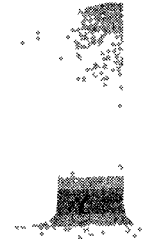


Сортировка



Как отсортировать последовательность записей в определенном порядке? Ответ обычно прост: используйте библиотечную функцию сортировки. Именно этот подход был применен в решении 1.1 и в программе поиска анаграмм из раздела 2.8 главы 2 (в последнем случае дважды). К сожалению, это не всегда срабатывает. Существующие подпрограммы сортировки может оказаться слишком утомительным подключать, и они могут оказаться слишком медленными для данной задачи (как в разделе 1.1). В такой ситуации у программиста нет выбора: приходится писать функцию сортировки самому.

11.1. Сортировка вставкой

Большинство картежников, сами того не сознавая, пользуются именно таким методом сортировки для упорядочения пришедших им карт. Когда игрок получает очередную карту, все предыдущие уже отсортированы, поэтому он просто вставляет ее в нужное место. Для сортировки массива $x[n]$ в порядке возрастания начинать следует с первого элемента, считая его отсортированной подпоследовательностью $x[0..0]$. Затем нужно вставлять элементы $x[1]$, ..., $x[n-1]$ в правильные позиции, как это делается в приведенном ниже псевдокоде:

```
for i = [1, n)
    /* инвариант  элементы x[0 .. i-1] отсортированы */
    /* цель  вставить x[i] в нужную позицию в массиве x[0 .. i] */
```

Последовательность сортировки массива из четырех элементов иллюстрируется ниже. Символ «|» показывает текущее значение переменной i ; элементы слева от этого символа уже отсортированы, а справа — нет.

```
3|1 4 2
1 3|4 2
```

```
1 3 4|2
1 2 3 4|
```

Вставка элемента в нужную позицию производится циклом, в котором элементы перебираются справа налево, а в переменной `j` хранится индекс очередного вставляемого элемента. В цикле текущий элемент переставляется местами с предыдущим, если этот предыдущий элемент существует (то есть `j > 0`) и текущий элемент еще не установлен в нужное положение (он и предыдущий элементы находятся в неправильном порядке). Итак, получившаяся программа сортировки вставкой приведена на листинге 11.1.

Листинг 11.1. Сортировка вставкой: версия 1

```
for i = [1, n)
  for (j = i, j > 0 && x[j-1] > x[j], j--)
    swap(j-1, j)
```

В тех редких случаях, когда мне нужно написать свою собственную сортировку, я начинаю именно с этой функции, потому что она очень проста — всего три очевидные строки.

Программисты, стремящиеся к оптимизации, могут счесть нерациональным вызов функции `swap` из тела внутреннего цикла. Программу можно ускорить, раскрыв функцию явно, хотя многие оптимизирующие компиляторы способны делать это за нас. Заменяем вызов функции нижеследующим кодом, в котором переменная `t` используется для обмена `x[j]` и `x[j-1]`.

```
t = x[j], x[j] = x[j-1], x[j-1] = t
```

На моем компьютере вторая версия сортировки работает примерно в три раза быстрее, чем первая.

После этого улучшения появляется возможность сделать следующий шаг. Поскольку переменной `t` несколько раз присваивается одно и то же значение (исходно находящееся в `x[i]`), мы можем вынести присваивания, относящиеся к этой переменной, за рамки внутреннего цикла, а также изменить вид сравнения, что даст третью версию сортировки вставкой (листинг 11.2).

Листинг 11.2. Сортировка вставкой: версия 3

```
for i = [1, n)
  t = x[i]
  for (j = i, j > 0 && x[j-1] > t, j--)
    x[j] = x[j-1]
  x[j] = t
```

Эта программа сдвигает элементы вправо до тех пор, пока они превосходят значение `t`, а потом ставит `t` в правильную позицию. Эта функция из пяти строк чуть сложнее своих предшественников, но на моем компьютере она работает примерно на 15% быстрее, чем вторая версия той же сортировки.

