



Использование рекуррентных уравнений для оценки времени работы алгоритма (на примере алгоритмов поиска и внутренней сортировки)

ОПРЕДЕЛЕНИЯ

Соотношения, которые связывают одни и те же функции, но с различными значениями аргументов, называются **рекуррентными соотношениями или рекуррентными уравнениями**.

Рекуррентное уравнение будем называть **правильным**, если значения аргументов у любой из функций в правой части соотношения меньше значения аргументов у любой из функций в левой части соотношения; если аргументов несколько, то достаточно уменьшения одного из них.

Правильное рекуррентное уравнение называется **полным**, если оно определено для всех допустимых значений аргументов.

$$\begin{cases} T(n) = T(n-1) + C_2, n \geq 1 \\ T(0) = C_1 \end{cases}$$

В дальнейшем будем предполагать (если не оговорено иное), что область определения функции $T(n)$ – это множество неотрицательных целых чисел $\{0, 1, 2, \dots\}$ и сама функция $T(n)$ принимает только неотрицательные целочисленные значения.

Это допущение вызвано тем, что функция $T(n)$ будет нами чаще всего использоваться для описания времени работы алгоритма.

Полное рекуррентное соотношение?

нет
$$\begin{cases} T(n) = T(n-1) + c_1 \cdot n, n \geq 2 \\ T(0) = c_1 \end{cases}$$

да
$$\begin{cases} T(n) = T(n-1) + c_1 \cdot n, n \geq 2 \\ T(1) = c_1 \end{cases}$$

да
$$\begin{cases} T(n) = T(n-1) + c_1 \cdot n, n \geq 1 \\ T(0) = c_1 \end{cases}$$

да
$$\begin{cases} T(n) = 2 \cdot T(n-2) + c_1 \cdot (n-1), n \geq 2 \\ T(1) = c_2, T(0) = c_3 \end{cases}$$

нет
$$\begin{cases} T(n) = T(n-2) + c_1 \cdot n, n \geq 1 \\ T(0) = c_2 \end{cases}$$

Поиск максимального и минимального элементов в массиве

Задан массив из n элементов. Рассмотрим два алгоритма нахождения максимального и минимального элементов.

Оценим число операций сравнения, выполненных каждым из алгоритмов.

Алгоритм 1

Последовательный поиск \max и \min

Первый элемент массива полагаем в качестве **max** и **min**.

Каждый из оставшихся $(n - 1)$ элементов сравниваем с **max** и **min**, и, если надо, то корректируем значения **max** и **min**.

```
template <class Iter>
std::pair<Iter, Iter> MinMaxElement2(Iter begin, Iter end) {
    std::pair<Iter, Iter> res = std::make_pair(begin, begin);
    for (Iter it = begin + 1; it != end; ++it) {
        if (*it < *res.first) {
            res.first = it;
        }
        if (*it > *res.second) {
            res.second = it;
        }
    }
    return res;
}
```

Оценим число операций сравнения для алгоритма последовательного поиска максимального и минимального элементов массива:

Способ 1. Просто подсчитаем:

$$0 + (n - 1) \cdot 2 = 2n - 2$$

Способ 2. Составим рекуррентное соотношение

$$\begin{cases} T(n) = T(n - 1) + 2, n \geq 2 \\ T(1) = 0 \end{cases}$$

Решение уравнения
методом ИТЕРАЦИЙ

$$T(n) = \underbrace{T(n - 1) + 2}_{1\text{-й шаг}} = [T(n - 1) = T(n - 2) + 2] =$$

$$\underbrace{T(n - 2) + 2 + 2}_{2\text{-й шаг}} = [T(n - 2) = T(n - 3) + 2] = \underbrace{T(n - 3) + 2 + 2 + 2}_{3\text{-й шаг}} = \dots$$

$$\dots = \underbrace{T(n - m) + 2 \cdot m}_{m\text{-й шаг}} = \left[\begin{matrix} T(n - m) = T(1) \\ n - m = 1; m = n - 1 \end{matrix} \right] = \underbrace{T(1) + 2 \cdot (n - 1)}_{(n-1)\text{-й шаг}} =$$

$$= 0 + 2(n - 1) = 2n - 2$$

Алгоритм 2

«Разделяй и властвуй»

(«метод турниров»)

1. Разделим массив на две части (предположим, что $n=2^k$).

2. В каждой из частей этим же алгоритмом найдём локальные $(\text{max}_1, \text{min}_1)$ $(\text{max}_2, \text{min}_2)$.

Если в рассматриваемой области остаётся только два элемента, то деление не выполняем, а за одно сравнение определим максимальный и минимальный элемент этой области.

3. Полагаем

max=наибольший $(\text{max}_1, \text{max}_2)$,

min=наименьший $(\text{min}_1, \text{min}_2)$.

```
template <class Iter>
std::pair<Iter, Iter> MinMaxElement(Iter begin, Iter end) {
    size_t n = end - begin;
    if (n == 1) {
        return std::make_pair(begin, begin);
    } else if (n == 2) {
        Iter first = begin;
        Iter second = begin + 1;
        if (*first < *second) {
            return std::make_pair(first, second);
        } else {
            return std::make_pair(second, first);
        }
    } else {
        Iter mid = begin + n / 2;
        std::pair<Iter, Iter> r1 = MinMaxElement(begin, mid);
        std::pair<Iter, Iter> r2 = MinMaxElement(mid, end);
        return std::make_pair(
            *r1.first < *r2.first ? r1.first : r2.first,
            *r1.second > *r2.second ? r1.second : r2.second
        );
    }
}
```

$$\begin{cases} T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 2, n = 2^k, k \geq 2 \\ T(2) = 1 \end{cases}$$

Сведения из математики

Сумма геометрической прогрессии:

$$q^1 + q^2 + \dots + q^k = \frac{q \cdot (q^k - 1)}{q - 1}$$

$$q^0 + q^1 + q^2 + \dots + q^k = \frac{q^{k+1} - 1}{q - 1}$$

Решим рекуррентное уравнение методом итераций:

$$\begin{cases} T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 2, n = 2^k, k \geq 2 \\ T(2) = 1 \end{cases}$$

Решение:

$$\begin{aligned} T(n) &= \underbrace{2 \cdot T\left(\frac{n}{2}\right) + 2}_{1\text{-й шаг}} = \left[T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{2^2}\right) + 2 \right] = \underbrace{2^2 \cdot T\left(\frac{n}{2^2}\right) + 2^2 + 2^1}_{2\text{-й шаг}} = \dots \\ &= \underbrace{2^m \cdot T\left(\frac{n}{2^m}\right) + 2^m + \dots + 2^2 + 2^1}_{m\text{-й шаг}} = 2^m \cdot T\left(\frac{n}{2^m}\right) + \frac{2 \cdot (2^m - 1)}{2 - 1} = \left[\begin{array}{l} T\left(\frac{n}{2^m}\right) = T(2) \\ \frac{n}{2^m} = 2; \quad 2^m = \frac{n}{2}; \quad m = \log_2 n - 1 \end{array} \right] = \\ &= \underbrace{\frac{n}{2} \cdot T(2) + 2 \cdot \left(\frac{n}{2} - 1\right)}_{(\log_2 n - 1)\text{-й шаг}} = \frac{n}{2} \cdot 1 + 2 \cdot \left(\frac{n}{2} - 1\right) = \frac{3}{2} \cdot n - 2 \end{aligned}$$

Поиск максимального и минимального элементов в массиве

Последовательный поиск

$$2n - 2$$

Метод «разделяй и властвуй»

$$\frac{3}{2}n - 2$$

Оба алгоритма работают за время: $\Theta(n)$.

В алгоритме последовательного поиска можно получить оценку $2n - 3$, выбирая на начальном этапе за одно сравнение из первых двух элементов массива максимальный и минимальный элемент. Затем оставшиеся $n - 2$ элемента сравниваются с максимальным и минимальным: $1 + 2(n - 2) = 2n - 3$.

Алгоритм, основанный на принципе «разделяй и властвуй», выполняет меньше сравнений, но на практике может быть медленнее из-за накладных расходов, вызванных рекурсией.

Существует не рекурсивный алгоритм, который выполняет $\sim 3 \cdot n / 2$ сравнений.
 В этом алгоритме поддерживается значение **max**, **min** на том префиксе, который уже пройден:

01		23		45				n-2	n-1
1 сравнение: max , min		для двух элементов за 1 сравнение находим max' и min' ; затем ещё за 2 сравнения: max =наибольший (max , max') min =наименьший (min , min')		для двух элементов за 1 сравнение находим max' и min' ; затем ещё за 2 сравнения: max =наибольший (max , max') min =наименьший (min , min')		и т.д.			
		3 сравнения		3 сравнения				3 сравнения	

$$\left\lceil \frac{3 \cdot (n - 2)}{2} \right\rceil + 1$$

В C ++ не рекурсивный алгоритм, который выполняет $\sim 3 \cdot n / 2$ сравнений, реализован как функция `std::minmax_element()` библиотеки STL.
https://en.cppreference.com/w/cpp/algorithm/minmax_element

Поиск элемента в упорядоченном массиве

Задан упорядоченный массив A из n элементов: $a_0 \leq a_1 \leq \dots \leq a_{n-1}$

В массиве элементы могут повторяться.

Необходимо определить, есть ли среди элементов массива заданный элемент x .

? $x = 5$

0	1	2	3	4	5	6	7	8
2	4	5	5	5	27	61	91	97

$n = 9$

БИНАРНЫЙ ПОИСК (дихотомия)

1. Определяем границы $[q, r)$ области поиска как $q = 0, r = n$.

2. Определяем индекс центрального элемента области поиска

$$k = \left\lfloor \frac{q + r}{2} \right\rfloor$$

3. Сравниваем a_k — элемент последовательности и число x .

Если элементы совпадают, то поиск завершён.

Если $x < a_k$, то продолжаем аналогичные действия, изменяя правую границу области поиска на k .

Если $x > a_k$, то продолжаем аналогичные действия, изменяя левую границу области поиска на $k + 1$.

4. Алгоритм прекращает работу, как только будет найден требуемый элемент либо станет верным равенство $q = r$ (в этом случае элемента в последовательности нет).

? $x=5$

0	1	2	3	4	5	6	7	8	
2	4	5	5	<u>5</u>	27	61	91	97	$n=9$

```
def BinarySearch(a, x):  
    q = 0, r = len(a)  
    while q < r:  
        k = (q + r) // 2  
        if x == a[k]:  
            return True  
        else if x < a[k]:  
            r = k  
        else: # x > a[k]  
            q = k + 1  
  
    return False
```

LowerBound

Задача поиска индекса первого элемента, большего, чем x , либо равного ему.

В случае отсутствия в массиве подходящих элементов договоримся, что возвращаемое значение будет равно n .

0	1	2	3	4	5	6	7	8
2	4	5	5	5	27	61	91	97

$x = 5$ LowerBound(5) = 2

$x = 6$ LowerBound(6) = 5

$x = 100$ LowerBound(100) = 9

```
def LowerBound(a, x):  
    q = 0, r = len(a)  
    while q < r:  
        k = (q + r) // 2  
        if x ≤ a[k]:  
            r = k  
        else: # x > a[k]  
            q = k + 1  
  
    return q
```

UpperBound

Задача поиска индекса первого элемента, строго большего, чем x .

В случае отсутствия в массиве подходящих элементов договоримся, что возвращаемое значение будет равно n .

	0	1	2	3	4	5	6	7	8
$a[i]$	2	4	5	5	5	27	61	91	97

$n = 9$

$x = 5$ `UpperBound(5) =` 5

$x = 6$ `Upper Bound(6) =` 5

$x = 100$ `LowerBound(100)=` 9

```
def UpperBound(a, x):  
    q = 0, r = len(a)  
    while q < r:  
        k = (q + r) // 2  
        if x < a[k]:  
            r = k  
        else: # x ≥ a[k]  
            q = k + 1  
  
    return q
```

Задача поиска в массиве
заданного элемента $= x$.

