Содержание

Задача сортировки 2
Простые сортировки 2
Сортировка простыми вставками 2
Сортировка бинарными вставками 4
Сортировка простым выбором 5
Сортировка простыми обменами 6
Улучшенные сортировки 7
Сортировка Шелла 7
Пирамидальная сортировка 9

Задача сортировки

Эта лекция посвящена сугубо алгоритмической проблеме упорядочения данных.

Необходимость отсортировать какие-либо величины возникает в программировании очень часто. К примеру, входные данные подаются "вперемешку", а вашей программе удобнее обрабатывать упорядоченную последовательность. Существуют ситуации, когда предварительная сортировка данных позволяет сократить содержательную часть алгоритма в разы, а время его работы - в десятки раз.

Однако верно и обратное. Сколь бы хорошим и эффективным ни был выбранный вами алгоритм, но если в качестве подзадачи он использует "плохую" сортировку, то вся работа по его оптимизации оказывается бесполезной. Неудачно реализованная сортировка входных данных способна заметно понизить ***эффективность*** алгоритма в целом.

Методы упорядочения подразделяются на ***внутренние*** (обрабатывающие массивы) и ***внешние*** (занимающиеся только файлами)[1].

Эту лекцию мы посвятим только внутренним сортировкам. Их важная особенность состоит в том, что эти алгоритмы не требуют дополнительной памяти: вся работа по упорядочению производится внутри одного и того же массива.

Простые сортировки

К ***простым внутренним сортировкам*** относят методы, сложность которых пропорциональна квадрату размерности входных данных. Иными словами, при *сортировке массива*, состоящего из N компонент, такие алгоритмы будут выполнять $C*N^2$ действий, где C - некоторая константа.

Количество действий, необходимых для упорядочения некоторой последовательности данных, конечно же, зависит не только от длины этой последовательности, но и от ее структуры. Например, если на вход подается уже упорядоченная последовательность (о чем программа, понятно, не знает), то количество действий будет значительно меньше, чем в случае перемешанных входных данных.

Как правило, сложность алгоритмов подсчитывают раздельно по количеству сравнений и по количеству перемещений данных в памяти (пересылок), поскольку выполнение этих операций занимает различное время. Однако точные значения удается найти редко, поэтому для оценки алгоритмов ограничиваются лишь понятием "пропорционально", которое не учитывает конкретные значения констант, входящих в итоговую формулу. Общую же*эффективность алгоритма* обычно оценивают "в среднем": как среднее арифметическое от сложности алгоритма "в лучшем случае" и "в худшем случае", то есть (Eff_best + Eff_worst)/2.

Сортировка простыми вставками

Самый простой способ сортировки[2], который приходит в голову, - это упорядочение данных по мере их поступления. В этом случае при вводе каждого нового значения можно опираться на тот факт, что все предыдущие элементы уже образуют отсортированную последовательность.

Алгоритм ПрВст

1. Первый элемент записать "не раздумывая".
2. Пока не закончится последовательность вводимых данных, для каждого нового ее элемента выполнять следующие действия:
   - начав с конца уже существующей упорядоченной последовательности, все ее элементы, которые больше, чем вновь вводимый элемент, сдвинуть на 1 шаг назад;
   - записать новый элемент на освободившееся место.

При этом, разумеется, можно прочитать все вводимые элементы одновременно, записать их в массив, а потом "воображать", что каждый очередной элемент был введен только что. На суть и структуру алгоритма это не повлияет.

