9. СОРТИРОВКА И ПОИСК ДАННЫХ

Операция упорядочивания данных по возрастанию или убыванию какого-либо признака (*ключа*) в информатике называется *сортировкой*. По подсчетам специалистов, на сортировку данных компьютеры всего мира тратят около 25% времени работы. В свою очередь, например, в трансляторах до 50% времени работы занимают операции, связанные с поиском данных.

Задача сортировки. Пусть элементы массива x[1], x[2],..., x[n] - целые числа (это не уменьшает общности). Требуется переставить эти элементы по неубыванию - так, чтобы $x[1] \le x[2] \le ... \le x[n]$, (или невозрастанию). Среди чисел x[1], ..., x[n] могут быть равные.

Сортировка устойчива, если не изменяется относительное расположение равных элементов. Это желательно, если они уже отсортированы по другим ключам - значениям каких-либо других характеристик элементов массива.

Сортировка выполняется "*на том же месте*" (in situ), если не требуются вспомогательные массивы (дополнительная память объемом порядка n).

Усилиями специалистов всего мира разработаны десятки алгоритмов сортировки и поиска данных, знакомство с которыми играет важную роль в образовании программиста. Очень полезно также изучение использованных в них разнообразных методов программирования. Рассмотрим кратко некоторые из этих алгоритмов.

Из многочисленных методов сортировки выделяются две важные группы: простые "квадратичные" алгоритмы с временем работы порядка $O(n^2)$ и усовершенствованные "логарифмические" алгоритмы, требующие времени порядка O(n log n). Иногда возможна сортировка подсчетом количества значений за время порядка n (задача 7.3 б).

Утверждение, что время работы алгоритма или необходимый объем памяти имеет порядок f(n) или O(f(n)) (читается "О-большое от f(n)") означает, что эта величина не превосходит $C \cdot f(n)$ для некоторого C и для всех n, где C – ненулевая константа, n - количество элементов данных, участвующих в сортировке или поиске.

Логарифм числа, по сути дела, представляет собой количество цифр этого числа, и логарифм большого числа во много раз меньше числа, т. е. $n \cdot \log n \ll n \cdot n = n^2$. Основание логарифма не влияет на характер зависимости времени сортировки от размера сортируемого массива, поскольку при изменении основания коэффициент C лишь умножается на логарифм старого основания по новому основанию, т. к. $\log_b x = \log_b a \cdot \log_a x$.

9.1. Простые методы сортировки

К простым (прямым) методам, обеспечивающим время сортировки порядка n^2, относятся алгоритмы простой вставки, бинарной вставки,

простого выбора и простого обмена. Простые методы выполняют сортировку на том же месте и удобны для малых массивов.

Метод простой вставки. На каждом этапе сортировки, начиная с k=2, текущий элемент x[k] вставляется в такое место среди уже отсортированных предыдущих k-1 элементов, чтобы упорядоченность сохранилась. Таким образом, в основу алгоритма 7.1 положен инвариант цикла "первые k-1 элементов упорядочены: x[1] ≤ x[2] ≤ ... ≤ x[k-1]". Инвариант цикла содержит основную идею алгоритма и его удобно записать в виде комментария после заголовка цикла.

Алгоритм 9.1. Сортировка простой вставкой

```
for ( k=2; k<=n; k++)
//    x[1] ... x[k-1] упорядочены:  x[1] ≤  x[2] ≤  ... ≤ x[k-1]
{  t = x[k];
   Вставить t среди x[1] ... x[k-1] перебором;  // просеивание t
}
```

Вставку очередного значения t на нужное место можно выполнить так:

```
for(j=k-1; j>0  &&  t<x[j]; j--)  x[j+1]=x[j];    // просеивание t
x[j+1] = t;
```

Выполнение этого цикла можно ускорить стандартным приемом *барьера* (устраняющим краевой эффект). Добавим перед массивом дополнительный элемент x[0], равный x[k]. В этом случае в цикле можно не проверять условие j > 0: элемент x[0] сыграет роль “барьера”, который не позволит индексу j выйти за пределы массива, т. к. при j=0 нарушится условие x[0] < x[j] и цикл остановится (прогр. 7.1).

Программа 9.1. Сортировка простой вставкой

```
//      Сортировка массива  x по неубыванию простой вставкой
void srt_vstvk (int x[], int n)
{  int k, j;
   for (k=2;  k<=n;  k++)
    // x[1]...x[k-1] упорядочены:  x[1] ≤  x[2] ≤  ... ≤ x[k-1]
    {  x[0] = x[k];                            // x[0] - барьер
       for(j=k-1; x[0]<x[j]; j--) x[j+1]=x[j]; //просеивание x[0]
       x[j+1] = x[0];
   }
}
```

Сортировка простой вставкой устойчива: если среди предыдущих элементов есть значение, равное x[k], то цикл поиска нужного места останавливается при x[j]=x[k] и x[k] вставляется после x[j]. Таким образом сохраняется относительный порядок равных элементов.

