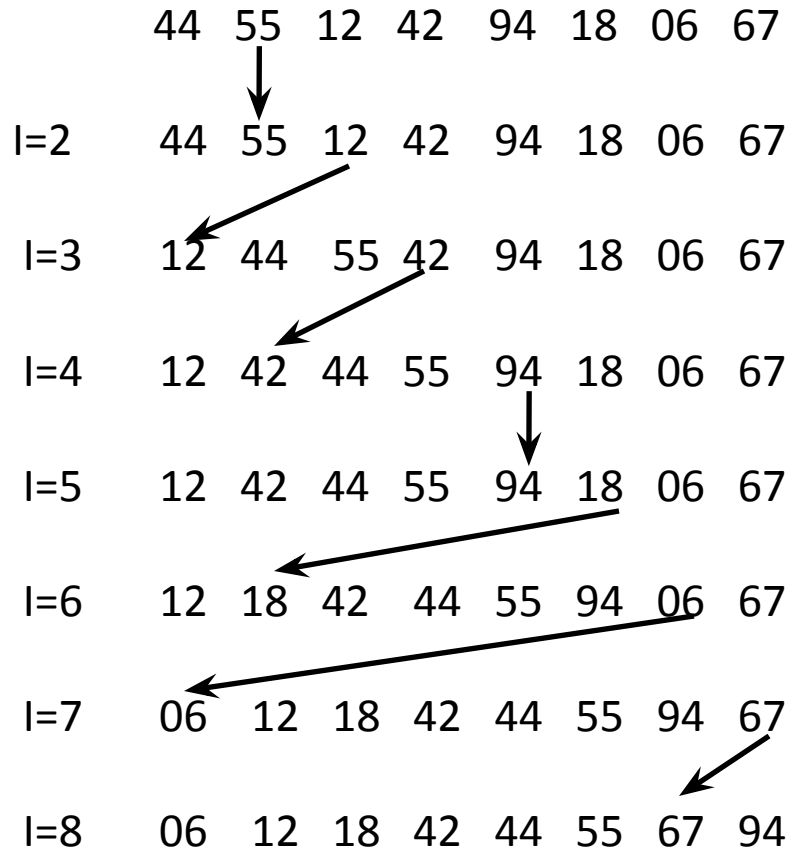


## Сортировка простыми включениями



```
For i: = 2 to n do
  Begin
    x: = a[i]; a[0]: =x;
    j: =j-1;
    While x<a[j] do
      Begin
        a[j+1]: = a[j];
        j: = j-1;
      End;
    a[j+1]: =x;
  end;
end;
```

Наименьшее число сравнений  
появятся, если элементы с самого  
начала упорядочены, а наихудшее,  
если элементы расположены в  
обратном порядке, т.е. сортировка  
простыми включениями  
демонстрирует естественное  
поведение, кроме того алгоритм  
описывает устойчивую сортировку,  
т.е. оставляет неизменным порядок  
элементов с одинаковыми ключами

Алгоритм сортировки простыми включениями можно улучшить тем, что готовая последовательность  $a_1, a_2, \dots, a_{i-1}$  уже упорядочена. Можно применить бинарный поиск, пока не будет найдено место включения.

Модифицированный алгоритм называется **сортировка бинарными включениями**.


```
For i: = 2 to n do
  begin
    x: = a[i]; L: =1; R: = i-1;
    While L<=R do
      begin
        m: = (L+R) div 2;
        If x<a[m] then R: = m-1
          else L: = m+1
      end;
    For j: =i-1 downto L do a[j+1]: =a[j];
    a[L]: = x;
  end;
```


**Анализ алгоритма сортировки бинарными включениями:** места включения в нижней части находятся несколько быстрее, чем в верхней части. Это дает преимущество в тех случаях, когда элементы изначально далеки от правильного порядка. Минимальное число сравнений требуется, если элементы вначале расположены в обратном порядке, а максимальное, – если он уже упорядочены. Следовательно, имеем случай неестественного поведения алгоритма сортировки. Улучшение, которое мы получаем, используя метод бинарного поиска, касается только числа сравнений, а не числа необходимых пересылок. Пересылка требует больше времени, чем сравнение, т.е. это улучшение не является решающим.


