(|V|-1)$ необходимо разделить на 2, поскольку между двумя вершинами (u, v) имеется 2 ребра: от u к v и от v к u.

Граф, в котором каждая пара вершин соединяется ребром, называется ***полным*** (complete). Стандартное обозначение полного графа, состоящего из $|V|$ вершин, — $K_{|V|}$. Граф, в котором отсутствует относительно небольшое количество ребер, называется ***плотным*** (dense), и, напротив, граф, в котором присутствует относительно небольшое количество ребер[12], называется ***разреженным*** (sparse). От того, с каким графом мы имеем дело (плотным или разреженным), зависит способ его представления и, следовательно, время выполнения разрабатываемого или используемого алгоритма.

Представление графов

Графы, используемые в компьютерных алгоритмах, могут быть представлены двумя принципиально разными способами: в виде матрицы смежности и в виде связанных списков смежных вершин. ***Матрица смежности*** (adjacency matrix) графа, состоящего из n вершин, представляет собой булеву матрицу размером $n \times n$, в которой каждая строка и каждый столбец соответствуют одной из вершин графа. Элемент этой матрицы, находящийся на пересечении i-ой строки и j-го

[12] В этих определениях фразу "относительно небольшое" следует понимать как количество ребер, малое относительно количества ребер полного графа. — *Прим. ред.*

столбца, равен 1 в случае, если i-я и j-я вершины графа соединяются ребром, и 0 в противном случае[13]. Например, на рис. 1.10*а* показана матрица смежности для графа, изображенного на рис. 1.9*а*. Обратите внимание, что матрица смежности для неориентированного графа всегда симметрична (понятно, почему?), т.е. $A[i,j] = A[j,i]$ для всех $0 \leqslant i,j \leqslant n-1$.

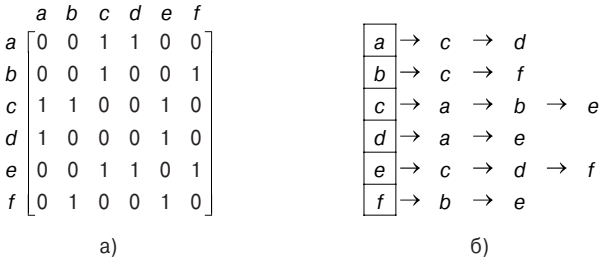


Рис. 1.10. Матрица смежности а) и связанные списки смежных вершин б) для графа, изображенного на рис. 1.9*а*

Связанные списки смежных вершин (adjacency linked lists) графа представляют собой совокупность связанных списков (по одному для каждой вершины), в которых содержится информация обо всех смежных вершинах текущей вершины (т.е. в каждом связанном списке содержится список всех вершин, соединенных ребром с текущей вершиной). Как правило, в начале каждого подобного списка располагается заголовок, содержащий информацию, идентифицирующую вершину, для которой этот список составлен. Например, на рис. 1.10*б* с помощью связанных списков смежных вершин представлен граф, изображенный на рис. 1.9*а*. Если взглянуть на все с другой стороны, то в связанных списках присутствуют вершины графа, которым в матрице смежности соответствуют единичные элементы.

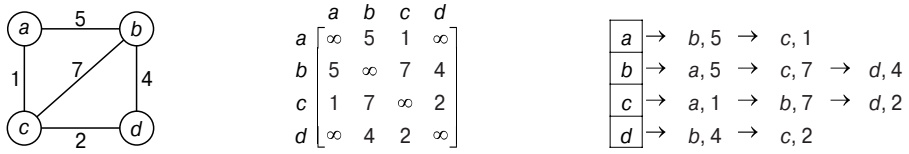
Связанные списки смежных вершин имеет смысл использовать для представления *разреженных* графов, поскольку при этом требуется меньше оперативной памяти по сравнению с матрицей смежности (которая, кстати, в этом случае будет состоять практически из одних нулей). Естественно, что дополнительную память, которую занимают указатели связанного списка, мы в расчет не принимаем. Однако ситуация в корне меняется в случае плотных графов. Вообще говоря, выбор способа представления графа зависит от типа задачи, алгоритма, используемого для ее решения и, возможно, от типа исходного графа (в зависимости от того, разреженный он или плотный).