Задача поиска индекса
первого элемента $\geq x$

Задача поиска индекса первого
элемента $> x$

```
def BinarySearch(a, x):
    q = 0, r = len(a)
    while q < r:
        k = (q + r) // 2
        if x == a[k]:
            return True
        else if x < a[k]:
            r = k
        else: # x > a[k]
            q = k + 1
    return False
```

```
def LowerBound(a, x):
    q = 0, r = len(a)
    while q < r:
        k = (q + r) // 2

        if x ≤ a[k]:
            r = k
        else: # x > a[k]
            q = k + 1
    return q
```

```
def UpperBound(a, x):
    q = 0, r = len(a)
    while q < r:
        k = (q + r) // 2

        if x < a[k]:
            r = k
        else: # x ≥ a[k]
            q = k + 1
    return q
```

```
def BinarySearch(a, x):
    q = 0, r = len(a)
    while q < r:
        k = (q + r) // 2
        if x == a[k]:
            return True
        else if x < a[k]:
            r = k
        else: # x > a[k]
            q = k + 1
    return False
```

Время работы алгоритма бинарного поиска

$$\begin{cases} T(n) = C_1 + T\left(\frac{n}{2}\right), n = 2^k, k \geq 1 \\ T(1) = C_2 \end{cases}$$

Решение:

$$\begin{aligned} T(n) &= \underbrace{C_1 + T\left(\frac{n}{2}\right)}_{\text{1-й шаг}} = \left[T\left(\frac{n}{2}\right) = C_1 + T\left(\frac{n}{2^2}\right) \right] = \underbrace{C_1 + C_1 + T\left(\frac{n}{2^2}\right)}_{\text{2-й шаг}} = \dots = \underbrace{m \cdot C_1 + T\left(\frac{n}{2^m}\right)}_{\text{m-й шаг}} = \\ &= \left[\begin{array}{l} T\left(\frac{n}{2^m}\right) = T(1) \\ \frac{n}{2^m} = 1; \quad 2^m = n \quad m = \log_2 n \end{array} \right] = \underbrace{C_1 \cdot \log_2 n + T(1)}_{(\log_2 n)\text{-й шаг}} = C_1 \cdot \log_2 n + C_2 \end{aligned}$$

Вычислительная сложность алгоритма в худшем случае $T(l) = O(\log l)$, $l = \Theta(n)$ — алгоритм полиномиальный.

В стандартной библиотеке языка C++

Функция `std::binary_search` выполняет бинарный поиск и возвращает логическое значение (есть элемент или нет).

Функции `std::lower_bound` и `std::upper_bound` действуют аналогично рассмотренным и возвращают итераторы.

В языке Java

для классов `Arrays` и `Collections` определён статический метод `binarySearch`, который совмещает в себе описанные выше функции `BinarySearch` и `LowerBound`, однако является менее гибким (при наличии в массиве нескольких элементов, равных искомому, метод может вернуть индекс любого).

В языке Python

бинарный поиск реализован в стандартном модуле `bisect`.

Задача

Задан упорядоченный массив из n элементов и число x .

Разработать алгоритм, который определит, сколько раз в массиве встречается заданное число?

Оценить время работы разработанного вами алгоритма.

Тернарный поиск –

метод поиска минимума или максимума функции на отрезке, которая либо сначала строго возрастает, затем строго убывает, либо наоборот.

Пусть имеется два значения L и R и известно, что максимум (минимум) функции $f(x)$ лежит на отрезке $[L, R]$.

Возьмем две точки на данном отрезке m_1 и m_2 , что $m_1 < m_2$, тогда имеется три варианта развития событий:

1. $f(m_1) < f(m_2) \Rightarrow$ максимум не может принадлежать отрезку $[L, m_1]$, продолжаем поиск на отрезке $[m_1, R]$;
2. $f(m_1) > f(m_2) \Rightarrow$ максимум не может принадлежать отрезку $[m_2, R]$ продолжаем поиски на отрезке $[L, m_2]$;
3. $f(m_1) = f(m_2) \Rightarrow$ максимум принадлежит отрезку $[m_1, m_2]$.

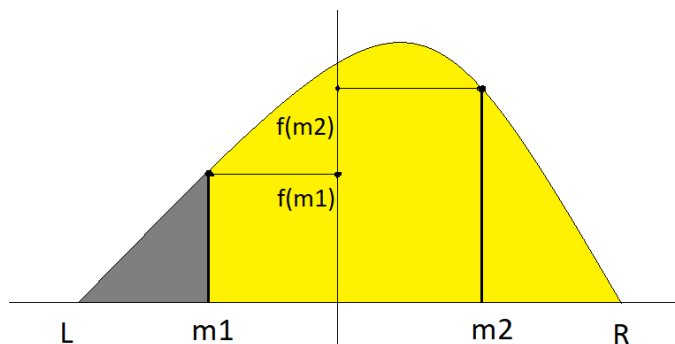
Используя данное соотношение, можно найти максимум с точностью до некоторого eps .

Говоря о выборе данных точек на каждом шаге, принято делить отрезок $[L, R]$ на три части точками:

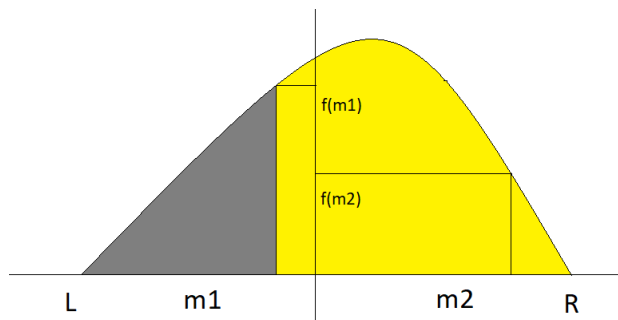
$$m_1 = L + (R - L)/3$$

$$m_2 = R - (R - L)/3$$

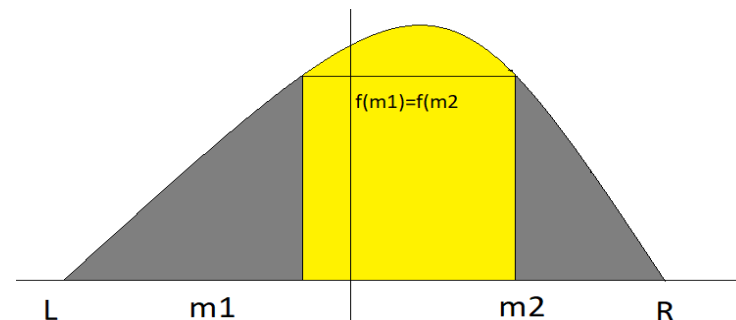
В ходе одного шага мы производим два вычисления функции и уменьшаем шаг в полтора раза, откуда следует, что время работы алгоритма $O(\log_{3/2}(R - L))$.



1-й случай



2-й случай



3-й случай

Алгоритмы сортировки

Пусть задана последовательность a_1, a_2, \dots, a_n из n элементов (записей), выбранных из множества, на котором задан линейный порядок.

Каждая запись a_j имеет ключ k_j , который управляет процессом сортировки.

Задача сортировки заключается в поиске перестановки $\pi = \pi_1, \pi_2, \dots, \pi_n$ этих n записей, после которой ключи записей расположились бы, например, в неубывающем порядке (будем предполагать порядок сортировки массива по неубыванию, если не оговорено иное):

$$k_{\pi_1} \leq k_{\pi_2} \leq \dots \leq k_{\pi_n}.$$

i	1	2	3	4	5	6	7	8	9	10
k_i	7	2	3	5	1	8	11	4	0	6
π_i	9	5	2	3	8	4	10	1	6	7

$$k_9 \leq k_5 \leq k_2 \leq k_3 \leq k_8 \leq k_4 \leq k_{10} \leq k_1 \leq k_6 \leq k_7$$

Алгоритм сортировки называют **устойчивым (стабильным)**, если в процессе сортировки относительное расположение элементов с одинаковыми ключами не изменяется:

$$\pi_i < \pi_j, \text{ если } k_{\pi_i} \leq k_{\pi_j} \text{ и } i < j.$$

i	1	2	3	4	5	6	7	8	9	10
k_i	7	2_1	2022	607	1	2_2	3_1	4	0	3_2
π_i	9	5	2	6	7	10	8	1	4	3

$$k_9 \leq k_5 \leq k_2 \leq k_6 \leq k_7 \leq k_{10} \leq k_8 \leq k_1 \leq k_4 \leq k_3$$

$$0 \leq 1 \leq 2_1 \leq 2_2 \leq 3_1 \leq 3_2 \leq 4 \leq 7 \leq 607 \leq 2022$$

Процесс сортировки данных может быть осуществлен различными алгоритмами. Если объем входных данных позволяет обходиться исключительно основной (оперативной) памятью, то говорят об алгоритмах **внутренней сортировки**, в противном случае – об алгоритмах **внешней сортировки**.

Большинство алгоритмов внутренней сортировки относятся к классу **сортировок сравнениями** (англ. *comparison sort*).

Алгоритмы сортировок сравнениями выполняют только операции сравнения элементов и их перемещения (обмены), но никак не используют их внутреннюю структуру.

Для алгоритмов сортировок сравнениями известна нижняя оценка:

$$\Omega(n \cdot \log n),$$

т.е. нельзя выполнить внутреннюю сортировку n записей асимптотически быстрее, чем $n \cdot \log n$.

Рассмотрим в качестве примера алгоритма внутренней сортировки, который **не принадлежит классу алгоритмов сортировки сравнениями**, устойчивый алгоритм сортировки подсчётом (*англ. counting sort*).

В сортировке подсчётом предполагается, что все входные числа целые и принадлежат интервалу от 0 до k , где k — некоторая целая константа.

если все числа $0 \leq a_i \leq r$, то диапазон возможных значений $[0..r]$;

если все числа целые и неотрицательные и $0 \leq l \leq a_i \leq r$, то диапазон возможных значений $[0..r - l]$, а при работе с числом мы вычитаем из него число l ;

если все числа целые, среди них могут быть отрицательные и $l \leq a_i \leq r$, то диапазон возможных значений $[0..r + |l|]$, а при работе с числом мы добавляем к нему величину $|l|$.

↓

-3	3	5	-2	-0
0	6	8	1	3

[0..8]

$-3 \leq a_i \leq 5$
 $|l|=3$

Алгоритм сортировки подсчётом применяется в случае, когда сортируемые элементы можно отобразить в диапазон возможных значений, который достаточно мал, по сравнению с числом сортируемых элементов.
Если $k = O(n)$, то время работы алгоритма сортировки подсчётом — $O(n)$.

Предположим, что элементы массива $A = \{a_0, \dots, a_5\}$ лежат в диапазоне $[0..8]$:

	0	1	2	3	4
$a[i]$	0	6 ₁	6 ₂	8	1

на 1-м этапе

создадим массив $C [0..r]$, где
 $c[i]$ - количество элементов массива, равных i ;

	0	1	2	3	4	5	6	7	8
$c[i]$	1	1	0	0	0	0	2	0	1

```
for i = 0 to r
    c[i] = 0;
for i = 0 to l - 1
    c[a[i]] = c[a[i]] + 1;
```

на 2-м этапе

$c[i]$ — количество элементов
массива A , которые $\leq i$.

