Тема 5. Сортировка

5.4. Сортировка выбором

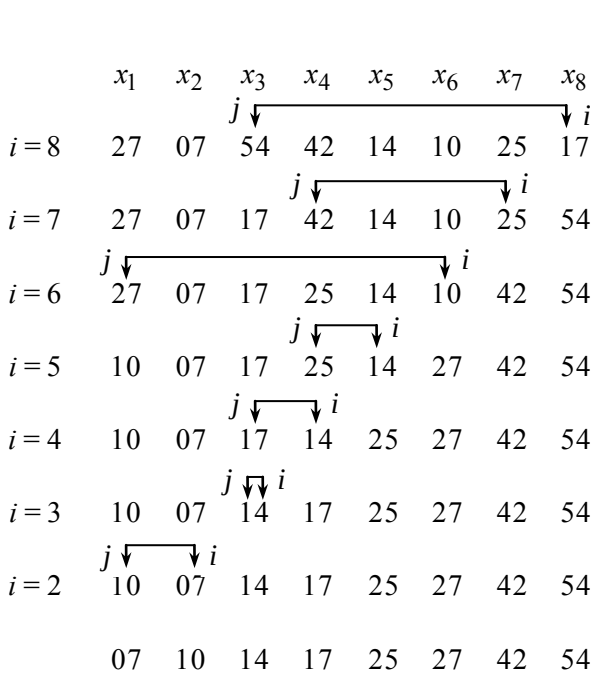
Общая идея сортировки выбором состоит в том, что на каждом шаге $i = 1, 2, \ldots, n$ осуществляется поиск i-го наибольшего (наименьшего) имени, которое затем помещается на свою окончательную позицию. Рассмотрим два метода такой сортировки, которые отличаются способом нахождения i-го наибольшего (наименьшего) имени.

5.4.1. Простая сортировка выбором

Простая сортировка выбором представлена алгоритмом 5.6, в котором *i*-е наибольшее имя находится очевидным способом просмотром оставшихся $n - i + 1$ имен. Процесс работы алгоритма для таблицы из $n = 8$ имен показан на рис. 5.6.

$$\textbf{for } i \leftarrow n \textbf{ downto } 2 \textbf{ do} \begin{cases} j \leftarrow 1 \\ \textbf{for } k \leftarrow 2 \textbf{ to } i \textbf{ do if } x_j < x_k \textbf{ then } j \leftarrow k \\ x_j \leftrightarrow x_i \end{cases}$$

Алгоритм 5.6. Простая сортировка выбором



for $i \leftarrow n$ **downto** 2 **do**

$$\begin{cases} j \leftarrow 1 \\ \textbf{for } k \leftarrow 2 \textbf{ to } i \textbf{ do if } x_j < x_k \textbf{ then } j \leftarrow k \\ x_j \leftrightarrow x_i \end{cases}$$

Рис. 5.6. Процесс работы алгоритма простой сортировки выбором

Анализ этого алгоритма прост, поскольку имеется два предопределенных цикла. Очевидно, что время работы алгоритма не зависит от содержимого исходной таблицы. Число сравнений имен на i-м шаге равно $n - i$, а таких шагов выполняется $n - 1$, т. е. общее число сравнений имен

$$C(n) = (n-1) + (n-2) + \cdots + 1 = \frac{1}{2} n(n-1) = O(n^2).$$

Число обменов определяется внешним циклом **for** и также не зависит от входа, т. е. имеем $M(n) = n - 1 = O(n)$.

Таким образом, алгоритм простой сортировки выбором имеет временную сложность $O(n^2)$ независимо от содержимого исходной таблицы. Он по эффективности несколько уступает простой сортировке вставками, но имеет существенное преимущество перед пузырьковой сортировкой.

Можно довести число обменов до нуля в лучшем случае (когда исходная таблица уже отсортирована), если запретить обмен при $i = j$ (в примере на рис. 5.6 такая ситуация возникает при $i = 3$) включением соответствующей проверки во внешний цикл **for** непосредственно перед операцией обмена. Однако для достаточно больших n это может привести к увеличению общего среднего времени работы алгоритма (если все $n!$ перестановок равновероятны), поскольку увеличивается длительность выполнения каждой итерации внешнего цикла **for** из-за дополнительной проверки условия «$i = j$».

5.4.2. Пирамидальная сортировка

Основным источником неэффективности простой сортировки выбором является поиск *i*-го наибольшего имени. В любом алгоритме нахождения максимума из *n* элементов, основанном на сравнении пар элементов, необходимо выполнить по крайней мере $n - 1$ сравнений. Это справедливо только для первого шага выбора. Более эффективный метод определения *i*-го наибольшего имени достигается тем, что при последующих выборах можно использовать информацию, полученную на предыдущих выборах. Такой подход к выбору *i*-го наибольшего имени используется в так называемом механизме турнира с выбыванием.

