

## 2. СОРТИРОВКА

### 2.1. ВВЕДЕНИЕ

Основное назначение данной главы — дать побольше примеров использования структур данных, введенных в предыдущей главе, и продемонстрировать, насколько глубоко влияет выбор той или иной структуры на алгоритмы, решающие поставленную задачу. Сортировка к тому же еще и сама достаточно хороший пример задачи, которую можно решать с помощью многих различных алгоритмов. Каждый из них имеет и свои достоинства, и свои недостатки, и выбирать алгоритмы нужно, исходя из конкретной постановки задачи.

В общем сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки — облегчить последующий поиск элементов в таком отсортированном множестве. Это почти универсальная, фундаментальная деятельность. Мы встречаемся с отсортированными объектами в телефонных книгах, в списках подходящих налогов, в оглавлениях книг, в библиотеках, в словарях, на складах — почти везде, где нужно искать хранимые объекты. Даже малышей учат держать свои вещи «в порядке», и они уже сталкиваются с некоторыми видами сортировок задолго до того, как познакомятся с азами арифметики \*).

Таким образом, разговор о сортировке вполне уместен и важен, если речь идет об обработке данных. Что может легче сортироваться, чем данные? Тем не менее наш первоначальный интерес к сортировке

---

\*) Между сортировкой и арифметикой нет никакой видимой связи, да, по-видимому, и невидимой нет. Сортировка — некий «первичный» процесс человеческой деятельности. — *Прим. перев.*

следующим образом:

```
TYPE item = RECORD key: INTEGER;  
                (* здесь описаны другие компоненты *)  
            END
```

(2.2)

Под «другими компонентами» подразумеваются данные, по существу относящиеся к сортируемым элементам, а ключ просто идентифицирует каждый элемент. Говоря об алгоритмах сортировки, мы будем обращать внимание лишь на компоненту-ключ, другие же компоненты можно даже и не определять. Поэтому из наших дальнейших рассмотрений вся сопутствующая информация опускается, и мы считаем, что тип элемента определен как `INTEGER`. Выбор `INTEGER` до некоторой степени произволен. Можно было взять и другой тип, на котором определяется общее отношение порядка.

Метод сортировки называется *устойчивым*, если в процессе сортировки относительное расположение элементов с равными ключами не изменяется. Устойчивость сортировки часто бывает желательной, если речь идет об элементах, уже упорядоченных (отсортированных) по некоторым вторичным ключам (т. е. свойствам), не влияющим на основной ключ.

Данную главу не следует рассматривать как исчерпывающий обзор методов сортировки. Наоборот, мы подробно рассматриваем лишь некоторые, отобранные, специфические приемы. За подробным обсуждением сортировок заинтересованный читатель может обратиться к великолепному и всеобъемлющему исследованию Д. Кнута [2.7] (см. также [2.10]).

## 2.2. СОРТИРОВКА МАССИВОВ

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться *на том же месте*, т. е. методы, в которых элементы из массива *a* передаются в результирующий массив *b*, представляют существенно меньший интерес. Ограни-

чив критерием экономии памяти наш выбор нужного метода среди многих возможных, мы будем сначала классифицировать методы по их экономичности, т. е. по времени их работы. Хорошей мерой эффективности может быть  $S$  — число необходимых сравнений ключей и  $M$  — число пересылок (перестановок) элементов. Эти числа суть функции от  $n$  — числа сортируемых элементов. Хотя хорошие алгоритмы сортировки требуют порядка  $n \cdot \log n$  сравнений, мы сначала разберем несколько простых и очевидных методов, их называют *прямыми*, где требуется порядка  $n^2$  сравнений ключей. Начинать разбор с прямых методов, не трогая быстрых алгоритмов, нас заставляют такие причины:

1. Прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок.

2. Программы этих методов легко понимать, и они коротки. Напомним, что сами программы также занимают память.

3. Усложненные методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых  $n$  прямые методы оказываются быстрее, хотя при больших  $n$  их использовать, конечно, не следует.

Методы сортировки «на том же месте» можно разбить в соответствии с определяющими их принципами на три основные категории:

1. Сортировки с помощью включения (by insertion).

2. Сортировки с помощью выделения (by selection).

3. Сортировки с помощью обменов (by exchange).

Теперь мы исследуем эти принципы и сравним их. Все программы оперируют переменной  $a$ , именно здесь хранятся переставляемые на месте элементы, и ссылаются на типы *item* (2.2) и *index*, определяемый следующим образом:

```
TYPE index = INTEGER;
VAR a: ARRAY[1 .. n] OF item
```

(2.3)

## 2.2.1. Сортировка с помощью прямого включения

Такой метод широко используется при игре в карты. Элементы (карты) мысленно делятся на уже «готовую» последовательность  $a_1, \dots, a_{i-1}$  и исходную последовательность. При каждом шаге, начиная с  $i = 2$  и увеличивая  $i$  каждый раз на единицу, из исходной последовательности извлекается  $i$ -й элемент и перекладывается в готовую последовательность, при этом он вставляется на нужное место.

Таблица 2.1. Пример сортировки с помощью прямого включения

|                 |    |    |    |    |    |    |    |    |
|-----------------|----|----|----|----|----|----|----|----|
| Начальные ключи | 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
| $i=2$           | 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
| $i=3$           | 12 | 44 | 55 | 42 | 94 | 18 | 06 | 67 |
| $i=4$           | 12 | 42 | 44 | 55 | 94 | 18 | 06 | 67 |
| $i=5$           | 12 | 42 | 44 | 55 | 94 | 18 | 06 | 67 |
| $i=6$           | 12 | 18 | 42 | 44 | 55 | 94 | 06 | 67 |
| $i=7$           | 06 | 12 | 18 | 42 | 44 | 55 | 94 | 67 |
| $i=8$           | 06 | 12 | 18 | 42 | 44 | 55 | 67 | 94 |

В табл. 2.1 показан в качестве примера процесс сортировки с помощью включения восьми случайно выбранных чисел. Алгоритм этой сортировки таков:

```
FOR  $i := 2$  TO  $n$  DO
   $x := a[i]$ ;
  включение  $x$  на соответствующее место среди  $a[1] \dots a[i]$ 
END
```

В реальном процессе поиска подходящего места удобно, чередуя сравнения и движения по последовательности, как бы просеивать  $x$ , т. е.  $x$  сравнивается с очередным элементом  $a_j$ , а затем либо  $x$  вставляется на свободное место, либо  $a_j$  сдвигается (передается) вправо и процесс «уходит» влево. Обратите внимание, что процесс просеивания может закончиться при выполнении одного из двух следующих различных условий:

1. Найден элемент  $a_j$  с ключом, меньшим чем ключ у  $x$ .
2. Достигнут левый конец готовой последовательности.

```

PROCEDURE StraightInsertion;
  VAR i, j: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    x := a[i]; a[0] := x; j := 1;
    WHILE x < a[j-1] DO a[j] := a[j-1]; j := j+1 END;
    a[j] := x
  END
END StraightInsertion

```

Прогр. 2.1. Сортировка с помощью прямого включения.

Такой типичный случай повторяющегося процесса с двумя условиями окончания позволяет нам воспользоваться хорошо известным приемом «барьера» (sentinel). Здесь его легко применить, поставив барьер  $a_0$  со значением  $x$ . (Заметим, что для этого необходимо расширить диапазон индекса в описании переменной  $a$  до  $0 \dots n$ .) Полный алгоритм приводится в прогр. 2.1 \*).

*Анализ метода прямого включения.* Число сравнений ключей ( $C_1$ ) при  $i$ -м просеивании самое большее равно  $i - 1$ , самое меньшее — 1; если предположить, что все перестановки из  $n$  ключей равновероятны, то среднее число сравнений —  $i/2$ . Число же пересылок (присваиваний элементов)  $M_1$  равно  $C_1 + 2$  (включая барьер). Поэтому общее число сравнений и число пересылок таковы:

$$\begin{array}{ll}
 C_{\min} = n-1 & M_{\min} = 3 \cdot (n-1) \\
 C_{\text{ave}} = (n^2 + n - 2)/4 & M_{\text{ave}} = (n^2 + 9n - 10)/4 \\
 C_{\max} = (n^2 + n - 4)/4 & M_{\max} = (n^2 + 3n - 4)/2
 \end{array} \quad (2.4)$$

Минимальные оценки встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие же оценки — когда они первоначально расположены в обратном порядке. В некотором смысле сортировка с помощью включений демонстри-

---

\*) Стоит отметить, что приведенная программа не совсем соответствует естественному ходу мышления при ее построении. Естественным кажется вести поиск не влево от очередного просматриваемого элемента, а с самого начала готовой последовательности. При этом и барьер специально ставить нет нужды. Попробуйте составить такую программу! — *Прим. перев.*

```

PROCEDURE BinaryInsertion;
  VAR i, j, m, L, R: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    x := a[i]; L := 1; R := i;
    WHILE L < R DO
      m := (L + R) DIV 2;
      IF a[m] <= x THEN L := m + 1 ELSE R := m END
    END;
    FOR j := i TO R + 1 BY -1 DO a[j] := a[j - 1] END;
    a[R] := x
  END
END BinaryInsertion

```

Прогр. 2.2. Сортировка методом деления пополам.

