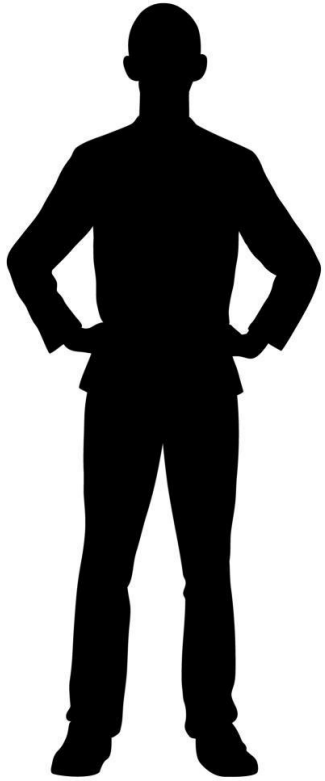
Сортировки

упорядочить известное

Как описать одного студента



- Имя
- Курс
- Группа
- Возраст
- Код ИСУ

Характеристика	Тип
Имя	char [100]
Курс	int
Группа	char [5]
Возраст	int
Код ИСУ	int

```
// структура для описания студента
struct student {
        char name [50];
        int course;
        char group[6];
        int age;
        int code_isu;
};
```

Вспомним студента ИТМО

- Часто приходится отбирать объекты из набора по какой - то характеристике:
  - здесь студенты 1 курса

Вспомним студента ИТМО

- Часто приходится отбирать объекты из набора по какой - то характеристике:
  - здесь студенты 1 курса: **.course=1**
  - здесь только 5 групп:

Вспомним студента ИТМО

- Часто приходится отбирать объекты из набора по какой - то характеристике:
  - здесь студенты 1 курса: **.course=1**
  - здесь только 5 групп: **.group in {05,06,07,08,09}**
  - здесь скорее всего возраст

Вспомним студента ИТМО

- Часто приходится отбирать объекты из набора по какой - то характеристике:
  - здесь студенты 1 курса: **.course=1**
  - здесь только 5 групп: **.group in {05,06,07,08,09}**
  - здесь скорее всего возраст: **.age>16 and <23**

Вспомним студента ИТМО

- Часто приходится отбирать объекты из набора по какой - то характеристике:
  - здесь студенты 1 курса: **.course=1**
  - здесь только 5 групп: **.group in {05,06,07,08,09}**
  - здесь скорее всего возраст: **.age>16 and <23**
- Выбрав нужных студентов, мы можем принять экзамен – поставив оценку и затем выбрать тех, кто не сдал

Вспомним студента ИТМО

- Часто приходится отбирать объекты из набора по какой - то характеристике:
  - здесь студенты 1 курса: **.course=1**
  - здесь только 5 групп: **.group in {05,06,07,08,09}**
  - здесь скорее всего возраст: **.age>16 and <23**
- Выбрав нужных студентов, мы можем принять экзамен – поставив оценку и затем выбрать тех, кто не сдал

Упорядочить выборку данных

- **КАК:** По убыванию /по возрастанию (по двум полям: объекты с одинаковыми значениями по первому полю сортируются по второму)
- **ЧТО:** Числа/строки/объекты/даты и т.д.

<u>Упорядочим нашу выборку:</u>

- По имени (по алфавиту – по убыванию)
- По баллам по дисциплине (по убыванию)
- По группе и по алфавиту (внутри групп по алфавиту)

Сортировка вставками

Отсортированная часть – набор упорядоченных элементов

Пример: 3 5 7 9 2 1 8 11 0

0	1	2	3	4	5	6	7	8
3	5	7	9	2	1	8	11	0

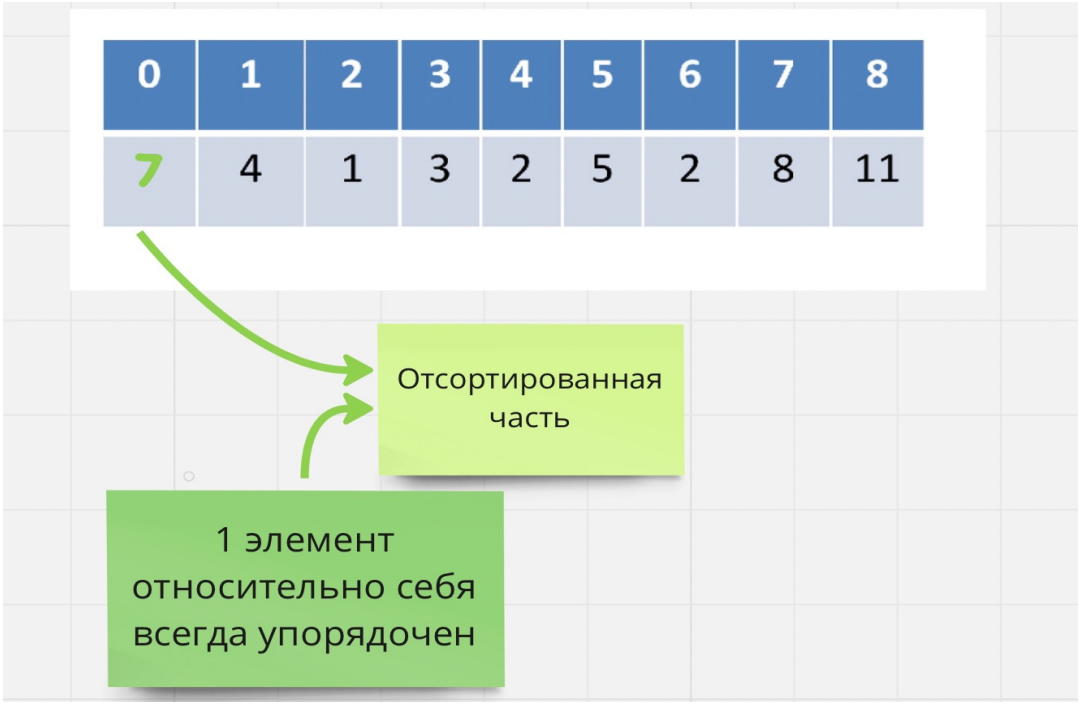
От A[0] по A[3] - отсортированная часть массива

От A[4] по A[8] - НЕотсортированная часть массива

Сортировка вставками: принцип

Исходный набор

0	1	2	3	4	5	6	7	8
7	4	1	3	2	5	2	8	11



Сортировка вставками: принцип

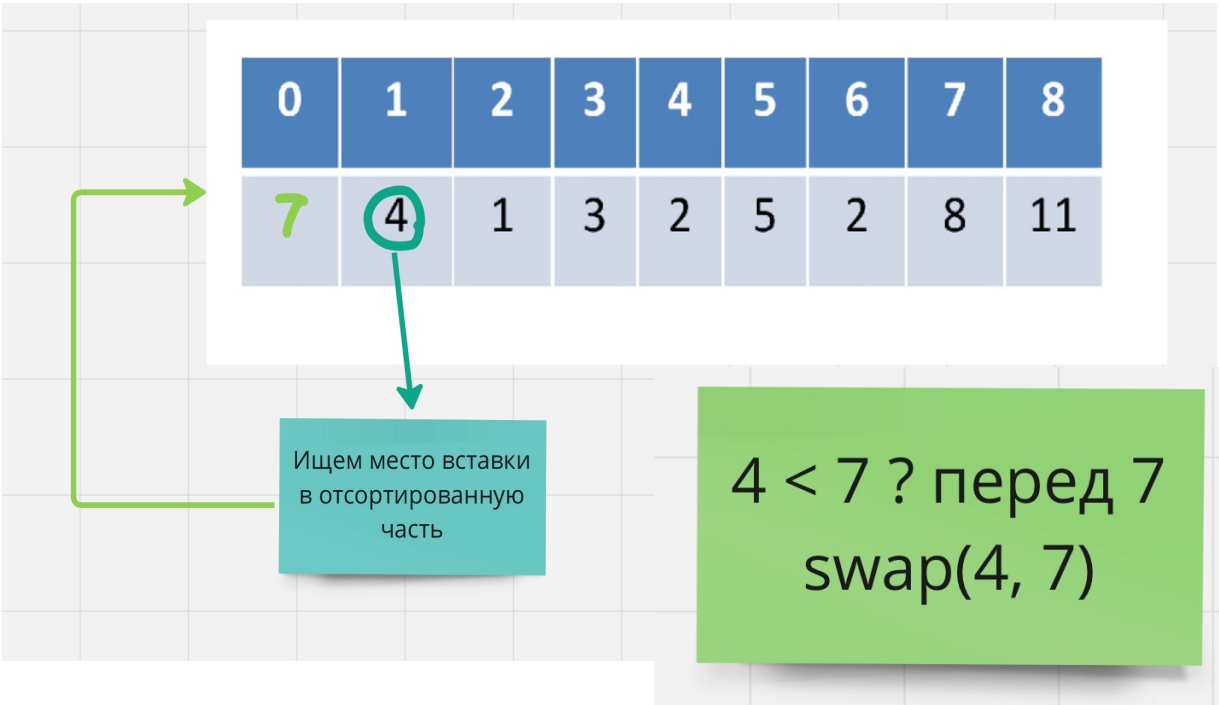
- Первый элемент считаем отсортированной частью
- Начиная со второго:
  - просматриваем элементы слева направо от i к i+1
  - выполняем вставку просматриваемого элемента в отсортированную часть
  - после вставки увеличиваем размер отсортированной части

Сортировка вставками: принцип

Первый элемент считаем отсортированной частью

Начиная со второго:

- просматриваем элементы слева направо от i к i+1
- выполняем вставку просматриваемого элемента в отсортированную часть
- после вставки увеличиваем размер отсортированной части



Просматриваем весь массив от A[1] до A[n – 1]

