## Квадратичные сортировки

П.Осипов

Между однотипными информационными объектами (например, числами или строками) возможны различные операции отношения. Некоторые из этих отношений могут быть очевидны и использоваться очень часто, например: «число А больше числа В», «строка А длиннее строки В», «слово А при алфавитном порядке должно стоять раньше слова В». Некоторые отношения могут возникать только при решении задач отдельного вида.

Большинство отношений, с которыми мы сталкиваемся в реальной жизни, обладают следующим свойством: если отношение верно для пары объектов (A,B) и верно для пары объектов (B,C), то из этого следует что оно верно также и для пары (A,C). Такие отношения называются транзитивными. Примером может служить отношение «тяжелее чем». И верно: если известно, что А тяжелее В, а В тяжелее С, то и без взвешивания понятно, что А тяжелее С. В качестве другого примера можно взять отношение «строка длиннее чем». Если известно, что строка А длиннее В, а В длиннее С, то не нужно никаких дополнительных операций чтобы заключить что А длиннее С.

Взяв за основу то или иное транзитивное отношение, данные можно выстраивать в последовательности таким образом, чтобы оно выполнялось для любой пары соседних элементов. Например, последовательность 3, 6, 7, 10, 11, 12, 20, 24 построена на основе отношения «больше». В любой паре последующий элемент больше предыдущего. А поскольку отношение «больше» является транзитивным, то это же будет верно для цепочки подряд идущих элементов любой длины и для всей последовательности в целом (так как второй элемент больше первого, третий больше второго и так далее, то последний больше первого). О расположенных таким образом элементах принято говорить, что они расположены в порядке возрастания.

Приведем другой пример. Последовательность слов ‘a’, ‘an’, ‘the’, ‘then’, ‘image’, ‘window’ выстроена в упорядоченную цепочку на основе соотношения «длиннее».

Еще один пример. На основе алфавитного порядка слов можно получить цепочку: “at”, “bat”, “cat”, “dog”, “mouse”, “rat”.

Процесс, при котором из некоторого набора исходных данных получается такая закономерная цепочка, называется сортировкой. Как и многие другие процессы, сортировка может быть представлена в виде алгоритма. Основное требование к алгоритму сортировки – это гарантия результата (то есть получения упорядоченного набора данных) при любом расположении исходных данных.

Прежде чем перечислить основные алгоритмы сортировки желательно, чтобы читатель прислушался к следующим соображениям. При написании программы есть выбор: реализовать алгоритм, который будет работать быстро (и при этом потратить время на его изучение, отладку, «поимку» частных случаев) или написать хорошо знакомый алгоритм (который может быть и не является самым оптимальным в своей «весовой категории», но зато не будет отвлекать внимание от решения основной задачи). Вовсе не обязательно хорошо знать и уметь писать все алгоритмы сортировки, которые будут разобраны в этой главе, достаточно выбрать для себя какой-то наиболее привлекательный и научиться его реализовывать по-настоящему хорошо. Чутье подсказывает, что выбор алгоритмов и методов работы с данными должен делаться для каждой задачи отдельно, в зависимости от количества данных, сложности и других параметров. Другими словами, если мы пишем программку «только для себя», в которой будет не более тысячи чисел – можно выбрать один метод сортировки, если программой будут пользоваться тысячи человек (например, будут обращаться к какому-нибудь Интернет-серверу) и данных ожидается порядка

нескольких миллиардов – то тут есть смысл искать менее знакомые, но более оптимальные алгоритмы.

И, перед тем как перейти к описаниям алгоритмов оговоримся, что все примеры будут касаться сортировки массивов, состоящих из целых чисел в порядке возрастания. Константой N будет обозначено количество элементов массива.

## Метод выбора

Давайте представим себе, что перед нами на листе бумаги написан набор чисел, который необходимо отсортировать по возрастанию. Как мы будем действовать? Один из возможных вариантов – найти наименьшее число и записать его в ответ первым, потом следующее за ним по величине и записать вторым, потом следующее и так далее.
Именно по такой стратегии и действует метод выбора. Просмотрим все элементы массива и найдем наименьшее число (можно находить и наибольшее, тогда немного изменятся дальнейшие действия, но сама эффективность метода не пострадает). Поменяем его с первым. Затем опять будем искать наименьший элемент, но уже не во всем массиве, а начиная со второго. Найденное число поставим на второе место в массиве. Теперь уже два самых маленьких элемента поставлены нами на «свои» места. Повторим поиск наименьшего, начиная с третьего элемента и поставим найденное число на третье место в массиве. Будем действовать по такой схеме, пока не упорядочим весь массив.

