# Методы сортировки

- **Сортировка - это процесс упорядочения некоторого множества элементов, на котором определены отношения порядка $>, <, \geq, \leq$.**

# Методы сортировки

- **Имеется последовательность однотипных записей, одно из полей которых выбрано в качестве ключевого (ключ сортировки).**

**Тип данных ключа должен включать операции сравнения ("=", ">", "<", ">=" и "<=").**

**Задачей сортировки является**

**преобразование исходной последовательности**

**в последовательность, содержащую те же записи, но в порядке возрастания (или убывания) значений ключа.**

# Методы сортировки

- ***внутренняя* сортировка, в которой предполагается, что данные находятся в оперативной памяти, и важно оптимизировать число действий программы (для методов, основанных на сравнении, число сравнений, обменов элементов и пр.)**
- ***внешняя*, в которой данные хранятся на внешнем устройстве с медленным доступом (магнитные лента, барабан, диск) и прежде всего надо снизить число обращений к этому устройству.**

# Методы сортировки

# Сортировка подсчетом

**В сортировке подсчетом (counting sort) предполагается, что все n входных элементов — целые числа, принадлежащие интервалу от 0 до k, где k - некоторая целая константа.**

# Сортировка подсчетом

**Основная идея сортировки подсчетом заключается в том, чтобы для каждого рассматриваемого элемента x определить количество элементов, которые меньше x.**

**С помощью этой информации элемент x можно разместить на той позиции выходного массива, где он должен находиться.**

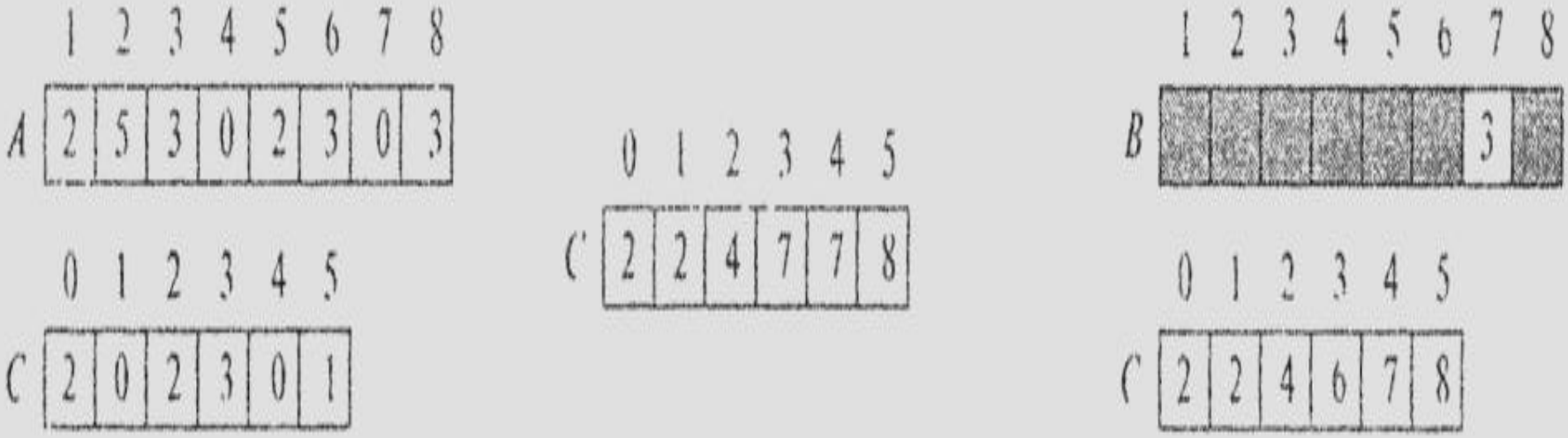
# Сортировка подсчетом

- **Исходный массив: A[1..n]**
- **Результат: B[1..n]**
- **Вспомогательный массив C[0..k]**

# Сортировка подсчетом

```
COUNTING_SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ В C[i] хранится количество элементов, равных i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ В C[i] — количество элементов, не превышающих i.
 9  for j ← length[A] downto 1
10       do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]] − 1
```