[13] В общем случае значения элементов матрицы смежности равны не только 0 или 1 (т.е. матрица не является булевой). Каждый элемент матрицы, находящийся на пересечении i-й строки и j-го столбца, соответствует количеству ребер, соединяющих i-ю и j-ю вершины графа. — *Прим. ред.*

Взвешенные графы

Под ***взвешенным графом*** (weighted graph) подразумевается граф, ребрам которого поставлены в соответствие числа, как показано на рис. 1.11*а*. Эти числа называются ***весами*** (weights) или ***ценами***, или ***стоимостями*** (costs). Интерес к подобного типа графам был вызван большим количеством прикладных задач, таких как задача поиска кратчайшего маршрута между двумя пунктами или уже упоминавшаяся выше задача коммивояжера. Интересно, что определение кратчайшего пути может использоваться как при транспортировке грузов, так и в сетях передачи данных.

Оба рассмотренных выше способа представления графов легко приспособить и для случая взвешенных графов. Если взвешенный граф представляется с помощью матрицы смежности A, то ее элемент $A[i, j]$ должен равняться весу ребра, соединяющего i-ю и j-ю вершины. Если же эти вершины не соединены ребром, то вместо веса элемент $A[i, j]$ должен содержать какой-нибудь специальный знак, например знак бесконечности ∞. Такая матрица называется ***матрицей весов*** (weight matrix) или ***матрицей стоимости*** (cost matrix). Подобный подход проиллюстрирован на рис. 1.11*б*. (В некоторых случаях удобнее, чтобы на главной диагонали матрицы смежности располагались нули, а не знаки бесконечности.) Что касается представления взвешенного графа с помощью связанных списков смежных вершин, то каждый узел списка нужно немного расширить, включив в него не только имя смежной вершины, но и весовой коэффициент соответствующего ребра, как показано на рис. 1.11*в*.



а) б) в)

Рис. 1.11. *а*) Взвешенный граф и его представление *б*) с помощью матрицы весов и *в*) связанных списков смежных вершин

Пути и циклы

Среди множества интересных свойств графа можно выделить два особенно важных для решения огромного количества прикладных задач: ***связность*** (connectivity) и ***ацикличность*** (acyclicity). Оба этих свойства основаны на понятии пути. ***Путь***, или ***маршрут***, (path) от вершины u к вершине v можно определить как последовательность смежных (т.е. соединенных ребром) вершин, которая начинается в вершине u и заканчивается в вершине v. Если все ребра графа различны, то такой путь называют ***простым*** (simple). Под ***длиной*** (length) пути понимает-

ся общее количество вершин в последовательности, определяющей путь, минус единица. Другими словами, длина пути равна количеству ребер, через которые он проходит. Например, на рис. 1.9*а* от вершины a до вершины f можно проложить простой путь длиной 3: a, c, b, f. В то же время между этими вершинами существует путь (не простой) длиной 5: a, c, e, c, b, f.

В случае ориентированных графов нас обычно будут интересовать ориентированные пути. Под ***ориентированным маршрутом*** (directed path) понимается последовательность вершин, в которой каждая следующая друг за другом пара вершин (u, v) соединена дугой, начинающейся в вершине u и заканчивающейся в вершине v. Например, на рис. 1.9*б* существует следующий ориентированный путь из вершины a к вершине f: a, c, e, f.

Граф называется ***связным*** (connected), если для любой пары его вершин u и v существует путь от u к v. Объяснить это свойство графа, что называется, "на пальцах" можно так: если создать модель этого графа, связав между собой веревками (которые будут выполнять роль ребер) несколько мячей (они будут выполнять роль вершин), то все предметы можно будет удержать в одной руке. Если граф не является связным, то модель будет состоять из нескольких отдельных участков, которые называются связными компонентами. Строго ***связный компонент*** (connected component) графа можно определить как максимальный связный подграф[14] данного графа, который нельзя расширить за счет включения дополнительной вершины, смежной с одной из ее вершин. Например, графы, изображенные на рис. 1.9*а* и 1.11*а*, являются связными, тогда как граф, изображенный на рис. 1.12, не является связным, поскольку нельзя проложить путь, скажем, от вершины a к вершине f. Граф, изображенный на рис. 1.12, имеет два связанных компонента с вершинами $\{a, b, c, d, e\}$ и $\{f, g, h, i\}$ соответственно.