Рассмотрим поэтапно работу данного метода на примере. Пусть дан исходный массив:

| 8 | 4 | 12 | 6 | 3 | 2 | 10 | 7 |
|---|---|---|---|---|---|---|---|

На первом шаге среди всех элементов массива найдем наименьший

| 8 | 4 | 12 | 6 | 3 | 2 | 10 | 7 |
|---|---|---|---|---|---|---|---|

и поменяем его местами с первым.

| 2 | 4 | 12 | 6 | 3 | 8 | 10 | 7 |
|---|---|---|---|---|---|---|---|

В этот момент массив можно мысленно разделить на две части. В левой, упорядоченной, хранятся уже рассмотренные элементы, расположение которых, к тому же, не изменится. В правой 0 еще неупорядоченные элементы. Повторим предыдущий шаг (поиск минимума и постановку его, путем обмена на первое место), но уже как бы с «укороченным» массивом (с той частью, где элементы еще не упорядочены). Найдем наименьший из оставшихся элементов

| 2 | 4 | 12 | 6 | 3 | 8 | 10 | 7 |
|---|---|---|---|---|---|---|---|

и поместим его в этот раз на второе место путем обмена со вторым элементом:

| 2 | 3 | 12 | 6 | 4 | 8 | 10 | 7 |
|---|---|---|---|---|---|---|---|

Повторим процесс для еще неупорядоченной части массива. Найдем в правой, неупорядоченной части наименьший элемент:

| 2 | 3 | 12 | 6 | 4 | 8 | 10 | 7 |
|---|---|---|---|---|---|---|---|

и поставим его на третье место в массиве.

| 2 | 3 | 4 | 6 | 12 | 8 | 10 | 7 |
|---|---|---|---|---|---|---|---|

Снова найдем наименьший элемент и поставим его на четвертую позицию:

| 2 | 3 | 4 | 6 | 12 | 8 | 10 | 7 |
|---|---|---|---|---|---|---|---|

Пятый шаг. Найдем наименьший элемент:

| 2 | 3 | 4 | 6 | 12 | 8 | 10 | 7 |
|---|---|---|---|---|---|---|---|

Поставим его на пятую позицию:

| 2 | 3 | 4 | 6 | 7 | 8 | 10 | 12 |
|---|---|---|---|---|---|---|---|

Повторим поиск и обмен до тех пор, пока в правой, неупорядоченной части массива не останется всего один элемент. Поскольку все элементы, кроме оставшегося были выбраны ранее, то они меньше его, так как на каждом шаге мы искали наименьший элемент. Это означает, что последний элемент будет самым большим в массиве и он должен стоять на последнем месте, где уже и находится. Поэтому чтобы поставить все элементы массива на их места достаточно повторить описанную процедуру N-1 раз.

Для реализации данного метода необходим двойной цикл. Параметром внешнего цикла может быть, например, количество уже упорядоченных элементов. Как уже отмечалось, во внешнем цикле достаточно N-1 шагов. Внутренний цикл обеспечивает поиск наименьшего элемента (а точнее, его номера) среди тех, которые еще не упорядочены. После этого элемент с найденным номером и первый элемент неупорядоченной части массива меняются местами.