рует истинно естественное поведение. Ясно, что приведенный алгоритм описывает процесс устойчивой сортировки: порядок элементов с равными ключами при нем остается неизменным.

Алгоритм с прямыми включениями можно легко улучшить, если обратить внимание на то, что готовая последовательность, в которую надо вставить новый элемент, сама уже упорядочена. Естественно остановиться на двоичном поиске, при котором делается попытка сравнения с серединой готовой последовательности, а затем процесс деления пополам идет до тех пор, пока не будет найдена точка включения. Такой модифицированный алгоритм сортировки называется *методом с двоичным включением* (binary insertion), и он представлен как прогр. 2.2.

*Анализ двоичного включения.* Место включения обнаружено, если  $L = R$ . Таким образом, в конце поиска интервал должен быть единичной длины; значит, деление его пополам производится  $i \log i$  раз. Таким образом,

$$C = \sum_{i=1}^n i \log i$$

Аппроксимируем эту сумму интегралом

$$\int_1^n \log x \, dx = n * (\log n - c) + c \quad (2.5)$$

где  $c = \log e = 1/\ln 2 = 1.4426 \dots$ . Число сравнений фактически не зависит от начального порядка элементов. Однако из-за того, что при делении, фигурирую-

шем при разбиении интервала поиска пополам, происходит отбрасывание дробной части, истинное число сравнений может оказаться на единицу больше. Все это приводит к тому, что в нижней части последовательности место включения отыскивается в среднем несколько быстрее, чем в верхней части, поэтому предпочтительна ситуация, в которой элементы чуть-чуть упорядочены. Фактически, если в исходном состоянии элементы расположены в обратном порядке, потребуется минимальное число сравнений, а если они уже упорядочены — максимальное число. Следовательно, можно говорить о неестественном поведении алгоритма поиска.

$$C \doteq n * (\log n - \log e \pm 0.5)$$

К несчастью, улучшения, порожденные введением двоичного поиска, касаются лишь числа сравнения, а не числа необходимых подвижек чисел. Поскольку движение элемента, т. е. самого ключа, и связанной с ним информации занимает значительно больше времени, чем сравнение двух ключей<sup>\*)</sup>, то фактически улучшения не столь уж существенны, ведь важный член  $M$  так и продолжает оставаться порядка  $n^2$ . И на самом деле, сортировка уже отсортированного массива потребует больше времени, чем в случае последовательной сортировки с прямыми включениями.

Этот пример показывает, что «очевидные улучшения» часто дают не столь уж большой выигрыш, как это кажется на первый взгляд, а в некоторых случаях (случающихся на самом деле) эти «улучшения» могут фактически привести к ухудшениям. После всего сказанного сортировка с помощью включений уже не кажется столь удобным методом для цифровых машин: включение одного элемента с последующим сдвигом на одну позицию целой группы элементов не экономно. Остается впечатление, что лучший результат дадут методы, где передвижка, пусть и на

---

<sup>\*)</sup> Это соображение верно не для всех машин. Дело в том, что сравнение связано с приостановкой «конвейера» процессора до принятия решения, а пересылка этот процесс не нарушает и идет достаточно быстро. Однако это верно для случая быстрых машин. — *Прим. перев.*

большие расстояния, будет связана лишь с одним-единственным элементом.

### 2.2.2. Сортировка с помощью прямого выбора

Этот прием основан на следующих принципах:

1. Выбирается элемент с наименьшим ключом.
2. Он меняется местами с первым элементом  $a_1$ .
3. Затем этот процесс повторяется с оставшимися  $n - 1$  элементами,  $n - 2$  элементами и т. д. до тех пор, пока не останется один, самый большой элемент.

Процесс работы этим методом с теми же восемью ключами, что и в табл. 2.1, приведен в табл. 2.2. Алгоритм формулируется так:

```
FOR i := 1 TO n-1 DO
    присвоить k индекс наименьшего из a[i]... a[n];
    поменять местами a[i] и a[k];
END
```

Такой метод — его называют *прямым выбором* — в некотором смысле противоположен прямому включению. При прямом включении на каждом шаге рассматриваются только *один* очередной элемент исходной последовательности и *все* элементы готовой последовательности, среди которых отыскивается точка включения; при прямом выборе для поиска *одного* элемента с наименьшим ключом просматриваются все элементы исходной последовательности и найденный помещается как очередной элемент в готовую последовательность. Полностью алгоритм прямого выбора приводится в прогр. 2.3.

Таблица 2.2. Пример сортировки с помощью прямого выбора

|                 |    |    |    |    |    |    |    |    |
|-----------------|----|----|----|----|----|----|----|----|
| Начальные ключи | 44 | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
|                 | 06 | 55 | 12 | 42 | 94 | 18 | 44 | 67 |
|                 | 06 | 12 | 55 | 42 | 94 | 18 | 44 | 67 |
|                 | 06 | 12 | 18 | 42 | 94 | 55 | 44 | 67 |
|                 | 06 | 12 | 18 | 42 | 94 | 55 | 44 | 67 |
|                 | 06 | 12 | 18 | 42 | 44 | 55 | 94 | 67 |
|                 | 06 | 12 | 18 | 42 | 44 | 55 | 94 | 67 |
|                 | 06 | 12 | 18 | 42 | 44 | 55 | 67 | 94 |



```

PROCEDURE StraightSelection;
  VAR i, j, k: index; x: item;
BEGIN
  FOR i := 1 TO n-1 DO
    k := i; x := a[i];
    FOR j := i+1 TO n DO
      IF a[j] < x THEN k := j; x := a[k] END
    END;
    a[k] := a[i]; a[i] := x
  END
END StraightSelection

```

Прогр. 2.3. Сортировка с помощью прямого выбора.

*Анализ прямого выбора.* Число сравнений ключей (C), очевидно, не зависит от начального порядка ключей. Можно сказать, что в этом смысле поведение этого метода менее естественно, чем поведение прямого включения. Для C имеем

$$C = (n^2 - n)/2$$

Число перестановок минимально

$$M_{\min} = 3 * (n - 1) \quad (2.6)$$

в случае изначально упорядоченных ключей и максимально

$$M_{\max} = n^2/4 + 3 * (n - 1)$$

если первоначально ключи располагались в обратном порядке. Для того чтобы определить  $M_{\text{avg}}$ , мы должны рассуждать так. Алгоритм просматривает массив, сравнивая каждый элемент с только что обнаруженной минимальной величиной; если он меньше первого, то выполняется некоторое присваивание. Вероятность, что второй элемент окажется меньше первого, равна  $1/2$ , с этой же вероятностью происходят присваивания минимуму. Вероятность, что третий элемент окажется меньше первых двух, равна  $1/3$ , а вероятность для четвертого оказаться наименьшим —  $1/4$  и т. д. Поэтому полное ожидаемое число пересылок равно  $H_n - 1$ , где  $H_n$  —  $n$ -е гармоническое число:

$$H_n = 1 + 1/2 + 1/3 + \dots + 1/n \quad (2.7)$$

$H_n$  можно выразить и так:

$$H_n = \ln n + g + 1/2n - 1/12n^2 + \dots \quad (2.8)$$

где  $g = 0.577216 \dots$  — константа Эйлера. Для достаточно больших  $n$  мы можем игнорировать дробные составляющие и поэтому аппроксимировать среднее число присваиваний на  $i$ -м просмотре выражением

$$F_i = \ln i + g + 1$$

Среднее число пересылок  $M_{avg}$  в сортировке с выбором есть сумма  $F_i$  с  $i$  от 1 до  $n$ :

$$M_{avg} = n * (g + 1) + (Si: 1 \leq i \leq n: \ln i)$$

Вновь аппроксимируя эту сумму дискретных членов интегралом

$$\text{Integral}(1 : n) \ln x \, dx = x * (\ln x - 1) = n * \ln(n) - n + 1$$

получаем, наконец, приблизительное значение

$$M_{avg} \doteq n * (\ln(n) + g)$$

Отсюда можно сделать заключение, что, как правило, алгоритм с прямым выбором предпочтительнее строгого включения. Однако, если ключи в начале упорядочены или почти упорядочены, прямое включение будет оставаться несколько более быстрым.

### 2.2.3. Сортировка с помощью прямого обмена

Классификация методов сортировки редко бывает осмысленной. Оба разбиравшихся до этого метода можно тоже рассматривать как «обменные» сортировки. В данном же, однако, разделе мы опишем метод, где обмен местами двух элементов представляет собой характернейшую особенность процесса. Изложенный ниже алгоритм прямого обмена основывается на сравнении и смене мест для пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы.

