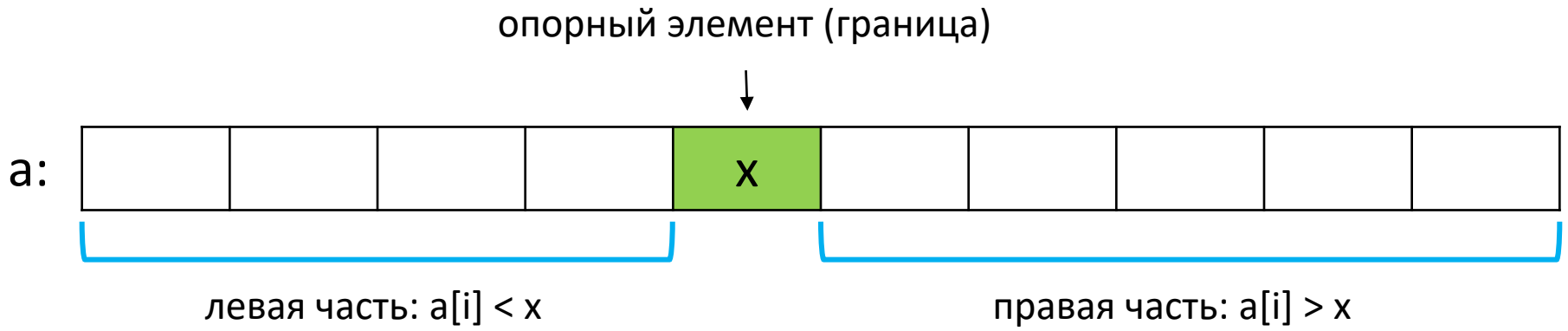
Сортировки

Часть 2

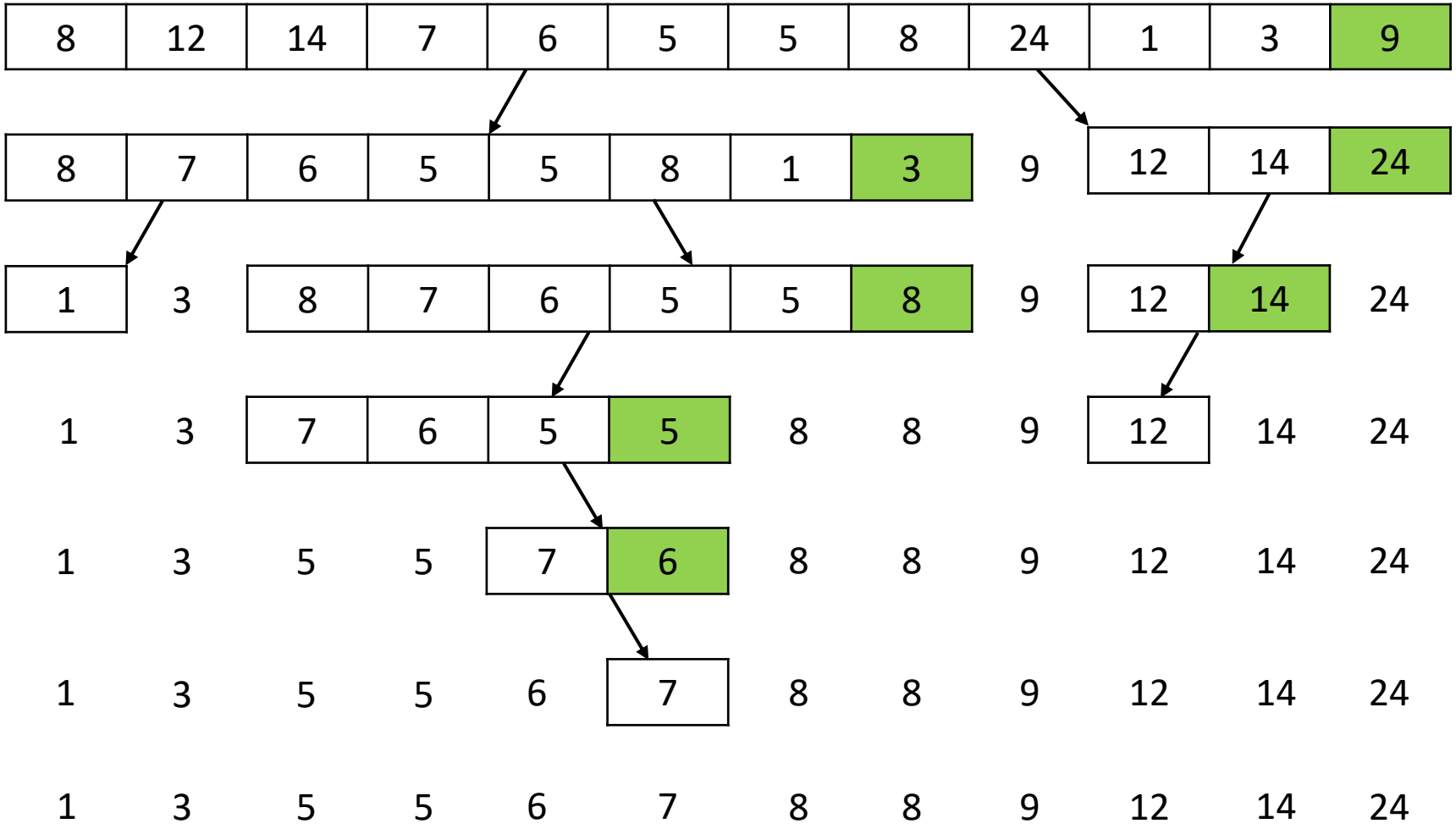
Быстрая сортировка: принцип

1. Разбиение (partition)
   - выбираем опорный (граничный) элемент
   - делим массив относительно опорного: все элементы массива, меньшие опорного – слева, большие – справа



2. Запускаем процесс разбиения отдельно в левой и правой частях полученного массива

Пример работы сортировки



Быстрая сортировка: функция сортировки

```
void quicksort(int a[n], int l, int r)
{
    if (l < r) // условие выхода из рекурсии (одноэлементный массив всегда отсортирован)
    {
        int q = partition(a, l, r); // функция разбиения массива относительно границы

        quicksort(a, l, q);
        quicksort(a, q + 1, r);  // рекурсивный запуск из левой и правой частей
    }
}
```

Note! Рекурсия: на спуске – разбиение и сортировка
на возврате – сбор массива

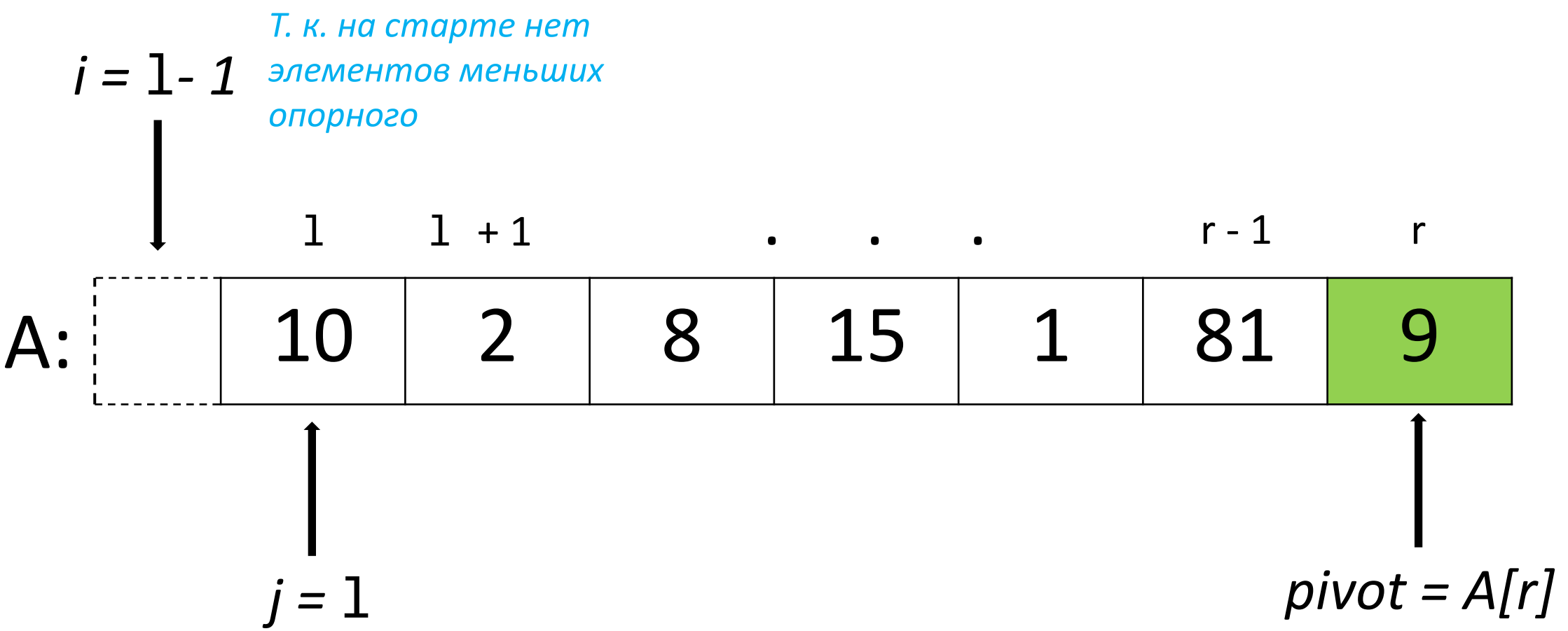
Быстрая сортировка: разбиение

Ключевой частью алгоритма является функция разбиения, изменяющая порядок элементов подмассива a[l…r] без привлечения дополнительной памяти

1. Разбиение Ломуто

- Опорным выбирается последний элемент в рассматриваемой части массива
- Необходимо 2 индекса:
  - i – отвечает за границу элементов меньших опорного, на старте считаем, что таких элементов нет
  - j – для просмотра массива
- Просматриваем массив, и каждый раз, когда находится элемент, меньший либо равный опорному, увеличиваем границу элементов меньших опорного (индекс i) и вставляем найденный элемент перед опорным в зону меньших опорного

Partition(A, l, r)



Partition(A, l, r)

?: A[j] < pivot

?: 10 < 9?