**Вывод:** сортировка включениями является не очень подходящим методом для ЭВМ. Лучших результатов можно ожидать от методов, при которых пересылки элементов выполняются только для отдельных

Метод основан на следующем правиле:  
Выбирается элемент с наименьшим ключом  
Меняется местами с первым элементом.


Эти операции повторяются с оставшимися  $n-1$  элементами,  
затем с  $n-2$  и т.д.

44 55 12 42 94 18 06 67  



06 55 12 42 94 18 44 67  


06 12 55 42 94 18 44 67  


06 12 18 42 94 55 44 67

6 12 18 42 94 55 44 67  


06 12 18 42 44 55 94 67

06 12 18 42 44 55 94 67  


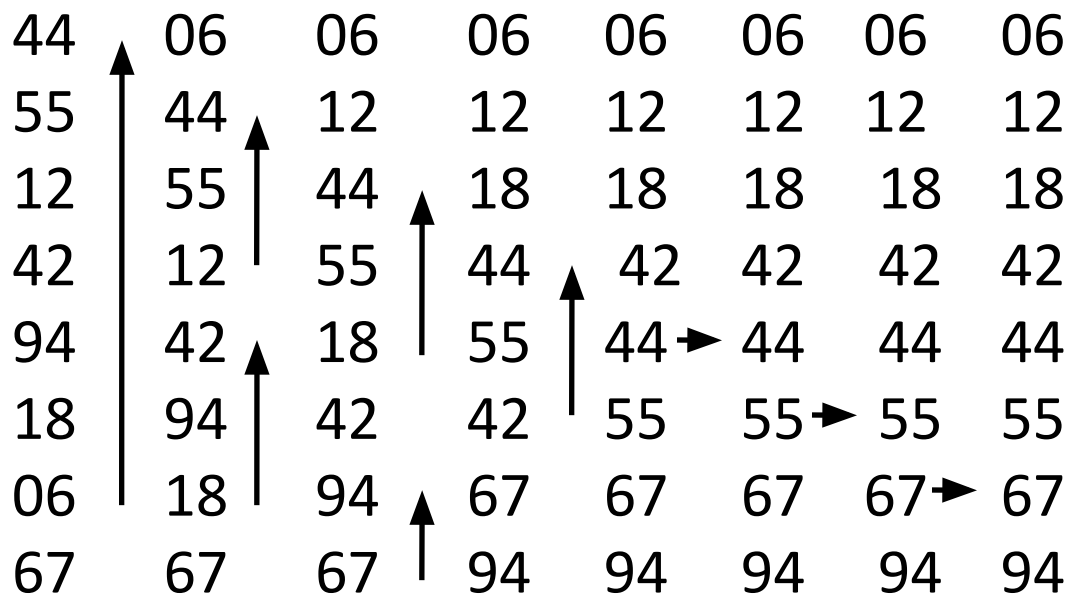
06 12 18 42 44 55 67 94

```
Var i, j, k: index; x: item;
begin
  For i: = 1 to n-1 do
    begin
      k: =i; x: = a[i];
      For j: =i+1 to n do
        If a[j] < x then begin
          k: =j;
          x: = a[j];
        end;
      a[k]: = a[i]; a[i]: =x;
    end;
  end;
```

### **Анализ сортировки простым выбором**

**Число сравнений ключей не зависит от начального порядка ключей.** Т.е. сортировка простым выбором ведет себя несколько менее естественно, чем сортировка простыми включениями. **Минимальное число пересылок получается в случае изначально упорядоченных ключей и максимальное, – если ключи расположены в обратном порядке.**

Алгоритм основан на принципе сравнения пары соседних элементов до тех пор, пока не будут отсортированы.





```
For i: =2 to n do
  Begin
    For j: = n downto i do
      If a[j-1] > a[j] Then
        Begin
          x: =a[j-1];
          a[j-1]: =a[j];
          a[j]: = x
        end;
      end;
    end;
  end;
```

1) Алгоритм можно оптимизировать (в примере 3 последних прохода не влияют на порядок элементов). Поэтому можно запомнить производился ли на данном проходе какой-либо обмен. Если нет, то алгоритм может закончить работу.