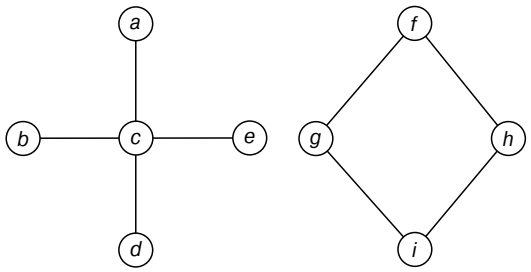


Рис. 1.12. Пример несвязного графа

Графы с несколькими компонентами связности успешно применяются при решении реальных прикладных задач. В качестве примера можно привести пред-

[14] ***Подграфом*** (subgraph) графа $G = \langle V, E\rangle$ является граф $G' = \langle V', E'\rangle$, такой, что $V' \subseteq V$ и $E' \subseteq E$.

ставление в виде графа системы магистральных шоссейных дорог (highway system), проложенных между штатами в США (можете объяснить, почему?).

Во многих прикладных задачах важно знать, содержит ли рассматриваемый граф циклы. Под ***циклом*** (cycle) понимается простой путь положительной длины, который начинается и заканчивается в одной и той же вершине. Например, в графе, изображенном на рис. 1.12, циклом является путь: f, h, i, g, f. Граф, не содержащий циклов, называют ***ациклическим*** (acyclic). Ациклические графы будут рассмотрены в следующем подразделе.

Деревья

Деревом (tree) (точнее, ***свободным деревом*** (free tree)) называется связный ациклический граф (рис. 1.13*а*). Граф, который не содержит циклов, но не обязательно является связным, называется ***лесом*** (forest). Каждая компонента связности леса является деревом (рис. 1.13*б*).

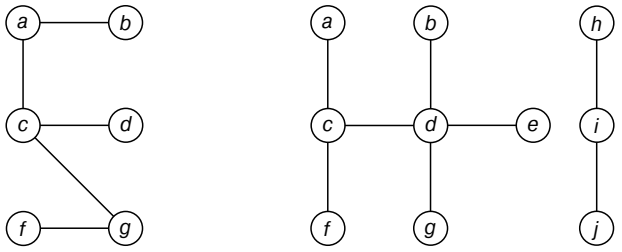


Рис. 1.13. Пример *а*) дерева и *б*) леса

Деревья имеют несколько важных свойств, не присущих другим графам. В частности, число ребер в дереве всегда на единицу меньше, чем число его вершин:

$$|E| = |V| - 1.$$

На примере графа, показанного на рис. 1.12, можно сделать вывод, что это свойство является необходимым, но не достаточным, чтобы граф был деревом. Тем не менее для связных графов это свойство является достаточным, поэтому с его помощью удобно определять, содержит ли граф цикл.

Корневые деревья

Еще одним важным свойством деревьев является следующее: для любых двух вершин дерева существует только один простой путь от одной вершины к другой. Это свойство позволяет назначить произвольный узел в свободном дереве ***корнем*** (root) и преобразовать свободное дерево в так называемое ***корневое дерево*** (rooted

tree). Последнее обычно изображается так, чтобы его корень был вверху, т.е. располагался на так называемом нулевом уровне дерева. Ниже корня располагаются смежные с ним вершины. Они составляют первый уровень дерева. Вершины, соединенные с корнем двумя ребрами, составляют следующий (т.е. второй) уровень дерева, и т.д. На рис. 1.14 показано, как преобразовать свободное дерево в корневое.