# Сортировка подсчетом



# Сортировка подсчетом

**Временная сложность- $O(n+k)$**

# Сортировка включением (вставками)

**Массив делится на 2 части:**

**Отсортированную и неотсортированную.**

**На каждом шаге берется очередной элемент из неотсортированной части и включается в отсортированную**

# Сортировка включением

- **Простое включение**

**Отсортировано начало массива**

**$A_1, A_2, \ldots, A_{i-1}$**

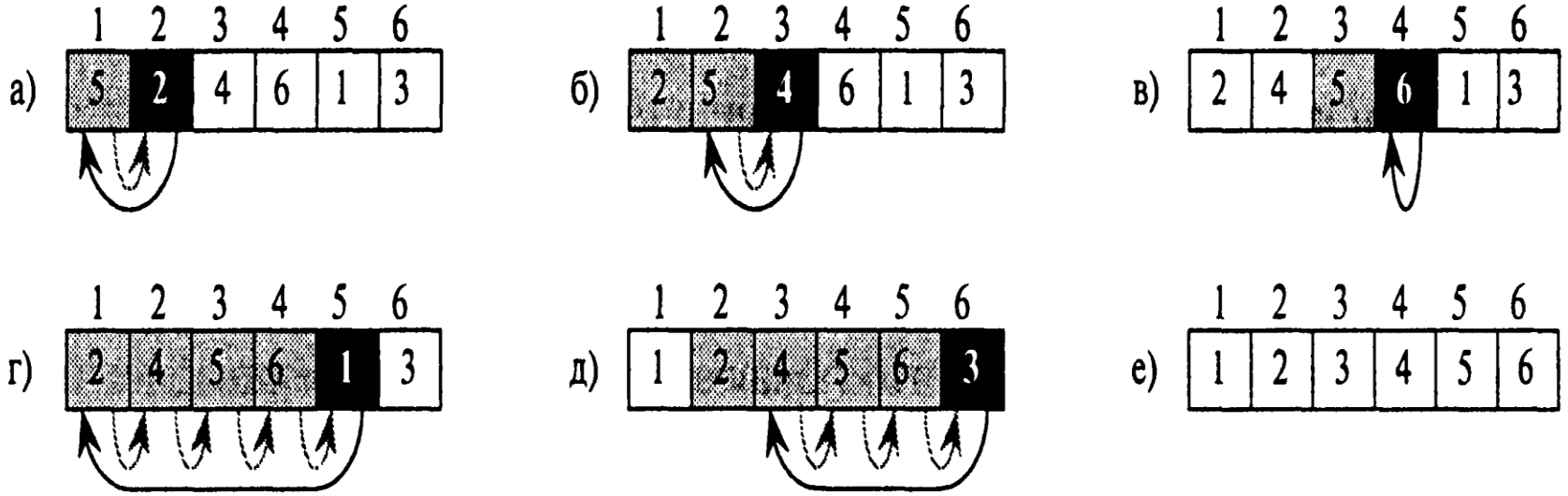
**Остаток массива $A_i, \ldots A_n$ – неотсортирован.**

**На очередном шаге Ai включается в отсортированную часть на соответствующее место**

# Сортировка включением

```
INSERTION_SORT(A)
 1  for j ← 2 to length[A]
 2       do key ← A[j]
 3          ▷ Вставка элемента A[j] в отсортированную
            ▷ последовательность A[1..j − 1]
 4          i ← j − 1
 5          while i > 0 и A[i] > key
 6               do A[i + 1] ← A[i]
 7                  i ← i − 1
 8          A[i + 1] ← key
```

# Сортировка включением



# Сортировка включением

**Алгоритм имеет сложность**

$$O(n^2),$$

**но в случае исходно отсортированного массива внутренний цикл не будет выполняться ни разу, поэтому метод имеет**

$$O(n).$$

# 2. Сортировка включением

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 27 | 51 | 14 | 31 | 42 | 1 | 8 | 24 | 3 | 59 | 33 | 44 | 53 | 16 | 10 | 38 | 50 | 21 | 36 |
| 4 | 14 | 27 | 51 | 31 | 42 | 1 | 8 | 24 | 3 | 59 | 33 | 44 | 53 | 16 | 10 | 38 | 50 | 21 | 36 |
| 4 | 14 | 27 | 31 | 51 | 42 | 1 | 8 | 24 | 3 | 59 | 33 | 44 | 53 | 16 | 10 | 38 | 50 | 21 | 36 |
| 4 | 14 | 27 | 31 | 42 | 51 | 1 | 8 | 24 | 3 | 59 | 33 | 44 | 53 | 16 | 10 | 38 | 50 | 21 | 36 |

```
public static void simpleSort(int[] data) {
    int i, j;
    for (i = 1; i < data.length; i++) {
        int c = data[i];
        for (j = i-1; j >= 0 && data[j] > c; j--) {
            data[j+1] = data[j];
        }
        data[j+1] = c;
    }
}
```