Время сортировки пропорционально количеству основных операций: сравнений и присваиваний. В худшем случае, когда элементы массива первоначально убывают, для вставки очередного элемента x[k] требуется

передвинуть все предыдущие элементы: k сравнений и k присваиваний, а их общее количество равно сумме арифметической прогрессии

$$2 + 3 + \ldots + n = (n+2)(n-1)/2.$$

В среднем придется передвигать около половины предшествующих элементов. Таким образом сортировка простой вставкой и в худшем случае и в среднем требует времени порядка n^2.

Метод бинарной вставки. Некоторым улучшением является поиск места x[R] для вставки t = x[k] методом деления пополам (дихотомии) в программе 9.2:

```
//Дихотомический поиск x[R] для вставки t среди x[1]...x[k-1]
  L=1; R=k;
  while (L < R)
  // x[1]≤...≤ x[L-2]≤ x[L-1] ≤ t < x[R+1]≤ x[R+2]≤...≤ x[k]
  {  m=(L+R)/2;
     if (x[m] <= t) L=m+1; else R=m;
  }
```

Поиск производится в области от x[L] до x[R]. Инвариантом цикла является условие: “в отрезке массива x[1], x[2], …, x[k] все элементы левее области поиска (с индексами меньше L) не превышают t, а все элементы правее области поиска (с индексами больше R) превышают t ”. Другими словами, в отрезке x[1], x[2], …, x[k] отсутствуют элементы, нарушающие это условие.

Программа 9.2. Сортировка бинарной вставкой (Дж. Мочли, 1946 г.)

```
//  Сортировка массива  x  по неубыванию бинарной вставкой
void srt_bivst (int x[], int n)
{  int k, j, t, m,
       L, R;   // Левая и правая границы индексов области поиска
   for (k=2;  k<=n;  k++)
   // x[1]...x[k-1] упорядочены:  x[1] ≤  x[2] ≤  ... ≤ x[k-1]
   { t=x[k];
    //Дихотомический поиск x[R] для вставки t среди x[1...x[k-1]
     L=1; R=k;
     while (L < R)
     // x[1]≤...≤ x[L-2]≤ x[L-1] ≤  t < x[R+1]≤ x[R+2]≤...≤ x[k]
     {  m=(L+R)/2;
        if (x[m] <= t) L=m+1; else R=m;
     }
     for (j=k; j>R; j--)  x[j] = x[j-1];  // сдвиг для вставки
     x[R] = t;
   }
}
```

Первоначально при L=1, R=k инвариант справедлив потому, что в этом отрезке вообще нет элементов с индексами меньше L или больше R.

Поскольку L < R, а при целочисленном делении дробная часть частного отбрасывается, то для m соблюдается условие $L \le m < R$, т. е. $L < m+1$. Поэтому при каждом повторении цикла либо увеличивается L, либо уменьшается R, область поиска сужается и обязательно сводится к одному

элементу при L=R. Легко проверить, что инвариант сохраняется при выполнении любой ветви оператора **if,** следовательно, и при выходе из цикла. Таким образом, место вставки находится правильно.

Метод бинарной вставки лучше простой вставки только тем, что уменьшается число сравнений при поиске места, а количество присваиваний при сдвиге и время сортировки остаются большими - порядка n^2.

Сортировка простым выбором основана на таком инварианте: “k-1 минимальных элементов массива x находятся на своих местах: x[1] ≤ x[2] ≤ ... ≤ x[k-1]”. На каждом этапе из неупорядоченной части массива от x[k] до x[n] выбирается минимальный элемент x[jmin] и обменивается с элементом x[k]. Таким образом упорядоченная часть массива удлиняется на один элемент (алг. 9.3).

Алгоритм 9.3. Сортировка простым выбором (поиском минимума)

```
for(k=1; k<n; k++)
//    k-1 минимальных элементов массива x на своих местах
{  jmin = индекс (min(x[k], ..., x[n]);
   Обменять значения x[k] и x[jmin];
}
```

Программа9.3.

```
//  Сортировка по неубыванию простым выбором  (поиском минимума)
void srt_vybor (int x[], int n)
{  int  k, j, jmin, min;
   for (k=1;  k < n;   k++)
   //   k-1 минимальных элементов массива x на своих местах
   {   min = x[k];    jmin = k;
       for (j=k+1; j<=n; j++)
          if (x[j]< min)  { min=x[j]; jmin=j; }
       x[jmin]=x[k];  x[k]= min;   //обмен x[k] <--> x[jmin]
   }
}
```

Сортировка простым обменом (методом пузырька) основана на таком же инварианте как и простой выбор: “k-1 минимальных элементов массива x находятся на своих местах: x[1] ≤ x[2] ≤, ... ≤ x[k-1]”. На каждом этапе неупорядоченная часть массива просматривается от x[n] по убыванию индексов до x[k] и сортируется каждая пара соседних элементов x[j] и x[j-1]: если их порядок нарушен, то они обменивается местами. В результате меньшие значения, “как пузырьки в воде, постепенно поднимаются к началу массива”, а минимальное из просмотренных значений окажется в элементе x[k]. Таким образом упорядоченная часть массива удлиняется на один элемент (алг. 9.4).