	0	1	2	3	4	5	6	7	8
$c[i]$	1	2	2	2	2	2	4	4	5

```
for j = 1 to r
    c[j] = c[j] + c[j - 1];
```

на 3-м этапе

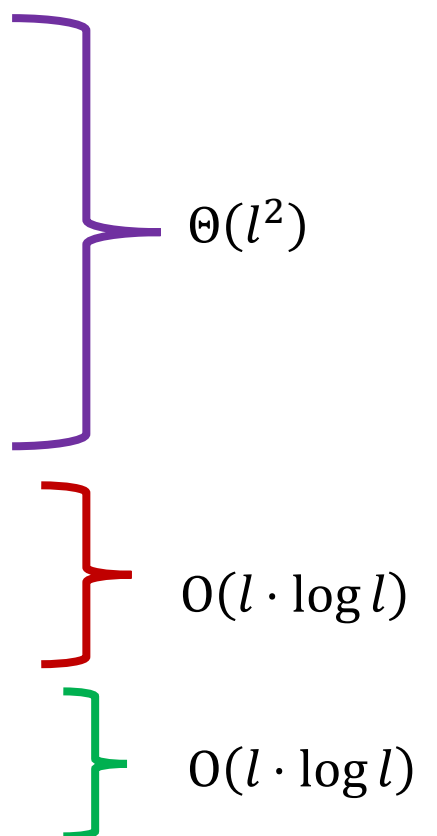
просматриваем массив A справа налево и заносим
элемент $a[i]$ в массив $A^{сорт}$ по индексу $c[a[i]] - 1$ и
уменьшаем значение на $c[a[i]]$ на 1.

	0	1	2	3	4
$a[i]$	0	6 ₁	6 ₂	8	1
$a^{сорт}[i]$	0	1	6 ₁	6 ₂	8

```
for i = n - 1 to 0
    {a^{сорт}[c[a[i]]-1] = a[i];
    c[a[i]] = c[a[i]] - 1 }
```

Время работы: $\Theta(n + r)$.
Память: $\Theta(n + r)$.

Алгоритмы внутренней сортировки сравнениями

1. Сортировка выбором (англ. *SelectionSort*)
 2. Обменные алгоритмы сортировки
 - Сортировка пузырьком (англ. *BubbleSort*)
 - Шейкерная сортировка (перемешиванием) (англ. *CocktailSort*)
 3. Сортировка вставками (включением) (англ. *InsertionSort*)
 4. Сортировка слиянием (англ. *MergeSort*)
 5. Быстрая сортировка Ч. Хоара (англ. *QuickSort*)
 6. Сортировка кучей (пирамидальная) (англ. *HeapSort*)
- 
- $\Theta(l^2)$
- $O(l \cdot \log l)$
- $O(l \cdot \log l)$

Для оценки времени работы алгоритмов внутренней сортировки сравнениями составим рекуррентное уравнение, решим его и оценим время работы алгоритма.

Сортировка выбором

	1	2	3	4	5	6	7
	2	3	7	14	70	0	1
1:	0	3	7	14	70	2	1
2:	0	1	7	14	70	2	3
3:	0	1	2	14	70	7	3
4:	0	1	2	3	70	7	14
5:	0	1	2	3	7	70	14
6:	0	1	2	3	7	14	70

На первой итерации среди n элементов массива найти элемент с минимальным ключом и поменять его с первым элементом. Теперь первый элемент стоит на своем месте.

Повторить описанные действия с оставшимися $n - 1$ элементом.

Процесс завершается через $n - 1$ итерацию.

Особенность: один обмен элементов массива в памяти компьютера на одну итерацию.

$$\begin{cases} T(n) = C_1 n + T(n - 1), n \geq 2 \\ T(1) = C_2 \end{cases}$$

Сортировка выбором

$$\begin{cases} T(n) = C_1 \cdot n + T(n-1), n \geq 2 \\ T(1) = C_2 \end{cases}$$

$$\begin{aligned} T(n) &= \underbrace{C_1 n + T(n-1)}_{1\text{-й шаг}} = [T(n-1) = C_1(n-1) + T(n-2)] = \\ &\underbrace{C_1 n + C_1(n-1) + T(n-2)}_{2\text{-й шаг}} = [T(n-2) = C_1(n-2) + T(n-3)] \\ &= \underbrace{C_1 n + C_1(n-1) + C_1(n-2) + T(n-3)}_{3\text{-й шаг}} = \dots \\ &\dots = \underbrace{C_1 n + C_1(n-1) + C_1(n-2) + \dots + C_1(n-m+1) + T(n-m)}_{m\text{-й шаг}} = \left[\begin{matrix} T(n-m) = T(1) \\ n-m = 1; m = n-1 \end{matrix} \right] = \\ &= \underbrace{C_1 n + C_1(n-1) + C_1(n-2) + \dots + C_1 \cdot 2 + T(1)}_{(n-1)\text{-й шаг}} = C_1 \cdot (2 + 3 + \dots + n) + C_2 \\ &= \frac{(n+2)}{2} \cdot (n-1) \cdot C_1 + C_2 \end{aligned}$$

$T(l) = O(l^2), l = \Theta(n)$ — алгоритм полиномиальный

Сортировка пузырьком

1	2	3	4	5	6	7	
2	3	7	14	70	0	1	←
0	2	3	7	14	70	1	←
0	1	2	3	7	14	70	←
0	1	2	3	7	14	70	←
0	1	2	3	7	14	70	←
0	1	2	3	7	14	70	←
0	1	2	3	7	14	70	

На первой итерации просматриваем массив справа налево и при каждом шаге меньший из двух соседних элементов перемещается к левой позиции (обменами).

Теперь первый элемент стоит на своем месте.

Повторить описанные действия с оставшимися $n - 1$ элементом.

Процесс завершается через $n - 1$ итерацию.

Особенность: на каждой итерации могут происходить многочисленные обмены элементов массива в памяти компьютера.

$$\begin{cases} T(n) = C_1 \cdot n + T(n - 1), n \geq 2 \\ T(1) = C_2 \end{cases}$$

$T(l) = O(l^2), l = \Theta(n)$ — алгоритм полиномиальный

Шейкерная сортировка

1	2	3	4	5	6	7
10	2	3	4	5	6	7
2	10	3	4	5	6	7
2	3	4	5	6	7	10
2	3	4	5	6	7	10

Отличия от пузырьковой сортировки:

1. Чередование направлений просмотра массива: при движении справа налево «всплывает самый лёгкий», при движении слева направо – «тонет самый тяжёлый».
2. Если при некотором проходе нет ни одного обмена, то сортировка досрочно завершается – массив отсортирован.
3. Сужение области просмотра: фиксируется индекс последнего обмена и при движении в противоположную сторону движение начинается с этого индекса.

$$\begin{cases} T(n) = C_1 n + C_2(n - 1) + T(n - 2), n \geq 2 \\ T(1) = C_3 \end{cases}$$

$T(l) = O(l^2), l = \Theta(n)$ – алгоритм полиномиальный

Сортировка вставками (включением)

1	2	3	4	5	6	7
2	3	1	14	7	0	4
2	3	1	14	7	0	4
1	2	3	14	7	0	4
1	2	3	14	7	0	4
1	2	3	7	14	0	4
0	1	2	3	7	14	4
0	1	2	3	4	7	14

Пусть элементы a_1, a_2, \dots, a_{i-1} уже упорядочены на предыдущих итерациях (первоначально в качестве упорядоченной части можно взять первый элемент массива).

На очередной итерации надо взять a_i (первый элемент из ещё неупорядоченной части) и включить в нужное место упорядоченной последовательности: a_1, a_2, \dots, a_{i-1} . В результате первые i элементов массива будут упорядочены. Данный процесс называют *просеиванием* (выполняется прямое или двоичное включение).

$$\begin{cases} T(n) = T(n-1) + C_1 \cdot n, n \geq 2 \\ T(1) = C_2 \end{cases}$$

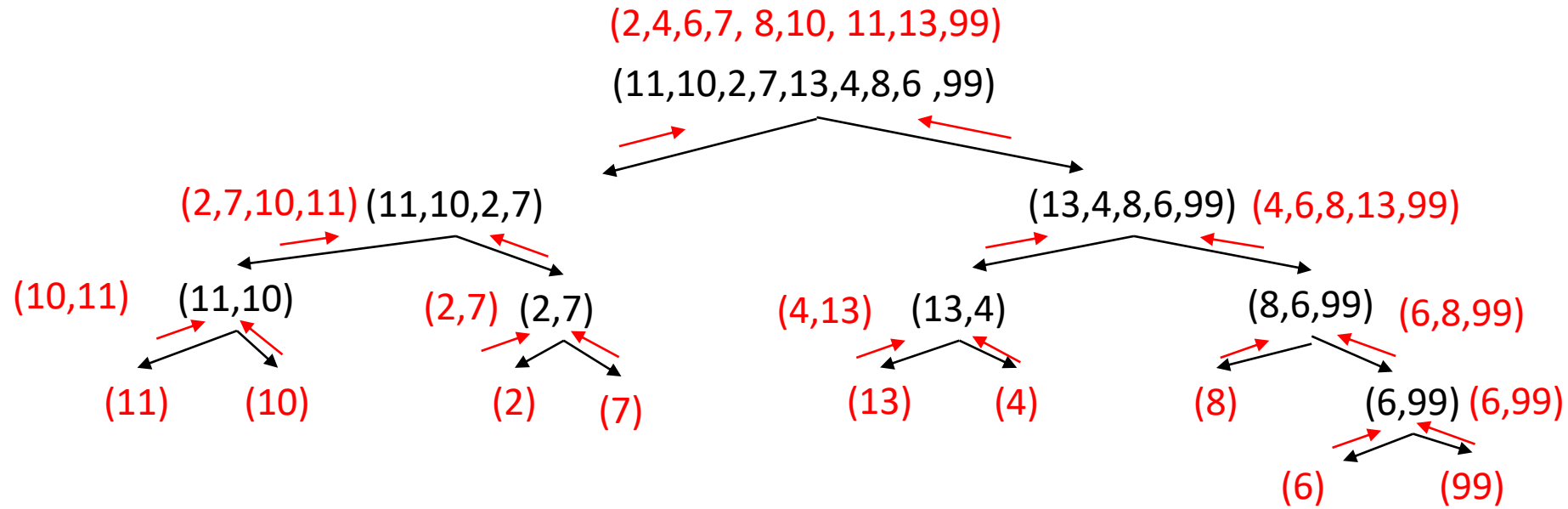
$T(l) = O(l^2), l = \Theta(n)$ — алгоритм полиномиальный.