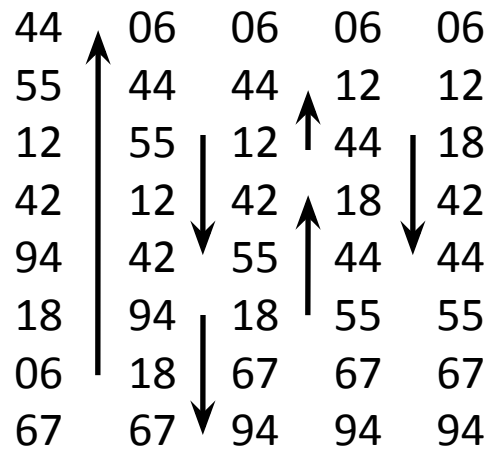
2) Можно запоминать не только сам факт обмена, но и индекс последнего обмена, т.к. элементы, которые расположены выше уже рассортированы.

3) **12 18 42 44 55 67 94 06**  
**94 06 12 18 42 44 55 67**

Неправильно расположенный пузырек в тяжелом конце массива всплывет на свое место за один проход, а неправильно расположенный пузырек в легком конце массива будет опускаться на правильное место только на один шаг на каждом проходе. Следовательно, предполагается менять, следующих друг за другом проходов.

Полученный в результате алгоритм называется **шейкер-сортировка**.

L=2        3        3        4        4  
R=8        8        7        7        4



```

Var i, j, k, L, R: Index; x: Item;
begin
  L: =2; R: =n; k: =n;
  repeat
    for j: =r downto L do
      if a[j-1] > a[j] then begin
        x: = a[j-1];
        a[j-1]: = a[j];
        a[j]: =x; k: =j;
      end;

    L: =k+1;
    for j: =L to R do
      if a[j-1] > a[j] then begin
        x: = a[j-1];
        a[j-1]: = a[j];
        a[j]: =x; k: =j;
      end ;

    R: =k-1;
  until L>R;
end;

```

**Анализ пузырьковой сортировки и шейкер-сортировки.** Все предложенные усовершенствования **уменьшают лишь число избыточных проверок**, но никак **не влияют на число обмена**. Все усовершенствования дают гораздо меньший эффект, чем можно было ожидать. Анализ показывает, что **сортировка обменом и ее небольшие улучшения хуже, чем сортировка включениями и выбором**. Алгоритм шейкер-сортировки выгодно использовать в тех случаях, когда **известно, что элементы почти упорядочены**. Можно показать среднее расстояние, на которое должен переместиться каждый из  $n$  элементов сортировки. Оно равно  $n/3$ . Все простые методы в основном перемещают каждый элемент на одну позицию на каждом элементарном шаге, поэтому они требуют  $n^2$  таких шагов. Любое улучшение должно основываться на принципе пересылки элементов за один шаг на большее расстояние.

## Сортировка включениями с убывающим приращением.

Ее иногда называют **сортировкой Шелла**.

44 55 12 42 94 18 06 67 – 4 – сортировка

44 18 06 42 94 55 12 67 – 2 – сортировка

06 18 12 42 44 55 94 67 – 1 – сортировка

06 12 18 42 44 55 67 94

На первом проходе группируются и сортируются все элементы, отстоящие друг от друга на 4 позиции. После этого элементы вновь объединяются в группы с элементами отстоящими друг от друга на 2 позиции и сортируются заново. На третьем проходе элементы сортируются обычной сортировкой.

На каждом шаге сортировки либо участвует сравнительно мало элементов, либо они уже довольно хорошо упорядочены и требуют относительно мало перестановок. Каждый проход использует результаты предыдущего прохода, поскольку каждая  $i$  – сортировка объединяет 2 группы, рассматриваемой предыдущей  $2i$  – сортировкой. Применима любая последовательность приращений, лишь бы последнее было равно единице, т.к. в худшем случае вся работа будет выполнена на последнем проходе. Оказывается, что программа дает лучший результат, когда приращения не являются степенями двойки. Программа разрабатывается вне связи с конкретной последовательностью приращений  $h_1, h_2, \dots, h_t$ ;  $h_t = 1$ ;  $h_{i+1} < h_i$ .

**Каждая  $h$ -сортировка программируется как сортировка простыми включениями.** Для того чтобы условие окончания поиска места включения было простым, используют барьер, но каждая  $h$ - сортировка своего барьера.