Алгоритм 9.4. Сортировка простым обменом (пузырьком)

```
for (k=1;  k < n;  k++)
```

```
//    k-1 минимальных элементов массива x на своих местах
for (j=n;  j>k;  j--)
   if (x[j-1] > x[j])  Обменять x[j] и x[j-1];
```

Программа 9.4.

```
//   Сортировка по возрастанию простым обменом (пузырьком)
void srt_obmen (int x[], int n)
{  int  k, j, t;
   for (k=1; k<n; k++)
     // k-1 минимальных элементов массива x на своих местах
     for (j=n; j>k; j--)
       if (x[j-1]> x[j])                      // Порядок нарушен
       { t=x[j];  x[j]=x[j-1];  x[j-1]=t; }  // x[j] <--> x[j-1]
}
```

Наиболее быстрым из простых методов является простой выбор, самым медленным - простой обмен.

9.2. Методы быстрой сортировки

Известно несколько усовершенствованных методов, обеспечивающих время сортировки порядка n·log n. Среди них - так называемая "быстрая" сортировка, пирамидальная сортировка и метод слияния. Для малых массивов улучшенные алгоритмы уступают по времени простым методам, но для больших массивов они более эффективны.

7.2.1. Быстрая сортировка

Метод “быстрой” сортировки – QuickSort (Ч. Э. Р. Хоар, 1962), названный так его автором за высокую скорость, действительно, на практике один из самых быстрых по среднему времени работы.

В сортируемом массиве выбирается значение s некоторого элемента, которое служит барьером. Элементы переставляются так, что массив разделяется на три части: меньшие чем s, равные s и большие чем s. При этом значение s окажется на том месте, где оно должно быть после сортировки. Остается отсортировать первую и третью части: это можно делать тем же или другим способом.

Одна из рекурсивных версий быстрой сортировки – в программе 9.5.

Программа 9.5. Рекурсивная быстрая сортировка

```
// Рекурсивная быстрая сортировка по неубыванию x[L]...x[R]
// Методы: разделение и обмен
void RecQuickSort (int x[], int L, int R)
{  int i, j, s, w;
```

```
    i=L;  j=R;  s=x[(L+R)/2];
    while (i<=j)
    //         x[L],...,x[i-1] <= s <= x[j+1],...,x[R]
    {  if(s > x[i]) i++;                 // s <= x[i]
       else if(x[j]> s) j--;              // x[j] <= s
       else { w=x[i]; x[i]=x[j]; x[j]=w;  i++;  j--; }
    }
    //      x[L], ..., x[i-1] <= s <= x[j+1], ..., x[R]
    if (j > L) RecQuickSort(x,L,j);      //Участок > 1 - сортируем
    if (i < R) RecQuickSort(x,i,R);      //Участок > 1 - сортируем

}
```

Здесь от рекурсии можно избавиться, используя *стек отложенных заданий*. Если при решении задачи возникают подзадачи, сразу решать можно не более одной из них, а остальные необходимо где-то запомнить. Для этой цели можно использовать стек отложенных заданий, элементами которого будут заказы на решение подзадач. Заказы помещаются в стек в порядке обратном тому, в котором следует их выполнять: самый срочный - в вершине стека.

Элементы стека удобно записывать в массив, запоминая индекс последнего элемента – вершины стека. Этот индекс называют *указателем стека*.

Если элемент стека состоит из нескольких величин, каждая из них помещается в свой массив с одним и тем же индексом вершины. Такие массивы называют *параллельными*, т. к. на рисунках их удобно изображать рядом, “параллельно”.

В данном случае в стек кладутся пары (L, R), интерпретируемые как отложенные задания на сортировку участков массива от x[L] до x[R]. Участки всех эти заказов не пересекаются, поэтому размер стека не превысит n.

Чтобы ограничиться стеком логарифмической глубины, будем помещать в стек больший из двух возникающих заказов, а меньший участок сразу сортировать. При использовании рекурсии также выгодно в первую очередь сортировать меньший участок для уменьшения необходимого стека.

Пусть f(n) - максимальная глубина стека, необходимая для сортировки таким способом массива не более чем из n элементов. Оценим f(n) сверху. После разбиения массива на два участка сначала сортируется более короткий, а заказ на более длинный участок помещается в стек.

Максимальная длина короткого участка равна n/2, и для его сортировки нужен стек глубиной f(n/2). Еще в одном элементе содержится заказ на более длинный участок, и общая глубина стека не превысит f(n/2)+1.

Затем сортируется более длинный участок размером не более n-1, так что

$$f(n) \le \max(f(n/2)+1, f(n-1)). \qquad (7.1)$$

Отсюда методом индукции можно доказать, что f(n) имеет порядок O(log n).

Этот вариант реализован в программе 9.6.

