

3.4. СОРТДЕРЕВОМ — УПОРЯДОЧЕНИЕ С ПОМОЩЬЮ $O(n \log n)$ СРАВНЕНИЙ

Так как любой сортирующий алгоритм, упорядочивающий с помощью сравнений, затрачивает по необходимости $n \log n$ сравнений для упорядочения хотя бы одной последовательности длины n , естественно спросить, существуют ли сортирующие алгоритмы, затрачивающие не более $O(n \log n)$ сравнений для упорядочения любой последовательности длины n . Один такой алгоритм мы уже видели — это сортировка слиянием из разд. 2.7. Другой алгоритм — Сортдеревом. Помимо того, что это полезный алгоритм упорядочения, в нем используется интересная структура данных, которая находит и другие приложения.

Сортдеревом лучше всего понять в терминах двоичного дерева вроде изображенного на рис. 3.5, у которого каждый лист имеет глубину d или $d-1$. Узлы дерева помечаются элементами последовательности, которую хотят упорядочить. Затем Сортдеревом меняет размещение этих элементов на дереве до тех пор, пока элемент, соответствующий произвольному узлу, станет не меньше элементов, соответствующих его сыновьям. Такое помеченное дерево мы будем называть *сортирующим*.

Пример 3.3. На рис. 3.5 изображено сортирующее дерево. Заметим, что последовательность элементов, лежащих на пути из любого листа в корень, линейно упорядочена и наибольший элемент в поддереве всегда соответствует его корню. \square

На следующем шаге алгоритма Сортдеревом из сортирующего дерева удаляется наибольший элемент — он соответствует корню дерева. Метка некоторого листа переносится в корень, а сам лист удаляется. Затем полученное дерево переделывается в сортирующее, и процесс повторяется. Последовательность элементов, удаленных из сортирующего дерева, упорядочена по невозрастанию.

Удобной структурой данных для сортирующего дерева служит такой массив A , что $A[1]$ — элемент в корне, а $A[2i]$ и $A[2i+1]$ —

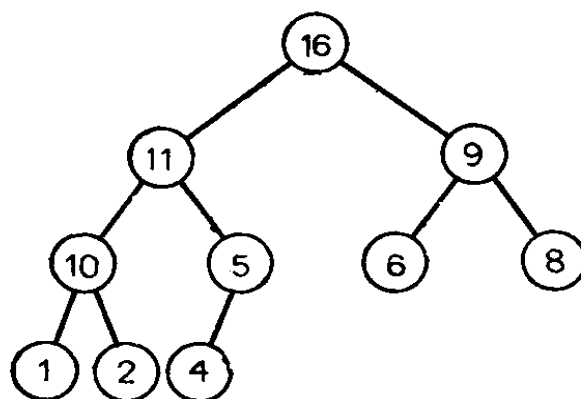


Рис. 3.5. Сортирующее дерево.

элементы в левом и правом сыновьях (если они существуют) того узла, в котором хранится $A[i]$.

Например, сортирующее дерево на рис. 3.5 можно представить массивом

16	11	9	10	5	6	8	1	2	4
----	----	---	----	---	---	---	---	---	---

Заметим, что узлы наименьшей глубины стоят в этом массиве первыми, а узлы равной глубины стоят в порядке слева направо.

Не каждое сортирующее дерево можно представить таким способом. На языке представления деревьев можно сказать, что для образования такого массива требуется, чтобы листья самого низкого уровня стояли как можно левее (как, например, на рис. 3.5).

Если сортирующее дерево представляется описанным массивом, то некоторые операции из алгоритма Сортдеревом легко выполнить. Например, согласно алгоритму, нужно удалить элемент из корня, где-то запомнить его, переделать оставшееся дерево в сортирующее и удалить непомятый лист. Можно удалить наибольший элемент из сортирующего дерева и запомнить его, поменяв местами $A[1]$ и $A[n]$, и затем не считать более ячейку n нашего массива частью сортирующего дерева. Ячейка n рассматривается как лист, удаленный из этого дерева. Для того чтобы переделать дерево, хранящееся в ячейках $1, 2, \dots, n-1$, в сортирующее, надо взять новый элемент $A[1]$ и провести его вдоль подходящего пути в дереве. Затем можно повторить процесс, меняя местами $A[1]$ и $A[n-1]$ и считая, что дерево занимает ячейки $1, 2, \dots, n-2$ и т. д.