# Сортировка включением

- БИНАРНЫЕ ВСТАВКИ (ЧИСЛО СРАВНЕНИЙ nlgn)

ПРОЦЕСС СОРТИРОВКИ МЕТОДОМ ДВУХПУТЕВЫХ ВСТАВОК

```
                503
                ^
          087   503
                ^
          087   503   512
          ^
    061   087   503   512
                      ^
    061   087   503   512   908
          ^
    061   087   170   503   512   908
                      ^
    061   087   170   503   512   897   908
          ^
061 087   170   275   503   512   897   908
```

# Сортировка Шелла (Donald Shell, 1959г.)

СОРТИРОВКА ШЕЛЛА СО СМЕЩЕНИЯМИ 8, 4, 2, 1

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Сортировка через 8:

503 087 154 061 612 170 765 275 653 426 512 509 908 677 897 703

Сортировка через 4:

503 087 154 061 612 170 512 275 653 426 765 509 908 677 897 703

Сортировка через 2:

154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703

Сортировка через 1:

061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

# Сортировка Шелла (Donald Shell, 1959г.)

**Вместо включения A[i] в подмассив предшествующих ему элементов, его включают в подсписок, содержащий элементы A[i-h], A[i-2h], A[i-3h] и тд,**

**где h - положительная константа. Таким образом формируется массив, в котором «h-серии» элементов, отстоящих друг от друга на h, сортируются отдельно.**

**Процесс возобновляется с новым значением h, меньшим предыдущего. И так до тех пор, пока не будет достигнуто значение h=1.**

# Сортировка Шелла

- **Для достаточно больших массивов рекомендуемой считается такая последовательность, что**

$$h_{i+1}=3h_i+1, \text{ а } h_1=1.$$

**Начинается процесс с $h_{m-2}$, $h_{m-2}$ первый такой член последовательности, что**

$$h_{m-2}\geq [n/9].$$

- **1, 4, 13, 40, 121…($h_{i+1}=3h_i+1$)**
- **1, 3, 7, 15, 31 ($h_{i+1}=2h_i+1$)**

```
h ←1;
 while h < N div 9 do h ← h*3 + 1;
 repeat     // цикл по сериям
   for k ← 1...   h do {сортировка k-ой серии}
        i ← h + k;
       while i <= N do //вкл. A[i]  на свое место в серии
         x ← A[i]; j ← i - h; //}
         while (j >= 1) and (A[j]>x) do //сдвиг
              A[j + h] ← A[j];  j ← j-h);
         A[j + h] ← x;   i ←i + h);
   h ← h div 3;{переход к нов.серии}
 while h>0;
```

