Тема 1.2.

Алгоритмы внутренней сортировки

Учитывая текущее плачевное состояние наших программ, можно сказать, что программирование определенно все ещё черная магия и, пока, мы не можем называть его технической дисциплиной.
Bill Clinton

Содержание

1. Постановка задачи сортировки
2. Простые алгоритмы сортировки
3. Быстрые алгоритмы сортировки

2. Постановка задачи сортировки

Эта Тема 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).

В этой Теме рассматриваются только внутренние методы сортировки.

Их важная особенность состоит в том, что эти алгоритмы не требуют дополнительной памяти:

- вся работа по упорядочению производится внутри *одного и того же массива*.

Под ***сортировкой последовательности*** понимают процесс перестановки элементов последовательности в определенном порядке: по возрастанию, убыванию, последней цифре, сумме делителей, … .

Пусть дана последовательность элементов $a_1, a_2, \ldots, a_n$.

Элементы этой последовательности – данные произвольного типа, на котором определено *отношение порядка* “**<**” (меньше) такое, что любые два различных элемента сравнимы между собой.

Сортировка означает перестановку элементов последовательности $a_{k1}, a_{k2}, \ldots, a_{kn}$ такую, что

$$a_{k1} < a_{k2} < \ldots < a_{kn}.$$

Основными требованиями к программе сортировки массива являются эффективность по времени и экономное использование памяти.

Это означает, что алгоритм не должен использовать дополнительных массивов и пересылок из массива ***a*** в эти массивы.

Постановка задачи сортировки в общем виде предполагает, что существуют только два типа действий с данными сортируемого типа:

- сравнение двух элементов (***x<y***)
- и пересылка элемента (***x:=y***).

Поэтому удобная мера сложности алгоритма сортировки массива ***a[1..n]***:

- по времени – количество сравнений ***C(n)***
- и количество пересылок ***M(n)***.

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. Этот факт принято обозначать следующей символикой: $O(N^2)$.

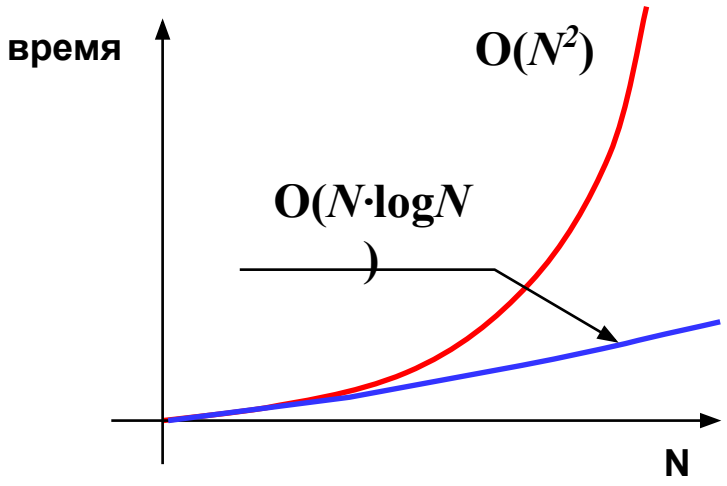
Сортировка

Алгоритмы:

- простые и понятные, но неэффективные для больших массивов
  - **метод пузырька**
  - **метод выбора**
- сложные, но эффективные
  - **«быстрая сортировка» (*Quick Sort*)**
  - **сортировка «кучей» (*Heap Sort*)**
  - **сортировка слиянием**
  - **пирамидальная сортировка**

сложность $O(N^2)$

сложность $O(N \cdot \log N)$



Простые сортировки

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.

Простые сортировки

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

$$(Eff_best + Eff_worst)/2.$$

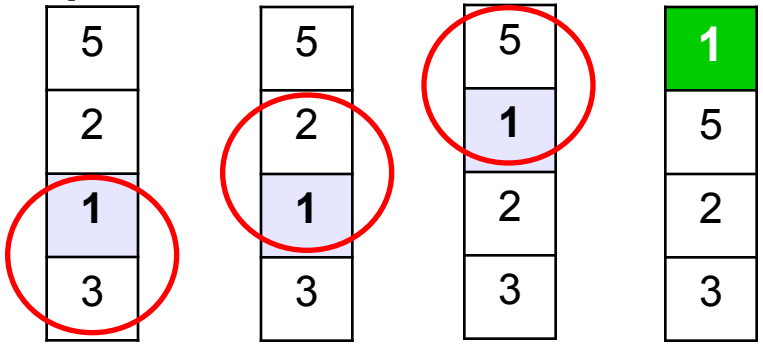
3. Простые алгоритмы сортировки

Метод пузырька

Идея – пузырек воздуха в стакане воды поднимается со дна вверх.

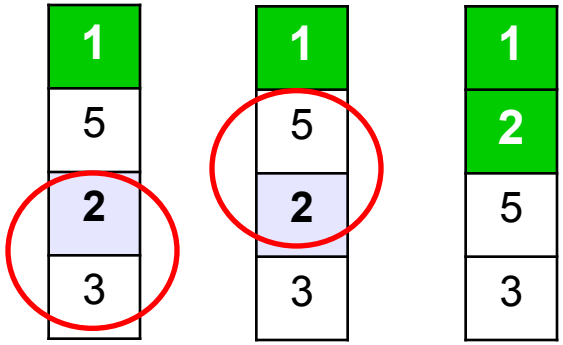
Для массивов – самый маленький («легкий» элемент перемещается вверх («всплывает»).

1-ый проход

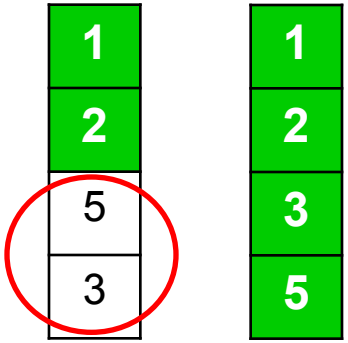


- начиная снизу, сравниваем два соседних элемента; если они стоят «неправильно», меняем их местами
- за 1 проход по массиву **один** элемент (самый маленький) становится на свое место

2-ой проход



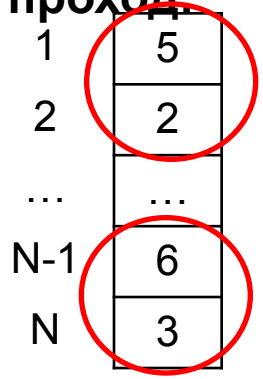
3-ий проход



Для сортировки массива из N элементов нужен N-1 проход (достаточно поставить на свои места N-1 элементов).