Пример 3.4. Рассмотрим на примере сортирующего дерева рис. 3.5, что происходит, когда мы поменяем местами первый и последний элементы массива, представляющего это дерево. Новый массив

4	11	9	10	5	6	8	1	2	16
---	----	---	----	---	---	---	---	---	----

соответствует помеченному дереву на рис. 3.6,а. Элемент 16 исключается из дальнейшего рассмотрения. Чтобы превратить полученное дерево в сортирующее, надо поменять местами элемент 4 с большим из его сыновей, т. е. с элементом 11.

В своем новом положении элемент 4 обладает сыновьями 10 и 5. Так как они больше 4, то 4 переставляется с 10, большим сыном. После этого сыновьями элемента 4 в новом положении становятся 1 и 2. Поскольку 4 превосходит их обоих, дальнейшие перестановки не нужны.

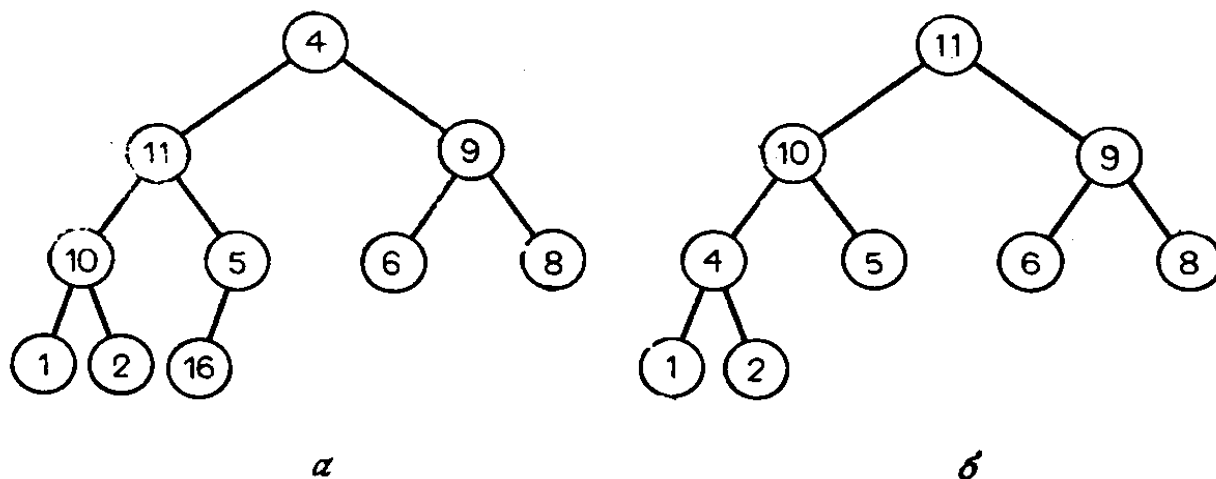


Рис. 3.6. *a* — результат перестановки элементов 4 и 16 в сортирующем дереве рис. 3.5; *б* — результат перестройки сортирующего дерева и удаления элемента 16.

Полученное в результате сортирующее дерево показано на рис. 3.6, *б*. Заметим, что хотя элемент 16 был удален из сортирующего дерева, он все же присутствует в конце массива A . \square

Теперь перейдем к формальному описанию алгоритма Сорт-деревом.

Пусть a_1, a_2, \dots, a_n — последовательность сортируемых элементов. Предположим, что вначале они помещаются в массив A именно в этом порядке, т. е. $A[i] = a_i$, $1 \leq i \leq n$. Первый шаг состоит в построении сортирующего дерева, т. е. элементы в A перераспределяются так, чтобы удовлетворялось *свойство сортирующего дерева*: $A[i] \geq A[2i]$ при $1 \leq i \leq n/2$ и $A[i] \geq A[2i+1]$ при $1 \leq i < n/2$. Это делают, строя, начиная с листьев, все большие и большие сортирующие деревья. Всякое поддереву, состоящее из листа, уже является сортирующим. Чтобы сделать поддерево высоты h сортирующим, надо переставить элемент в корне с наибольшим из элементов, соответствующих его сыновьям, если, конечно, он меньше какого-то из них. Такое действие может испортить сортирующее дерево высоты $h-1$, и тогда его надо снова перестроить в сортирующее. Приведем точное описание этого алгоритма.