Фрагмент программы может быть написан, к примеру, так.

```
const
        N = 100;

var
 a    : array [1..N] of integer;
 first: integer;                    {позиция, с которой начинается «неупорядоченная часть
                                     массива. это число также равно номеру текущего шага цикла}
 i  : integer;
 imin  : integer;
 temp  : integer;
begin

{ввод массива}

 for first:=1 to N-1  do            { последовательно отодвигаем границу неупорядоченной части}
 begin
    imin:=first;
    for i:=first+1 to N do          {поиск номера наименьшего числа}
      if a[i]<a[imin] then imin:=i;

    temp:=a[first];                 {ставим число в начало неупорядоченной части массива}
    a[first]:=a[imin];
    a[imin]:=temp;
  end;

{массив упорядочен и готов к использованию}

end.
```

Время работы этого метода можно оценить как $O(N^2)$ при любых исходных данных. Действительно, на первом шаге, чтобы найти наименьшее число необходимо сделать N-1 сравнение. На втором шаге – N-2, на третьем N-3 и так далее, на последнем шаге останется сделать всего 1 сравнение. То есть общее количество операций можно выразить по формуле суммы арифметической прогрессии: (N-1)+(N-2)+...+3+2+1=((N-1)+1)(N-1)/2=N(N-1)/2

## Метод вставки

Идея данного метода состоит в следующем. Пусть есть несколько элементов, уже упорядоченных по возрастанию и к ним требуется добавить новый. Найдем такую позицию, куда этот элемент можно поставить, не нарушая упорядоченности массива. Все элементы массива, начиная с найденной позиции, сдвинем вправо и поставим новый элемент на освободившееся место. Рассмотрим пример. Пусть дана уже упорядоченная часть массива, состоящая из пяти чисел:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 7 | 12 | 18 | | | |

И пусть сюда необходимо добавить новое число 9. Очевидно, что это число должно в массиве занимать позицию номер 4. Сдвинем числа, освободив нужную позицию. Получим:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 7 | | 12 | 18 | | |

После этого ничто не мешает поставить новое число на освободившееся место.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 7 | 9 | 12 | 18 | | |

Данный метод, в частности, позволяет формировать уже упорядоченный массив на этапе его ввода, по мере поступления данных. Такая программа может выглядеть, например, так:

```
const
        N = 100;
var
        a       : array [1..N] of Integer;
        k,i,j   : integer;
        data    : integer;

begin
{1}     for k:=1 to N do begin
{2}             read(data);

{3}             i:=k-1;
{4}             while (i>0)and(a[i]>data) do dec(i);

{5}             inc(i);
{6}             for j:=k-1 downto i do a[j+1]:=a[j];
{7}             a[i]:=data;
        end;
        {массив сформирован уже упорядоченным}
end.
```

Пояснения к тексту программы. В строке {1} k – это порядковый номер числа, которое на текущем шаге добавляется к массиву. Это означает, что в массиве на момент этого шага k-1 число, причем эти числа уже упорядочены {3}. Строка {4} представляет собой последовательный поиск справа налево, то есть от больших элементов к меньшим. Цикл закончится либо когда i будет равно 0, либо когда элемент номер i будет меньше того, который мы хотим добавить. В любом случае, сдвигать мы должны элементы массива с номерами от i+1 и далее {5}. Примечание общего порядка: данные по которым делается поиск – упорядочены, это позволяет использовать бинарный или любой другой более быстрый метод поиска. Поэтому при определении времени работы алгоритма будем ориентироваться на операции сдвига, а временем на операции поиска вообще можно пренебречь. Строка {6} – цикл, сдвигающий элементы массива.

Время работы данного алгоритма в *худшем случае* как $O(N^2)$. Худшим будет случай, когда для вставки нового элемента нужно будет двигать все предыдущие, то есть когда на вход подается массив, упорядоченный в обратную сторону.

В отличие от метода выбора, данный метод в *лучшем случае* (когда сдвигать ничего не нужно) будет работать за O(N). Если рассмотреть *средний случай*, то время будет $O(N^2)$.

Метод становится очень эффективным в случае, если необходимо в уже упорядоченный массив внести некие небольшие изменения. В этом случае несколько вставок будут выполняться быстрее, чем сортировка всего массива заново каким-то другим методом.

## Метод простого обмена

Перед тем, как излагать этот метод сортировки, внимательно прислушаемся к одному небольшому примечанию. *Массив упорядочен по возрастанию, если меньшие числа в нем стоят раньше*

*больших*. Чем меньше число, тем раньше оно должно стоять в массиве, чем больше число – тем позже. Другими словами, чем больше число, тем больше его номер в массиве (индекс).
Если мы последовательно рассмотрим все пары соседних чисел, и в каждой из них это свойство выполняется, то массив можно считать упорядоченным.