Как и в упоминавшемся методе прямого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если мы будем рас-

Таблица 2.3. Пример пузырьковой сортировки

| i=1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
|-----|----|----|----|----|----|----|----|
| 44  | 06 | 06 | 06 | 06 | 06 | 06 | 06 |
| 55  | 44 | 12 | 12 | 12 | 12 | 12 | 12 |
| 12  | 55 | 44 | 18 | 18 | 18 | 18 | 18 |
| 42  | 12 | 55 | 44 | 42 | 42 | 42 | 42 |
| 94  | 42 | 18 | 55 | 44 | 44 | 44 | 44 |
| 18  | 94 | 42 | 42 | 55 | 55 | 55 | 55 |
| 06  | 18 | 94 | 67 | 67 | 67 | 67 | 67 |
| 67  | 67 | 67 | 94 | 94 | 94 | 94 | 94 |

смаатривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в чане с водой, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу (см. табл. 2.3). Такой метод широко известен под именем «*пузырьковая сортировка*». В своем простейшем виде он представлен в прогр. 2.4.

Улучшения этого алгоритма напрашиваются сами собой. На примере в табл. 2.3 видно, что три последних прохода не влияют на порядок элементов из-за того, что они уже отсортированы. Очевидный прием улучшения этого алгоритма — запоминать, были или не были перестановки в процессе некоторого прохода. Если в последнем проходе перестановок не было, то алгоритм можно заканчивать. Это улучшение однако, можно опять же улучшить, если запоминать не только сам факт, что обмен имел место, но и положение

```

PROCEDURE BubbleSort;
  VAR i, j: index; x: item;
BEGIN
  FOR i := 2 TO n DO
    FOR j := n TO i BY -1 DO
      IF a[j-1] > a[j] THEN
        x := a[j-1]; a[j-1] := a[j]; a[j] := x
      END
    END
  END
END BubbleSort

```

Прогр. 2.4. Пузырьковая сортировка.

(индекс) последнего обмена. Ясно, что все пары соседних элементов выше этого индекса  $k$  уже находятся в желаемом порядке. Поэтому просмотры можно заканчивать на этом индексе, а не идти до заранее определенного нижнего предела для  $i$ . Внимательный программист заметит тем не менее здесь некоторую своеобразную асимметрию. Один плохо расположенный пузырек на «тяжелом конце» в массиве с обратным порядком будет перемещаться на нужное место в один проход, но плохо расположенный элемент на «легком конце» будет просачиваться на свое нужное место на один шаг при каждом проходе\*). Например, массив

12 18 42 44 55 67 94 06

с помощью усовершенствованной «пузырьковой» сортировки можно упорядочить за один просмотр, а для сортировки массива

94 06 12 18 42 44 55 67

требуется семь просмотров. Такая неестественная симметрия наводит на мысль о третьем улучшении: чередовать направление последовательных просмотров. Получающийся при этом алгоритм мы соответственно назовем «шейкерной» сортировкой (ShakerSort)\*\*). Таблица 2.4 иллюстрирует сортировку новым способом тех же (табл. 2.3) восьми ключей.

*Анализ пузырьковой и шейкерной сортировок.* Число сравнений в строго обменном алгоритме

$$C = (n^2 - n)/2 \quad (2.10)$$

а минимальное, среднее и максимальное число перемещений элементов (присваиваний) равно соответственно

$$M_{\min} = 0, \quad M_{\text{avg}} = 3 * (n^2 - n)/2, \quad M_{\max} = 3 * (n^2 - n)/4 \quad (2.11)$$

---

\*) Проще было бы сказать так: всплывает пузырек сразу, за один проход, а тонет очень медленно, за один проход на одну позицию. — *Прим. перев.*

\*\*) Напомним, что шейкером называется нечто вроде двух накрывающих друг друга стаканов, в которых встряхиванием вверх-вниз готовят коктейль. — *Прим. перев.*

Таблица 2.4. Пример шейкерной сортировки

|      |    |    |    |    |    |
|------|----|----|----|----|----|
| L=   | 2  | 3  | 3  | 4  | 4  |
| R=   | 8  | 8  | 7  | 7  | 4  |
| dir= | ↑  | ↓  | ↑  | ↓  | ↑  |
|      | 44 | 06 | 06 | 06 | 06 |
|      | 55 | 44 | 44 | 12 | 12 |
|      | 12 | 55 | 12 | 44 | 18 |
|      | 42 | 12 | 42 | 18 | 42 |
|      | 94 | 42 | 55 | 42 | 44 |
|      | 18 | 94 | 18 | 55 | 55 |
|      | 06 | 18 | 67 | 67 | 67 |
|      | 67 | 67 | 94 | 94 | 94 |

Анализ же улучшенных методов, особенно шейкерной сортировки, довольно сложен. Минимальное число сравнений  $C_{\min} = n - 1$ . Кнут считает, что для улучшенной пузырьковой сортировки среднее число проходов пропорционально  $n - k_1 n^{1/2}$ , а среднее число сравнений пропорционально  $1/2(n^2 - n(k_2 + \ln n))$ . Обратите, однако, внимание на то, что все перечисленные выше усовершенствования не влияют на число перемещений, они лишь сокращают число излишних двойных проверок. К несчастью, обмен местами двух элементов — чаще всего более дорогостоящая опера-

```
PROCEDURE ShakerSort;
```

```
  VAR j, k, L, R: index; x: item;
```

```
  BEGIN L := 2; R := n; k := n;
```

```
    REPEAT
```

```
      FOR j := R TO L BY -1 DO
```

```
        IF a[j-1] > a[j] THEN
```

```
          x := a[j-1]; a[j-1] := a[j]; a[j] := x; k := j
```

```
        END
```

```
      END;
```

```
      L := k+1;
```

```
      FOR j := L TO R BY +1 DO
```

```
        IF a[j-1] > a[j] THEN
```

```
          x := a[j-1]; a[j-1] := a[j]; a[j] := x; k := j
```

```
        END
```

```
      END;
```

```
      R := k-1
```

```
    UNTIL L > R
```

```
  END ShakerSort
```

Прогр. 2.5. Шейкерная сортировка.

ция, чем сравнение ключей, поэтому наши, вроде очевидные, улучшения дают не такой уж большой выигрыш, как мы интуитивно ожидали.

Такой анализ показывает, что «обменная» сортировка и ее небольшие усовершенствования представляют собой нечто среднее между сортировками с помощью включений и с помощью выбора. Фактически в пузырьковой сортировке нет ничего ценного, кроме ее привлекательного названия (Bubblesort). Шейкерная же сортировка с успехом используется в тех случаях, когда известно, что элементы почти упорядочены — на практике это бывает весьма редко.

Можно показать, что среднее расстояние, на которое должен продвигаться каждый из  $n$  элементов во время сортировки, равно  $n/3$  «мест». Эта цифра является целью в поиске улучшений, т. е. в поиске более эффективных методов сортировки. Все строгие приемы сортировки фактически передвигают каждый элемент на всяком элементарном шаге на одну позицию. Поэтому они требуют порядка  $n^2$  таких шагов. Следовательно, в основу любых улучшений должен быть положен принцип перемещения на каждом такте элементов на большие расстояния.

Далее мы рассматриваем три улучшенных метода: по одному для каждого из основных строгих методов сортировки — включения, выбора и обмена.

## 2.3. УЛУЧШЕННЫЕ МЕТОДЫ СОРТИРОВКИ

### 2.3.1. Сортировка с помощью включений с уменьшающимися расстояниями

В 1959 г. Д. Шеллом было предложено усовершенствование сортировки с помощью прямого включения. Сам метод объясняется и демонстрируется на нашем стандартном примере (см. табл. 2.5). Сначала отдельно группируются и сортируются элементы, отстоящие друг от друга на расстоянии 4. Такой процесс называется четверной сортировкой. В нашем примере восемь элементов и каждая группа состоит точно из двух элементов. После первого прохода элементы перегруппировываются — теперь каждый элемент группы

Таблица 2.5. Сортировка с помощью включений с уменьшающимися расстояниями

|                           |    |    |    |    |    |    |    |
|---------------------------|----|----|----|----|----|----|----|
| 44                        | 55 | 12 | 42 | 94 | 18 | 06 | 67 |
| четверная сортировка дает |    |    |    |    |    |    |    |
| 44                        | 18 | 06 | 42 | 94 | 55 | 12 | 67 |
| двойная сортировка дает   |    |    |    |    |    |    |    |
| 06                        | 18 | 12 | 42 | 44 | 55 | 94 | 67 |
| одинарная сортировка дает |    |    |    |    |    |    |    |
| 06                        | 12 | 18 | 42 | 44 | 55 | 67 | 94 |

отстоит от другого на две позиции — и вновь сортируются. Это называется двойной сортировкой. И наконец, на третьем проходе идет обычная или одинарная сортировка.

На первый взгляд можно засомневаться: если необходимо несколько процессов сортировки, причем в каждый включаются все элементы, то не добавят ли они больше работы, чем сэкономят? Однако на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуется сравнительно немного перестановок.

