Алгоритмы и структуры данных

ЛЕКЦИЯ 3

Сортировки

Валенда Н.А.

Кафедра Программной инженерии,
факультет КН, ХНУРЕ

Сортировки

- Вставками
- Метод пузырька
- Пирамидальная сортировка
- Быстрая сортировка

Сортировка

Сортировка - процесс перестановки объектов заданной совокупности в определенном порядке (возрастающем или убывающем).

Целью сортировки обычно является облегчение последующего поиска элементов в отсортированном множестве.

Задача сортировки

Дана последовательность из *n* ключей

$$a_1, a_2, \dots, a_n$$

Требуется упорядочить ключи по неубыванию или по невозрастанию – найти перестановку (i_1, i_2, …, i_n) ключей

По неубыванию (non-decreasing order)

$$a_{i_1} \le a_{i_2} \le \cdots \le a_{i_n}$$

По невозрастанию (non-increasing order)

$$a_{i_1} \ge a_{i_2} \ge \cdots \ge a_{i_n}$$

Устойчивость сортировки

При устойчивой сортировке относительный порядок элементов с одинаковыми ключами не меняется.

Устойчивая сортировка — сортировка, которая не меняет относительный порядок сортируемых элементов, имеющих одинаковые ключи.

Использование дополнительной памяти

Алгоритм сортировки не использующий дополнительной памяти (кроме сортируемого массива) называется алгоритмом **сортировки на месте** (in-place sort)

Сортировка вставками

Массив состоит из двух частей – упорядоченной и неупорядоченной. Алгоритм встраивает элементы неупорядоченной части в отсортированную часть массива.

На каждом шаге алгоритма выбираем один элемент из неупорядоченной части и вставляем его на нужную позицию в уже отсортированной части массива.

Повторяем до тех пор, пока весь набор входных данных не будет отсортирован. Элементы обрабатываются по порядку их появления во входном массиве.

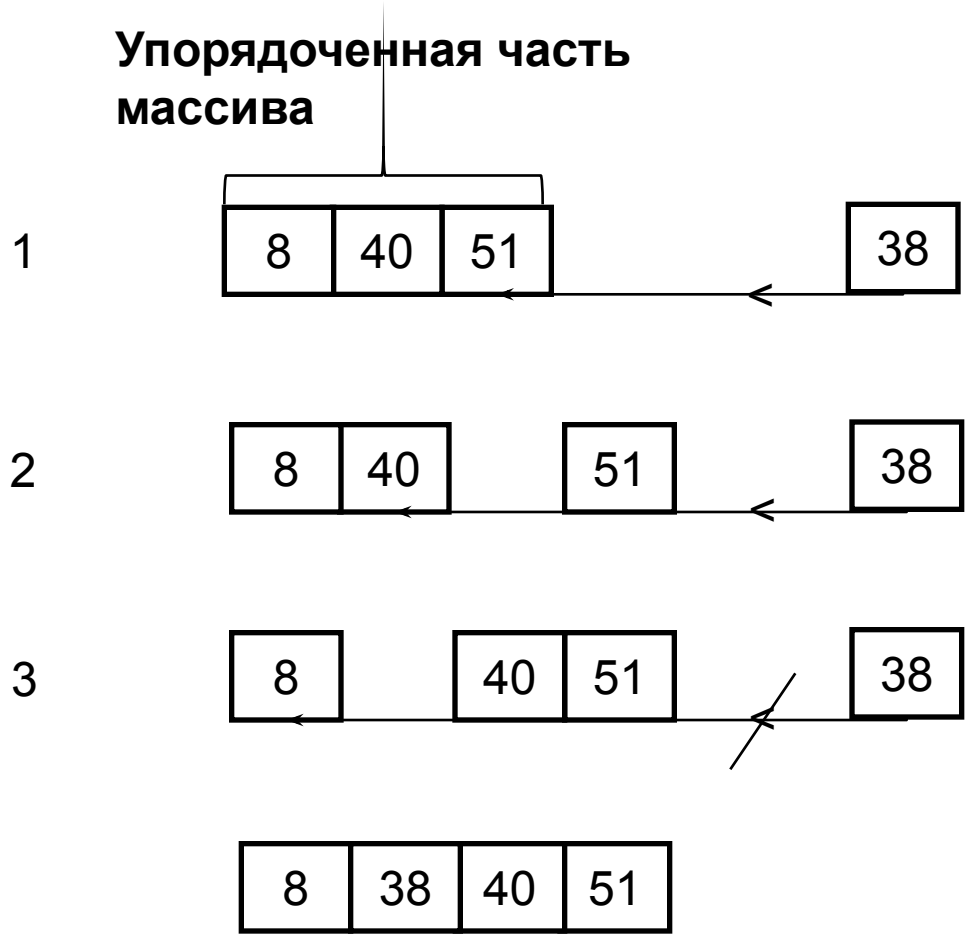
Сортировка вставками

Упорядоченная часть массива

Неупорядоченная часть массива

40	|	**51**	8	38	90	14	2	63
40	51	|	**8**	38	90	14	2	63
8	40	51	|	**38**	90	14	2	63
8	38	40	51	|	**90**	14	2	63
8	38	40	51	90	|	**14**	2	63
8	14	38	40	51	90	|	**2**	63
2	8	14	38	40	51	90	|	**63**
2	8	14	38	40	51	63	90	|

Вставка в упорядоченную часть массива



Сортировка вставками

```
public static void insert_sort(int[] a, int n)
        {
            int x;
            int i, j;
            for (i = 1; i < n; i++)
            {
                x = a[i];
                for (j = i - 1; (j >= 0)&&(x<a[j]); j--)
                    a[j + 1] = a[j];
                a[j + 1] = x;
             }
        }
```

Анализ сортировки вставками

Худший случай: упорядоченный в обратном порядке массив

$T(N) = 1 + 2 + 3 + \ldots + N - 1 = (N * (N - 1))/2 = O(N^2)$

Лучший случай:

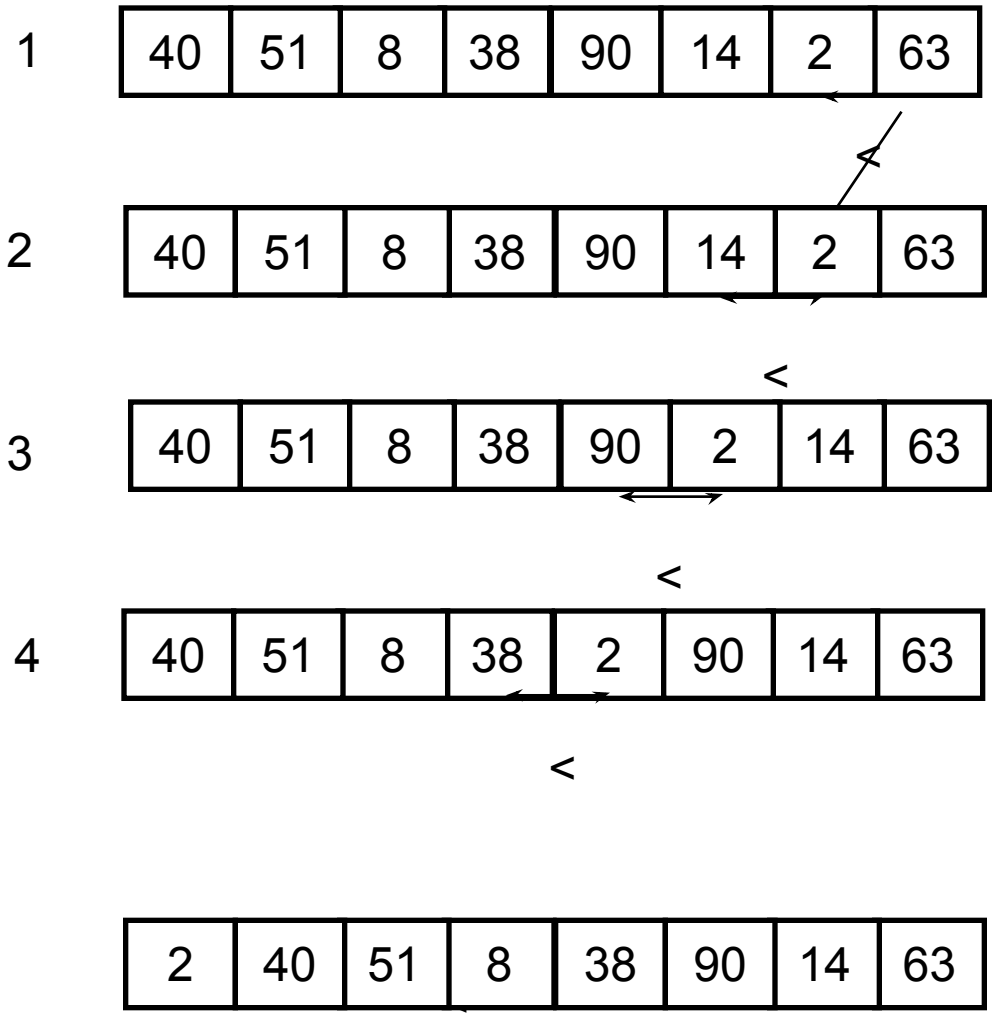
$T(N) = \theta(N)$

Общее время: $T(N) = \theta(N^2)$

Сортировка пузырьком

Алгоритм состоит в повторяющихся проходах по сортируемому массиву. На каждой итерации последовательно сравниваются соседние элементы, и, если порядок в паре неверный, то элементы меняют местами.
За каждый проход по массиву как минимум один элемент встает на свое место, поэтому необходимо совершить не более n-1 проходов, где n размер массива, чтобы отсортировать массив.

Сортировка пузырьком



Сортировка простым обменом. Метод пузырька

```
public static void bubble_sort(int[] a, int n)
{
    int temp;
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = n - 1; j > i; j--)
        {

            if (a[j] < a[j - 1])
            {
                temp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = temp;
            }
        }
    }
}
```

Анализ сортировки метод пузырька

Наихудший случай: упорядоченный в обратном порядке массив

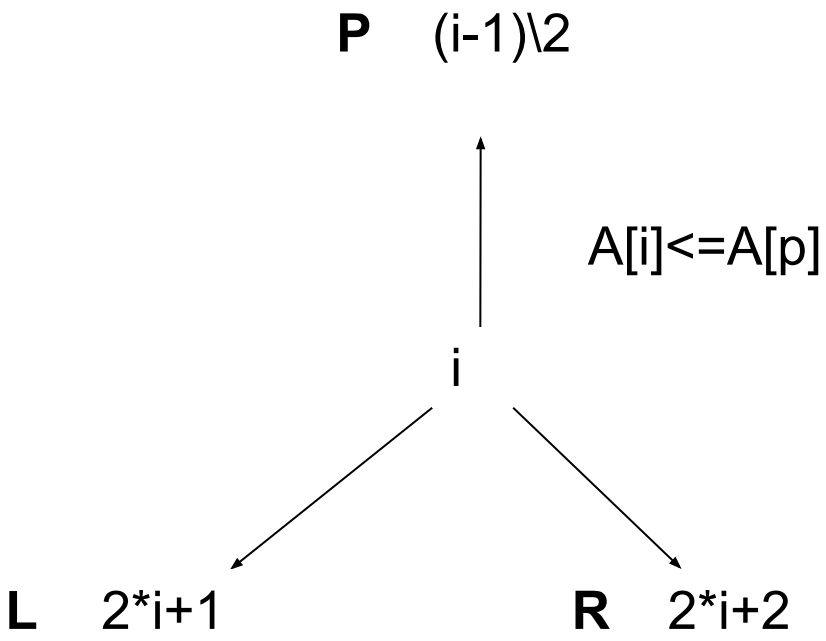
Количество сравнений:

$$T(N) = (N - 1) + (N - 2) + \ldots + 1 = (N * (N-1))/2 = O(N^2)$$

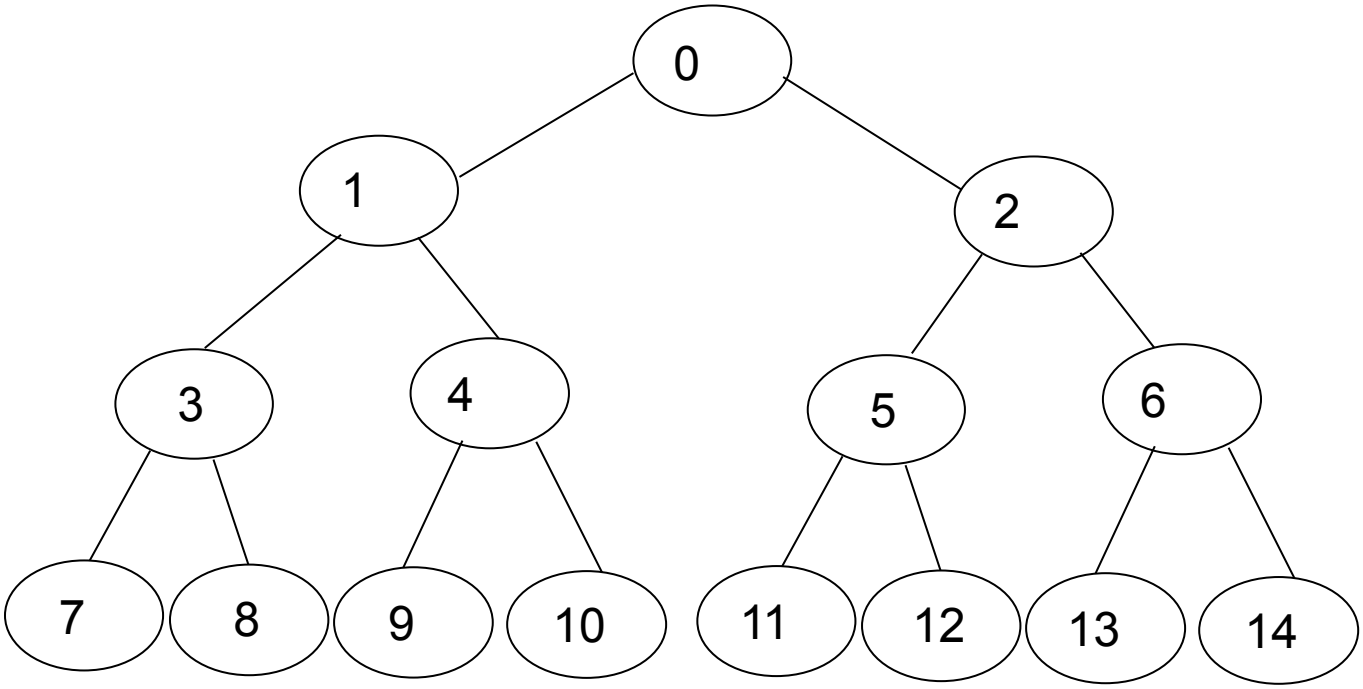
Общее время: $T(N) = \theta(N^2)$

Пирамидальная сортировка

Пирамида (binary heap) — это структура данных, представляющая собой объект-массив, который можно рассматривать как почти полное бинарное дерево . Каждый узел этого дерева соответствует определенному элементу массива и для него выполняется следующее свойство:

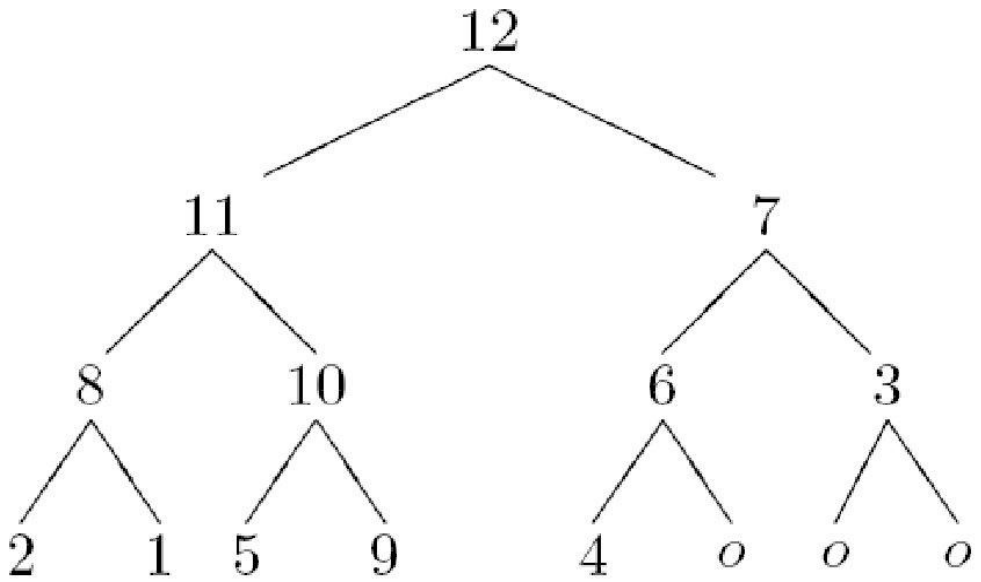


Распределение индексов массива в дереве

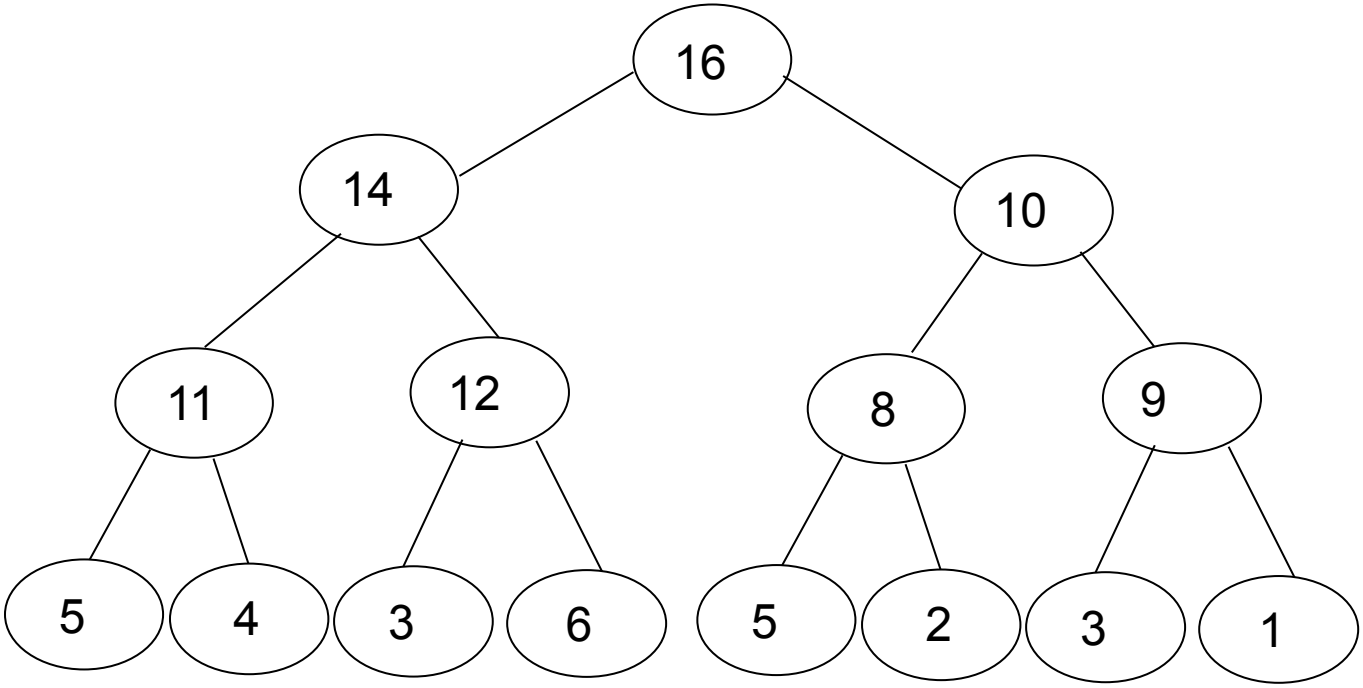


На всех уровнях, кроме, последнего, дерево полностью заполнено (заполненный уровень — это такой, который содержит максимально возможное количество узлов). Последний уровень заполняется слева направо до тех пор, пока в массиве не закончатся элементы.