Именно на этом и базируется основная идея метода обмена. Будем сравнивать пары соседних элементов массива. Если в какой-то паре большее число стоит левее меньшего, то их надо поменять местами.
Рассмотрим работу метода на примере. Пусть дан массив из 6 целых чисел, которые необходимо расположить в порядке возрастания.

| 4 | 3 | 5 | 8 | 6 | 2 |
|---|---|---|---|---|---|

Двигаться будем слева направо (хотя это не принципиально).
Шаг 1. Сравним 4 и 3. Число 4 больше (а значит должно стоять правее) - меняем местами эти элементы. Новый массив:

| 3 | 4 | 5 | 8 | 6 | 2 |
|---|---|---|---|---|---|

Независимо от того, меняли мы местами элементы или нет – просто переходим к следующей паре.
Шаг 2. Сравним 4 и 5. Числа в паре стоят в «правильном» порядке – меньшее левее большего. Ничего не делаем с этими элементами. Массив остается:

| 4 | 3 | 5 | 8 | 6 | 2 |
|---|---|---|---|---|---|

Переходим к следующей паре.
Шаг 3. Сравниваем 5 и 8. Эти числа также стоят в «правильном» порядке, поэтому массив не изменяется:

| 4 | 3 | 5 | 8 | 6 | 2 |
|---|---|---|---|---|---|

Берем следующую пару.
Шаг 4. Сравниваем 8 и 6. Здесь большее число стоит левее меньшего, поэтому меняем числа местами, получаем новый массив:

| 4 | 3 | 5 | 6 | 8 | 2 |
|---|---|---|---|---|---|

Переходим к следующей паре.
Шаг 5. Сравниваем 8 и 2.Опять большее число левее меньшего – переставляем числа.

| 4 | 3 | 5 | 6 | 2 | 8 |
|---|---|---|---|---|---|

Пары закончились.

Правда, массив пока мало похож на отсортированный. Однако, можно сделать одно важное наблюдение: ***после полного прохода по массиву хотя бы одно число (а именно – самое большое) будет поставлено на свое место.*** Именно поэтому в некоторых книжках этот метод также носит название «метод пузырька» или просто «пузырек». Наибольшее число, как пузырек из глубины стакана с газировкой, «всплывает» на поверхность (на последнее место в массиве) и остается там. Теперь, если мы сделаем второй проход по массиву, то еще одно число встанет на свое место (а может быть и не одно, но об этом чуть позже). Значит, чтобы гарантированно получить отсортированный массив, надо сделать столько проходов, сколько элементов в массиве.

*(Небольшое отступление: на самом деле достаточно сделать N-1 проход по массиву, потому что когда все числа кроме одного будут на своих местах, последнее тоже само собой окажется на своем месте.)*

Итак, мы выяснили, что для успешной сортировки надо сделать N-1 проход, по N-1 сравнению на каждом проходе. Таким образом уже вырисовывается структура программы – двойной цикл. Внешний цикл – по количеству проходов, внутренний – по тем парам элементов, которые мы сравниваем.

Пример программы сортировки массива.

```
const
        N=100;
```

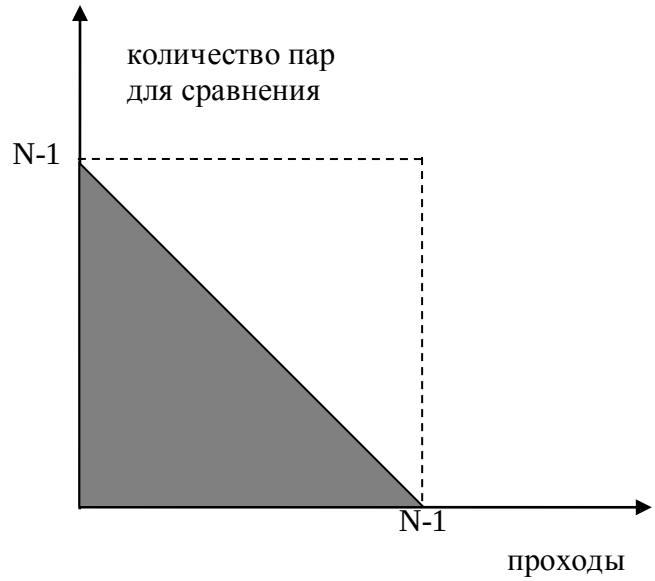
```
var
        a       : array [1..N] of Integer;
        i, j    : integer;
        temp    : integer;
begin
        {ввод массива}

        for i:=1 to N-1 do                      {по проходам}
                for j:=1 to N-1 do              {по парам элементов}
                        if a[j]>a[j+1] then begin       {если в паре левый элемент больше правого}
                                temp:=a[j];             {меняем их местами}
                                a[j]:=a[j+1];           {если нет, то просто переходим к следующей}
                                a[j+1]:=temp;
                        end;

        {массив упорядочен и готов к использованию}
end.
```