Алгоритм 3.3. Построение сортирующего дерева

Вход. Массив элементов $A[i]$, $1 \leq i \leq n$.

Выход. Элементы массива A , организованные в виде сортирующего дерева, т. е. $A[i] \leq A[\lfloor n/2 \rfloor]$ для $1 < i \leq n$.

Метод. В основе алгоритма лежит рекурсивная процедура ПЕРЕСЫПКА. Ее параметры i и j задают область ячеек массива A , обладающую свойством сортирующего дерева; корень строящегося дерева помещается в i .

procedure ПЕРЕСЫПКА (i, j):

1. **if** i — не лист и какой-то его сын содержит элемент, превосходящий элемент в i **then**
 begin
2. пусть k — тот сын узла i , в котором хранится
 наибольший элемент;
3. переставить $A[i]$ и $A[k]$;
4. ПЕРЕСЫПКА (k, j)
- end**

Параметр j используется, чтобы определить, является ли i листом и имеет он одного или двух сыновей. Если $i > j/2$, то i — лист, и процедуре ПЕРЕСЫПКА (i, j) ничего не нужно делать, поскольку $A[i]$ — уже сортирующее дерево.

Алгоритм, превращающий весь массив A в сортирующее дерево, выглядит просто:

procedure ПОСТРСОРТДЕРЕВА:

for $i \leftarrow n^1$ **step** -1 **until** 1 **do** ПЕРЕСЫПКА (i, n) □

Покажем, что алгоритм 3.3 преобразует A в сортирующее дерево за линейное время.

Лемма 3.2. Если узлы $i+1, i+2, \dots, n$ являются корнями сортирующих деревьев, то после вызова процедуры ПЕРЕСЫПКА (i, n) все узлы $i, i+1, \dots, n$ будут корнями сортирующих деревьев.

Доказательство. Доказательство проводится возвратной индукцией по i .

Базис, т. е. случай $i=n$, тривиален, так как узел n должен быть листом и условие, проверяемое в строке 1, гарантирует, что ПЕРЕСЫПКА (n, n) ничего не делает.

Для шага индукции заметим, что если i — лист или у него нет сына с большим элементом, то доказывать нечего (как и при обосновании базиса). Но если у узла i есть один сын (т. е. если $2i=n$) и $A[i] < A[2i]$, то строка 3 процедуры ПЕРЕСЫПКА переставляет $A[i]$ и $A[2i]$. В строке 4 вызывается ПЕРЕСЫПКА ($2i, n$); поэтому из предположения индукции вытекает, что дерево с корнем $2i$ будет переделано в сортирующее. Что касается узлов $i+1, i+2, \dots, 2i-1$, то они и не переставали быть корнями сортирующих деревьев. Так как после этой новой перестановки в массиве A выполняется неравенство $A[i] > A[2i]$, то дерево с корнем i также оказывается сортирующим.

Аналогично, если узел i имеет двух сыновей (т. е. если $2i+1 \leq n$) и наибольший из элементов в $A[2i]$ и в $A[2i+1]$ больше элемента

¹⁾ На практике мы бы начали с $\lfloor n/2 \rfloor$.

в $A[i]$, то, рассуждая, как и выше, можно показать, что после вызова процедуры ПЕРЕСЫПКА(i, n) все узлы $i, i+1, \dots, n$ будут корнями сортирующих деревьев. \square

Теорема 3.5. Алгоритм 3.3 преобразует A в сортирующее дерево за линейное время.

Доказательство. Применяя лемму 3.2, можно с помощью простой возвратной индукции по i показать, что узел i становится корнем какого-нибудь сортирующего дерева для всех $i, 1 \leq i \leq n$.

Пусть $T(h)$ — время выполнения процедуры ПЕРЕСЫПКА на узле высоты h . Тогда $T(h) \leq T(h-1) + c$ для некоторой постоянной c . Отсюда вытекает, что $T(h)$ есть $O(h)$.