Для случайного расположения элементов во входном массиве, как и в худшем случае (обратный порядок сортировки), время выполнения сортировки вставкой пропорционально n^2 . Таблица 11.1 содержит данные о времени выполнения трех программ, когда на вход подается n случайных целых чисел.

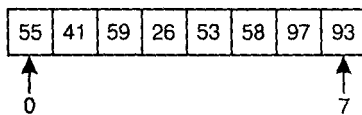
Таблица 11.1. Сортировка вставкой: время выполнения различных версий программы

Программа	Объем (строк на языке C)	Время, нс
Сортировка вставкой (1)	3	$11,9n^2$
Сортировка вставкой (2)	5	$3,8n^2$
Сортировка вставкой (3)	5	$3,2n^2$

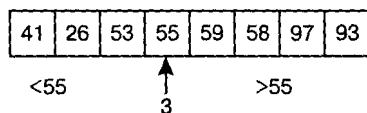
Третьей программе требуется несколько миллисекунд для сортировки $n = 1000$ целых чисел, треть секунды на $n = 10\,000$ целых, и почти час на сортировку миллиона чисел. Скоро мы встретимся с программой, сортирующей миллион чисел меньше, чем за секунду. Если входной массив уже почти отсортирован, сортировка вставкой работает гораздо быстрее, поскольку все элементы сдвигаются лишь на небольшое расстояние. Алгоритм в разделе 11.3 данной главы основан именно на этом свойстве.

11.2. Простая быстрая сортировка

Этот алгоритм был впервые описан К. А. Р. Хоаром в его классической статье «Быстрая сортировка» (C. A. R. Hoare, Quicksort. Computer Journal, 5, 1, April 1962, p. 10-15). В этом алгоритме используется подход «разделяй и властвуй», уже упоминавшийся в разделе 8.3: *чтобы отсортировать массив, мы разделяем его на два подмассива и сортируем каждый из них рекурсивно*. Например, для сортировки массива из восьми элементов

**Рис. 11.1.** Сортируемый массив

мы разбиваем его первым элементом (55), так чтобы все элементы, меньшие 55, расположились слева от него, а все большие — справа.

**Рис. 11.2.** Разбиение массива

Если затем рекурсивно отсортировать подмножества $x[0..2]$ и $x[4..7]$ по отдельности, весь массив окажется отсортирован.

Среднее время работы этого алгоритма оказывается существенно меньшим $O(n^2)$ сортировки вставкой, поскольку операция разбиения дает гораздо больше для достижения цели. После разбиения n элементов примерно половина из них окажется ниже выделенного элемента, а половина — выше. За то же самое время

в программе сортировки вставкой на свое место устанавливается один-единственный элемент.

Итак, у нас есть набросок рекурсивной функции. Опишем рабочую часть массива с помощью индексов l , u (нижняя и верхняя границы). Рекурсия останавливается, когда мы добираемся до массива с числом элементов, меньшим двух. Код программы приведен на листинге 11.3.

Листинг 11.3. Быстрая сортировка (набросок)

```
void qsort (l, u)
    if l >= u then
        /* не более одного элемента - ничего не делаем */
        return
    /* цель разбить массив относительно какого-либо элемента,
    который
    оказывается на правильном месте */
    qsort(l, p-1)
    qsort(p+1, u)
```

Для разбиения массива относительно значения t мы начнем с простой схемы, о которой я узнал от Нико Ломуто. Более быстрый вариант программы, выполняющий эту задачу, будет приведен в следующем разделе¹, но эта функция так проста, что в ней трудно ошибиться, и она ни в коем случае не может быть названа медленной. Пусть дано значение t , нужно упорядочить массив $x[a..b]$ и вычислить индекс m такой, что все элементы, меньшие t , находятся слева от элемента $x[m]$, а все большие — справа. Мы решим эту задачу с помощью простого цикла `for`, сканирующего массив слева направо, используя переменные i и m для сохранения инварианта (рис. 11.3).