Сортировка слиянием

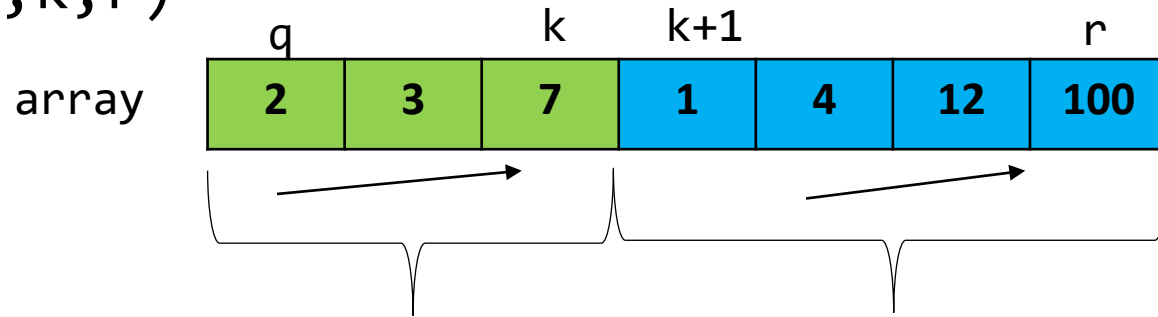
1. Делим последовательность элементов на две части (границы q и r включаем; если сортируемая последовательность состояла из n элементов, то первая часть может содержать $\lfloor n/2 \rfloor$ первых элементов, а вторая часть – оставшиеся; порядок следования элементов в каждой из полученных частей совпадает с их порядком следования в исходной последовательности). Если в последовательности только один элемент, то деление не выполняем.
2. Сортируем отдельно каждую из полученных частей этим же алгоритмом.
3. Производим слияние отсортированных частей последовательности так, чтобы сохранилась упорядоченность.

```
def MergeSort (q,r):  
    if q ≠ r:  
        k = (q + r) // 2  
        MergeSort (q,k)  
        MergeSort (k+1,r)  
        MergeList (q,k,r)
```

```
def MergeSort (q,r):
    if l ≠ r:
        k = (q + r) // 2
        MergeSort (q,k)
        MergeSort (k+1,r)
        MergeList (q,k,r)
```

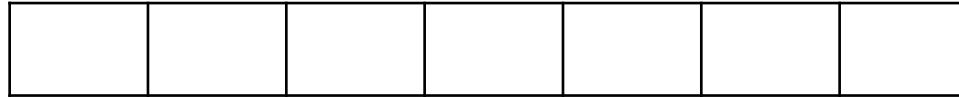


MergeList (q, k, r)



Вводим дополнительную память (список вывода), которая по размеру зависит от n .

additional memory

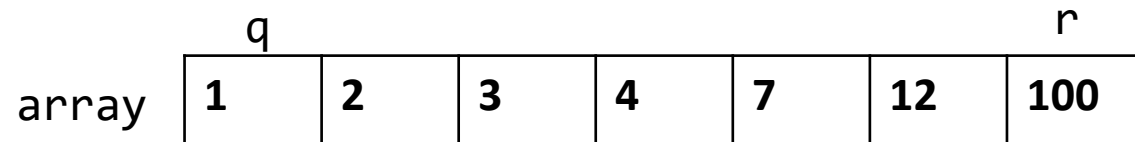


При слиянии двух упорядоченных частей, которые в исходном массиве занимают смежные области, сравниваем наименьшие элементы каждой из отсортированных частей и меньший из них отправляем в список вывода; повторяем описанные действия до тех пор, пока не исчерпается одна из частей; все оставшиеся элементы другой части пересылаем в список вывода.

additional memory

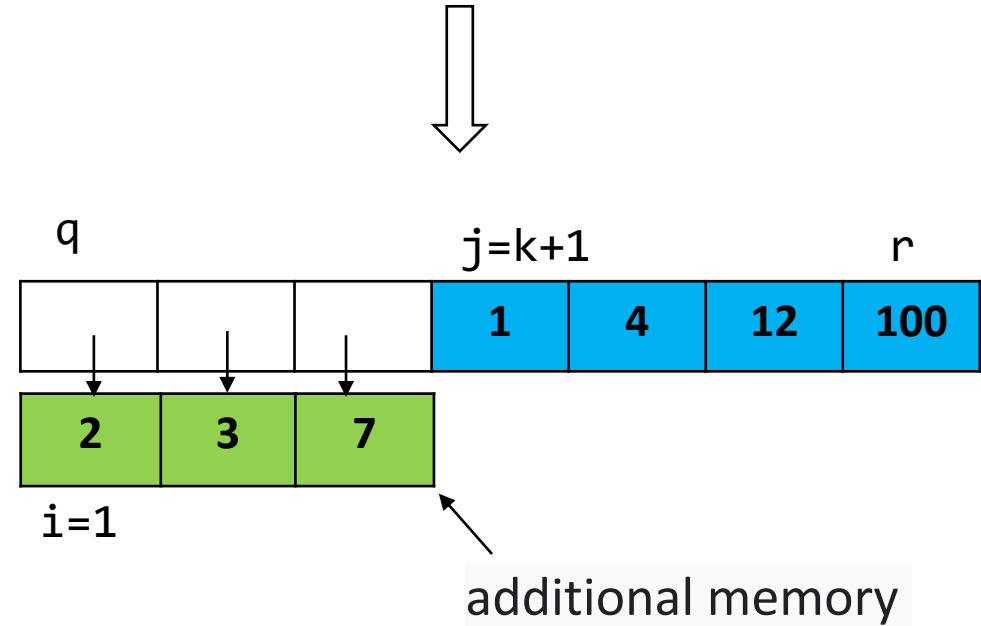
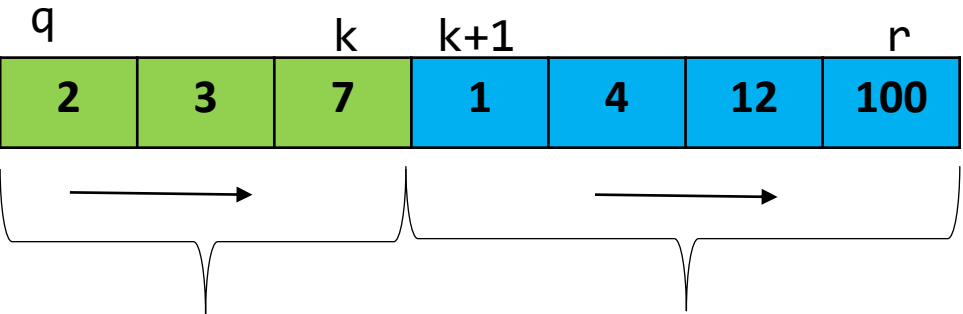


Затем из дополнительной памяти пересылаем элементы в исходный массив, начиная с индекса q и заканчивая r (т.е. на позиции, которые в массиве занимали элементы рассмотренных частей).



MergeList (q,k,r)

Как уменьшить дополнительную память, которая нужна при выполнении функции MergeList?



q				r		
1				4	12	100
1	2			4	12	100
1	2	3		4	12	100
1	2	3	4		12	100
1	2	3	4	7	12	100
1	2	3	4	7	12	100

2	3	7
	3	7
		7
		7

$$\begin{cases} T(n) = C_1 + 2 \cdot T\left(\frac{n}{2}\right) + C_2 \cdot n, n = 2^k, k \geq 1 \\ T(1) = C_3 \end{cases}$$

Решение:

$$\begin{aligned} T(n) &= \underbrace{C_1 + 2 \cdot T\left(\frac{n}{2}\right) + C_2 \cdot n}_{\text{1-й шаг}} = \left[T\left(\frac{n}{2}\right) = C_1 + 2 \cdot T\left(\frac{n}{2^2}\right) + C_2 \cdot \frac{n}{2} \right] = \\ &= \underbrace{2^0 \cdot C_1 + 2^1 \cdot C_1 + 2^2 \cdot T\left(\frac{n}{2^2}\right) + C_2 \cdot n + C_2 \cdot n}_{\text{2-й шаг}} = \dots = \\ &= \underbrace{C_1 \cdot (2^0 + 2^1 + \dots + 2^{m-1}) + 2^m \cdot T\left(\frac{n}{2^m}\right) + m \cdot C_2 \cdot n}_{\text{m-й шаг}} = \\ &= \left[\begin{array}{l} T\left(\frac{n}{2^m}\right) = T(1) \\ \frac{n}{2^m} = 1; \quad 2^m = n \quad m = \log_2 n \end{array} \right] = \underbrace{C_1 \cdot (n - 1) + n \cdot T(1) + C_2 \cdot n \cdot \log_2 n}_{(\log_2 n)\text{-й шаг}} = \\ &= C_2 \cdot n \cdot \log_2 n + C_3 \cdot n + C_1 \cdot (n - 1) = C_2 \cdot n \cdot \log_2 n + (C_3 + C_1) \cdot n - C_1 \end{aligned}$$

```
def MergeSort (l,r):
    if l != r:
        k = (l + r) // 2
        MergeSort (l,k)
        MergeSort (k+1,r)
        MergeList (l,k,r)
```

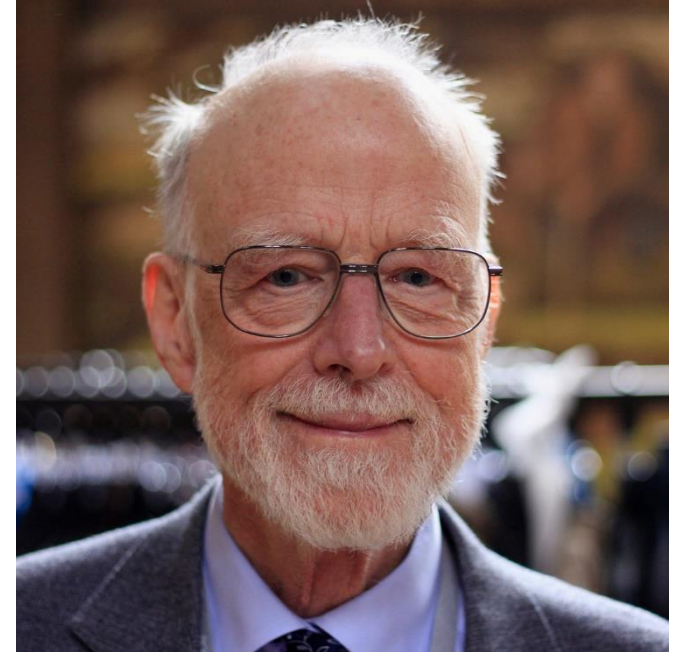
$T(l) = O(l \cdot \log l), l = \Theta(n)$ — алгоритм полиномиальный.

Быстрая сортировка Ч. Хоара (C.A.R. Hoare)

В **1960** году английский учёный Ч. Хоар разработал алгоритм «быстрой сортировки», который является наиболее популярным до настоящего времени.

Чарльз Энтони Ричард Хоар

Charles Antony Richard Hoare



Дата рождения: 11 января **1934** года

Страна: Великобритания

Научная сфера: Информатика

Награды: Премия Тьюринга, медаль
«Пионер компьютерной техники»

Известен как разработчик «быстрой
сортировки»

QuickSort (q, r)

Если $q \geq r$ то QuickSort (q, r) завершает работу.

Если $q < r$ то

```
def QuickSort( $q, r$ ):  
    if  $q < r$ :  
         $p = \text{Partition}(q, r)$   
        QuickSort( $q, p-1$ )  
        QuickSort( $p+1, r$ )
```

1. Выбирается разделитель (сепаратор, опорный элемент) (англ. pivot) – некоторый элемент x из рассматриваемой области. Например, в качестве сепаратора можно выбрать первый элемент области, т.е. $x = \text{array}[q]$.

