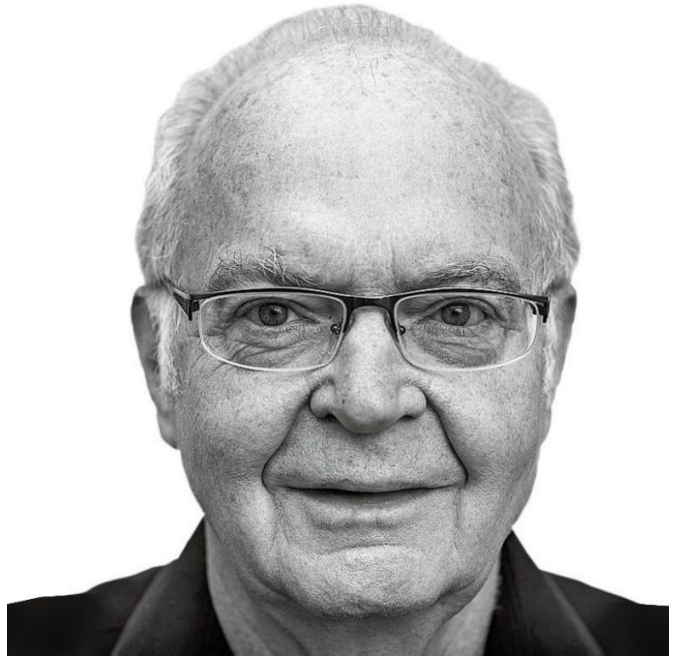
Лекция 3. Итеративные сортировки

Горденко Мария Константиновна

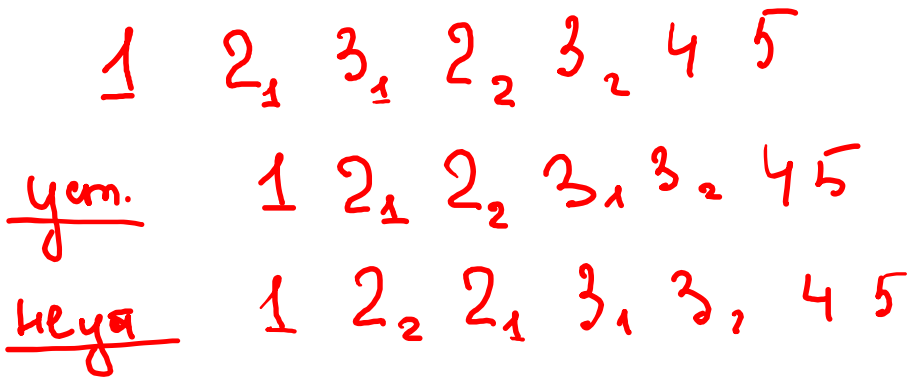
Задача сортировки

- Сортировка – это упорядочивание набора однотипных данных по возрастанию или убыванию.



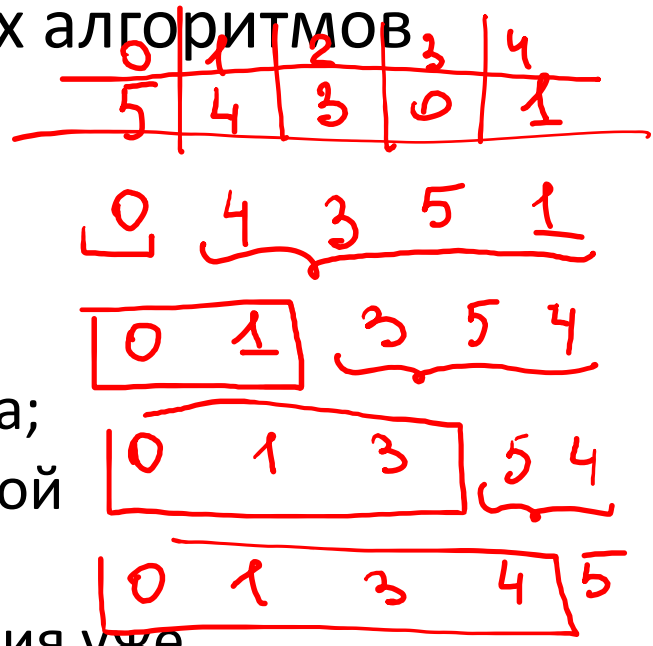
Критерии оценки алгоритмов сортировки

- Время сортировки
- Требуемая память
- Устойчивость



Сортировка выбором (Selection sort)

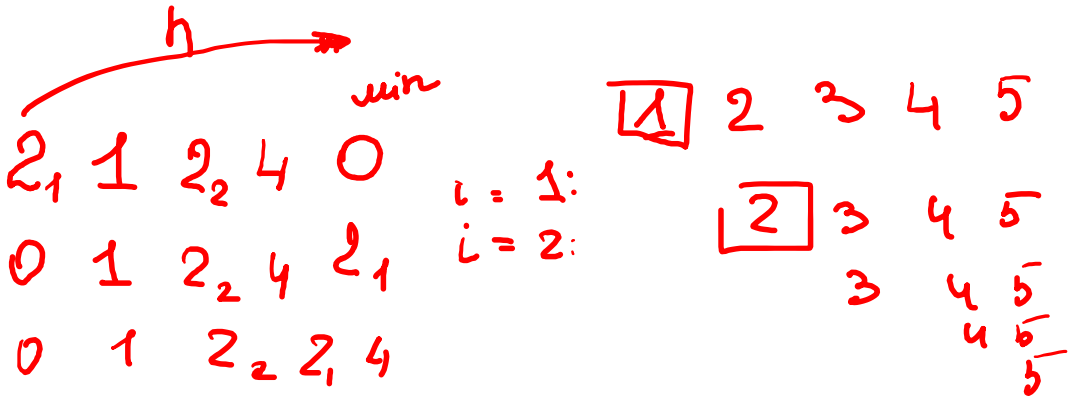
- Сортировка выбором является одним из простейших алгоритмов сортировки
- Может быть как устойчивой, так и не устойчивой
- Шаги алгоритма:
  - находим минимальное значение в текущей части массива;
    - производим обмен этого значения со значением на первой неотсортированной позиции;
    - далее сортируем хвост массива, исключив из рассмотрения уже отсортированные элементы



Сортировка выбором (Selection sort)

```
1. function selectionSort(T[n] a):
2.     for i = 1 to n - 1
3.         min = i
4.         for j = i + 1 to n
5.             if a[j] < a[min]
6.                 min = j
7.         swap(a[i], a[min])
```

$n + (n-1) + (n-2) \dots + 1 = \left(\frac{1+n}{2}\right) \cdot n \approx O(n^2)$