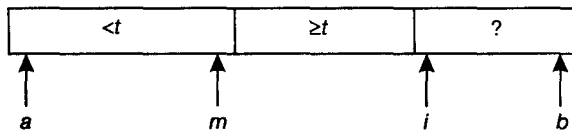


Рис. 11.3. Инвариант цикла сканирования в алгоритме быстрой сортировки

Когда проверяется i -й элемент, необходимо рассмотреть две ситуации. Если $x[i] \geq t$, то инвариант остается истинным. Если же $x[i] < t$, то восстановить инвариант можно, увеличив индекс m (который указывает новое местоположение меньшего элемента) и поменяв местами $x[i]$ и $x[m]$. В итоге получаем код, осуществляющий разбиение массива:

```
m = a-1
for i = [a, b]
    if x[i] < t
        swap(++m, i)
```

¹ В большинстве реализаций быстрой сортировки используется двустороннее разделение, описанное в следующем разделе. Хотя основная идея этого кода проста, детали реализации всегда казались мне весьма сложными — однажды я провел почти два дня в поисках ошибки в коротком цикле разбиения. Один из читателей черновика этой статьи пожаловался, что стандартный метод на самом деле проще, чем метод Ломуто, и в подтверждение этому набросал код программы. Я прекратил изучение этой программы, найдя две ошибки.

11.2. Простая быстрая сортировка 145

В алгоритме Quicksort нам нужно разбить массив $x[l..u]$ относительно значения $t=x[l]$, так что $a = l + 1$, а $b = u$. Инвариант цикла разбиения можно будет изобразить следующим образом (рис. 11.4).

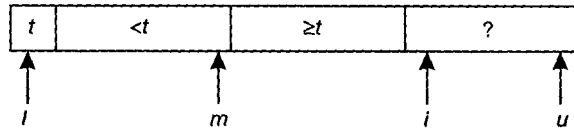


Рис. 11.4. Инвариант цикла разбиения массива

После завершения цикла получим картину, изображенную на рис. 11.5.

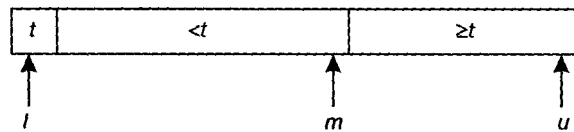


Рис. 11.5. Состояние массива после завершения цикла разбиения

Затем мы поменяем местами $x[l]$ и $x[m]$, что даст рис. 11.6¹.

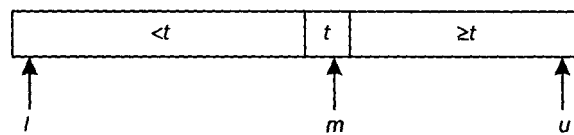


Рис. 11.6. Состояние после перестановки элементов

Теперь можно рекурсивно вызвать функцию с параметрами $(l, m-1)$ и $(t+1, u)$.

Получившийся код дает первую завершённую версию быстрой сортировки. Он приведен в листинге 11.4. Для сортировки массива $x[n]$ функцию следует вызвать как `qsort(0, n-1)`.

Листинг 11.4. Быстрая сортировка: версия 1

```
void qsort(l, u)
{
    if (l >= u)
        return;
    m = l;
    for (i = l+1; i <= u; i++)
        /* инвариант x[l+1..m] < x[l] &&
           x[m+1..i-1] >= x[l] */
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    /* x[l..m-1] < x[m] <= x[m+1..u] */
    qsort1(l, m-1);
    qsort1(m+1, u);
}
```

¹ Кажется соблазнительным пропустить этот этап и вызвать рекурсию с параметрами (l, t) и $(t+1, u)$. К сожалению, при этом мы попадаем в бесконечный цикл, если t является наибольшим элементом массива. Возможно, я бы обнаружил эту ошибку в процессе проверки условия завершения, но читатель может догадаться, как я в действительности на нее наткнулся. М. Джейкоб дал элегантное доказательство некорректности такой программы: поскольку $x[l]$ никогда не переставляется, эта сортировка будет работать только в том случае, если $x[0]$ будет минимальным элементом массива.