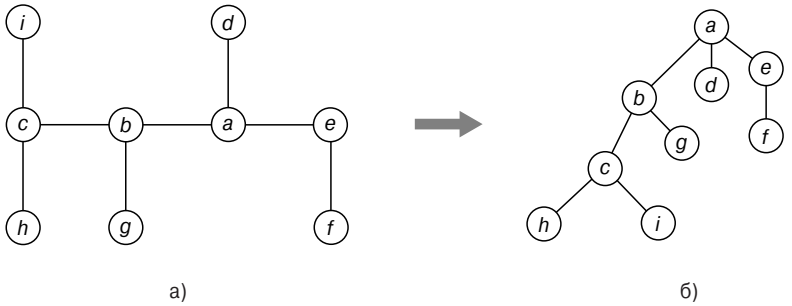


Рис. 1.14. Преобразование *а*) свободного дерева в *б*) корневое

Корневые деревья играют в информатике гораздо более важную роль, чем свободные деревья. Поэтому часто для краткости их называют просто “деревьями”. Их основное назначение — описание иерархических структур, таких как структура каталогов файловой системы или организационная структура предприятия. Однако существуют и другие, менее очевидные области применения деревьев, например для организации словарей (об этом пойдет речь в следующем подразделе), для эффективного хранения очень больших массивов данных (см. раздел 7.4), для кодирования данных (см. раздел 9.4). Как вы увидите в главе 2, деревья также находят применение при анализе рекурсивных алгоритмов. И, чтобы закончить этот далеко не полный перечень возможных применений деревьев, следует упомянуть так называемые ***деревья пространств состояний*** (state-space trees). С их помощью можно объяснить два важных метода проектирования алгоритмов: поиск с возвратом (backtracking) и метод ветвей и границ (branch-and-bound). О них речь пойдет в разделах 11.1 и 11.2.

Для любой вершины v в дереве T верно следующее: все вершины, принадлежащие простому маршруту от корня дерева до этой вершины, называются ***предками*** (ancestors) v. Сама вершина v обычно считается собственным предком. Множество предков, в которое не включен собственный предок, называется множеством ***истинных предков*** (proper ancestors), или просто ***истинными предками***. Если (u, v) является последним ребром, принадлежащим простому маршруту, ведущему из корня дерева до вершины v (и $u \neq v$), то говорят, что u является ***родителем*** (parent) v, а v называют ***потомком***, или ***дочерней*** (child) вершиной u. Вершины, имеющие общего родителя, называются ***родственными***, или ***сестринскими***

(sibling). Вершина, у которой нет потомков, называется ***листом*** (leaf). Вершина, у которой есть как минимум один потомок, называется ***родительской*** (parental). Совокупность всех вершин, для которых вершина v является предком, называют ***потомками*** (descendants) v. Вершина v вместе со всем своими потомками называется ***поддеревом*** (subtree) дерева T с корнем (поддерева) в вершине v. Таким образом, для дерева, изображенного на рис. 1.14*б*, корнем является вершина a; вершины d, g, f, h и i являются листьями; вершины a, b, e и c являются родительскими. Родительской вершиной для b является вершина a; потомками вершины b являются вершины c и g; родственными для b являются вершины d и e; вершинами поддерева с корнем в b являются: $\{b, c, g, h, i\}$.

Под ***глубиной*** (depth) вершины v понимается длина простого пути от корня до этой вершины. Под ***высотой*** (height) дерева понимается длина наибольшего простого пути от его корня до одного из листьев. Например, в дереве, изображенном на рис. 1.14*б*, глубина вершины c равна 2, а высота дерева равна 3. Таким образом, если сосчитать уровни дерева сверху вниз, начиная с корня (ему соответствует нулевой уровень), то глубина вершины будет равна уровню этой вершины в дереве, а высота дерева будет соответствовать максимальному уровню, на котором могут находиться вершины в дереве. (Обратите внимание, что в некоторых книгах высота дерева определяется как количество уровней в дереве. При подобном определении высота дерева будет на единицу больше, чем высота, которая соответствует длине наибольшего простого маршрута от корня до одного из листьев.)

Упорядоченные деревья

Упорядоченным (ordered) называется такое корневое дерево, в котором упорядочены все потомки каждой вершины. Принято считать, что на диаграмме деревья изображаются так, чтобы все потомки были упорядочены слева направо. ***Двоичное*** (***бинарное***) ***дерево*** (binary tree) можно определить как упорядоченное дерево, каждая вершина которого имеет не более двух потомков, причем каждый из потомков считается либо ***левым*** (left child) либо ***правым*** (right child) потомком своего родителя. Поддерево, корень которого находится в левом (правом) потомке вершины, называется ***левым*** (***правым***) поддеревом этой вершины. Пример бинарного дерева показан на рис. 1.15*а.*

На рис. 1.15*б* вершинам бинарного дерева, показанного на рис. 1.15*а*, назначены числа. Обратите внимание, что число, которое назначено каждой родительской вершине, больше, чем число, назначенное ее левому потомку, и меньше, чем число, назначенное ее правому потомку. Деревья подобного типа называют ***бинарным деревом поиска*** (binary search trees). Бинарные деревья и бинарные деревья поиска находят широкое применение в информатике. С некоторыми из них вы столкнетесь при чтении этой книги. Бинарные деревья поиска являются частным случаем более общего типа деревьев поиска, которые называются ***многоканальными де-***