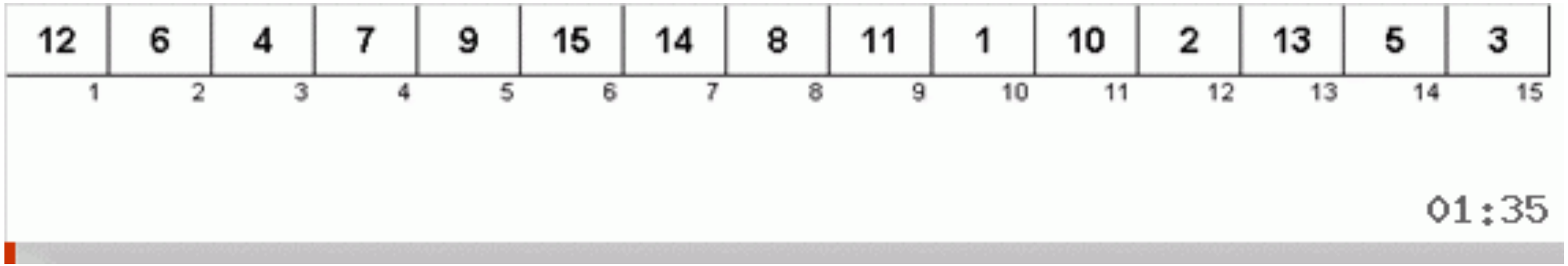
Сортировка выбором (Selection sort)

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Дополнительная память - $O(1)$

Устойчивость

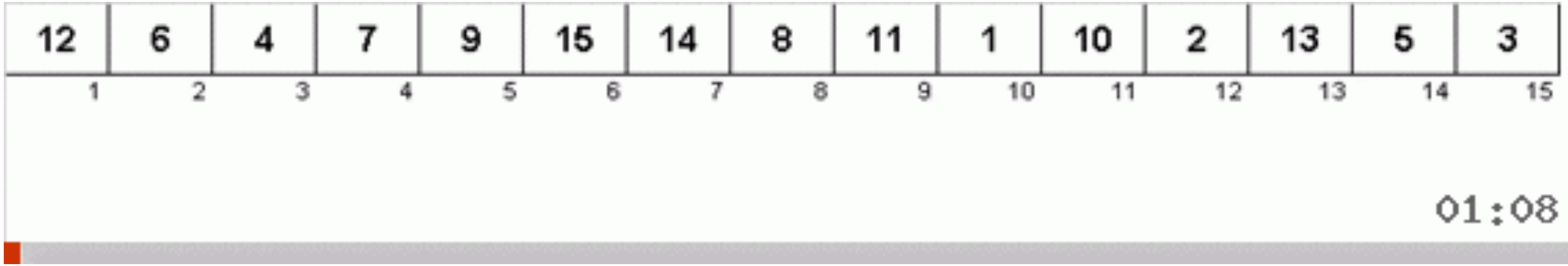
- Устойчива или нет?

Пример



https://habr.com/ru/post/422085/

Двухсторонняя сортировка выбором (Double selection sort)