Программа

1-ый проход:



сравниваются пары

`A[N-1] и A[N],  A[N-2] и A[N-1]`

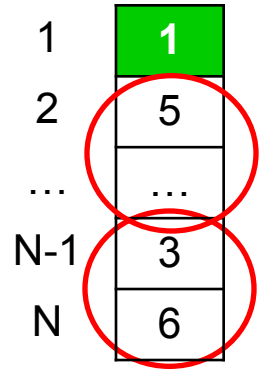
…

`A[1] и A[2]`

`A[j] и A[j+1]`

```
for j:=N-1 downto 1 do
  if A[j] > A[j+1] then begin
    c:=A[j]; A[j]:=A[j+1]; A[j+1]:=c;
  end;
```

2-ой проход



A[1] уже на своем месте!

```
for j:=N-1 downto 2 do
  if A[j] > A[j+1] then begin
    c:=A[j]; A[j]:=A[j+1]; A[j+1]:=c;
  end;
```

i-ый проход

```
for j:=N-1 downto i do
  ...
```

Программа

```
program qq;
const N = 10;
var A: array[1..N] of integer;
    i, j, c: integer;
begin
  { заполнить массив }
  { вывести исходный массив }
  for i:=1 to N-1 do begin
    for j:=N-1 downto i do
      if A[j] > A[j+1] then begin
        c := A[j];
        A[j] := A[j+1];
        A[j+1] := c;
      end;
  end;
  { вывести полученный массив }
end.
```

Почему цикл по i до N-1?

элементы выше **A[i]** уже поставлены

Эффективность метода пузырька

Внешний цикл выполнился ***n*–1** раз. Внутренний цикл выполняется ***j*** раз (***K* = *n*–2, *n*–1, ..., 1**).

Каждое выполнение тела внутреннего цикла заключается в одном сравнении и, возможно, в одной перестановке. Поэтому

$$C(n) = 1+2+ \ldots + n-1 = n*(n-1)/2,$$

$$M(n) = n*(n-1)/2.$$

В худшем случае (когда элементы исходного массива расположены в порядке убывания)

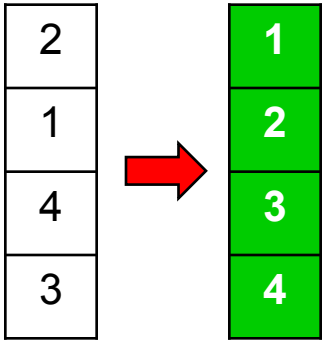
$$C(n) = n*(n-1)/2 = O(n^2),$$

$$M(n) = n*(n-1)/2 = O(n^2).$$

Метод пузырька с флажком

Идея – если при выполнении метода пузырька не было обменов, массив уже отсортирован и остальные проходы не нужны.

Реализация: **переменная-флаг**, показывающая, был ли обмен; если она равна **`False`**, то выход.




```
var flag: boolean;
```

```
repeat
  flag := False; { сбросить флаг }
  for j:=N-1 downto 1 do
    if A[j] > A[j+1] then begin
      c := A[j];
      A[j] := A[j+1];
      A[j+1] := c;
      flag := True; { поднять флаг }
    end;
until not flag;  { выход при flag=False }
```

? Как улучшить?

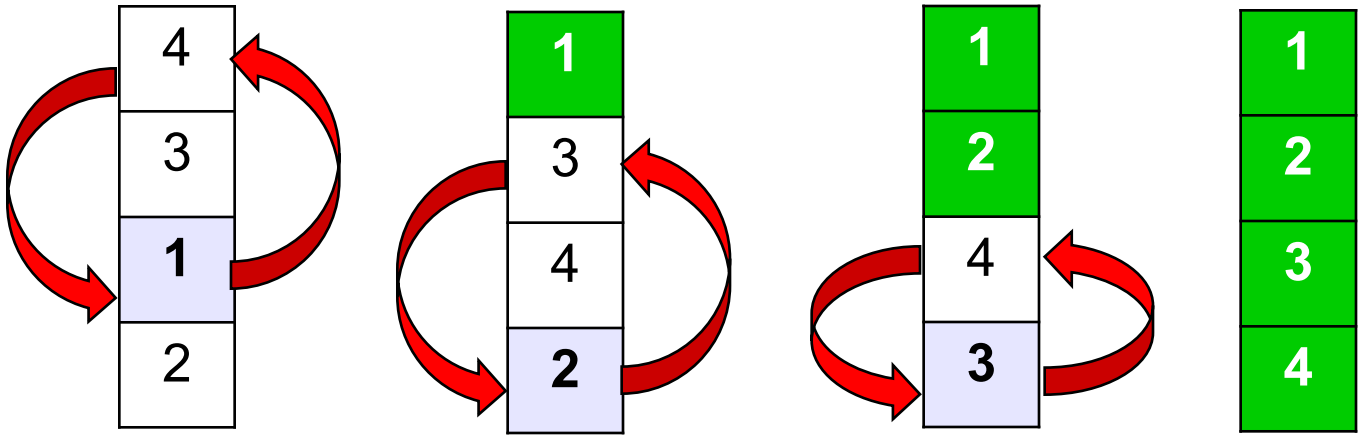
Метод пузырька с флажком

```
  i := 0;
repeat
  i := i + 1;
  flag := False; { сбросить флаг }
  for j:=N-1 downto i do
    if A[j] > A[j+1] then begin
      c := A[j];
      A[j] := A[j+1];
      A[j+1] := c;
      flag := True; { поднять флаг }
    end;
until not flag;  { выход при flag=False }
```

Метод выбора

Идея:

- найти минимальный элемент и поставить на первое место (поменять местами с **`A[1]`**)
- **из оставшихся** найти минимальный элемент и поставить на второе место (поменять местами с **`A[2]`**), и т.д.



Метод выбора

нужно N-1 проходов

```
for i := 1 to N-1 do begin
  nMin = i;
  for j:= i+1 to N do
    if A[j] < A[nMin] then nMin:=j;
  if nMin <> i then begin
    c:=A[i];
    A[i]:=A[nMin];
    A[nMin]:=c;
  end;
end;
```

поиск минимального от A[i] до A[N]

если нужно, переставляем

? Можно ли убрать if?

Сортировка простыми вставками

Самый простой способ сортировки, 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.

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.

Сортировка простыми вставками

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.

