{ ЛЕКЦИЯ 3 }

{ Алгоритмы сортировки и поиска }

{ Алгоритмы сортировки }

Задача сортировки.

Пусть есть множество из N элементов R_1, R_2,… R_N. Каждый элемент характеризуется некоторой информацией и ключом K_1. На множестве ключей определены операции сравнения: «>», «<» и т.д.

Задачей сортировки является нахождение такой перестановки ключей p_1, p_2,… p_N, после которой ключи расположились бы в заданном порядке:

- неубывания $$k_{p_1} \le k_{p_2} \le \ldots \le k_{p_n}$$
- невозрастания $$k_{p_1} \ge k_{p_2} \ge \ldots \ge k_{p_n}$$

Для классификации алгоритмов сортировки используются:

- сложность;
- потребности в **дополнительной** памяти;
- области хранения данных (внутренняя (в ОЗУ) и внешняя сортировка (вне ОЗУ));
- свойство устойчивые (меняется ли положение элементов с одинаковыми ключами);
- наличие в алгоритме операции сравнения.

{ Случайная сортировка }

Алгоритм:

- перемешать последовательность случайным образом;
- проверить выполнено ли условие сортировки.

Возможно, самый неэффективный алгоритм.

Сложность: **O(n*n!)**.

(Колода в 32 карты будет сортироваться компьютером в среднем $2{,}7 \cdot 10^{19}$ лет.)

{ Сортировка выбором }

Алгоритм:

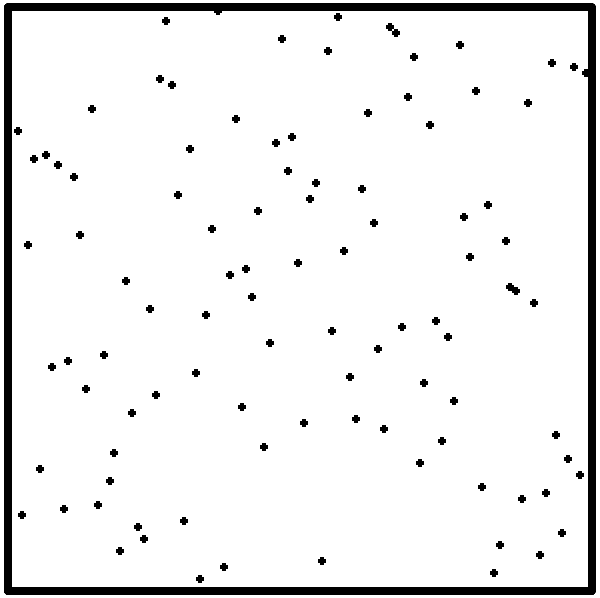
- найти наименьший элемент в неотсортированной части массива;
- поставить его в начало;
- сдвинуть начало неотсортированной части.

Сложность: **O(*n*²)**.

```
def selection_sort(arrayToSort):
    a = arrayToSort
    n = len(a)
    for i in range(n):
        idxMin = i
        for j in range(i+1, n):
            if a[j] < a[idxMin]:
                idxMin = j
        tmp = a[idxMin]
        a[idxMin] = a[i]
        a[i] = tmp
    return a
```

```
ary = [0,3,5,1,2,3,5,4,2,34,43,24]
print (selection_sort(ary))
```

```
[0, 1, 2, 2, 3, 3, 4, 5, 5, 24, 34, 43]
```



{ Сортировка вставками }

Алгоритм:

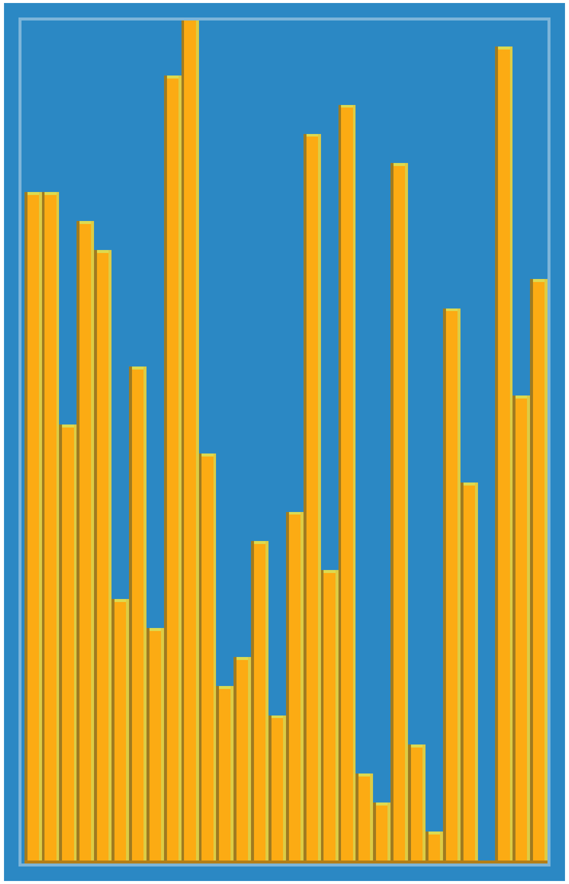
- из неотсортированной части берется элемент;
- вставляется в отсортированную часть на своё мосто (в начале массива).