Программа 9.6. Нерекурсивная быстрая сортировка

```
//Нерекурсивная быстрая сортировка по неубыванию x[L]...x[R]
//с экономией памяти для стека
void QuickSort (int x[], int n)
{ int i, j, s, w, L, R, ist, stL[NMAX], stR[NMAX];
  ist=1;  L=1;  R=n;     // (stL[1], stR[1]) - фиктивный заказ
  do
  {  i=L;  j=R;  s=x[(L+R)/2];
     do //  x[L],...,x[i-1] <= s <= x[j+1],...,x[R]
     {  while (s > x[i]) i++;              // s <= x[i]
        while (x[j]> s)  j--;              // x[j] <= s
        if(i<=j)  { w=x[i]; x[i]=x[j]; x[j]=w; i++;  j--; }
     } while (i <= j);
     //       x[L], ..., x[i-1]  <=  s  <=  x[j+1], ..., x[R]
     if (j-L > R-i)  // участок x[i]… x[R] короче чем x[L]… x[j]
     { stL[++ist]=L;  stR[ist]=j;    //Стек <==(L,j)
       if(i<R) L=i;                 // Короткий > 1 - сортируем
       else {L=stL[ist];R=stR[ist--];}       //Стек ==>(L,R)
     }else
     { if(i<R) { stL[++ist]=i;  stR[ist]=R; } //Стек<==(i,R)
       if(L<j) R=j;                 // Короткий > 1 - сортируем
       else {L=stL[ist]; R=stR[ist--];}      //Стек ==>(L,R)
     }
  } while (ist>0);
}
```

Время работы алгоритма быстрой сортировки - случайная величина. Можно доказать, что в среднем он работает не больше $C \cdot n \cdot \log(n)$, но в худшем случае требует времени порядка n^2.

9.2.2. Пирамидальная сортировка

В пирамидальной сортировке (Дж. Уильямс и Р.У. Флойд, 1964) используется изящный метод выбора элементов из дерева. На первом этапе из элементов сортируемого массива x[1], x[2], ..., x[n] строится бинарное дерево (называемое пирамидой), хранимое с помощью адресной арифметики в этом же массиве ("на том же месте") без использования дополнительной

памяти. На втором этапе из пирамиды один за другим выбираются максимальные элементы и помещаются в упорядоченную часть массива, заполняемую от конца массива к началу.

Пусть x[1], ..., x[k], … - массив, подлежащий сортировке. Этот массив или его часть до индекса k можно рассматривать как *дерево*, *вершинами* (*узлами*) которого являются элементы массива. Будем говорить, что число x[j] находится в узле j. *Корень* дерева - число x[1] - находится в узле 1 (рис. 9.1).

Из корня вниз выходят линии в два узла – *сыновья* корня, из каждого сына - в два других узла – их сыновья и так далее. Узел считается *отцом* своих сыновей. Такое дерево, в котором узел имеет не более двух сыновей, называется *двоичным* или *бинарным деревом*.

Отцом узла j является узел j / 2 (деление здесь целочисленное с отбрасыванием дробной части частного). Узел j может иметь сыновей 2*j и 2*j+1. Если оба этих индекса больше k, то сыновей нет; такая вершина называется *листом*. Если 2*j = k, то узел j имеет одного сына 2*j. Таким образом, зная индекс узла, можно передвигаться от него к другим узлам вниз и вверх по дереву.

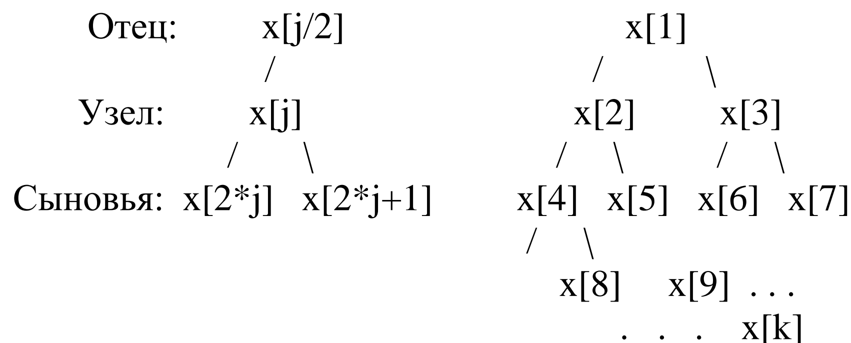


Рис. 9.1. Представление бинарного дерева из k элементов
с помощью адресной арифметики

Подобный метод представления данных, когда существует закономерность, позволяющая вычислять расположение элементов, называется *адресной арифметикой*. В данном случае бинарное дерево представляется в виде массива с помощью адресной арифметики.