Сортировка простыми вставками

Фрагмент программы:

```
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

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***.

Метод прямых вставок с барьером

Фрагмент программы:

```
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;
```

Метод прямых вставок с барьером

Эффективность алгоритма.

Понятно, что для этой сортировки наилучшим будет случай, когда на вход подается уже упорядоченная последовательность данных. Тогда алгоритм совершит

- ***N*-1** сравнение
- и **0** пересылок данных.

В худшем же случае - когда входная последовательность упорядочена "наоборот" будет

- сравнений ***(N+1)*N/2***,
- а пересылок ***(N-1)*(N+3)***.

Таким образом, этот алгоритм имеет сложность $O(N^2)$ по обоим параметрам.

Метод прямых вставок с барьером

Пример сортировки.

Предположим, что нужно отсортировать следующий набор чисел:

5 3 4 3 6 2 1

Выполняя алгоритм, 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	1+2
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.

Сортировка бинарными вставками

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.

Сортировка бинарными вставками

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.

Сортировка бинарными вставками

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

Сортировка бинарными вставками

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.

Сортировка бинарными вставками

Будет ли такой алгоритм универсальным?

Давайте проверим, что же произойдет, если мы станем искать позицию 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.

Сортировка бинарными вставками

Добавим число 21 в последовательность.

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.

Сортировка бинарными вставками

Фрагмент программы:

```
for  i:= 2  to  n  do
     if  a[i-1]>a[i]  then  begin
           x:= a[i];
           left:= 1;   right:= i-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;
```

Сортировка бинарными вставками

Эффективность алгоритма.

Теперь на каждом шаге выполняется не N, а $log_2 N$ проверок, что уже значительно лучше (для примера, сравните 1000 и $10 = log_2 1024$).

Следовательно, всего будет совершено $N * log_2 N$ сравнений.

По количеству пересылок алгоритм по-прежнему имеет сложность $O(N^2)$.

4. Быстрые алгоритмы сортировки

В отличие от простых сортировок, имеющих сложность ***O(N^2)***, к ***улучшенным сортировкам*** относятся алгоритмы с общей сложностью ***O(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 базируется на уже известном нам алгоритме простых вставок.

Смысл ее состоит в раздельной сортировке методом простых вставок 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.

Сортировка Шелла

Сортировку Шелла придумал Дональд Л. Шелл.

Ее необычность состоит в том, что она рассматривает весь список как совокупность перемешанных подсписков.

На первом шаге эти подсписки представляют собой просто пары элементов.

На втором шаге в каждой группе по четыре элемента.

При повторении процесса число элементов в каждом подсписке увеличивается, а число подсписков, соответственно, падает.

Сортировка Шелла

Сортирует элементы массива ***A[1..n]*** следующим образом:

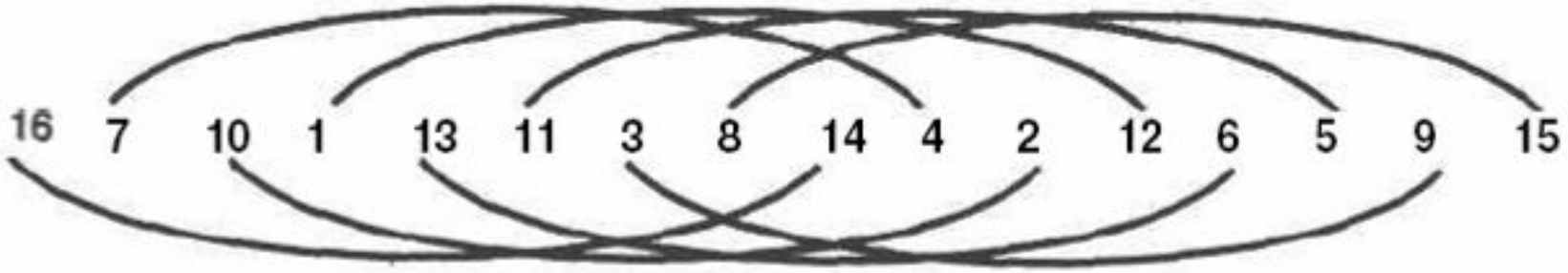
- на первом шаге упорядочиваются элементы ***n*/2** пар (***A[i], A[n/2 + i]***) для **1 <** ***i*** **<** ***n*/2**,
- на втором шаге упорядочиваются элементы в ***n*/4** группах из четырех элементов (***A[i], A[n/4 + i], A[n/2 + i], A[3n/4 + i]***) для **1 <** ***i*** **<** ***n*/4**,
- на третьем шаге упорядочиваются элементы в ***n*/8** группах из восьми элементов и т.д.;
- на последнем шаге упорядочиваются элементы сразу во всем массиве ***A***.

На каждом шаге для упорядочивания элементов используется метод сортировки вставками.

Сортировка Шелла

На рис. а изображены восемь подсписков, по два элемента в каждом, в которых

- ✔ первый подсписок содержит первый и девятый элементы,
- ✔ второй подсписок — второй и десятый элементы,
- ✔ и так далее.

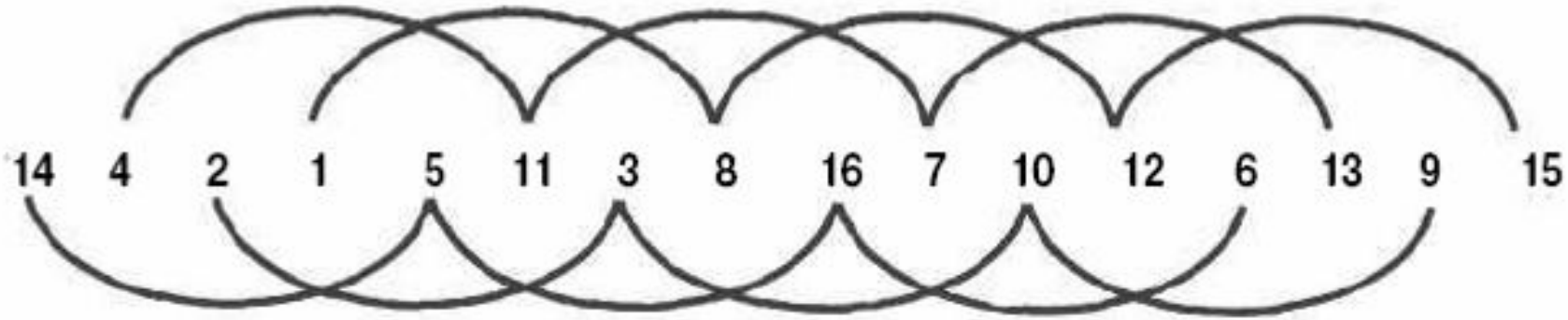


(а) Проход 1

Сортировка Шелла

На рис. б мы видим уже четыре подсписка по четыре элемента в каждом:

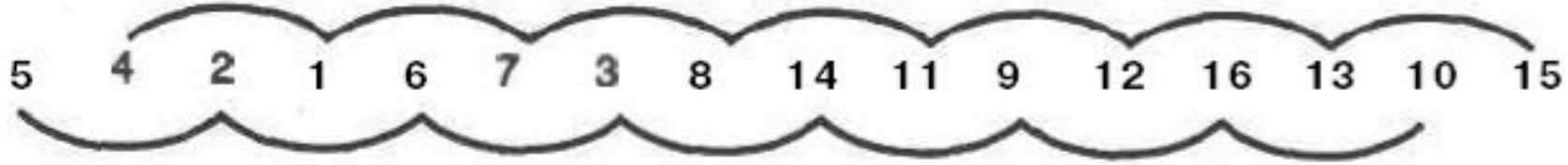
- ✔ первый подсписок на этот раз содержит первый, пятый, девятый и тринадцатый элементы.
- ✔ второй подсписок состоит из второго, шестого, десятого и четырнадцатого элементов.



(б) Проход 2

Сортировка Шелла

На рис. в показаны два подсписка, состоящие из элементов с нечетными и четными номерами соответственно.



(в) Проход 3

Сортировка Шелла

На рис. г мы вновь возвращаемся к одному списку.

(г) Проход 4

Сортировка Шелла

Фрагмент программы:

```
incr:= n div 2;
while  incr>0 do begin
  for  i:=incr+1 to n do  begin
   j:= i-incr;
   while  j>0 do
     if A[j]>A[j+incr] then  begin
        c:= A[j]; A[j]:=A[j+incr];
        A[j+incr]:=c;
        j:=j-incr
     end
     else   j:=0    { останов проверки}
  end;
  incr:= incr div 2
end;
```

Сортировка Шелла

Полный анализ сортировки Шелла чрезвычайно сложен, и мы не собираемся на нем останавливаться.

Было доказано, что сложность этого алгоритма в наихудшем случае при выбранных нами значениях шага равна $O(n^{3/2})$.

Полный анализ сортировки Шелла и влияния на сложность последовательности шагов можно найти в третьем томе книги Д. Кнута *Искусство программирования,* М., Мир.

Пирамидальная сортировка

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

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.

Пирамидальная сортировка: 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 последовательно задерживаются сначала крупные, а затем все более мелкие частицы.

Пирамидальная сортировка: просеивание

Итак, будем рассматривать наш линейный массив как пирамидальную структуру:

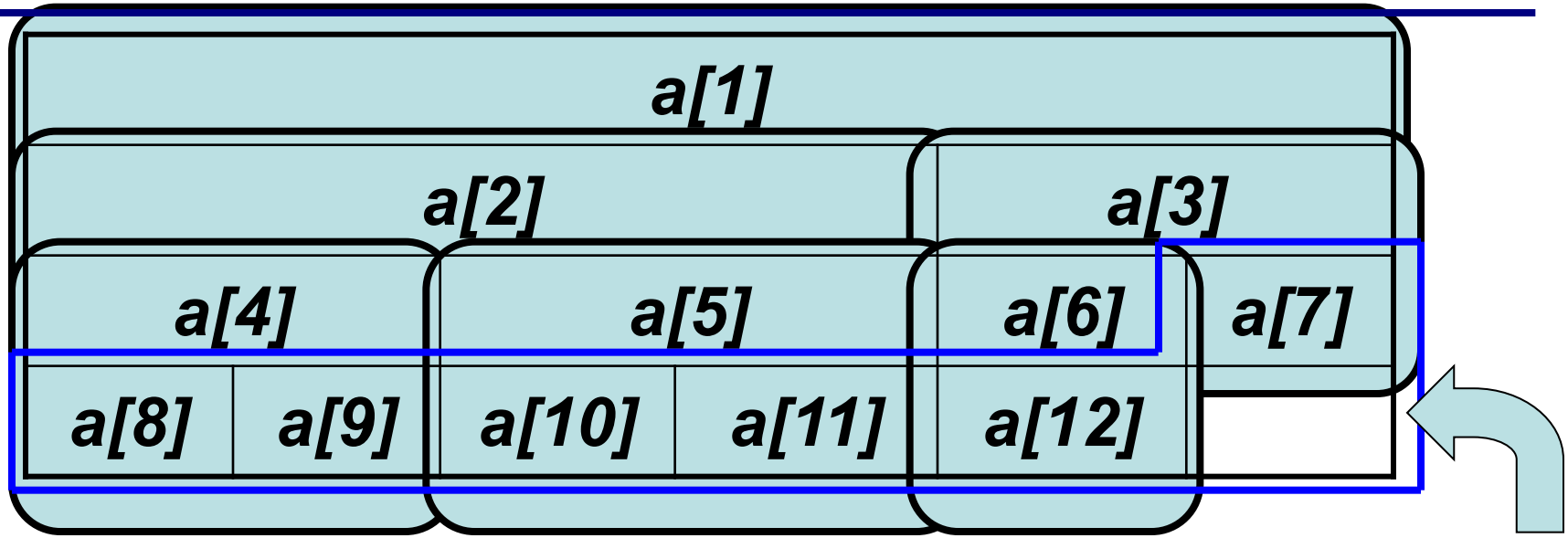
<table>
<tr><td colspan="8">a[1]</td></tr>
<tr><td colspan="6">a[2]</td><td colspan="2">a[3]</td></tr>
<tr><td colspan="4">a[4]</td><td colspan="2">a[5]</td><td>a[6]</td><td>a[7]</td></tr>
<tr><td>a[8]</td><td>a[9]</td><td>a[10]</td><td>a[11]</td><td>a[12]</td><td></td></tr>
</table>

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.

Пирамидальная сортировка: просеивание



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) проверяем 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.

Пирамидальная сортировка: просеивание

Фрагмент программы алгоритма просеивания:

```
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;
```

Пирамидальная сортировка: просеивание

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):

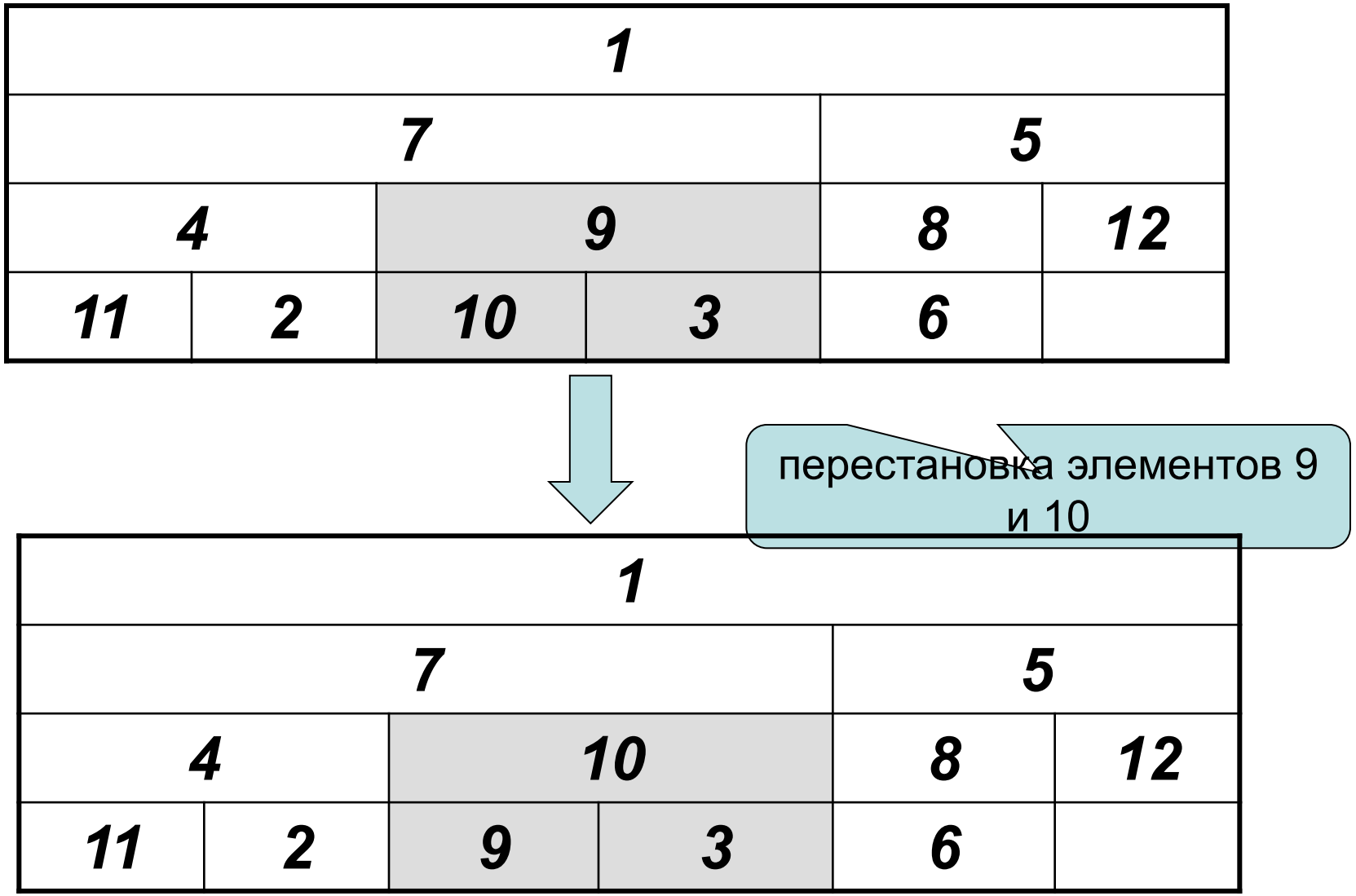
<table>
<tr><td colspan="6">1</td></tr>
<tr><td colspan="4">7</td><td colspan="2">5</td></tr>
<tr><td colspan="2">4</td><td colspan="2">9</td><td>8</td><td>12</td></tr>
<tr><td>11</td><td>2</td><td>10</td><td>3</td><td>6</td><td></td></tr>
</table>

Пирамидальная сортировка: просеивание

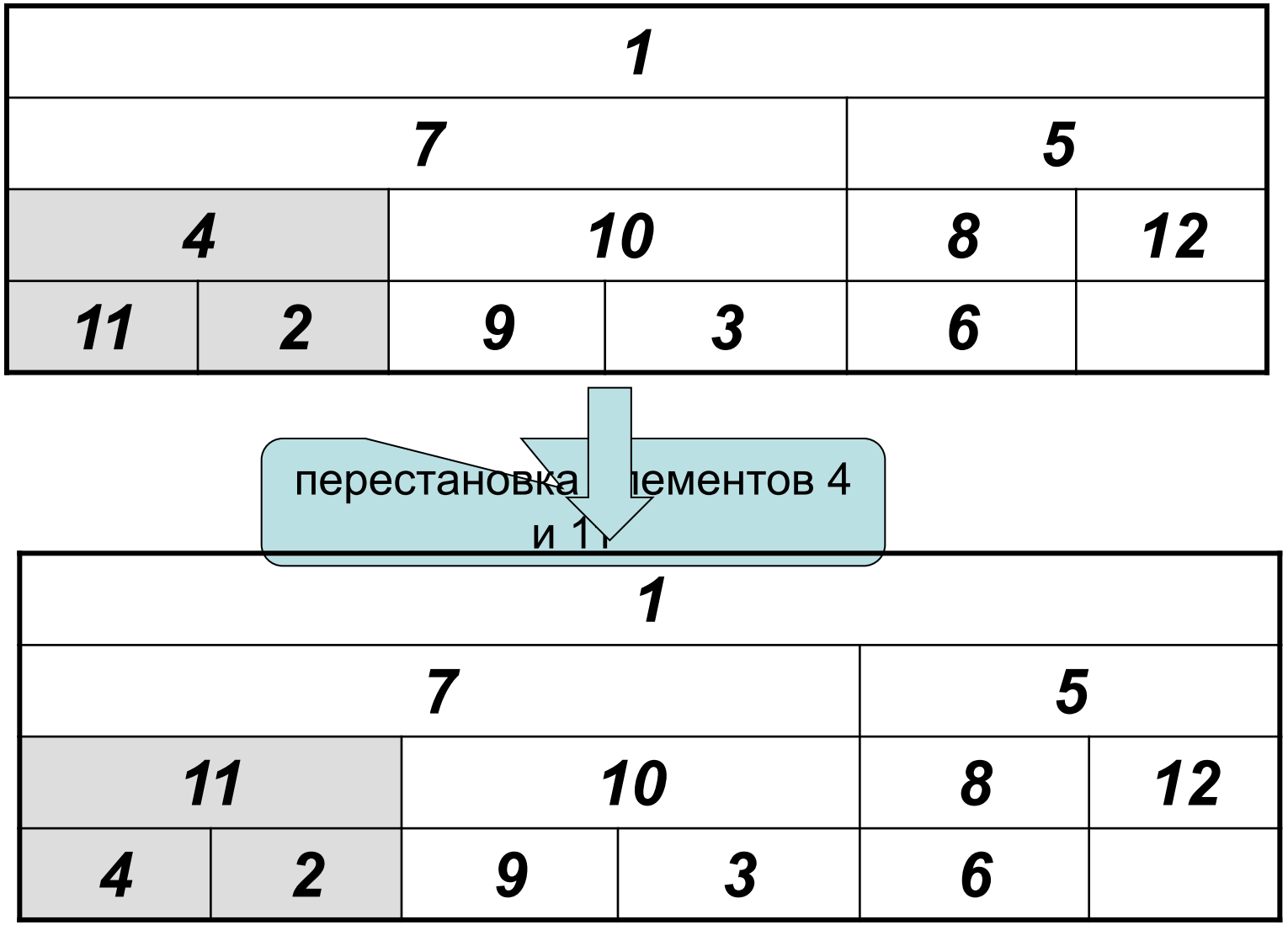
1					
7				5	
4		9		8	12
11	2	10	3	6	

перестановка не требуется

Пирамидальная сортировка: просеивание



Пирамидальная сортировка: просеивание



Пирамидальная сортировка: просеивание

Пирамидальная сортировка: просеивание

Пирамидальная сортировка: просеивание

Пирамидальная сортировка: просеивание

Итак, мы превратили исходный массив в пирамиду:

- в любой тройке ***a[i]***, ***a[2*i]*** и ***a[2*i+1]*** максимум находится "сверху".

Пирамидальная сортировка: алгоритм

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).

Пирамидальная сортировка

Часть программы, реализующая нулевой шаг алгоритма, приведена в пункте "Просеивание", поэтому здесь приведена только реализацией основного шага 1:

```
for i:= N downto 2 do begin
  x:=a[1];  a[1]:=a[i];  a[i]:=x;
  j:= 1;  flag:=true;
  while (j<=((i-1)div 2)) and flag 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   flag:=false
  end
end;
```

Пирамидальная сортировка: 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 не будем далее прорисовывать структуру пирамиды.

Подчеркивание будет отмечать элементы, участвовавшие в просеивании, а квадратными скобками – те элементы, которые участвуют в дальнейшей обработке.

Пирамидальная сортировка: пример

1) Меняем местами ***a[1]*** и ***a[12]***:

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

2) Просеивая элемент ***a[1],*** последовательно получаем:

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

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

   [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) Просеивая ***a[1]***, последовательно получаем:

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

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

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

Пирамидальная сортировка: пример

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, 1, 8, 7, 3, 6, 5, 4, 2], 10, 11, 12;

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

[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, 2, 4, 3, 6, 5, 1], 9, 10, 11, 12;

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

Пирамидальная сортировка: пример

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, 1, 8, 7, 3, 6, 5, 4, 2], 10, 11, 12;

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

[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, 2, 4, 3, 6, 5, 1], 9, 10, 11, 12;

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

Пирамидальная сортировка: пример

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, 1, 6, 4, 3, 2, 5], 8, 9, 10, 11, 12;

[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;

Пирамидальная сортировка: пример

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;

Пирамидальная сортировка: пример

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.

Пирамидальная сортировка

Эффективность алгоритма

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 ***O(N*log N)***.

Сортировка слиянием

Идея метода

1. Делим массив элементов на две части:
   - если сортируемый массив состоит из n элементов,
     - ✔ то первая часть может состоять из ⌊n/2⌋ элементов,
     - ✔ а вторая – из оставшихся;
     - ✔ порядок следования элементов в каждой из полученных частей совпадает с их порядком следования в исходном массиве.
2. Сортируем отдельно каждую из частей.
3. Производим слияние отсортированных частей массива:
   - при слиянии сравниваем наименьшие элементы каждой из отсортированных частей и меньший из них отправляем в результирующий массив;
   - повторяем описанные действия до тех пор, пока не исчерпается одна из частей;
   - все оставшиеся элементы другой части переписываем в результирующий массив.

Сортировка слиянием

```
Procedure SortMerge ( var A: Massiv; first, last: integer);
Var c: integer;
begin
  if first <> last then begin
    c:= (first + last) div 2;
    SortMerge(A, first, c );
    SortMerge(A, c+1, last );
    Merge(A, first, c, last );
  end;
end.
```

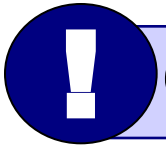
В программе используется процедура **`Merge(A, first, c, last )`**, которая выполняет слияние двух отсортированных частей массива **`A`** таким образом, чтобы сохранилась упорядоченность элементов в результирующем массиве.

Сортировка слиянием