- За один проход выбираем и минимум, и максимум

for $i = 1 \dots \frac{n}{2}$

for $j = i \dots n-i$

max

min

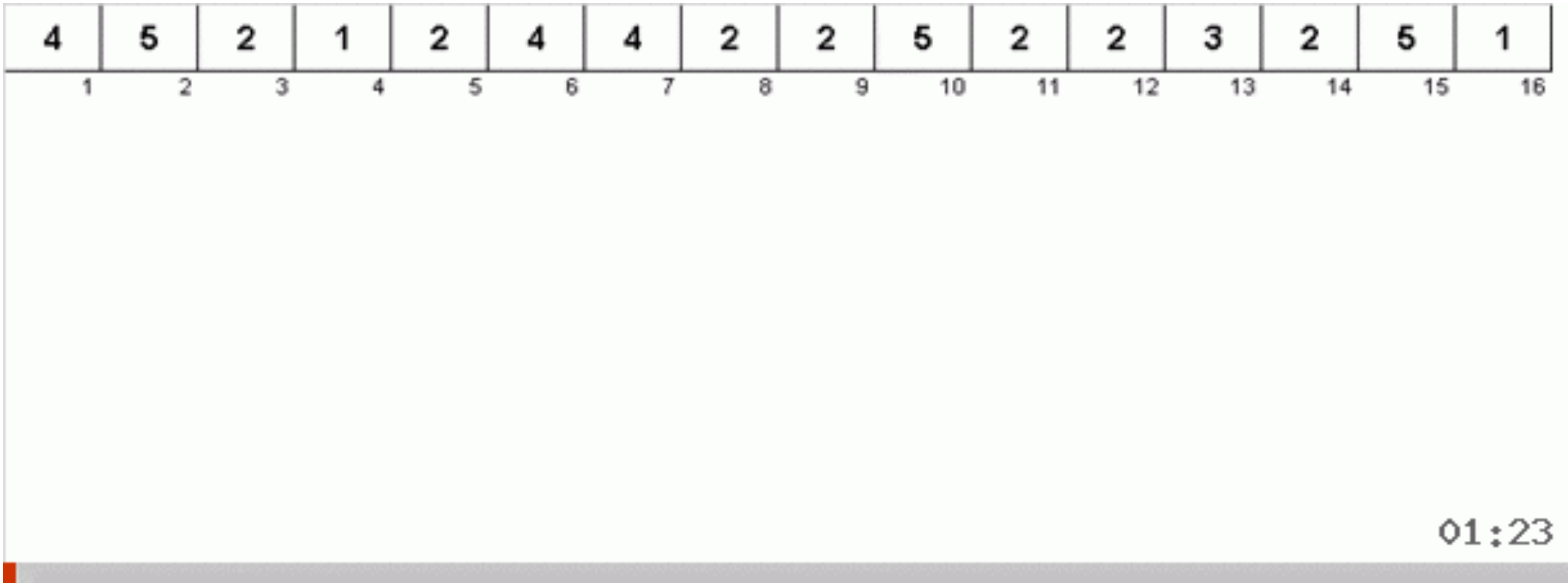
swap

swap

$n^{\frac{2}{2}}$

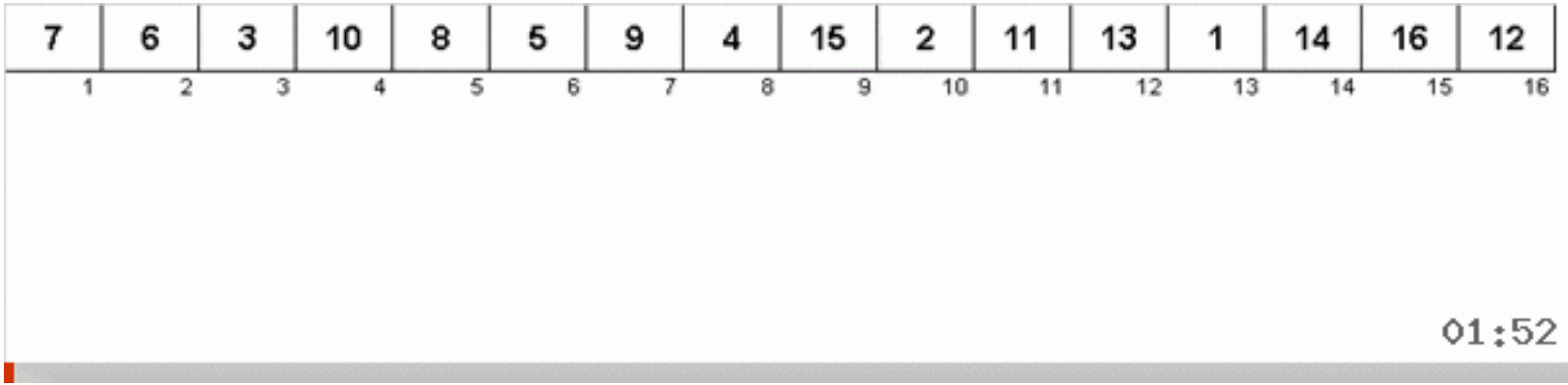
https://habr.com/ru/post/422085/

Бинго-сортировка (Bingo sort)



https://habr.com/ru/post/422085/

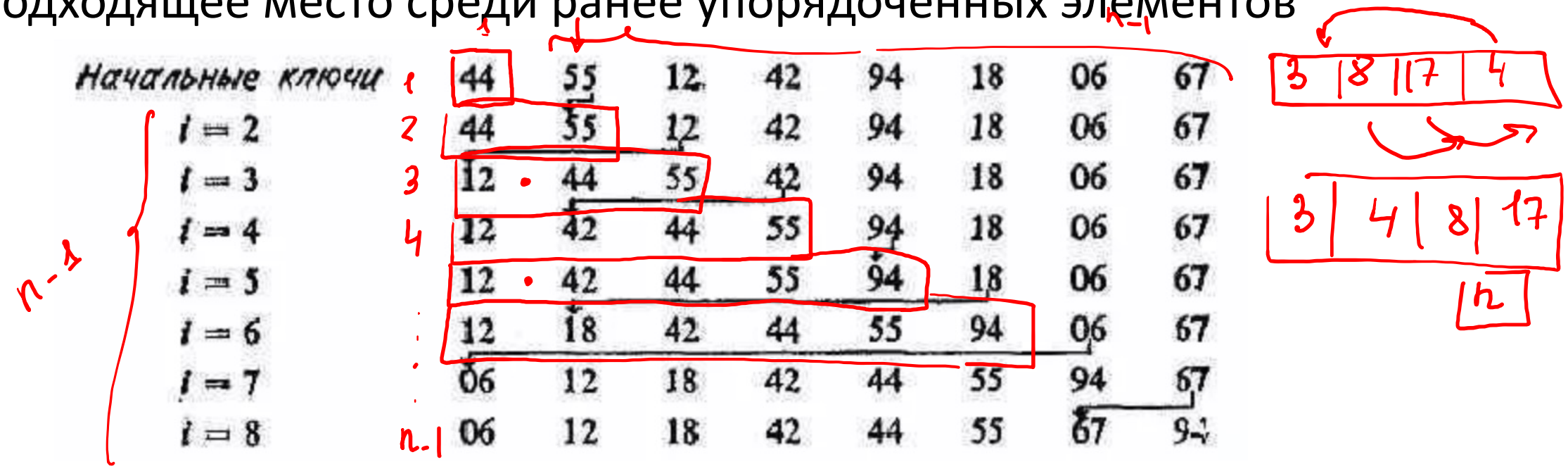
Цикличная сортировка (Cycle sort)



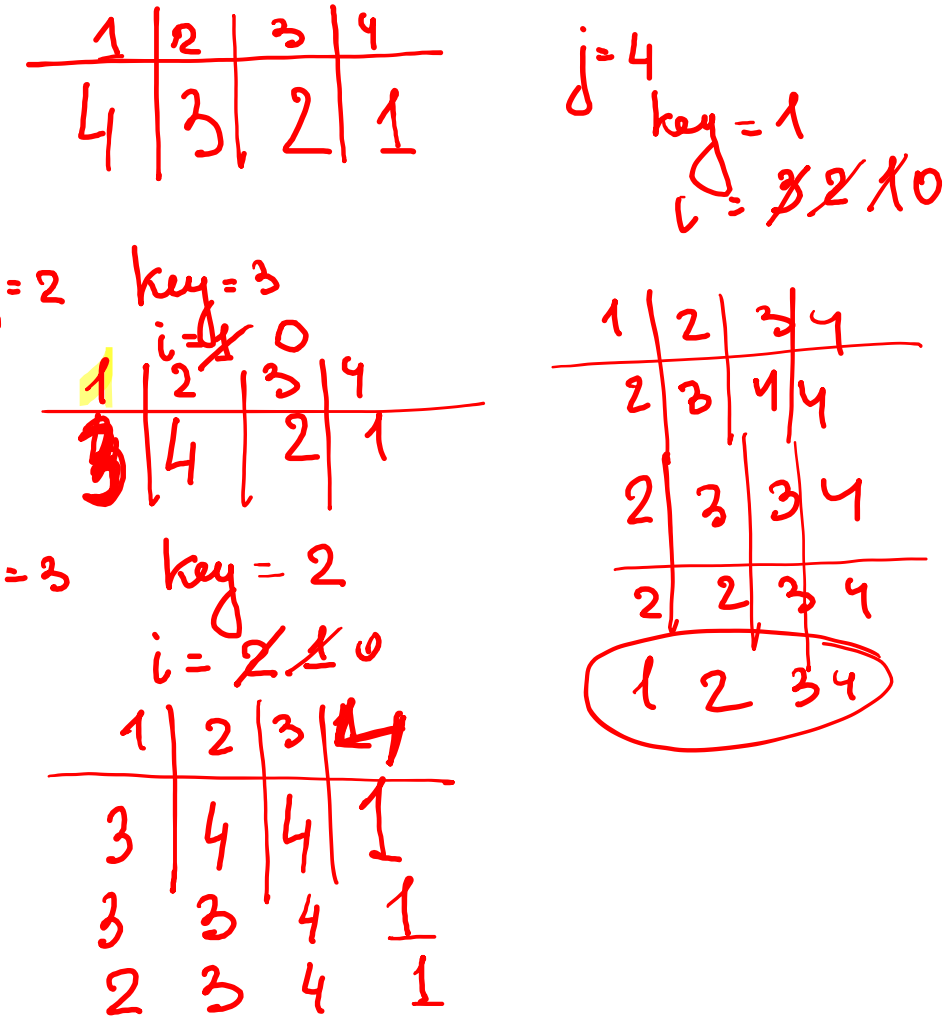
https://habr.com/ru/post/422085/

Сортировка вставками (Insertion sort)