Реализация алгоритма ПрВст

```
for i:= 2 to N do
 if a[i-1]>a[i] then                {*}
  begin x:= a[i];
      j:= i-1;
   while (j>0)and(a[j]>x) do    {**}
     begin a[j+1]:= a[j];
        j:= j-1;
          end;
  a[j+1]:= x;
 end;
```

Метод прямых вставок с барьером (ПрВстБар)

Для того чтобы сократить количество сравнений, производимых нашей программой, дополним сортируемый массив нулевой компонентой (это следует сделать в разделе описаний var) и будем записывать в нее поочередно каждый вставляемый элемент (сравните строки {*} и {**} в приведенных вариантах программы). В тех случаях, когда вставляемое значение окажется меньше, чем a[1], компонента a[0] будет работать как "барьер", не дающий индексу j выйти за *нижнюю границу* массива. Кроме того, компонента a[0] может заменить собою и дополнительную переменную x:

```
for i:= 2 to N do
 if a[i-1]>a[i] then
  begin a[0]:= a[i];                {*}
      j:= i-1;
       while a[j]>a[0] do          {**}
     begin a[j+1]:= a[j];
      j:= j-1;
         end;
  a[j+1]:= a[0];
 end;
```

Эффективность алгоритма ПрВстБар

Понятно, что для этой сортировки наилучшим будет случай, когда на вход подается уже упорядоченная последовательность данных. Тогда алгоритм ПрВстБар совершит N-1 сравнение и 0 пересылок данных.

В худшем же случае - когда входная последовательность упорядочена "наоборот" - сравнений будет уже (N+1)*N/2, а пересылок (N-1)*(N+3). Таким образом, этот алгоритм имеет сложность ~N^2 (читается "порядка эн квадрат") по обоим параметрам.

Пример сортировки

Предположим, что нужно отсортировать следующий набор чисел:

```
5 3 4 3 6 2 1
```

Выполняя алгоритм ПрВстБар, мы получим такие результаты (подчеркнута уже отсортированная часть массива, полужирным выделена сдвигаемая последовательность, а квадратиком выделен вставляемый элемент):

Состояние массива	Сдвиги	Сравнения	Пересылки данных
0 шаг: 5343621			
1 шаг: 5343621	1	1+ 1[3]	1+ 2[4]
2 шаг: 3543621	1	1+1	1+2
3 шаг: 3453621	2	2+1	2+2
4 шаг: 3345621	0	1	0
5 шаг: 3345621	5	5+1	5+2
6 шаг: 2334561	6	6+1	6+2
Результат: 1233456	15	20	25

Сортировка бинарными вставками

Сортировку простыми вставками можно немного улучшить: поиск "подходящего места" в упорядоченной последовательности можно вести более экономичным способом, который называется Двоичный поиск в упорядоченной последовательности. Он напоминает детскую игру "больше-меньше": после каждого сравнения обрабатываемая последовательность сокращается в два раза.

Пусть, к примеру, нужно найти место для элемента 7 в таком массиве:

```
[2 4 6 8 10 12 14 16 18]
```

Найдем средний элемент этой последовательности (10) и сравним с ним семерку. После этого все, что больше 10 (да и саму десятку тоже), можно смело исключить из дальнейшего рассмотрения:

```
[2 4 6 8] 10 12 14 16 18
```

Снова возьмем середину в отмеченном куске последовательности, чтобы сравнить ее с семеркой. Однако здесь нас поджидает небольшая проблема: точной середины у новой последовательности нет, поэтому нужно решить, который из двух центральных элементов станет этой "серединой". От того, к какому краю будет смещаться выбор в таких "симметричных" случаях, зависит окончательная реализация нашего алгоритма. Давайте договоримся, что новой "серединой" последовательности всегда будет становиться левый центральный элемент. Это соответствует вычислению номера "середины" по формуле

```
nomer_sred:= (nomer_lev + nomer_prav)div 2
```

Итак, отсечем половину последовательности:

```
2 4 [6 8] 10 12 14 16 18
```

И снова:

```
2 4 6 [8] 10 12 14 16 18
2 4 6][8 10 12 14 16 18
```

Таким образом, мы нашли в исходной последовательности место, "подходящее" для нового элемента. Если бы в той же самой последовательности нужно было найти позицию не для семерки, а для девятки, то последовательность границ рассматриваемых промежутков была бы такой:

```
[2 4 6 8] 10 12 14 16 18
2 4 [6 8] 10 12 14 16 18
2 4 6 [8] 10 12 14 16 18
2 4 6 8][10 12 14 16 18
```

Из приведенных примеров уже видно, что поиск ведется до тех пор, пока левая граница не окажется правее(!) правой границы. Кроме того, по завершении этого поиска последней левой границей окажется как раз тот элемент, на котором необходимо закончить сдвиг "хвоста" последовательности.