Ясно, что такой метод в результате дает упорядоченный массив, и, конечно же, сразу видно, что каждый проход от предыдущих только выигрывает (так как каждая  $i$ -сортировка объединяет две группы, уже отсортированные  $2i$ -сортировкой). Так же очевидно, что расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу. Однако совсем не очевидно, что такой прием «уменьшающихся расстояний» может дать лучшие результаты, если расстояния не будут степенями двойки. Поэтому приводимая программа не ориентирована на некую определенную последовательность расстояний. Все  $t$  расстояний обозначаются соответственно  $h_1, h_2, \dots, h_t$ , для них выполняются условия

$$h_t = 1, h_{i+1} < h_i \quad (2.12)$$

Каждая  $h$ -сортировка программируется как сортировка с помощью прямого включения. Причем про-

```

PROCEDURE ShellSort;
  CONST t = 4;
  VAR i, j, k, s: index;
      x: item; m: 1..t;
      h: ARRAY [1..t] OF INTEGER;
BEGIN h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
  FOR m := 1 TO t DO
    k := h[m]; s := -k; (* место барьера *)
    FOR i := k+1 TO n DO
      x := a[i]; j := i-k;
      IF s = 0 THEN s := -k END;
      s := s+1; a[s] := x;
      WHILE x < a[j] DO a[j+k] := a[j]; j := j-k END;
      a[j+k] := x
    END
  END
END ShellSort

```

Прогр. 2.6. Сортировка Шелла.

стота условия окончания поиска места для включения обеспечивается методом барьеров. Ясно, что каждая из сортировок нуждается в постановке своего собственного барьера и программу для определения его местоположения необходимо делать насколько возможно простой. Поэтому приходится расширять массив не только на одну-единственную компоненту  $a_0$ , а на  $h_1$  компонент. Его описание теперь выглядит так:

a: ARRAY[—  $h_1$  .. n] OF item

Сам алгоритм для  $t=4$  описывается процедурой *ShellSort* [2.11] в прогр. 2.6.

*Анализ сортировки Шелла.* Анализ этого алгоритма поставил несколько весьма трудных математических проблем, многие из которых так еще и не решены. В частности, не известно, какие расстояния дают наилучшие результаты. Но вот удивительный факт: они не должны быть множителями один другого. Это позволяет избежать явления уже очевидного из приведенного выше примера, когда при каждом проходе взаимодействуют две цепочки, которые до этого нигде еще не пересекались. И действительно, желательно, чтобы взаимодействие различных цепочек проходило как можно чаще. Справедлива такая теорема: если  $k$ -отсортированную последовательность



$i$ -отсортировать, то она остается  $k$ -отсортированной. В работе [2.8] Кнут показывает, что имеет смысл использовать такую последовательность (она записана в обратном порядке): 1, 4, 13, 40, 121, ..., где  $h_{k-1} = 3h_k + 1$ ,  $h_t = 1$  и  $t = \lfloor \log_3 n \rfloor - 1$ . Он рекомендует и другую последовательность: 1, 3, 7, 15, 31, ... где  $h_{k-1} = 2h_k + 1$ ,  $h_t = 1$  и  $t = \lfloor \log_2 n \rfloor - 1$ . Математический анализ показывает, что в последнем случае для сортировки  $n$  элементов методом Шелла затраты пропорциональны  $n^{1.2}$ . Хотя это число значительно лучше  $n^2$ , тем не менее мы не ориентируемся в дальнейшем на этот метод, поскольку существуют еще лучшие алгоритмы.

### 2.3.2. Сортировка с помощью дерева

Метод сортировки с помощью прямого выбора основан на повторяющихся поисках наименьшего ключа среди  $n$  элементов, среди оставшихся  $n - 1$  элементов и т. д. Обнаружение наименьшего среди  $n$  элементов требует — это очевидно —  $n - 1$  сравнения, среди  $n - 1$  уже нужно  $n - 2$  сравнений и т. д. Сумма первых  $n - 1$  целых равна  $1/2(n^2 - n)$ . Как же в таком случае можно усовершенствовать упомянутый метод сортировки? Этого можно добиться, только оставляя после каждого прохода больше информации, чем просто идентификация единственного минимального элемента. Например, сделав  $n/2$  сравнений, можно определить в каждой паре ключей меньший. С помощью  $n/4$  сравнений — меньший из пары уже выбранных меньших и т. д. Прodelав  $n - 1$  сравнений, мы можем построить дерево выбора вроде представленного на рис. 2.3 и идентифицировать его корень как нужный нам наименьший ключ [2.2].

Второй этап сортировки — спуск вдоль пути, отмеченного наименьшим элементом, и исключение его из дерева путем замены либо на пустой элемент (дырку) в самом низу, либо на элемент из соседней ветви в промежуточных вершинах (см. рис. 2.4 и 2.5). Элемент, передвинувшийся в корень дерева, вновь будет наименьшим (теперь уже вторым) ключом, и его можно исключить. После  $n$  таких шагов дерево ста-

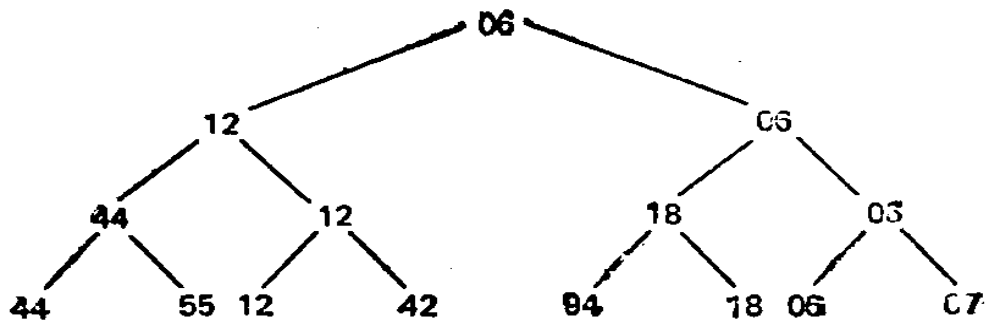


Рис. 2.3. Повторяющиеся выборы среди двух ключей.

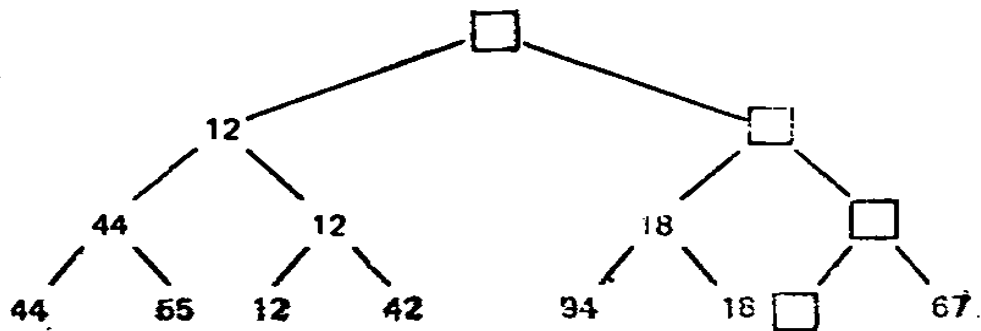


Рис. 2.4. Выбор наименьшего ключа.

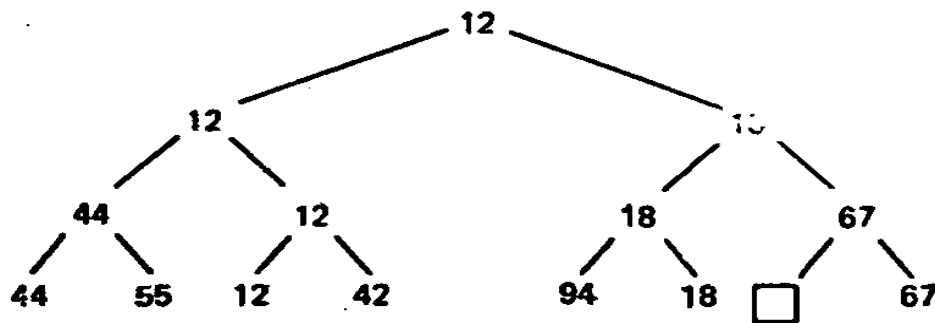


Рис. 2.5. Заполнение дырок.

нет пустым (т. е. в нем останутся только дырки), и процесс сортировки заканчивается. Обратите внимание — на каждом из  $n$  шагов выбора требуется только  $\log n$  сравнений. Поэтому на весь процесс понадобится порядка  $n \cdot \log n$  элементарных операций плюс еще  $n$  шагов на построение дерева. Это весьма существенное улучшение не только прямого метода, требующего  $n^2$  шагов, но и даже метода Шелла, где нужно  $n^{1.2}$  шага. Естественно, сохранение дополнительной информации делает задачу более изощренной, поэтому в сортировке по дереву каждый отдельный шаг усложняется. Ведь в конце концов для сохранения избыточной информации, получаемой при

начальном проходе, создается некоторая древообразная структура. Наша следующая задача — найти приемы эффективной организации этой информации.