Нет

i = l- *1*

l l + 1 . . . r - 1 r

A: | | 10 | 2 | 8 | 15 | 1 | 81 | 9 |

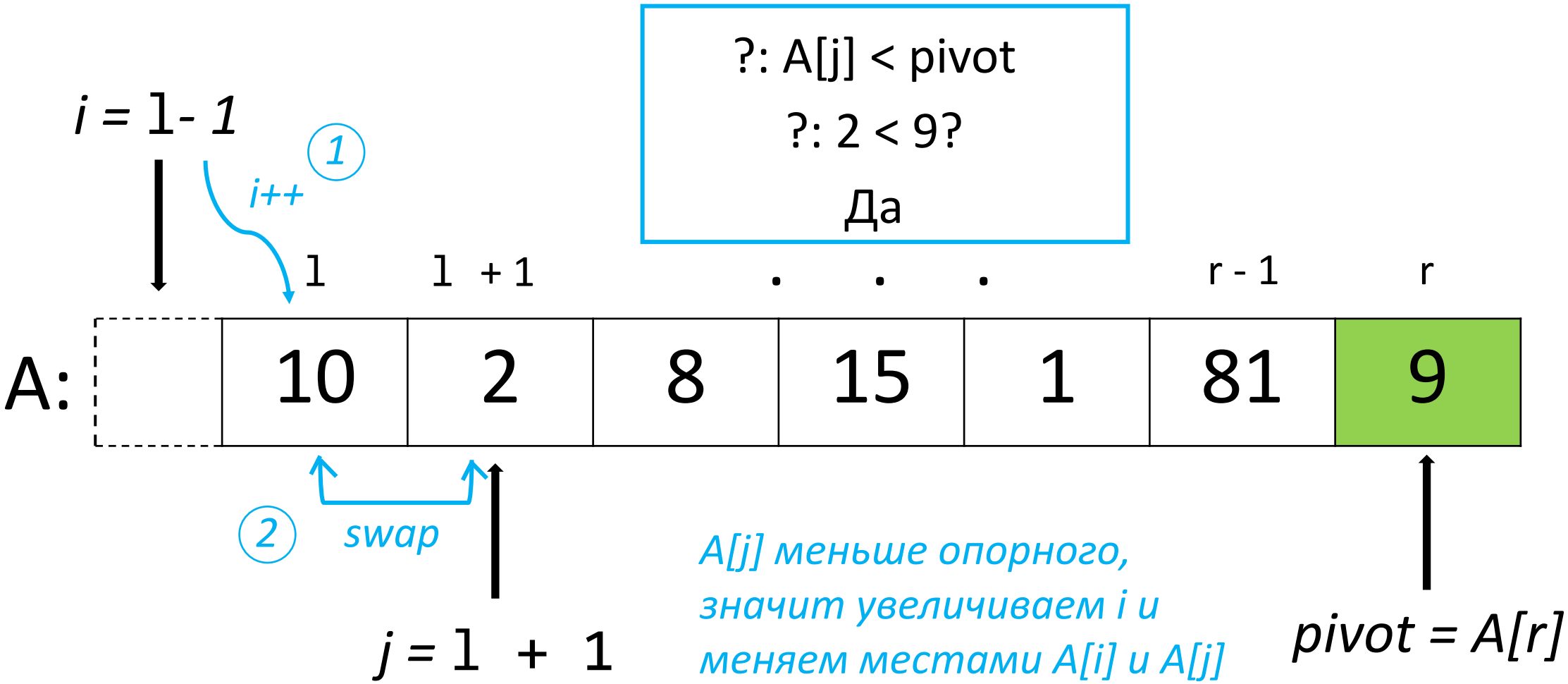
j = l

j++

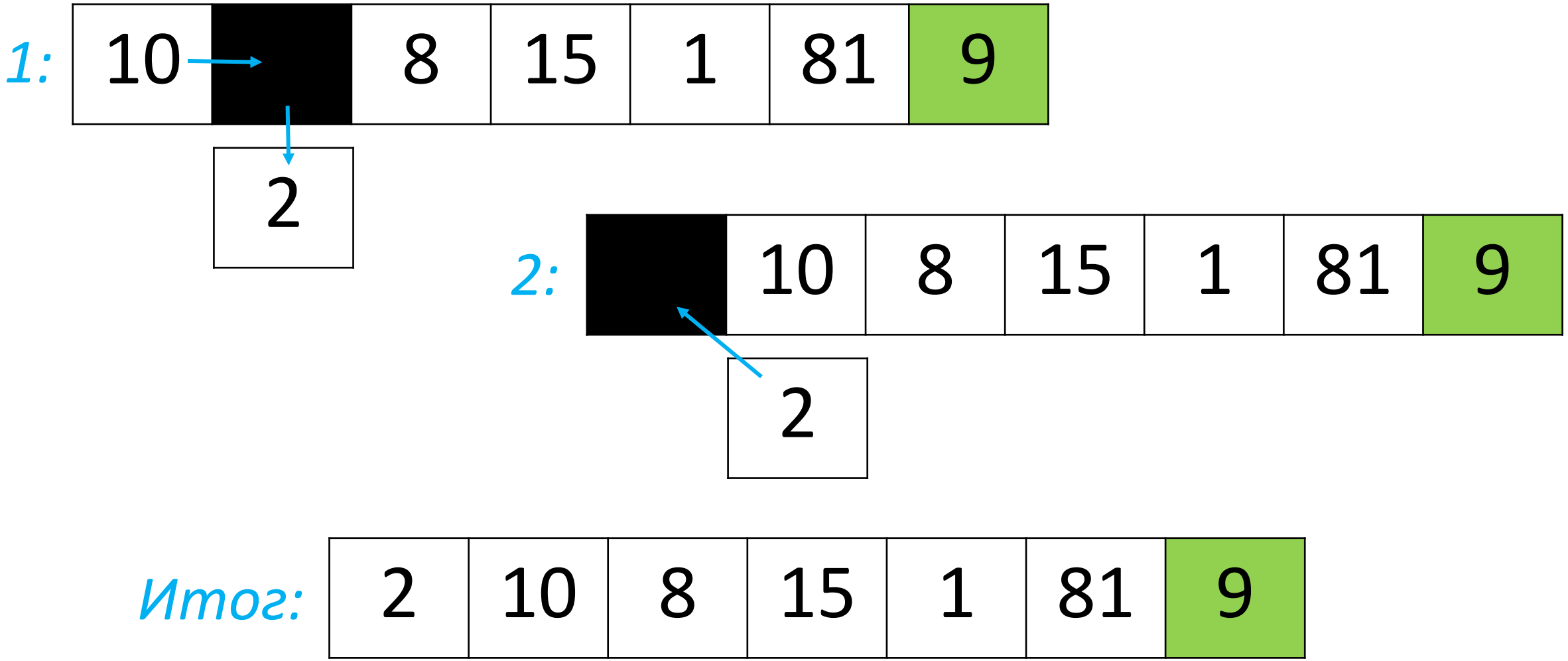
Пока A[j] > pivot, увеличиваем j и не делаем ничего с i, т. к. нет элементов меньше опорного

pivot = A[r]

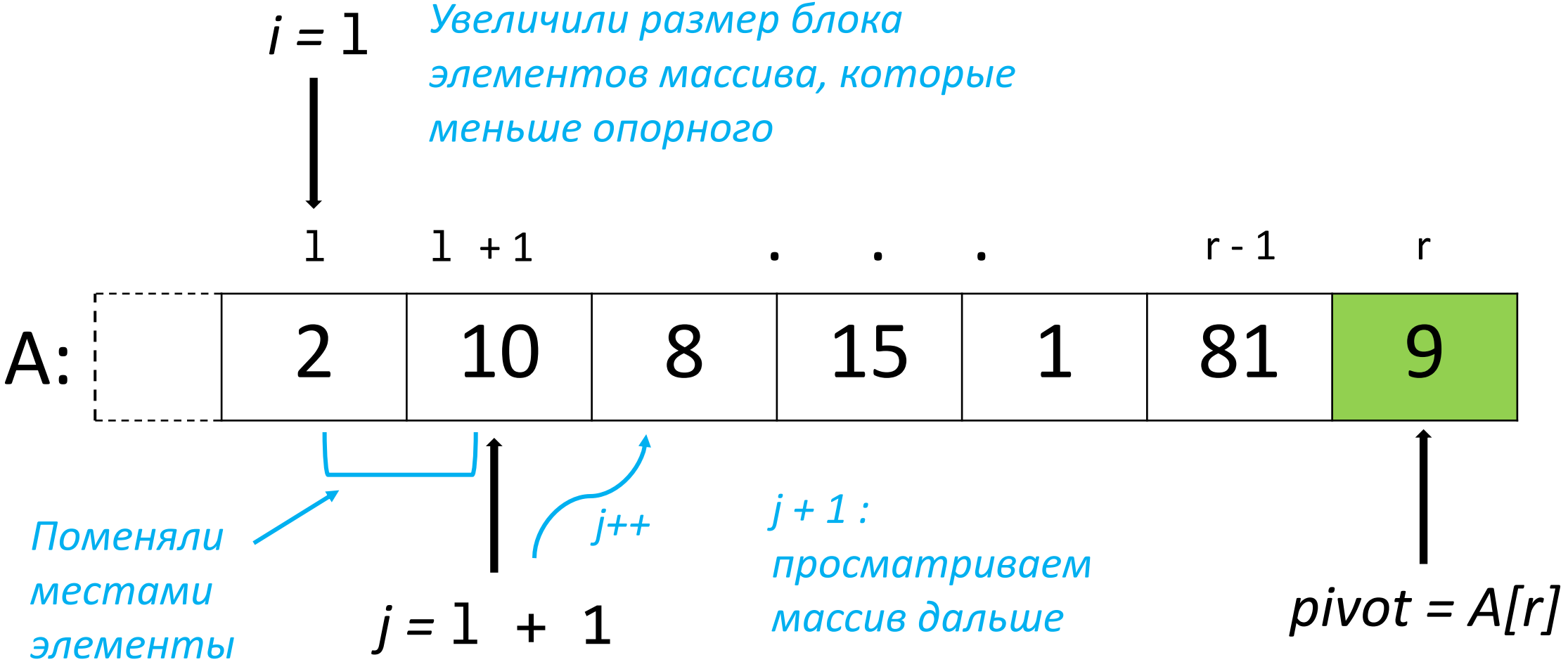
Partition(A, l, r)



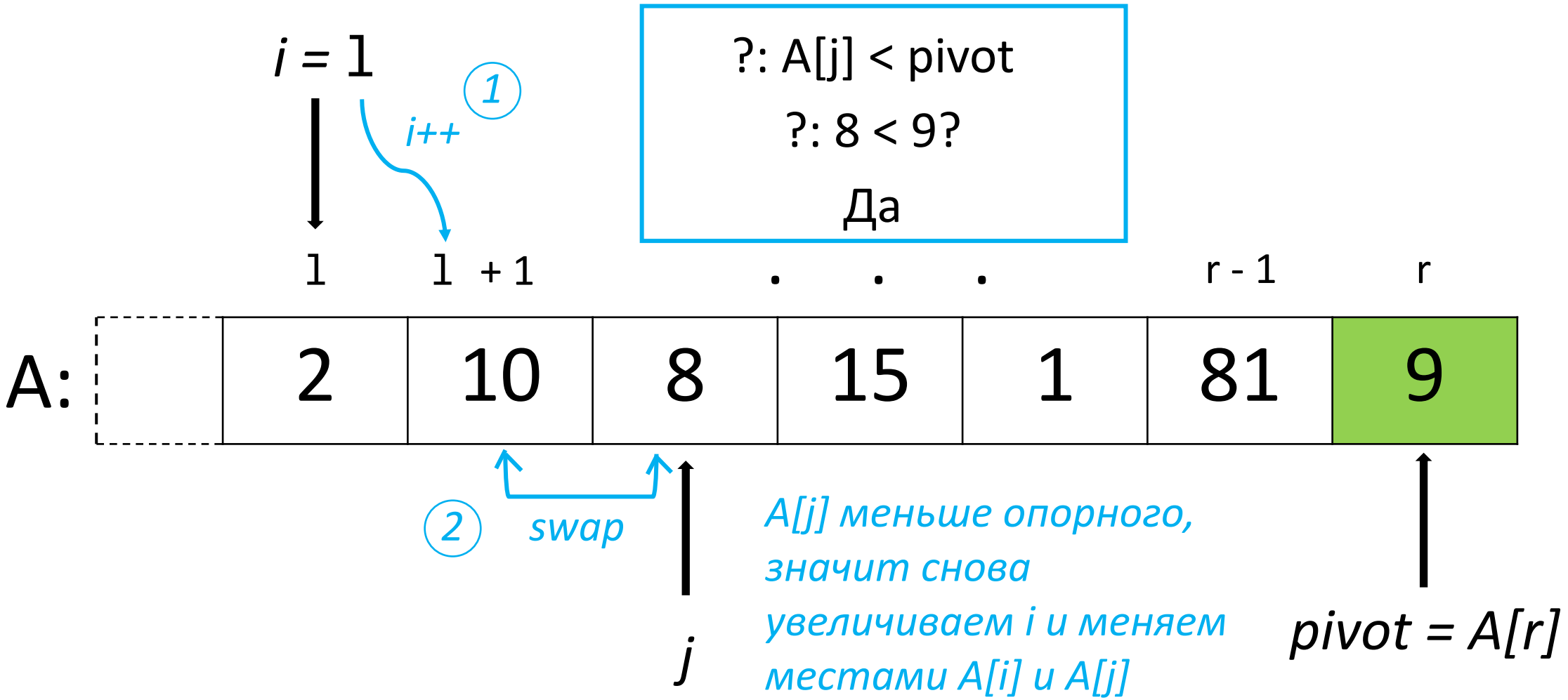
Процедура swap



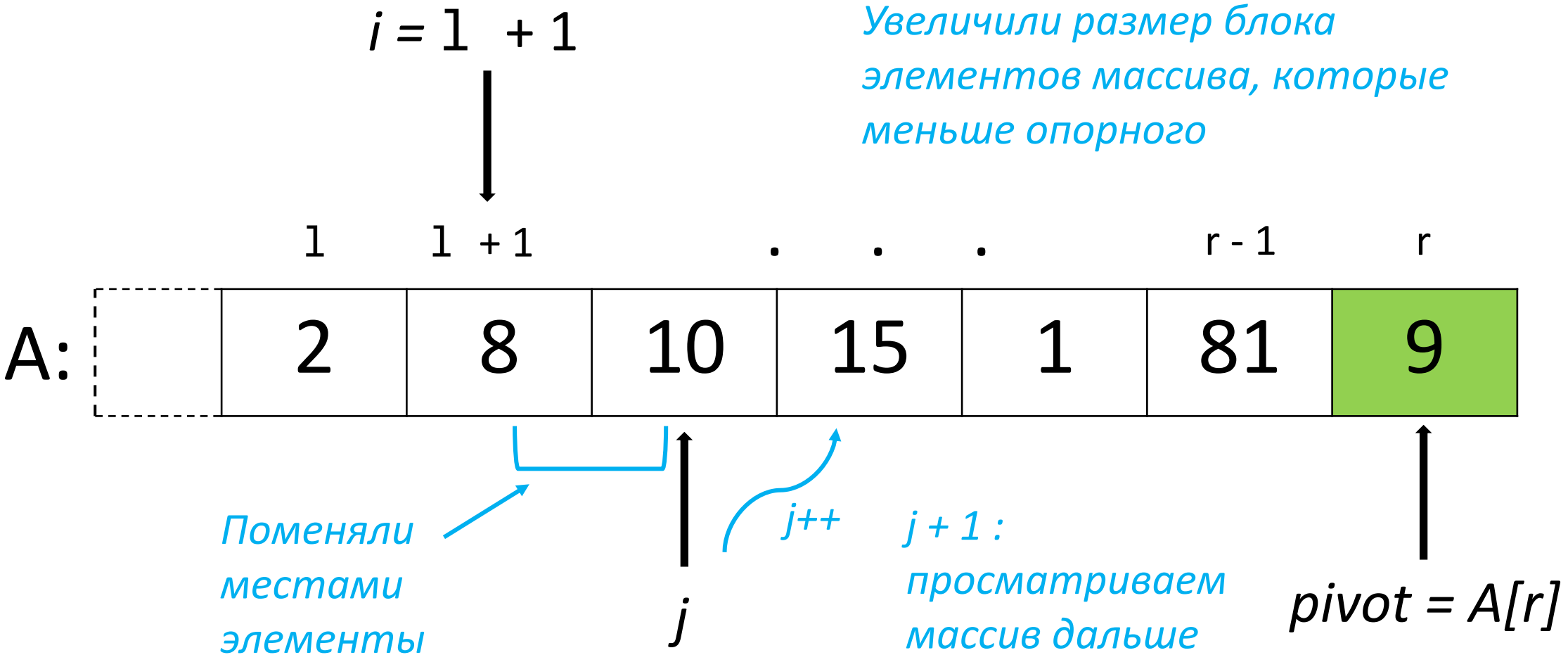
Partition(A, l, r)