Пирамида для 12-ти элементов



Пирамида (двоичная куча)



Пирамида (двоичная куча) в виде массива

16, 14, 10, 11, 12, 8, 9, 5, 4, 3, 6, 5, 2, 3, 1

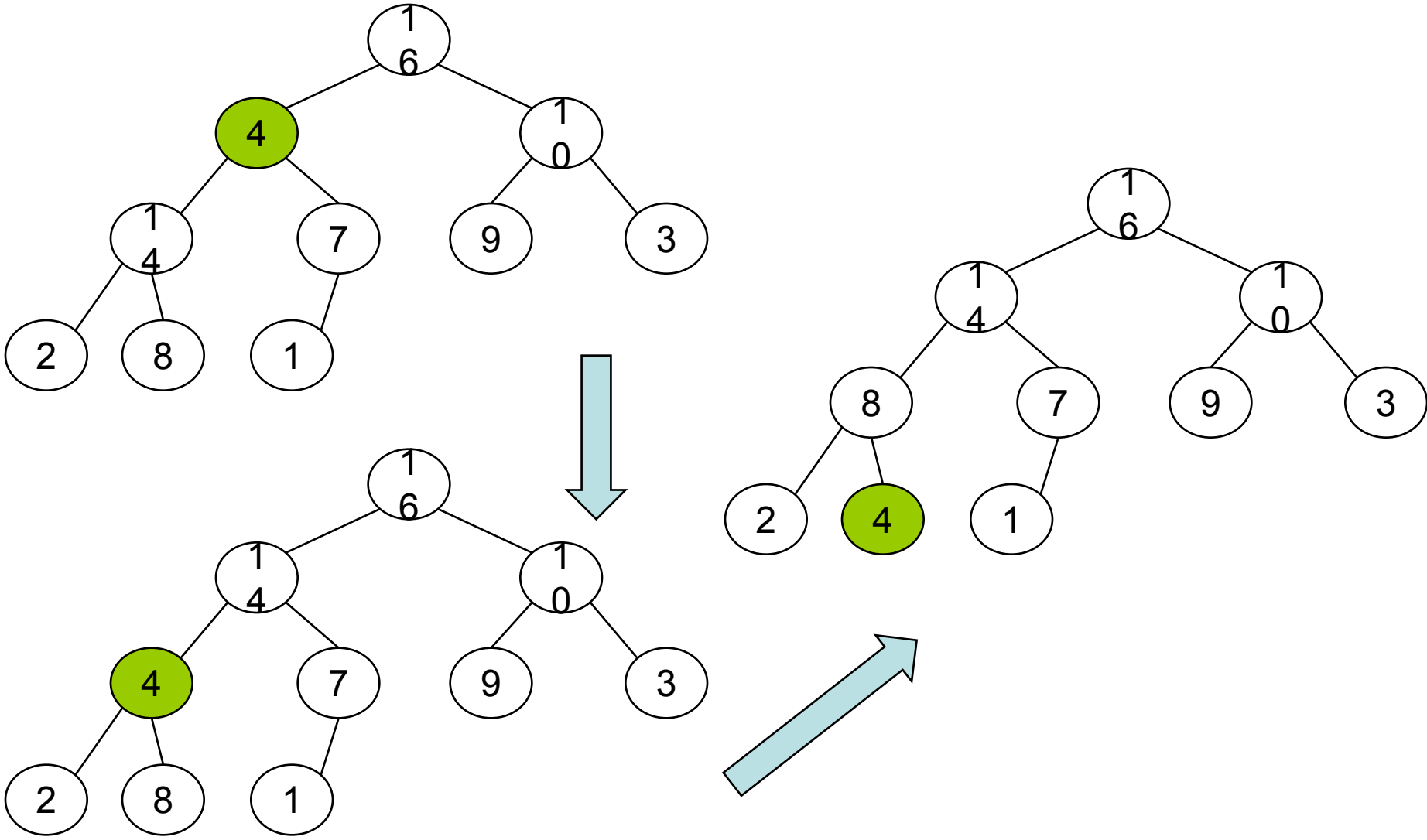
Алгоритм пирамидальной сортировки

1. Входную последовательность превращаем в пирамиду.

2. Голову пирамиды меняем местами с последним элементом и уменьшаем размер пирамиды на 1. Затем восстанавливаем пирамиду. И повторяем до тех пор пока в пирамиде не останется один элемент

Сохранение основного свойства

```
public static void Heap(int[] A, int i, int size)
{
int l,r,largest,k;
l=2*i+1;
r=2*i+2;
largest=(l<size&&A[l]>A[i])? l : i;
if (r<size&&A[r]>A[largest])  largest=r;
if (largest!=i) {
        k=A[i];
        A[i]=A[largest];
        A[largest]=k;
        Heap(A, largest, size);
        }
}
```

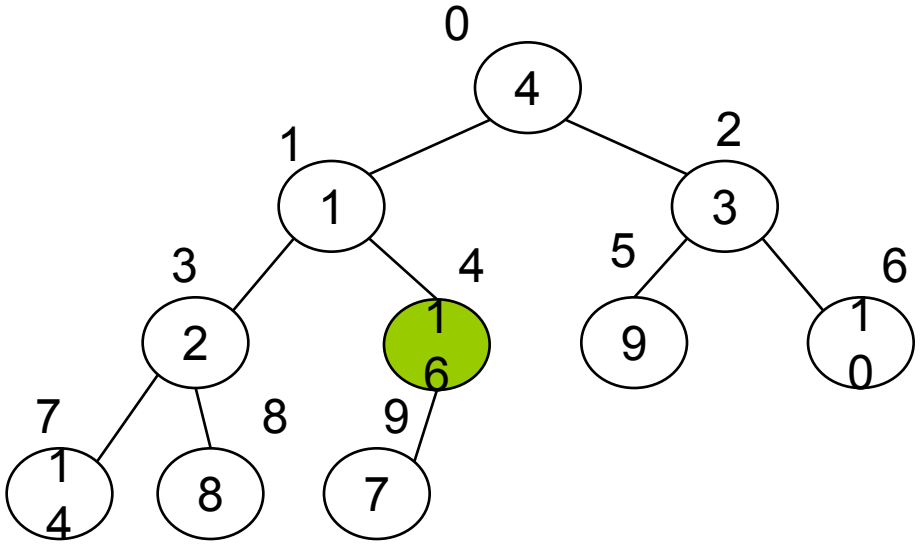

16
4
10
14
7
9
3
2
8
1
16
14
10
4
7
9
3
2
8
1
16
14
10
8
7
9
3
2
4
1

Алгоритм пирамидальной сортировки

```
public static void heap_sort (int[] A , int N)
{   int I, t;
    for (i = N/2-1; i >= 0; i--) // построение пирамиды
     Heap (A, i, N);
    for(i = N-1; i > 0; i--) { // сортировка
     t = A[0] ;
     A[0] = A[i] ;
     A[i] = t;
     Heap(A, 0, i);
    }
}
```