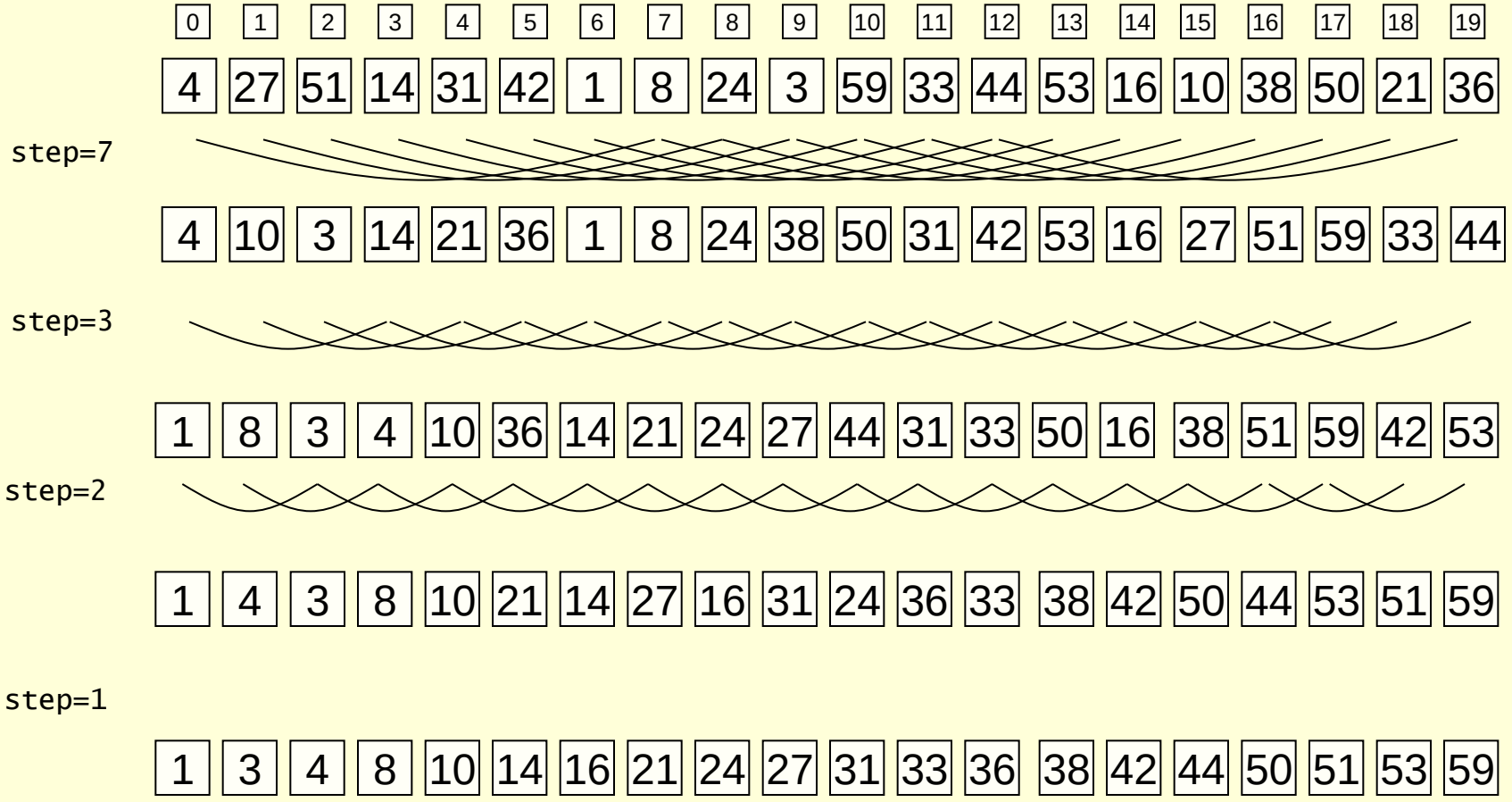
# Сортировка Шелла

- Сэджвик (1986г)

$$h_s = \begin{cases} 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1, & \text{если } s \text{ четно;} \\ 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1, & \text{если } s \text{ нечетно.} \end{cases}$$

- $(h_0,h_1,h_2,\ldots)=(1,5,19,41,109,209..)$
- Конец последовательности – $h_{t-1}$ если $3h_t \geq n$
- среднее количество операций: $O(n^{7/6})$,
- в худшем случае - порядка $O(n^{4/3})$.
- в среднем $1{,}66n^{1{,}25}$ перемещений.

# 4. Сортировка Шелла



# 4. Сортировка Шелла

```
public static void ShellSort(int[] data) {
    int n = data.length;    // Длина массива
    int step = n;           // Шаг поисков и вставки
    int i, j;
    do {
        // Вычисляем новый шаг
        step = step / 3 + 1;
        // Производим сортировку простыми вставками с заданным шагом
        for (i = step; i < n; i++) {
            int c = data[i];
            for (j = i-step; j >= 0 && data[j] > c; j -= step) {
                data[j+step] = data[j];
            }
            data[j+step] = c;
        }
    } while (step != 1);
}
```

http://www.youtube.com/watch?feature=player_detailpage&v=lvts84Qfo8o

Количество перестановок элементов
(по результатам экспериментов со случайным массивом)

| | n = 25 | n = 1000 | n = 100000 |
|---|---|---|---|
| Сортировка Шелла | 50 | 7700 | 2 100 000 |
| Сортировка простыми вставками | 150 | 240 000 | 2.5 млрд. |