**Массив  $a$  необходимо дополнить не одной компонентой  $a[0]$ , а  $h_1$  компонентами:  $a:\text{array}[-h_1..n]$  of item**

```

const t=4;
var
  i, j, k, s: index; x: Item;
  m: 1..t;
  h: array[1..t] of integer;
begin
  h[1]=9; h[2]= 5; h[3]=3; h[4]=1;
  for m: =1 to t do
    begin
      k: = h[m]; s: =-k;{место барьера}
      for i: =k+1 to n do
        begin
          x: =a[i]; j: =i-k;
          if s=0 Then s: =-k; s: =s+1; a[s]; =x;
          while x < a[j] do
            begin
              a[j+k]: = a[j]; j: =j-k;
            end;
            a[j+k]: =x;
          end;{for...}
        end;{for...}
      end;
    end;
  end;
end;

```



**Анализ сортировки Шелла:** основной вопрос в этой сортировке: какую выбрать последовательность приращений. Например, считается, что эти приращения не должны быть кратны друг другу, тогда получается лучший результат.

**Кнут** предлагает:

$$\begin{array}{l} 1, 4, 13, 40, 121 \quad \{h_{k-1} = 3h_k + 1\} \\ 1, 3, 7, 15, 31 \quad \{h_{k-1} = 2h_k + 1\} \end{array}$$

Анализ показывает, что затраты, которые требуются для сортировки  $n$  элементов (Шелла), **пропорциональны  $n$** . Ясно, что это значительное улучшение по сравнению с предыдущими. Но есть алгоритмы, работающие еще лучше.

## Сортировка с разделением

Иначе ее называют «быстрая сортировка» (сортировка Хоара). Основана на том факте, что для достижения большей эффективности желательно производить обмены элементов на больших расстояниях.

Предположим, дано  $n$  элементов с ключами, расположенными в обратном порядке. Их можно рассортировать, выполнив  $n/2$  обменов. Поменять местами самый левый и самый правый элементы, постепенно продвигаясь к середине.

### Алгоритм



Выберем случайным образом какой-либо элемент массива ( $X$ ), посмотрим массив, двигаясь слева направо, пока не найдем элемент  $a[i] > X$ . А затем посмотрим массив, двигаясь справа налево, пока не найдем элемент  $a[j] < X$  и поменяем местами эти два элемента. Затем продолжим процесс просмотра с обменом, пока 2 просмотра не встретятся где-то в середине.

```
var w, x: item;
    i, j: integer;
begin
    i: =1; j: =n;
    {Выбор случайного элемента }
    repeat
        while a[i] <x do i: =i+1;
        while x < a[j] do j: =j-1;
        if i<= j then begin
            w: =a[i]; a[i]: =a[j]; a[j]: =w;
            i: =i+1; j: =j-1;
        end;
    until i>j
end;
```

Пример:

44 55 12 42 94 06 18 67

Для такого массива потребуется всего 2 прохода и конечное значение индексов  $i=5, j=3$ .

18 06 12 42 94 55 44 67

Алгоритм может быть неуклюжим: массив с  $n$  одинаковыми ключами.

**Наша цель** не только разделить исходный массив на две части большую и меньшую  $x$ , но и рассортировать его. Разделив массив надо сделать тоже самое с обеими полученными частями, затем с частями этих частей и т.д., пока каждая часть не будет содержать только один элемент, т. е. **надо применить рекурсию**.

Procedure QuicSort;

Procedure Sort (L, R: index);

var i, j: index; x, w: item;

begin

  i: =L; j: =R; x: =a[(L+R)div 2];

  repeat

    while a[i] < x do i: =i+1;

    while a[j] > x do j: =j-1;

    if i <= j then begin

      w: =a[i]; a[i]: =a[j]; a[j]: =w;

      i: =i+1; j: =j-1;

    end;

  until i > j;

    if L < j then Sort (L, j);

    if i < R then Sort (i, R);

end; {sort}

begin

  Sort (1, n);

end;

## Анализ сортировки Хоара

В лучшем случае - число сравнений –  $O(n \cdot \log n)$ ;

число обменов –  $(n/6) \cdot \log n$

Но QS имеет «подводные камни». При небольших  $n$  её эффективность невелика. Кроме того, существует проблема наихудшего случая – когда в качестве  $x$  выбирается наибольшее значение в подмассиве – скорость работы оказывается  $O(n^2)$