Сложность: **O(*n*²)**.

```
def insertion_sort(arrayToSort):
    a = arrayToSort
    n = len(a)
    for i in range(n):
        v = a[i]
        j = i
        while (a[j-1] > v) and (j > 0):
            a[j] = a[j-1]
            j = j - 1
        a[j] = v
    return a
```

```
ary = [0,3,5,1,2,3,5,4,2,34,43,24]
print (insertion_sort(ary))
```

```
[0, 1, 2, 2, 3, 3, 4, 5, 5, 24, 34, 43]
```



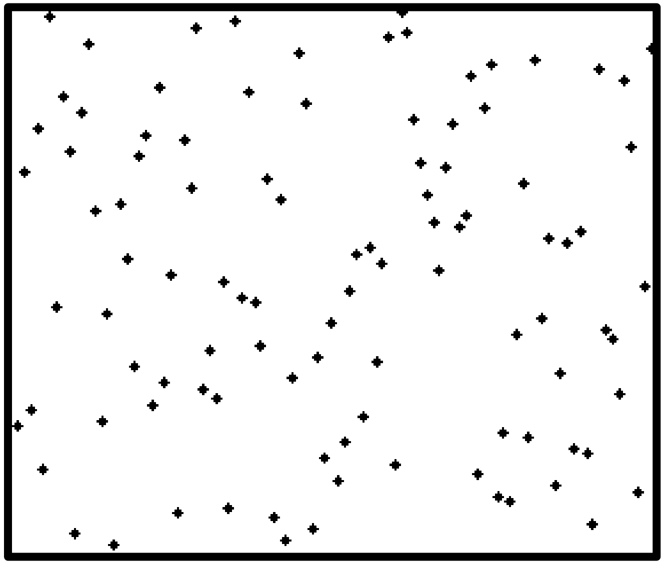
{ Сортировка “Методом Пузырька” }

Алгоритм:

- последовательно сравниваются пары элементов идущих друг за другом;
- в случае несоответствия выбранному порядку меняются местами.

Сложность: **O(*n*²)**.

```
def bubble_sort(arrayToSort):
    a = arrayToSort
    n = len(a)
    for i in range(n,0,-1):
        for j in range(1, i):
            if a[j-1] > a[j]:
                tmp = a[j-1]
                a[j-1] = a[j]
                a[j] = tmp
    return a
```



```
ary = [0,3,5,1,2,3,5,4,2,34,43,24]
print (bubble_sort(ary))
```

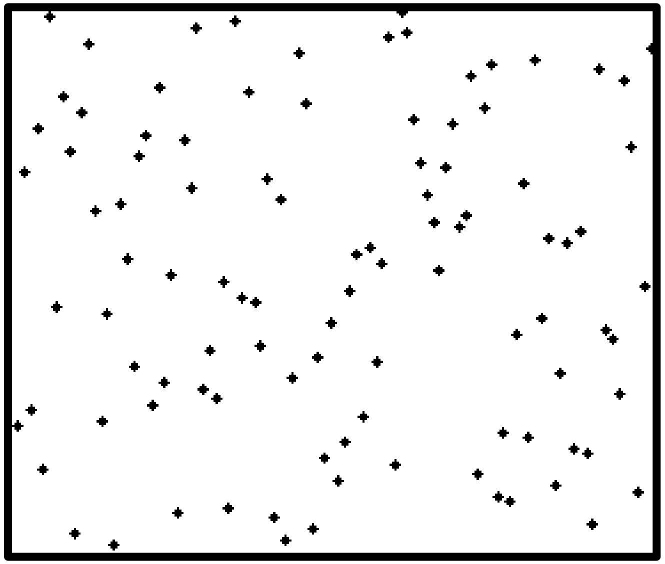
```
[0, 1, 2, 2, 3, 3, 4, 5, 5, 24, 34, 43]
```

{ Сортировка “Методом Пузырька” }

Алгоритм:
- последовательно сравниваются пары элементов идущих друг за другом;
- в случае несоответствия выбранному порядку меняются местами.