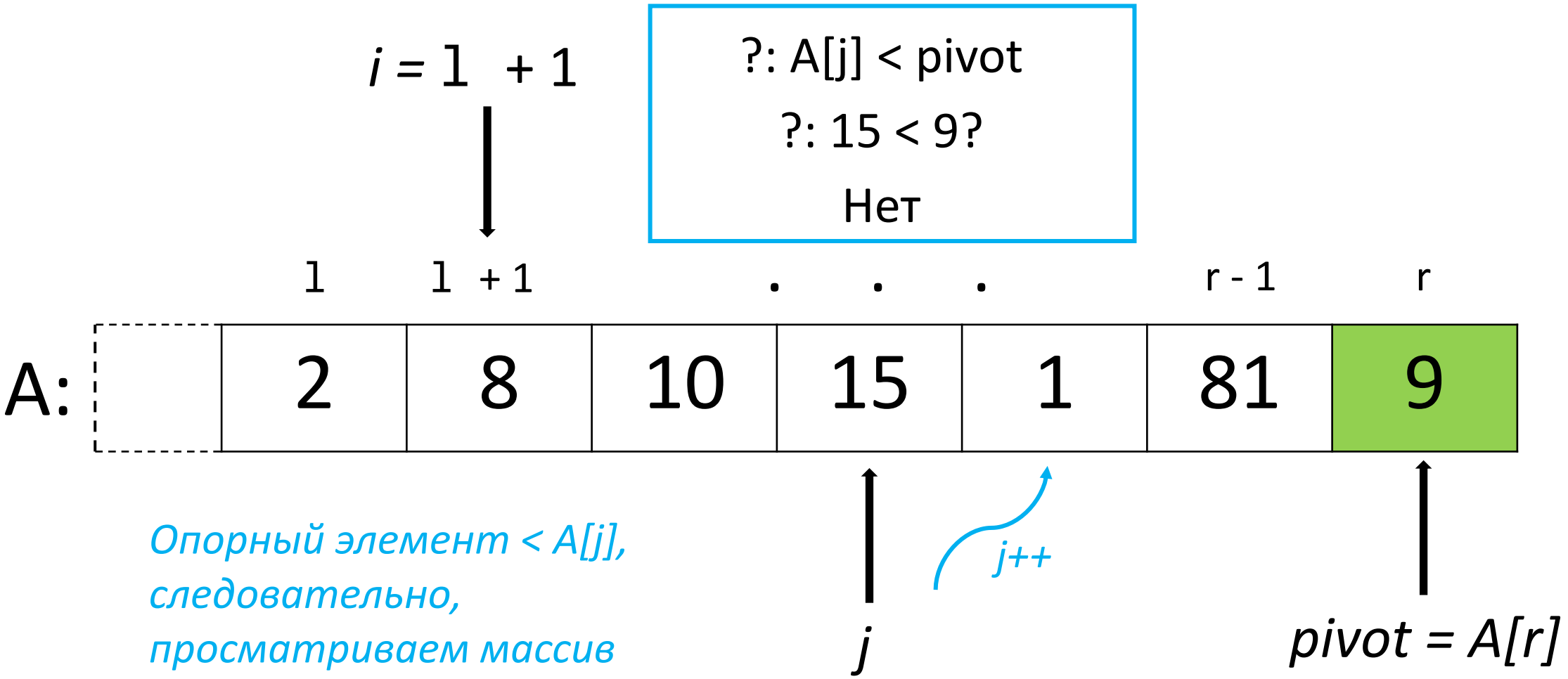
Partition(A, l, r)



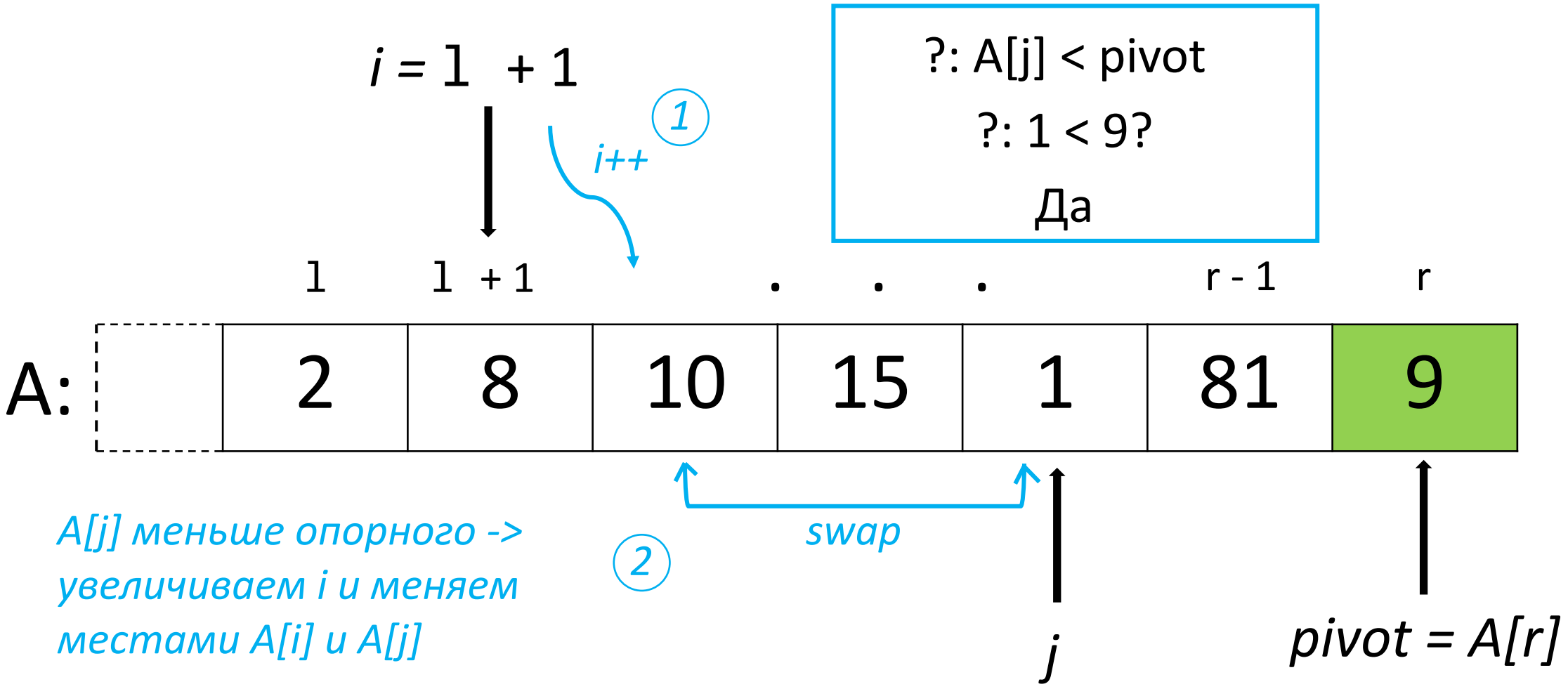
Partition(A, l, r)



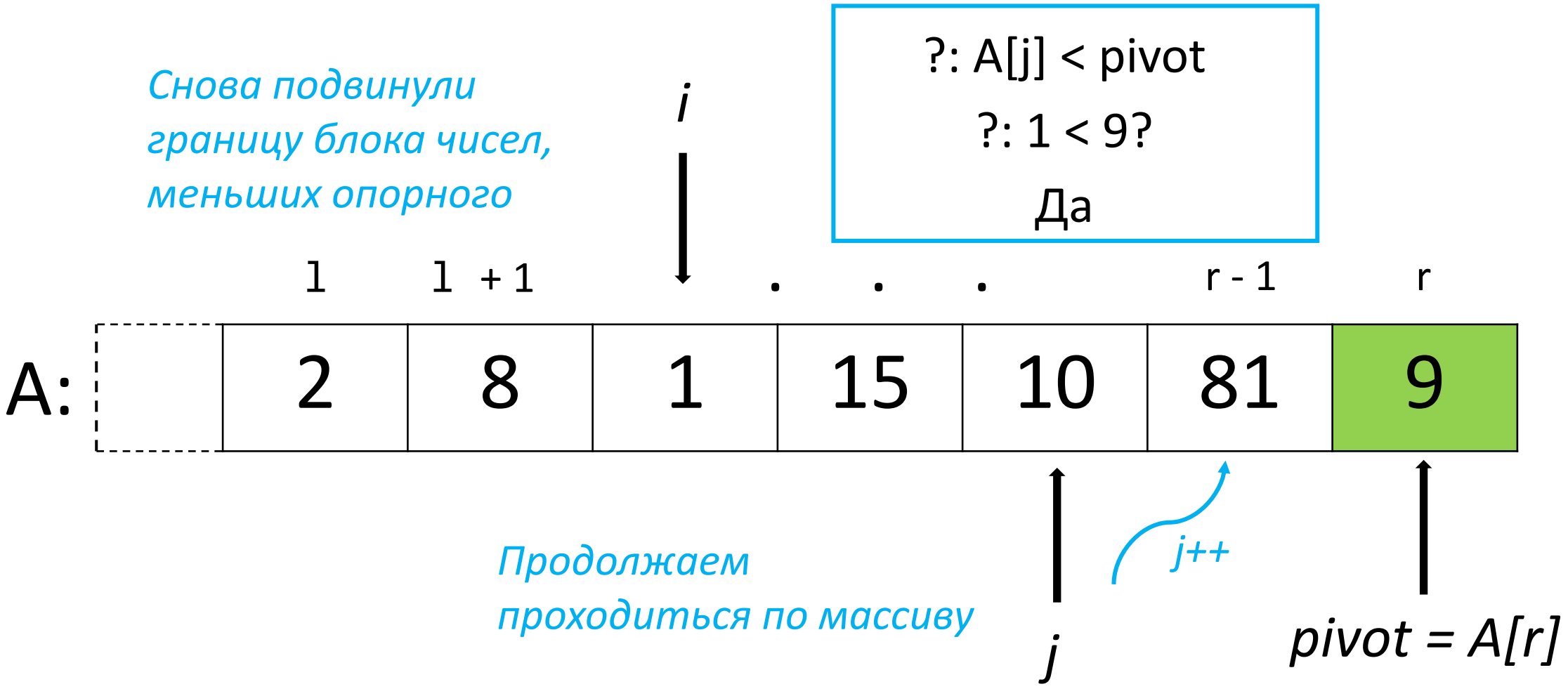
Partition(A, l, r)



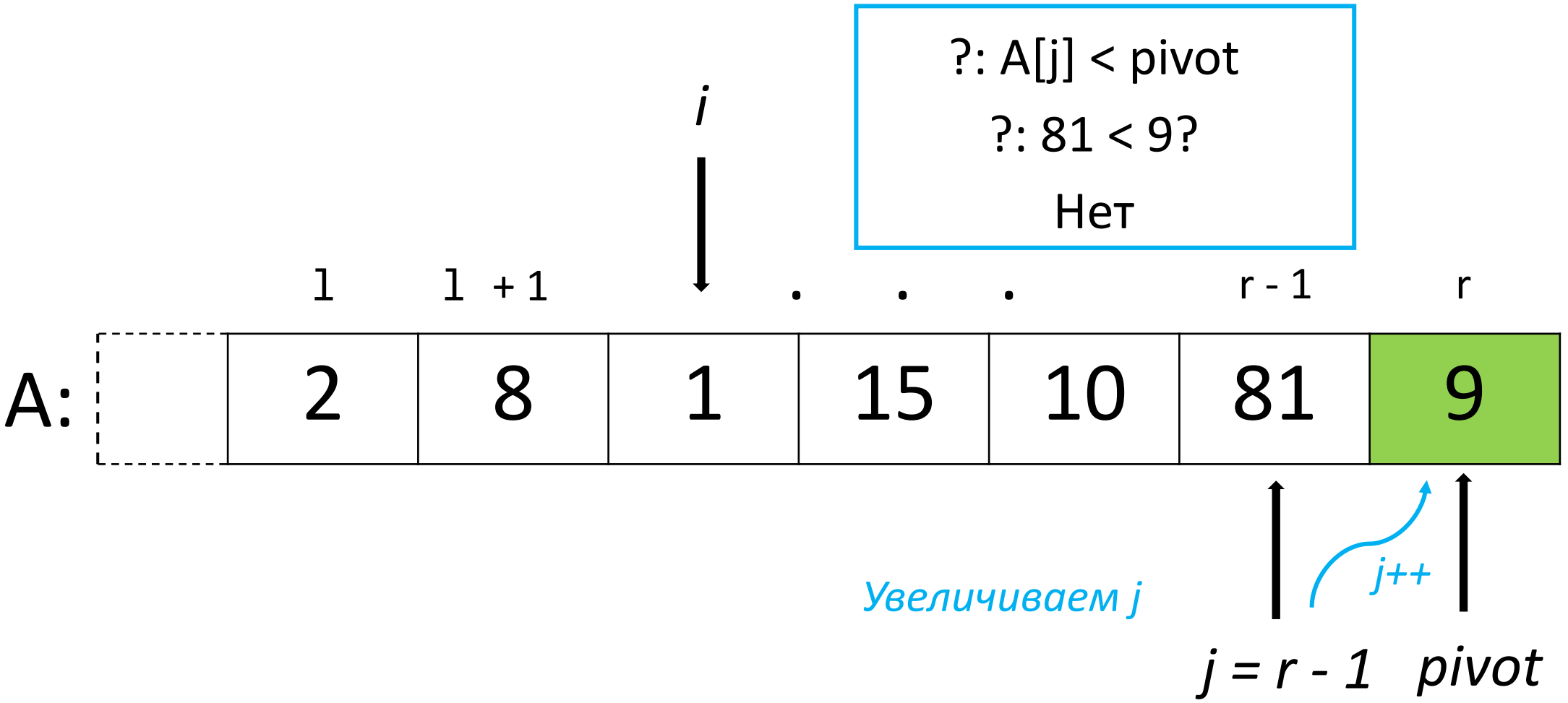
Partition(A, l, r)