Будет ли такой алгоритм универсальным? Давайте проверим, что же произойдет, если мы станем искать позицию не для семерки или девятки, а для единицы:

```
[2 4 6 8] 10 12 14 16 18
[2] 4 6 8 10 12 14 16 18
][2 4 6 8 10 12 14 16 18
```

Как видим, правая граница становится неопределенной - выходит за пределы массива. Будет ли этот факт иметь какие-либо неприятные последствия? Очевидно, нет, поскольку нас интересует не правая, а левая граница.

"А что будет, если мы захотим добавить 21?" - спросит особо въедливый читатель. Проверим это:

```
2 4 6 8 10 [12 14 16 18]
2 4 6 8 10 12 14 [16 18]
2 4 6 8 10 12 14 16 [18]
2 4 6 8 10 12 14 16 18][
```

Кажется, будто все плохо: левая граница вышла за пределы массива; непонятно, что нужно сдвигать...

Вспомним, однако, что в реальности на (N+1)-й позиции как раз и находится вставляемый элемент (21). Таким образом, если левая граница вышла за рассматриваемый диапазон, получается, что ничего сдвигать не нужно. Вообще

же такие действия выглядят явно лишними, поэтому от них стоит застраховаться, введя одну дополнительную проверку в текст алгоритма.

Реализация алгоритма БинВст

```
for i:= 2 to n do
    if a[i-1]>a[i] then
  begin x:= a[i];
   left:= 1;
   right:= n-1;
   repeat
     sred:= (left+right) div 2;
     if a[sred]<x then left:= sred+1
                  else right:= sred-1;
       until left>right;
   for j:= i-1 downto left do a[j+1]:= a[j];
   a[left]:= x;
 end;
```

Эффективность алгоритма БинВст

Теперь на каждом шаге выполняется не N, а log N проверок[5], что уже значительно лучше (для примера, сравните 1000 и 10 = log 1024). Следовательно, всего будет совершено N*log N сравнений. Впрочем, улучшение это не слишком значительное, ведь по количеству пересылок наш алгоритм по-прежнему имеет сложность "порядка N^2".

Сортировка простым выбором

Попробуем теперь сократить количество пересылок элементов.

Алгоритм ПрВыб

На каждом шаге (всего их будет ровно N-1) будем производить такие действия:

1. найдем минимум среди всех еще не упорядоченных элементов;
2. поменяем его местами с первым "по очереди" не отсортированным элементом. Мы надеемся, что читателям очевидно, почему к концу работы этого алгоритма последний (N-й) элемент массива автоматически окажется максимальным.

Реализация ПрВыб

```
for i:= 1 to n-1 do
  begin min_ind:= i;
   for j:= i+1 to n do
   if a[j]<=a[min_ind]    {***}
          then min_ind:= j;
   if min_ind<>i
    then begin
     x:= a[i];
     a[i]:= a[min_ind];
     a[min_ind]:= x;
   end;
  end;
```

Эффективность алгоритма ПрВыб

В лучшем случае (если исходная последовательность уже упорядочена), алгоритм ПрВыб произведет (N-1)*(N+2)/2 сравнений и 0 пересылок данных. В остальных же случаях количество сравнений останется прежним, а вот количество пересылок элементов массива будет равным 3*(N-1).

Таким образом, алгоритм ПрВыб имеет *квадратичную сложность* (~N^2) по сравнениям и линейную (~N) - по пересылкам.