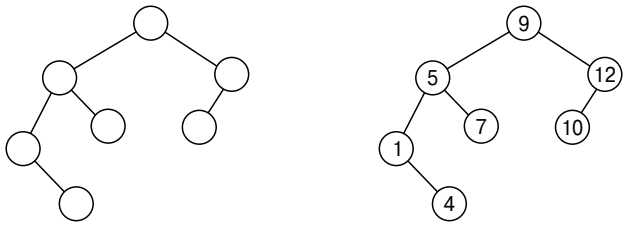


Рис. 1.15. Пример *а*) бинарного дерева и *б*) бинарного дерева поиска

ревьями поиска (multiway search trees). Последние незаменимы при организации эффективного хранения на дисках файлов очень больших размеров.

Ниже в этой книге мы покажем, что эффективность большинства основных алгоритмов, в которых используются бинарные деревья поиска и их вариации, непосредственно зависит от высоты дерева. Поэтому приведенное ниже неравенство для высоты бинарного дерева h, содержащего n вершин, особенно важно для анализа подобных алгоритмов:

$$\lfloor \log_2 n \rfloor \leqslant h \leqslant n - 1.$$

При использовании в компьютерных программах бинарные деревья обычно реализуют в виде связанного списка, узлы которого соответствуют вершинам дерева. В каждый узел помещается информация о соответствующей ему вершине дерева (ее имя, присвоенное значение и т.д.), а также два указателя на его левого и правого потомка. На рис. 1.16 показана одна из подобных реализаций бинарного дерева поиска, показанного на рис. 1.15*б*.

Упорядоченное дерево в компьютере можно представить в виде родительской вершины с указателями на всех ее потомков. Однако такое представление не совсем удобно в том случае, когда количество потомков в процессе работы программы меняется в очень широких пределах. Чтобы избежать подобного неудобства, можно воспользоваться структурой данных, каждый узел которой будет иметь только два указателя (по аналогии с представлением бинарных деревьев). При этом левый указатель будет содержать ссылку на первого потомка текущей вершины, а правый — на следующую родственную ей вершину. Поэтому такое представление произвольного упорядоченного дерева называют "***первый потомок — следующий родственник***" (first child — next sibling). Таким образом, все родственные вершины будут связаны в один список с помощью правого указателя текущей вершины. При этом ссылка на первый элемент списка будет содержаться в левом указателе ее родителя. На рис. 1.17*а* показано подобное представление дерева, показанного на рис. 1.14*б*. Нетрудно заметить, что в результате подобного представления упорядоченное дерево оказалось преобразовано в бинарное. Поэтому

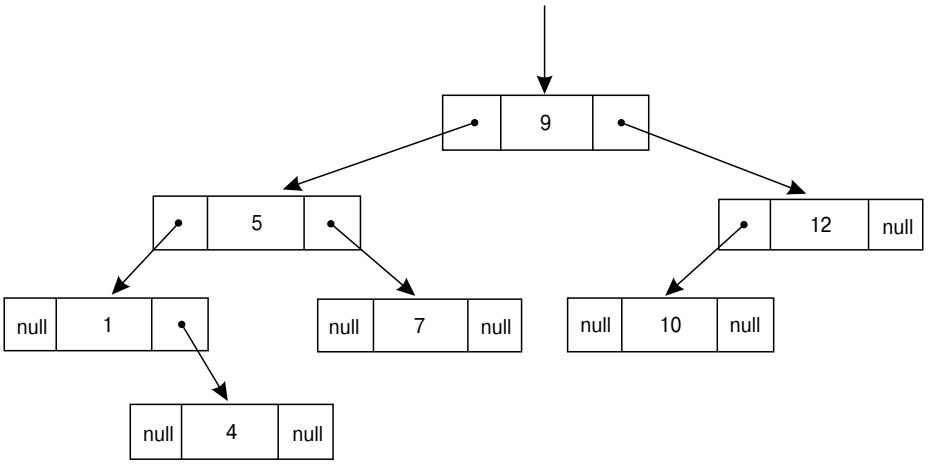


Рис. 1.16. Типовая реализация бинарного дерева поиска, показанного на рис. 1.15*б*

говорят, что данное бинарное дерево ассоциировано с упорядоченным деревом. Чтобы убедиться в этом, достаточно "развернуть" стрелки указателей по часовой стрелке примерно на 45°, как показано на рис 1.17*б*.