Конечно, хотелось бы, в частности, избавиться от дырок, которыми в конечном итоге будет заполнено все дерево и которые порождают много ненужных сравнений. Кроме того, надо было бы поискать такое представление дерева из  $n$  элементов, которое требует лишь  $n$  единиц памяти, а не  $2n - 1$ , как это было раньше. Этих целей действительно удалось добиться в методе *Heapsort*\*) , изобретенном Д. Уилльямсом [2.14], где было получено, очевидно, существенное улучшение традиционных сортировок с помощью деревьев. Пирамида определяется как последовательность ключей  $h_L, h_{L+1}, \dots, h_R$ , такая, что

$$h_i \leq h_{2i} \text{ и } h_i \leq h_{2i+1} \text{ для } i = L \dots R/2. \quad (2.13)$$

Если любое двоичное дерево рассматривать как массив по схеме на рис. 2.6, то можно говорить, что деревья сортировок на рис. 2.7 и 2.8 суть пирамиды, а элемент  $h_1$ , в частности, их наименьший элемент:  $h_1 = \min(h_1, h_2, \dots, h_n)$ . Предположим, есть некоторая пирамида с заданными элементами  $h_{L+1}, \dots, h_R$  для некоторых значений  $L$  и  $R$  и нужно добавить новый элемент  $x$ , образуя расширенную пирамиду  $h_L, \dots, h_R$ . Возьмем, например, в качестве исходной пирамиду  $h_1, \dots, h_7$ , показанную на рис. 2.7, и добавим к ней слева элемент  $h_1 = 44$  \*\*). Новая пирамида получается так: сначала  $x$  ставится наверх древовидной структуры, а затем он постепенно опускается вниз каждый раз по направлению наименьшего из двух примыкающих к нему элементов, а сам этот элемент передвигается вверх. В приведенном примере значение 44 сначала меняется местами с 06, затем с 12 и

---

\*) Здесь мы оставляем попытки перевести названия соответствующих методов на русский язык, ибо ни уже не более чем собственные имена, хотя в названиях первых упоминавшихся методов еще фигурировал некоторый элемент описания сути самого приема сортировки. — *Прим. перев.*

\*\*) Это несколько противоречит утверждению, что  $h_{L+1}, \dots, h_R$  — пирамида, но надеемся, сам читатель разберется, что хотел сказать автор. — *Прим. перев.*

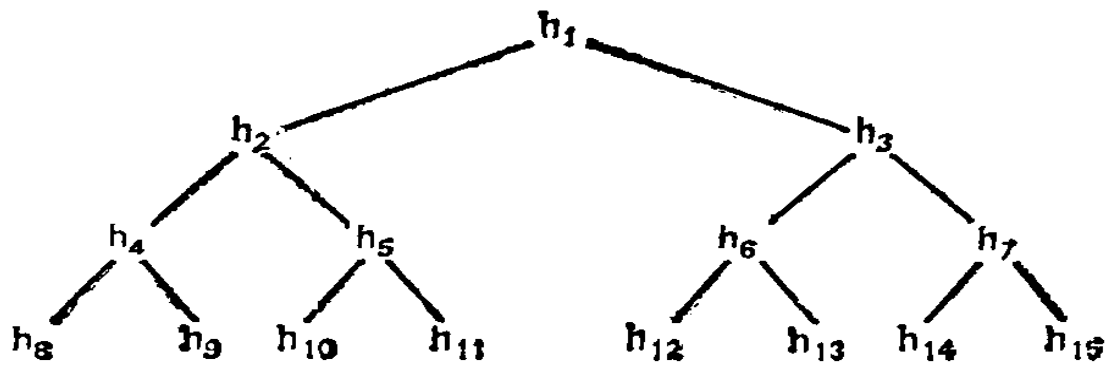
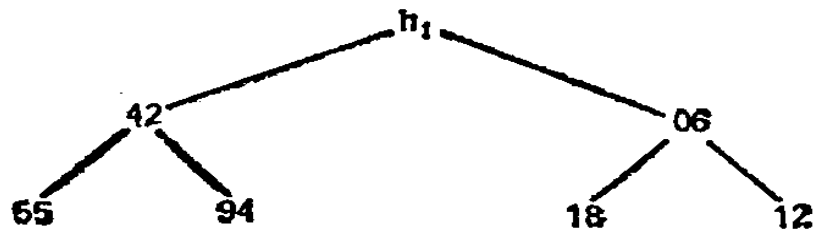
Рис. 2.6. Массив  $h$ , представленный в виде двоичного дерева.

Рис. 2.7. Пирамида из семи элементов.

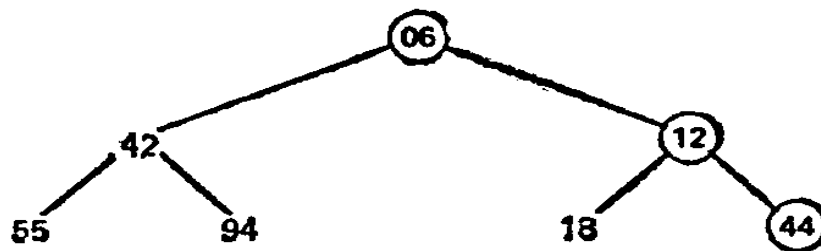


Рис. 2.8. Просеивание ключа 44 через пирамиду.

в результате образуется дерево, представленное на рис. 2.8. Теперь мы сформулируем этот сдвигающий алгоритм так:  $i, j$  — пара индексов, фиксирующих элементы, меняющиеся на каждом шаге местами. Читателю остается лишь убедиться самому, что предложенный метод сдвигов действительно сохраняет неизменным условия (2.13), определяющие пирамиду.

Р. Флойдом был предложен некий «лаконичный» способ построения пирамиды «на том же месте». Его процедура сдвига представлена как прогр. 2.7. Здесь  $h_1 \dots h_n$  — некий массив, причем  $h_m \dots h_n$  ( $m = (n \text{ DIV } 2) + 1$ ) уже образуют пирамиду, поскольку индексов  $i, j$ , удовлетворяющих отношениям  $j = 2i$  (или  $j = 2i + 1$ ), просто не существует. Эти элементы образуют как бы нижний слой соответствующего двоичного дерева (см. рис. 2.6), для них никакой

```

PROCEDURE sift(L, R: index);
  VAR i, j: index; x: item;
BEGIN i := L; j := 2*L; x := a[L];
  IF (j < R) & (a[j+1] < a[j]) THEN j := j+1 END;
  WHILE (j <= R) & (a[j] < x) DO
    a[i] := a[j]; i := j; j := 2*j;
    IF (j < R) & (a[j+1] < a[j]) THEN j := j+1 END
  END
END sift

```

Прогр. 2.7. Sift.

упорядоченности не требуется. Теперь пирамида расширяется влево; каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент. Табл. 2.6 иллюстрирует весь этот процесс, а получающаяся пирамида показана на рис. 2.6.

Следовательно, процесс формирования пирамиды из  $n$  элементов  $h_1 \dots h_n$  на том же самом месте описывается так:

```

L := (n DIV 2) + 1;
WHILE L > 1 DO L := L - 1; sift(L, n) END

```

Для того чтобы получить не только частичную, но и полную упорядоченность среди элементов, нужно проделать  $n$  сдвигающих шагов, причем после каждого шага на вершину дерева выталкивается очередной (наименьший) элемент. И вновь возникает вопрос: где хранить «всплывающие» верхние элементы и можно ли или нельзя проводить обращение на том же месте? Существует, конечно, такой выход: каждый раз брать последнюю компоненту пирамиды (скажем, это будет  $x$ ), прятать верхний элемент пирамиды в освободившемся теперь месте, а  $x$  сдвигать в нужное место. В табл. 2.7 приведены необходимые в этом слу-

Таблица 2.6. Построение пирамиды

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 44 | 55 | 12 | 42 |    | 94 | 18 | 06 | 67 |
| 44 | 55 | 12 |    | 42 | 94 | 18 | 06 | 67 |
| 44 | 55 |    | 06 | 42 | 94 | 18 | 12 | 67 |
| 44 |    | 42 | 06 | 55 | 94 | 18 | 12 | 67 |
| 05 | 42 | 12 | 55 | 94 | 18 | 44 | 67 |    |

Таблица 2.7. Пример процесса сортировки с помощью Heapsort

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 06 | 42 | 12 | 55 | 94 | 18 | 44 | 67 |
| 12 | 42 | 18 | 55 | 94 | 67 | 44 | 06 |
| 18 | 42 | 44 | 55 | 94 | 67 | 12 | 06 |
| 42 | 55 | 44 | 67 | 94 | 18 | 12 | 06 |
| 44 | 55 | 94 | 67 | 42 | 18 | 12 | 06 |
| 55 | 67 | 94 | 44 | 42 | 18 | 12 | 06 |
| 67 | 94 | 55 | 44 | 42 | 18 | 12 | 06 |
| 94 | 67 | 55 | 44 | 42 | 18 | 12 | 06 |

чае  $n - 1$  шагов. Сам процесс описывается с помощью процедуры *sift* (прогр. 2.7) таким образом:

```

R := n;
WHILE R > 1 DO
  x := a[1]; a[1] := a[R]; a[R] := x;
  R := R - 1; sift(1, R)
END

```

Пример из табл. 2.7 показывает, что получающийся порядок фактически является обратным. Однако это можно легко исправить, изменив направление «упорядочивающего отношения» в процедуре *sift*. В конце концов получаем процедуру Heapsort (прогр. 2.8),

```

PROCEDURE HeapSort;
  VAR L, R: index; x: item;

  PROCEDURE sift(L, R: index);
    VAR i, j: index; x: item;
    BEGIN i := L; j := 2 * L; x := a[L];
    IF (j < R) & (a[j] < a[j + 1]) THEN j := j + 1 END;
    WHILE (j <= R) & (x < a[j]) DO
      a[i] := a[j]; i := j; j := 2 * j;
      IF (j < R) & (a[j] < a[j + 1]) THEN j := j + 1 END
    END
  END sift;

  BEGIN L := (n DIV 2) + 1; R := n;
  WHILE L > 1 DO L := L - 1; sift(L, R) END;
  WHILE R > 1 DO
    x := a[1]; a[1] := a[R]; a[R] := x;
    R := R - 1; sift(L, R)
  END
END HeapSort

```

Прогр. 2.8. Heapsort.