Замечание. Если перед вами поставлена задача отсортировать строки двумерного массива (размерности NxN) по значениям его первого столбца, то сложность алгоритма ПрВыб, модифицированного для решения этой задачи, будет квадратичной (N^2 сравнений и N^2 пересылок), а алгоритма БинВст - кубической (N*log N сравнений и N^3 пересылок). Комментарии, как говорится, излишни.

Пример сортировки

Предположим, что нужно отсортировать тот же набор чисел, при помощи которого мы иллюстрировали метод сортировки простыми вставками:

```
5 3 4 3 6 2 1
```

Теперь мы будем придерживаться алгоритма ПрВыб (подчеркнута несортированная часть массива, а квадратиком выделен ее минимальный элемент):

```
1 шаг:          5343621
2 шаг:          1343625
3 шаг:          1243635 {***}[6]
4 шаг:          1233645{ничего не делаем}
5 шаг:          123 3645
6 шаг:          1233 465
результат:    12334 56
```

Сортировка простыми обменами

Рассмотренными сортировками, конечно же, не исчерпываются все возможные методы упорядочения массивов.

Существуют, например, алгоритмы, основанные на обмене двух соседних элементов: Пузырьковая и Шейкерная сортировки. Обе имеют сложность порядка N^2, однако и по скорости работы на любых входных данных, и по простоте реализации они проигрывают другим простым сортировкам. Поэтому мы настойчиво советуем читателю не прельщаться красивыми названиями, за которыми не стоит никакой особенной выгоды.

Тем же, кто все-таки желает ознакомиться с обменными сортировками, а также с подробными данными по сравнению различных сортировок, мы рекомендуем труды Д. Кнута[7] или Н. Вирта[8].

Улучшенные сортировки

В отличие от простых сортировок, имеющих сложность ~N^2, к ***улучшенным сортировкам*** относятся алгоритмы с общей сложностью ~N*logN.

Необходимо, однако, отметить, что на небольших наборах сортируемых данных (N<100) эффективность быстрых сортировок не столь очевидна: выигрыш становится заметным только при больших N. Следовательно, если необходимо отсортировать маленький набор данных, то выгоднее взять одну из простых сортировок.

Сортировка Шелла

Эта сортировка[1] базируется на уже известном нам алгоритме простых вставок ПрВст. Смысл ее состоит в раздельной сортировке методом ПрВст нескольких частей, на которые разбивается исходный массив. Эти разбиения помогают сократить количество пересылок: для того, чтобы освободить "правильное" место для очередного элемента, приходится уже сдвигать меньшее количество элементов.

Алгоритм УлШелл

На каждом шаге (пусть переменная t хранит номер этого шага) нужно произвести следующие действия:

1. вычленить все подпоследовательности, расстояние между элементами которых составляет k_t;
2. каждую из этих подпоследовательностей отсортировать методом ПрВст.

Нахождение убывающей последовательности расстояний k_t, k_{t-1}..., k_1 составляет главную проблему этого алгоритма. Многочисленные исследования позволили выявить ее обязательные свойства:

- $k_1 = 1$;
- для всех t $k_t > k_{t-1}$;
- желательно также, чтобы все k_t не были кратными друг другу (для того, чтобы не повторялась обработка ранее отсортированных элементов).

Дональд Кнут предлагает две "хорошие" последовательности расстояний:

$1, 4, 13, 40, 121, \quad (k_t = 1+3*k_{t-1})$

$1, 3, 7, 15, 31, \_ \quad (k_t = 1+2*k_{t-1} = 2^t -1)$

Первая из них подходит для сортировок достаточно длинных массивов, вторая же более удобна для коротких. Поэтому мы остановимся именно на ней (желающим запрограммировать первый вариант предоставляется возможность самостоятельно внести необходимые изменения в текст реализации алгоритма).

Как же определить начальное значение для t (а вместе с ним, естественно, и для k_t)?