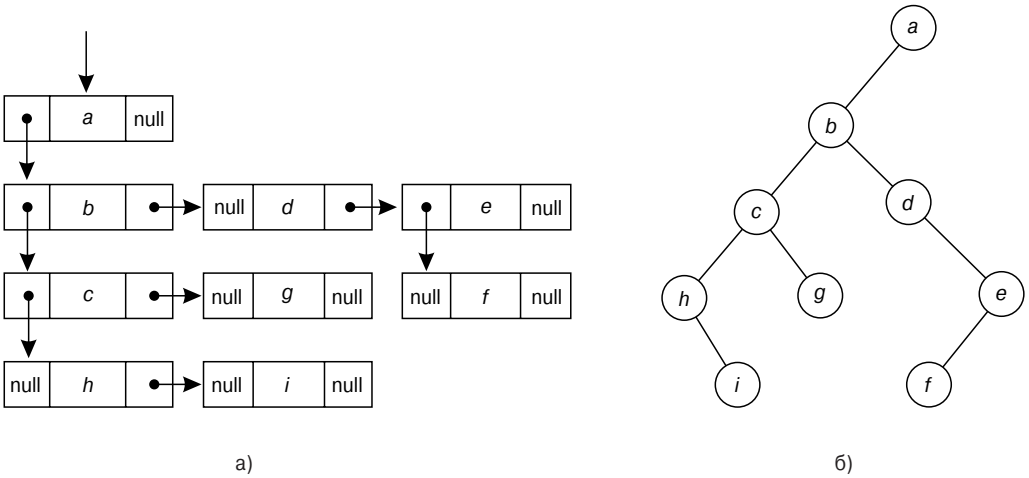


Рис. 1.17. *а*) Представление графа, показанного на рис. 1.14*б* в виде структуры "первый потомок — следующий родственник"; *б*) ассоциированное с ним бинарное дерево

Множества и словари

Понятие множества играет основную роль в математике. ***Множество*** (set) можно охарактеризовать как неупорядоченную совокупность (которая может быть

пуста) отдельных элементов, называемых элементами множества. Конкретное множество определяется либо в виде явного списка элементов (например, $S = \{2, 3, 5, 7\}$), либо в виде указания условия, которому должны удовлетворять все элементы множества и только они (например, $S = \{n,$ где n — простое число, меньшее $10\}$). Основными операциями над множествами являются: проверка на членство в данном множестве (т.е. принадлежит ли элемент множеству; другими словами, нужно найти заданный элемент среди элементов множества), поиск объединения двух множеств (т.е. формирование нового множества, элементы которого принадлежат либо первому, либо второму множеству, либо обоим множествам одновременно), а также поиск пересечения двух множеств (т.е. формирование нового множества, элементы которого принадлежат только обоим множествам одновременно).

При разработке компьютерных приложений множества можно реализовать двумя способами. В первом случае рассматриваются только множества, которые являются подмножествами некоторого большого множества U, называемого ***универсальным множеством***, или ***универсумом*** (universal set). Если множество U состоит из n элементов, то любое его подмножество S можно представить в виде строки из n битов, называемой ***битовым вектором*** (bit vector), в которой i-й элемент равен 1 тогда и только тогда, когда i-й элемент множества U включен в подмножество S. Поэтому, возвращаясь к нашему примеру, если $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, то подмножество $S = \{2, 3, 5, 7\}$ можно представить в виде битовой строки 011010100. Описанный способ представления множеств позволяет реализовать стандартный набор операций над множеством в виде очень быстрых процедур. Однако платой за скорость является большой объем оперативной памяти, необходимой для представления элементов множества[15].