В задаче 2 в конце главы описана модификация алгоритма разбиения, предложенная Бобом Седжвиком (Bob Sedgewick). Она дает несколько более быстрый алгоритм `qsort2`.

Правильность этой программы была практически полностью доказана при ее выводе (как и должно быть). Доказательство ведется по индукции. Внешний оператор `if` правильно обрабатывает пустые и одноэлементные массивы, а алгоритм разбиения правильно подготавливает массивы большего размера для последующих рекурсивных вызовов. Программа не может войти в бесконечную последовательность рекурсивных вызовов, поскольку элемент `x[t]` при каждом таком вызове исключается. Таким же образом была доказана конечность алгоритма в разделе 4.3 (двоичный поиск).

Программа быстрой сортировки работает за время $O(n \log n)$ и требует в среднем $O(\log n)$ памяти на стеке. Под «средним» случаем подразумевается случайная перестановка не равных друг другу элементов, поступающая на вход алгоритма. Математические аргументы в пользу этого вывода аналогичны алгоритмам, приведенным в разделе 8.3. В большинстве учебников по теории алгоритмов подробно анализируется время выполнения быстрой сортировки. Кроме того, в них доказывается, что любая основанная на сравнении сортировка не может выполнить менее $O(n \log n)$ сравнений, поэтому быстрая сортировка наиболее близка к идеалу.

Функция `qsort1` — это самая простая среди известных мне версия алгоритма быстрой сортировки. Она иллюстрирует важнейшие свойства этого алгоритма. Во-первых, он действительно быстр: в моей системе миллион случайных целых чисел упорядочивается чуть больше, чем за секунду, — вдвое быстрее, чем хорошо отлаженная библиотечная функция `qsort` языка C (последняя предназначена для сортировки различных типов данных, поэтому оказывается более медленной). Эта сортировка может быть удобной для некоторых приложений с хорошими входными данными, но она обладает и другим характерным свойством быстрых сортировок: для некоторых типов входных данных время ее работы может быть квадратичным. В следующем разделе рассмотрены более робастные быстрые сортировки.

11.3. Улучшенные быстрые сортировки

Функция `qsort1` быстро сортирует массив случайных чисел, но что, если на вход будет подана уже упорядоченная последовательность? Как мы видели в разделе 2.4 главы 2, программисты часто используют сортировку для того, чтобы одинаковые элементы оказались рядом. Следовательно, нужно рассмотреть крайний случай: массив из n одинаковых элементов. Сортировка вставкой работает на таких данных очень быстро: каждый элемент сдвигается на 0 позиций, поэтому время выполнения растет как $O(n)$. Функция `qsort1` справляется с такими данными очень плохо. Каждое из $n-1$ разбиений требует время $O(n)$ для выделения одного эле-

11.3. Улучшенные быстрые сортировки 147

мента, поэтому полное время выполнения растет как $O(n^2)$. Время обработки для $n = 1\,000\,000$ возрастает с одной секунды до двух часов.

Можно обойти эту проблему, используя двусторонний алгоритм разбиения с приведенным на рис. 11.7 инвариантом.

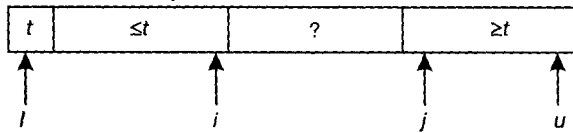


Рис. 11.7. Инвариант двустороннего разбиения

Индексы i и j инициализируются граничными индексами разбиваемого массива. Главный цикл содержит два вложенных цикла. Первый вложенный цикл сдвигает i вверх, пропуская меньшие элементы, а второй увеличивает j , пропуская большие элементы и останавливаясь на меньшем. Главный цикл проверяет, не пересекаются ли эти индексы, и переставляет соответствующие элементы.