Можно, конечно, шаг за шагом проверять, возможно ли вычленить из сортируемого массива подпоследовательность (хотя бы длины 2) с расстояниями 1, 3, 7, 15 и т.д. между ее элементами. Однако такой способ довольно неэффективен. Мы поступим иначе, ведь у нас есть формула для вычисления $k_t = 2^t -1$.

Итак, длина нашего массива (N) должна попадать в такие границы:

$k_t <= N -1 < k_{t+1}$

или, что то же самое,

$2^t <= N < 2^{t+1}$

Прологарифмируем эти неравенства (по основанию 2):

$t <= \log N < t+1$

Таким образом, стало ясно, что t можно вычислить по следующей формуле:

$t = trunc(\log N))$

К сожалению, язык Pascal предоставляет возможность логарифмировать только по основанию e (*натуральный логарифм*). Поэтому нам придется вспомнить знакомое из курса средней школы правило "превращения" логарифмов:

$\log_m x = \log_z x / \log_z m$

В нашем случае m = 2, z = e. Таким образом, для начального t получаем:

```
t:= trunc(ln(N)/ln(2)).
```

Однако при таком t часть подпоследовательностей будет иметь длину 2, а часть - и вовсе 1. Сортировать такие подпоследовательности незачем, поэтому стоит сразу же отступить еще на 1 шаг:

```
t:= trunc(ln(N)/ln(2))-1
```

Расстояние между элементами в любой подпоследовательности вычисляется так:

```
k:= (1 shl t)-1;          {k= 2^t-1}
```

Количество подпоследовательностей будет равно в точности k. В самом деле, каждый из первых k элементов служит началом для очередной подпоследовательности. А дальше, начиная с (k+1)-го, все элементы уже являются членами некоторой, ранее появившейся подпоследовательности, значит, никакая новая подпоследовательность не сможет начаться в середине массива.

Сколько же элементов будет входить в каждую подпоследовательность? Ответ таков: если длину всей сортируемой последовательности (N) можно разделить на шаг k без остатка, тогда все подпоследовательности будут иметь одинаковую длину, а именно:

```
s:= N div k;
```

Если же N не делится на шаг k нацело, то первые p подпоследовательностей будут длиннее на 1. Количество таких "удлиненных" подпоследовательностей совпадает с длиной "хвоста" - остатка от деления N на шаг k:

```
P:= N mod k;
```

Реализация алгоритма УлШелл

Ради большей наглядности мы пожертвовали эффективностью и воспользовались алгоритмом ПрВст, а не ПрВстБар или БинВст. Дотошному же читателю предоставляется возможность самостоятельно улучшить предлагаемую реализацию:

```
program shell_sort;
const n=18;
   a:array[1..n] of integer
     =(18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1);
var ii,m,x,s,p,t,k,r,i,j: integer;
begin
 t:= trunc(ln(n)/ln(2));
  repeat
   t:= t-1;
   k:= (1 shl t)-1;
   p:= n mod k;
   s:= n div k;
   if p=0 then p:= k
     else s:= s+1;

     writeln(k,'-сортировка');
      for i:= 1 to k do   {берем      и      длинные,      и      короткие
подпоследовательности}
          begin
        if i= p+1 then s:= s-1;   (для коротких - уменьшаем     длину}
           for j:= 1 to s-1 do    {метод ПрВст с шагом k}
        if a[i+(j-1)*k]>a[i+j*k]
             then begin x:= a[i+j*k];
                  m:= i+(j-1)*k;
                  while (m>0) and (a[m]>x) do
                  begin a[m+k]:= a[m];
                         m:= m-k;
                        end;
                        a[m+k]:= x;
                 end;
       for ii:= 1 to n do write(a[ii],' ');
           writeln;
    end;
   until k=1;
```

```
end.
```