*Анализ Heapsort.* На первый взгляд вовсе не очевидно, что такой метод сортировки дает хорошие результаты. Ведь в конце концов большие элементы, прежде чем попадут на свое место в правой части, сначала сдвигаются влево. И действительно, процедуру не рекомендуется применять для небольшого, вроде нашего примера, числа элементов. Для больших же  $n$  Heapsort очень эффективна; чем больше  $n$ , тем лучше она работает. Она даже становится сравнимой с сортировкой Шелла.

В худшем случае нужно  $n/2$  сдвигающих шагов, они сдвигают элементы на  $\log(n/2)$ ,  $\log(n/2 - 1)$ , ..., ...,  $\log(n - 1)$  позиций (логарифм (по основанию 2) «обрубается» до следующего меньшего целого). Следовательно, фаза сортировки требует  $n - 1$  сдвигов с самое большое  $\log(n - 1)$ ,  $\log(n - 2)$ , ..., 1 перемещениями. Кроме того, нужно еще  $n - 1$  перемещений для просачивания сдвинутого элемента на некоторое расстояние вправо. Эти соображения показывают, что даже в самом плохом из возможных случаев Heapsort потребует  $n * \log n$  шагов. Великолепная производительность в таких плохих случаях — одно из привлекательных свойств Heapsort.

Совсем не ясно, когда следует ожидать наихудшей (или наилучшей) производительности. Но вообще-то кажется, что Heapsort «любит» начальные последовательности, в которых элементы более или менее отсортированы в обратном порядке. Поэтому ее поведение несколько неестественно. Если мы имеем дело с обратным порядком, то фаза порождения пирамиды не требует каких-либо перемещений. Среднее число перемещений приблизительно равно  $n/2 * \log(n)$ , причем отклонения от этого значения относительно невелики.

### 2.3.3. Сортировка с помощью разделения

Разобравшись в двух усовершенствованных методах сортировки, построенных на принципах включения и выбора, мы теперь коснемся третьего улучшенного метода, основанного на обмене. Если учесть, что пузырьковая сортировка в среднем была самой неэф-

фективной из всех трех алгоритмов прямой (строгой) сортировки, то следует ожидать относительно существенного улучшения. И все же это выглядит как некий сюрприз: улучшение метода, основанного на обмене, о котором мы будем сейчас говорить, оказывается, приводит к самому лучшему из известных в данный момент методу сортировки для массивов. Его производительность столь впечатляюща, что изобретатель Ч. Хоар [2.5] и [2.6] даже назвал метод *быстрой сортировкой* (Quicksort).

В Quicksort исходят из того соображения, что для достижения наилучшей эффективности сначала лучше производить перестановки на большие расстояния. Предположим, у нас есть  $n$  элементов, расположенных по ключам в обратном порядке. Их можно отсортировать за  $n/2$  обменов, сначала поменять местами самый левый с самым правым, а затем последовательно двигаться с двух сторон. Конечно, это возможно только в том случае, когда мы знаем, что порядок действительно обратный. Но из этого примера можно извлечь и нечто действительно поучительное.

Давайте, попытаемся воспользоваться таким алгоритмом: выберем наугад какой-либо элемент (назовем его  $x$ ) и будем просматривать слева наш массив до тех пор, пока не обнаружим элемент  $a_i > x$ , затем будем просматривать массив справа, пока не встретим  $a_j < x$ . Теперь поменяем местами эти два элемента и продолжим наш процесс просмотра и обмена, пока оба просмотра не встретятся где-то в середине массива. В результате массив окажется разбитым на левую часть, с ключами меньше (или равными)  $x$ , и правую — с ключами больше (или равными)  $x$ . Теперь этот процесс разделения представим в виде процедуры (прогр. 2.9). Обратите внимание, что вместо отношений  $>$  и  $<$  используются  $\geq$  и  $\leq$ , а в заголовке цикла с WHILE — их отрицания  $< \text{и} >$ . При таких изменениях  $x$  выступает в роли барьера для того и другого просмотра. Если взять в качестве  $x$  для сравнения средний ключ 42, то в массиве ключей

44 55 12 42 94 06 18 67



```

PROCEDURE partition;
  VAR w, x: item;
BEGIN i := 1; j := n;
  случайно выбрать x;
  REPEAT
    WHILE a[i] < x DO i := i + 1 END;
    WHILE x < a[j] DO j := j - 1 END;
    IF i <= j THEN
      w := a[i]; a[i] := a[j]; a[j] := w; i := i + 1; j := j - 1
    END
  UNTIL i > j
END partition

```

Прогр. 2.9. Сортировка с помощью разделения.

для разделения понадобятся два обмена:  $18 \leftrightarrow 44$  и  $6 \leftrightarrow 55$

18 06 12 42 94 55 44 67

последние значения индексов таковы:  $i = 5$ , а  $j = 3$ . Ключи  $a_1 \dots a_{i-1}$  меньше или равны ключу  $x = 42$ , а ключи  $a_{j+1} \dots a_n$  больше или равны  $x$ . Следовательно, существует две части, а именно

$$\begin{aligned}
 A_k: 1 \leq k < i: a_k \leq x \\
 A_k: j < k \leq n: x \leq a_k
 \end{aligned}
 \tag{2.14}$$

Описанный алгоритм очень прост и эффективен, поскольку главные сравниваемые величины  $i$ ,  $j$  и  $x$  можно хранить во время просмотра в быстрых регистрах машины. Однако он может оказаться и неудачным, что, например, происходит в случае  $n$  идентичных ключей: для разделения нужно  $n/2$  обменов. Этих вовсе необязательных обменов можно избежать, если операторы просмотра заменить на такие:

```

WHILE a[i] <= x DO i := i + 1 END;
WHILE x <= a[j] DO j := j - 1 END

```

Однако в этом случае, выбранный элемент  $x$ , находящийся среди компонент массива, уже не работает как барьер для двух просмотров. В результате просмотры массива со всеми идентичными ключами приведут, если только не использовать более сложные условия их окончания, к переходу через границы массива. Про-

стота условий, употребленных в прогр. 2.9, вполне оправдывает те дополнительные обмены, которые происходят в среднем относительно редко. Можно еще немного сэкономить, если изменить заголовок, управляющий самым обменом: от  $i \leq j$  перейти к  $i < j$ . Однако это изменение не должно касаться двух операторов:  $i := i + 1$ ,  $j := j - 1$ . Поэтому для них требуется отдельный условный оператор. Убедиться в правильности алгоритма разделения можно, удостоверившись, что отношения (2.14) представляют собой инварианты оператора цикла с REPEAT. Вначале при  $i = 1$  и  $j = n$  их истинность тривиальна, а при выходе  $i > j$  они дают как раз желаемый результат.

Теперь напомним, что наша цель — не только провести разделение на части исходного массива элементов, но и отсортировать его. Сортировку от разделения отделяет, однако, лишь небольшой шаг: нужно применить этот процесс к получившимся двум частям, затем к частям частей, и так до тех пор, пока каждая из частей не будет состоять из одного-единственного элемента. Эти действия описываются программой 2.10.

Процедура *sort* рекурсивно обращается сама к себе. Рекурсии в алгоритмах — это очень мощный механизм, его мы будем рассматривать в гл. 3.

```
PROCEDURE QuickSort;
```

```
  PROCEDURE sort(L, R: index);
```

```
    VAR i, j: index; w, x: item;
```

```
  BEGIN i := L; j := R;
```

```
    x := a[(L+R) DIV 2];
```

```
    REPEAT
```

```
      WHILE a[i] < x DO i := i+1 END;
```

```
      WHILE x < a[j] DO j := j-1 END;
```

```
      IF i <= j THEN
```

```
        w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
```

```
      END
```

```
    UNTIL i > j;
```

```
    IF L < j THEN sort(L, j) END;
```

```
    IF i < R THEN sort(i, R) END
```

```
  END sort;
```

```
BEGIN sort(1, n)
```

```
END QuickSort
```

Прогр. 2.10. Quicksort.

Во многих «старых» языках программирования от рекурсий по некоторым техническим причинам отказывались. Поэтому мы теперь покажем, как тот же алгоритм можно выразить и нерекурсивной процедурой. Решение, очевидно, заключается в том, что рекурсию нужно заменить итерацией. При этом понадобятся, конечно, некоторые дополнительные операции по сохранению нужной информации.