- Сортировка вставками — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов



Анализ алгоритма



N

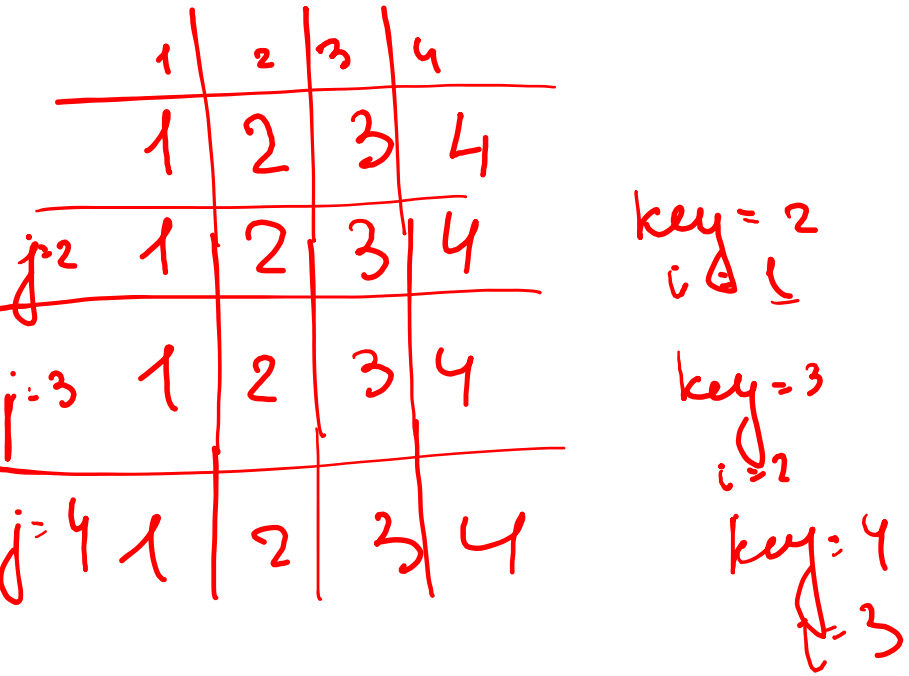
```
1. for j = 2 to A.length do
2.     key = A[j]
3.     i = j - 1
4.     while (i > 0 and A[i] > key) do
5.         A[i + 1] = A[i]
6.         i = i - 1
7.     end while
8.     A[i+1] = key
9. end for
```

$1 + 2 + 3 + \dots + N-1$ ($N-1$ слагаемое)

$\frac{1+(N-1)}{2} \cdot (N-1) \Rightarrow O(N^2)$

Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Анализ алгоритма

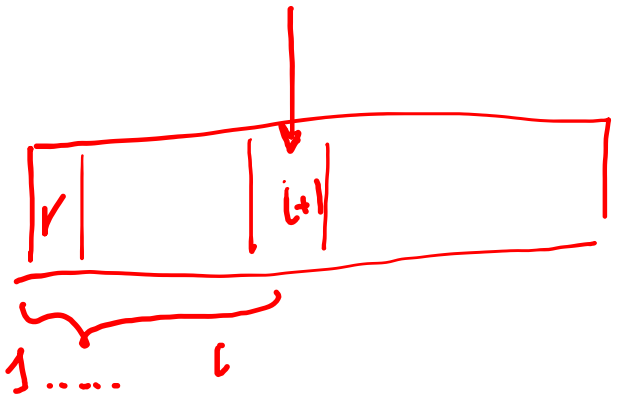



```
1. for j = 2 to A.length do
2.     key = A[j]
3.     i = j - 1
4.     while (i > 0 and A[i] > key) do
5.         A[i + 1] = A[i]
6.         i = i - 1
7.     end while
8.     A[i+1] = key
9. end for
```

N-1 → O(N)

Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Анализ алгоритма




```
1. for j = 2 to A.length do
2.     key = A[j]
3.     i = j - 1
4.     while (i > 0 and A[i] > key) do
5.         A[i + 1] = A[i]
6.         i = i - 1
7.     end while
8.     A[i+1] = key
9. end for
```

$$\sum_{i=1}^{N} \frac{1+i}{2} = \frac{1}{2}\sum_{i=1}^{N}(1+i) = \frac{1}{2}\cdot(+1+2+1+3+\dots 1+N) =$$

$$= \frac{1}{2}\left(N-1+\frac{2+N}{2}\cdot(N-1)\right) = \frac{1}{2}\left(N-1+\frac{2N+N^2-2-N}{2}\right) \to O(N^2)$$

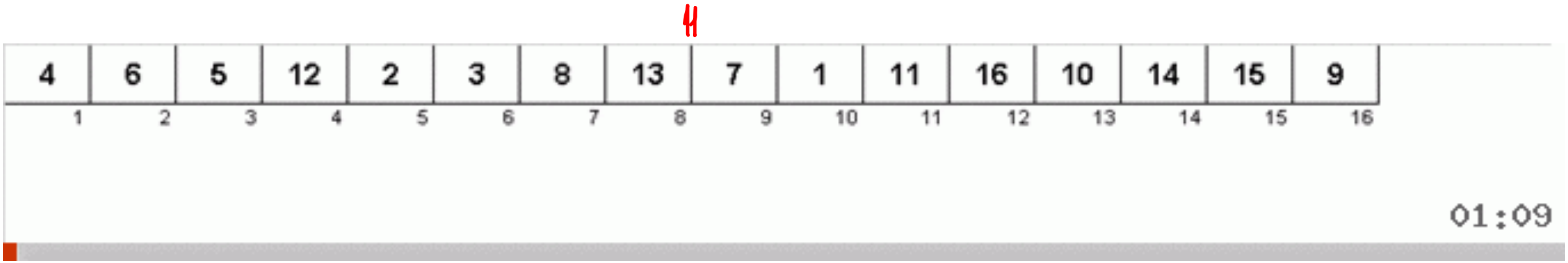
Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Сортировка вставками (Insertion sort)

- Время работы в худшем и среднем случае - $O(n^2)$
- Время работы в лучшем случае - $O(n)$
- Дополнительная память - $O(1)$
- Устойчивость?

```
1. for j = 2 to A.length do
2.     key = A[j]
3.     i = j - 1
4.     while (i > 0 and A[i] > key) do
5.         A[i + 1] = A[i]
6.         i = i - 1
7.     end while
8.     A[i+1] = key
9. end for
```

Еще пример



https://habr.com/ru/post/415935/

Сортировка бинарными вставками

- Мы можем использовать бинарный поиск, чтобы уменьшить количество сравнений в обычной сортировке вставкой . Бинарная сортировка вставок использует бинарный поиск, чтобы найти правильное место для вставки выбранного элемента на каждой итерации.
- При обычной сортировке вставкой в худшем случае требуется $O\ (n)$ сравнений (на n-й итерации). Мы можем уменьшить его до $O\ (\log n)$, используя бинарный поиск .