Алгоритм 3.3 вызывает процедуру ПЕРЕСЫПКА, если не считать рекурсивных вызовов, один раз для каждого узла. Поэтому время, затрачиваемое на ПОСТРСОРТДЕРЕВА, имеет тот же порядок, что и сумма высот всех узлов. Но узлов высоты i не больше, чем $\lceil n/2^{i+1} \rceil$. Следовательно, общее время, затрачиваемое процедурой ПОСТРСОРТДЕРЕВА, имеет порядок $\sum_{i=1}^n i n/2^i$, т. е. $O(n)$. \square

Теперь можно завершить детальное описание алгоритма СОРТДЕРЕВОМ. Коль скоро элементы массива A преобразованы в сортирующее дерево, некоторые элементы удаляются из корня по одному за раз. Это делается перестановкой $A[1]$ и $A[n]$ и последующим преобразованием $A[1], A[2], \dots, A[n-1]$ в сортирующее дерево. Затем переставляются $A[1]$ и $A[n-1]$, а $A[1], A[2], \dots, A[n-2]$ преобразуется в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда $A[1], A[2], \dots, A[n]$ — упорядоченная последовательность.

Алгоритм 3.4. Сортдеревом

Вход. Массив элементов $A[i], 1 \leq i \leq n$.

Выход. Элементы массива A , расположенные в порядке неубывания.

Метод. Применяется процедура ПОСТРСОРТДЕРЕВА, т. е. алгоритм 3.3. Наш алгоритм выглядит так:

```
begin
    ПОСТРСОРТДЕРЕВА;
    for  $i \leftarrow n$  step  $-1$  until 2 do
        begin
            переставить  $A[1]$  и  $A[i]$ ;
            ПЕРЕСЫПКА(1,  $i-1$ )
        end
    end
```

\square

Теорема 3.6. *Алгоритм 3.4 упорядочивает последовательность из n элементов за время $O(n \log n)$.*

Доказательство. Корректность алгоритма доказывается индукцией по числу выполнений основного цикла, которое обозначим через m . Предположение индукции состоит в том, что после m итераций список $A[n-m+1], \dots, A[n]$ содержит m наибольших элементов, расположенных в порядке неубывания, а список $A[1], \dots, A[n-m]$ образует сортирующее дерево. Детали доказательства оставляем в качестве упражнения. Время выполнения процедуры ПЕРЕСЫПКА(1, i) есть $O(\log i)$. Следовательно, алгоритм 3.4 имеет сложность $O(n \log n)$. \square

Следствие. *Алгоритм Сортдерево имеет временную сложность $O_C(n \log n)$.*

3.5. БЫСТРСОРТ — УПОРЯДОЧЕНИЕ ЗА СРЕДНЕЕ ВРЕМЯ $O(n \log n)$

До сих пор мы рассматривали поведение сортирующих алгоритмов только в худшем случае. Для многих приложений более реалистичной мерой временной сложности является усредненное время работы. В случае сортировки с помощью деревьев решений мы видим, что никакой сортирующий алгоритм не может иметь среднюю временную сложность, существенно меньшую $n \log n$. Однако известны алгоритмы сортировки, которые работают в худшем случае cn^2 времени, где c — некоторая постоянная, но среднее время работы которых относит их к лучшим алгоритмам сортировки. Примером такого алгоритма служит алгоритм Быстрсорт, рассматриваемый в этом разделе.

Чтобы можно было рассуждать о среднем времени работы алгоритма, мы должны договориться о вероятностном распределении входов. Для сортировки естественно допустить, что мы и сделаем, что любая перестановка упорядочиваемой последовательности равновероятна в качестве входа. При таком допущении уже можно оценить снизу среднее число сравнений, необходимых для упорядочения последовательности из n элементов.

Общий метод состоит в том, чтобы поставить в соответствие каждому листу v дерева решений вероятность быть достигнутым при данном входе. Зная распределение вероятностей на входах, можно найти вероятности, поставленные в соответствие листьям. Таким образом, можно определить среднее число сравнений, производимых данным алгоритмом сортировки, если вычислить сумму $\sum_i p_i d_i$,

взятую по всем листьям дерева решений данного алгоритма, в которой p_i — вероятность достижения i -го листа, а d_i — его глубина. Это число называется *средней глубиной* дерева решений. Итак, мы пришли к следующему обобщению теоремы 3.4.