Механизм *турнира с выбыванием* заключается в следующем: сравниваются пары $x_1 : x_2$, $x_3 : x_4$, …, $x_{n-1} : x_n$, затем сравниваются «победители» (т. е. большие имена) этих сравнений и т. д. Пример такой процедуры для $n = 16$ показан на рис. 5.7. Для определения наибольшего имени этот процесс потребует $n - 1$ сравнений имен. Но, определив наибольшее имя, мы будем иметь информацию о втором по величине имени: оно должно быть одним из тех, которые «потерпели поражение» от наибольшего имени. Следовательно, второе по величине имя теперь можно определить, заменяя наибольшее имя на $-\infty$ и вновь осуществляя сравнение вдоль пути от наибольшего имени к корню (с $-\infty$ сравнение не производится). Эта процедура для дерева на рис. 5.7 показана на рис. 5.8. Выполняются сравнения 44:68, 63:68, 54:68, т. е. второе по величине имя 68 определяется за 3 сравнения (вместо 14 сравнений при простом выборе).

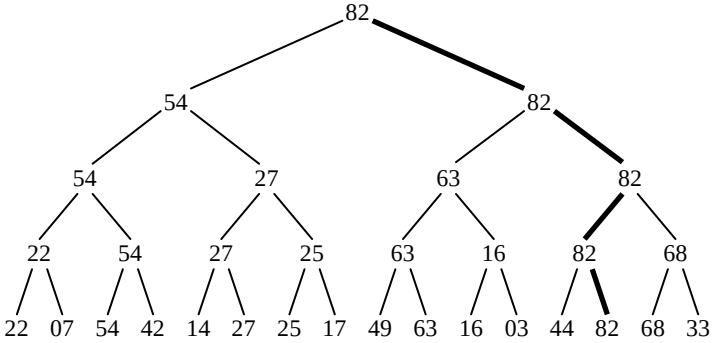


Рис. 5.7. Турнир с выбыванием для поиска наибольшего имени

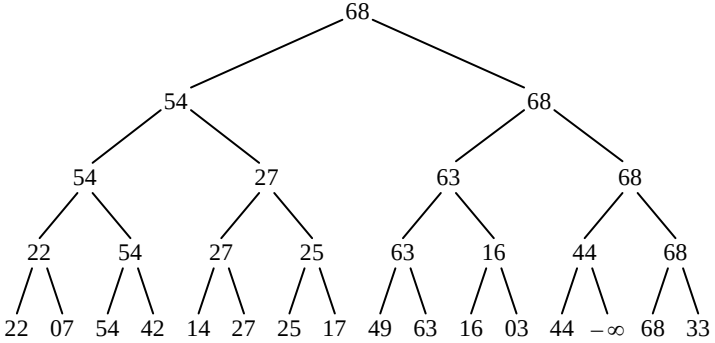


Рис. 5.8. Турнир с выбыванием для поиска второго наибольшего имени

Таким образом, поскольку дерево имеет высоту $\lceil \log n \rceil$, второе по величине имя можно найти за $\lceil \log n \rceil - 1$ сравнений вместо $n - 2$, используемых в простой сортировке выбором. Этот процесс можно продолжить. Найдя второе по величине имя, заменяем его на $-\infty$ и вновь выполняем $\lceil \log n \rceil - 1$ сравнений, чтобы найти следующее, и т. д. Очевидно, что в целом процесс использует не больше

$$n - 1 + (n-1)(\lceil \log n \rceil - 1) \approx n \lceil \log n \rceil$$

сравнений имен.

Идею турнира с выбыванием легко использовать при сортировке, если имена образуют пирамиду. *Пирамида* – это полностью сбалансированное бинарное дерево высоты h, в котором все листья находятся на расстоянии h или $h - 1$ от корня и все потомки вершины меньше его самого; кроме того, в нем все листья уровня h максимально смещены влево. Множество из 12 имен, организованных в виде пирамиды, показано на рис. 5.9.