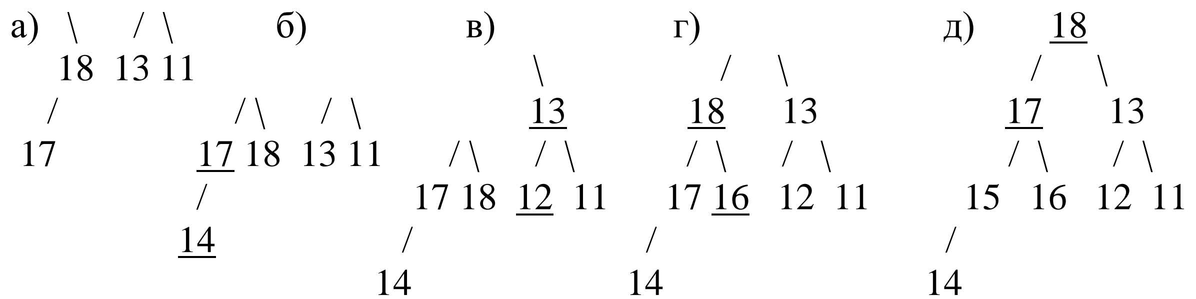
Для каждого i из 1, 2, ..., k рассмотрим i-поддерево с корнем в i. Оно содержит узел i и всех его *потомков*: сыновей, внуков и т. д., пока не выйдем из отрезка с индексами от L до R. Узел с индексом i, назовем *пирамидальным*, если его значение x[i] является максимальным элементом i-поддерева; i-поддерево назовем пирамидальным, если все его узлы пирамидальны. В частности, любой лист (узел с таким индексом i, что 2*i > k) образует пирамидальное поддерево из одного узла.

Отрезок массива x[L], x[L+1], ..., x[R] назовем *пирамидой*, если все его элементы являются пирамидальными узлами, т. е. удовлетворяют условиям:

$$x[i] \ge x[2*i], \quad \text{если } 2*i \le R,$$
$$\text{и} \quad x[i] \ge x[2*i+1], \quad \text{если } 2*i+1 \le R, \qquad (9.2)$$
$$\text{где } L \le i \le R.$$

Пирамида содержит одно дерево или несколько деревьев – *лес*. В частности, вторая половина массива x[n/2+1], ..., x[n] состоит только из листьев и поэтому образует пирамиду. Если все дерево x[1], ..., x[n] образует пирамиду, то в корне дерева x[1] находится максимальное значение массива.

Первоначально пирамиду с границами L=n/2+1 и R=n образует вторая половина массива x[n/2+1], ..., x[n]. Затем в пирамиду включаются элемент за элементом за счет уменьшения на 1 ее левой границы L, пока пирамида не охватит весь массив.




```
        Обмены          Шаг                      ←| Пирамида        L  R
                                j =  1   2   3   4      5   6  7   8
Исходный массив:   а)    x[j] = 15  16  12  14 | 18  13  11  17     5  8
   x[4] <--> x[8]   б)    x[j] =             | 17    <------>   14     4
   x[3] <--> x[6]   в)    x[j] =          | 13   <--->  12              3
   x[2] <--> x[5]   г)    x[j] =       | 18  <---->   16                2
   x[1] <--> x[2]   д)    x[j] =  18 - 15 .                              1
   x[2] <--> x[4]         x[j] = |     17 <-->15
        Пирамида:         x[j] =  18  17  13  15    16  12  11  14
```

Рис. 9.2. Шаги построения пирамиды

После каждого продвижения левой границы требуется восстановить пирамидальность отрезка, которая может нарушиться в новом узле.

Получим следующий алгоритм:

```
// 1. Построение пирамиды x[1]...x[n]
L=n/2+1;  R=n;
//             x[L]...x[R] - пирамида
```

```
   while (L > 1)  { L--;  Восстановить пирамиду (L,R); }
```

Восстановление пирамидальности отрезка x[L], ..., x[R] в узле L требуется и на втором этапе сортировки, а поэтому оформляется как подпрограмма.

Значение нового узла обменивается местами с большим из его сыновей и так просеивается вниз по дереву за время порядка log n, пока не остановится в пирамидальном узле, который либо является листом, либо не уступает по значению своим сыновьям (i, j - индексы текущего узла и его большего сына):

```
// Просеивание: восстановление пирамидальности x[L]...x[R]
   i=L;  j = индекс большего сына (x[i]);
   //      x[L]...x[R] - пирамида везде, кроме, возможно, x[i]
   while (у x[i] есть сын  &&  больший сын x[j] > x[i])
   {  Обменять местами x[i] и x[j];   i=j;
      j = индекс большего сына (x[i]);
   }
```

На рис. 9.2 показана трассировочная таблица – изменившиеся значения сортируемого массива x[1], ..., x[n], n=8, после каждого шага алгоритма построения пирамиды x[1], x[2], ..., x[n], включающей весь массив. Вертикальная черта отделяет уже построенную пирамиду от остальной части массива. Подчеркнуты значения, участвующие в обмене.

На втором этапе сортировки начало массива занимает пирамида, а за ее правой границей R - уже упорядоченная часть массива (первоначально пустая), которая заполняется и растет от конца к началу массива (по убыванию элементов) благодаря уменьшению R, пока не охватит весь массив (рис. 9.3).