# Сортировка извлечением (выбором)

**Вариант 1**

**массив делится на уже отсортированную часть**

$$A_{i+1}, A_{i+2} \ldots A_n$$

**и неотсортированную**

$$A_1, A_2, \ldots, A_i$$

1. **На каждом шаге извлекается мах из неотсортированной части**
2. **Найденный мах ставится в начало отсортированной части**

# Сортировка извлечением (выбором)

- **Простое извлечение**

```
for i ←N downto 2 do
  MaxIndex ← 1;
     for j ← 2 to i do
        if A[j]>A[MaxIndex] then MaxIndex ← j;
  Tmp ← A[i];
   A[i] ← A[MaxIndex];
   A[MaxIndex] ← Tmp;
```

# Сортировка извлечением (выбором)

СОРТИРОВКА ПОСРЕДСТВОМ ПРОСТОГО ВЫБОРА

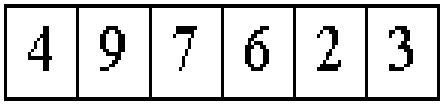
| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 503 | 087 | 512 | 061 | **908** | 170 | **897** | 275 | 653 | 426 | 154 | 509 | 612 | 677 | **765** | **703** |
| 503 | 087 | 512 | 061 | 703 | 170 | **897** | 275 | 653 | 426 | 154 | 509 | 612 | 677 | **765** | 908 |
| 503 | 087 | 512 | 061 | 703 | 170 | **765** | 275 | 653 | 426 | 154 | 509 | 612 | **677** | 897 | 908 |
| 503 | 087 | 512 | 061 | **703** | 170 | **677** | 275 | **653** | 426 | 154 | 509 | **612** | 765 | 897 | 908 |
| 503 | 087 | 512 | 061 | 612 | 170 | **677** | 275 | **653** | 426 | 154 | **509** | 703 | 765 | 897 | 908 |
| 503 | 087 | 512 | 061 | 612 | 170 | 509 | 275 | **653** | **426** | **154** | 677 | 703 | 765 | 897 | 908 |
| ... | | | | | | | | | | | | | | | |
| 061 | 087 | 154 | 170 | 275 | 426 | 503 | 509 | 512 | 612 | 653 | 677 | 703 | 765 | 897 | 908 |

# Сортировка извлечением (выбором)

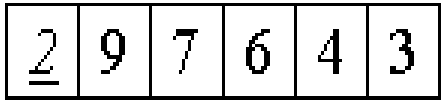
- Вариант 2
- последовательность создается, начиная с левого конца массива. Алгоритм состоит из **n** последовательных шагов, начиная с первого и заканчивая **(n-1)**-м.
- На **i**-м шаге выбираем наименьший из элементов **A[i] ... A[n]** и меняем его местами с **A[i].**

# Сортировка извлечением (выбором)

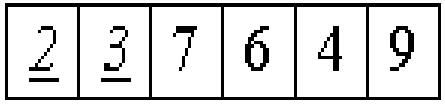
- 



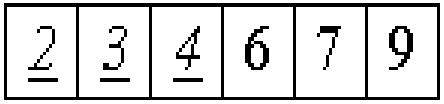

исходная последовательность



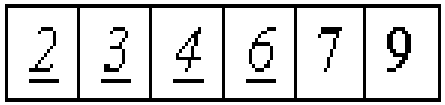

шаг 0: 2 ⟷ 4



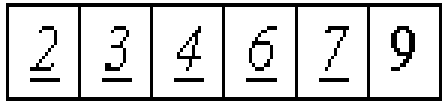

шаг 1: 3 ⟷ 9




шаг 2: 4 ⟷ 7




шаг 3: 6 ⟷ 6




шаг 4: 7 ⟷ 7

# Сортировка извлечением (выбором)

**С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:**

$$n + (n-1) + (n-2) + (n-3) + \ldots + 1 = 1/2 * ( n^2-n ) = O(n^2).$$

# Сортировка извлечением(выбором)

**Усовершенствование простого выбора:**

**Лемма . В любом алгоритме нахождения максимума среди n элементов, основанном на сравнении пар элементов, необходимо выполнить, по крайней мере, n — 1 сравнений.**

# Сортировка извлечением (выбором)

503 087 512 061 | 908 170 897 275 | 653 426 154 509 | 612 677 765 703

512, 908, 653, 765.