Partition(A, l, r)



Partition(A, l, r)



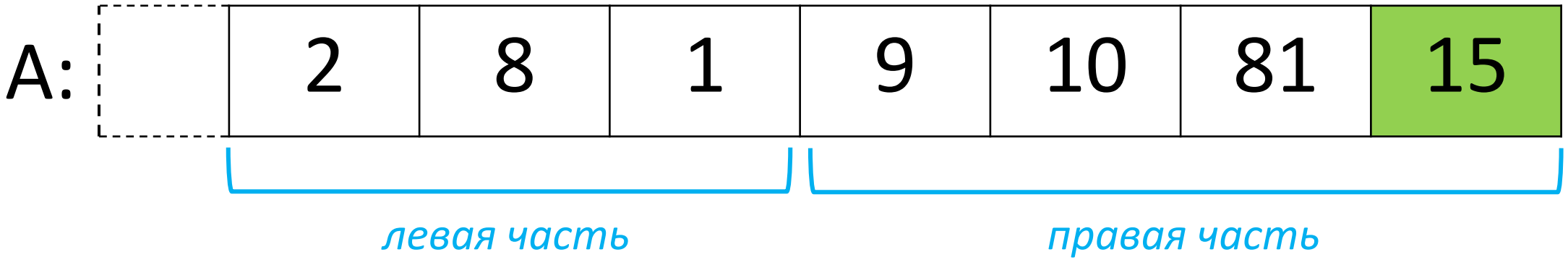
Partition(A, 1, r)

Дошли до опорного, *A[j] = pivot*, *j = r* *условие завершения разбиения*

Чтобы разбиение было верным, опорный элемент должен находиться между элементами большими, чем он, и меньшими, чем он. Значит его нужно поставить на позицию *i + 1*

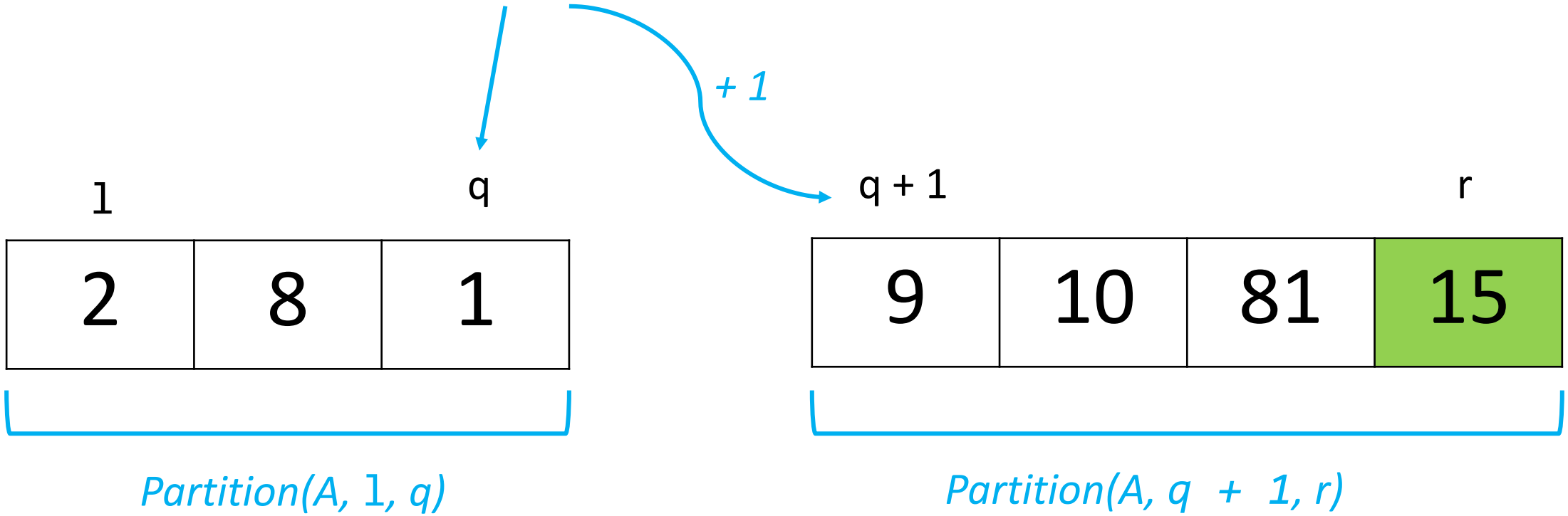
Partition(A, 1, r)

Теперь все элементы, меньшие 9, стоят слева от нее, а все большие - справа



Однако левая и правая часть сами по себе не отсортированы, поэтому теперь необходимо запустить быструю сортировку отдельно от левой и правой частей

q = Partition(A, l, r)



Note? Нужно ли опорный элемент отправлять в повторное разбиение, ведь элемент точно на своем месте

Partition(A, q + 2, r)

Разбиение Ломуто: код

```
int partition(int a[n], int l, int r)
{
    int pivot = a[r]; // pivot – опорный элемент
    int i = l - 1;    // i – граница элементов, меньших опорного

    for (int j = l; j < r; j++) // просматриваем ту часть массива, от которой сейчас вызвались
    {
        if (a[j] <= pivot) // условие: опорный элемент больше либо равен просматриваемому
        {
            i = i + 1; // увеличиваем границу меньших опорного
            swap(a[i], a[j]); // вставляем туда найденный просматриваемый элемент
        }
    }
    swap(a[i + 1], a[r]); // меняем местами опорный элемент и первый элемент после
                          // меньших опорного
    return i + 1;
}
```

2. Разбиение Хоара

mid = (l + r) / 2
pivot = A[mid]

- Опорным выбирается элемент в середине массива, относительно него идет разбиение
- Необходимо 2 индекса (i в начале массива, j – в конце), которые приближаются друг к другу, пока не найдется пара элементов:
  - по i: элемент больше опорного и расположен перед ним (до среднего элемента) *A[i] > pivot*
  - по j: элемент меньше опорного и расположен после него (после среднего элемента) *A[j] < pivot*
- Найденные элементы меняются местами *swap(A[i], A[j])*
- Обмен происходит до тех пор, пока индексы i и j не пересекутся *i < j*

i++ *j--*

Note!

- Разбиение Хоара эффективнее разбиения Ломуто, т. к. происходит в среднем в 3 раза меньше обменов (swap) элементов, и разбиение эффективнее даже когда все элементы равны
- Следует заметить, что конечная позиция опорного элемента необязательно совпадает с его начальной позицией (в середине массива)

! Рассмотреть принцип сортировки с большим числом одинаковых значений

Разбиение Хоара: код

```
void quicksort(int a[n], int l, int r)
{
    int i = l;
    int j = r;
    int mid = (l + r) / 2;

    while (i <= j)
    {
        while (a[i] < mid)
        {
            i++;
        }
        while (a[j] > mid)
        {
            j--;
        }
        if (i <= j)
        {
            swap(a[i], a[j]);
            i++;
            j--;
        }
    }

    if (l < j)
    {
        quicksort(l, j);
    }
    if (r > i)
    {
        quicksort(i, r);
    }
}
```

итерация №

пробегаем по i

пробегаем по j

меняем неправильно стоящие элементы

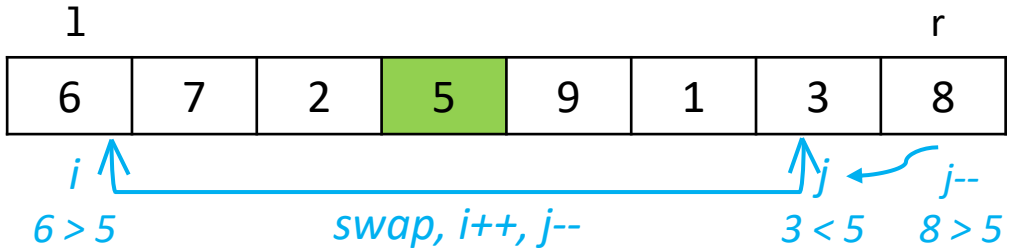
=>

3	1	2

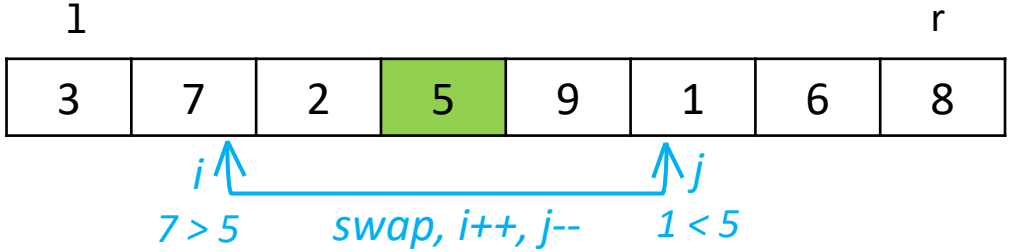
=>

9	7	6	8

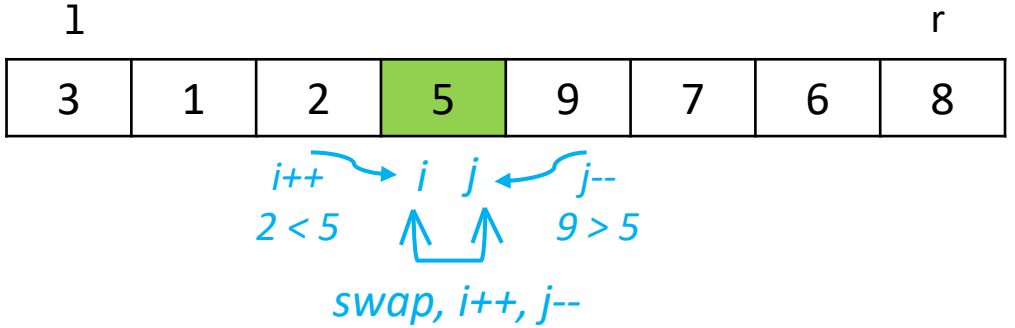
№ 1:



№ 2:



№ 3:



Итого a[i] = 9, a[j] = 2

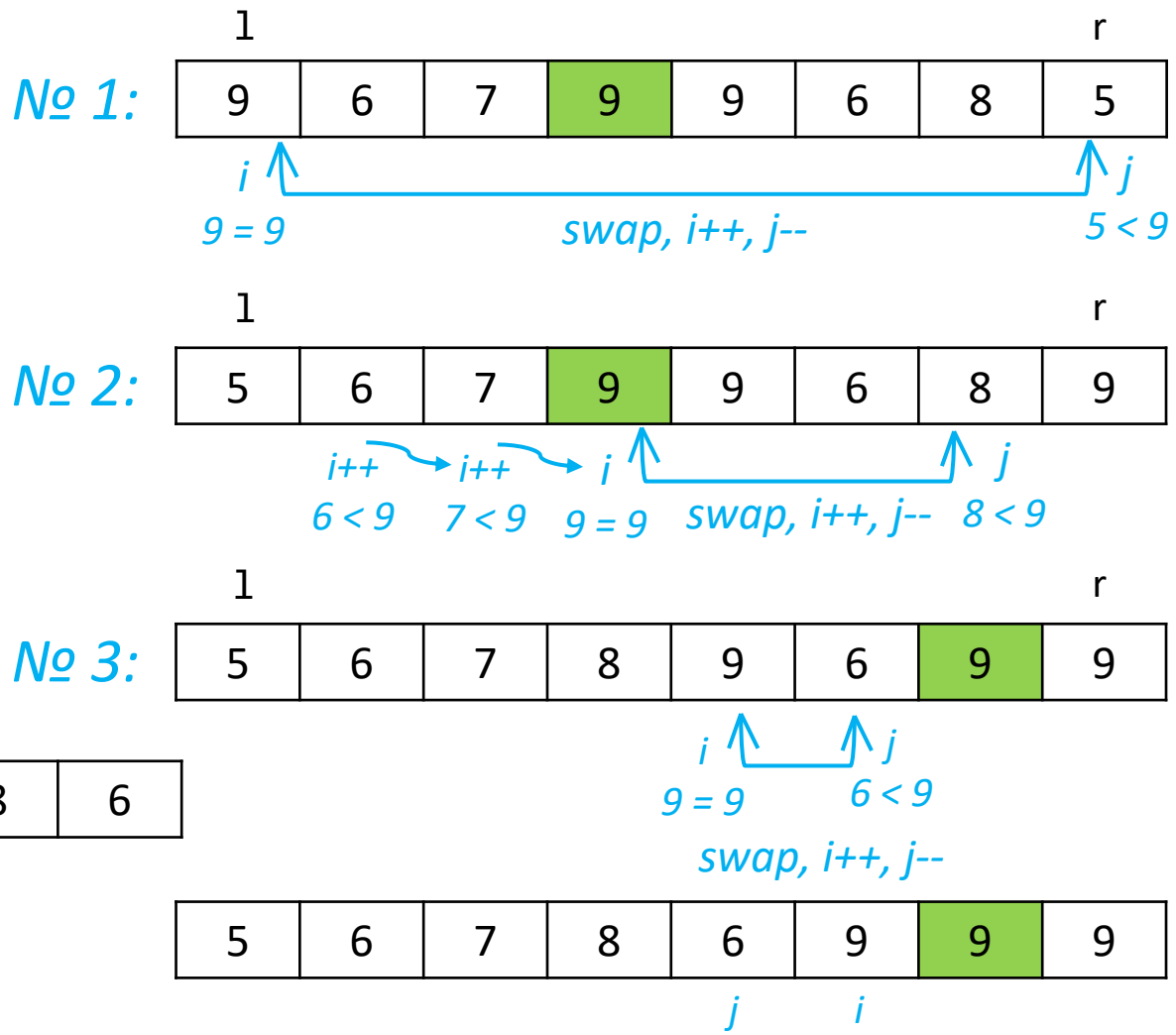
Запускаемся рекурсивно от левой и правой частей

```
void quicksort(int a[n], int l, int r)
{
    int i = l;
    int j = r;
    int mid = (l + r) / 2;

    while (i <= j)
    {
        while (a[i] < mid)
        {
            i++;
        }
        while (a[j] > mid)
        {
            j--;
        }
        if (i <= j)
        {
            swap(a[i], a[j]);
            i++;
            j--;
        }
    }

    if (l < j)
    {
        quicksort(l, j);
    }
    if (r > i)
    {
        quicksort(i, r);
    }
}
```