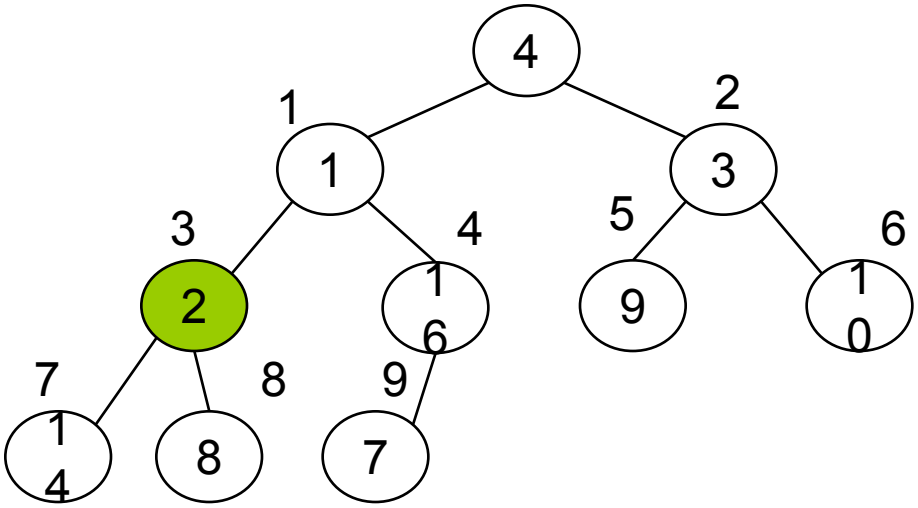
Построение пирамиды из массива

1



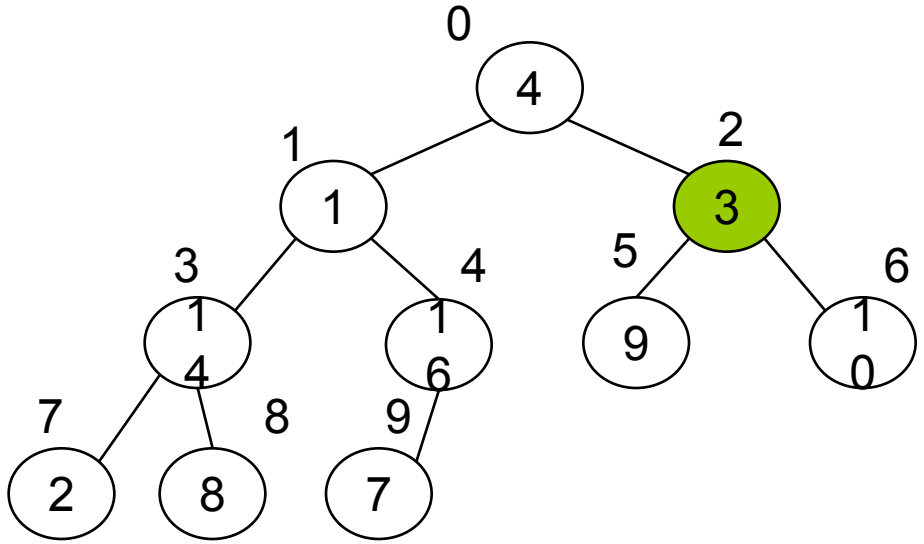
4, 1, 3, 2, 16, 9,10, 14, 8,7

2

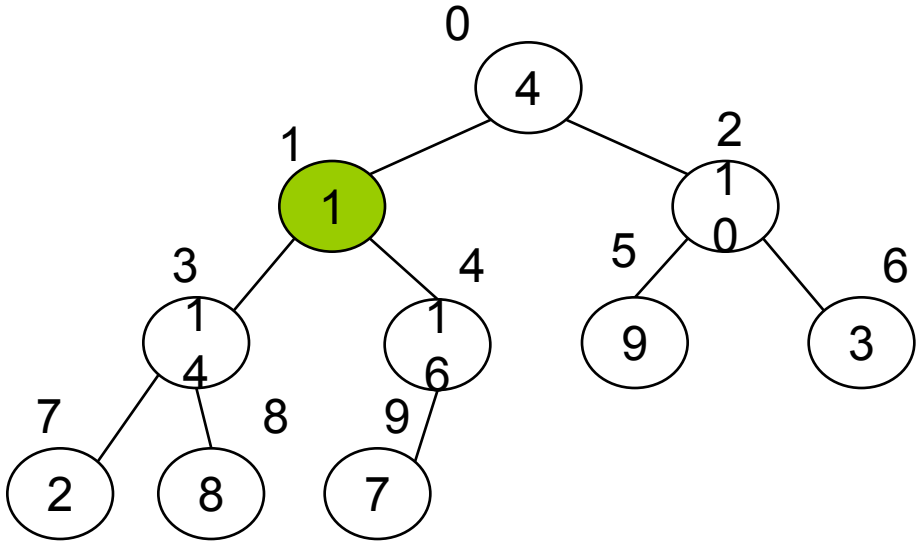


Построение пирамиды из массива

3

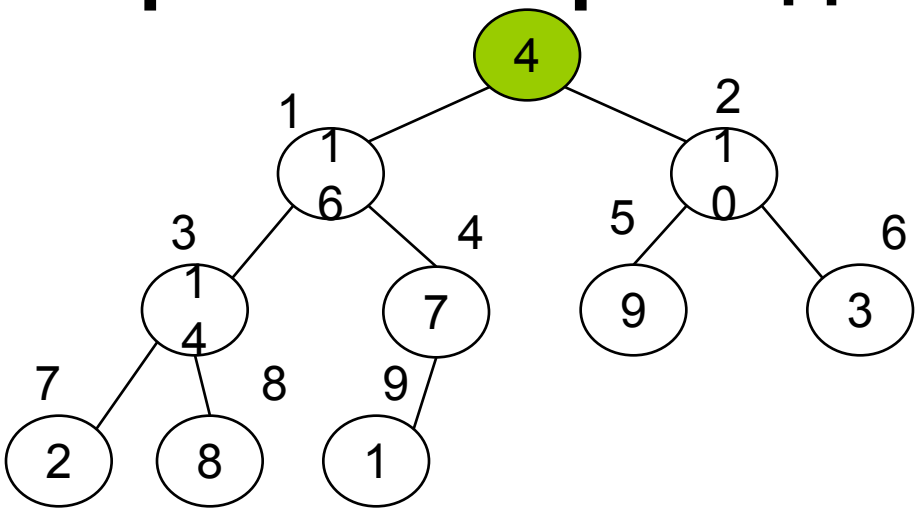


4

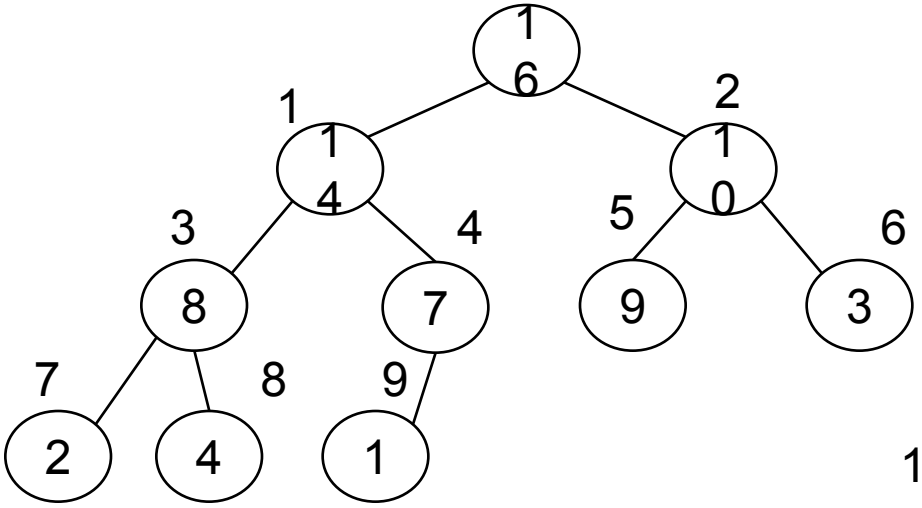


Построение пирамиды из массива

5



6



16, 14, 10, 8, 7, 9, 3, 2, 4, 1

Анализ пирамидальной сортировки

Время работы Heap определяется отношением $T(N) \leq T(2*N/3)+\theta(1)$

На основании первого случая теоремы получаем оценку $T(N)=O(log(N))$,

(высота пирамиды из n элементов равна $O(log(N))$

Время построения пирамиды: $O(N \cdot log(N))$

Время сортировки: $O(N \cdot log(N))$

Общее время: $T(N) = O(N \cdot log(N))$

Быстрая сортировка

Алгоритм разработан Чарльзом Хоаром в 1960 г.

1. Разделяем массив на два подмассива *[1 ...m] и [m +1 . ..N]*, причем $\forall i, j : 1 \le i \le m \wedge m < j \le N \Rightarrow a_i \le a_j$
2. Рекурсивно сортируем получившиеся два подмассива.

Быстрая сортировка использует стратегию Разделяй и властвуй