Теорема 3.7. В предположении, что все перестановки n -элементной последовательности появляются на входе с равными вероятностями, любое дерево решений, упорядочивающее последовательность из n элементов, имеет среднюю глубину не менее $\log n!$.

Доказательство. Обозначим через $D(T)$ сумму глубин листьев двоичного дерева T . Пусть $D(T)$ — ее наименьшее значение, взятое по всем двоичным деревьям T с m листьями. Покажем индукцией по m , что $D(m) \geq m \log m$.

Базис, т. е. случай $m=1$, тривиален. Допустим, что предположение индукции верно для всех значений m , меньших k . Рассмотрим дерево решений T с k листьями. Оно состоит из корня с левым поддеревом T_i с i листьями и правым поддеревом T_{k-i} с $k-i$ листьями при некотором i , $1 \leq i < k$. Ясно, что

$$D(T) = i + D(T_i) + (k-i) + D(T_{k-i}).$$

Поэтому наименьшее значение $D(T)$ дается равенством

$$D(k) = \min_{1 \leq i < k} [k + D(i) + D(k-i)]. \quad (3.1)$$

Учитывая предположение индукции, получаем отсюда

$$D(k) \geq k + \min_{1 \leq i < k} [i \log i + (k-i) \log (k-i)]. \quad (3.2)$$

Легко показать, что этот минимум достигается при $i=k/2$. Следовательно,

$$D(k) \geq k + k \log \frac{k}{2} = k \log k.$$

Таким образом, $D(m) \geq m \log m$ для всех $m \geq 1$.

Теперь мы утверждаем, что дерево решений T , упорядочивающее n случайных элементов, имеет не меньше $n!$ листьев. Более того, в точности $n!$ листьев появляются с вероятностью $1/n!$ каждый, а остальные — с вероятностью 0. Не изменяя средней глубины дерева T можно удалить из него все узлы, которые являются предками только листьев вероятности 0. Тогда останется дерево T' с $n!$ листьями, каждый из которых достигается с вероятностью $1/n!$. Так как $D(T') \geq n! \log n!$, то средняя глубина дерева T' (а значит, и T) не меньше $(1/n!) n! \log n! = \log n!$. \square

Следствие. Любая сортировка с помощью сравнений выполняет в среднем не менее $cn \log n$ сравнений для некоторой постоянной $c > 0$.

Заслуживает упоминания эффективный алгоритм, называемый Быстрсорт, поскольку среднее время его работы, хотя и ограничено снизу функцией $cn \log n$ для некоторой постоянной c (как и у всякой сортировки с помощью сравнений), но составляет лишь часть

```

procedure БЫСТРСОРТ( $S$ ):
1. if  $S$  содержит не больше одного элемента then return  $S$ 
   else
     begin
2.   выбрать произвольный элемент  $a$  из  $S$ ;
3.   пусть  $S_1$ ,  $S_2$  и  $S_3$  — последовательности элементов из  $S$ ,
       соответственно меньших, равных и больших  $a$ ;
4.   return (БЫСТРСОРТ( $S_1$ ), затем  $S_2$ , затем БЫСТРСОРТ( $S_3$ ))
     end

```

Рис. 3.7. Программа Быстрсорт.

времени работы других известных алгоритмов при их реализации на большинстве существующих машин. В худшем случае Быстрсорт имеет квадратичное время работы, но для многих приложений это не существенно.

Алгоритм 3.5. Быстрсорт

Вход. Последовательность S из n элементов a_1, a_2, \dots, a_n .

Выход. Элементы последовательности S , расположенные по порядку.

Метод. Рекурсивная процедура БЫСТРСОРТ определяется на рис. 3.7. Алгоритм состоит в вызове БЫСТРСОРТ(S). \square

Теорема 3.8. Алгоритм 3.5. упорядочивает последовательность из n элементов за среднее время $O(n \log n)$.