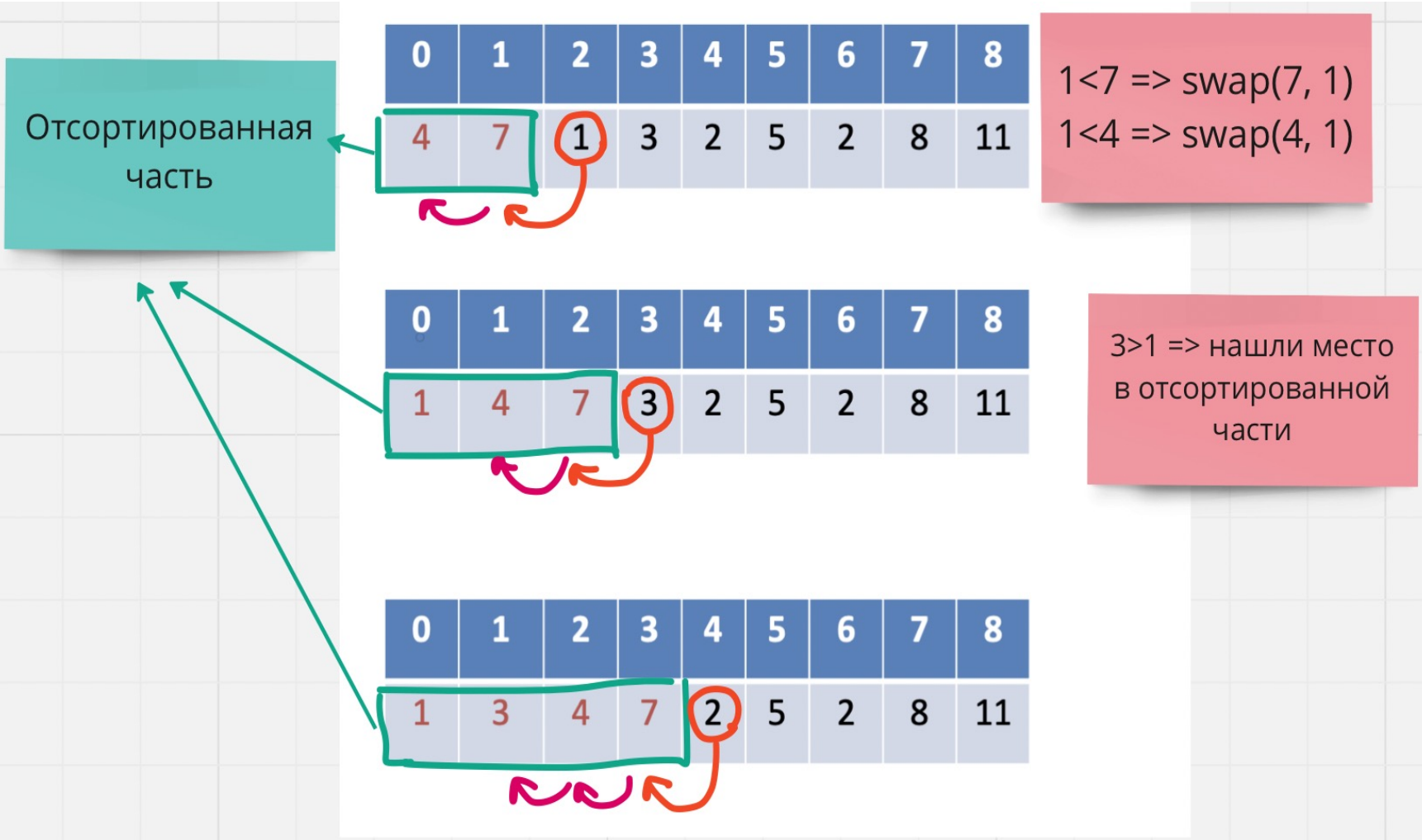
1<7 => swap(7, 1)
1<4 => swap(4, 1)

3>1 => нашли место в отсортированной части

2<7 => swap(7,2)

2<4 => swap(2,4)

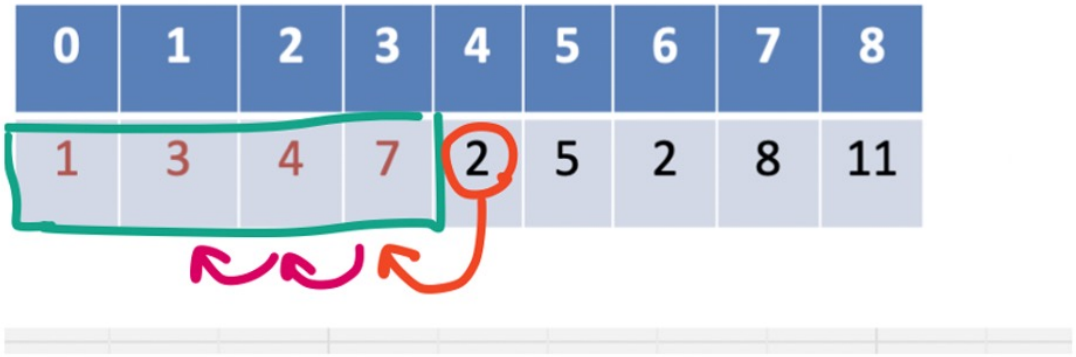
2< 3 => swap(2,3)


Отсортированная часть
0 1 2 3 4 5 6 7 8
4 7 1 3 2 5 2 8 11
1<7 => swap(7, 1)
1<4 => swap(4, 1)
0 1 2 3 4 5 6 7 8
1 4 7 3 2 5 2 8 11
3>1 => нашли место в отсортированной части
0 1 2 3 4 5 6 7 8
1 3 4 7 2 5 2 8 11

Что внутри каждого swap?

- a <-> b

$$\begin{cases} x = a \\ a = b \\ b = x \end{cases}$$

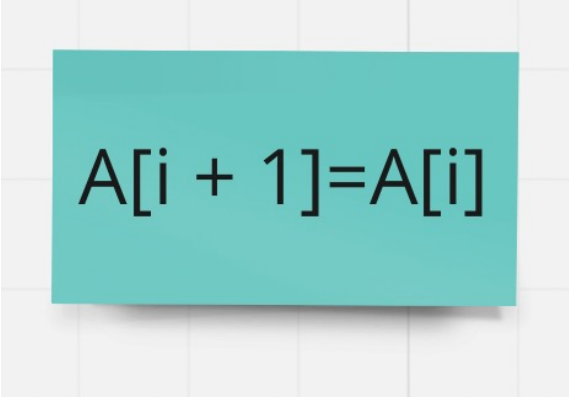


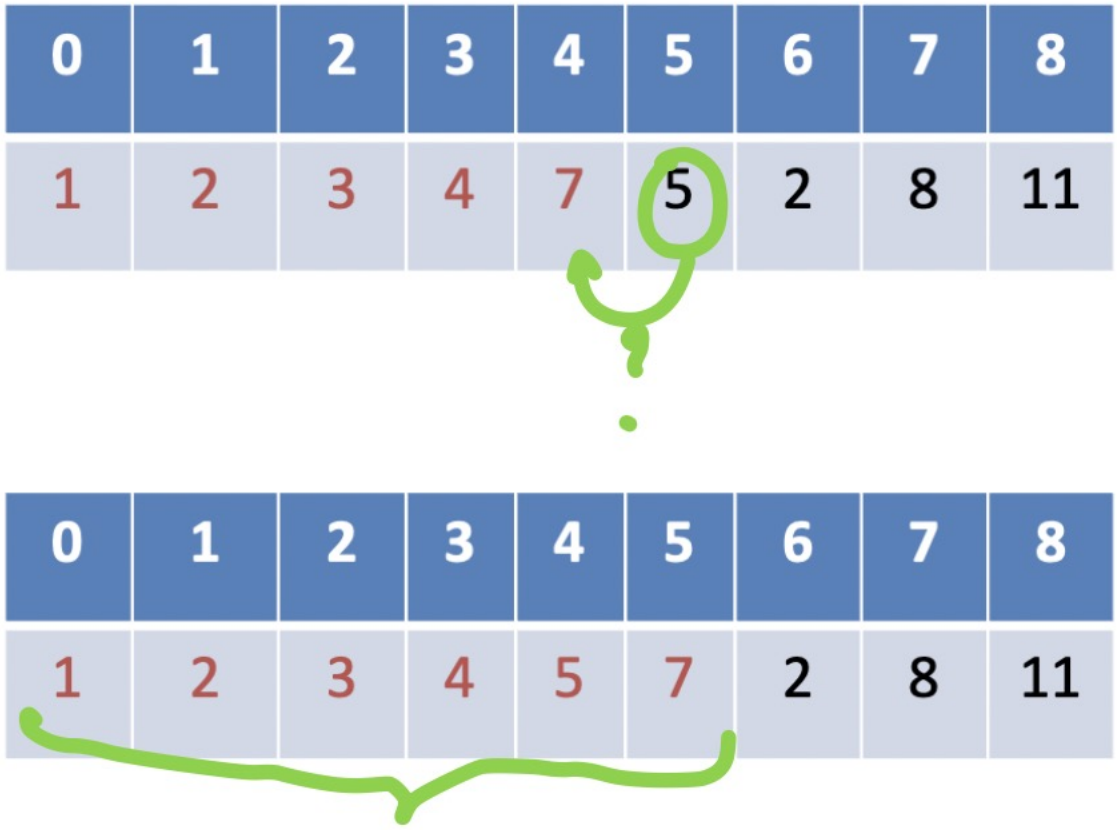
3 * swap = 9 операций

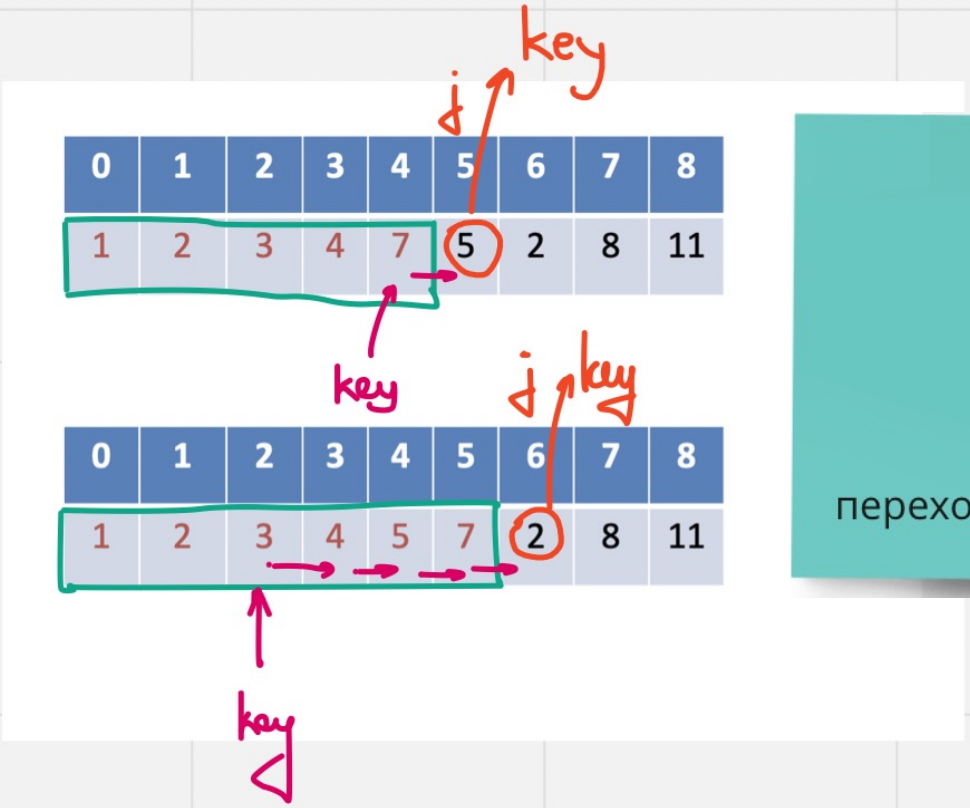
Как оптимальнее?

Как оптимальнее?