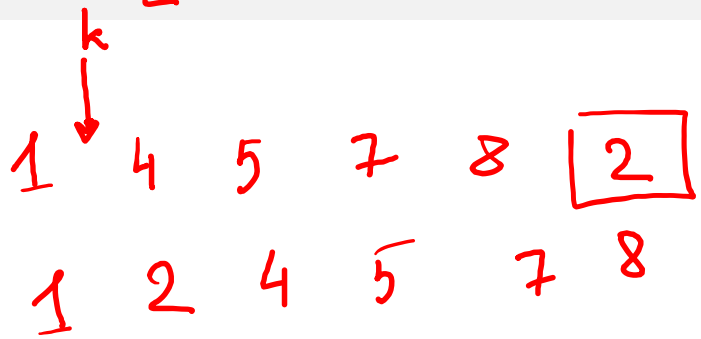
Сортировка бинарными вставками

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Время работы в лучшем случае - $O(n)$
- Дополнительная память - $O(1)$

```
1. function insertionSort(a):
2.   for i = 1 to n - 1
3.     j = i - 1
4.     k = binSearch(a, a[i], 0, j)
5.     for m = j downto k
6.       swap(a[m], a[m+1])
```

logN

n log n + n·n



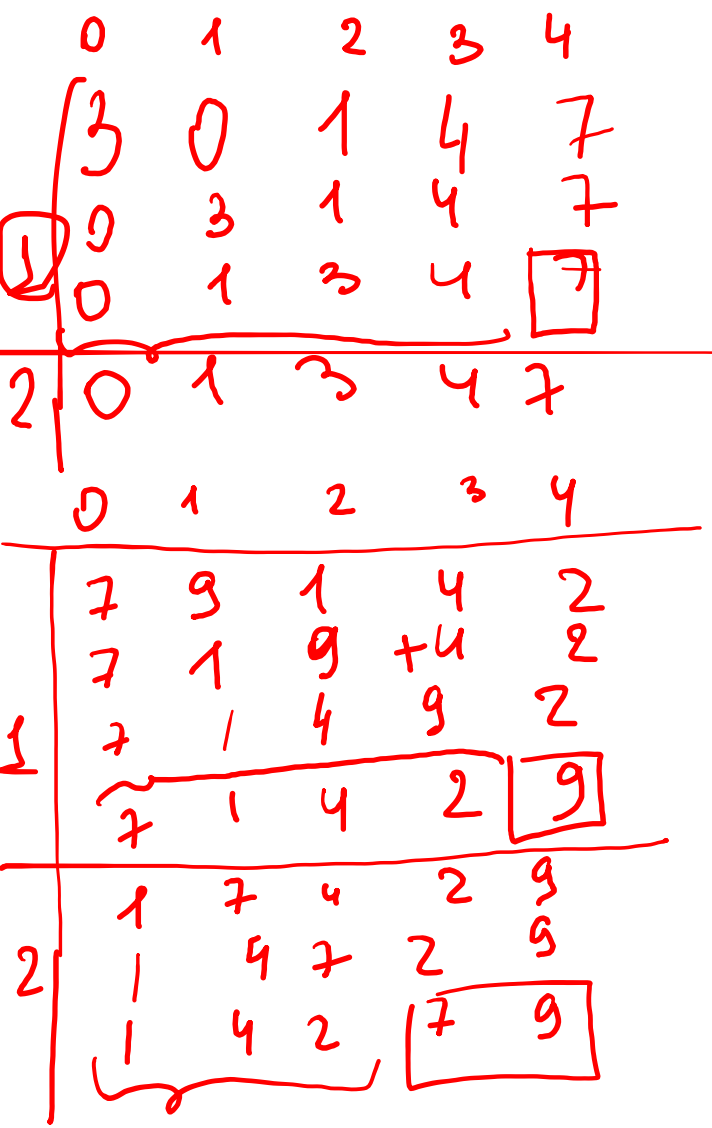
Сортировка вставками. Достоинства и недостатки