512, 897, 653, 765

512, 275, 653, 765

# Сортировка извлечением(выбором)

***В общем случае, если n — точный квадрат, можно разделить массив на $\sqrt{n}$ групп по $\sqrt{n}$ элементов***

***Любой выбор, кроме первого, требует не более чем $\sqrt{n}$ -2 сравнений внутри группы ранее выбранного элемента плюс $\sqrt{n}$ -1 сравнений среди "лидеров групп"***

**Этот метод получил название квадратичный выбор**

**общее время его работы составляет порядка $O(n\sqrt{n})$ что существенно лучше, чем $O(n^2)$**

# Heapsort (Williams,Floyd)

- В алгоритме выбора проделав n-1 сравнение, мы можем построить дерево выбора и идентифицировать его корень как мин элемент
- Второй этап сортировки – спуск вдоль пути, отмеченного наименьшим элементом, и исключение его из дерева
- Элемент продвинувшийся в корень опять будет мин
- После n шагов дерево становится пустым и сортировка заканчивается

## Heapsort (Williams,Floyd)

tree1

- **На каждом из n шагов выбора требуется только log n сравнений. Поэтому на весь процесс понадобится порядка**

**n*log(n) , т.е. T(n) = O(n*log n )**

**элементарных операций и n шагов на построение дерева**

## Heapsort (Williams,Floyd)

- **Пирамида (двоичное дерево) определяется как последовательность**

$H_L, H_{L+1}, \ldots H_R$ **такая, что**

$H_i \le H_{2i}$ **&** $H_i \le H_{2i+1}$ **для** $i=L \ldots R/2$

$H1 = \min(H_1, H_2, \ldots H_N)$

tree2

## Heapsort (Williams,Floyd)

**Алгоритм Флойда:**

**Построение пирамиды на «том же месте»:**

**пусть $H_1, H_2,\ldots H_N$ есть сортируемый массив, причем $H_M,\ldots H_N$, где**

**$M= (N \text{ div } 2) +1$ образует нижний слой пирамиды, поскольку нет индексов**

**i, j таких что $j=2i$ или $j=2i+1$, т.е. для этого слоя упорядоченности не требуется**

## Heapsort (Williams,Floyd)

**Алгоритм Флойда:**

**Пирамида расширяется влево -**

**каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент, пока элементы стоящие слева от $H_M$**

**не будут образовывать пирамиду**

(table1)

**Такая процедура называется – Sift**

## Heapsort (Williams,Floyd)

**Sift(L, R: index);**

// i, j: index; x: item; x – элемент вставляемый в пирамиду

// i, j - пара индексов, фиксирующих элементы, меняющиеся на каждом шаге местами.

```
i ← L; j ← 2*L;  X ← a[L];
    if (j < R) & (a[j+1] < a[j])  then j ← j+1
    while (j < = R) & (a[j] < X) do
        a[i] ← a[j]; a[j] ← X; i ← j; j ← 2*j;
        if (j < R) & (a[j+1] < a[j]  then j ← j+1
// end Sift;
```

## Heapsort (Williams,Floyd)

Процесс формирования пирамиды из **n** элементов $\mathbf{h_1 \ldots h_n}$. на том же самом месте описывается так:

```
L ← (n div 2)+ 1;
while L > 1 do
    L ← L - 1;
    sift(L, n)
```