- Сохраняем вставленный элемент перед вставками i = j - 1
- Сдвигаем значение пока вставляемый элемент меньше A[i+1] = A[i]
- Вставляем элемент на нужную позицию A[i+1] = key




0 1 2 3 4 5 6 7 8
1 2 3 4 7 5 2 8 11
0 1 2 3 4 5 6 7 8
1 2 3 4 5 7 2 8 11
Отсортированная часть



Сохраняем key = A[j]
i = j -1
Пока key < A[i]
A[i + 1] = A[i]
i = i -1
A[i + 1] = key
переходим к следующему элементу j++

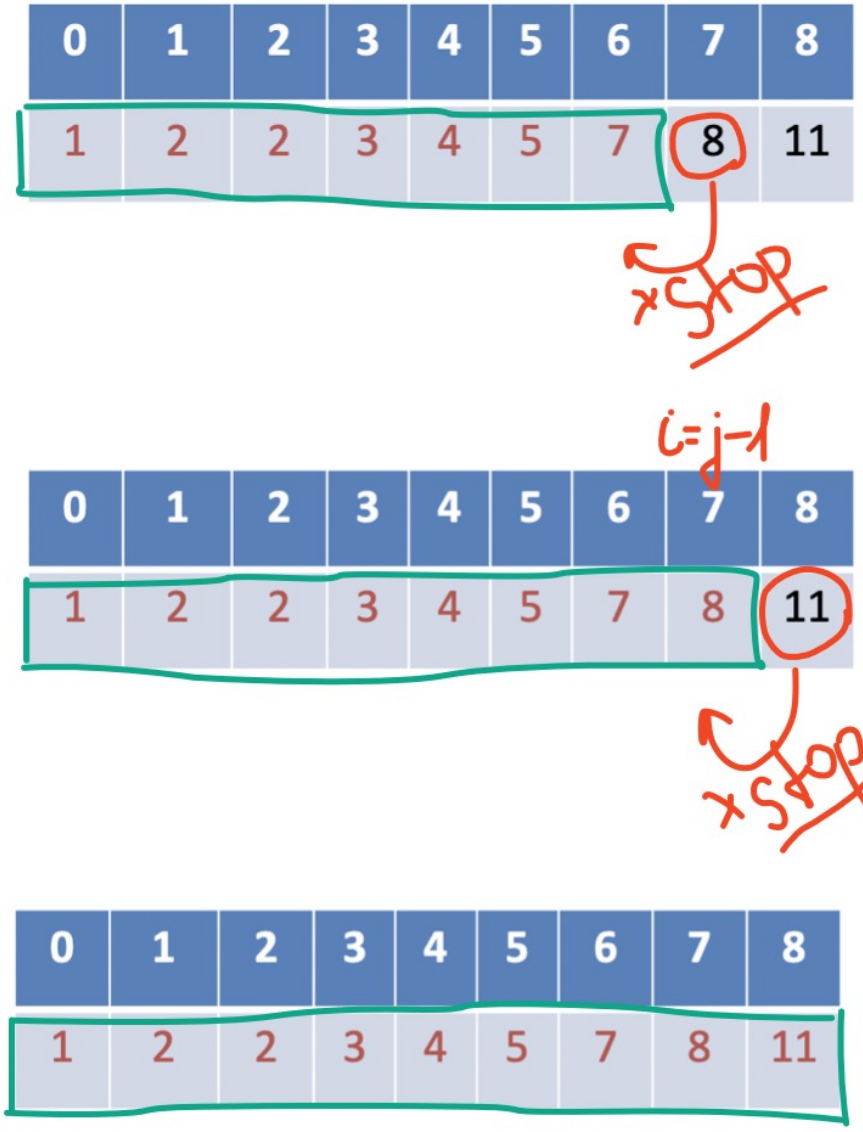
Вместо 4 swap = 12 операций, всего 4 операции сдвига
В три раза меньше!

8>7
если вставляемый элемент больше последнего отсортированного => увеличиваем границу отсортированной части

A[i] > key (условие)
почему не в A[j] ?
Потому что в A[j]
будем сдвигать при
поиске места key

Все!

0	1	2	3	4	5	6	7	8
1	2	2	3	4	5	7	8	11



8>7
если вставляемый элемент больше последнего отсортированного => увеличиваем границу отсортированной части

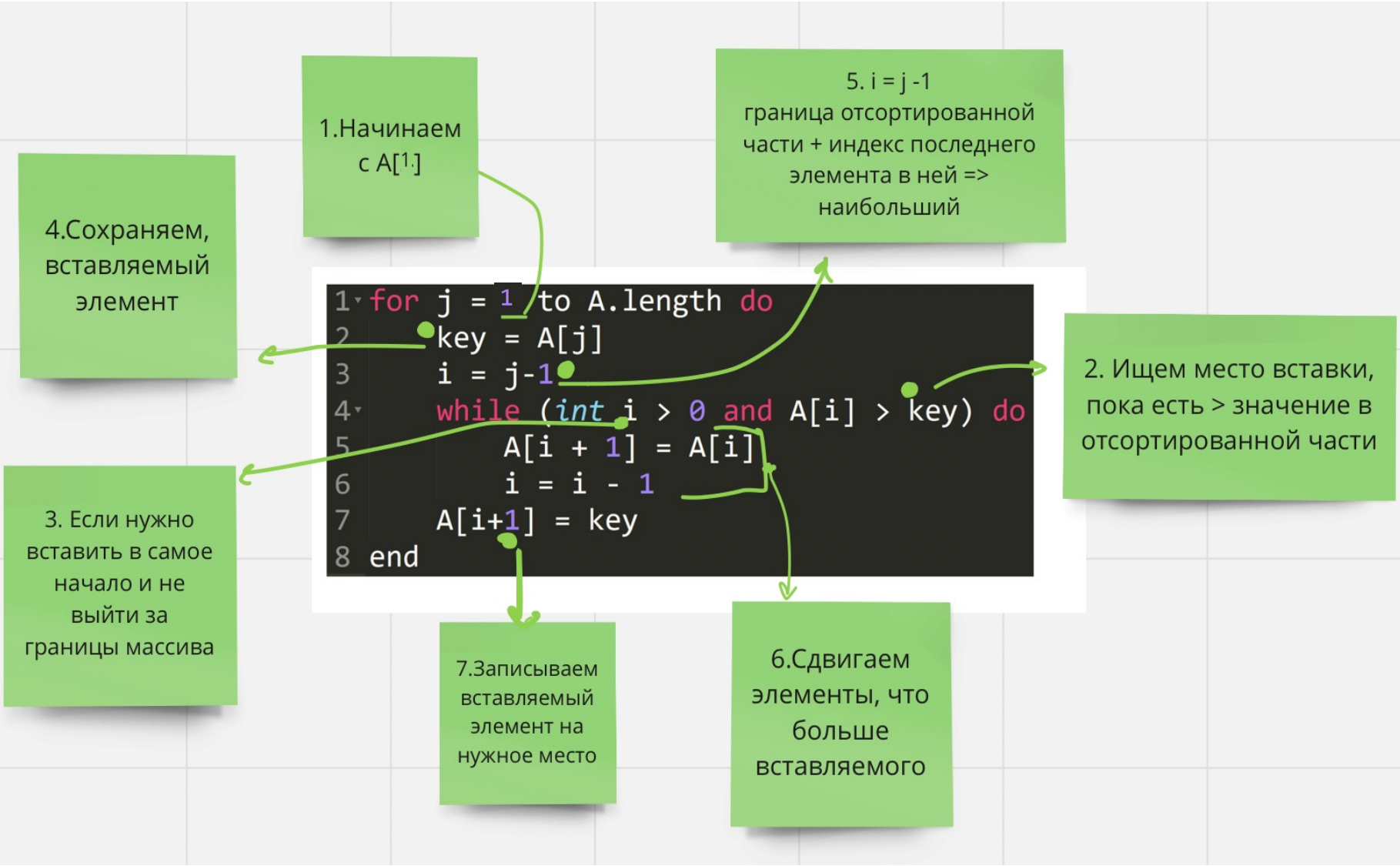
A[i] > key (условие)
почему не в A[j] ?
Потому что в A[j] будем сдвигать при поиске места key

Мы прошли весь массив, сортировка закончена!

Сортировка вставками: код

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

Сортировка вставками: код



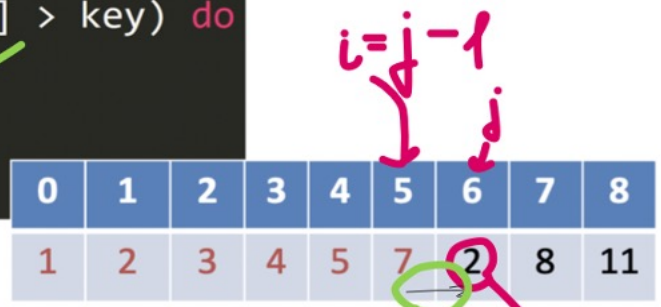
Сортировка вставками: код

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

0	1	2	3	4	5	6	7	8
1	2	3	4	5	7	2	8	11

Сортировка вставками: код

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



i = i -1
i = 4
i + 1 = 5
key = 2 < A[i] = 5
A[i+ 1] = A[i]
A[5] = A[4]

i = 5
i +1 = 6
A[i + 1] = A[i]
2 <- 7

A: 1 2 3 4 5 5 7 8 11

key = 2

A: 1 2 3 4 5 7 7 8 11

key = 2

Оценим по времени и памяти: код

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

По памяти 3 переменные => M(n) = 3 ~ O(1)

По времени T(n) = (1 + 2 + 2) * (n – 1) + $\sum_{j=2}^{n} j(3 + 2 +$

$2) = 5n - 5 + \sum_{j+2}^{n} 7\,\mathrm{j} = 5\mathrm{n} - 5 + \frac{7*(2+n)*n}{2}$ ~ O(n^2)

Анализ алгоритма

Рассматриваем наш алгоритм со всех сторон, подбираем все возможные наборы данных для его проверки.
Выделяем закономерности поведения алгоритма в зависимости от входных данных!

Лучший	Средний	Худший
- Это время, за которое отрабатывает алгоритм в самом лучшем случае! Подбираем набор данных такой, что алгоритм даст наилучший результат по времени работы и оцениваем асимптотически временную функцию! Фиксируем закономерность данных для лучшего случая!	Как работает алгоритм в большинстве случаев! <u>ВАЖНО:</u> 1) Оценки в лучшем, среднем и худшем могут между собой совпадает 2) Лучший и худший случаи могут иметь низкую частоту возникновения – ключевой оценкой будет работа в среднем	- Это время, за которое отрабатывает алгоритм в самом худшем случае! Подбираем набор данных, при котором алгоритм дает худшее время работы! Фиксируем закономерность! Даем асимптотическую оценку времени!

Анализ алгоритма

Лучший	Средний	Худший
10%	70%	20%

Оценка работы алгоритма в среднем будет самой показательной, но стоит рассматривать все оценки, так как процент худших случаев велик!

Лучший	Средний	Худший
3%	30%	67%

Рассматриваем оценку худшего случая как основную, так как худшие случаи максимально частотные

Лучший	Средний	Худший
	100%	

В данном случае алгоритм всегда работает одинаково!

Лучший	Средний	Худший
10%	**90%**	

Аналогично анализируется Алгоритм и по ПАМЯТИ!

Сортировка ВСТАВКАМИ

ПО ВРЕМЕНИ

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

ПО ПАМЯТИ

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

В данном случае алгоритм всегда работает одинаково!