Результат работы

7-сортировки

```
4 17  16  15 14 13 12 11 10 9 8 7  6  5  18 3  2  1
4 3   16  15 14 13 12 11 10 9 8 7  6  5  18 17 2  1
4 3   2   15 14 13 12 11 10 9 8 7  6  5  18 17 16 1
4 3   2   1  14 13 12 11 10 9 8 7  6  5  18 17 16 15
4 3   2   1  7  13 12 11 10 9 8 14 6  5  18 17 16 15
4 3   2   1  7  6  12 11 10 9 8 14 13 5  18 17 16 15
4 3   2   1  7  6  5  11 10 9 8 14 13 12 18 17 16 15
```

3-сортировки

```
1 3 2 4 7 6 5 11 10 9 8  14 13 12 18 17 16 15
1 3 2 4 7 6 5 8  10 9 11 14 13 12 18 17 16 15
1 3 2 4 7 6 5 8  10 9 11 14 13 12 15 17 16 18
```

1-сортировка

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

Эффективность алгоритма УлШелл

Довольно сложными методами, в изложение которых мы не будем углубляться, показано, что *алгоритм Шелла* имеет сложность $\sim N^{3/2}$. И хотя это несколько хуже, чем N*logN, все-таки эта сортировка относится к улучшенным.

Пример сравнения сортировок: Вновь возьмем последовательность, для сортировки которой методом простых вставок ПрВст потребовалось 15 сдвигов (25 пересылок и 20 сравнений):

```
5  3  4  3  6  2  1
```

Теперь применим к ней метод Шелла.

Здесь N = 7, поэтому:

```
t= trunc(log 7) = 2
k= 2^2-1 = 3                  {начнем с 3-сортировки}
p= 7 mod 3 = 1              {кол-во длинных подпоследовательностей}
s= (7 div 3)+1 = 3   {длина длинной подпоследовательности}
```

1. 3-сортировки:

```
2.   5 3 1 -> 1 3 5  {3 сдвига: 7 пересылок, 5 сравнений}
3.   3 6   -> 3 6    {0 сдвигов: 0 пересылок, 1 сравнение}
        4 2   -> 2 4{1 сдвиг: 3 пересылки, 2 сравнения}
```

   Всего 4 сдвига: 10 пересылок, 8 сравнений Итог 3-сортировок: 1 3 2 3 6 4 5

4. 1-сортировка:

```
5.   Состояние массива  Сдвиги  Сравнения  Пересылки данных
6.
7.   0 шаг:        1323645
8.   1 шаг:        1323645        0         1          0
9.   2 шаг:        1323645        1        1+1        1+2
10.  3 шаг:        1233645        0         1          0
11.  4 шаг:        1233645        0         1          0
12.  5 шаг:        1233645        1        1+1        1+2
13.  6 шаг:        1233465        1        1+1        1+2
        результат:    1233456              3          9          9
```

При сортировке методом Шелла в сумме получилось 7 сдвигов (19 пересылок и 17 сравнений). Выигрыш по сравнению с методом простых вставок составляет 53% (24% экономится на пересылках и 15% - на сравнениях)[2]. Если вместо метода простых вставок ПрВст использовать метод бинарных вставок БинВст, то выигрыш по количеству сравнений будет ощутимее.

Кроме того, не нужно забывать, что в нашем примере последовательность очень коротка: N = 7. Для больших N (скажем, N = 10000) преимущество метода Шелла станет еще заметнее.

Пирамидальная сортировка

Попытаемся теперь усовершенствовать другой рассмотренный выше простой алгоритм: сортировку простым выбором ПрВыб.

Р. Флойд предложил перестроить *линейный массив* в *пирамиду* - своеобразное *бинарное дерево*, - а затем искать минимум только среди тех элементов, которые находятся непосредственно "под" текущим вставляемым.