На каждом шаге максимальный элемент пирамиды x[1] обменивается местами с ее правым элементом x[R] и таким образом попадает на свое место в уже упорядоченную часть массива, после чего исключается из пирамиды благодаря уменьшению R на 1. Затем с помощью описанной выше подпрограммы просеивания восстанавливается пирамидальность дерева x[1], ..., x[R], возможно, нарушенная новым значением его корня x[1] (прогр. 9.7).

Обмены	Шаг	Пирамида ← | Упорядоч. Часть									L	R
		j =	1	2	3	4	5	6	7	8		
Исходная пирамида:	д)	x[j] =	18	17	13	15	16	12	11	14 |	1	8
x[1] <--> x[8]	е)	x[j] =	14		<--------------			-------------->		| 18		7
x[1] <--> x[2]		x[j] =	17 -	14						|		

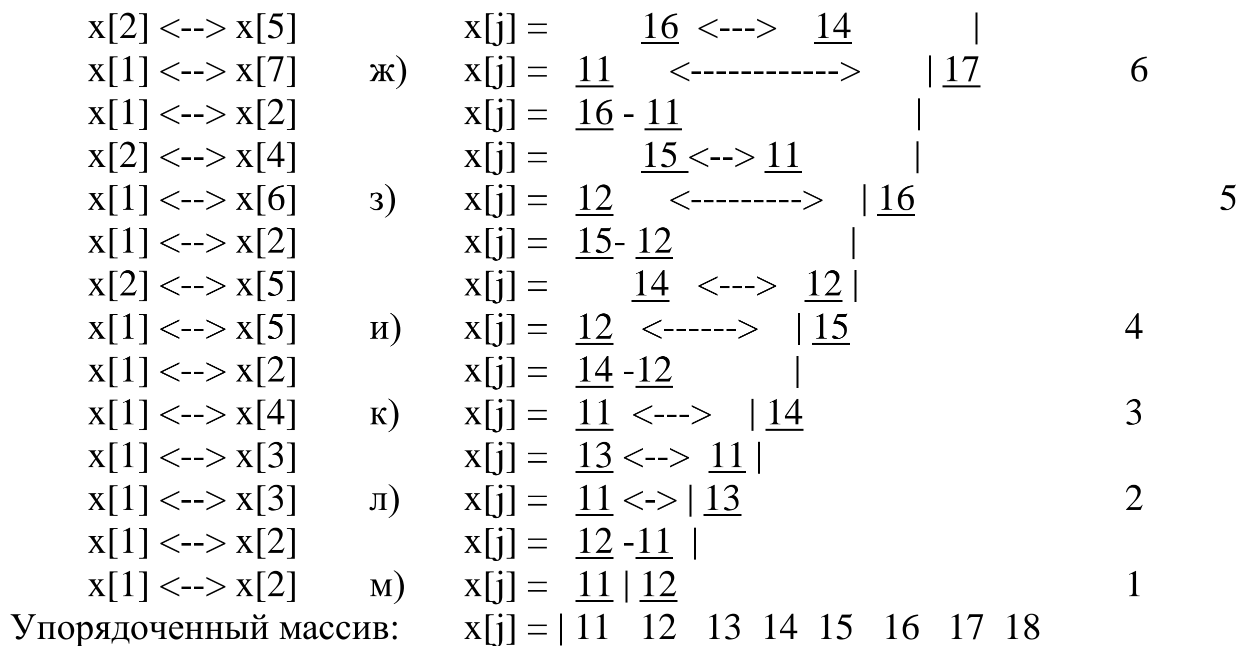


Рис. 9.3. Шаги расширения упорядоченной части массива

Программа 9.7. Пирамидальная сортировка (сортировка кучей)

```
// Пирамидальная сортировка
// Методы: бинарное дерево, адресная арифметика

void sift (int x[], int L, int R)
// Просеивание (восстановление пирамидальности) x[L]...x[R]
//          x[L]...x[R] - пирамида везде, кроме, возможно, x[L]
{  int i, z,      // z=x[i] - текущий узел
       j;         // x[j]=max(x[2i],x[2i+1]) больший сын тек.узла
   i=L;  j=2*L;  z=x[L];
   if (j<R && x[j]<x[j+1]) j++;  //j=индекс(max(x[2i],x[2i+1])
   // x[L]...x[R] - пирамида везде, кроме, возможно, x[i]=z
   while (j<=R && x[j]>z)      // у x[i] есть больший сын > x[i]
   {  x[i]=x[j]; x[j]=z; i=j; j=2*j;  //x[i]<-->x[j]; i=j; j=2j;
      if (j<R && x[j]<x[j+1]) j++;    //j=индекс(max сын(x[i])
   }
}



void HeapSort (int x[], int n)
//              Пирамидальная сортировка (сортировка кучей)
{ int L, R, z;
  // 1. Построение пирамиды x[1]...x[n]
  L=(n/2)+1;  R=n;
  //    x[L]...x[R] - пирамида (для L=(n/2)+1,n/2,...2,1)
  while (L > 1)  { L--;  sift(x,L,R); }
  //2.Выбор из пирамиды максимальных значений и запись с конца x
```

```
  //       x[L]...x[R]-пирамида; x[L]...x[R]<=x[R+1]<=...<=x[n]
  while (R > 1)
  {  z=x[1]; x[1]=x[R]; x[R]=z;    //max(x[1]...x[R]) в конец x
     R--;  sift(x,L,R);
  }
}
```

Каждый из двух этапов и сортировка требуют времени порядка n·log n.