Сложность: **O(*n*²)**.

```python
def bubble_sort(arrayToSort):
    a = arrayToSort
    n = len(a)
    for i in range(n,0,-1):
        for j in range(1, i):
            if a[j-1] > a[j]:
                tmp = a[j-1]
                a[j-1] = a[j]
                a[j] = tmp
    return a
```



```python
ary = [0,3,5,1,2,3,5,4,2,34,43,24]
print (bubble_sort(ary))
```

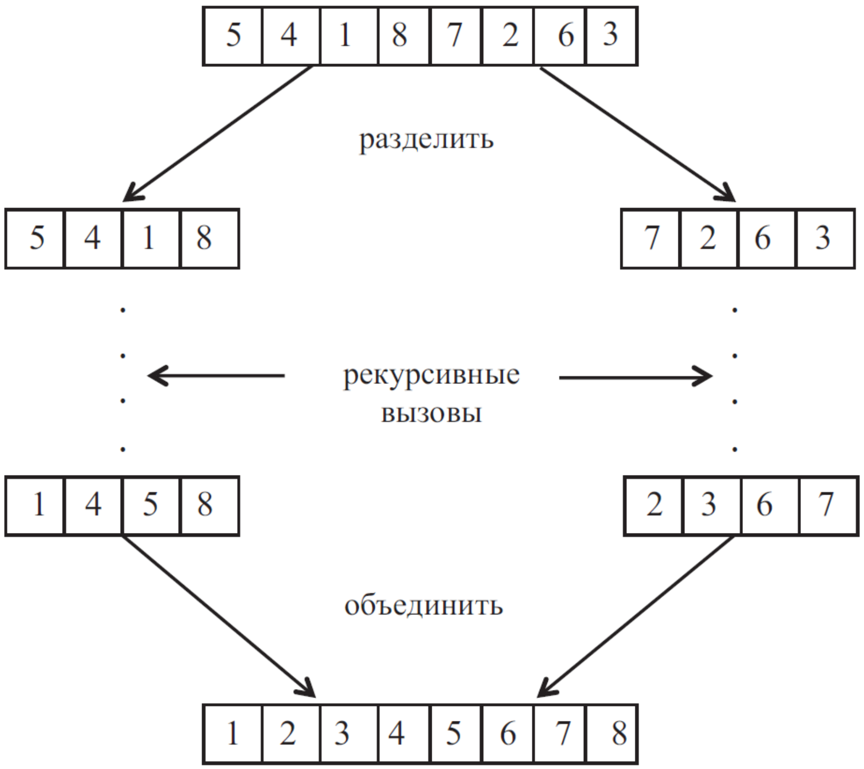
```
[0, 1, 2, 2, 3, 3, 4, 5, 5, 24, 34, 43]
```

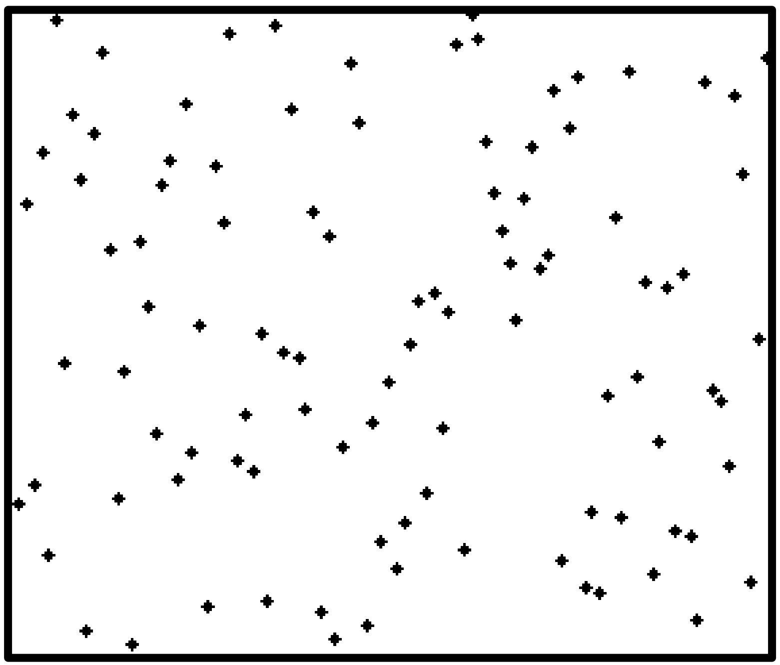
{ Сортировка слиянием }

Алгоритм:

- Сортируемый массив разбивается на две части примерно одинакового размера;
- Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
- Два упорядоченных массива половинного размера соединяются в один.

Сложность: **O(*n log*$_2$ *n*)**.