2. Относительно сепаратора x массив разделим на три части (алгоритм Н. Ламута):

I часть - элементы строго меньше x (в array располагаются по индексам от q до $p-1$);

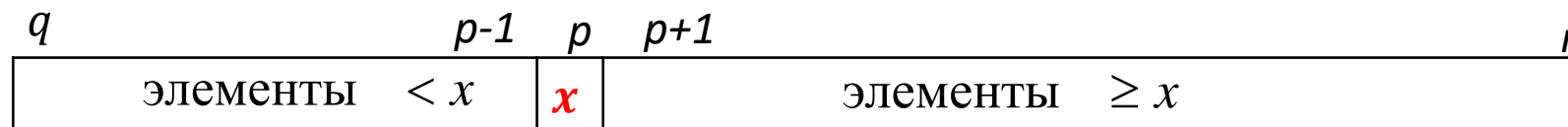
II часть - элемент x (в array располагается по индексу p);

III часть – элементы больше или равные x (в array располагаются по индексам от $p+1$ до r);

3. Рекурсивно вызываем алгоритм для первой и третьей части (если они не пустые):

QuickSort($q, p-1$)

QuickSort($p+1, r$)



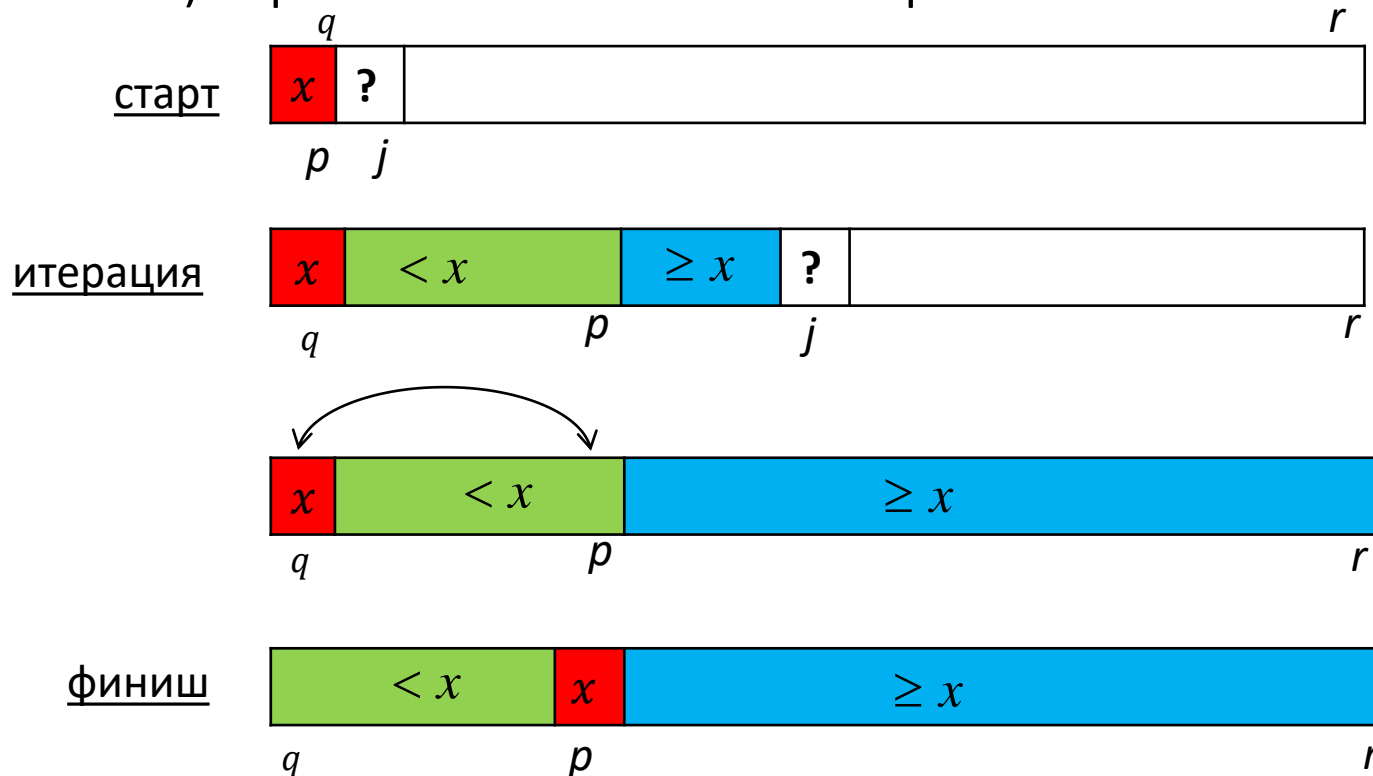
Разделение

(предложено Нико Ламуто)

В качестве сепаратора x выбираем первый элемент рассматриваемой области.

Относительно x массив разделим на три части функцией **Partition**:

- 1) в первой части окажутся все элементы, которые строго меньше x ;
- 2) во второй части - элемент x ;
- 3) в третьей части – больше или равные x .



```
def QuickSort(q, r):  
    if q < r:  
        p=Partition(q,r)  
        QuickSort(q, p-1)  
        QuickSort(p+1, r)
```

```
def Partition (q,r):  
    x=array[q]  
    p=q  
    j=p+1  
    while j<=r:  
        if array[j]>=x:  
            j+=1  
        else: # array[j]<x  
            p+=1  
            array[p]↔array[j]  
            j+=1  
    array[l]↔array[p]  
    return p
```

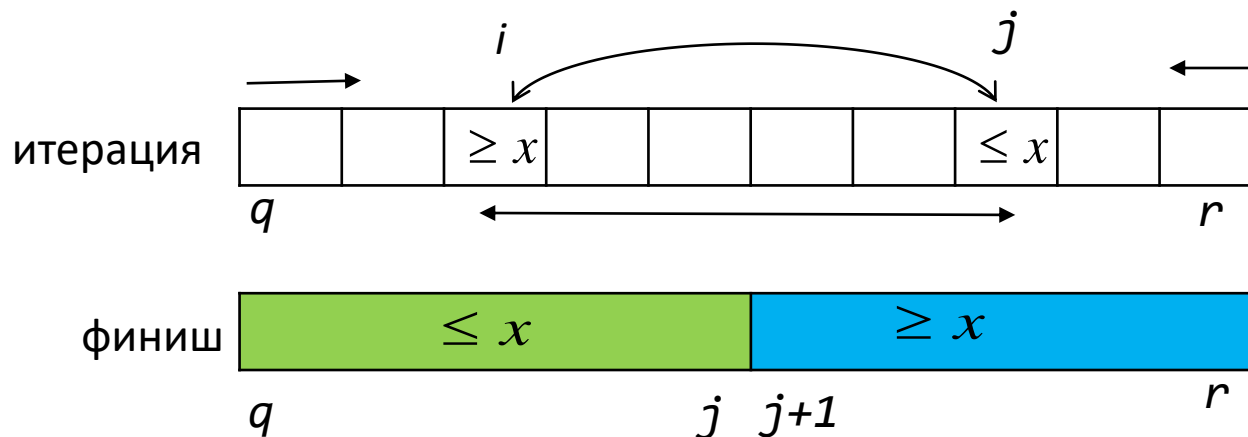
Разделение (предложено Чарльзом Хоаром)

В качестве сепаратора x будем выбирать первый элемент рассматриваемой области.

Относительно сепаратора x массив разделим на две части функцией **Hoare_Partition**:

(1) в первой части окажутся все элементы, которые равны или меньше x (зелёная заливка) ;

(2) во второй— элементы, которые равны или больше x (синяя заливка) .



Hoare_Partition (A, q, r)

$x \leftarrow a[q]$

$i \leftarrow q - 1$

$j \leftarrow r + 1$

while TRUE

do repeat $j \leftarrow j - 1$

until $a[j] \leq x$

repeat $i \leftarrow i + 1$

until $a[i] \geq x$

if $i < j$

then $a[i] \leftrightarrow a[j]$

else return j

В результате разделения будет сформирован индекс j , для которого справедливы неравенства:

$$q \leq j < p$$

и каждый элемент подмассива $A[q .. j]$ не превышает значений каждого элемента подмассива $A[j + 1 .. r]$.

Исходный алгоритм разделения, предложенный Ч. Хоаром

Hoare_Partition

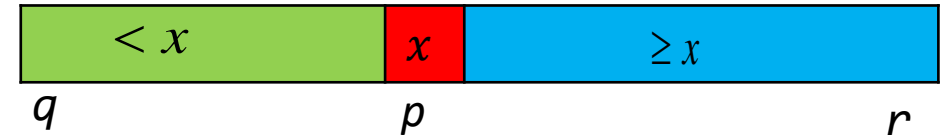


Худший случай:

например, все данные различны и упорядочены, например, по возрастанию. Тогда в качестве сепаратора на каждом этапе разделения будет выбираться минимальный элемент и сортируемая область сократится только на 1 элемент.

Версия алгоритма разбиения, предложенная Нико Ламуто

Partition



Худший случай:

например, все данные одинаковы или упорядочены, например, по возрастанию. Сортируемая область в каждом из случаев сократится только на 1 элемент.

Худший случай

$$\begin{cases} T(n) = C_1 + C_2 \cdot n + T(n-1), n \geq 2 \\ T(1) = C_3 \end{cases}$$

Время работы **QuickSort** в худшем случае:

$$T(l) = O(l^2), l = \Theta(n)$$

Среднее время работы алгоритма **QuickSort** по всем возможным наборам входных данных :

$$T(l) = O(l \cdot \log l), l = \Theta(n).$$

деление на классы идёт на каждом этапе разделения Partition, а класс характеризуется той позицией p , куда будет помещён сепаратор после того, как будет произведено разделение

Если на каждом этапе разделения в качестве сепаратора выбирать **средний по значению элемент (медиана)** и **делать это за линейное от количества элементов массива время**, то время работы алгоритма сортировки **QuickSort** в худшем случае:

$$\begin{cases} T(n) = C_1 \cdot n + C_2 \cdot n + 2T\left(\frac{n}{2}\right), n \geq 2^k, k \geq 2 \\ T(1) = C_3 \end{cases}$$

$$T(l) = O(l \cdot \log l), l = \Theta(n).$$

C++ `std::sort()`

Основой служит алгоритм быстрой сортировки – модифицированный **QuickSort**, он же IntroSort, разработанный специально для `std`. Отличие от QuickSort состоит в том, что количество рекурсивных операций не идет до самого конца, как в чистом QuickSort. Если количество итераций (процедур разделения массива) превысило $1.5 \cdot \log_2(n)$, где n - длина всего массива, то рекурсивные операции прекращаются:

- (1) если количество оставшихся элементов меньше 32-х, то оставшийся фрагмент сортируется методом вставки **InsertionSort** (сортировка вставками работает за время $O(n^2)$ и для больших массивов не используется, но на малых длинах эффективна ввиду простоты реализации);
- (2) если количество оставшихся элементов более 32-х элементов, то этот фрагмент сортируется пирамидальным методом **HeapSort** в чистом его виде (сортировка кучей в худшем случае работает за $O(n \log n)$, не устойчива).

Java `java.util.Collections.sort()`

Сортировка реализована на базе сортировки слиянием **MergeSort**, которая выбрана разработчиками из-за её устойчивости (показывает лучшую производительность по сравнению с другими устойчивыми алгоритмами сортировки, например, таким алгоритмом, как «пузырёк»).

Python `sort()` и `sorted()`

Функции в Python реализуют алгоритм **TimSort** (опубликован в 2002 году американским учёным Тимом Петерсом Tim Peters), основанный на сортировке слиянием **MergeSort** и сортировке вставкой **InsertionSort**. Основная идея алгоритма: по специальному алгоритму входной массив разделяется на подмассивы. Каждый подмассив сортируется сортировкой вставками. Отсортированные подмассивы собираются в единый массив с помощью модифицированной сортировки слиянием (<https://neerc.ifmo.ru/wiki/index.php?title=Timsort>)

Алгоритмы
нахождения k -го наименьшего
элемента

Определение


Элемент, который стоит на k —месте в отсортированном по не убыванию массиве, называется k —м наименьшим элементом (k —й порядковой статистикой).

Например, для массива

	1	2	3	4	5	6	8
$a[i]$	2	3	7	1	40	12	100

если $k = 4$, то k -й наименьший элемент равен 7:

	1	2	3	4	5	6	8
$a[i]^{\text{сорт.}}$	1	2	3	7	12	40	100



Определение

Если n – нечётно то **медианой** (англ. median) массива из n элементов называется такой его элемент, который стоит на месте $k = \left\lfloor \frac{n+1}{2} \right\rfloor$ в упорядоченном массиве.

	1	2	3	4	5
$a[i]$	2	3	7	1	40
$a[i]^{\text{сорт.}}$	1	2	3	7	40

Если n – чётно, то **нижней медианой** массива из n элементов называют называется такой его элемент, который стоит на месте $k = \left\lfloor \frac{n+1}{2} \right\rfloor$ в упорядоченном массиве.