=> | 5 | 6 | 7 | 8 | 6 |

=> | 9 | 9 | 9 |

Еще один пример



Заметим, что опорный не обязательно остается на месте

Итого a[i] = 9, a[j] = 6

```
void quicksort(int a[n], int l, int r)
{
    int i = l;
    int j = r;
    int mid = (l + r) / 2;

    while (i <= j)
    {
        while (a[i] < mid)
        {
            i++;
        }
        while (a[j] > mid)
        {
            j--;
        }
        if (i <= j)
        {
            swap(a[i], a[j]);
            i++;
            j--;
        }
    }

    if (l < j)
    {
        quicksort(l, j);
    }
    if (r > i)
    {
        quicksort(i, r);
    }
}
```

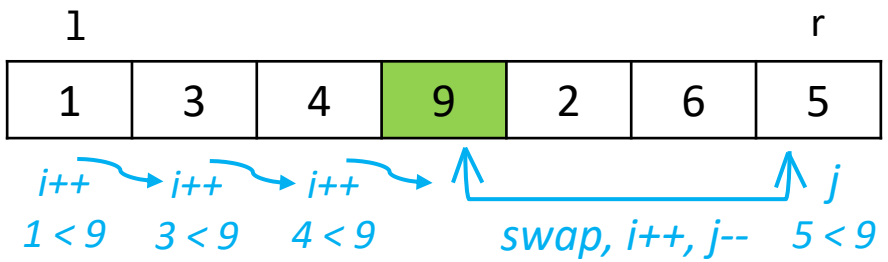
=> | 1 | 3 | 4 | 5 | 2 | 6 |

=> ничего не будет, т. к. *r = i*

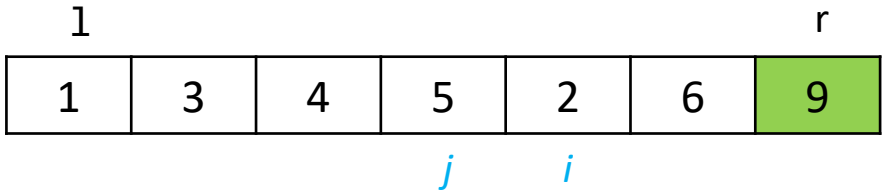
Таким образом мы уменьшили размер сортируемого массива всего на 1 элемент

Больше примеров

№ 1:



№ 2:

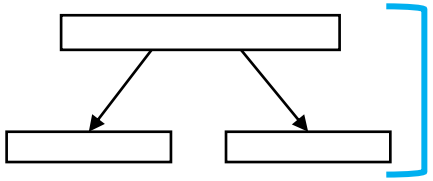


1. *i* дойдет до *r*
2. *a[j] > mid* никогда не выполнится
3. *i <= j* больше нет => swap больше не будет, и while закончится

Быстрая сортировка: оценка сложности

Время:

Лучший	Средний	Худший
O(n log n)	O(n log n)	$O(n^2)$



O(n) на просмотр массива

Глубина рекурсии – log n

=> O(n) * log n раз =

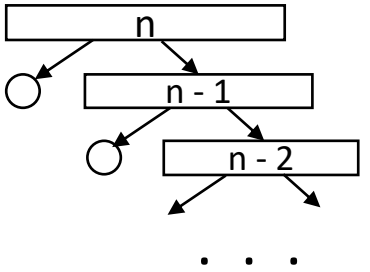
= O(n log n)

Память:

Лучший	Средний	Худший
O(log n)	O(log n)	O(n)

log n – глубина рекурсии => храним точки возврата

Глубина – n



Доказательство временной сложности

Рассмотрим **худший случай**: разбиение на *1* и *n - 1* элемент

$T(n) = O(1)$ при $n = 1$

$T(n) = T(n - 1) + \Theta(n) \Rightarrow \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$

$T(n - 1) = T(n - 2) + \Theta(n - 1)$

$T(n - 2) = T(n - 3) + \Theta(n - 2)$

. . .

$T(1) = O(1)$

> ***Note!*** Подробное доказательство в Кормене

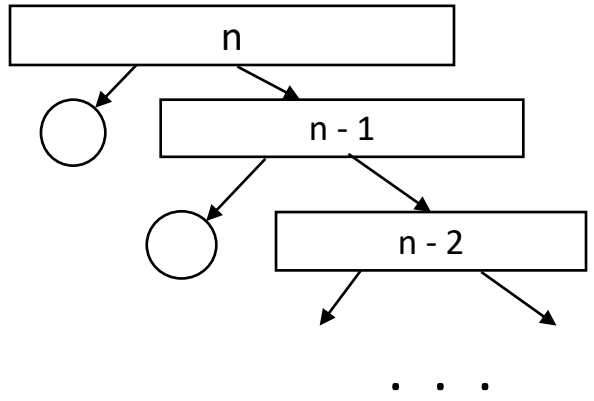
Средний и лучший: по аналогии с Merge Sort

$T(n) \rightarrow n \log n$ т. к. $T(n) \rightarrow \underbrace{2T(\frac{n}{2}) + \Theta(n)}$

Но разбиение не напополам

Доказательство сложности по памяти

```
void quicksort(int a[n], int l, int r)
{
    if (l < r)
    {
        int q = partition(a, l, r);

        quicksort(a, l, q);
        quicksort(a, q + 1, r);
    }
}
```



- Partition: ~O(1) на переменные (i и j)
- Рекурсия: для хранения точек возврата рекурсии в стеке ~O(log n) – глубина рекурсии
- В худшем случае массив разбивается ~n раз, а значит в стеке необходимо хранить O(n), т. к. глубина рекурсии увеличивается до O(n)

Быстрая сортировка: устойчивость

Исходный массив:

5	6	7	9_1	9_2	6	8	9_3

После разбиения Хоара:

5	6	7	8	6	9_2	9_1	9_3

Из примера видно, что 9 встали в порядке, отличном от изначального. На самом деле, оба разбиения не дают устойчивости.

Быстрая сортировка **НЕ устойчивая**!

Быстрая сортировка: модификации

Выбор опорного элемента должен давать стабильное распределение элементов в соотношении 50/50 относительно опорного

- Метод 1: выбор случайным образом

Худший (по времени)
$O(n^2)$

→ **O(log n)**

```
t = random(left, right)
mid = (left + right) / 2
x = a[t]
swap(a[t], a[mid])
```

Решение 1 не точное, но показатели хорошие у обоих

- Метод 2: используем среднее значение между крайними

```
x = (a[left] + a[right]) / 2
```

Худший (по памяти)
O(n)

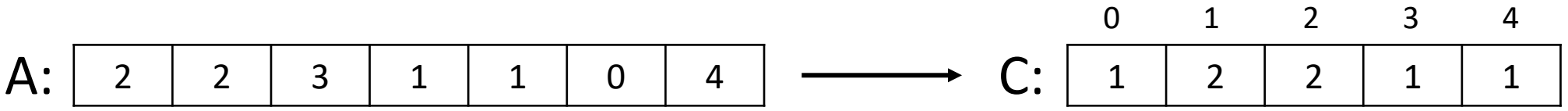
→ **O(log n)**

Линейные сортировки

работают за O(n)

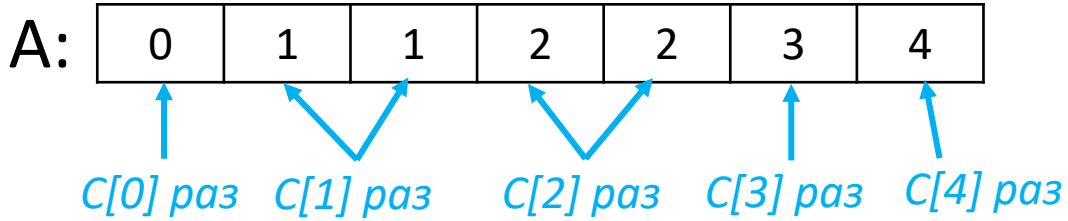
Сортировка подсчетом: принцип

Есть исходный массив ***A[n]***, состоящий из целых чисел от *0* до *k - 1*. Создадим массив ***C[k]*** для подсчета количества повторений числа из массива ***A***



встречаются элементы от 0 до 4 => k = 5

- Последовательно пройдем по массиву ***A*** и запишем в ***C[i]*** количество чисел равных *i*
- Теперь достаточно пройти по массиву ***C[i]*** и для каждого *i* (индекса в массиве ***C***) в массив ***A*** последовательно записать *i* столько раз, сколько оно встретилось, а именно ***C[i]*** раз



Сортировка подсчетом: код

```
void simple_counting_sort(int a[n])
{
    for (int number = 0; number < k; number++)
    {
        c[number] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        c[a[i]] = c[a[i]] + 1;
    }

    int pos = 0;
    for (int number = 0; number < k; number++)
    {
        for (int i = 0; i < c[number]; i++)
        {
            a[pos] = number;
            pos = pos + 1;
        }
    }
}
```

обнуляем C

C[i] раз записываем number

A[n], n = 9:

4	3	4	2	0	4	3	3	4

встречаются элементы от 0 до 4 => k = 5

Шаг 1: C:

0	1	2	3	4
0	0	0	0	0

обнуляем C[k] = 0

Шаг 2: C:

0	1	2	3	4
1	0	1	3	4

посчитаем число вхождений A[i], i = 0 … n – 1 идем по A C[A[i]]++

Шаг 3: A:

0	2	3	3	3	4	4	4	4

C:

0	1	2	3	4
1	0	1	3	4

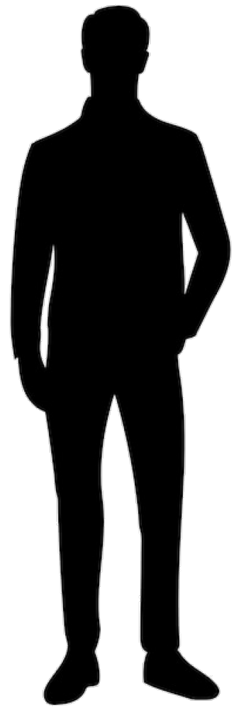
1 раз 0 раз 1 раз 3 раза 4 раза

расставим в A[i] нужные значения по возрастанию, каждый i по C[i] раз

Сортировать можно только целые числа?