table1

## Heapsort (Williams,Floyd)

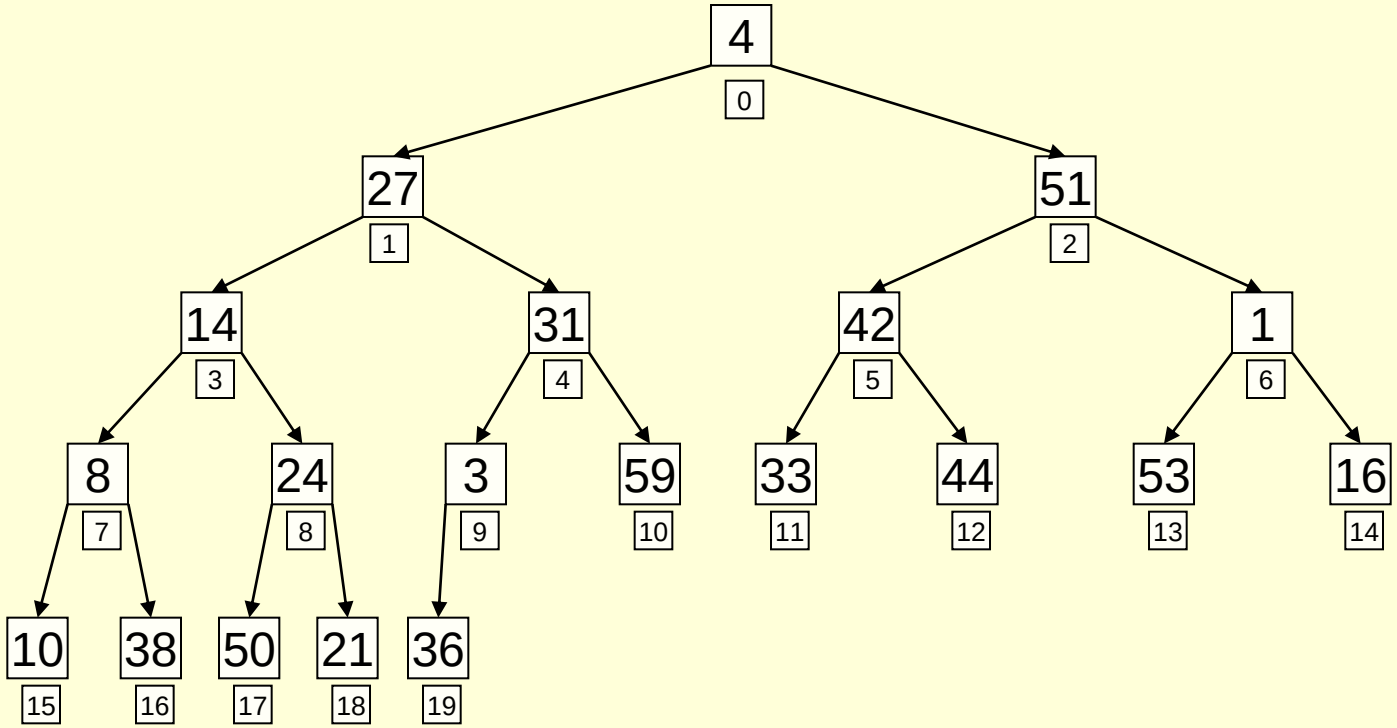
n сдвигающих шагов выталкивания наименьших элементов на вершину дерева с последующим смещением в конец массива:

```
R ← n;
while R>1 do
    x ← a[1]; a[1] ← a[R]; a[R] ← x;
    R ← R-1; sift(1,R )
```

table2

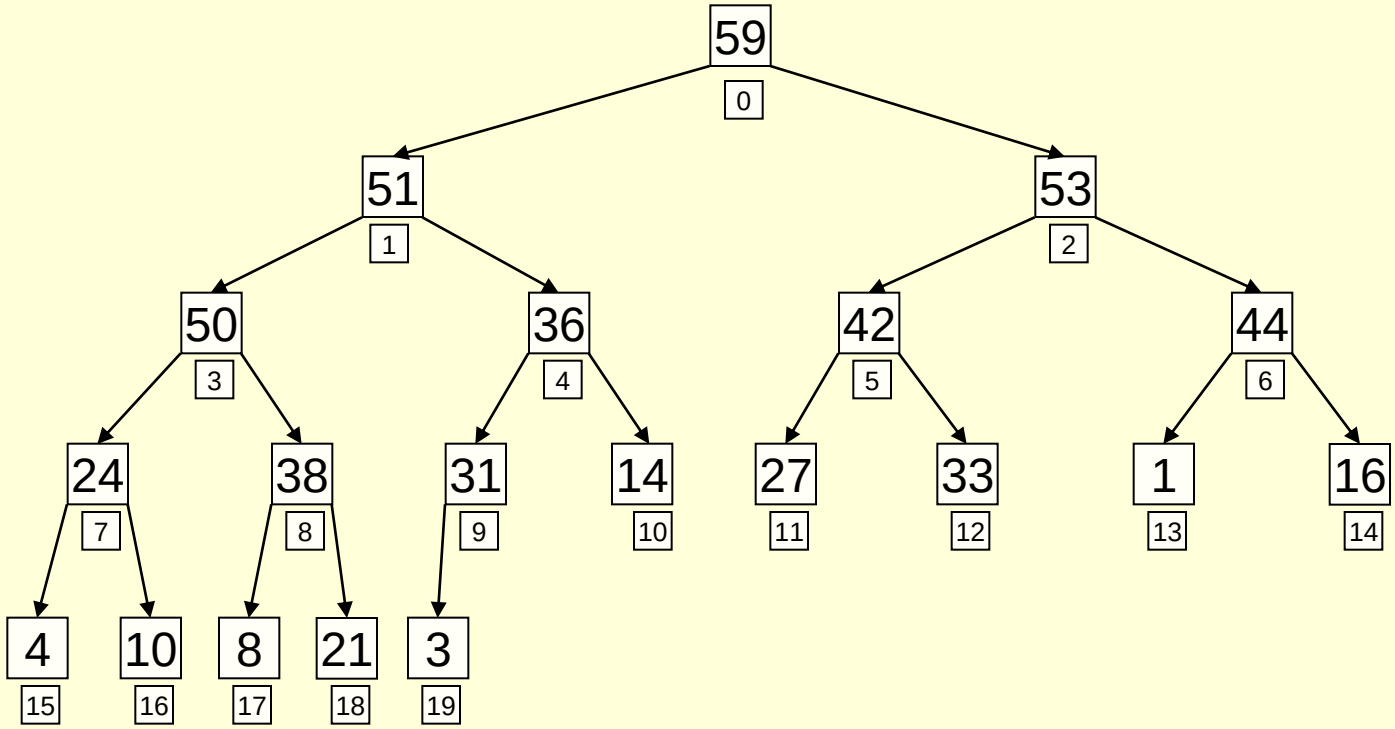
# Пирамидальная сортировка (HeapSort)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 27 | 51 | 14 | 31 | 42 | 1 | 8 | 24 | 3 | 59 | 33 | 44 | 53 | 16 | 10 | 38 | 50 | 21 | 36 |