Суть итеративного решения заключается во введении списка требуемых разделений, т. е. разделений, которые необходимо провести. На каждом этапе возникают две задачи по разделению. И только к одной из них мы можем непосредственно сразу же приступить в очередной итерации, другая же заносится в упомянутый список. При этом, конечно, существенно, что требования из списка выполняются несколько специфическим образом, а именно в обратном порядке. Следовательно, первое из перечисленных требований выполняется последним, и наоборот. В приводимой нерекурсивной версии Quicksort каждое требование задается просто левым и правым индексами — это границы части, требующей дальнейшего деления. Таким образом, мы вводим переменную-массив под именем *stack* и индекс *s*, указывающий на самую последнюю строку в стеке (см. прогр. 2.11). О том, как выбрать подходящий размер стека *M*, речь пойдет при анализе работы Quicksort.

**Анализ Quicksort.** Для исследования производительности Quicksort сначала необходимо разобраться, как идет процесс деления. Выбрав некоторое граничное значение *x*, мы затем проходим целиком по всему массиву. Следовательно, при этом выполняется точно *n* сравнений. Число же обменов можно определить из следующих вероятностных соображений. При заданной границе значений *x* ожидаемое число операций обмена равно числу элементов в левой части разделяемой последовательности, т. е.  $n - 1$ , умноженному на вероятность того, что при обмене каждый такой элемент попадает на свое место. Обмен происходит, если этот элемент перед этим находился в правой части. Вероятность этого равна  $(n - (x - 1)) / n$ . Поэтому ожидаемое число обменов есть

```

PROCEDURE NonRecursiveQuickSort;
  CONST M = 12;
  VAR i, j, L, R: index; x, w: item;
      s: [0..M];
      stack: ARRAY [1..M] OF RECORD L, R: index END;
BEGIN s := 1; stack[1].L := 1; stack[s].R := n;
  REPEAT (* выбор из стека последнего запроса *)
    L := stack[s].L; R := stack[s].R; s := s-1;
    REPEAT (* разделение a[L]...a[R] *)
      i := L; j := R; x := a[(L+R) DIV 2];
      REPEAT
        WHILE a[i] < x DO i := i+1 END;
        WHILE x < a[j] DO j := j-1 END;
        IF i <= j THEN
          w := a[i]; a[i] := a[j]; a[j] := w; i := i+1; j := j-1
        END
      UNTIL i > j;
      IF i < R THEN (* запись в стек запроса из правой части *)
        s := s+1; stack[s].L := i; stack[s].R := R
      END;
      R := j (* теперь L и R ограничивают левую часть *)
    UNTIL L >= R
  UNTIL s = 0
END NonRecursiveQuickSort

```

Прогр. 2.11. Нерекурсивная версия Quicksort.

среднее этих ожидаемых значений для всех возможных границ  $x$ .

$$\begin{aligned}
 M &= [Sx: 1 \leq x \leq n: (x-1) \cdot (n-(x-1))/n]/n \\
 &= [Su: 0 \leq u \leq n-1: u \cdot (n-u)]/n^2 \\
 &= n \cdot (n-1)/2n - (2n^2 - 3n + 1)/6n = (n-1/n)/6
 \end{aligned} \tag{2.15}$$

Представим себе, что мы счастливики и нам всегда удастся выбрать в качестве границы медиану, в этом случае каждый процесс разделения расщепляет массив на две половины и для сортировки требуется всего  $\log n$  проходов. В результате общее число сравнений равно  $n \cdot \log n$ , а общее число обменов —  $n \cdot \log(n)/6$ . Нельзя, конечно, ожидать, что мы каждый раз будем выбирать медиану. Вероятность этого составляет только  $1/n$ . Удивительный, однако, факт: средняя производительность Quicksort при случайном выборе границы отличается от упомянутого оптимального варианта лишь коэффициентом  $2 \cdot \ln(2)$ .

Как бы то ни было, но Quicksort присущи и некоторые недостатки. Главный из них — недостаточно высокая производительность при небольших  $n$ , впрочем этим грешат все усовершенствованные методы. Но перед другими усовершенствованными методами этот имеет то преимущество, что для обработки небольших частей в него можно легко включить какой-либо из прямых методов сортировки. Это особенно удобно делать в случае рекурсивной версии программы.

Остается все еще вопрос о самом плохом случае. Как тогда будет работать Quicksort? К несчастью, ответ на этот вопрос неутешителен и демонстрирует одно неприятное свойство Quicksort. Разберем, скажем, тот несчастный случай, когда каждый раз для сравнения выбирается наибольшее из всех значений в указанной части. Тогда на каждом этапе сегмент из  $n$  элементов будет расщепляться на левую часть, состоящую из  $n - 1$  элементов, и правую, состоящую из одного-единственного элемента. В результате потребуется  $n$  (а не  $\log n$ ) разделений и наихудшая производительность метода будет порядка  $n^2$ .

Явно видно, что главное заключается в выборе элемента для сравнения —  $x$ . В нашей редакции им становится средний элемент. Заметим, однако, что почти с тем же успехом можно выбирать первый или последний. В этих случаях хуже всего будет, если массив окажется первоначально уже упорядочен, ведь Quicksort определенно «не любит» такую тривиальную работу и предпочитает иметь дело с неупорядоченными массивами. Выбирая средний элемент, мы как бы затушевываем эту странную особенность Quicksort, поскольку в этом случае первоначально упорядоченный массив будет уже оптимальным вариантом. Таким образом, фактически средняя производительность при выборе среднего элемента чуточку улучшается. Сам Хоар предполагает, что  $x$  надо выбирать случайно, а для небольших выборок, вроде всего трех ключей, останавливаться на медиане [2.12, 2.13]. Такой разумный выбор мало влияет на среднюю производительность Quicksort, но зато значительно ее улучшает (в наихудших случаях). В некотором смысле быстрая сортировка напоминает азартную

игру: всегда следует учитывать, сколько можно проиграть в случае невезения.

Из этих соображений можно извлечь один важный урок, касающийся непосредственно программирования. Что же следует из упомянутого случая самого плохого поведения программы 2.11? Мы ее построили так, что каждое разделение заканчивается образованием правой части, состоящей лишь из единственного элемента, и требование последующей сортировки этой части заносится в стек. Следовательно, максимальное число требований, т. е. общий размер стека требований, равно  $n$ . Конечно, как правило, эта граница не достигается. (Заметим, что с рекурсивной версией программы дело обстоит не лучше, а даже хуже, поскольку системы, допускающие рекурсивные обращения к процедурам, автоматически сохраняют значения локальных переменных и параметров всех активаций таких процедур. Для этого используют некоторый неявный стек.)

Выход из положения заключается в том, что в стек надо прятать требование сортировки для более длинной части и сразу же продолжать разделение меньшей части. В этом случае размер стека  $M$  можно ограничить числом  $\log n$ . Все изменения, которые нужно внести в программу 2.11, сосредоточены в разделе, устанавливающем новые требования. Теперь он выглядит так:

```

IF  $j < R - i$  THEN
  IF  $i < R$  THEN (* запись в стек запроса на сортировку правой части *)
     $s := s + 1$ ;  $stack[s].L := i$ ;  $stack[s].R := R$ 
  END;
   $R := j$  (* продолжение сортировки левой части *)
ELSE
  IF  $L < j$  THEN (* запись в стек запроса на сортировку левой части *)
     $s := s + 1$ ;  $stack[s].L := L$ ;  $stack[s].R := j$ 
  END;
   $L := i$  (* продолжение сортировки правой части *)
END

```

(2 16)

#### 2.3.4. Нахождение медианы

*Медианой* для  $n$  элементов называется элемент, меньший (или равный) половине из  $n$  элементов и больший (или равный) другой половине из  $n$  эле-

ментов. Например, медиана для элементов

16 12 99 95 18 87 10

равна 18. Задача поиска медианы тесно связана с проблемой сортировки, поскольку очевидный метод определения медианы заключается в том, чтобы отсортировать  $n$  элементов, а затем выбрать средний элемент. Однако разделение, выполняемое программой 2.9, позволяет потенциально отыскивать медиану значительно быстрее. Этот прием можно легко обобщить и для поиска среди  $n$  элементов  $k$ -го наименьшего числа. В этом случае поиск медианы — просто частный случай  $k = n/2$ . Алгоритм, предложенный Ч. Хоаром [2.4], работает следующим образом. Сначала применяется операция разделения из Quicksort с  $L = 1$  и  $R = n$ , в качестве разделяющего значения  $x$  берется  $a_k$ . В результате получаем индексы  $i$  и  $j$ , удовлетворяющие таким условиям:

1.  $a_h < x$  для всех  $h < i$
  2.  $a_h > x$  для всех  $h > j$
  3.  $i > j$
- (2.17)

При этом мы сталкиваемся с одним из таких трех случаев:

1. Разделяющее значение  $x$  было слишком мало, и граница между двумя частями лежит ниже нужной величины  $k$ . Процесс разделения повторяется для элементов  $a_i \dots a_R$  (рис. 2.9).

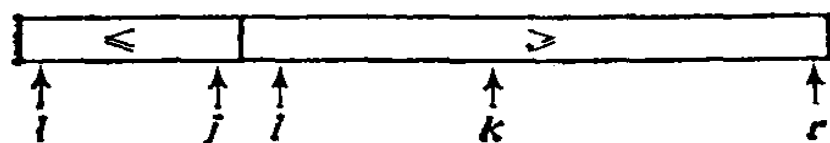


Рис. 2.9. Граница слишком мала.