Вспомним:

- Сортируют не просто значения, а объекты, которые обладают различными характеристиками – нужна применимость для сортировки сложных/комбинированных данных
- Нельзя просто перезаписать значения ключей, нужно пересортировать объекты



- *ФИ*
- *возраст*
- *курс*
- *группа*
- *код*

Пример: сортировать по

- *группе*
- *курсу*

↓

алфавит маленький

↓

подсчетом быстро

Решение проблемы:

Используем вспомогательный массив ***B*** для записи результата. С помощью массива ***C*** будем получать, какое значение из ***A*** куда нужно перенести в ***B***

Сортировка подсчетом для объектов (ver 2.0)

```
void counting_sort(int a[n])
{
    for (int i = 0; i < k; i++)
    {
        c[i] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        c[a[i]] = c[a[i]] + 1;
    }

    for (int i = 1; i < k; i++)
    {
        c[i] = c[i] + c[i - 1];
    }
    for (int i = n - 1; i >= 0; i--)
    {
        b[c[a[i]]] = a[i];
        c[a[i]]--;
    }
}
```

аналогично обычной

	1	2	3	4	5	6	7	8	9
A:	3	4	1	0	5	0	3	1	3

	0	1	2	3	4	5
C:	2	2	0	3	1	1

номер позиции в B для элемента из A (с какой позиции с конца нужно записать A[i] согласно сортировке)

записываем в B по порядку с конца элементы из A

вспомогательный массив, т. к. будем перемещать объекты

Сортировка подсчетом для объектов (ver 2.0)

```
void counting_sort(int a[n])
{
    for (int i = 0; i < k; i++)
    {
        c[i] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        c[a[i]] = c[a[i]] + 1;
    }

    for (int i = 1; i < k; i++)
    {
        c[i] = c[i] + c[i - 1];
    }
    for (int i = n - 1; i >= 0; i--)
    {
        b[c[a[i]]] = a[i];
        c[a[i]]--;
    }
}
```

	1	2	3	4	5	6	7	8	9
A:	3	4	1	0	5	0	3	1	3

	0	1	2	3	4	5
C:	2	4	4	7	8	9

C[i] – индекс последнего элемента со значением i в отсортированном массиве (при нумерации с 1)

индекс первого элемента со значением i вычисляется как C[i - 1] + 1 (в случае первого элемента начало массива)

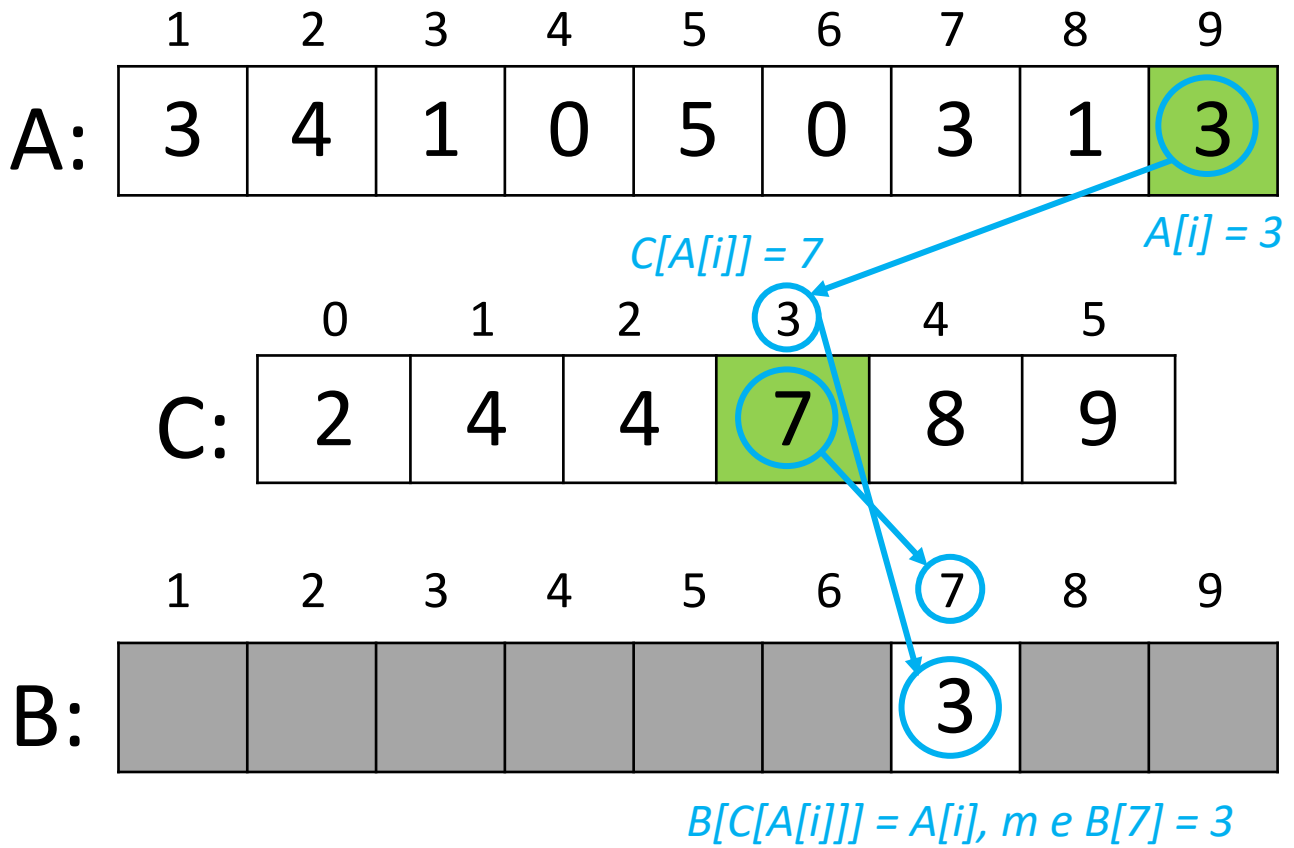
На каких позициях в отсортированном массиве должны стоять числа 0…5:

0	1	2	3	4	5
1 - 2	3 - 4	5 - 4 *то есть таких нет*	5 - 7	8 - 8	9 - 9

Сортировка подсчетом для объектов (ver 2.0)

```
for (int i = n - 1; i >= 0; i--)
{
    b[c[a[i]]] = a[i];
    c[a[i]]--;
}
```

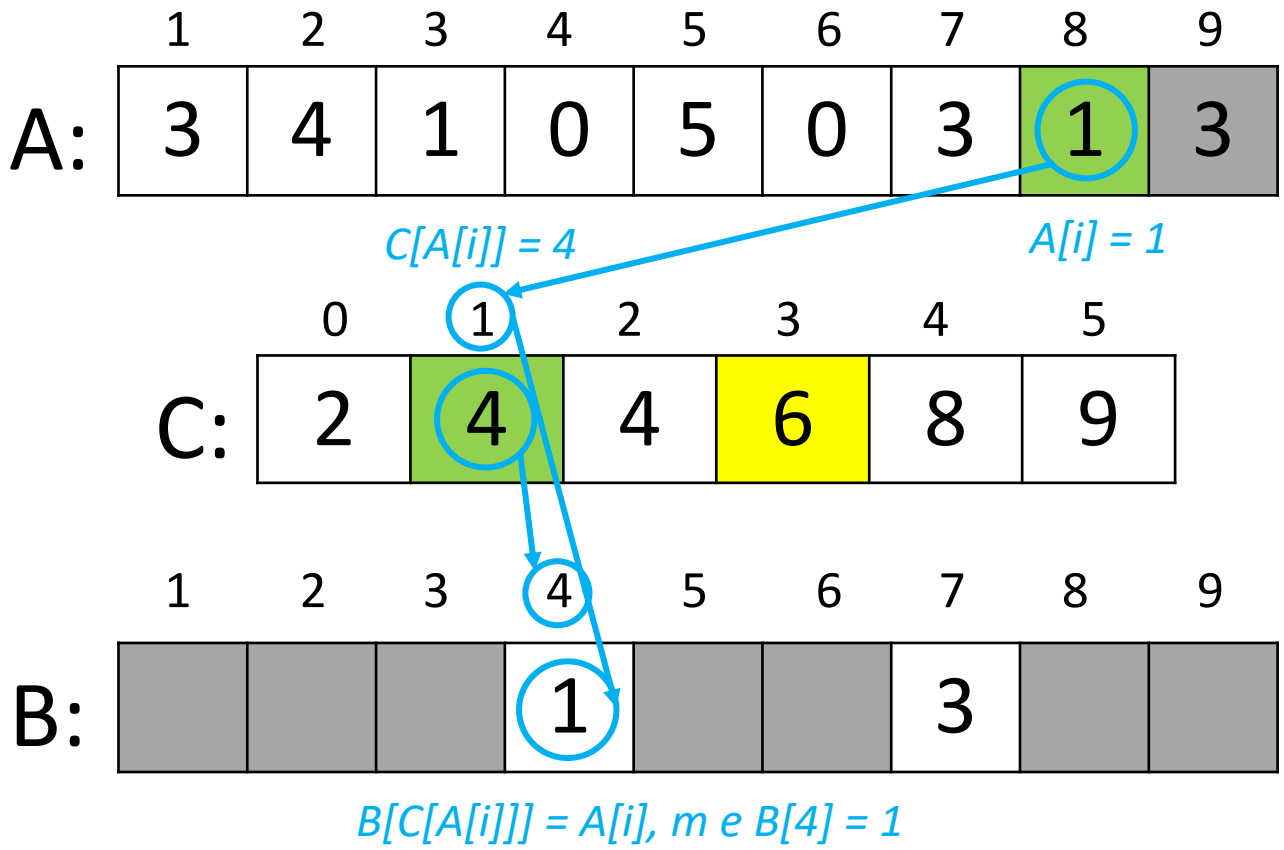
Проходимся с конца по массиву A и, используя вычисленные в массиве C границы, расставляем элементы из A в B на нужные места



Сортировка подсчетом для объектов (ver 2.0)

```
for (int i = n - 1; i >= 0; i--)
{
    b[c[a[i]]] = a[i];
    c[a[i]]--;
}
```

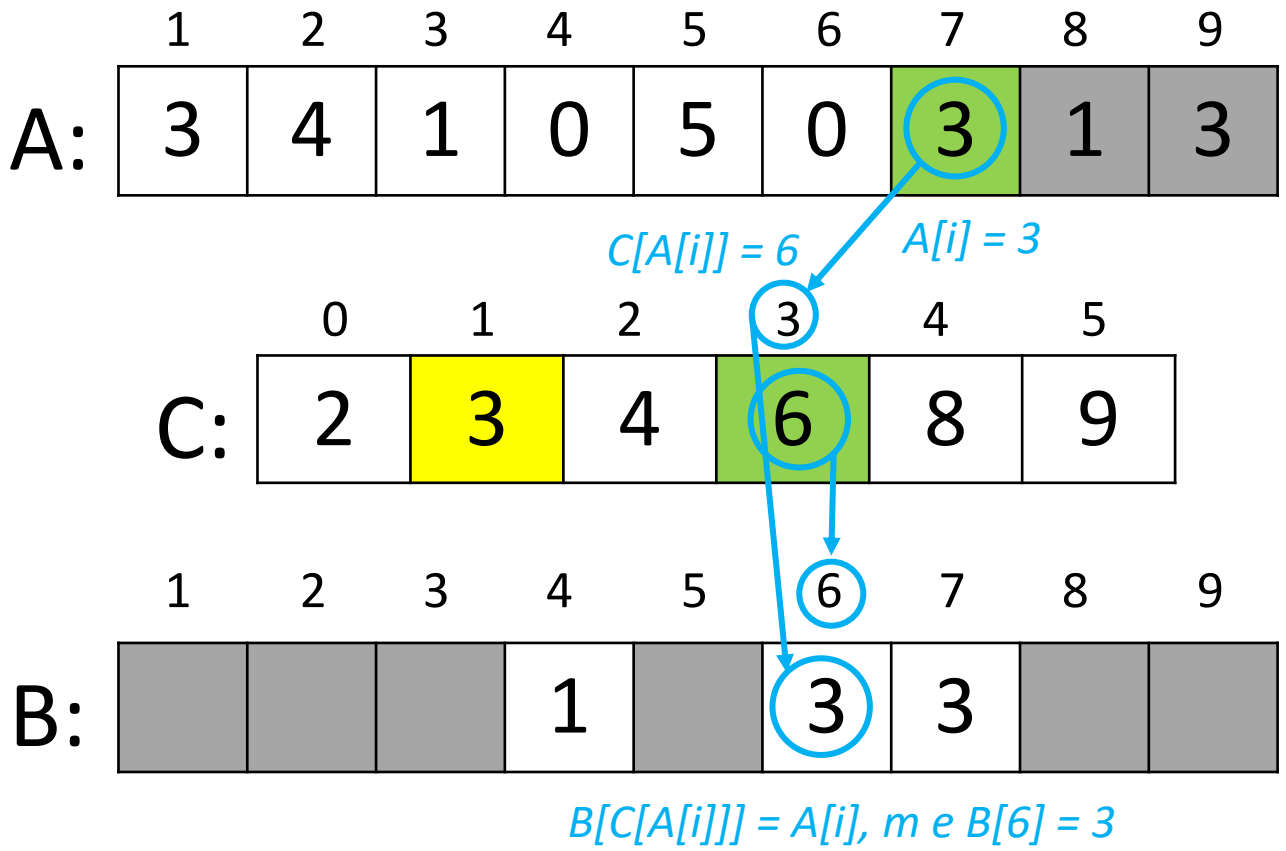
Проходимся с конца по массиву A и, используя вычисленные в массиве C границы, расставляем элементы из A в B на нужные места



Сортировка подсчетом для объектов (ver 2.0)

```
for (int i = n - 1; i >= 0; i--)
{
    b[c[a[i]]] = a[i];
    c[a[i]]--;
}
```