Удобно использовать линейное представление пирамиды, когда она хранится по уровням в одномерном массиве. Тогда сыновья имени из i-й позиции размещаются в позициях $2i$ (левый) и $2i + 1$ (правый). Таким образом, пирамида, представленная на рис. 5.9, принимает вид:

i:	1	2	3	4	5	6	7	8	9	10	11	12
x_i:	63	49	54	42	16	27	25	17	22	14	07	03

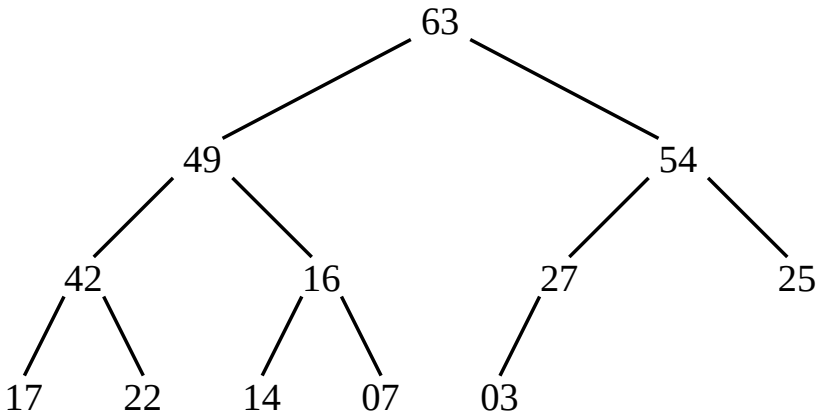


Рис. 5.9. Пирамида из 12 имен

В пирамиде наибольшее имя всегда должно находиться в корне, т. е. всегда в первой позиции массива, представляющего пирамиду. Обмен местами первого имени с n-м помещает наибольшее имя в его правильную позицию, но нарушает свойство пирамидальности в первых $n-1$ именах. Если имеется возможность сначала построить пирамиду из исходной таблицы, а затем ее эффективно восстанавливать, то можно производить сортировку следующим образом:

Построить пирамиду из $x_1, x_2, \ldots, x_n$

for $i \leftarrow n$ **downto** 2 **do** $\begin{cases} x_1 \leftrightarrow x_i \\ \text{Восстановить пирамиду} \\ \quad \text{в } x_1, x_2, \ldots, x_{i-1} \end{cases}$

Это общее описание *пирамидальной сортировки*, состоящей из *фазы построения* пирамиды и *фазы выбора*.

Рассмотрим сначала вопросы восстановления пирамиды. В результате обмена первого имени с последним свойство пирамидальности нарушается в корне дерева, но оба поддерева этого дерева являются пирамидами. Поэтому для восстановления пирамиды необходимо сравнить корень с большим из сыновей. Если корень больше, дерево уже является пирамидой, но если корень меньше, то его надо поменять местами с большим сыном. Далее процесс рекурсивно продолжается для поддерева, корень которого заменялся. Таким образом, процедуру *RESTORE* (*f*, *l*) восстановления пирамиды из последовательности x_f, x_{f+1}, …, x_l в предположении, что все поддеревья являются пирамидами, можно записать следующим образом:

procedure $RESTORE(f, l)$

$$\textbf{if } x_f \neq \text{лист } \textbf{then} \begin{cases} \text{Пусть } x_m \text{ есть больший из сыновей } x_f \\ \textbf{if } x_m > x_f \textbf{ then} \begin{cases} x_m \leftrightarrow x_f \\ RESTORE(m, l) \end{cases} \end{cases}$$

return

Очевидно, что x_f является листом только тогда, когда $f > \lfloor l/2 \rfloor$. Представив этот процесс итеративным способом и дополнив деталями, получим алгоритм 5.7. Процесс восстановления пирамиды из последовательности $x_1, \ldots, x_{11}$, в которой свойство пирамидальности было нарушено в результате обмена $x_1 = 63$ с $x_{12} = 03$ (до обмена имена $x_1, \ldots, x_{12}$ образовывали пирамиду) показан на рис. 5.10.

procedure *RESTORE*(f, l)
  $j \leftarrow f$
  while $j \le \lfloor l/2 \rfloor$ **do**
    if $2j < l$ **and** $x_{2j} < x_{2j+1}$
      then $m \leftarrow 2j+1$
      else $m \leftarrow 2j$
    if $x_m > x_j$
      then $x_m \leftrightarrow x_j$; $j \leftarrow m$
      else $j \leftarrow l$
return