Если n – чётно, то **верхней медианой** массива из n элементов называют называется такой его элемент, который стоит на месте $k = \left\lceil \frac{n+1}{2} \right\rceil$ в упорядоченном массиве.

	1	2	3	4	5	6
$a[i]$	2	3	7	1	40	12

	1	2	3	4	5	6
$a[i]^{\text{сорт.}}$	1	2	3	7	12	40
			↑ нижняя медиана	↑ верхняя медиана		

Если не оговорено иное, то медианой массива из n элементов будем считать нижнюю медиану, т.е. для любой чётности n полагаем $k = \left\lfloor \frac{n+1}{2} \right\rfloor$.

Алгоритм 1.

Отсортируем массив, например, сортировкой слиянием.

Возьмём в отсортированном массиве элемент по индексу k .

Время работы алгоритма 1 в худшем случае $\Omega(n \cdot \log n)$.

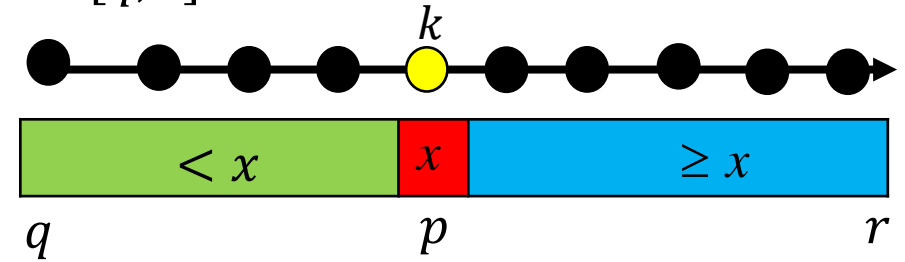
Алгоритм 2

Изначально $q = 1, r = n$, в качестве опорного элемента x возьмем первый элемент рассматриваемой области (в рандомизированной версии опорный элемент сначала выбирается случайным образом (random sampling) среди элементов с индексами от q до r , а затем он меняется с первым элементом рассматриваемой области).

Относительно x выполняется разделение массива на отрезке $[q, r]$

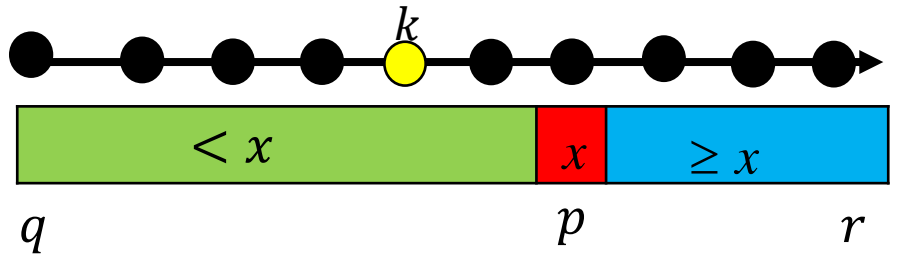
Случай 1.

Если $k = p - q + 1$, то опорный элемент x — ответ.



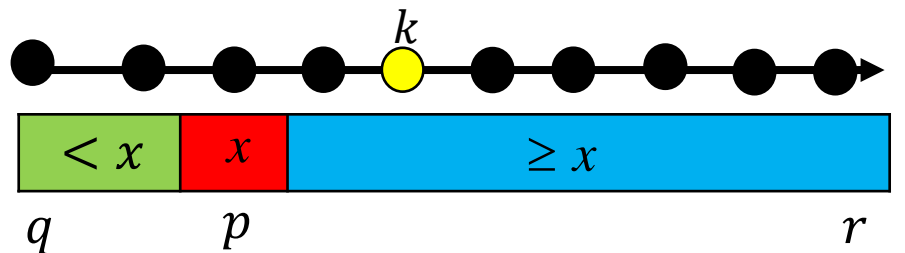
Случай 2.

Если $k < p - q + 1$, то продолжаем рекурсивно поиск k -го наименьшего элемента на отрезке $[q, p - 1]$.



Случай 3.

Если $k > p - q + 1$, то продолжаем рекурсивно поиск k -го наименьшего элемента на отрезке $[p + 1, r]$, полагая $k = k - (p - q + 1)$.



Время работы детерминированного алгоритма в худшем случае $\Omega(n^2)$ (например, на каждом шаге в качестве опорного элемента выбирался максимальный элемент рассматриваемой области, что приводило к тому, что каждый раз рассматриваемая область уменьшалась только на один элемент).

Рандомизированный алгоритм в среднем алгоритм работает хорошо за время $O(n)$.

Алгоритм 3. BFPRT- алгоритм (1973 г. Manual **B**lum, **R**obert W. **F**loyd, **V**aughan R. **P**ratt, **R**onald L. **R**ivest и **R**obert Endre **T**arjan)

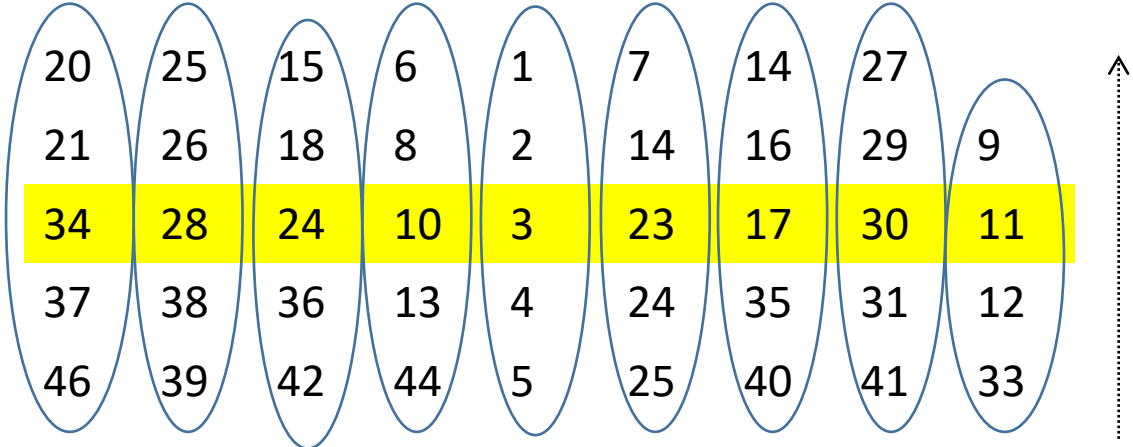
- 1. Разбиваем исходный массив A из n элементов на $\lfloor \frac{n}{5} \rfloor$ групп по 5 элементов в каждой и ещё одну группу, в которой оставшиеся $n \bmod 5$ элементов (если число элементов кратно 5, то эта группа – пустая). Каждую группу сортируем и находим её медиану. Это требует времени $C_1 \cdot n$.
- 2. Из найденных на шаге 1 медиан строим последовательность M длины $\lfloor \frac{n}{5} \rfloor$ (жёлтая заливка) и этим же алгоритмом рекурсивно находим её медиану x . Время – $T\left(\lfloor \frac{n}{5} \rfloor\right)$.
- 3. Элемент x выбираем в качестве опорного элемента и выполняем процесс деления исходного массива A из n элементов. Это требует времени $C_2 \cdot n$.

Для простоты рассуждений будем считать, что все элементы различны. Тогда количество элементов, которые $< x$ (зелёная область) :

$$\geq 3 \cdot \left(\left\lfloor \frac{1}{2} \cdot \left\lfloor \frac{n}{5} \right\rfloor \right\rfloor - 2 \right) \geq \frac{3n}{10} - 6,$$

аналогично, в синей области, имеется $\geq \frac{3n}{10} - 6$ элементов, величины которых $> x$.

- 4. По аналогии с алгоритмом 2, либо завершаем алгоритм, либо отбрасываем одну из частей и решаем рекурсивно задачу поиска искомого элемента за время, не превышающее величины $T\left(\frac{7 \cdot n}{10} + 6\right)$.



1	6		14	7	15	25	27	20
2	8	9	16	14	18	26	29	21
3	10	11	17	$x = 23$	24	28	30	34
4	13	12	35	24	36	38	31	37
5	44	33	40	25	42	39	41	46

$$T(n) \leq C_1 \cdot n + T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + C_2 \cdot n + T\left(\frac{7 \cdot n}{10} + 6\right)$$

Время работы алгоритма в худшем случае $O(n)$.

Алгоритм
лексикографической сортировки
(сортировка «вычёрпыванием»)

Определения

Задано некоторое конечное непустое множество символов Σ , называемое алфавитом ($|\Sigma|$ — мощность алфавита).

Строка — произвольная конечная последовательность символов из алфавита:

$$T = (t_0, t_1, \dots, t_{n-1}), t_i \in \Sigma.$$

Пусть Σ — множество букв латинского алфавита, тогда, например,

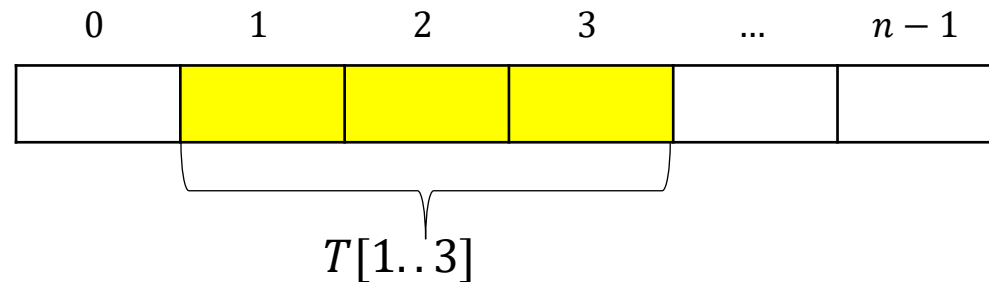
$$T = (a, b, a, c, a, b, a).$$

Латинский алфавит

A a	B b	C c	D d	E e	F f	G g
а	бэ	цэ	дэ	е/э	эф	гэ/жэ
H h	I i	J j	K k	L l	M m	N n
ха/аш	и	йот/жи	ка	эль	эм	эн
O o	P p	Q q	R r	S s	T t	U u
о	пэ	ку	эр	эс	тэ	у
V v	W w	X x	Y y	Z z		
вэ	дубль-вэ	икс	игрек/ ипсилон	зед		

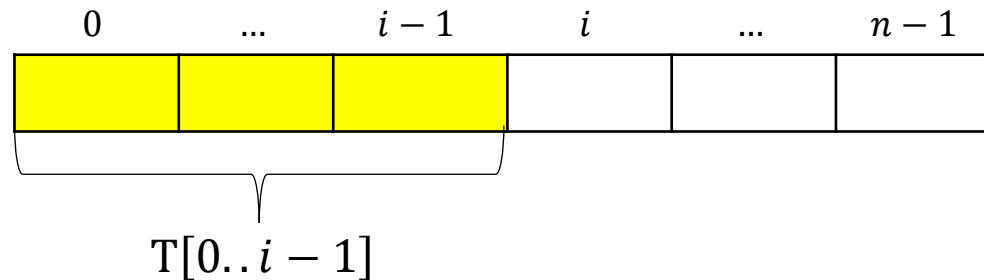
Подстрока – непрерывная последовательность строки

$$T[i..j] = (t_i, t_1, \dots, t_j), t_i \in \Sigma.$$

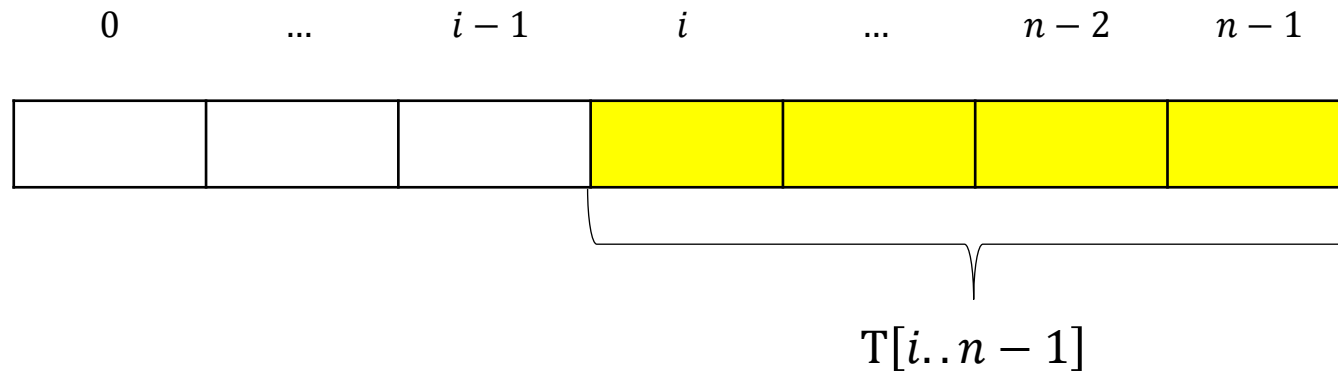


Если не оговорено иное, то при $i > j$ считаем, что подстроки $T[i..j]$ не существует.