Просеивание

Для начала необходимо перестроить исходный массив так, чтобы он превратился в *пирамиду*, где каждый элемент "опирается" на два меньших. Этот процесс назвали просеиванием, потому что он очень напоминает процесс разделения некоторой смеси (камней, монет, т.п.) на фракции в соответствии с размерам частиц: на нескольких грохотах[3)] последовательно задерживаются сначала крупные, а затем все более мелкие частицы.

Итак, будем рассматривать наш *линейный массив* как пирамидальную структуру:

a[1]						
a[2]				a[3]		
a[4]		a[5]		a[6]		a[7]
a[8]	a[9]	a[10]	a[11]	a[12]		

Видно, что любой элемент a[i] (1<=i<=N div 2) "опирается" на элементы a[2*i] и a[2*i+1]. И в каждой такой тройке максимальный элемент должен находится "сверху". Конечно, исходный массив может и не удовлетворять этому свойству, поэтому его потребуется немного перестроить.

Начнем процесс просеивания "снизу". Половина элементов (с ((N div 2)+1)-го по N-й) являются основанием *пирамиды*, их просеивать не нужно. А для всех остальных элементов (двигаясь от конца массива к началу) мы будем проверять тройки a[i], a[2*i] и a[2*i+1] и перемещать максимум "наверх" - в элемент a[i].

При этом, если в результате одного перемещения нарушается пирамидальность в другой (ниже лежащей) тройке элементов, там снова необходимо "навести порядок" - и так до самого "низа" *пирамиды*:

```
for i:= (N div 2)downto 1 do
 begin j:= i;
  while j<=(N div 2) do
    begin k:= 2*j;
          if (k+1<=N) and (a[k]<a[k+1])
                  then k:= k+1;
          if a[k]>a[j]
                  then begin x:= a[j];
                          a[j]:= a[k];
                          a[k]:= x;
                          j:= k
                  end
          else break
     end
end;
```

Пример результата просеивания

Возьмем массив [1,7,5,4,9,8,12,11,2,10,3,6] (N = 12).

Его исходное состояние таково (серым цветом выделено "основание" *пирамиды*, не требующее просеивания):

1											
7				5							
4		9				8		**12**			
11	**2**		**10**		**3**		**6**				

После первых трех просеиваний (a[6], a[5], a[4]) получим такую картину (здесь и далее серым цветом выделяем участников просеивания):

1											
7				5							
4		9				**8**		12			
11	2		10		3		**6**				

1
7 5
11 **10 9** 8 12
11 2 **9 10** 3 6

1
7 5
11 4 10 8 12
4 11 2 9 3 6

Просеивание двух следующих элементов (a[3] и a[2]) тоже не вызовет вопросов - для каждого из них будет достаточно только одного шага:

1
7 **12 5**
11 10 8 **5 12**
4 2 9 3 6

1
11 7 5
7 11 10 8 12
4 2 9 3 6

А вот для просеивания последнего элемента (a[1]) понадобится целых три шага:

12 1
11 **1 12**
7 1 10 8 5
4 2 9 3 6

12
11 **8 1**
7 10 **1 8** 5
4 2 9 3 6

12
11 8
7 10 **6 1** 5
4 2 9 3 **1 6**

Итак, мы превратили исходный массив в *пирамиду*: в любой тройке a[i], a[2*i] и a[2*i+1] максимум находится "сверху".

Алгоритм УлПир

Для того чтобы отсортировать массив методом *Пирамиды*, необходимо выполнить такую последовательность действий:

0-й шаг: Превратить исходный массив в *пирамиду* (с помощью просеивания).

1-й шаг: Для N-1 элементов, начиная с последнего, производить следующие действия:

- поменять местами очередной "рабочий" элемент с первым;
- просеять (новый) первый элемент, не затрагивая, однако, уже отсортированный хвост последовательности (элементы с i-го по N-й).