Алгоритм 5.7. Процедура восстановления пирамиды

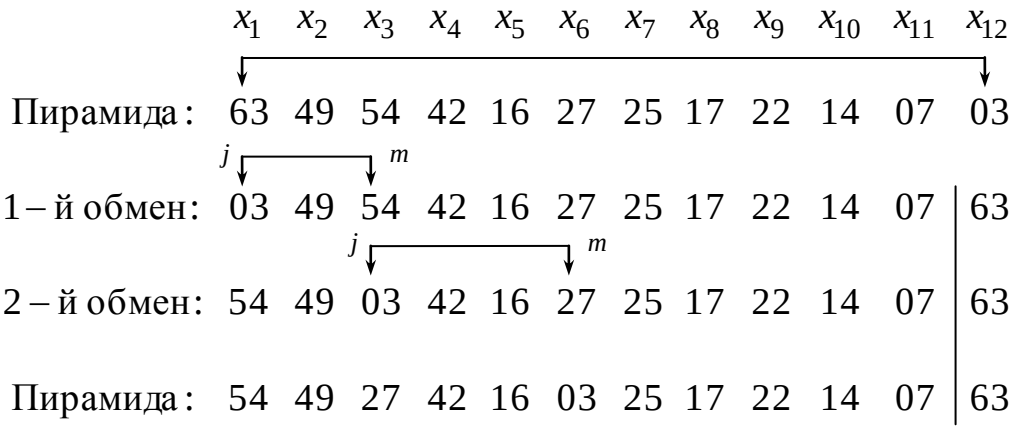


Рис. 5.10. Процесс восстановления пирамиды

Для построения пирамиды необходимо обратить внимание на то, что свойство пирамидальности уже тривиально выполняется для каждого листа, т. е. для всех x_i при $i = \lfloor n/2 \rfloor + 1, \ldots, n$. Тогда обращение к процедуре *RESTORE* (i, n) восстановления пирамиды для $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 1$ преобразует произвольную таблицу в пирамиду на всех высших уровнях (рис. 5.11).

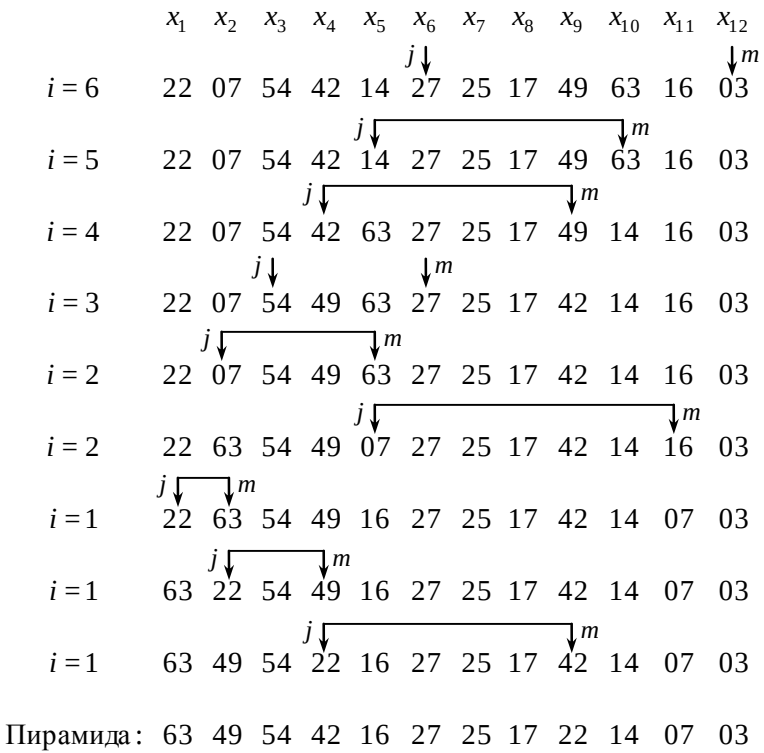


Рис. 5.11. Процесс построения пирамиды

Таким образом, полный процесс пирамидальной сортировки можно представить алгоритмом 5.8. В этом алгоритме первый цикл **for** строит пирамиду, т. е. реализует фазу построения, а второй цикл **for** реализует фазу выбора.

$$\textbf{for } i \leftarrow \lfloor n/2 \rfloor \textbf{ downto } 1 \textbf{ do } RESTORE(i, n)$$
$$\textbf{for } i \leftarrow n \textbf{ downto } 2 \textbf{ do } \begin{cases} x_1 \leftrightarrow x_i \\ RESTORE(1, i-1) \end{cases}$$

Алгоритм 5.8. Пирамидальная сортировка

Анализ пирамидальной сортировки достаточно сложен. Поэтому приведем здесь только результаты анализа. В фазе построения выполняется не более $O(n)$ обменов и не более $O(n)$ сравнений имен. В фазе выбора производится не более $n \log n + O(n)$ обменов и не более $2n \log n + O(n)$ сравнений имен. Таким образом, для пирамидальной сортировки любой таблицы требуется $O(n \log n)$ операций, в то время как в быстрой сортировке при некоторых таблицах совершается $O(n^2)$ операций. Несмотря на это, быстрая сортировка в среднем более эффективна, чем пирамидальная.