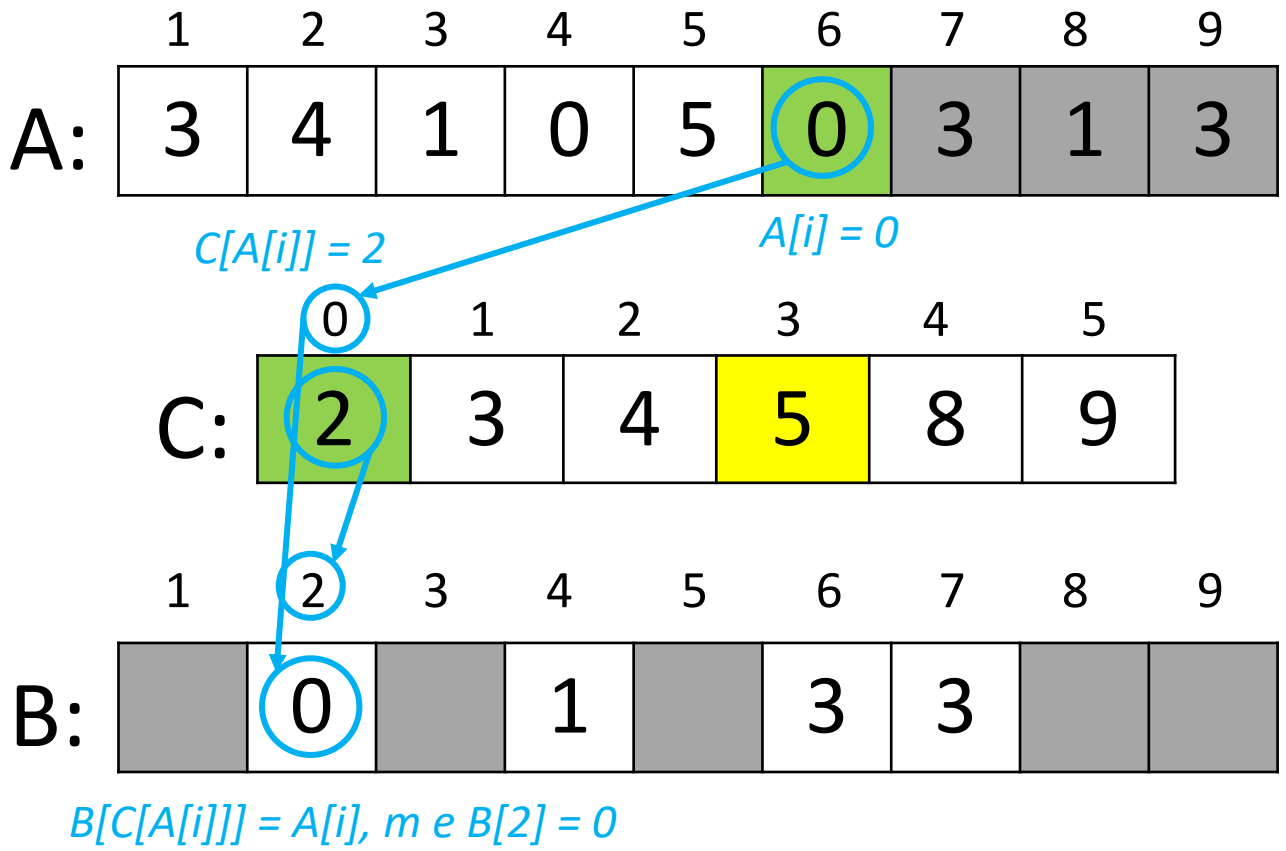
Проходимся с конца по массиву A и, используя вычисленные в массиве C границы, расставляем элементы из A в B на нужные места



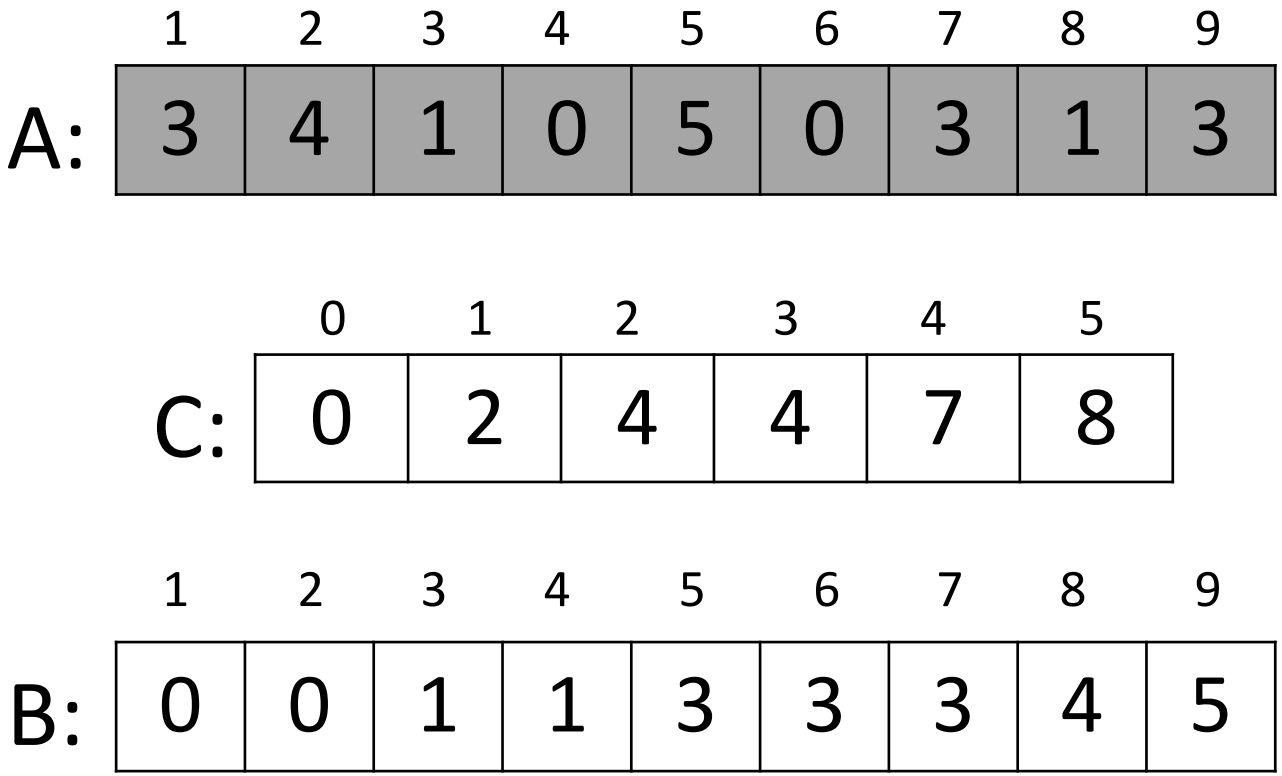
Сортировка подсчетом для объектов (ver 2.0)

```
for (int i = n - 1; i >= 0; i--)
{
    b[c[a[i]]] = a[i];
    c[a[i]]--;
}
```

Проходимся с конца по массиву A и, используя вычисленные в массиве C границы, расставляем элементы из A в B на нужные места



Сортировка подсчетом для объектов (ver 2.0)



Итого получили отсортированный массив B

Сортировка подсчетом: оценка сложности

Время:

Лучший	Средний	Худший
O(k + n)	O(k + n)	O(k + n)

ver 1.0

Лучший	Средний	Худший
O(n)	O(n)	O(n)

ver 2.0

На практике сортировку подсчетом имеет смысл применять, если $k = O(n)$, поэтому можно считать время работы алгоритма равным $\Theta(2n) = \Theta(n)$

Память:

Лучший	Средний	Худший
O(k)	O(k)	O(k)

ver 1.0

Лучший	Средний	Худший
O(k + n)	O(k + n)	O(k + n)

ver 2.0

Доказательство временной сложности 1.0

```
void simple_counting_sort(int a[n])
{
    for (int number = 0; number < k; number++)
    {
        c[number] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        c[a[i]] = c[a[i]] + 1;
    }

    int pos = 0;
    for (int number = 0; number < k; number++)
    {
        for (int i = 0; i < c[number]; i++)
        {
            a[pos] = number;
            pos = pos + 1;
        }
    }
}
```

+1 +1 +1 — C[k] => +k

k итераций k + 1

+2 +1 +4

n итераций 4n + 2

+1 +1 +2 +1 +2 +1

+2

проход по всему A: n итераций

3n + 2k + 2

k + 4 -> O(k)

k -> n => O(n)

$T(n) = 7n + 3k + 6 \sim O(n + k)$

если $k \to n$, $T(n) \sim O(2n) = O(n)$

Доказательство временной сложности 2.0

```
void counting_sort(int a[n])
{
    for (int i = 0; i < k; i++)
    {
        c[i] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        c[a[i]] = c[a[i]] + 1;
    }

    for (int i = 1; i < k; i++)
    {
        c[i] = c[i] + c[i - 1];
    }
    for (int i = n - 1; i >= 0; i--)
    {
        b[c[a[i]]] = a[i];
        c[a[i]]--;
    }
}
```

k k + 1

n 4n + 2

k 3k + 1

n 5n + 2

$$T(n) = 9n + 4k + 6 \sim O(n + k)$$

$$\text{если } k \to n,\ T(n) \sim O(2n) = O(n)$$

Сортировка подсчетом: устойчивость

- Первая реализация **НЕ устойчивая**!

В данном случае реализация простая и наивная и работает только с целыми числами, а так как мы перезаписываем числа, ни о какой устойчивости речи не идет

- Вторая реализация **устойчивая**!

Мы расставляем объекты с одинаковыми значениями ключа сортировки по их исходным позициям относительно друг друга

Сортировка подсчетом: модификации

- **Неизвестен диапазон чисел**

Если диапазон значений не известен заранее, то его можно найти с помощью линейного поиска минимума и максимума в исходном массиве, что не повлияет на асимптотику алгоритма, так как поиск выполняется за O(n).

- **Отрицательные числа**

Нужно учитывать, что минимум может быть отрицательным, в то время как в массиве C индексы от 0 до k−1. Поэтому при работе с массивом C из исходного A[i] необходимо вычитать минимум, а при обратной записи в B[i] прибавлять его.

Цифровая сортировка: принцип

Имеем множество последовательностей одинаковой длины (не обязательно из чисел), состоящих из элементов, которые могут быть сравнимы между собой. Требуется отсортировать эти последовательности в лексикографическом порядке

Будем называть элементы, из которых состоят сортируемые объекты, разрядами. Тогда алгоритм состоит в последовательной сортировке объектов какой-либо устойчивой сортировкой по каждому разряду, в порядке от младшего разряда к старшему

- для чисел уже существует понятие разряда
- для строк можно в качестве разрядов использовать отдельные символы, которые сравниваются по таблице кодировок

Цифровая сортировка: код

```
radix_sort(A, d)
{
    for (i = 1 ... d)
    {
        // выполнить любую устойчивую сортировку
        // массива A по цифре i
    }
}
```