9.2.3. Сортировка слиянием

Сортировка слиянием (Джон фон Нейман, 1945 г.) основана на возможности объединить (слить) за время порядка n в один упорядоченный массив два или более упорядоченных массива (или частей, отрезков массива) суммарной длины n.

Метод простого слияния. Первоначально сливаемые отрезки состоят из одного элемента, а после этапа слияния из каждой пары таких отрезков получится отрезок длины 2. Объединение пар отрезков длины 2 даст упорядоченные отрезки длины 4 и т. д. На каждом этапе слияния длина получаемых отрезков удваивается, и не позднее чем через $\log_2 n$ этапов образуется один упорядоченный отрезок длины n, охватывающий весь массив.

Поскольку длительность каждого этапа порядка n, общее время сортировки даже в худшем случае будет порядка n·log n.

Сортировка естественным слиянием. Используются упорядоченные отрезки исходного массива x, длина которых может превышать 1. Отрезок, просматриваемый по возрастанию индекса от начала массива, сливается с отрезком, просматриваемым по убыванию индекса от конца массива. Результат записывается в начало вспомогательного массива длины n.

Результат слияния второй пары отрезков записывается по убыванию индексов в конец вспомогательного массива y и т. д.

На следующем этапе массивы меняются ролями: сливаются пары отрезков, читаемых от начала и конца вспомогательного массива, а результаты слияния каждой пары попеременно записываются в начало и конец исходного массива. В программе 9.8 используется вспомогательная подпрограмма, реализующая один этап сортировки.

Программа 9.8. Сортировка естественным слиянием

```
/*      Этап сортировки естественным слиянием x --> y        */
/*   Значение: 1 - сортировка закончена, 0 - нет             */
int etap (int x[], int y[], int n)
{  int i, j, k, kpr, d, esti, estj, r;
```

```
   i=1; j=n; k=1; kpr=n; d=1;
   while (i<=j)
   { /*Слить упоряд. отрезки x слева и справа и записать в y*/
      esti=1; estj=1;           /* оба отрезка продолжаются */
      while (i<=j && esti && estj)
      {   if x[i]<=x[j]) { y[k]= x[i++]; esti = x[i]>=x[i-1]; }
          else           { y[k]= x[j--]; estj = x[j]>=x[j+1]; }
          k += d;
      }
      while (i<=j && esti)      /* остаток левого отрезка   */
      {  y[k]=x[i++];  esti = x[i]>=x[i-1];  k += d; }
      while (i<=j && estj)      /* остаток правого отрезка  */
      {  y[k]=x[j--];  estj = x[j]>=x[j+1];  k += d; }
      r=k; k=kpr; kpr=r; d=-d;  /* смена направления записи */
   }
   return kpr>n;             /* в конце сортировки kpr==n+1 */
}

/*      Сортировка x по неубыванию естественным слиянием    */
/*  Значение 1 - результат в массиве x, 0 - результат в y   */
int est_sliyan (int x[], int y[], int n)
{  while (! etap(x,y,n))           /* этап переписи из x в y */
     if (etap(y,x,n)) return 1;    /* этап переписи из y в x */
   return 0;                       /* результат в массиве y  */
}
```

При четном числе этапов результат сортировки окажется в исходном массиве, а при нечетном - во вспомогательном массиве. Недостатком метода слияния является необходимость вспомогательной памяти объемом не менее сортируемого массива.

Поскольку чтение и запись элементов происходит всегда последовательно, вместо массивов можно использовать файлы, т. е. метод слияния пригоден не только для *внутренней сортировки* массивов в оперативной памяти, но и для *внешней сортировки* - сортировки файлов во внешней памяти (в том числе на магнитных лентах, где возможно только последовательное чтение и запись данных). Это является важным достоинством.

Сортировка слиянием не требует, чтобы весь сортируемый массив помещался в оперативной памяти. Можно сначала отсортировать такие куски, которые помещаются в памяти, а затем объединять полученные файлы.

В случае внешней сортировки на каждом этапе сливаемые отрезки читаются из двух или более входных файлов, а получаемые отрезки записываются по очереди в такое же количество выходных файлов. На следующем этапе входные и выходные файлы меняются ролями.

9.2.4. Задачи, связанные с сортировкой

Родственная сортировке задача - поиск *медианы массива*, т. е. такого значения элемента, который имел бы индекс n / 2, если массив из n элементов отсортировать по неубыванию. Более общей задачей является поиск k-го наименьшего элемента, который оказался бы на k–м месте после сортировки массива.