Выбираем в массиве некоторый элемент, который будем называть *опорным элементом*. Известные стратегии: выбирать постоянно один и тот же элемент, например, первый, средний или последний по положению; выбирать элемент со случайно выбранным индексом.

Операция *разделения* массива: реорганизуем массив таким образом, чтобы все элементы, меньшие или равные опорному элементу, оказались слева от него, а все элементы, большие или равные опорному — справа от него.

Два индекса — l и r, приравниваются к минимальному и максимальному индексу разделяемого массива.

Вычисляется индекс опорного элемента. Устанавливаются i=l; j=r.

Индекс i последовательно увеличивается до тех пор, пока i-й элемент не превысит опорный.

Индекс j последовательно уменьшается до тех пор, пока j-й элемент не окажется меньше опорного.

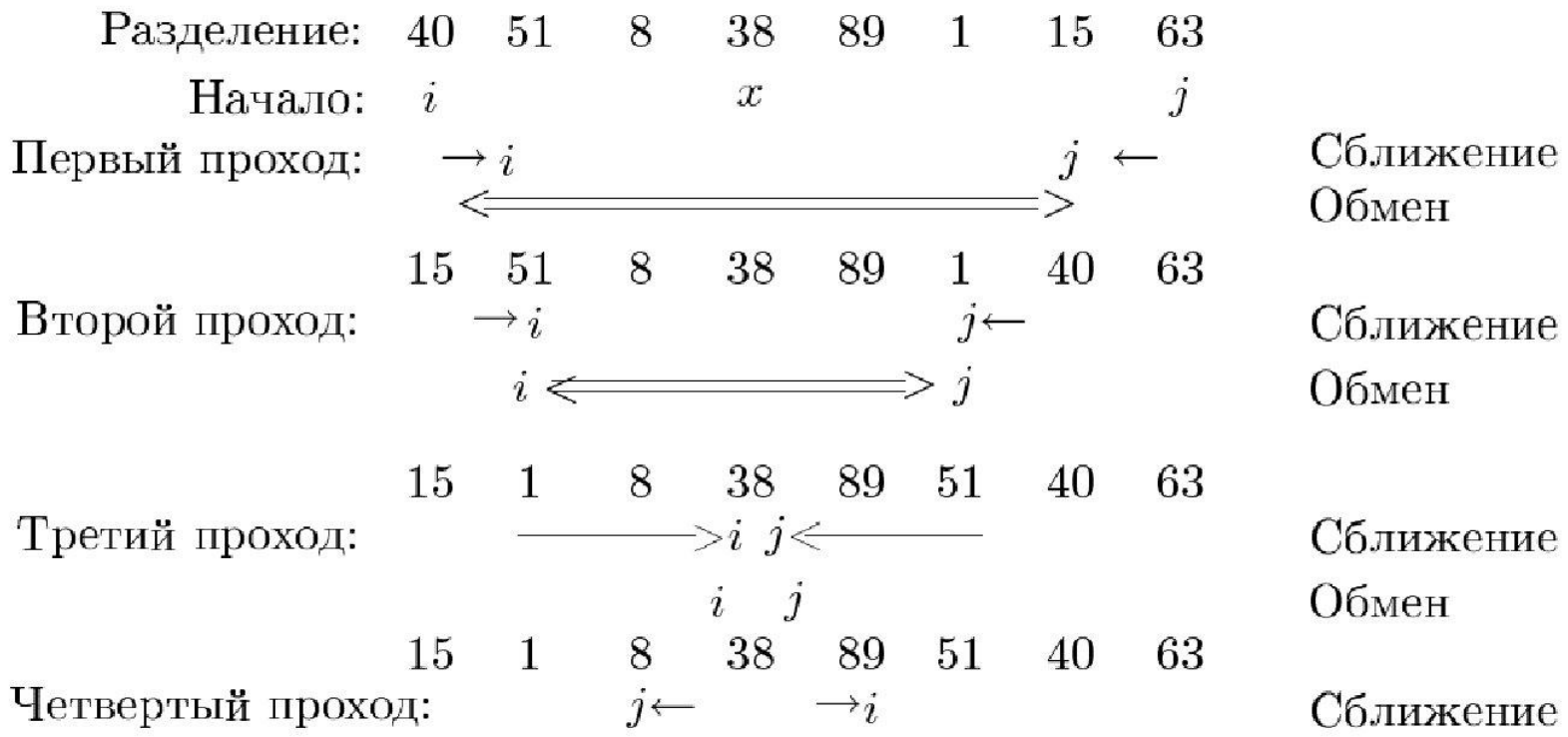
Если i <j — найденную пару элементов нужно обменять местами и продолжить операцию разделения с i++, j—.

Если i = j — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент.