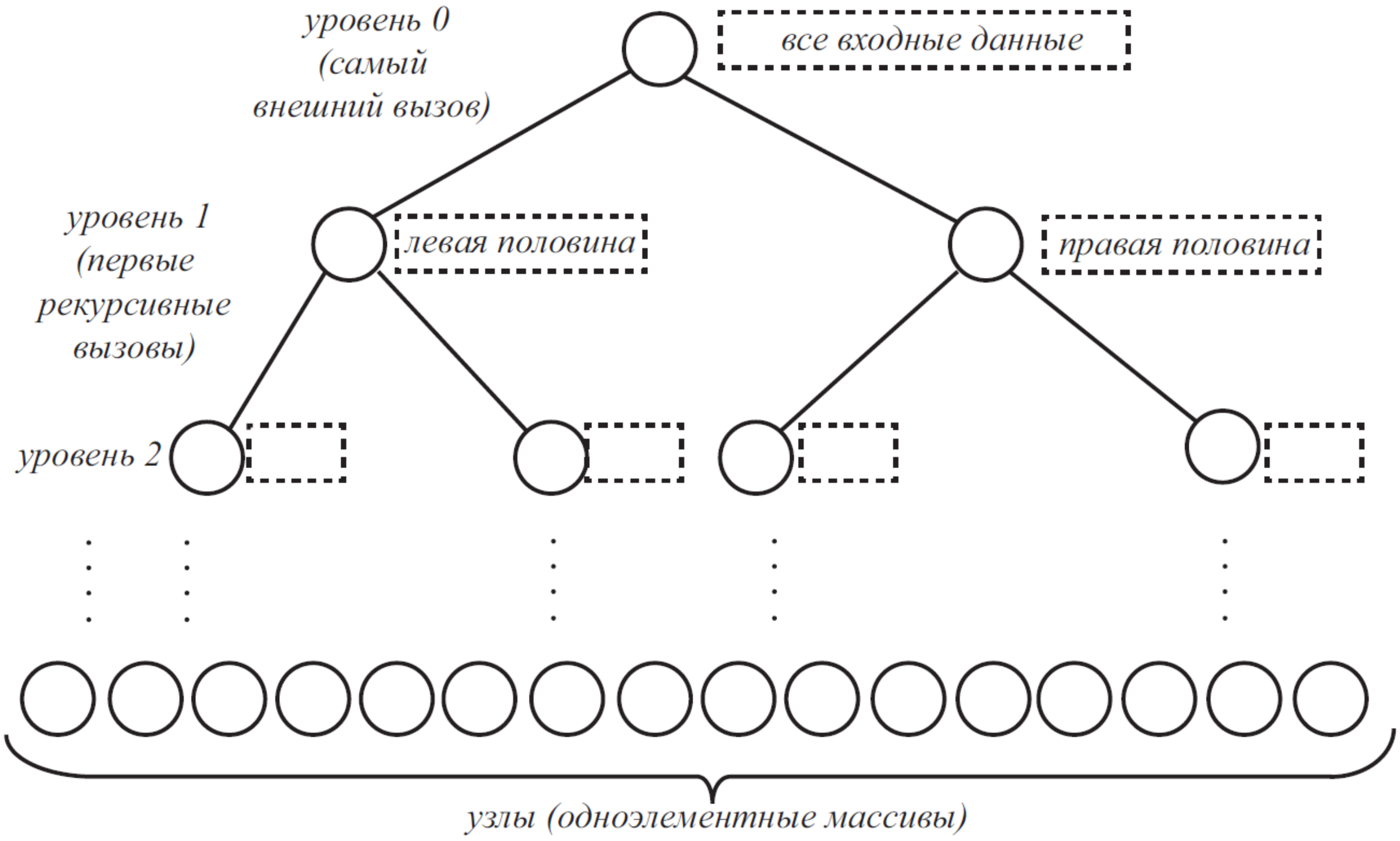


{ Сортировка слиянием }

Алгоритм:

- Сортируемый массив разбивается на две части примерно одинакового размера;
- Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
- Два упорядоченных массива половинного размера соединяются в один.

Сложность: $\mathbf{O}(n \log_2 n)$.