Эффективным методом поиска k-го наименьшего элемента за среднее время порядка $n \cdot \log n$ является упрощенный алгоритм быстрой сортировки: после каждого разделения массива на части сортируется только тот его отрезок, в котором находится X[k], до тех пор, пока он не уменьшится до одного элемента, который и будет равняться искомому значению (прогр. 9.9).

Программа 9.9. Поиск k-го наименьшего элемента

```
// Значение k-го наименьшего элемента массива x[1]...x[n]
// значения x переставляются, k-е значение - на свое место в
x[k])
int k_element (int x[], int n, int k)
{  int i, j, s, w, L, R;
   L=1; R=n;
   while (L<R)     // k-й наименьший элемент на отрезке x[L..R]
   {  i=L; j=R; s=x[k];
      do //   x[L], ..., x[i-1] <= s <= x[j+1], ..., x[R]
      {  while (s>x[i]) i++;                     // s <= x[i]
         while (x[j]>s) j--;                     // x[j] <= s
         if (i<=j)  { w=x[i]; x[i]=x[j]; x[j]=w; i++; j--; }
      } while (i<=j);
      // x[L], ..., x[i-1] <= s <= x[j+1], ..., x[R]; j < i
      if (j<k) L=i;       // k-й элемент на отрезке x[i]...x[R]
      if (k<i) R=j;       // k-й элемент на отрезке x[L]...x[j]
   }
   return x[k];
}
```

Упражнения и задачи

9.1. Составить тесты и трассировочные таблицы выполнения алгоритмов.

а) Программа 9.1. Сортировка простой вставкой.
б) Программа 9.2. Сортировка бинарной вставкой.
в) Программа 9.3. Сортировка простым выбором.
г) Программа 9.4. Сортировка простым обменом.
д) Программа 9.5. Рекурсивная быстрая сортировка.
е) Программа 9.6. Нерекурсивная быстрая сортировка.
ж) Программа 9.7. Пирамидальная сортировка (сортировка кучей).
з) Программа 9.8. Сортировка естественным слиянием.
и) Программа 9.9. Поиск k-го наименьшего элемента

9.2. Изменить

а) программы 9.1 – 9.9, чтобы выполнялась сортировка по невозрастанию;

б) программу 9.3, чтобы выбирался не минимальный, а максимальный элемент;

9.3. Составить подпрограмму сортировки массива заданным методом.

а) Сортировка *подсчетом индекса.* Для каждого элемента подсчитать количество меньших или равных ему элементов, которое и определит его индекс в упорядоченном массиве.

б) Сортировка *подсчетом количества значений.* В сортируемом массиве подсчитывается количество экземпляров каждого возможного значения (значения должны быть целыми числами в диапазоне от А до В). Необходим массив счетчиков, размер которого равен количеству возможных значений. Для элемента x[i] сортируемого массива увеличивается на 1 счетчик с индексом x[i]-A. Сортируемый массив заполняется заново в требуемом порядке нужными значениями в подсчитанных количествах. Время сортировки порядка n.

в) Сортировка символьных строк в лексикографическом порядке (как в словарях).

г) Сортировка строк из символов кириллицы (русского алфавита) в лексикографическом порядке (как в словарях).

д) Сортировка простым обменом (программа 9.4) с исключением лишних просмотров, когда при очередном просмотре не было перестановок (массив уже отсортирован).

е) Сортировка простым обменом (программа 9.4) с чередованием направления просмотров (вперед – назад), чтобы избежать асимметрии, когда, например, массив 1, 3, 4, 7, 8, 9 сортируется за один просмотр, а для массива 4, 7, 8, 9, 1, 3 требуется n-1 = 5 просмотров.

ж) *Шейкер-сортировка* – сочетание методов д) и е).

з) С*ортировка с убывающим шагом* – метод Д. Л. Шелла (1959) – усовершенствованная сортировка обменом. Сортировка обменом выполняется сначала с шагом n/2, потом n/4, и т. д. до 1. Можно использовать другую последовательность шагов, лучшим вариантов считается убывающая последовательность взаимно простых чисел (не имеющих общих делителей кроме 1). Время сортировки порядка $n^{1.2}$.

и) *Цифровая сортировка*. Сначала элементы массива сортируются по младшей цифре (в системе с любым основанием), затем по второй цифре справа и т. д. до старшей цифры (сортировка должна быть устойчивой – сохранять относительное расположение элементов с равными ключами). После i шагов i-разрядные концы чисел расположены в правильном порядке (доказывается индукцией по i).

к) *Карманная сортировка*. При цифровой сортировке в каждом разряде вместо сравнений цифр во вспомогательный массив переписываются сначала числа с цифрой 0 в данном разряде, затем с цифрой 1 и т. д. Отводится свой участок массива - «карман» - для чисел с цифрой 0 в данном разряде, цифрой 1 и т. д. Время сортировки будет порядка $k \cdot n$, где k – количество разрядов.

9.4. Найти количество различных чисел среди элементов данного массива. Число действий порядка $n \cdot \log n$.