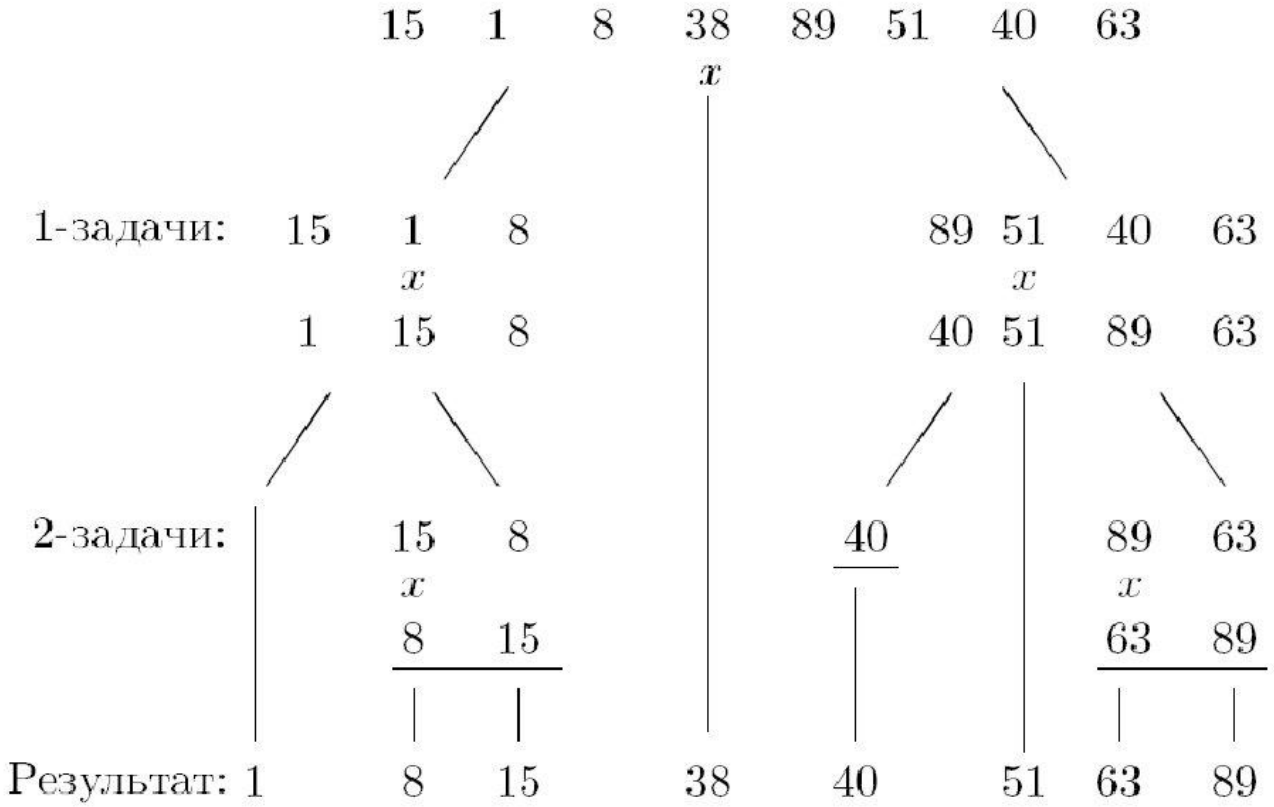
Рекурсивно упорядочиваем левую и правую часть массива.

Базой рекурсии являются наборы, состоящие из одного или двух элементов. Первый возвращается в исходном виде, во втором, при необходимости, сортировка сводится к перестановке двух элементов. Все такие отрезки уже упорядочены в процессе разделения.

Работа быстрой сортировки



Работа быстрой сортировки



Алгоритм быстрой сортировки

```
static void quickSort(int[] a, int l, int r)    {
        int temp;
        int x = a[l + (r - l) / 2];
        int i = l;
        int j = r;
       while (i <= j)
        {
            while (a[i] < x) i++;
            while (a[j] > x) j--;
            if (i <= j)
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
                i++;
                j--;
            }
        }
        if (i < r)  quickSort(a, i, r);
        if (l < j)  quickSort(a, l, j);
    }
```

Анализ быстрой сортировки (наихудший случай)

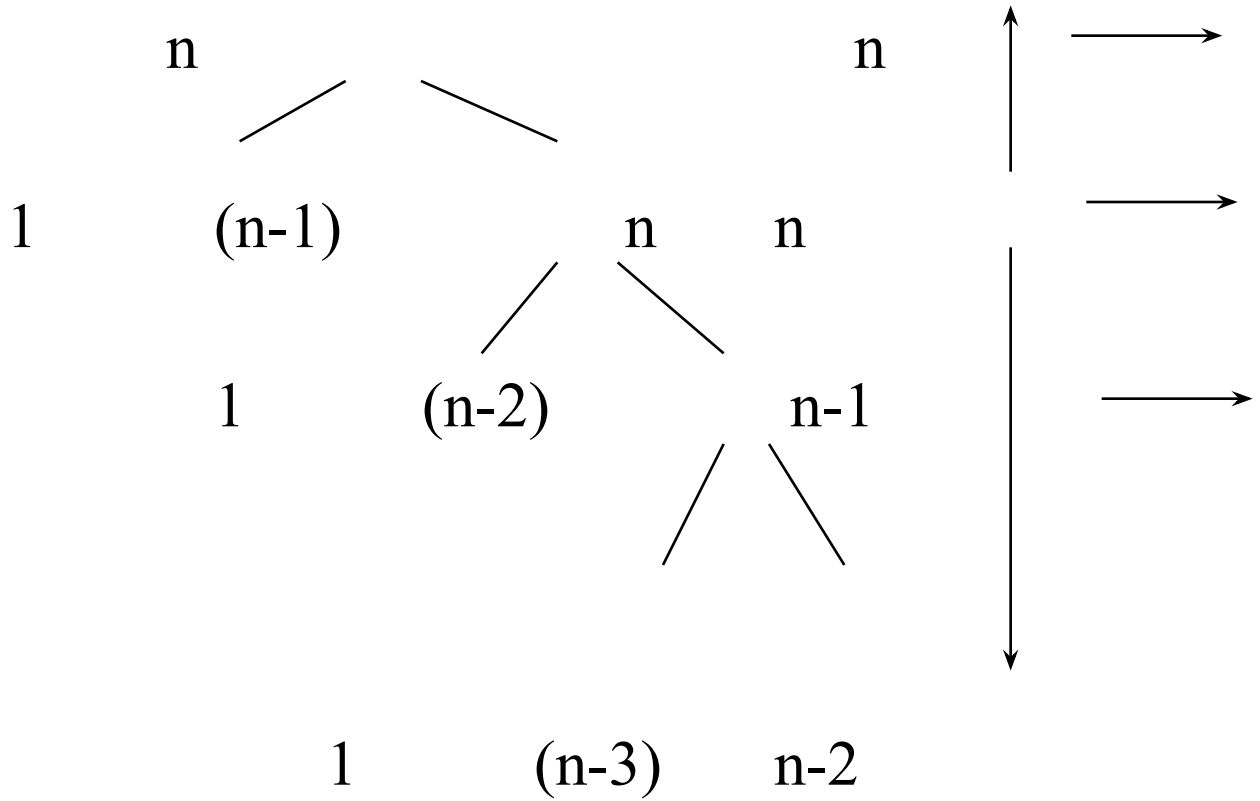
Предположим, что все элементы различны.