И так далее…

# Пирамидальная сортировка (продолжение)




И так далее…

# Алгоритм пирамидальной сортировки

```
public static void heapSort(int[] data) {
    int n = data.length;   // Длина массива

    buildPyramid:          // Построение пирамиды
    for (int i = 1; i < n; i++) {
        // Inv: Элементы data[0:i-1] уже образуют пирамиду;
        int c = data[i];   // "Протаскиваемое" значение
        int p = i, q;      // Индексы для "протаскивания" вверх к вершине
        do { q = p;
             if ((p = (q-1) >> 1) >= 0 && data[p] < c) data[q] = data[p];
             else {
                 data[q] = c;
                 continue buildPyramid;
             }
        } while (true);
    }

    meltPyramid:           // Постепенное разрушение пирамиды
    for (int i = n-1; i > 0; i--) {
        int c = data[i];
        data[i] = data[0];
        int q, p = 0;      // Индексы для протаскивания
        do { q = p;
             p = (q << 1) | 1;
             if (p >= i) { // Вышли за границу пирамиды
                 data[q] = c;
                 continue meltPyramid;
             }
             if (p < i-1 && data[p+1] > data[p]) p++;
             if (data[p] > c) data[q] = data[p];
             else {
                 data[q] = c;
                 continue meltPyramid;
             }
        } while (true);
    }
}
```

## Heapsort (Williams,Floyd)

```
Procedure HeapSort;
 var L, R: index; x: item;
 Procedure sift(L, R: index);
  var i, j: index; x: item;
    begin i := L; j := 2*L;  X :=  a[L];
      if (j < R) & (a[j+1] < a[j]  then j := j+1
      while (j < = R) & (a[j] < X) do
         begin
           a[i]:=a[j]; a[j]:= X; i:=j; j:=2*j;
           if (j < R) & (a[j+1] < a[j]  then j := j+1
         end;
    end sift;
```

## Heapsort (Williams,Floyd)

```
begin
   L:=(n div 2)+ 1;R := n;
 while L > 1 do
   begin
     L :== L - 1;  sift(L, r)
   end;
  while R>1 do
     begin
         x:=a[1]; a[1] := a[R]; a[R] := x;
         R := R-1; sift(1,R )
      end;
end  HeapSort;
```

## Heapsort (Williams,Floyd)

**В самом плохом из возможных случаев Heapsort потребует n*log n шагов.**

**т.е. T(n) = O (n*log n)**

**Heapsort «любит» начальные последовательности, в которых элементы более или менее отсортированы, то фаза порождения пирамиды потребует мало перемещений.**

**Среднее число перемещений приблизительно равно n/2*log(n), причем отклонения от этого значения незначительны.**