Но как такой код будет работать в ситуации, когда все элементы равны? Первая мысль: пропустить эти элементы, чтобы не делать лишней работы, но в результате получается квадратичный для массива из одинаковых элементов алгоритм. Поэтому каждое сканирование будет останавливаться на одинаковых элементах, которые затем будут обмениваться. Хотя в этом варианте обменов будет производиться больше, чем требуется, такая программа будет превращать худший случай с массивом из одинаковых элементов в лучший, требующий почти в точности \log_2 и сравнений. Программа, реализующая описанный алгоритм разбиения, приведена в листинге 11.5.

Листинг 11.5. Быстрая сортировка (версия 3)

```
void qsort3(l, u)
    if l >= u
        return
    t = x[l], i = l, j = u+1
    loop
        do i++ while i <= u && x[i] < t
        do j-- while x[j] > t
        if i > j
            break
        swap(i, j)
    swap(l, j)
    qsort3(l, j-1)
    qsort3(j+1, u)
```

Избавляясь от квадратичного поведения в худшем случае, этот код и в среднем делает меньше обменов, чем `qsortl`.

Рассмотренные нами программы быстрой сортировки разбивали массив относительно первого встреченного элемента. Это хорошо подходит для случайных входных данных, но может сильно замедлить работу для некоторых упорядоченных последовательностей. Если массив уже отсортирован по возрастанию, его придется разбивать относительно первого элемента, затем относительно второго и так

далее, что потребует времени $O(n^2)$. Мы можем избежать этого, выбирая элемент для разбиения случайным образом — обменивая местами элемент $x[l]$ со случайным элементом из диапазона $x[l..u]$:

```
swap(l, randint(l..u))
```

Если у вас нет функции `randint`, обратитесь к задаче 2 из главы 12 данной книги, которая посвящена написанию собственного генератора случайных чисел. Каким бы кодом вы ни пользовались, внимательно проследите за тем, чтобы функция `randint` возвращала значение из диапазона $[l, u]$ — выход за его границы приведет к ошибкам. Объединив случайный выбор центрального элемента с двусторонним разбиением, мы получим программу быстрой сортировки, работающую за время $O(n \log n)$ для любого входного массива. Усреднение делается вызовом генератора случайных чисел, а не анализом возможного распределения входных данных.

Наша программа быстрой сортировки большую часть времени тратит на сортировку очень маленьких подмножеств. Такие массивы было бы проще всего сортировать каким-либо несложным методом вроде сортировки вставкой, а не тратить на них всю мощь быстрой сортировки. Боб Седжвик разработал весьма хитроумную реализацию этой идеи. Когда функция быстрой сортировки вызывается для небольшого массива (то есть l и u близки), она не делает ничего. Реализуется это путем замены первого оператора `if` нашей функции на следующий код:

```
if u-l > cutoff
    return
```

Здесь `cutoff` — некоторое небольшое целое число. После завершения работы функции массив будет не отсортирован до конца, но разбит на небольшие группы случайно упорядоченных элементов, причем все элементы одной группы будут меньше любого элемента всех групп, расположенных справа от данной. Сортировать элементы внутри групп нужно каким-то другим методом, и тут лучше всего подходит сортировка вставкой, поскольку массив уже почти упорядочен.

Для решения задачи сортировки целиком придется выполнить два вызова:

```
qsort4(0, n-1)
isort3()
```

Задача 3 посвящена выбору оптимальной величины порога `cutoff`.

На последнем этапе оптимизации программы можно раскрыть вызов функции `swap` во внутреннем цикле (поскольку другие два вызова `swap` лежат вне внутреннего цикла, их раскрытие не даст ощутимого результата). Последняя версия программы Quicksort приведена в листинге 11.6.