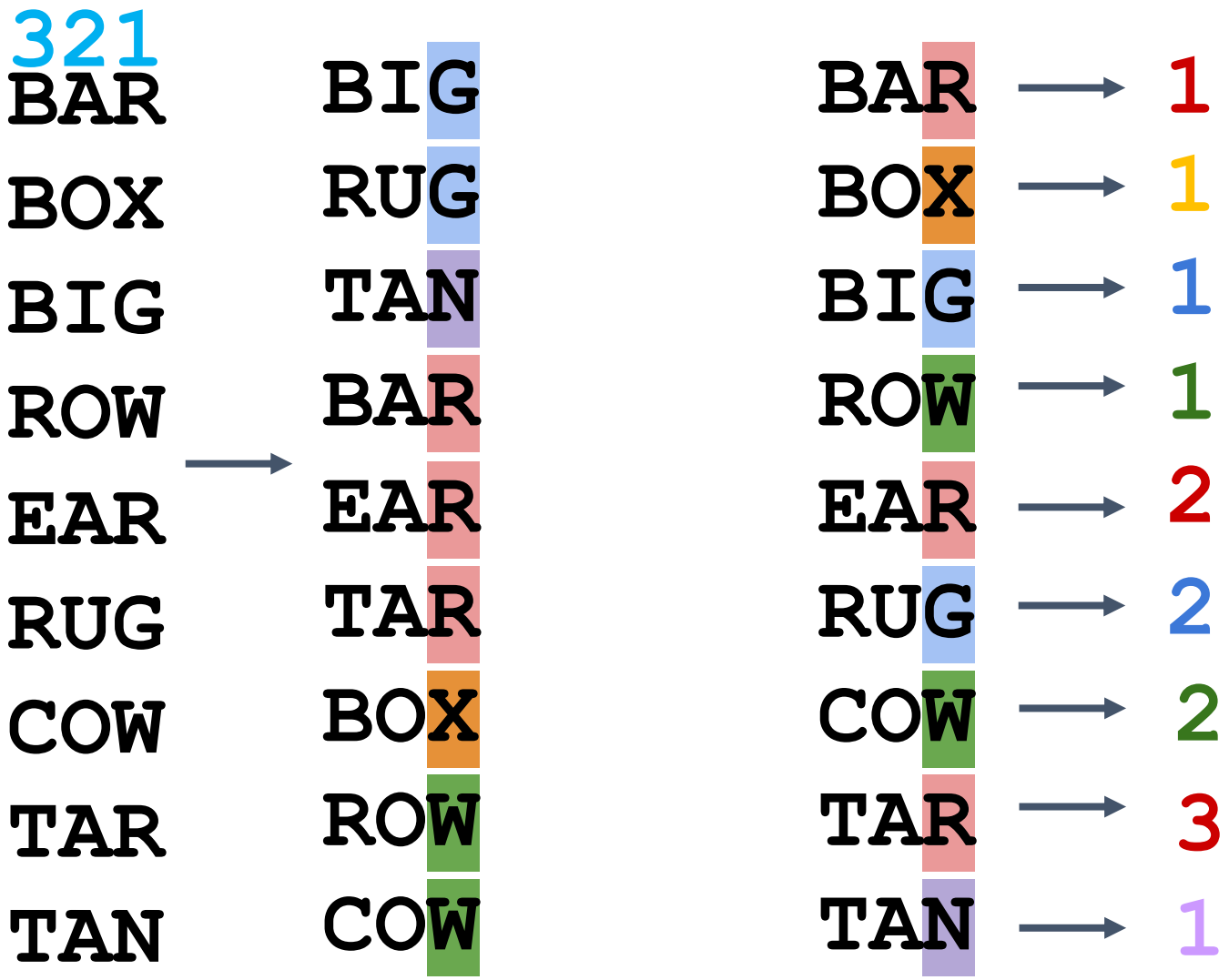
разряды для сортировки

321

BAR
BOX
BIG
ROW
EAR
RUG
COW
TAR
TAN

Хотим отсортировать следующие строки

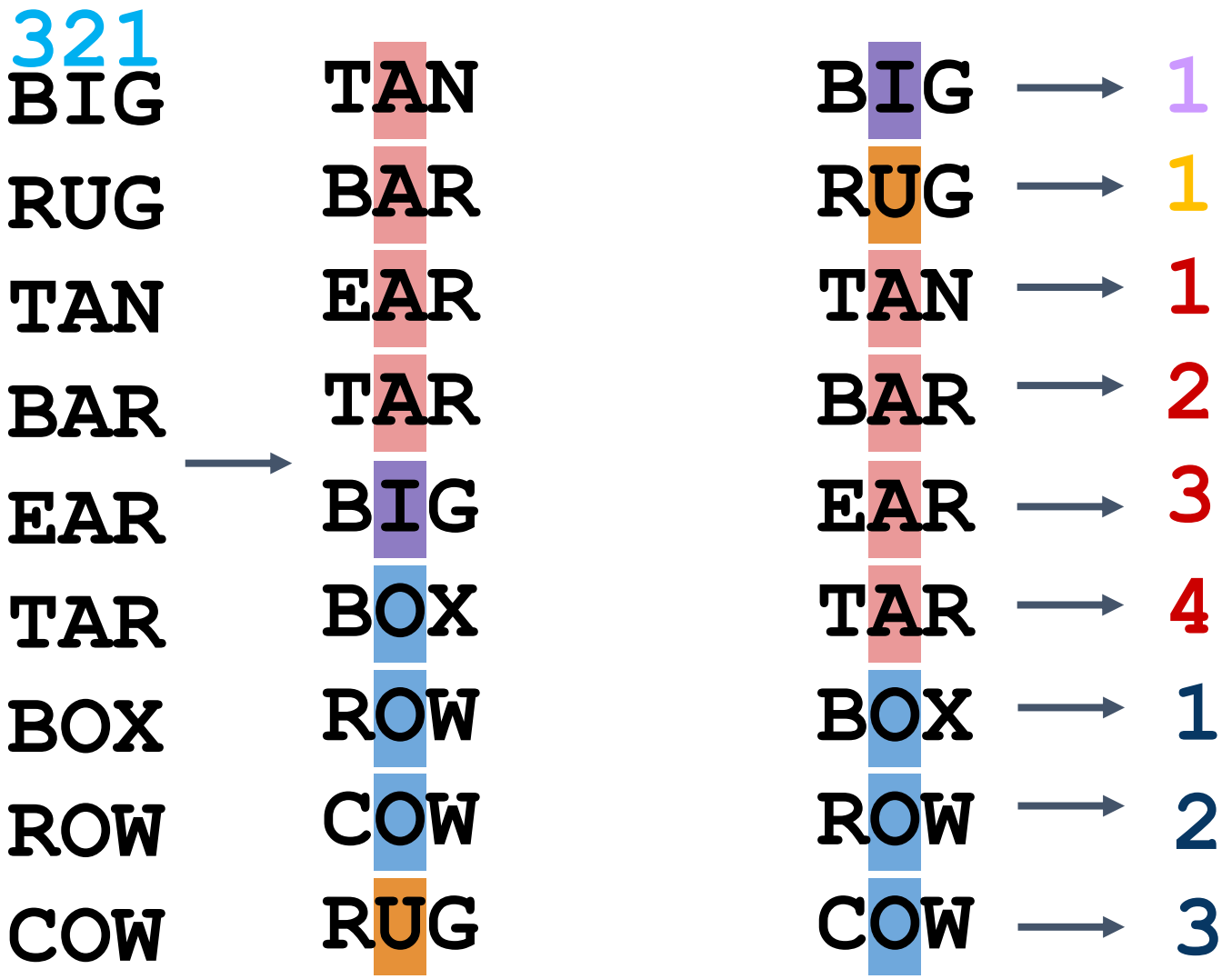
Цифровая сортировка: пример



Отсортировали по первому разряду

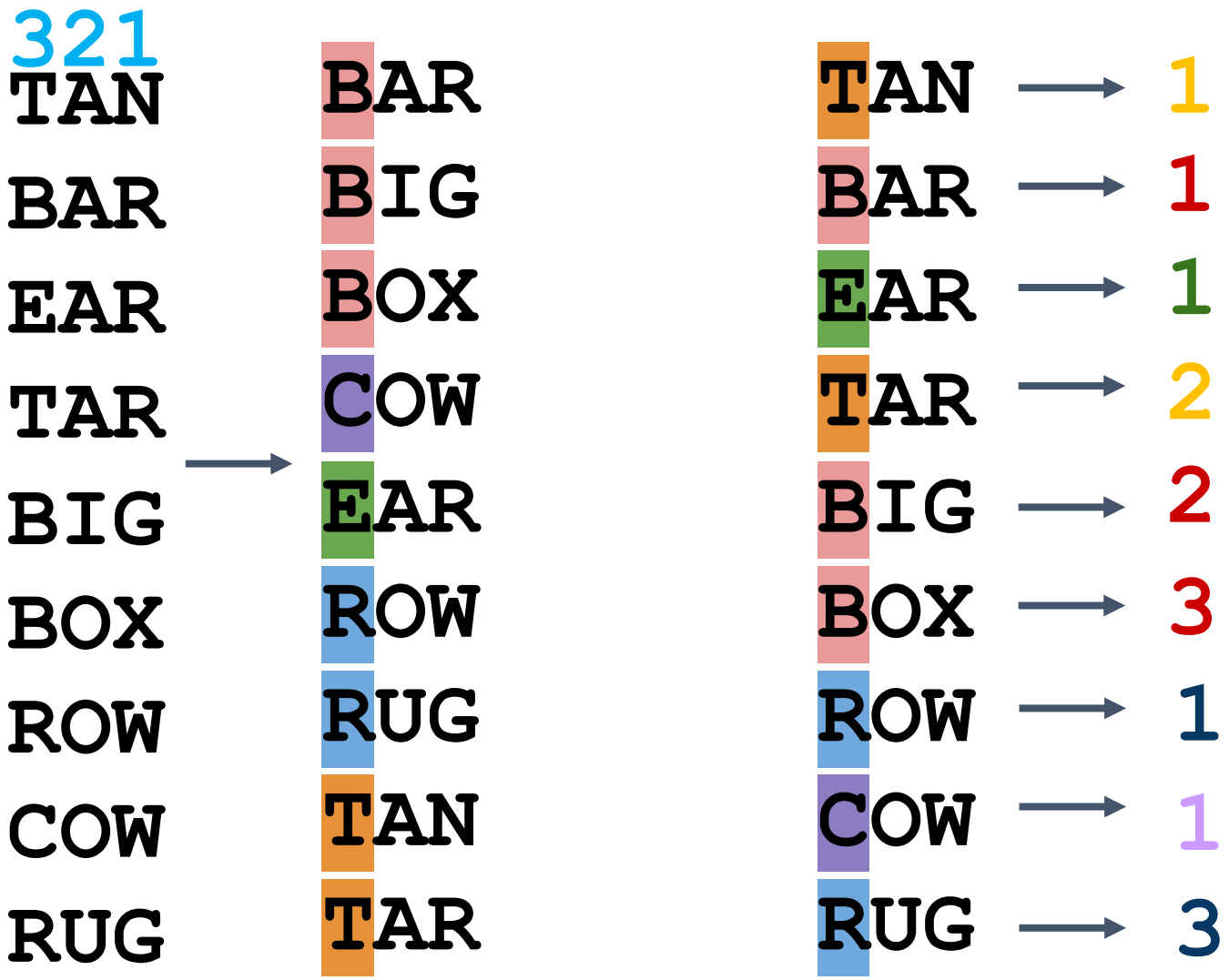
Note! Обязательно используем линейные сортировки (например, подсчетом). На ограниченном алфавите покажет себя отлично: английский алфавит или цифры – алфавиты с небольшой мощностью

Цифровая сортировка: пример



Отсортировали по второму разряду

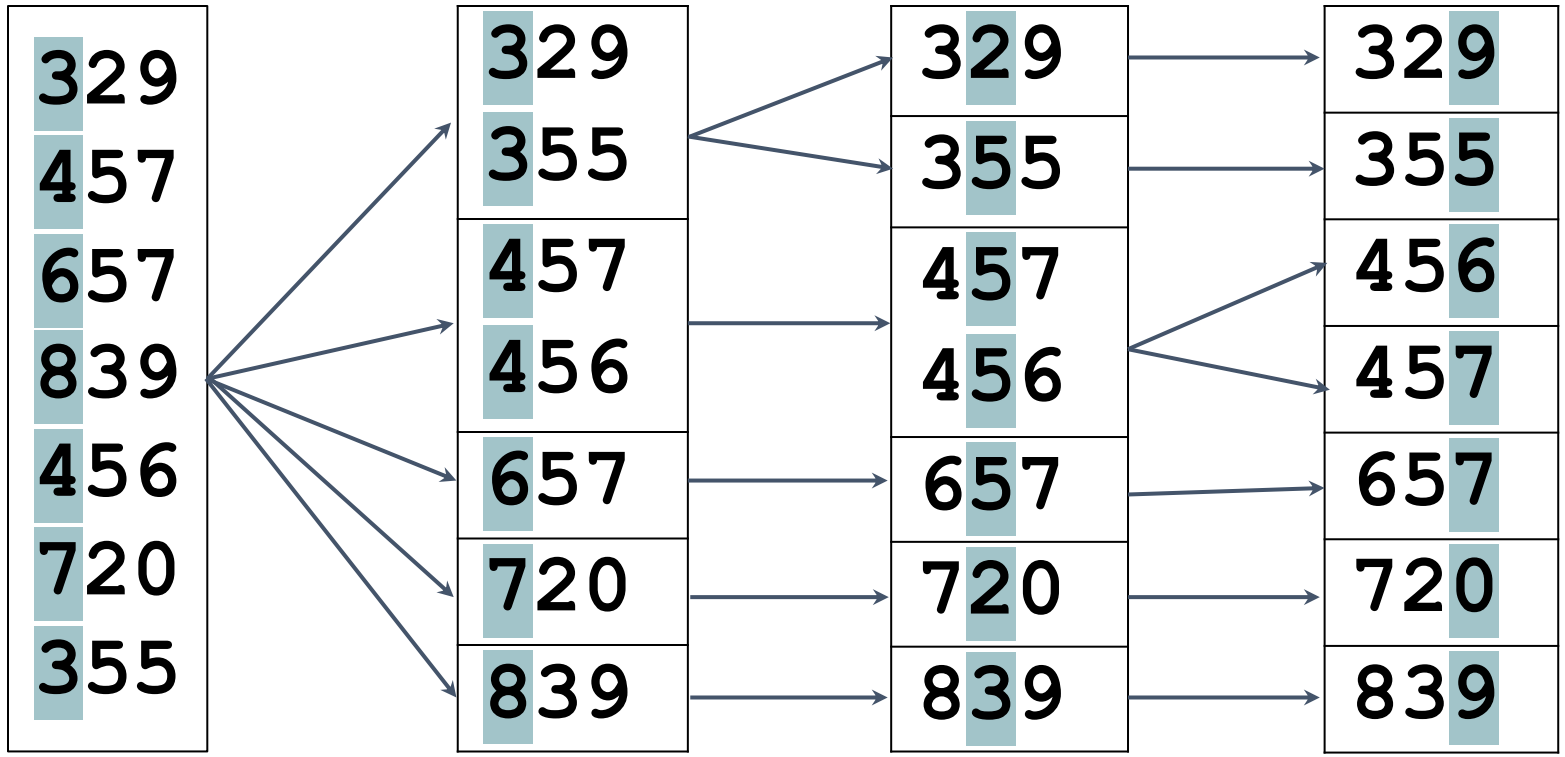
Цифровая сортировка: пример



Отсортировали по третьему разряду

Получился лексикографический порядок

Цифровая сортировка: пример с числами



При сортировке бьем на блоки одинаковых и сортируем уже внутри блоков отдельно, сохраняя их порядок

Что если использовать Merge Sort по разрядам?

$$O(d * T(n)) = O(d * n \log n) = O(n^2)$$

число разрядов

скорость сортировки разрядов

Цифровая сортировка: оценка сложности

d – число разрядов, **T(n)** – время работы устойчивой сортировки, в данной оценке используем сортировку подсчетом

Время:

Лучший	Средний	Худший
O(d * T(n))	O(d * T(n))	O(d * T(n))

Лучший	Средний	Худший
O(d * n)	O(d * n)	O(d * n)

Память:

Лучший	Средний	Худший
O(M(n))	O(M(n))	O(M(n))

Лучший	Средний	Худший
O(k + n)	O(k + n)	O(k + n)