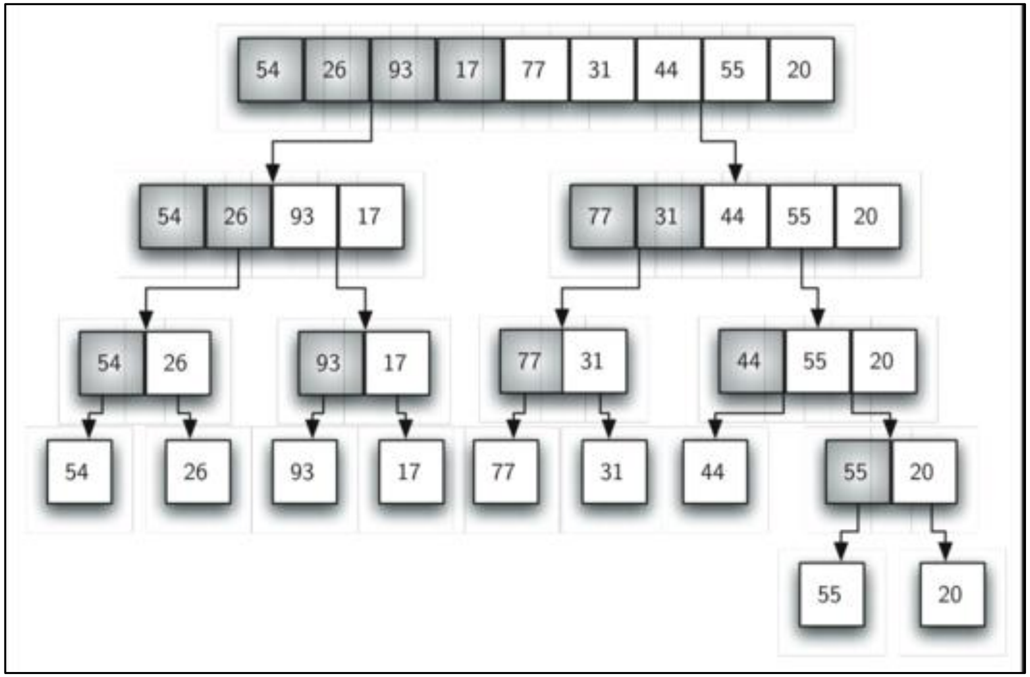


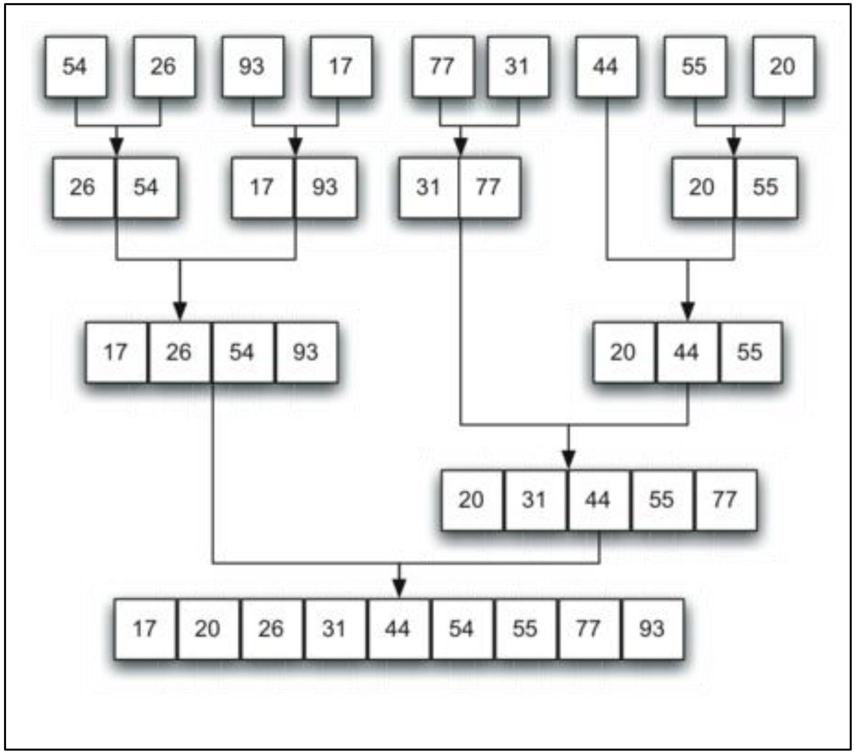
{ Сортировка слиянием }

Алгоритм:

- Сортируемый массив разбивается на две части примерно одинакового размера;
- Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
- Два упорядоченных массива половинного размера соединяются в один.

Сложность: $\mathbf{O(\mathit{n} \log_2 \mathit{n})}$.