Сортировка ВСТАВКАМИ

ПО ВРЕМЕНИ

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

Если массив отсортирован, то внутрь **while** мы не зайдем не разу => **for j = 2 to n ~ O(n)**

$\sum_{j=2}^{n} 7j \text{ - } > Cn^2 \text{ - } > 0(n^2)$ Но в среднем коэффициент C не большой!

ПО ПАМЯТИ

Лучший	Средний	Худший
O(1)	O(1)	O(1)

В данном случае алгоритм всегда работает одинаково!

Без разницы со swap или сдвигом мы используем только переменные << n => O(1)

Инвариант цикла

Инвариант цикла – это логическое выражение, зависящее от задачи, решаемой при помощи рассматриваемого цикла.

Инварианты циклов используются в теории верификации (проверки) алгоритмов для доказательства правильности выполнения цикла.

Инварианты корректных алгоритмов обладают следующими свойствами:

Инициализация. Инвариант справедлив перед инициализацией цикла

Сохранение. Если инвариант выполняется перед очередной итерацией цикла, то он также выполняется после нее.

Завершение. По завершении цикла инварианты позволяют убедиться в корректности алгоритма.

Доказательство

Инвариант: на j-й итерации цикла массив A[1…(*j*–1)] состоит из исходных элементов, расположенных в порядке возрастания.

Инициализация. На первой итерации $j = 2$, массив A[1…1] состоит из одного исходного элемента, расположенного по возрастанию.

Сохранение. на *j*-й итерации элементы массива $A[j–1]$, A[j–2] … $A[j–m]$ сдвигаются до тех пор, пока не будет найдено место для $A[j]$, куда он и будет помещен. Инвариант – истинный.

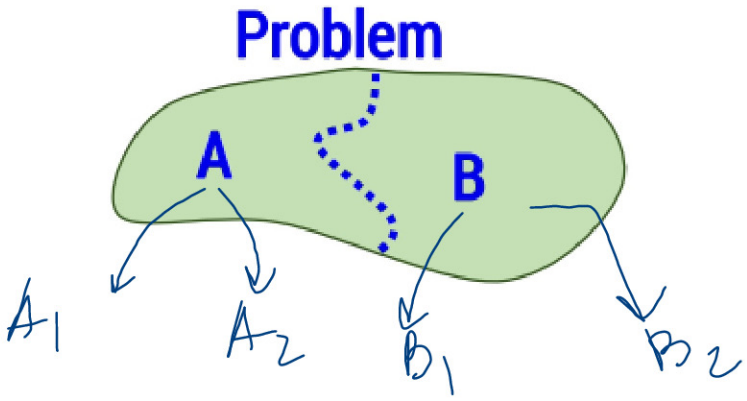
```
INSERTION_SORT(A)
1  for j ← 2 to len(A) do
2      k ← A[j]
3      // comment
4      i ← j – 1
5      while i > 0  и A[i] > k do
6          A[i+1] = A[i]
7          i ← i – 1
8      A[i+1] = k
```

Завершение. На *j*+1 итерации массив A[1…j] состоит из исходных элементов, расположенных по возрастанию.

Метод декомпозиции: разделяй и властвуй

1. Задача разбивается на несколько меньших экземпляров той же задачи
2. Решаются сформированные меньшие экземпляры задачи (обычно рекурсивно)
3. При необходимости решение исходной задачи формируется как комбинация решений меньших экземпляров задачи

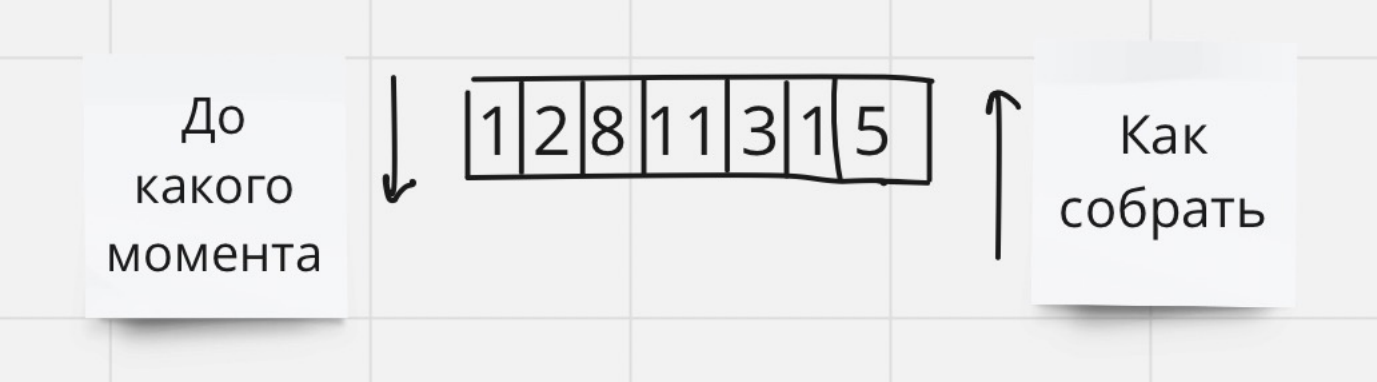
Problem = SubProblemA + SubProblemB



Метод декомпозиции: особенности

1. Верно выбрать **размер** подзадач, до которого идет разбиение (1 элемент –> отсортирован)
2. После решения подзадач необходимо корректно выполнить **сбор** решения всей задачи
3. **Рекурсия** эффективный инструмент для данного подхода: например, на спуске идет разбиение, а на возврате решение подзадач и сбор единого решение (вариантов много)

Рассмотрим на простом примере:
Найти сумму чисел



Метод декомпозиции: пример

Вычислить сумму чисел: $a0$, $a1$, . . . , $an-1$.

$$a_0 + a_1 + \cdots + a_{n-1} =$$

$$(a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1})$$

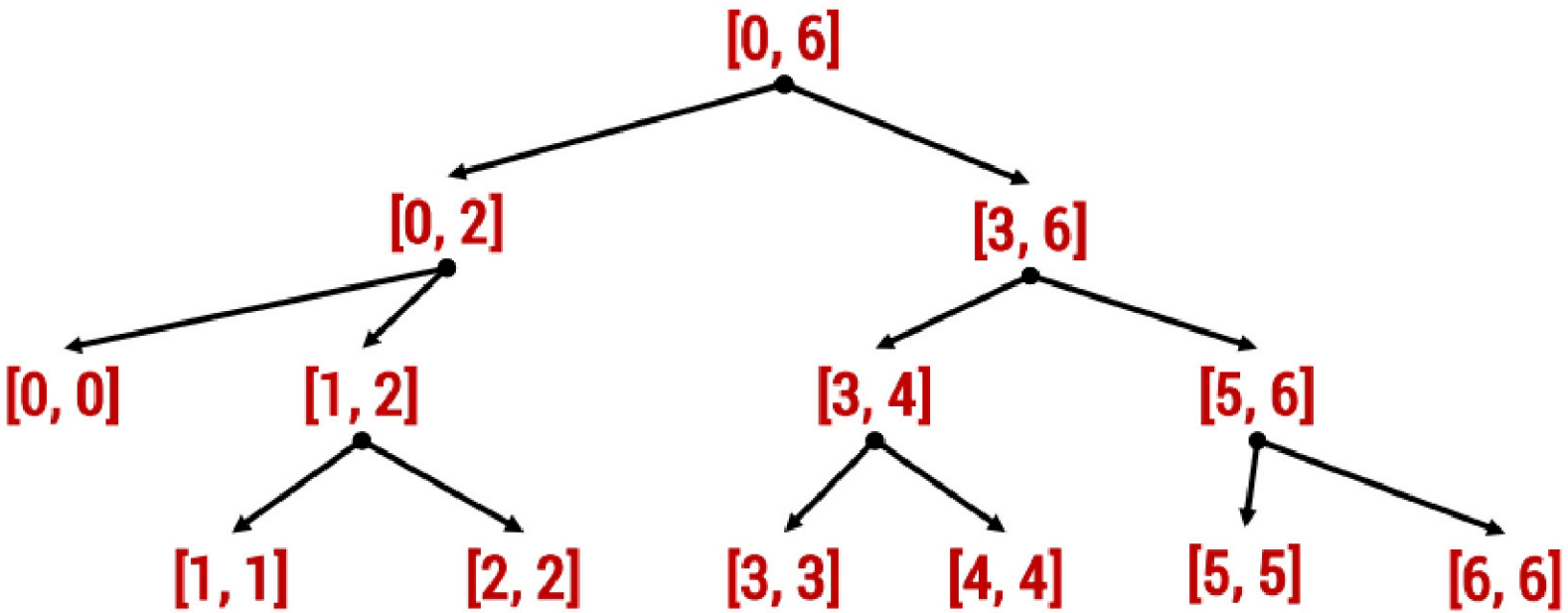
$$4 + 5 + 1 + 9 + 13 + 11 + 7 =$$

$$(4 + 5 + 1) + (9 + 13 + 11 + 7) =$$

$$= ((4) + (5 + 1)) + ((9 + 13) + (11 + 7)) = 50$$

Метод декомпозиции: пример

Структура рекурсивных вызовов функции sum(0, 6)



$$4 + 5 + 1 + 9 + 13 + 11 + 7 = (4 + 5 + 1) + (9 + 13 + 11 + 7) =$$

$$= ((4) + (5 + 1)) + ((9 + 13) + (11 + 7)) = 50$$

Метод декомпозиции: пример реализации

```
int sum(int *a, int l, int r)
{
    int k;

    if (l == r)
        return a[l];

    k = (r - l + 1) / 2;
    return sum(a, l, l + k - 1) + sum(a, l + k, r);
}

int main()
{
    s = sum(a, 0, N - 1);
}
```

Метод декомпозиции: пример реализации

```
int sum(int *a, int l, int r)
{
    int k;

    if (l == r)
        return a[l];

    k = (r - l + 1) / 2;
    return sum(a, l, l + k - 1) + sum(a, l + k, r);
}

int main()
{
    s = sum(a, 0, N - 1);
}
```