Чаще всего в компьютерных программах используется второй способ представления множества. Он заключается в представлении элементов множества в виде связанного списка. Само собой разумеется, речь идет о представлении только конечных множеств. К счастью, в отличие от математики, данный вид множеств наиболее широко используется в компьютерных приложениях. Тем не менее необходимо обратить внимание на два принципиальных различия между множествами и списками. Во-первых, в множестве не может быть одинаковых элементов, а в списке — может. Иногда это требование к уникальности элементов обходят путем введения так называемого ***мультимножества*** (multiset), или ***пакета*** (bag), т.е. неупорядоченной совокупности элементов, которые необязательно должны быть различными. Во-вторых, поскольку множество — это неупорядоченная совокупность элементов, изменение порядка его элементов не изменяет само множество. В свою очередь, список определяется как упорядоченная со-

[15] В нашей книге это первый пример компромисса между временем выполнения алгоритма и требуемым объемом оперативной памяти. Более подробно мы поговорим об этом в главе 7.

вокупность элементов, что в корне противоположно понятию множества. Выше мы перечислили важные теоретические различия между множеством и списком, однако, к счастью, они не так важны для большинства приложений. Кроме того, стоит упомянуть, что в том случае, когда множество представляется в виде списка, то для ряда приложений имеет смысл поддерживать его в отсортированном виде.

В прикладных программах над множеством или мультимножеством чаще всего выполняются следующие операции: поиск заданного элемента, добавление нового элемента и удаление элемента из совокупности. Структуру данных, задействованную в реализации трех этих операций, называют ***словарем*** (dictionary). Заметим, что о взаимосвязи этой структуры данных с задачами поиска говорилось выше в разделе 1.3. Очевидно, что здесь речь идет о выполнении поиска в динамически изменяемом контексте. Поэтому эффективная реализация словаря должна быть результатом достижения компромисса между эффективным выполнением поиска и двух других операций над множеством. В действительности существует несколько способов реализации словаря. Они могут быть как очень простыми, с использованием обычных массивов сортированных (или несортированных) элементов, так и довольно сложными. В последнем случае используются методы хеширования и сбалансированные деревья поиска, которые будут описаны в последующих главах этой книги.

При решении ряда прикладных задач из области вычислительной техники нужно динамически разбить множество, состоящее из n элементов, на ряд непересекающихся подмножеств. После инициализации множества как набора из n одноэлементных подмножеств задача сводится к последовательности операций поиска и объединения. Эта задача называется задачей ***объединения множества*** (set union). Эффективные алгоритмы решения этой задачи будут описаны в разделе 9.2 при рассмотрении одного из важных ее применений.

Вы, наверное, уже заметили, что в приведенном выше обзоре основных структур данных мы почти всегда упоминали об особенностях операций, которые обычно выполняются над рассматриваемыми структурами данных. Подобную тесную взаимосвязь между структурами данных и выполняемыми над ними операциями специалисты в области информатики заметили уже давно. Это навело их на мысль о существовании так называемых ***абстрактных типов данных*** (abstract data type, или ***ADT***), т.е. множества абстрактных объектов, представляющих элементы данных, и определенного на нем набора операций, которые могут быть выполнены над элементами этого множества. В качестве иллюстрации этого понятия перечитайте, например, приведенные выше определения приоритетной очереди и словаря. Несмотря на то что абстрактные типы данных можно реализовать с помощью старых процедурных языков программирования, таких как Pascal (в качестве примера см. [5]), все-таки намного удобнее делать это с помощью объектно-ориентированных языков программирования, таких как C++ или Java, в которых абстрактные типы данных поддерживаются с помощью ***классов*** (classes).

Упражнения 1.4

1. Поясните, как можно реализовать каждую из перечисленных ниже операций над массивом так, чтобы время ее выполнения не зависело от размера массива n.

а) Удаление i-го элемента массива $1 \leqslant i \leqslant n$).

б) Удаление i-го элемента отсортированного массива (разумеется, образовавшийся после удаления массив также должен остаться отсортированным).

2. Предположим, нужно найти число в списке, состоящем из n элементов. Насколько упростится решение, если список будет отсортирован? Рассмотрите отдельные решения для случаев, когда

а) список представлен в виде массива;

б) список представлен в виде связанного списка.

3. а) Каким будет содержимое стека после выполнения каждой команды приведенной ниже последовательности, считая, что в исходном состоянии стек пуст:

$$push\,(a),\ push\,(b),\ pop,\ push\,(c),\ push\,(d),\ pop.$$

б) Изобразите содержимое очереди после выполнения каждой команды приведенной ниже последовательности, считая, что в исходном состоянии очередь пуста:

$$enqueue\,(a),\ enqueue\,(b),\ dequeue,\ enqueue\,(c),\ enqueue\,(d),\ dequeue.$$

4. а) Предположим, A является матрицей смежности неориентированного графа. Объясните, какое из свойств матрицы свидетельствует о том, что:

1) граф является полным;

2) граф содержит петли, т.е. какая-либо из вершин соединена ребром сама с собой;

3) граф содержит изолированную вершину, т.е. в нем встречается вершина, не соединенная ребрами с другими вершинами.