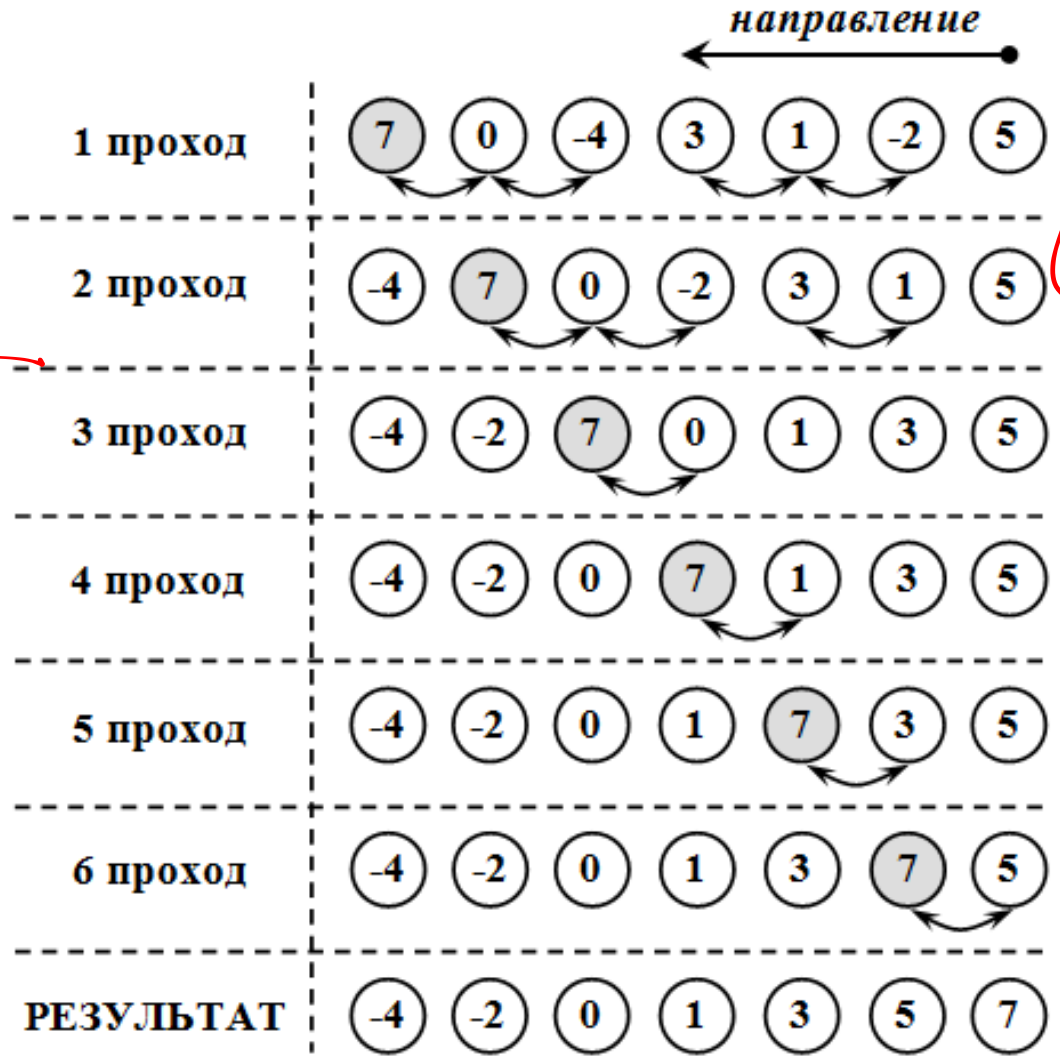
- эффективен на небольших наборах данных, на наборах данных до десятков элементов может оказаться лучшим;
- эффективен на наборах данных, которые уже частично отсортированы;
- это устойчивый алгоритм сортировки (не меняет порядок элементов, которые уже отсортированы);
- может сортировать список по мере его получения;
- не требует доп.памяти
- **Минусом** же является высокая сложность алгоритма: $O(n^2)$.

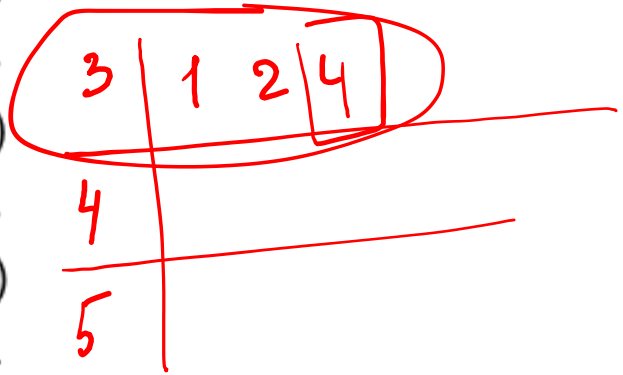
Сортировка пузырьком (bubble sort)

- Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции как пузырёк в воде, отсюда и название алгоритма).

Сортировка пузырьком (bubble sort)



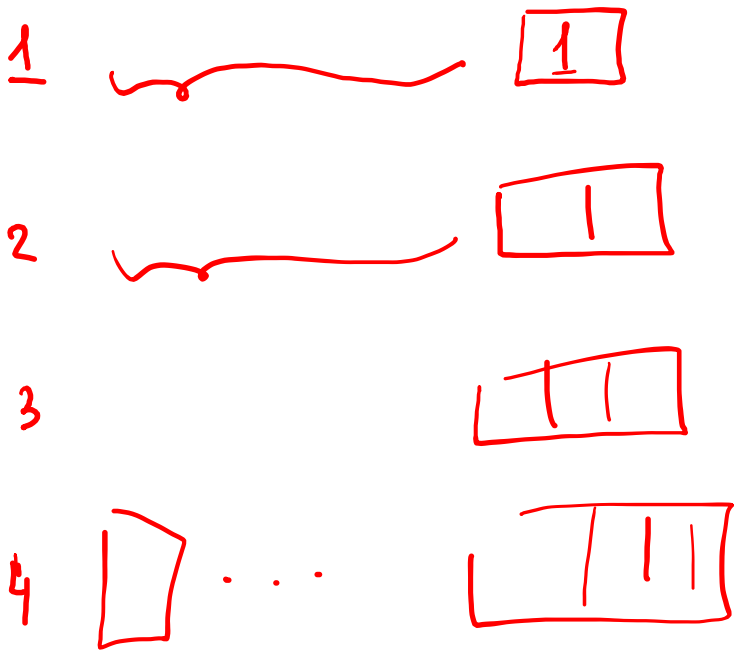




Сортировка пузырьком

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Дополнительная память - $O(1)$

```
1. function bubbleSort(a):
2.   for i = 0 to n - 2
3.     for j = 0 to n - i - 2
4.       if a[j] > a[j + 1]
5.         swap(a[j], a[j + 1])
```



Пример

i	*j*		*1*	*2*	*3*	*4*	*5*	*6*
1	1		4	5	9	1	3	6
	2		4	5	9	1	3	6
	3	обмен	4	5	9	1	3	6
	4	обмен	4	5	1	9	3	6
	5	обмен	4	5	1	3	9	6
2	1		4	5	1	3	6	**9**
	2	обмен	4	5	1	3	6	**9**
	3	обмен	4	1	5	3	6	**9**
	4		4	1	3	5	6	**9**
3	1	обмен	4	1	3	5	**6**	**9**
	2	обмен	1	4	3	5	**6**	**9**
	3		1	3	4	5	**6**	**9**
4	1		1	3	4	**5**	**6**	**9**
	2		1	3	4	**5**	**6**	**9**
5	1		1	3	**4**	**5**	**6**	**9**

Оптимизация (условие Айверсона 1)

- Если после выполнения внутреннего цикла не произошло ни одного обмена, то массив уже отсортирован, и продолжать что-то делать бессмысленно.