Реализация алгоритма УлПир

Часть программы, реализующую нулевой шаг алгоритма УлПир, мы привели в пункте "Просеивание", поэтому здесь ограничимся только реализацией основного шага 1:

```
for i:= N downto 2 do
 begin x:= a[1];
   a[1]:= a[i];
   a[i]:= x;
   j:= 1;
```

```
  while j<=((i-1)div 2) do
   begin k:= 2*j;
     if (k+1<=i-1) and (a[k]<a[k+1])
           then k:= k+1;
     if a[k]>a[j]
           then begin x:= a[j];
                      a[j]:= a[k];
                      a[k]:= x;
                      j:= k
           end
             else break
   end
end;
```

Пример. Продолжим сортировку массива, для которого мы уже построили *пирамиду*: [12,11,8,7,10,6,5,4,2,9,3,1]. С целью экономии места мы не будем далее прорисовывать структуру *пирамиды*, оставляя это несложное упражнение читателям. Подчеркивание будет отмечать элементы, участвовавшие в просеивании, а полужирный шрифт - элементы, исключенные из дальнейшей обработки:

```
1)  Меняем местами a[1] и a[12]:  [1,11,8,7,10,6,5,4,2,9,3,12];
2)  Просеиваем элемент a[1], получаем:   [11,10,8,7,9,6,5,4,2,1,3,12];
3)  Меняем местами a[1] и a[11]:  [3,10,8,7,9,6,5,4,2,1,11,12];
4)  Просеиваем a[1], получаем:           [10,9,8,7,3,6,5,4,2,1,11,12];
5)  Меняем местами a[1] и a[10]:  [1,9,8,7,3,6,5,4,2,10,11,12];
6)  Просеиваем элемент a[1]:             [9,7,8,4,3,6,5,1,2,10,11,12];
7)  Меняем местами a[1] и a[9]:          [2,7,8,4,3,6,5,1,9,10,11,12];
8)  Просеиваем элемент a[1]:             [8,7,6,4,3,2,5,1,9,10,11,12];
9)  Меняем местами a[1] и a[8]:   [1,7,6,4,3,2,5,8,9,10,11,12];
10) Просеиваем элемент a[1]:             [7,4,6,1,3,2,5,8,9,10,11,12];
11) Меняем местами a[1] и a[7]:   [5,4,6,1,3,2,7,8,9,10,11,12];
12) Просеиваем элемент a[1]:             [6,4,5,1,3,2,7,8,9,10,11,12];
13) Меняем местами a[1] и a[6]:   [2,4,5,1,3,6,7,8,9,10,11,12];
14) Просеиваем элемент a[1]:             [5,4,2,1,3,6,7,8,9,10,11,12];
15) Меняем местами a[1] и a[5]:   [3,4,2,1,5,6,7,8,9,10,11,12];
16) Просеиваем элемент a[1]:             [4,3,2,1,5,6,7,8,9,10,11,12];
17) Меняем местами a[1] и a[4]:   [1,3,2,4,5,6,7,8,9,10,11,12];
18) Просеиваем элемент a[1]:             [3,1,2,4,5,6,7,8,9,10,11,12];
19) Меняем местами a[1] и a[3]:   [2,1,3,4,5,6,7,8,9,10,11,12];
20) Просеивать уже ничего не нужно;
21) Меняем местами a[1] и a[2]:   [1,2,3,4,5,6,7,8,9,10,11,12];
22) Просеивать ничего не нужно, сортировка закончена.
```

Эффективность алгоритма УлПир

Пирамидальная сортировка хорошо работает с большими массивами, однако на маленьких примерах (N<20) выгода от ее применения может быть не слишком очевидна.

В среднем этот алгоритм имеет сложность, пропорциональную N*log N.

Быстрая сортировка

Существует еще один метод улучшенной сортировки, имеющий среднюю сложность порядка N*log N: так называемая Быстрая сортировка[41]. Этот алгоритм является усовершенствованием обменных сортировок. Его реализация наиболее удобна в рекурсивном варианте, поэтому мы вернемся к ее изучению после того, как познакомимся с рекурсивными процедурами и функциями (см. лекцию 9).