2. Выбранная граница  $x$  была слишком большой. Операции разделения следует повторить для элементов  $a_L \dots a_j$  (рис. 2.10).

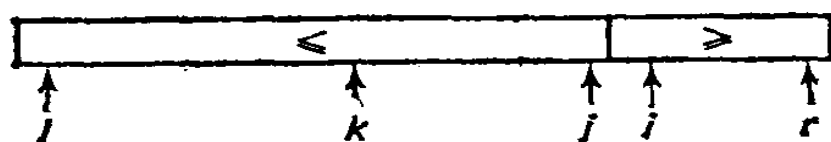


Рис. 2.10. Граница слишком велика.

3.  $j < k < i$ : элемент  $a_k$  разделяет массив на две части в нужной пропорции, следовательно, это то, что нужно (рис. 2.11).



Рис. 2.11. Верная граница.

Процессы деления повторяются до тех пор, пока не возникнет третий случай. Сами итерации описываются следующим фрагментом программы:

```
L := 1; R := n;
WHILE L < R DO
  x := a[k]; разделение (a[L]... a[R]);
  IF j < k THEN L := i END;
  IF k < i THEN R := j END
END
```

(2.18)

За формальным доказательством корректности этого алгоритма мы отсылаем читателя к оригинальной работе самого Хоара. Теперь легко получается и вся программа *Find*. Если предположить, что каждое деление в среднем разбивает часть, где нахо-

```
PROCEDURE Find(k: INTEGER);
  VAR L, R, i, j: index; w, x: item;
BEGIN L := 1; R := n;
  WHILE L < R DO
    x := a[k]; i := L; j := R;
    REPEAT
      WHILE a[i] < x DO i := i + 1 END;
      WHILE x < a[j] DO j := j - 1 END;
      IF i <= j THEN
        w := a[i]; a[i] := a[j]; a[j] := w; i := i + 1; j := j - 1
      END
    UNTIL i > j;
    IF j < k THEN L := i END;
    IF k < i THEN R := j END
  END
END Find
```

Прогр. 2.12. Поиск  $k$ -го наибольшего элемента.



дится желаемая величина, пополам, то число требуемых сравнений равно

$$n + n/2 + n/4 + \dots + 1 \doteq 2n$$

т. е. оно порядка  $n$ . Это и объясняет «мощность» программы *Find* в случаях нахождения медианы и других подобных величин и ее превосходство над прямыми методами, где сначала сортируется все множество кандидатов, а затем уже выбирается  $k$ -й элемент (в лучшем случае на это потребуется порядка  $n \cdot \log(n)$  операций). Однако в самой неблагоприятной ситуации каждый шаг деления будет уменьшать множество кандидатов только на единицу, и в результате потребуется порядка  $n^2$  сравнений. И вновь напомним: едва ли стоит пользоваться этим алгоритмом, если число элементов невелико, скажем порядка 10 \*)

### 2.3.5. Сравнение методов сортировки массивов

Заканчивая наш обзор методов сортировки, мы попытаемся сравнить их эффективность. Как и раньше,  $n$  — число сортируемых элементов, а  $S$  и  $M$  соответственно число необходимых сравнений ключей и число обменов. Для всех прямых методов сортировки можно дать точные аналитические формулы. Они приводятся в табл. 2.8. Столбцы *Min*, *Avg*, *Max* определяют соответственно минимальное, усредненное и максимальное по всем  $n!$  перестановкам из  $n$  элементов значения.

Для усовершенствованных методов нет сколько-либо осмысленных, простых и точных формул. Существование, однако, что в случае сортировки Шелла вычислительные затраты составляют  $c \cdot n^{1.2}$ , а для *Heapsort* и *Quicksort* —  $c \cdot n \cdot \log n$ , где  $c$  — соответствующие коэффициенты.

Эти формулы просто вводят грубую меру для производительности как функции  $n$  и позволяют раз-

---

\*) Автор, как правило, не учитывает в оценках коэффициенты, входящие в оценку производительности. При небольших  $n$  они могут играть большую роль, чем сами приводимые теоретические оценки. Особенно это существенно при использовании рекурсивных процедур. Даже при специальной архитектуре, ориентированной на использование процедур, итеративные методы лучше рекурсивных. — *Прим. перев.*

Таблица 2.8. Сравнение прямых методов сортировки

|                     |       | Min           | Avg                      | Max                   |
|---------------------|-------|---------------|--------------------------|-----------------------|
| Прямое<br>включение | $C =$ | $n-1$         | $(n^2 + n - 2)/4$        | $(n^2 - n)/2 - 1$     |
|                     | $M =$ | $2(n-1)$      | $(n^2 - 9n - 10)/4$      | $(n^2 - 3n - 4)/2$    |
| Прямой<br>выбор     | $C =$ | $(n^2 - n)/2$ | $(n^2 - n)/2$            | $(n^2 - n)/2$         |
|                     | $M =$ | $3(n-1)$      | $n \cdot (\ln n + 0.57)$ | $n^2/4 + 3(n-1)$      |
| Прямой<br>обмен     | $C =$ | $(n^2 - n)/2$ | $(n^2 - n)/2$            | $(n^2 - n)/2$         |
|                     | $M =$ | 0             | $(n^2 - n) \cdot 0.75$   | $(n^2 - n) \cdot 1.5$ |

бить алгоритмы сортировки на примитивные, прямые методы ( $n^2$ ) и на усложненные или «логарифмические» методы ( $n \cdot \log(n)$ ). Однако для практических целей полезно иметь некоторые экспериментальные данные, способные пролить свет на те коэффициенты  $c$ , которыми один метод отличается от другого. Более того, формулы не учитывают других затрат на вычисления, кроме сравнения ключей и переносов элементов, таких, например, как управление циклами и т. д. Ясно, что эти факторы во многом зависят от отдельной вычислительной системы, но тем не менее результаты экспериментов довольно информативны. В табл. 2.9 собраны времена (в секундах) работы, обсуждавшихся выше методов сортировки, реализованных в системе Модуля-2 на персональной ЭВМ Lilith. Три столбца содержат времена сортировки уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке. В начале приводятся цифры для 256 элементов, а ниже — для 2048. Четко прослеживается отличие квадратичных методов от логарифмических. Кроме того, заслуживают внимания следующие особенности.

1. Улучшение двоичного включения по сравнению с прямым включением действительно почти ничего не дает, а в случае упорядоченного массива даже получается отрицательный эффект.

2. Пузырьковая сортировка определенно наихудшая из всех сравниваемых.

Ее усовершенствованная версия, шейкерная сортировка, продолжает оставаться плохой по сравнению

Таблица 2.9. Время работы различных программ сортировки

|                   | Упорядо-<br>ченный | Случай-<br>ный | В обратном<br>порядке |
|-------------------|--------------------|----------------|-----------------------|
| n = 256           |                    |                |                       |
| StraightInsertion | 0.02               | 0.82           | 1.64                  |
| BinaryInsertion   | 0.12               | 0.70           | 1.30                  |
| StraightSelection | 0.94               | 0.96           | 1.18                  |
| BubbleSort        | 1.26               | 2.04           | 2.80                  |
| ShakerSort        | 0.02               | 1.66           | 2.92                  |
| ShellSort         | 0.10               | 0.24           | 0.28                  |
| HeapSort          | 0.20               | 0.20           | 0.20                  |
| QuickSort         | 0.08               | 0.12           | 0.08                  |
| NonRecQuickSort   | 0.08               | 0.12           | 0.08                  |
| StraightMerge     | 0.18               | 0.18           | 0.18                  |
| n = 2048          |                    |                |                       |
| StraightInsertion | 0.22               | 50.74          | 103.80                |
| BinaryInsertion   | 1.16               | 37.66          | 76.06                 |
| StraightSelection | 58.18              | 58.34          | 73.45                 |
| BubbleSort        | 80.18              | 128.84         | 178.66                |
| ShakerSort        | 0.16               | 104.44         | 187.35                |
| ShellSort         | 0.80               | 7.08           | 12.34                 |
| HeapSort          | 2.32               | 2.22           | 2.12                  |
| QuickSort         | 0.72               | 1.22           | 0.76                  |
| NonRecQuickSort   | 0.72               | 1.32           | 0.80                  |
| StraightMerge     | 1.98               | 2.06           | 1.98                  |

с прямым включением и прямым выбором (за исключением патологического случая уже упорядоченного массива).

3. Quicksort лучше в 2—3 раза, чем Heapsort. Она сортирует массив, расположенный в обратном порядке, практически с той же скоростью, что и уже упорядоченный.

## 2.4. СОРТИРОВКА ПОСЛЕДОВАТЕЛЬНОСТЕЙ

### 2.4.1. Прямое слияние

К сожалению, алгоритмы сортировки, приведенные в предыдущем разделе, невозможно применять для данных, которые из-за своего размера не помещаются в оперативной памяти машины и находятся, например, на внешних, последовательных запоминающих устройствах памяти, таких, как ленты или диски.