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Анализ алгоритма

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Анализ алгоритма

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Анализ алгоритма

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Оптимизация (условие Айверсона 1)

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Время работы в лучшем случае - $O(n)$
- Дополнительная память - $O(1)$

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Оптимизация (условие Айверсона 2)

- Запоминаем, в какой позиции t был последний обмен на предыдущей итерации внешнего цикла.
- Это – верхняя граница просмотра массива Bound на следующей итерации
- Если t = 0 после выполнения внутреннего цикла, значит, обменов не было, алгоритм заканчивает работу.
- Основная идея – **уменьшаем количество проходов внутреннего цикла**

Модификации

- Сортировка чет-нечет
- Сортировка расческой
- Сортировка перемешиванием (шейкерная сортировка)

	Устойчивость	Онлайн	Лучший случай	Средний случай	Худший случай	Доп. память
Сортировка выбором	нет	нет	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка вставками	да	да	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка бин. вставками	да	да	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка пузырьком	да	нет	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка пузырьком с условием Айверсона 1	да	нет	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

a_n – набор данных

$[l, r) \longrightarrow$ sum

	0	1	2	3	4	5	6	7
	1	4	0	6	7	2	3	9
0	1	5	5	11	18	20	23	32

$[1, 4) \rightarrow 10$

$[2, 5) \rightarrow 13$

хороший $Q \lll n$

$n = 10^9$

$Q \approx n = 10^9$

1 алгоритм

```
for i = 1 ... Q
    l, r
    sum = 0
    for j = l ... r
        sum += a[j]
```

$O(Q \cdot n) \rightarrow O(n^2)$

2 алгоритм $\quad O(n + Q) \rightarrow O(n)$

```
for i := 0 ... N
    sum[i] := sum[i-1] + a[i]
for i := 1 Q
    l, r
    print ( sum[R] - sum[L] )
```

sum[r] = a[0] + a[1] + a[2] + ... a[l] + ... a[r-1]

sum[l] = a[0] + a[1] + a[2] + ... a[l-1]

Префиксные суммы

Префиксными суммами массива $[a_0, a_1, a_2, \dots, a_{n-1}]$ называется массив $[s_0, s_1, s_2, \dots, s_n]$, определенный следующим образом:

$$\begin{cases} s_0 = 0 \\ s_1 = a_0 \\ s_2 = a_0 + a_1 \\ s_3 = a_0 + a_1 + a_2 \\ \dots \\ s_n = a_0 + a_1 + a_2 + \dots + a_{n-1} \end{cases}$$

Обратите внимание, что в такой индексации

- s_k равен сумме первых *k* массива *a*, не включая a_k,
- длина *s* на единицу больше длины *a*,
- s_0 всегда равен нулю.

```
1.  int *s = new int[n + 1];
2.  s[0] = 0;
3.  for (int i = 0; i < n; ++i)
4.    s[i + 1] = s[i] + a[i];
5.  . . . .
6.  delete[] s;
```

© Горденко М.К., 2022, CC BY

Задача

Задача. Дан массив целых чисел, и приходят запросы вида «найти сумму на полуинтервале с позиции *l* до позиции *r*». Нужно отвечать на запросы за $O(1)$.

© Горденко М.К., 2022, CC BY

Задача

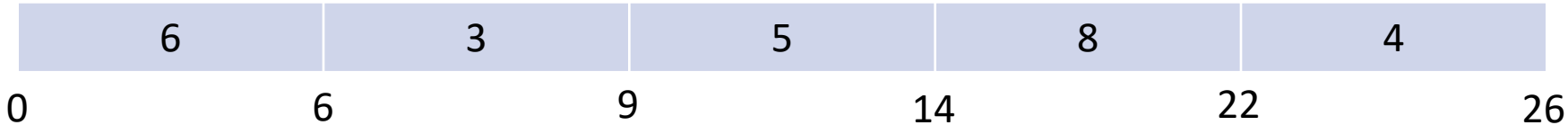
Произведем предварительный подсчет перед ответами на запросы: массив префиксных сумм для исходного массива. Тогда в случае, если бы во всех запросах левая граница s_l была бы равна нулю, то ответом на запрос была бы стандартная префиксная сумма s_r, однако как поступить, если эта граница не равна нулю?

В префиксной сумме s_r содержатся все нужные нам элементы, однако присутствуют и лишние, а именно $a_0, a_1, \dots, a_{r-1}$. Заметим, что такая сумма будет равна посчитанной префиксной сумме s_l.

Таким образом, выполняется тождество:

$$a_l + a_{l+1} + \cdots + a_{r-1} = s_r - s_l$$

Для ответа на запрос поиска суммы на произвольном полуинтервале необходимо вычесть друг из друга две известные префиксные суммы.



© Горденко М.К., 2022, CC BY

Другие операции

- побитовое исключающее «или», также известное как xor и обозначаемое $\oplus$,
- сложение по модулю,
- умножение и умножение по модулю (обратное — деление).

© Горденко М.К., 2022, CC BY

a, Q (l, r) Найти кол-во элементов = X(0) в интервале [l, r)

	0	1	2	3	4	5	6	7	8
	1	2	0	0	3	0	0	1	0
	1	2	3	4	5	6	7	8	9
сочет	0	0	0	1	1	1	3	4	5

[3 – 5) → 3 – 2 = 1

[2 – 7) → 4

[4, 8) → 4 – 2 = 2

$O(N)$

$Q(l, r)$

произведение?

$O(N+Q)$

$E(N)$

	0	1	2	3	4	5	6	7	8
	1	2	3	0	2	0	2	3	4
	0	1	2	3	4	5	6	7	8	9
p	1	1	2	6	1	2	1	2	6	24
c	0	0	0	0	1	1	2	2	2	2