```
Procedure Merge(var A: Massiv; first, c, last: integer);
Var B: Massiv;
    i, j, k: integer;
begin
  i:=first; l:=c+1;  k:=1;
  while (i<=c) and (j<=last) do
   if A[i] < A[j] then begin
      B[k]:=A[i];
      k:=k+1;   i:=i+1
   end
   else begin
      B[k]:=A[j];
      k:=k+1;   j:=j+1
   end
```

Сортировка слиянием

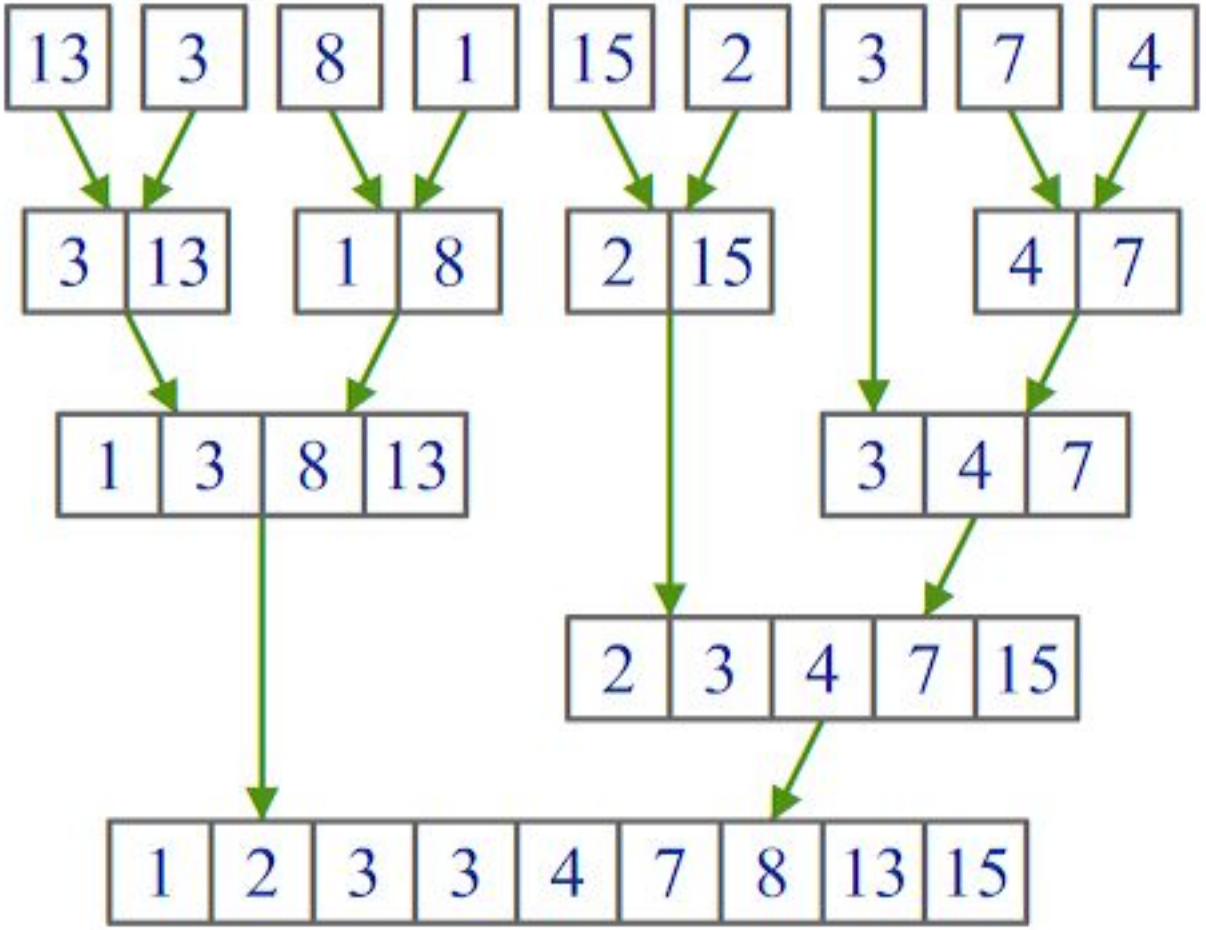
```
  while (i<=c) do begin
   B[k]:=A[i];
   k:=k+1;   i:=i+1
  end;
  while (j<=last) do begin
   B[k]:=A[j];
   k:=k+1;   j:=j+1
  end;
  k:=0;
  for i:=first to last do begin
   k:=k+1;
   A[i]:=B[k]
  end;
end;
```



Сложность (в среднем) $O(N \log N)$!

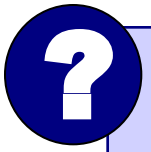
Сортировка слиянием

Пример



«Быстрая сортировка» *(Quick Sort)*

Идея – более эффективно переставлять элементы, расположенные дальше друг от друга.

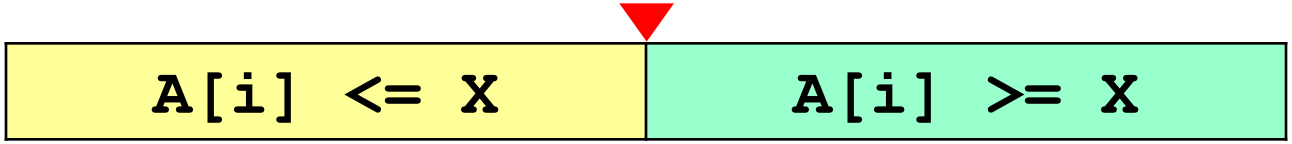


Сколько перестановок нужно, если массив отсортирован по убыванию, а надо – по возрастанию?

`N div 2`

1 шаг: выбрать некоторый элемент массива X

2 шаг: переставить элементы так:



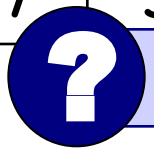
при сортировке элементы не покидают « свою область»!

3 шаг: так же отсортировать две получившиеся области

Разделяй и властвуй (англ. *divide and conquer*)

«Быстрая сортировка» *(Quick Sort)*

78	6	82	67	55	44	34



Как лучше выбрать X?

Медиана – такое значение X, что слева и справа от него в отсортированном массиве стоит одинаковое число элементов (*для этого надо отсортировать массив…*).

Разделение:

1)выбрать средний элемент массива (**X=67**)

78	6	82	67	55	44	34

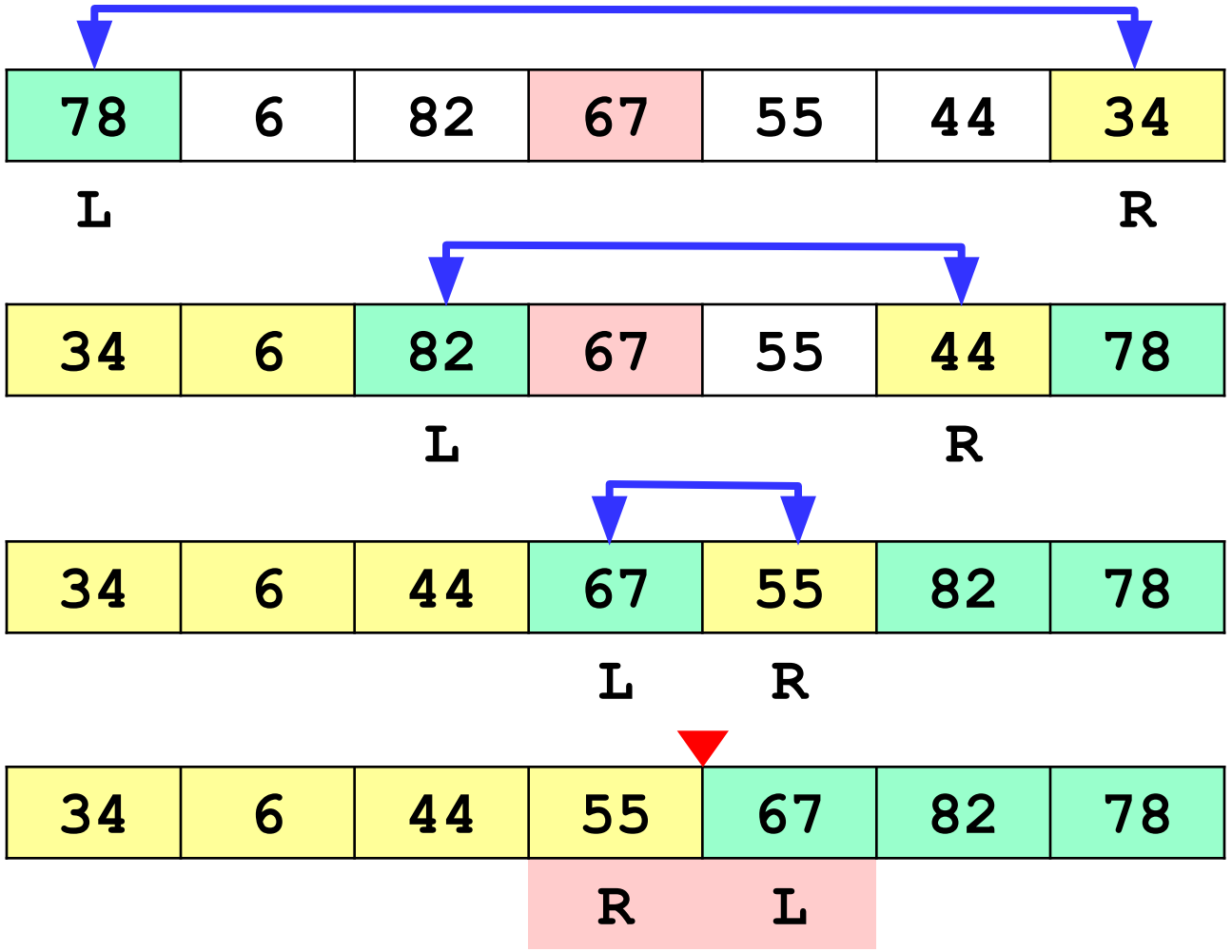
1)установить **L:=1**, **R:=N**

2)увеличивая **L**, найти первый элемент **A[L]**, который **>= X** (должен стоять справа)

3)уменьшая **R**, найти первый элемент **A[R]**, который **<= X** (должен стоять слева)

4)если **L<=R**, поменять местами **A[L]** и **A[R]** и перейти к п. 3

«Быстрая сортировка» *(Quick Sort)*



L > R : разделение закончено

«Быстрая сортировка» *(Quick Sort)*