До 1 элемента возвращаем его значение как $\sum$

Спуск

Делим пополам

Левая часть

Правая часть

На возврате складываем левую и правую части

СОРТИРОВКИ: методом декомопзиции

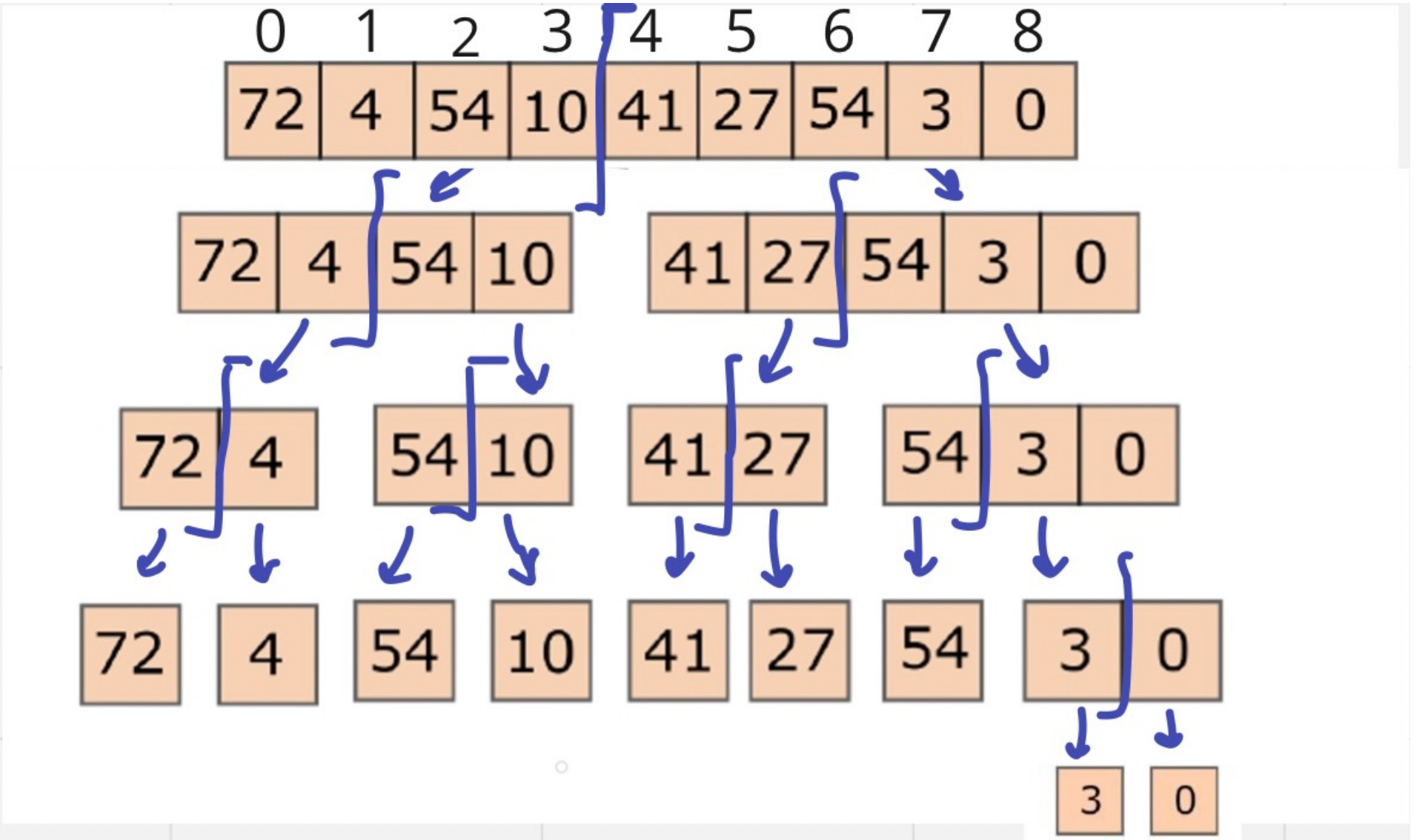
1. Как в сортировке вставками: один элемент - это отсортированный элемент = подразбиваем задачу сортировки массива до 1 элемента
2. Объединяем решения по парам: два одно элементных в двух элементные, два двух элементные в 4х элементные и так далее.
3. Четное разбиение не всегда может выйти – значит можем объединять 1 и 2 элементные, но также по парам (1+2) = три элемента
4. **Рекурсия**: на спуске – разбиваем, на возврате сортируем в рамках подзадач и объединяем решения

Рассмотрим сортировку слияниями (Merge Sort)

Сортировка слиянием

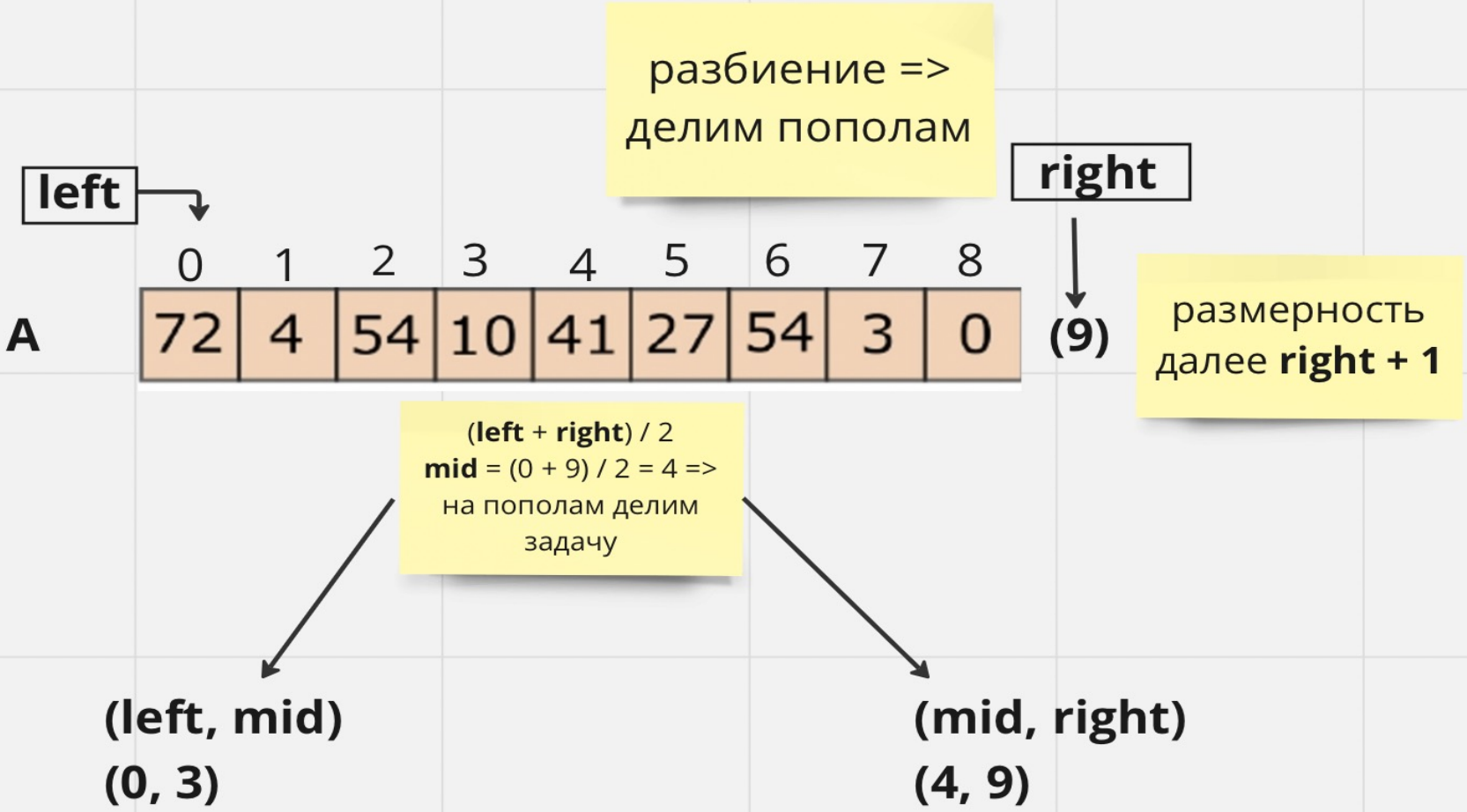
Сортировка слиянием

Делим пополам, если нечетный размер, то справа на один больше



Рассмотрим подробнее, как сделать разбиение по индексам

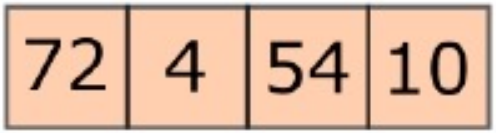
Сортировка слиянием

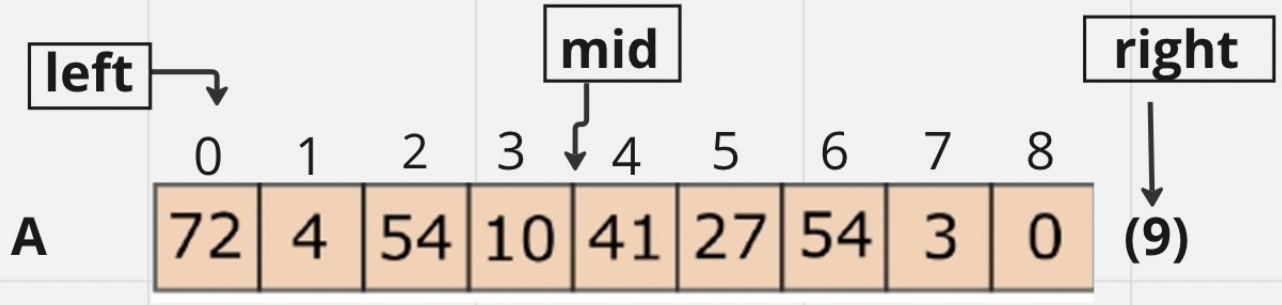


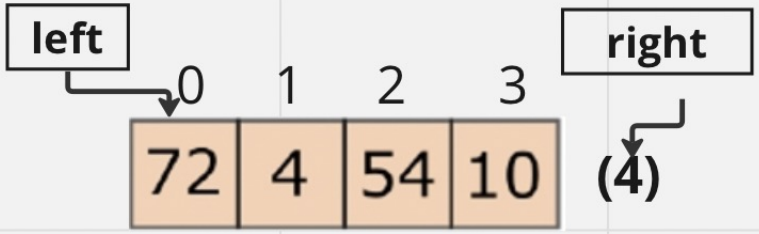
То есть если размер нечетный, то в правом подмаслите на 1 элемент больше

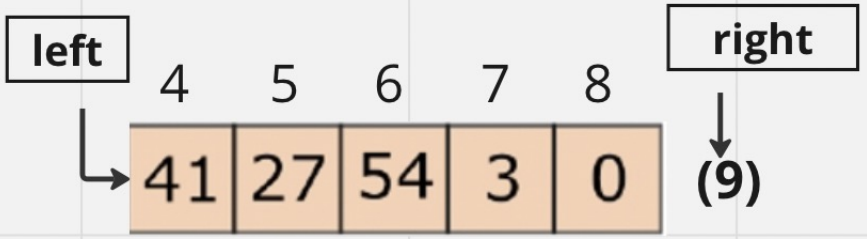
*Если не присылать **Right + 1** или на старте **размерность**, то придется каждый раз учитывать четность размера*

Сортировка слиянием









mid1 = (0 + 4) / 2 = 2

(left1, mid1) (mid1, right1)

(0, 2) (2, 4)