По структуре программы видно, что такая сортировка требует (N-1)(N-1) операций. То есть время работы программы можно оценить как $O(N^2)$.

Приведем пару соображений по оптимизации написанного кода. Представим себе, что у нас есть массив на два миллиона чисел, и мы уже сделали миллион проходов. Это значит, что как минимум миллион чисел в массиве уже стоят на своих законных местах в конце массива. Следовательно, нет никакого смысла проходить правую половину массива, потому что там никаких изменений точно уже не будет. Итак, если на первом проходе мы делаем N-1 сравнение, то на втором достаточно N-2, на третьем - N-3 и так далее по мере увеличения количества чисел, которые стоят на своих местах. Таким образом кусочек программы, отвечающий за сортировку можно переписать так:

```
        for i:=1 to N-1 do                      {по проходам}
                for j:=1 to N-i do              {по парам элементов}
                        if a[j]>a[j+1] then begin       {если в паре левый элемент больше правого}
                                temp:=a[j];             {меняем их местами}
                                a[j]:=a[j+1];           {если нет, то просто переходим к следующей}
                                a[j+1]:=temp;
                        end;
```

Как видите, изменился ровно один символ. В первом цикле разность N-1 заменена разностью N-i. Однако, количество операций (а вместе с ним и время работы) нам удалось сократить вдвое. Это легко увидеть, если нарисовать следующий график:




Площадь закрашенного треугольника будет как раз представлять количество сравнений. А треугольник есть ничто иное как половинка квадрата со стороной (N-1). Хотя, следует признать, что оценка времени работы алгоритма все еще составляет $O(N^2)$.

Второе соображение для оптимизации метода простого обмена основано на таком утверждении: *если за полный проход в массиве не сделано ни одной перестановки, то его можно считать отсортированным*. Что значит «не сделано ни одной перестановки»? Это значит, что все пары соседних чисел расположены «правильно», то есть большее число идет позже меньшего, поэтому они в перестановках не нуждаются. Это позволяет значительно сократить время в случаях, когда более или менее повезло с исходными данными.
Например, в массиве 8, 1, 2, 3, 4, 5, 6 будет вообще достаточно одного прохода, чтобы вытолкнуть восьмерку на последнее место.
Существенных изменений в структуре программы не будет – как был двойной цикл, так и остался. Просто внешний цикл будет заменен на цикл с условием. Программа в этом случае может выглядеть, например, так:

```
const
        N=100;
var
        a       : array [1..N] of Integer;
        i,j     : integer;
        temp    : integer;
        flag    : boolean;                      {признак того случались ли перестановки}
begin
        {ввод массива}

        i:=0;
        repeat
                inc(i);                         {проходы будем считать, чтобы не все из них}
                flag:=false;                    {делать до конца. перед проходом сбросим}
                for j:=1 to N-i do              {признак того, что перестановки были.}
                        if a[j]>a[j+1] then begin       {если в паре левый элемент больше правого}
                                temp:=a[j];             {меняем их местами}
                                a[j]:=a[j+1];           {если нет, то просто переходим к следующей}
                                a[j+1]:=temp;
                                flag:=true;             {если была перестановка – установим признак}
                        end;
```

```
        until not(flag);

        {массив упорядочен и готов к использованию}
end.
```

Стоит заметить, что в худшем случае (а именно, если на входе дан массив, упорядоченный в другую сторону) время работы все равно будет составлять $O(N^2)$ и по количеству операций сравнения программа не будет отличаться от предыдущего примера.