б) Дайте ответы на те же вопросы для представления в виде связанных списков смежности.

5. Приведите подробное описание алгоритма преобразования свободного дерева в корневое дерево, корень которого находится в заранее определенной вершине свободного дерева.

6. Докажите приведенное ниже неравенство для высоты бинарного дерева h, содержащего n вершин:

$$\lfloor \log_2 n \rfloor \leqslant h \leqslant n - 1.$$

7. Покажите, как можно реализовать абстрактный тип данных очереди с приоритетами в виде:

а) (несортированного) массива;

б) отсортированного массива;

в) бинарного дерева поиска.

8. Как бы вы реализовали словарь, содержащий достаточно малое количество элементов n, при условии, что все его элементы различны (например, представляют собой названия 50 штатов США)? Опишите реализацию каждой операции, выполняемой словарем.

9. Для каждой из перечисленных ниже задач укажите наиболее подходящую с вашей точки зрения структуру данных.

а) Система автоматического ответа на телефонные звонки с учетом их приоритетности.

б) Система отправки покупателям счетов за сделанные заказы, которая бы учитывала порядок поступления заказов.

в) Реализация калькулятора для вычисления простых арифметических выражений.

10. Разработайте алгоритм проверки того, являются ли два заданных слова анаграммами, т.е. того, что одно из слов может быть получено путем перестановки букв другого слова. (Например, анаграммами являются слова *кот* и *ток*.)

Резюме

- *Алгоритм* — это последовательность четко определенных инструкций, предназначенных для решения некоторой задачи за ограниченный промежуток времени. Входные данные, обрабатываемые алгоритмом, определяют *экземпляры задачи*, решаемой с помощью выбранного алгоритма.
- Алгоритмы могут быть описаны на естественном языке либо на псевдокоде. Они также могут быть реализованы в виде компьютерных программ.
- Существует несколько способов классификации алгоритмов, но среди них можно выделить две принципиальные альтернативы:

  - группирование алгоритмов в соответствии с типом решаемой с их помощью задачи;
  - группирование алгоритмов в соответствии с лежащими в их основе методами проектирования.
- Важными типами задач являются: сортировка, поиск, обработка строк, задачи теории графов, комбинаторные задачи, геометрические задачи и численные задачи.
- *Метод проектирования* алгоритма — это универсальный подход, применяемый для алгоритмического решения широкого круга задач, относящихся к различным областям вычислительной техники.
- Несмотря на то что проектирование алгоритма, безусловно, — творческий процесс, в нем тем не менее можно выделить последовательность взаимосвязанных действий, которые необходимо выполнить, как показано на рис. 1.2.
- Хороший алгоритм получается, как правило, в результате кропотливой циклической работы, связанной с возможными переделками.
- Часто одну и ту же задачу можно решить с помощью нескольких алгоритмов. Например, для вычисления наибольшего общего делителя двух целых чисел в этой главе было описано три алгоритма: *алгоритм Евклида*, алгоритм последовательного перебора целых чисел и алгоритм, который вы изучали в средней школе. Последний алгоритм был нами немного усовершенствован: для получения упорядоченного списка простых чисел мы применили алгоритм под названием *решето Эратосфена*.
- Алгоритмы работают с данными. Поэтому для эффективного решения алгоритмической задачи необходимо правильно структурировать данные. Среди простых структур данных самыми важными являются *массив* и *связанный список*. Они используются для представления более абстрактных структур данных, таких как *список*, *стек*, *очередь*, *граф* (представляется с помощью *матрицы смежности* или *связанных списков смежных вершин*), *бинарное дерево* и *множество*.
- Множество абстрактных объектов, представляющих элементы данных и набор операций, которые могут быть выполнены над элементами этого множества, называются *абстрактным типом данных* (ADT). Важными примерами абстрактных типов данных являются *список*, *стек*, *очередь*, *очередь с приоритетами* и *словарь*. В современных языках программирования реализация абстрактных типов данных поддерживается с помощью классов.