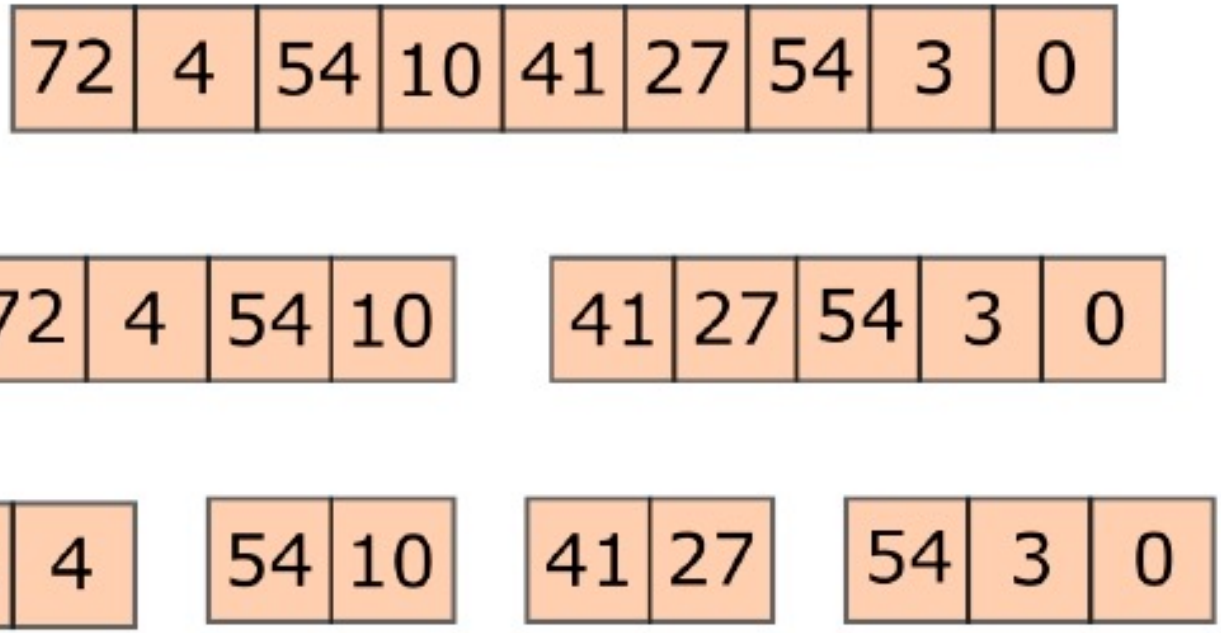
mid2 = (4 + 9) / 2 = 6

(left2, mid2) (mid2, right2)

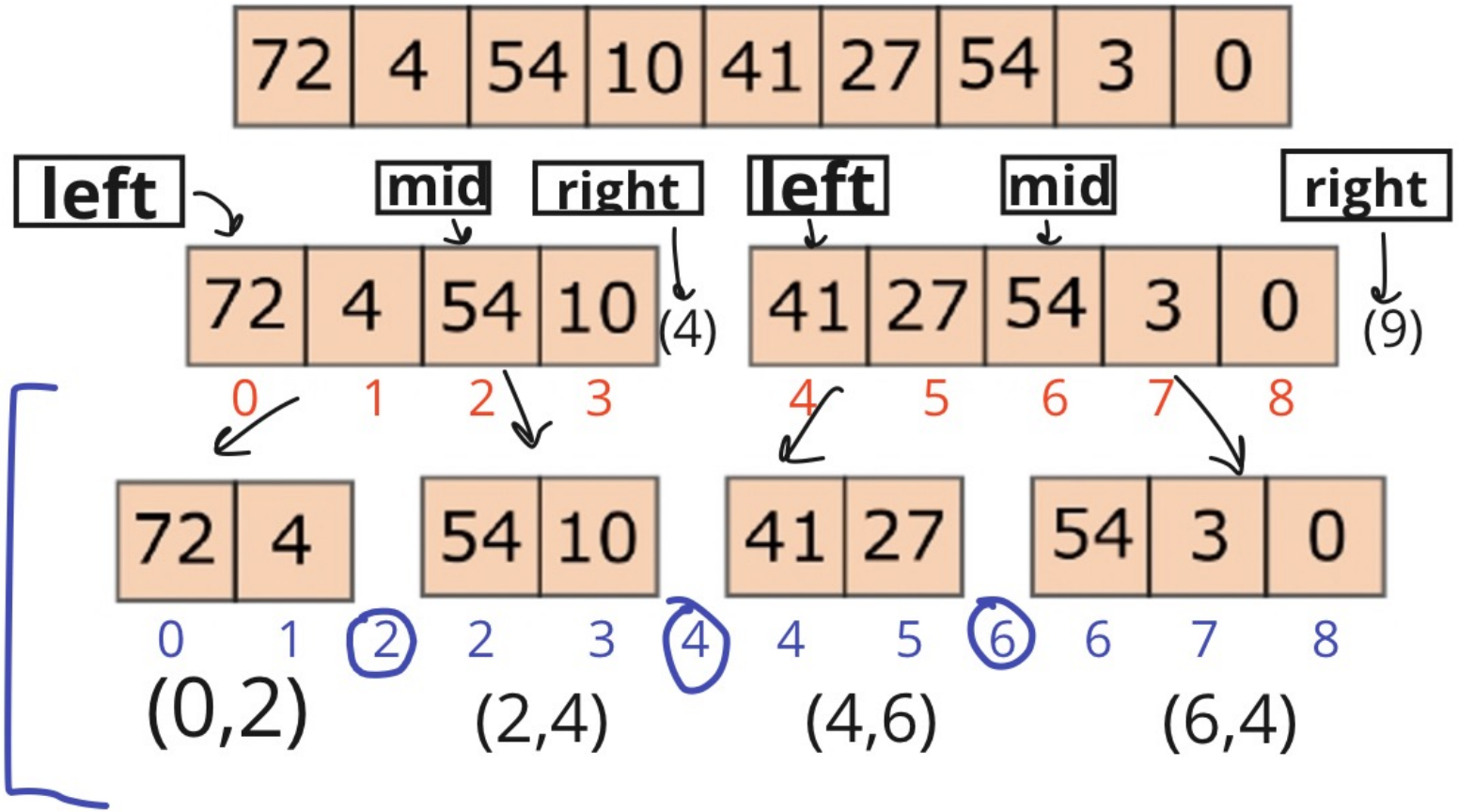
(4, 6) (6,9)