Подстрока $T[0..i - 1]$, которая состоит из первых i символов строки, называется **префиксом** ($T[0..i - 1]$ - префикс длины i).



Подстрока $T[i..n - 1]$, которая состоит из последних $n - i$ символов строки, называется i -ым **суффиксом**.



Собственный суффикс/префикс - не совпадающий со всей строкой.

Пусть S – некоторое множество, на котором задан $<$ - линейный порядок.

Лексикографическим порядком на множестве S называют такое продолжение отношения $<$ (предшествования) на кортежи (списки) элементов из S , при котором $(s_1, s_2, \dots, s_p) < (t_1, t_2, \dots, t_q)$ означает выполнение одного из условий:

1) существует такое целое j , что $s_j < t_j$ и для всех $i < j$ справедливо $s_i = t_i$;

2) $p \leq q$ и $s_i = t_i$ при $1 \leq i \leq p$.

Предположим, что элементы кортежей заключены в интервале от 'a' до 'z', т.е. S – множество букв латинского алфавита Σ и

$$(a < b < c < \dots < z).$$

Тогда лексикографический порядок кортежей: $aaaa < aab < b < ba < baa < baaab < c$.

Предположим, что все **кортежи имеют одинаковую длину** k , число кортежей равно n , а индексы элементов кортежей изменяются от 0 до $k - 1$:

$$T^j = (t_0^j, t_1^j, \dots, t_{k-1}^j), j = 0, \dots, n - 1.$$

1. Создадим **очередь для сортировки**, куда добавим все рассматриваемые кортежи.
2. Организуем количество очередей («**черпаков**»), равное количеству букв в алфавите, предположим, что число «черпаков» — m .
3. Выполним **k итераций**:
 - на i -ой итерации идет сортировка по $(k - i)$ — ой компоненте каждого кортежа, т.е. некоторый кортеж t^j удаляется из исходной очереди для сортировки и добавляется в «черпак», который соответствует символу t_{k-i}^j ;
 - после того, как очередь для сортировки станет пустой, формируем новую очередь для сортировки, путём переписывания (удаления и добавления) элементов всех непустых «черпаков», начиная с «черпака», который соответствует символу 'a', и заканчивая - 'z'.

aab aaa baa bab bbb cab

1-я итерация

*aa**b** aa**a** ba**a** ba**b** bb**b** cab**b***

2-я итерация

*a**aa** b**aa** a**ab** b**ab** bb**b** cab**b***

3-я итерация

*a**aa** b**aa** a**ab** b**ab** cab**b** bb**b***

Итог:

aaa < aab < baa < bab < bbb < cab

a	aa a	ba a		
b	aa b	ba b	bb b	ca b
c	-			


a	a aa	b aa	a ab	b ab	ca b
b	b bb				
c	-				

a	a aa	a ab			
b	b aa	b ab	b bb		
c	c ab				

Время работы алгоритма лексикографической сортировки
кортежей одинаковой длины:

сумма длин всех
кортежей

склеивание
очередей


$$\Theta(k \cdot n + k \cdot |\Sigma|) = \Theta(k \cdot (n + |\Sigma|)), \text{ где}$$

n — число кортежей одинаковой длины k ,
 $|\Sigma|$ — число различных символов в кортежах.

Предположим, что **кортежи имеют разную длину**, число кортежей — n , а индексы элементов j -го кортежа изменяются от 0 до $|T^j| - 1$:

$$T^j = (t_0^j, t_1^j, \dots, t_{|T^j|-1}^j), j = 0, \dots, n - 1.$$

Пусть l_{max} — длина самого длинного кортежа:

$$l_{max} = \max_{0 \leq j \leq n-1} |T^j|,$$

тогда число итераций алгоритма равно l_{max} .

На первой итерации в очередь для сортировки помещаются кортежи длины l_{max} и выполняется сортировка вычерпыванием только по компоненте $l_{max} - 1$ рассматриваемых кортежей.

После этого в исходную очередь для сортировки заносятся сначала кортежи длины $l_{max} - 2$, а затем добавляются элементы непустых сгенерированных «черпаков», начиная с элементов «черпака» который соответствует символу 'а', и заканчивая — 'z'.

На последующих этапах происходит сортировка по компоненте $l_{max} - 2, l_{max} - 3, \dots, 0$ аналогичным образом.

aab ba baa bab ab cab c

1-я итерация

*aa**b** baa ba**b** ca**b***
 ↑ ↑ ↑ ↑

2-я итерация

*b**a** ab baa a**a**b b**a**b cab*
 ↑ ↑ ↑ ↑ ↑ ↑

3-я итерация

***c** ba ba**a** a**a**b ba**b** **c**ab **a**b*
 ↑ ↑ ↑ ↑ ↑ ↑ ↑

<i>a</i>	<i>baa</i>			
<i>b</i>	<i>aab</i>	<i>bab</i>	<i>cab</i>	
<i>c</i>				

<i>a</i>	<i>ba</i>	<i>baa</i>	<i>aab</i>	<i>bab</i>	<i>cab</i>
<i>b</i>	<i>ab</i>				
<i>c</i>					

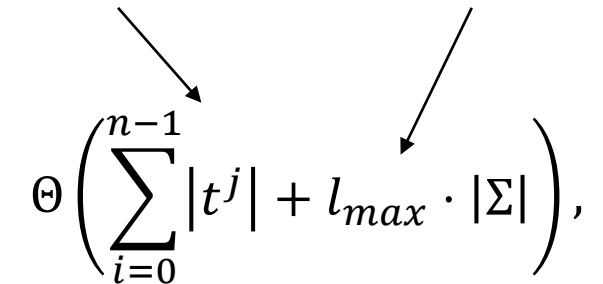
<i>a</i>	<i>aab</i>	<i>ab</i>			
<i>b</i>	<i>ba</i>	<i>baa</i>	<i>bab</i>		
<i>c</i>	<i>c</i>	<i>cab</i>			

aab < ab < ba < baa < bab < c < cab

Время работы алгоритма лексикографической сортировки кортежей разной длины:

сумма длин всех
кортежей

склеивание
очередей


$$\Theta \left(\sum_{i=0}^{n-1} |t^j| + l_{max} \cdot |\Sigma| \right),$$

где

n — число кортежей,

l_{max} — длина самого длинного кортежа,

$|\Sigma|$ — число различных символов в кортежах.

Для сортировки n кортежей можно использовать, например, сортировку слиянием (***merge sort***).

Время работы алгоритма сортировки будет зависеть от того, как будут сравниваться два кортежа.

Пусть l_{max} — длина самого длинного кортежа.

Тогда непосредственное сравнение двух кортежей приведет к тому, что время работы алгоритма сортировки:

$$O(l_{max} \cdot n \cdot \log n).$$

Сортировка
«вычёрпыванием»:

$$\Theta \left(\sum_{i=0}^{n-1} |t^j| + l_{max} \cdot |\Sigma| \right),$$

где

n — число кортежей,

l_{max} — длина самого длинного кортежа,

$|\Sigma|$ — число различных символов в кортежах.

Внешняя сортировка

Предположим, что объем входных такой, что все данные не могут одновременно поместиться в основную (оперативную) память машины.

Сортировка данных, хранящихся во вторичной памяти, называется **внешней сортировкой**.

В различных языках программирования предусмотрен **файловый тип данных**, предназначенный для представления **данных, хранящихся во вторичной памяти**.

Операционная система делит вторичную память на блоки одинакового размера, а файл можно рассматривать, как связанный список блоков (размер блока зависит от ОС и обычно находится в пределах от 512 до 4 096 байт).

Базовая операция для файла – перенос одного блока в буфер, находящийся в основной памяти.

Буфер – зарезервированная область **в основной памяти, размер которой соответствует размеру блока** (может резервироваться память под несколько буферов – *буферный пул*).

Блоки считываются в том порядке, в котором они появляются в списке блоков: считывается в буфер первый блок, затем он заменяется на второй блок (при этом предыдущее содержимое буфера теряется) и т.д.

Процесс записи файла также можно рассматривать, как процесс создания файла в буфере: когда записи «заносятся» в файл, фактически они помещаются в буфер для этого файла – непосредственно за записями, которые находятся там. Если очередная запись не помещается в буфер целиком, то содержимое буфера копируется в свободный блок вторичной памяти, который присоединяется к списку блоков для данного файла.

Природа устройств вторичной памяти такова, что время, необходимое для поиска блока и считывания его в основную память, достаточно велико в сравнении со временем, которое требуется для обработки данных, содержащихся в этом блоке.

Под «*бездействием*» компьютера будем понимать периоды ожидания, пока блок будет прочитан в основную память или записан из основной памяти во внешнюю.

Поэтому, оценивая время работы алгоритмов, в которых используются данные, хранящиеся в виде файлов, в первую очередь надо учитывать количество обращений к блокам, т.е. сколько раз мы считываем в основную память или записываем блок во вторичную память. При этом размер блока в ОС фиксирован и мы не можем его изменить для ускорения алгоритма.

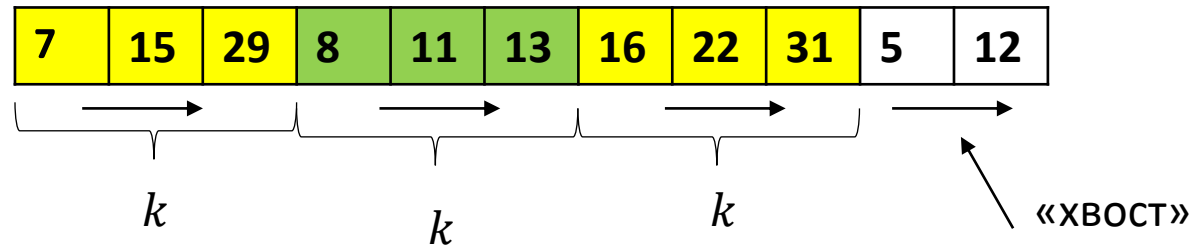
Мера качества алгоритма, работающего с внешней памятью – количество обращений к блокам памяти.

Алгоритм внешней сортировки слиянием (двухпутёвое слияние)

Предположим, что нужно отсортировать n записей .

Файл организуется в виде постепенно увеличивающихся серий, т.е. последовательностей записей r_1, r_2, \dots, r_k , таких, что $r_i \leq r_{i+1}, 1 \leq i < k$.

В примере последовательность целых чисел организована в виде серий длины $k = 3$.