Листинг 11.6. Быстрая сортировка: версия 4 (итоговая)

```
void qsort4(l, u)
{
    if u - l < cutoff
        return
    swap(l, randint(l, u))
    t = x[l]; i = l; j = u+1
    loop
```



```

do i++ while i <= u && x[i] < t
do j-- while x[j] > t
if i > j
    break
temp = x[i] x[i] = x[j], x[j] = temp
swap(i j)
qsort4(l, j-1)
qsort4(j+1 u)

```

Задачи 4 и 11 посвящены дальнейшему улучшению производительности быстрой сортировки. В табл. 11.2 приведены сводные данные по всем версиям быстрой сортировки. Правая колонка указывает среднее время работы в наносекундах, требуемое для сортировки массива из n случайных целых чисел. Многие алгоритмы могут вести себя как квадратичные для некоторых конкретных входных данных.

Таблица 11.2. Быстрая сортировка

Программа	Объем кода (строк языка C)	Время, нс
Библиотечная функция языка C qsort	3	$137n \log_2 n$
Быстрая сортировка 1	9	$60n \log_2 n$
Быстрая сортировка 2	9	$56n \log_2 n$
Быстрая сортировка 3	14	$44n \log_2 n$
Быстрая сортировка 4	15+5	$36n \log_2 n$
Библиотечная функция C++ sort	1	$30n \log_2 n$

Функция qsort состоит из 15 строк быстрой сортировки и 5 строк сортировки вставкой. Для миллиона случайных чисел время выполнения лежит в диапазоне от 0,6 с для библиотечной функции sort языка C++ до 2,7 с для библиотечной функции qsort языка C. В главе 14 мы изучим алгоритм, гарантированно сортирующий n целых чисел за $O(n \log n)$ для любых входных данных (даже в худшем случае).

11.4. Принципы

Это упражнение дает нам несколько ценных уроков о сортировке в частности и программировании вообще.

Библиотечная функция qsort

Библиотечная функция qsort проста в использовании и работает достаточно быстро. Она работает медленнее, чем самодельные версии быстрой сортировки, только потому, что предназначена для применения к различным типам данных, поэтому сравнение элементов осуществляется через вызов внешней функции. Интерфейс функции sort языка C++ существенно проще: массив x сортируется вызовом `sort(x, x+n)`. Кроме того, она достаточно эффективно реализована. Если системная

сортировка отвечает вашим требованиям, не задумывайтесь и отметайте идею написания своей собственной.

Сортировка вставкой

Сортировку вставкой легко написать, и она достаточно эффективна для небольших задач. Сортировка 10 000 целых чисел с помощью этого алгоритма требует на моем компьютере всего лишь треть секунды.

Случай больших n

Для больших n жизненно важным оказывается время выполнения алгоритма быстрой сортировки $O(n \log n)$. Методы разработки алгоритмов из главы 8 дали нам идею (принцип «разделяй и властвуй»), а верификация, обсуждавшаяся в главе 4, позволила нам реализовать эту идею в ясном и эффективном коде.

Хотя значительное увеличение производительности обычно достигается сменой алгоритма, методы оптимизации из главы 9 ускорили работу сортировки вставкой в 4 раза, а быстрой сортировки — в 2 раза.

11.5. Задачи

1. Сортировка, как и любое другое мощное средство, часто используется там, где ее не следует использовать, и не используется там, где без нее не обойтись. Объясните, как можно недооценить или переоценить важность сортировки при получении статистических данных (минимум, максимум, среднее, медиана и мода распределения) для некоторого массива вещественных чисел.
2. [R. Sedgwick] Ускорьте схему разбиения Ломуто (Lomuto), используя элемент $x[1]$ в качестве маркера. Покажите, как эта функция позволяет избавиться от вызова `swap` после цикла.
3. Какими экспериментами можно найти оптимальное значение `cutoff` в некоторой системе?
4. Хотя в среднем быстрая сортировка требует $O(\log n)$ памяти на стеке, в худшем случае может потребоваться объем памяти $O(n)$. Объясните, откуда проистекает этот недостаток, и постарайтесь избавиться от него.
5. [M. D. McIlroy] Покажите, как можно использовать схему разбиения Лому-то для сортировки битовых строк переменной длины за время, пропорциональное сумме их длин.
6. Используйте методы, описанные в этой главе, для реализации других алгоритмов сортировки. Сортировка выбором помещает наименьший элемент в $x[0]$, затем наименьший из оставшихся в $x[1]$ и так далее. Сортировка Шелла (с уменьшающимся инкрементом) работает аналогично сортировке вставкой, но элементы сдвигаются на h позиций, а не на одну. Начальное значение h затем уменьшается в процессе работы.