{ Сортировка слиянием }

```
def merge_sort(arrayToSort):
    if len(arrayToSort)>1:
        mid = len(arrayToSort) // 2
        lefthalf = arrayToSort[:mid]
        righthalf = arrayToSort[mid:]

        merge_sort(lefthalf)
        merge_sort(righthalf)

        i, j, k = 0, 0, 0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                arrayToSort[k] = lefthalf[i];   i=i+1
            else:
                arrayToSort[k]=righthalf[j];  j=j+1
            k=k+1
        while i < len(lefthalf):
            arrayToSort[k]=lefthalf[i];    i=i+1; k=k+1
        while j < len(righthalf):
            arrayToSort[k]=righthalf[j];  j=j+1; k=k+1
```

```
alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
merge_sort(alist); print(alist)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

{ Быстрая сортировка }

Является улучшенным вариантом алгоритма сортировки с помощью прямого обмена («Пузырьковая сортировка»), весьма низкой эффективности. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

Таким образом, улучшение неэффективного прямого метода сортировки дало один из наиболее эффективных методов.

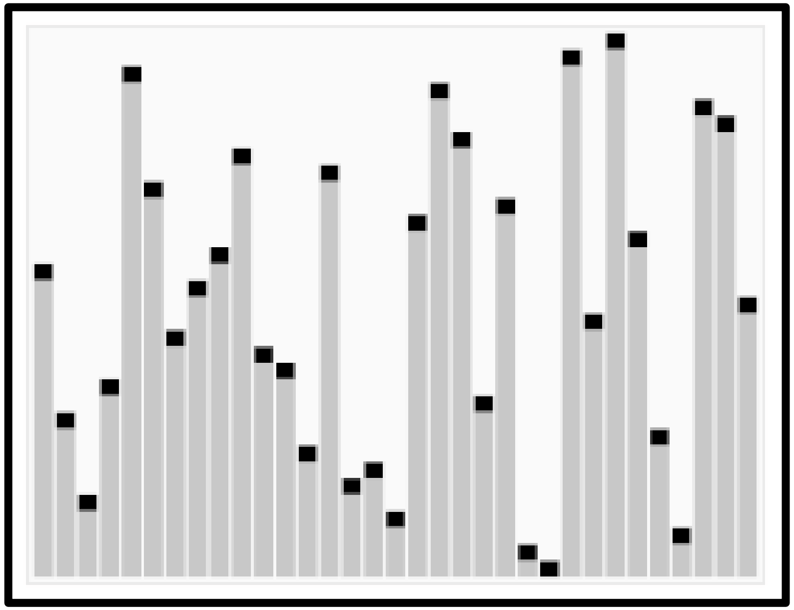
Алгоритм:

- выбрать (опорным) элемент из массива;
- перераспределить элементы в массиве так, что элементы меньше опорного помещаются перед ним, а больше или равные после;
- применить первые два шага к подмассивам слева и справа от опорных элементов, пока в подмассивах не останется не более одного элемента.

Сложность: Средняя $\mathbf{O}(n \log_2 n)$, Худшая $\mathbf{O}(n^2)$.

Худший случай.

Если каждое разделение даёт два подмассива размерами 1 и *n*-1, т.е. при каждом разбиении больший массив будет укорачиваться на 1. Это может произойти, если за опорный будет выбраться либо наименьший, либо наибольший элемент из всех обрабатываемых.



При выборе опорного элемента — первого или последнего в массиве, — такой эффект даст уже отсортированный массив.

{ Быстрая сортировка }

```
def quick_sort(a, l, r):
    if (r > l):
        v, i, j = a[r], l - 1, r

        while (True):
            i, j = i + 1, j - 1
            while(a[i] < v): i = i + 1
            while(a[j] > v): j = j - 1
            if (i >= j): break
            a[i], a[j] = a[j], a[i]

        a[i], a[r] = a[r], a[i]

        quicksort(a, l, i - 1)
        quicksort(a, i + 1, r)
```

```
ary = [7,8,1,2,3,4,13,5,1,2,44,5,1]
quick_sort(ary, 0, len(ary)-1)
print (ary)
```

```
[1, 1, 1, 2, 2, 3, 4, 5, 5, 7, 8, 13, 44]
```

{ Нахождение медианы }

k-й порядковой статистикой массива называется *k*-й по величине элемент массива:

- максимальный (минимальный) элемент массива: 1-ая (N-ая) порядковая статистика;
- медиана – «средний» по величине элемент, примерно половина элементов не больше, примерно половина элементов не больше, примерно половина – не меньше.

Алгоритм нахождения *k*-й порядковой статистики методом «разделяй и властвуй»:

- выберем случайным образом элемент *v* массива *S*;
- разобьём массив на три: S_l, элементы которого меньше, чем *v*; S_v, элементы которого равны *v*, и S_r, элементы которого больше, чем *v;*
- введём функцию Selection(S, k), где *S* — массив, а *k* — номер порядковой статистики:

$$selection(S,k)=\begin{cases} selection(S_l,k), & если \quad k\le |S_l| \\ v, & если \quad |S_l|<k\le |S_l|+|S_v| \\ selection(S_r,k-|S_l|-|S_v|), & если \quad k>|S_l|+|S_v| \end{cases}$$

{ Алгоритмы поиска }

Задача поиска.

Пусть есть множество из N элементов R_1, R_2,… R_N. Каждый элемент характеризуется некоторой информацией и ключом K_i. На множестве ключей определены операции сравнения: «>», «<» и т.д.

Задачей поиска является нахождение индекса ключа, совпадающего со значением *key.*

Алгоритмы поиска:

- линейный, последовательный поиск (неотсортированный массив);
- поиск сужением зоны (отсортированный массив).

Выбором структуры данных (устройством хранимой информации) можно расставить приоритеты:

- быстрое и простое изменение данных;
- Быстрый поиск.

{ Последовательный поиск }

Рассмотрим алгоритм поиска с помощью последовательного сравнения.