if $c[r] == c[l]$ $\frac{p[r]}{p[l]}$ else 0

$[1-5)$: $\frac{12}{1} = 12$

$[2-6)$: $\frac{48}{2} = 24 \rightarrow 0$

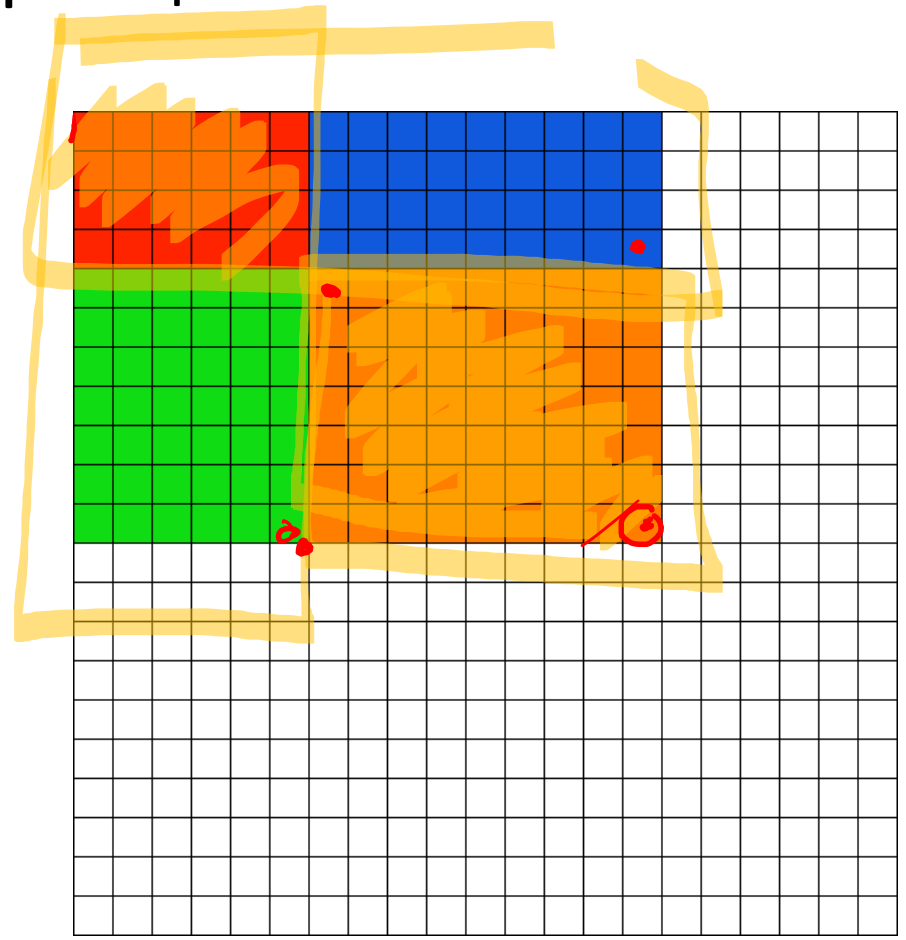
$[6-8)$: $\frac{6}{1} = 6$

Префиксные суммы в матрицах

Задача. Необходимо отвечать на запросы вида «найти сумму в прямоугольнике (x_1, y_1, x_2, y_2)».

Разложим его на следующие запросы на «префиксах»:

$$\begin{cases} sum(x_1, y_1, x_2, y_2) = sum(0,0, x_2, y_2) - \\ sum(0,0, x_1 - 1, y_2) - sum(0,0, x_2, y_1 - 1) + \\ sum(0,0, x_1 - 1, y_1 - 1) \\ pref[0][0] = a[0][0] \\ pref[0][i] = pref[0][i-1] + a[0][i] \\ pref[i][0] = pref[i-1][0] + a[i][0] \\ pref[i][j] = pref[i-1][j] + pref[i][j-1] - \\ pref[i-1][j-1] + a[i][j] \end{cases}$$



© Горденко М.К., 2022, CC BY

Задача

- **Задача.** Эффективно вычислять количество нулей в отрезке массива.

© Горденко М.К., 2022, CC BY

Задача

- **Задача.** Эффективно вычислять произведение на отрезке массива.

© Горденко М.К., 2022, CC BY

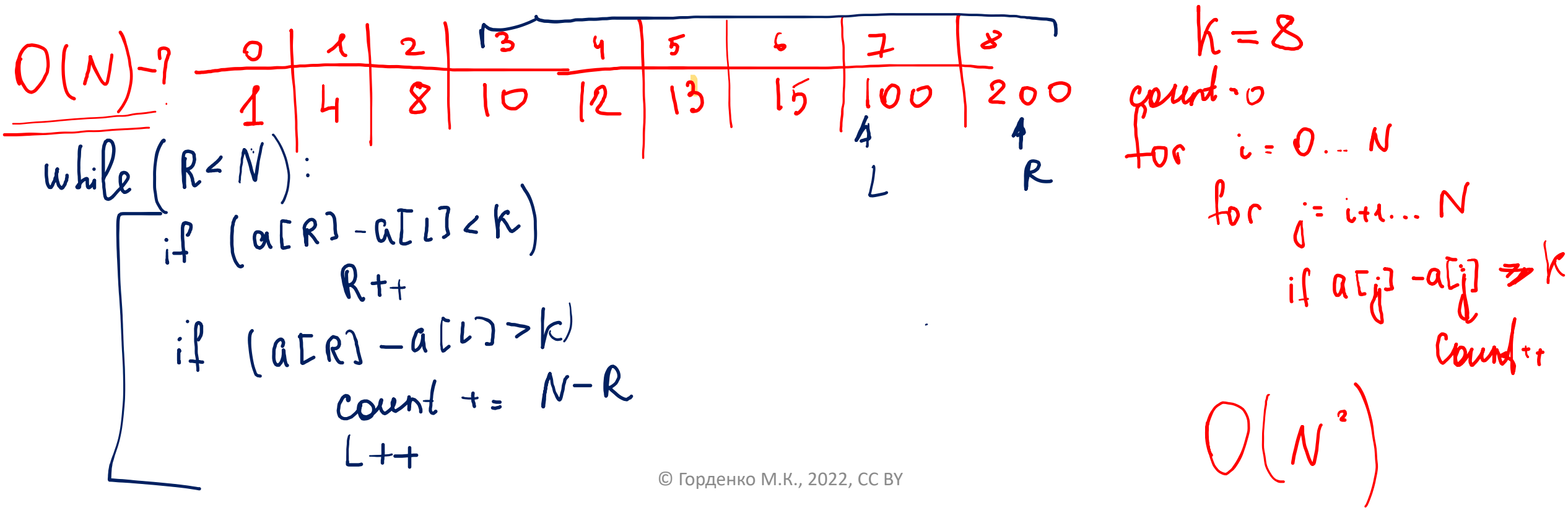
Два указателя

- Два указателя — это простой и эффективный метод, который обычно используется для поиска пар в отсортированном массиве.

© Горденко М.К., 2022, CC BY

Задача

- **Задача.** Найти количество пар элементов *a* и *b* в отсортированном массиве, такие что $b - a > K$.



© Горденко М.К., 2022, CC BY

Задача

- Наивное решение: бинарный поиск. Будем считать, что массив уже отсортирован. Для каждого элемента *a* найдем первый справа элемент *b*, который входит в ответ в паре с *a*. Нетрудно заметить, что все элементы, большие *b*, также входят в ответ.
  Итоговая сложность: $O(N * \log N)$.
- А можно ли быстрее? Да, давайте рассматривать **два указателя** – два индекса *first* и *second*. Будем перебирать *first* слева направо и поддерживать для каждого *first* такой первый элемент *second* справа от него, что $a[second] - a[first] > K$. Тогда в пару к $a = [first]$ подходят ровно $n - second$ элементов массива, начиная с *second*.

```
1.  int second = 0;
2.  int ans = 0;
3.  for (int first = 0; first < n; ++first) {
4.    while (second != n && a[second] - a[first] <= r)
5.      ++second;
6.    ans += n - second;
7.  }
```

© Горденко М.К., 2022, CC BY