## Обмен с убывающими смещениями

Сразу оговоримся, что существует такой вариант исходных данных, при которых данный метод будет работать за время $O(N^2)$. Однако, в среднем случае этот метод способен дать значительный выигрыш по времени по сравнению с методом простого обмена.
Заметим следующее: каждый проход метода простого обмена способен сместить элемент массива не более чем на одну позицию влево. Поэтому, наибольшее время займет работа метода простого обмена в случаях, когда наименьший элемент находится на последнем месте. В методе обмена с убывающими смещениями добавляются несколько первых шагов, цель которых при необходимости поменять местами элементы, находящиеся друг от друга на расстоянии большем единицы. То есть поместить числа по возможности поближе к их окончательным позициям в массиве. Тогда количество шагов для метода простого обмена значительно сократится.

Итак, давайте рассмотрим работу этого метода на примере. Пусть дан массив из 8 чисел, который требуется упорядочить по возрастанию:

| 4 | 6 | 8 | 2 | 5 | 0 | 1 | 7 |
|---|---|---|---|---|---|---|---|

Выберем шаг d=[N/2] (в какую сторону округлять, в принципе не очень важно, обычно размеры массивов таковы, что на фоне тысяч элементов единичка при округлении как-то не замечается).
В нашем примере шаг будет равен d=[8/2]=4.
На первом проходе будем сравнивать пары элементов массива, расположенные на расстоянии d. В нашем примере это будут 1-й и 5-й, 2-й и 6-й, 3-й и 7-й, 4-й и 8-й. Как и ранее, в методе простого обмена, если расположение элементов в паре нас не устраивает – будем менять их местами.
Покажем подробно первый проход:

| **4** | 6 | 8 | 2 | **5** | 0 | 1 | 7 |
|---|---|---|---|---|---|---|---|

Шаг 1. Сравнить числа 4 и 5. Местами менять не будем.

| 4 | **6** | 8 | 2 | 5 | **0** | 1 | 7 |
|---|---|---|---|---|---|---|---|

Шаг 2. Сравнить числа 6 и 0. Поменять местами, чтобы большее стояло правее. Расположение чисел в массиве станет таким:

| 4 | 0 | 8 | 2 | 5 | 6 | 1 | 7 |
|---|---|---|---|---|---|---|---|

| 4 | 0 | **8** | 2 | 5 | 6 | **1** | 7 |
|---|---|---|---|---|---|---|---|

Шаг 3. Сравнить числа 8 и 1. Поменять местами. Теперь в массиве числа будут расположены так:

| 4 | 0 | 1 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

| 4 | 0 | 1 | **2** | 5 | 6 | 8 | **7** |
|---|---|---|---|---|---|---|---|

Шаг 4. Сравнить числа 2 и 7 и оставить их на своих местах.

После такого подготовительного прохода элементы массива расположены довольно близко к тем местам, которые они должны занять после сортировки. По крайней мере все «большие» элементы находятся в правой половине, все «маленькие» - в левой. Конечно, так будет далеко не всегда, но в среднем случае такие подготовительные действия могут быть очень полезны.
Далее, если d>1, то d уменьшаем в 2 раза и делаем еще один проход. Теперь будем сравнивать числа, находящиеся на расстоянии 2: 1-е и 3-е, 2-е и 4-е, 3-е и 5-е, 4-е и 6-е, 5-е и 7-е, 6-е и 8-е.