11.6. Дополнительная литература 151

7. Реализации программ сортировки из этой главы можно *найти* на веб-сайте книги. Измерьте время их работы в вашей системе и сведите результаты в таблицу, аналогичную табл. 11.2.
8. Набросайте руководство пользователю вашей системы по выбору метода сортировки. Метод должен учитывать важность времени выполнения, требуемой памяти, времени программиста (затрачиваемого на разработку и обслуживание программы), общность (что, если нужно отсортировать символьные строки, в которых записаны римские числа?), стабильность (элементы с одинаковыми ключами должны оставаться в исходном порядке), особенности входных данных и другие. В качестве теста можете испробовать это руководство на задаче из главы 1.
9. Напишите программу нахождения k -го минимума в массиве $x[0..n-1]$ за время $O(n)$. Допускается перестановка элементов.
10. Проведите эксперименты и постройте диаграмму времени работы быстрой сортировки на различных входных данных.
11. Напишите функцию разбиения с «широким главным элементом», постусловие которой будет таким:

$<t$	$=t$	$>t$
------	------	------

Как можно включить такую функцию в программу быстрой сортировки?

12. Изучите методы сортировки, используемые в обычной жизни (сортировка почты, разменные автоматы и другие).
13. Программы быстрой сортировки из этой главы выбирают центральный элемент случайным образом. Изучите возможные варианты, включая выбор медианы по некоторому подмножеству исходного массива.
14. Программы быстрой сортировки, рассматриваемые в данной главе, описывают подмножество с помощью двух целочисленных индексов. Это необходимо в языках типа Java, в которых указатели на элементы массивов отсутствуют. В языках C и C++ можно отсортировать массив целых чисел, используя приведенный ниже формат вызова функции:
`void qsort(int x[], int n)`
для исходного и всех последующих рекурсивных вызовов. Измените алгоритмы этой главы, ориентируясь на использование данного интерфейса.

11.6. Дополнительная литература

С момента выхода первого издания этой книги Addison Wesley в 1973 году, самым популярным справочником на эту тему стала книга Д. Кнута «Искусство программирования для ЭВМ, том 3: сортировка и поиск». В ней подробно описываются все основные алгоритмы, проводится математический анализ времени их работы. В книге также приведены примеры реализации алгоритмов на ассемблере. Пред-

лагаемые упражнения и ссылки на другие книги содержат важные модификации исходных алгоритмов. К моменту выхода второго издания в 1998 году Кнут обновил и отредактировал книгу. Используемый им ассемблер MIX устарел, но принципы разработки, иллюстрируемые кодом, бессмертны.

Третье издание книги Боба Седжвика «Алгоритмы» (Bob Sedgwick, Algorithms) содержит описание более современных подходов к сортировке и поиску. Части 1-4 посвящены «Основам», «Структурам данных», «Сортировке» и «Поиску». Книга «Алгоритмы на С» была опубликована издательством Addison Wesley в 1997 году, «Алгоритмы на С++» (в соавторстве с Крисом ван Вайком) — в 1998 году, а «Алгоритмы на Java» (в соавторстве с Тимом Линдхолмом) вышла в 1999 году. Упор делается на реализацию алгоритмов на различных языках программирования (по вашему выбору), тогда как описание их производительности дается на интуитивном уровне.

Книги этих двух авторов являются основными руководствами по сортировке, поиску (глава 13) и «кучам» (глава 14).