Доказательство. Корректность алгоритма 3.5 доказывается прямой индукцией по длине последовательности S . Чтобы проще было анализировать время работы, допустим, что все элементы в S различны. Это допущение максимизирует размеры последовательностей S_1 и S_3 , которые строятся в строке 3, и тем самым максимизирует среднее время, затрачиваемое в рекурсивных вызовах в строке 4. Пусть $T(n)$ — среднее время, затрачиваемое алгоритмом БЫСТРСОРТ на упорядочение последовательности из n элементов. Ясно, что $T(0) = T(1) = b$ для некоторой постоянной b .

Допустим, что элемент a , выбираемый в строке 2, является i -м наименьшим элементом среди n элементов последовательности S . Тогда на два рекурсивных вызова БЫСТРСОРТ в строке 4 тратится среднее время $T(i-1)$ и $T(n-i)$ соответственно. Так как i равновероятно принимает любое значение между 1 и n , а итоговое построение последовательности БЫСТРСОРТ(S) очевидно занимает

время cn для некоторой постоянной c , то

$$T(n) \leq cn + \frac{1}{n} \sum_{i=1}^n [T(i-1) + T(n-i)] \text{ для } n \geq 2. \quad (3.3)$$

Алгебраические преобразования в (3.3) приводят к неравенству

$$T(n) \leq cn + \frac{2}{n} \sum_{i=0}^{n-1} T(i). \quad (3.4)$$

Покажем, что при $n \geq 2$ справедливо неравенство $T(n) \leq kn \ln n$, где $k=2c+2b$ и $b=T(0)=T(1)$. Для базиса ($n=2$) неравенство $T(2) \leq 2c+2b$ непосредственно вытекает из (3.4). Для проведения шага индукции запишем (3.4) в виде

$$T(n) \leq cn + \frac{4b}{n} + \frac{2}{n} \sum_{i=2}^{n-1} ki \ln i. \quad (3.5)$$

Так как функция $i \ln i$ вогнута, легко показать, что

$$\sum_{i=2}^{n-1} i \ln i \leq \int_2^n x \ln x dx \leq \frac{n^2 \ln n}{2} - \frac{n^2}{4}. \quad (3.6)$$

Подставляя (3.6) в (3.5), получаем

$$T(n) \leq cn + \frac{4b}{n} + kn \ln n - \frac{kn}{2}. \quad (3.7)$$

Поскольку $n \geq 2$ и $k=2c+2b$, то $cn + 4b/n \leq kn/2$. Таким образом, неравенство $T(n) \leq kn \ln n$ следует из (3.7). \square

Рассмотрим две детали, важные для практической реализации алгоритма. Первая — способ “произвольного” выбора элемента a в строке 2 процедуры БЫСТРСОРТ. При реализации этого оператора может возникнуть искушение встать на простой путь, а именно всегда выбирать, скажем, первый элемент последовательности S . Подобный выбор мог бы оказаться причиной значительно худшей работы алгоритма БЫСТРСОРТ, чем это вытекает из (3.3). Последовательность, к которой применяется подпрограмма сортировки, часто бывает уже “как-то” рассортирована, так что первый элемент мал с вероятностью выше средней. Читатель может проверить, что в крайнем случае, когда БЫСТРСОРТ начинает работу на уже упорядоченной последовательности без повторений, а в качестве элемента a всегда выбирается первый элемент из S , в последовательности S всегда будет на один элемент меньше, чем в той, из которой она строится. В этом случае БЫСТРСОРТ требует квадратичного числа шагов.

Лучшей техникой для выбора разбивающего элемента в строке 2 было бы использование генератора случайных чисел для порождения целого числа i , $1 \leq i \leq |S|$ ¹⁾, и выбора затем i -го элемента из S в качестве a . Более простой подход — произвести выборку элементов из S , а затем взять ее медиану в качестве разбивающего элемента. Например, в качестве разбивающего элемента можно было бы взять медиану выборки, состоящей из первого, среднего и последнего элементов последовательности S .