```
procedure QuickSort ( var A: Massiv; first, last: integer);
var L, R, c, X: integer;
begin
  if first < last then begin
    X:=A[(first+last) div 2];
    L:=first; R:=last;
    while L <= R do begin
      while A[L]<X do L:=L+1;
      while A[R]>X do R:=R-1;
      if L<=R then begin
        c:=A[L]; A[L]:=A[R]; A[R]:=c;
        L:=L+1; R:=R-1;
      end;
    end;
    QuickSort(A, first, R);   QuickSort(A, L, last);
  end;
end.
```

ограничение рекурсии

разделение

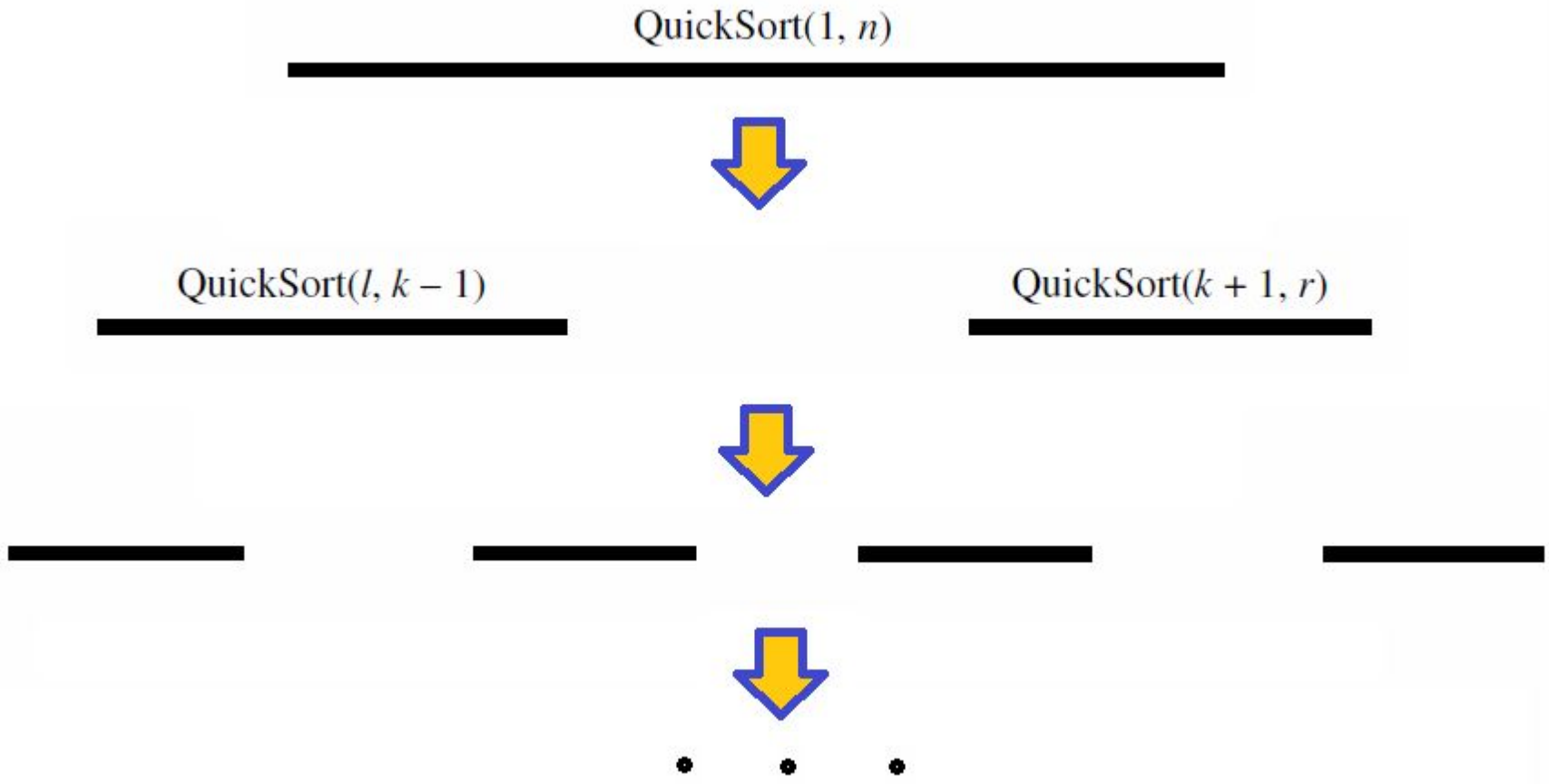
обмен

двигаемся дальше

сортируем две части

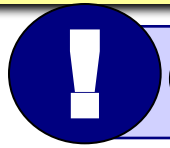
«Быстрая сортировка» *(Quick Sort)*

Схема выполнения алгоритма «быстрой сортировки»:



«Быстрая сортировка» *(Quick Sort)*

```
program qq;
const N = 10;
Type massiv=array[1..N] of integer;
var A:massiv;
procedure QuickSort ( var A: massiv:
   first, last: integer);
...
begin
   { заполнить массив }
   { вывести исходный массив на экран }
   Quicksort (A, 1, N ); { сортировка }
   { вывести результат }
end.
```



Сложность (в среднем) $O(N \log N)$!

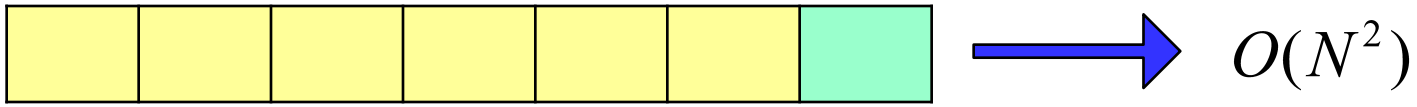
Количество перестановок

(случайные данные)

N	*QuickSort* $O(N \log N)$	*«пузырек»* $O(N^2)$
10	11	24
100	184	2263
200	426	9055
500	1346	63529
1000	3074	248547

? От чего зависит скорость?

? Как хуже всего выбирать X?



Вычислительная сложность алгоритмов сортировки