После первого прохода массив был таким:

| 4 | 0 | 1 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Разберем второй проход пошагово:

| 4 | 0 | 1 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Шаг 1. Сравним 4 и 1. Они расположены не так, как нам это нужно. Поменяем местами. Числа в массиве будут расположены так:

| 1 | 0 | 4 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

| 1 | 0 | 4 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Шаг 2. Сравним 0 и 2. Их расположение нас устраивает, поэтому просто перейдем к следующей паре.

| 1 | 0 | 4 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Шаг 3. Сравним 4 и 5. Эти числа должны остаться на своих местах.

| 1 | 0 | 4 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Шаг 4. Сравним 2 и 6. Переставлять также не надо.

| 1 | 0 | 4 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Шаг 5. Сравним 5 и 8. Опять переставлять не надо.

| 1 | 0 | 4 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Шаг 6. Сравним 6 и 7. И здесь также переставлять не надо.
Проход закончен. Получен массив:

| 1 | 0 | 4 | 2 | 5 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Теперь, уменьшив d в 2 раза, получим d=1. С этого момента метод работает как метод простого обмена с проверкой, случались ли перестановки элементов на последнем проходе или нет.
Что касается нашего примера, то для его сортировки достаточно двух проходов – одного чтобы расположить все числа в нужном порядке и второго, чтобы убедиться что перестановок не произойдет.
Структура программы не сильно изменилась по сравнению с предыдущими примерами – все тот же двойной цикл. Внешний цикл в случае необходимости изменяет величину шага d, и условием выхода из него является отсутствие перестановок при d=1. Внутренний цикл организует полный проход по массиву со сравнением всех пар элементов на расстоянии d.

Текст программы может выглядеть, например, так:

```
Const
        N = 100;

var
        a       : array [1..N] of integer;
        d       : integer;                      {расстояние между элементами при проходе}
        i       : integer;                      {переменная для организации прохода по массиву}
        flag    : boolean;                      {признак происходили ли перестановки}
        temp    : integer;

begin

{ввод массива}

d:=N;                                   { ! первая «принципиальная» строка. см. комментарий после }
```

```
                                    {текста программы}
 repeat
        flag:=false;                {сбросим признак перестановок перед проходом}
        if d>1 then d:=d div 2;     { ! вторая «принципиальная» строка}

        for i:=1 to N-d do          {проход по всему массиву со сравнением элементов на}
                if a[i]>a[i+d] then begin       {расстоянии d}
                        temp:=a[i];
                        a[i]:=a[i+d];
                        a[i+d]:=temp;
                        flag:=true;
       end;

  until (d=1)and(not(flag));        {когда расстояние стало равно 1 и перестановки кончились,}
                                    {массив можно считать отсортированным}

{массив упорядочен и готов к использованию}
end.
```

Чем же так принципиальны те строки, которые выделены в тексте программы? Своим расположением. Порядком выполнения. Наиболее часто встречающейся ошибкой в реализации этого метода является такой порядок действий:

```
d:=N div 2; {сразу}
repeat
        {проход по массиву с возможными перестановками}
        if d>1 then d:=d div 2;
until (d=1)and(not(flag));
```

В случае такого написания метод может просто перестать работать. Допустим, что только что закончился проход с шагом d=2, и на этом проходе не было перестановок. Следует ли из этого, что массив упорядочен? Нет. Далее сразу после прохода следует уменьшение d и он становится равным 1. И тут мы подходим к проверке условия окончания цикла. И оно выполняется. Чего быть не должно. Потому что с шагом d=1 прохода еще не было.

## Вопросы и упражнения

1. Напишите программу, проверяющую является ли введенный массив из 10 целых чисел упорядоченным по убыванию.
2. Возможна следующая реализация метода простого обмена, в котором признаком того были ли перестановки во время последнего прохода, служит не отдельная переменная, а та, которая используется для обмена элементов:

```
        repeat
                inc(i);
                temp:=0;
                for j:=1 to N-i do
                        if a[j]>a[j+1] then begin
                                temp:=a[j];
                                a[j]:=a[j+1];
                                a[j+1]:=temp;
                        end;
        until temp=0;
```

Укажите достоинства и недостатки такого решения. В каких случаях такая реализация может привести к неработоспособности алгоритма?

3. Что произойдет, если в приведенной реализации метода обмена с убывающими смещениями в строке

if d>1 then d:=d div 2; убрать условие и записать ее просто как d:=d div 2;?

4. Приведите пример неупорядоченного массива из 8 элементов, в котором при шаге d=2 в методе обмена с убывающим смещением, перестановок не будет.

5. Для N=8 найдите такое расположение чисел в исходном массиве, что первые шаги метода обмена с убывающим смещением (при изменении шага d по закону d:=d div 2) не приведут к улучшению этого расположения.