Вторая деталь — как эффективно разбить S на три последовательности S_1 , S_2 и S_3 ? Можно (и желательно) иметь в массиве A все n исходных элементов. Так как процедура БЫСТРСОРТ вызывает себя рекурсивно, ее аргумент S всегда будет находиться в последовательных компонентах массива, скажем $A[f]$, $A[f+1]$, ..., $A[l]$ для некоторых f и l , $1 \leq f \leq l \leq n$. Выбрав “произвольный” элемент a , можно устроить разбиение последовательности S на этом же месте. Иными словами, можно расположить S_1 в компонентах $A[f]$, $A[f+1]$, ..., $A[k]$, а $S_2 \cup S_3$ — в $A[k+1]$, $A[k+2]$, ..., $A[l]$ при некотором k , $f \leq k \leq l$. Затем можно, если нужно, расщепить $S_2 \cup S_3$, но обычно эффективнее просто рекурсивно вызвать БЫСТРСОРТ на S_1 и $S_2 \cup S_3$, если оба этих множества не пусты.

По-видимому, самый легкий способ разбить S на том же месте — это использовать два указателя на массив, назовем их i и j . Вначале

```

begin
1.   $i \leftarrow f$ ;
2.   $j \leftarrow l$ ;
3.  while  $i \leq j$  do
      begin
4.      while  $A[j] \geq a$  и  $j \geq f$  do  $j \leftarrow j - 1$ ;
5.      while  $A[j] < a$  и  $i \leq l$  do  $i \leftarrow i + 1$ ;
6.      if  $i < j$  then
          begin
7.          переставить  $A[i]$  и  $A[j]$ ;
8.           $i \leftarrow i + 1$ ;
9.           $j \leftarrow j - 1$ 
          end
      end
      end
end
end

```

Рис. 3.8. Разбиение S на S_1 и $S_2 \cup S_3$ на месте их расположения.

¹⁾ Через $|S|$ обозначена длина последовательности S .

$i=f$, и все время в $A[f], \dots, A[i-1]$ будут находиться элементы из S_1 . Аналогично вначале $j=l$, а в $A[j+1], \dots, A[l]$ все время будут находиться элементы из $S_2 \cup S_3$. Это разбиение производит подпрограмма на рис. 3.8.

Затем можно вызвать БЫСТРСОРТ для массива $A[f], \dots, A[i-1]$, т. е. S_1 , и для массива $A[j+1], \dots, A[l]$, т. е. $S_2 \cup S_3$. Но если $i=f$ (и тогда $S_1 = \emptyset$), то надо сначала удалить из $S_2 \cup S_3$ хотя бы один элемент, равный a . Удобно удалять тот элемент, по которому производилось разбиение. Следует также заметить, что если это представление в виде массива применяется для последовательностей, то можно подать аргументы для БЫСТРСОРТ, просто поставив указатели на первую и последнюю ячейку используемого куска массива.

Пример 3.5. Разобьем массив A

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

по элементу $a=3$. **while**-оператор (строка 4) уменьшает j с 9 до 7, поскольку числа $A[9]=3$ и $A[8]=8$ оба не меньше a , но $A[7]=1 < a$. Строка 5 не увеличивает i с его начального значения 1, поскольку $A[1]=6 \geq a$. Поэтому мы переставляем $A[1]$ и $A[7]$, полагаем $i=2$, $j=6$ и получаем массив на рис. 3.9, а. Результаты, получаемые после следующих двух срабатываний цикла в строках 3—9, показаны на рис. 3.9, б и в. В этот момент $i > j$, и выполнение **while**-оператора, стоящего в строке 3, заканчивается. \square

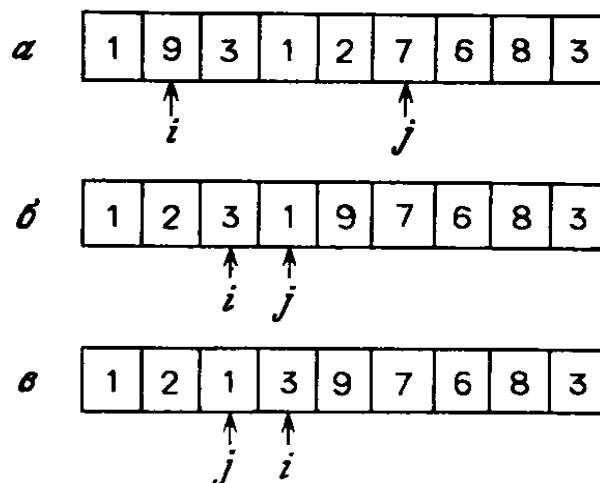


Рис. 3.9. Разбиение массива.