```python
def dummy_search (a, key):
    n = len(a)
    for i in range(n):
        if a[i] == key:
            return i
    return n
```

```python
ary = [7,8,1,2,3,4,13,5,1,2,44,5,1]
print(dummy_search (ary, 13))
```

```
6
```

```c
int dummy_search(int a[], int N, int key) {

    for (int i = 0; i < N; i++) {
        if (a[i] == key) {
            return i;
        }
    }
    return N;
}
```

```python
def clever_search (a, key):
    n = len(a)
    i=0
    while a[i]!=key:
        i=i+1
    return i
```

```python
ary2 = [7,8,1,2,3,4,13,5,1,2,44,5,1, 13]
print(clever_search (ary2, 13))
```

```
6
```

```c
int clever_search(int a[], int N, int key) {
    a[n] = key;
    int i;
    for (i = 0; a[i] != key; i++) {
        ;
    }
    return i;
}
```

Методы сужения области – аналогии с поиском корня.

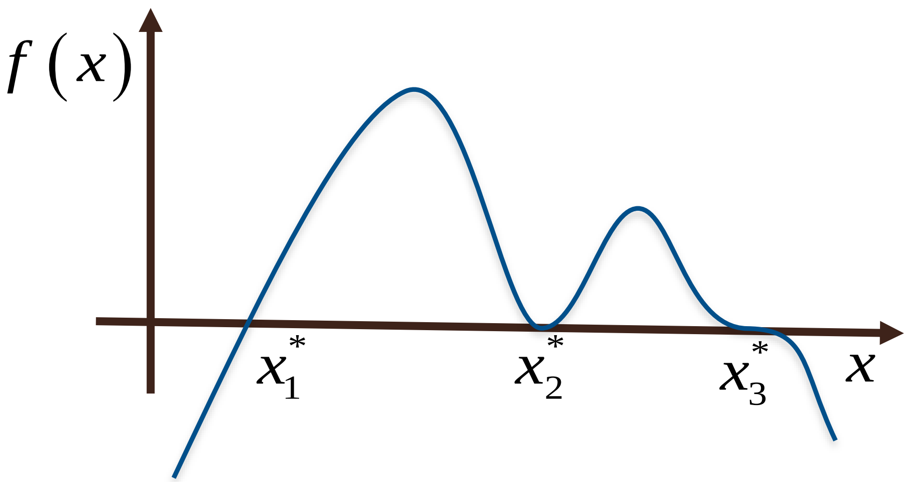
Корни нелинейного уравнения.

Пусть дана некоторая функция $f(x)$ и требуется найти значения x, для которых

$$f(x)=0.$$

Определение. Значение x^*, при котором $f(x^*) = 0$, называется корнем (или решением) уравнения.

Геометрически корень уравнения есть точка пересечения графика функции $y = f(x)$ с осью абсцисс.



На графике изображены три корня:

$$f(x_1^*)=0,\quad f(x_2^*)=0,\quad f(x_3^*)=0.$$

При этом они отличаются

$$f'(x_1^*)>0,\quad f'(x_2^*)=0,\quad f'(x_3^*)=0.$$

Корни типа x_1^* называются простыми, а x_1^*, x_2^* – кратными (непростыми).

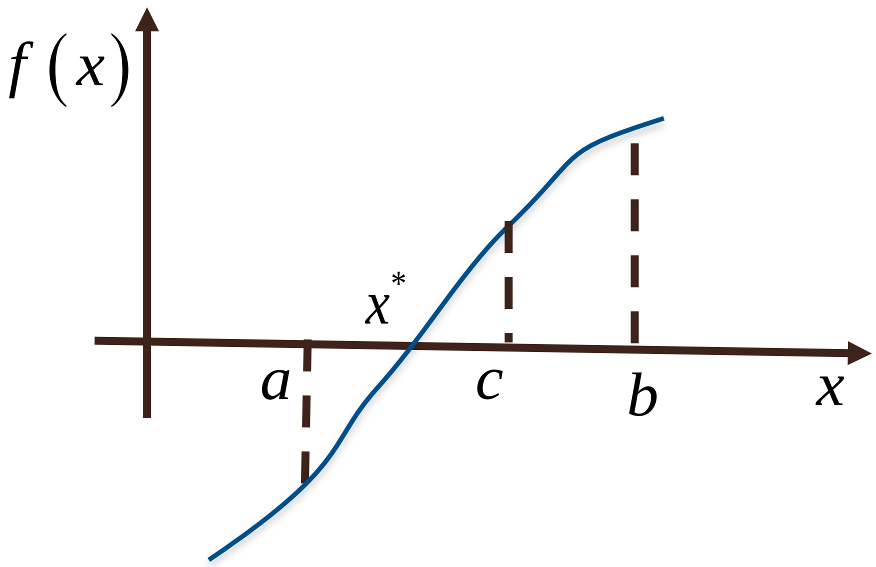
Большинство методов решения уравнений ориентировано на отыскание простых корней.

Методы сужения области – аналогии с поиском корня.

Двухточечные методы (уменьшение отрезка локализации).

Идея локализации: Пусть известен отрезок локализации корня $[a_0,b_0]$ найдём отрезок локализации меньшей длинны $[a_1,b_1]$, потом ещё меньшей и так пока отрезок не будет меньше заданной точности ε. Тогда корень можно выразить как

$$x = \frac{a_n + b_n}{2}, \quad f(a_n)\,f(b_n) < 0, \quad |a_n - b_n| < \varepsilon.$$