Наихудший случай: когда при разделении один из массивов не имеет элементов.

$$
\begin{aligned}
T(n) &= T(0) + T(n - 1) + \Theta(n) \\
&= \Theta(1) + T(n - 1) + \Theta(n) \\
&= T(n - 1) + \Theta(n) \\
&= \Theta(n^2)
\end{aligned}
$$

Анализ быстрой сортировки (наихудший случай)

$T(n)=T(n-1)+n$



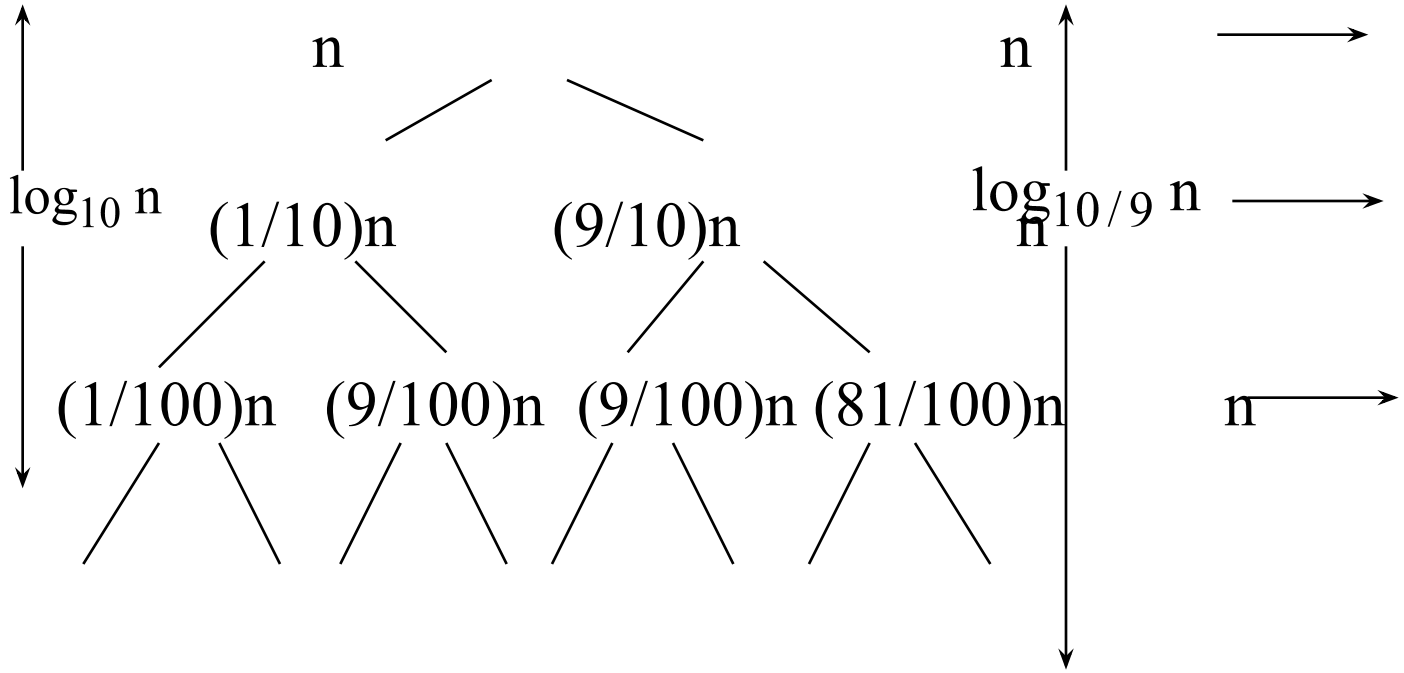
Анализ быстрой сортировки (наилучший случай)

Наилучший случай: когда при разделении массив разделяется на равные части.

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \bullet log(n))$$

Анализ быстрой сортировки (средний случай)

$T(n)=T(9n/10)+ T(n/10)+ n$



$T(n)=\Theta(n*\log n)$

Сравнение сортировок

Алгоритм	Лучший случай	Средний случай	Худший случай	Память	Свойства
Сортировка вставками (Insertion Sort)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Устойчивый, на месте, online
Сортировка выбором (Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Устойчивость зависит от реализации, на месте
Быстрая сортировка (Quick Sort)	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	Неустойчивая
Сортировка слиянием (Merge sort)	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Устойчивая
Пирамидальная сортировка (Heap sort)	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	На месте, неустойчивый

Рандомизированная быстрая сортировка

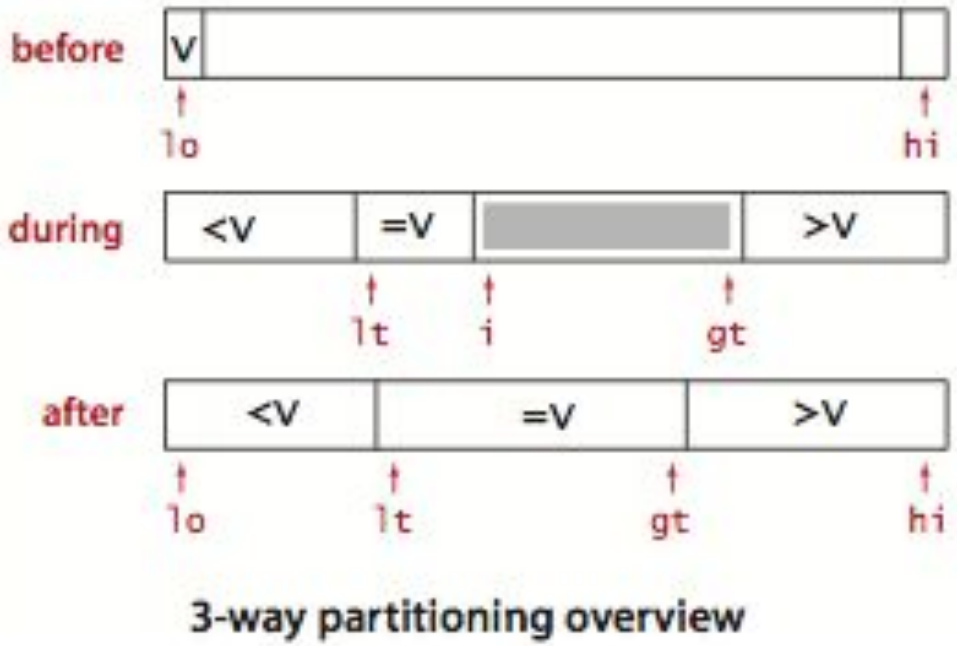
- Опорный элемент выбирается случайным образом

- Время работы не зависит от порядка элементов во входных данных;

- Не нужно предположений о распределении входных данных;

- Нет входных данных, которые приводят к наихудшему случаю;

Усовершенствованная быстрая сортировка



3-way partitioning overview

- a[i] < v: обменять a[lt] и a[i] lt++ , i++
- a[i] > v: обменять a[i] и a[gt] gt--
- a[i] = v: i++