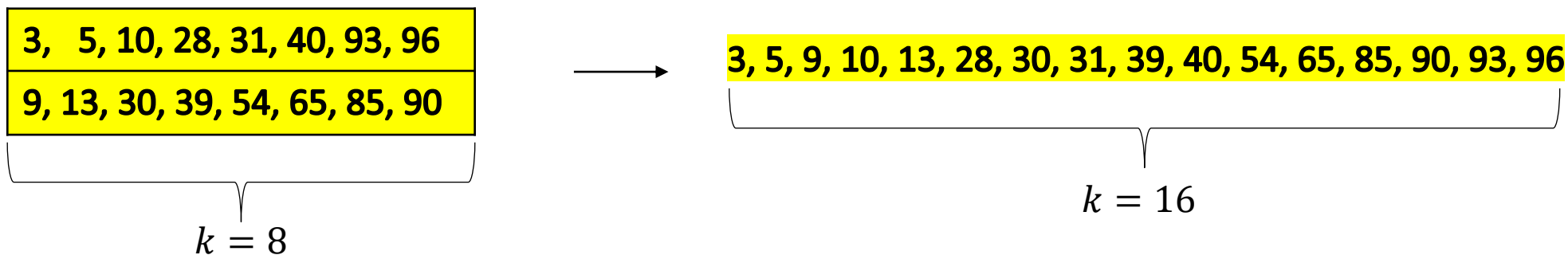


Предположим, что заданы два файла f_1 и f_2 , которые организованы в виде серий длины k и для которых выполняются следующие условия:

- 1) количество «серий», включая «хвосты» в f_1 и f_2 отличаются не больше, чем на 1;
- 2) только один из файлов f_1 или f_2 может иметь «хвост»;
- 3) файл с «хвостом» имеет не меньше серий, чем другой файл.

Первоначально можно разделить все n записей, которые надо отсортировать, на два исходных файла f_1 и f_2 (желательно, чтобы записей в этих файлах было поровну), считаем, что каждый файл состоит из серий длины $k = 1$.

В последующем, когда будем говорить про объединение серий длины k , то при этом предполагается, что выполняется слияние двух серий длины k в серию длины $2 \cdot k$ (т.е. из двух упорядоченных последовательностей длины k получаем упорядоченную последовательность длины $2 \cdot k$).



АЛГОРИТМ

$i = 0$ число фаз (итераций) алгоритма

$k = 1$ длина серии

f_1 и f_2 исходные файлы

g_1 и g_2 результирующие файлы пустые

пока $2^i < n$ выполнять следующие действия:

- ✓ пока не закончится один из исходных файлов, считываем из f_1 и f_2 серии длины k , объединяем их в серию длины $2 \cdot k$ и записываем её в один из результирующих файлов g_1 или g_2 (переключаясь последовательно между результирующими файлами, можно добиться того, что они будут удовлетворять условиям (1-3);
- ✓ если оба файла f_1 и f_2 одновременно стали пустыми, то ничего не делать, иначе переписать «хвост» у оставшегося непустого файла в тот результирующий файл, куда бы шла запись очередной серии длины $2 \cdot k$;
- ✓ $k = 2 \cdot k$;
- ✓ $i = i + 1$;
- ✓ в качестве исходных рассматриваем результирующие файлы, а в качестве результирующих – исходные.

Если $n = 2^k$, то после k -й фазы один из результирующих файлов будет пустым, а второй будет содержать единственную серию длины n , т.е. будет отсортирован.

Если $n < 2^k$, то после k -й фазы один из результирующих файлов будет пустым, а второй будет содержать «хвост» длины n , т.е. будет отсортирован.

$$n = 16 (=2^4)$$

начальная фаза
 $k = 2^0$ (длина серии)

f_1	28	3	93	10	54	65	30	90
f_2	31	5	96	40	85	9	39	13

первая фаза
 $k = 2^1$

g_1	28, 31	93, 96	54, 85	30, 39
g_2	3, 5	10, 40	9, 65	13, 90

вторая фаза
 $k = 2^2$

f_1	3, 5, 28, 31	9, 54, 65, 85
f_2	10, 40, 93, 96	13, 30, 39, 90

третья фаза
 $k = 2^3$

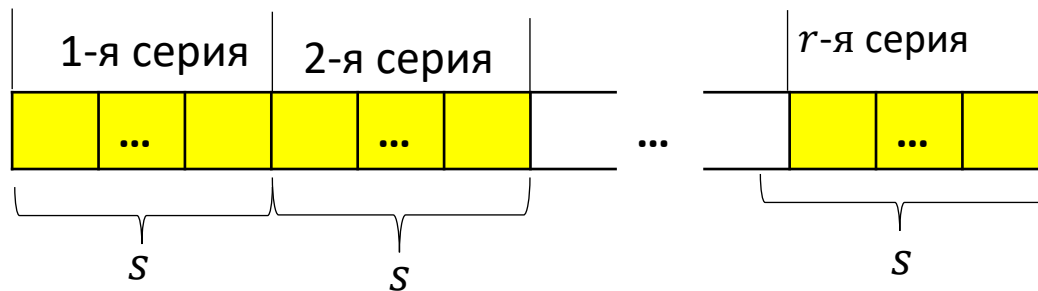
g_1	3, 5, 10, 28, 31, 40, 93, 96
g_2	9, 13, 30, 39, 54, 65, 85, 90

четвёртая фаза
 $k = 2^4$

f_1	3, 5, 9, 10, 13, 28, 30, 31, 39, 40, 54, 65, 85, 90, 93, 96
f_2	

Для ускорения можно **начинать работу не с серий длины 1, а с серий длины $s > 1$.**

Серии длины s можно сформировать на начальном этапе: считывать в оперативную память сразу по s элементов (некоторое разумное число элементов, которое можно одновременно хранить в памяти), сортировать их алгоритмом внутренней сортировки за $O(s \cdot \log s)$, поочерёдно сохранять серии длины s в файлы f_1 и f_2 .



Так как общее число серий на начальном этапе: $r = \lceil n/s \rceil$, то время выполнения этапа:

$$O(r \cdot s \cdot \log s) = \mathbf{O(n \cdot \log s)}.$$

$$\begin{aligned} r = \lceil n/s \rceil &< n/s + 1, s \cdot r < n + s < 2 \cdot n \\ r = \lceil n/s \rceil &\geq n/s, s \cdot r \geq n \end{aligned}$$

Количество фаз алгоритма внешней сортировки слиянием $-\lceil \log_2 r \rceil$, так как после каждой фазы число серий уменьшается в двое (изначально r серий).

Подсчитаем общее число сравнений алгоритма, учитывая, что при объединении двух серий длины k выполняется $(2 \cdot k - 1)$ сравнение:

на первой фазе объединялись серии длины s , число объединений не более, чем $r/2^1$, поэтому надо выполнить не более, чем

$$\frac{r}{2^1} \cdot (2^1 \cdot s - 1) \text{ сравнений}$$

на второй фазе объединялись серии длины $2^1 \cdot s$, число объединений не более, чем $r/2^2$, поэтому надо выполнить не более, чем

$$\frac{r}{2^2} \cdot (2^2 \cdot s - 1) \text{ сравнений}$$

и так далее

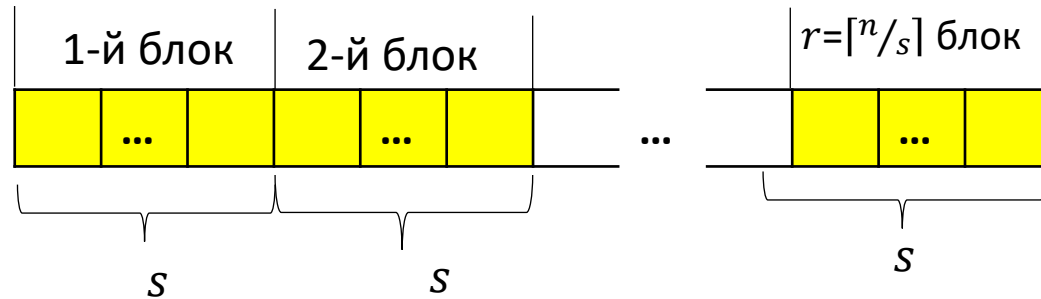
$$\sum_{i=1}^{\lceil \log_2 r \rceil} \left(r \cdot s - \frac{r}{2^i} \right) = r \cdot s \cdot \lceil \log_2 r \rceil - r \cdot \sum_{i=1}^{\lceil \log_2 r \rceil} \frac{1}{2^i} \approx n \cdot \log_2 r - r.$$

Тогда время алгоритма в целом, включая начальный этап:

$$O(n \cdot \log s) + O(n \cdot \log r) = O(n \cdot \log(s \cdot r)) = \mathbf{O(n \cdot \log n)}.$$

Мера качества алгоритма, работающего с внешней памятью – количество обращений к блокам памяти, поэтому подсчитаем **общее число чтения блоков**.

Предположим, что максимальное число элементов, которые могут одновременно храниться в оперативной памяти, равно s и на начальном этапе мы прочитали r таких блоков.



На каждой фазе при объединении двух серий мы могли считывать из каждого файла только блоки размера $s/2$, а так как считать нужно все данные, то общее число считываний блоков на одной фазе:

$$\frac{n}{s/2} = 2 \cdot r.$$

Так как количество фаз $-\lceil \log_2 r \rceil$, то общее число чтения блоков:

$$2 \cdot r \cdot \lceil \log_2 r \rceil = O(r \cdot \log r).$$

Минимизация полного времени

Предположим, что файлы организованы в виде серий размер которых намного превышает размер блока (буфера оперативной памяти), поэтому, чтобы объединить две такие серии, надо прочитать несколько блоков из каждого файла.

Каков порядок считывания блоков из файлов?

Порядок считывания блоков из файлов можно организовать так: определить, у какой из двух серий будут первой выбраны все её записи, находящиеся в данный момент в основной памяти (для каждого файла будем хранить ключ последней записи последнего блока, считанного из файла), и пополнять запас записей именно для этой серии.

Если какая-то серия себя исчерпала, то очередной блок считывается из не исчерпавшей себя серии.

Наличие одного канала по обмену данными между основной памятью и внешними устройствами – «узкое место», которое будет тормозить работу системы в целом.

Увеличим число каналов связи?

Предположим, что существует $2 \cdot m$ дисководов, каждый из которых имеет свой канал доступа к основной памяти:

$$f_1, f_2, f_3, \dots, f_m$$

$$g_1, g_2, g_3, \dots, g_m$$

Многоканальная сортировка

Для выполнения $2 \cdot t$ – канальной сортировки разместим на t дисководах t файлов, организованных в виде серий длины k :

$$f_1, f_2, f_3, \dots, f_m.$$

Тогда можно прочитать t серий (по одной из каждого файла) и объединить их в одну серию длиной $t \cdot k$ (для объединения серий можно использовать такие структуры данных, как «бинарная куча» или поисковые деревья и выполнить объединение за время $O(\log t)$ на одну запись).

Затем серия помещается в один из выходных файлов:

$$g_1, g_2, g_3, \dots, g_m.$$

Если у нас имеется первоначально n записей, а после каждой фазы длина серии увеличивается в t раз, то после i -ой фазы серии будут иметь длину t^i .

Предположим, что

$$t^{i-1} < n \leq t^i,$$

тогда

$$\log_t n \leq i < \log_t n + 1,$$

следовательно, после $\lceil \log_t n \rceil$ фаз все записи будут отсортированы.

Выполнить общую задачу в iRunner

Структуры данных

[0.1. Бинарный поиск \(уметь см. реализовать BinarySearch, LowerBound, UpperBound\)](#)



[Реализация сортировок в C++ и Python](#) (подготовлено студентами 2 курса ПИ, 2020 г.)





Спасибо за внимание!