Алгоритм. Известен отрезок $[a_0,b_0]$.

Будем повторять следующие действия пока $|a_i - b_i| > \varepsilon$:

1. Выберем точку c.
2. Сравним $f(a_i)$, $f(b_i)$, $f(c)$
   - если $f(a_i)\,f(c) < 0$, то $b_{i+1} = c$,
   - если $f(b_i)\,f(c) < 0$, то $a_{i+1} = c$.

После удовлетворения заданной точности (за n итераций):

$$x = \frac{a_n + b_n}{2}.$$

(для решения задачи поиска все операции деления выполняются целочисленно)

Методы сужения области – аналогии с поиском корня.

Двухточечные методы (уменьшение отрезка локализации).

Методы выбора точки *c*:

Метод половинного деления (метод дихотомии)

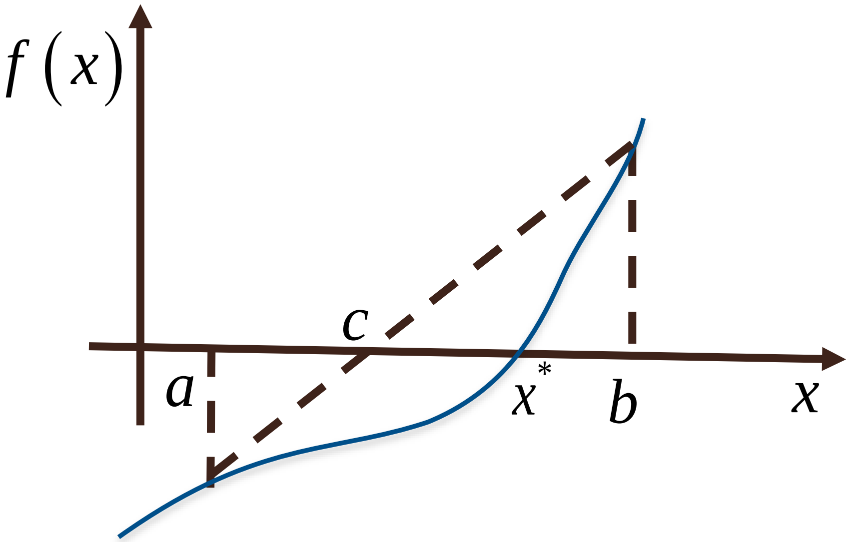
$$c = \frac{a_i + b_i}{2}.$$

Метод золотого сечения

$$c = a_i + \frac{b_i - a_i}{\Phi} \quad или \quad c = b_i - \frac{b_i - a_i}{\Phi} \quad \left(\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618 \right).$$

Метод хорд

$$c = b_i - \frac{b_i - a_i}{f(b_i) - f(a_i)} f(b_i).$$



(для решения задачи поиска все операции деления выполняются целочисленно)

{ Сортировка в *Python* }

В Python есть встроенная функция sorted() для сортировки итерируемых объектов и метод list.sort() для сортировки списка с заменой исходного.

Сделать обычную сортировку по возрастанию просто — достаточно вызвать функцию sorted(), которая вернёт новый отсортированный список:

```
sorted([5, 2, 3, 1, 4])
```

```
[1, 2, 3, 4, 5]
```

```
a = [5, 2, 3, 1, 4]
print (a.sort())
```

```
[1, 2, 3, 4, 5]
```

У list.sort() и sorted() есть параметр key для указания функции, которая будет вызываться на каждом элементе до сравнения. Значение параметра key должно быть функцией, принимающей один аргумент и возвращающей ключ для сортировки.

```
student_tuples = [ ('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10) ]
sorted (student_tuples, key = lambda student: student[2])
```

```
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

{ ВСЁ }