Дополнительные вычисления

Сортировка слиянием

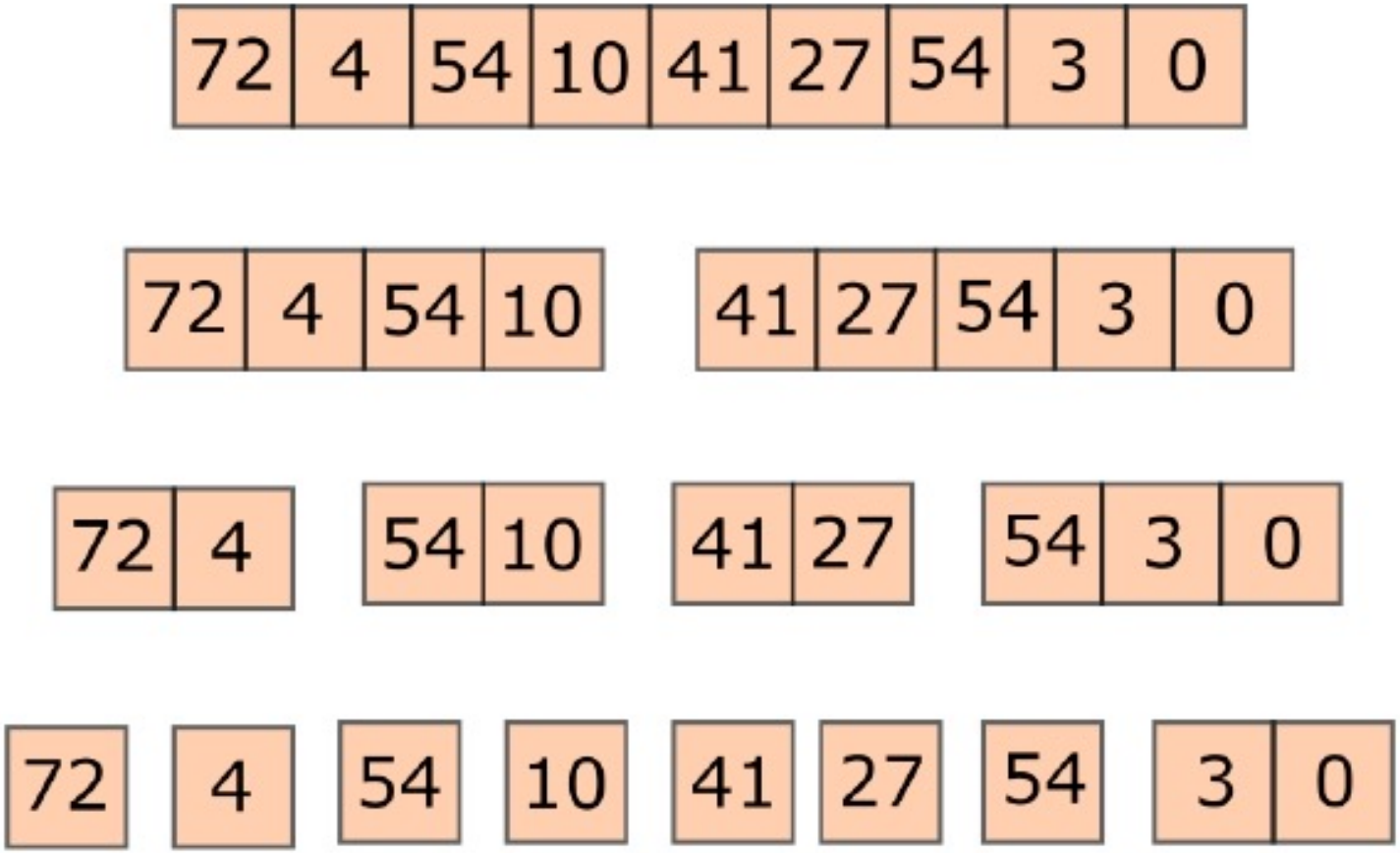


Сортировка слиянием

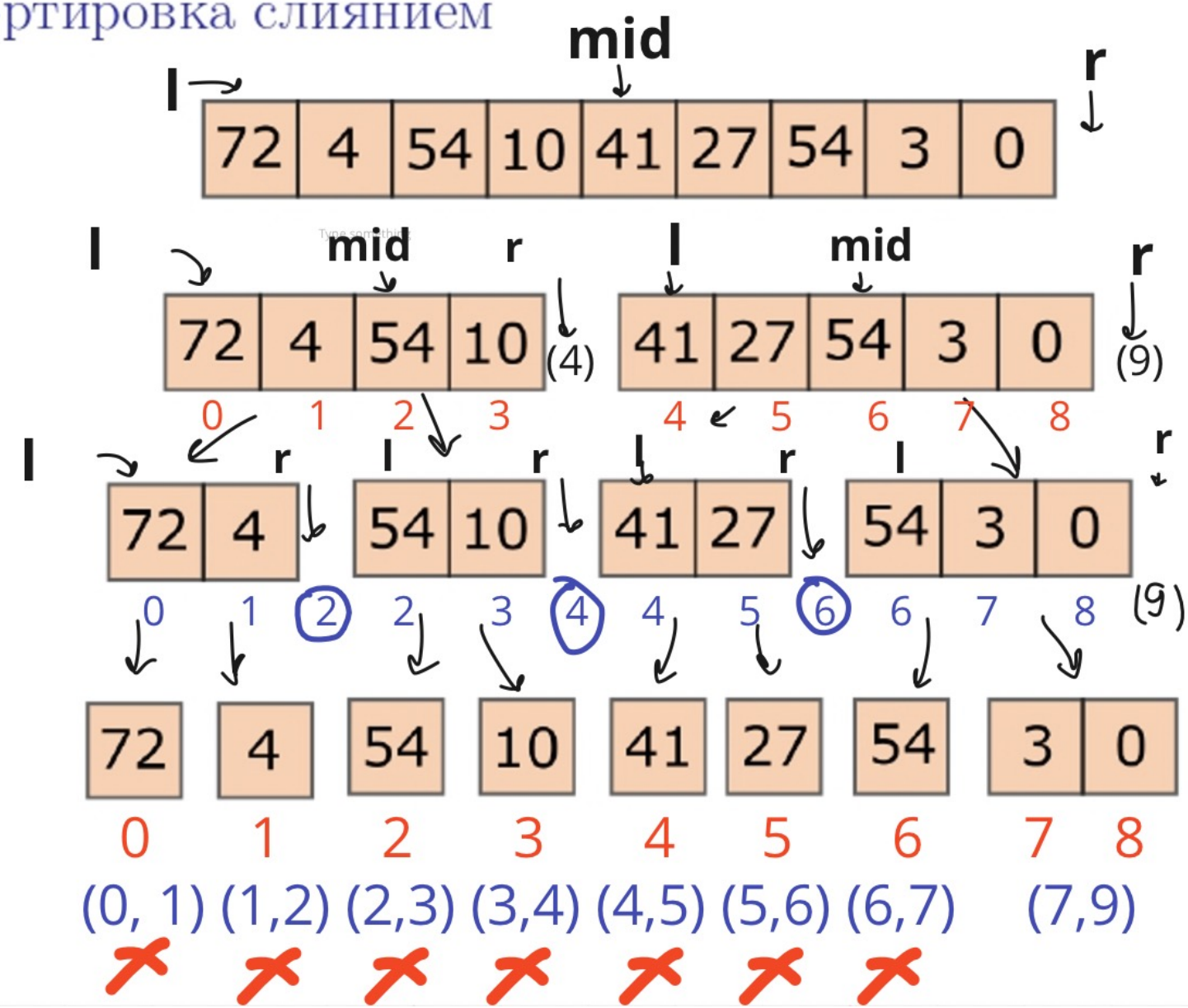


Вычисление
по аналогии

Сортировка слиянием

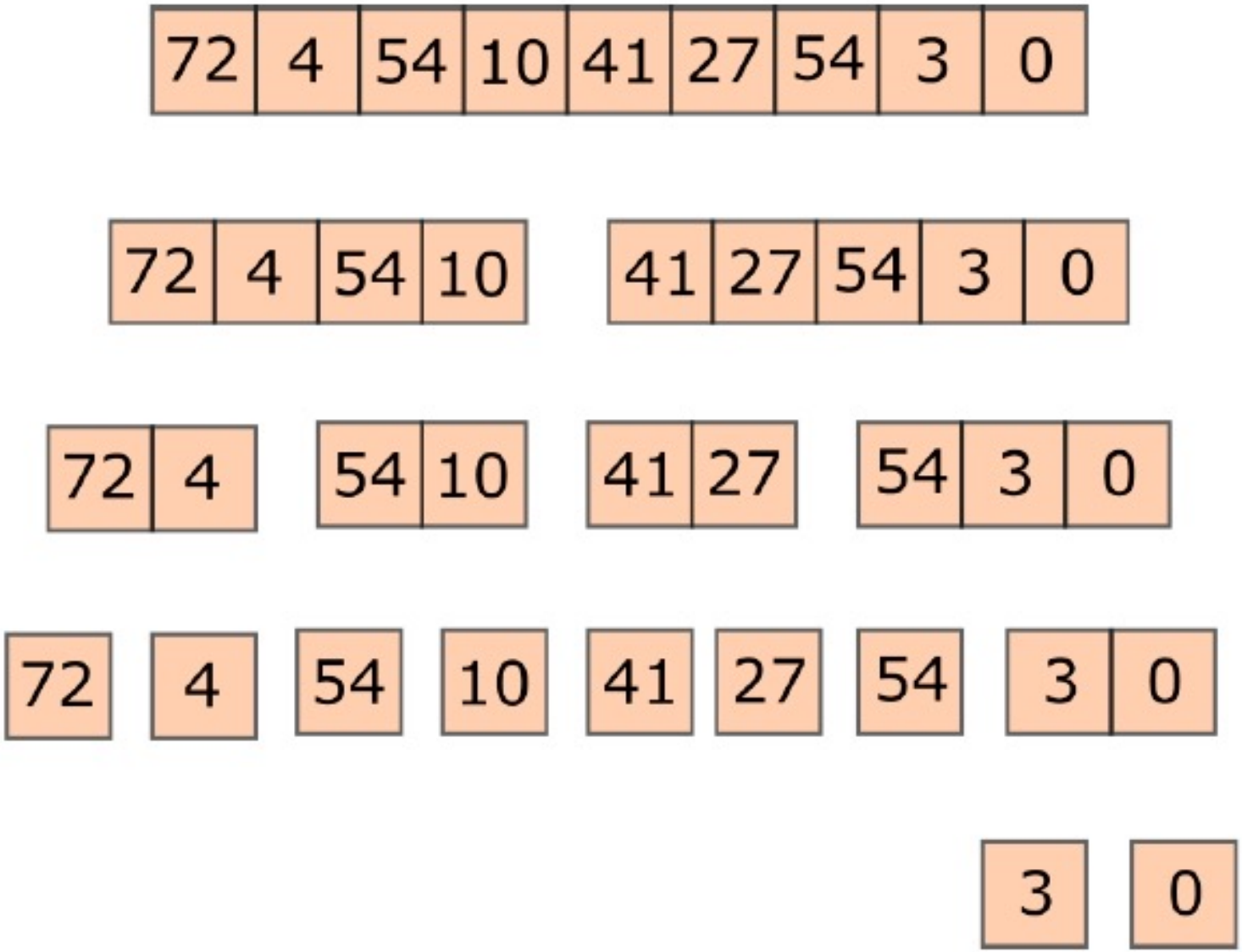


Сортировка слиянием



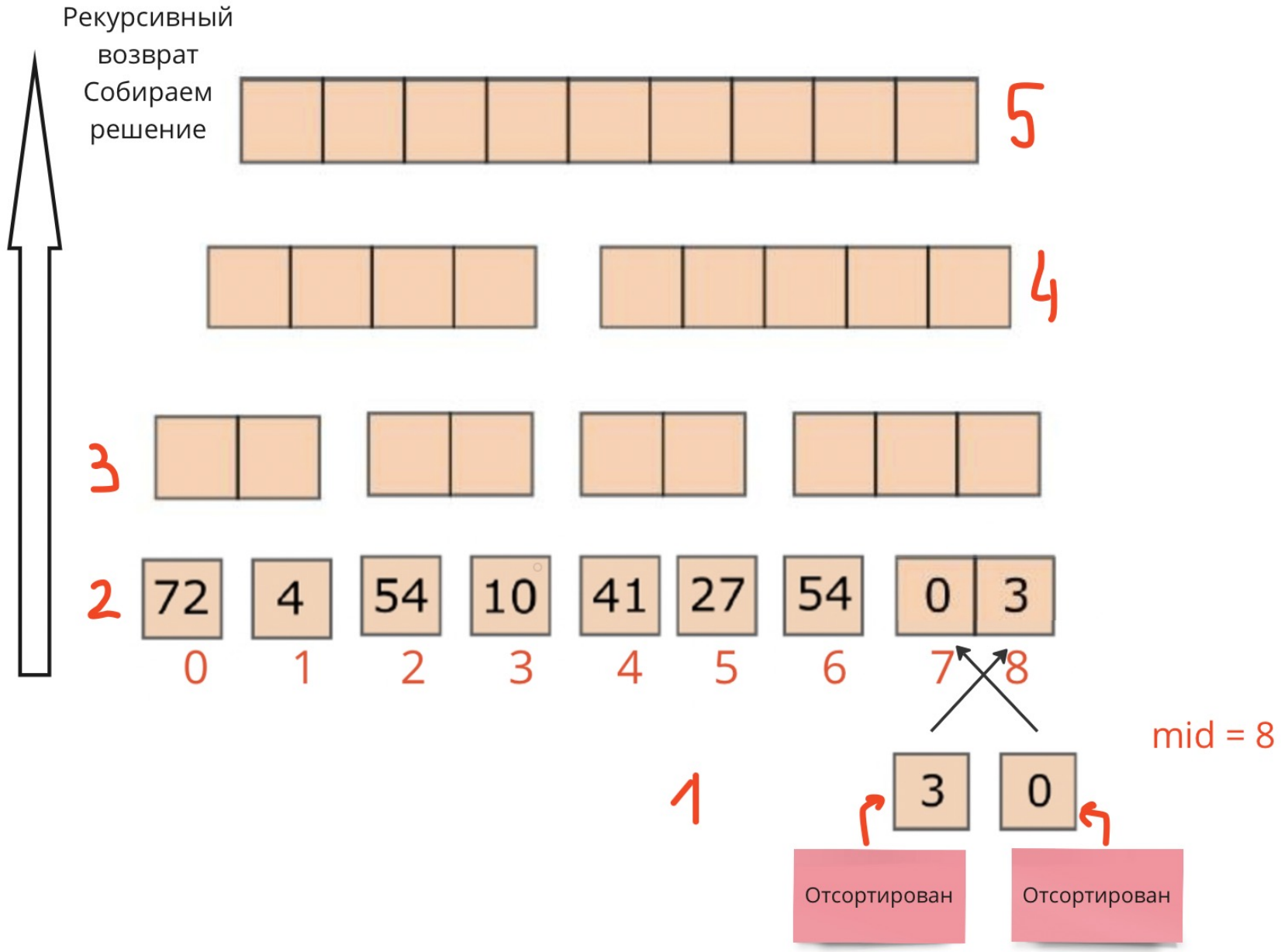
Выход из рекурсии: left + 1 >= right

Сортировка слиянием

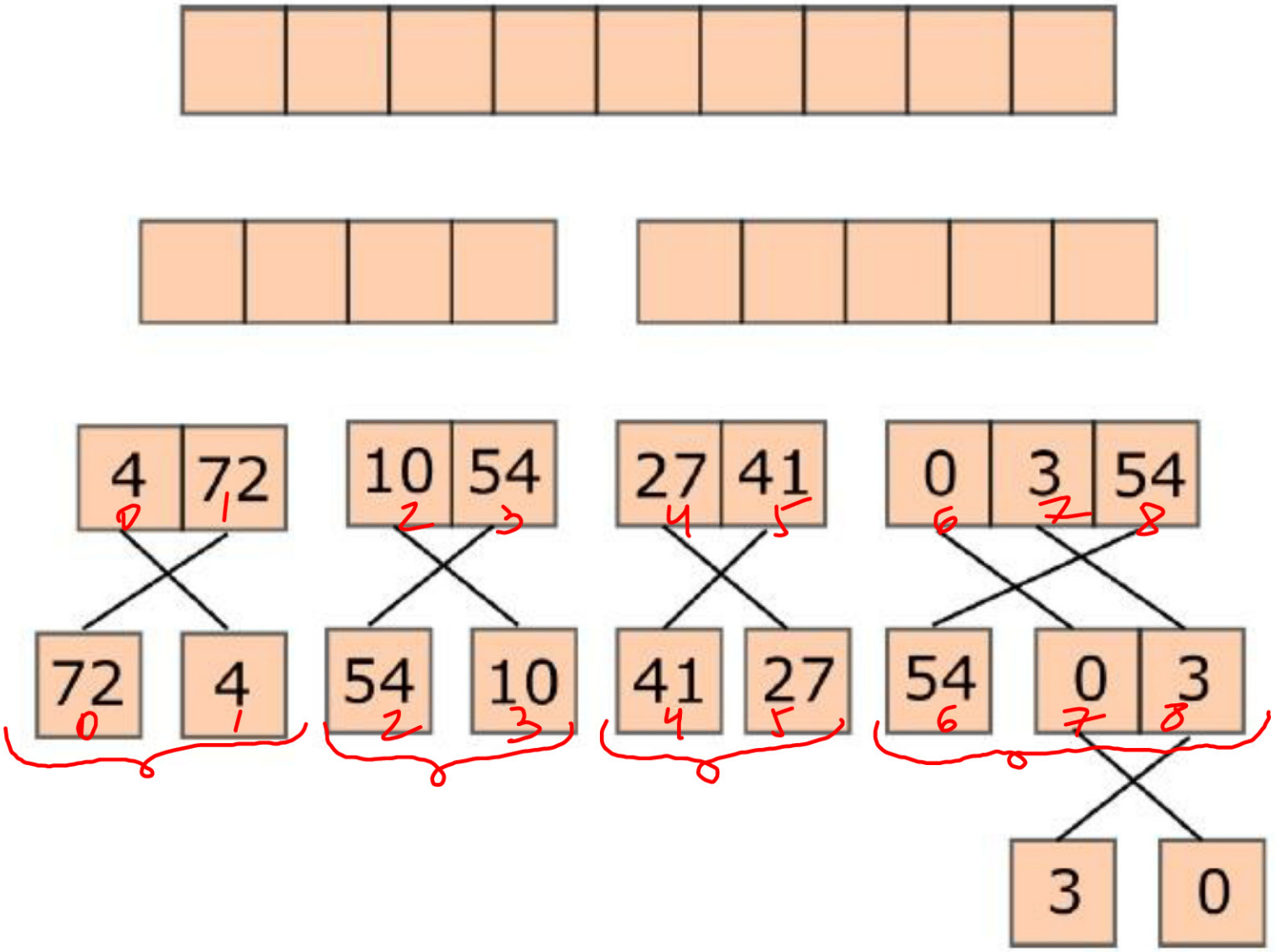


Сортировка слиянием

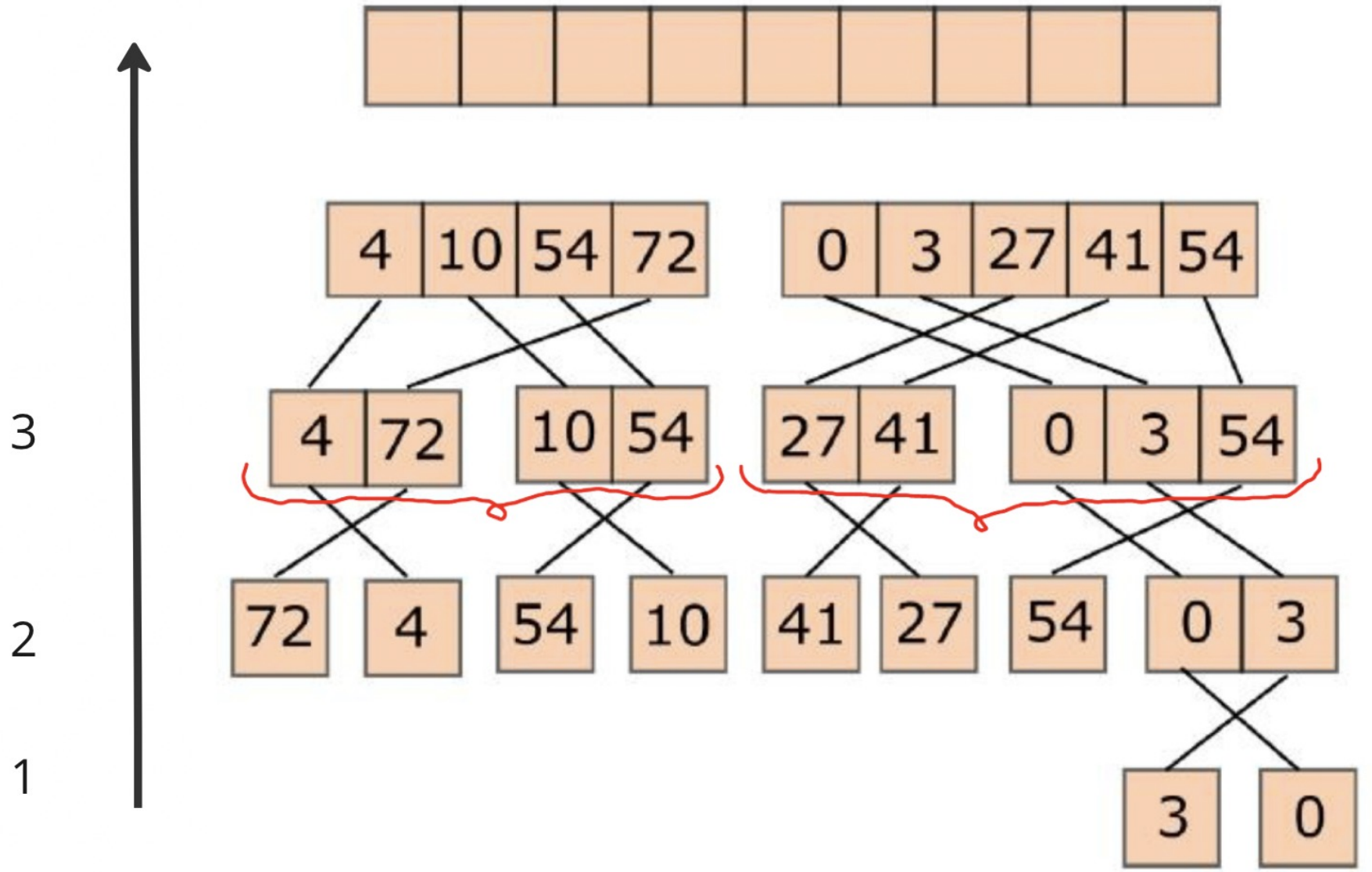
Сортировка слиянием



Сортировка слиянием

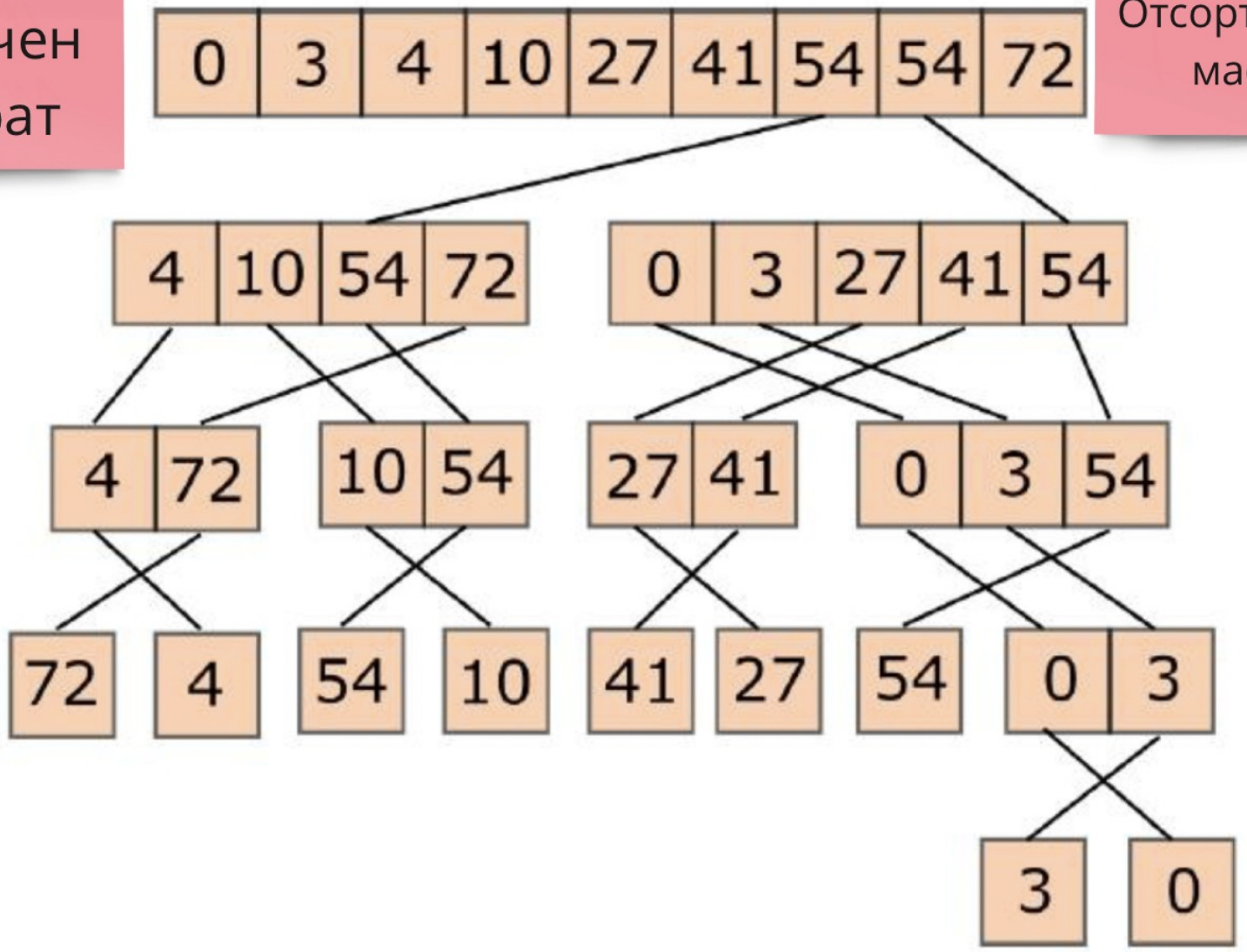


Сортировка слиянием



Сортировка слиянием

Закончен возврат



Отсортирован массив

Рекурсивная реализация

```
function mergeSortRecursive(a : int[n]; left, right : int):
    if left + 1 >= right
        return
    mid = (left + right) / 2
    mergeSortRecursive(a, left, mid)
    mergeSortRecursive(a, mid, right)
    merge(a, left, mid, right)
```

Рекурсивная реализация

```
function mergeSortRecursive(a : int[n]; left, right : int):
    if left + 1 >= right
        return
    mid = (left + right) / 2
    mergeSortRecursive(a, left, mid)
    mergeSortRecursive(a, mid, right)
    merge(a, left, mid, right)
```

Условие выхода из рекурсии

Рекурсия

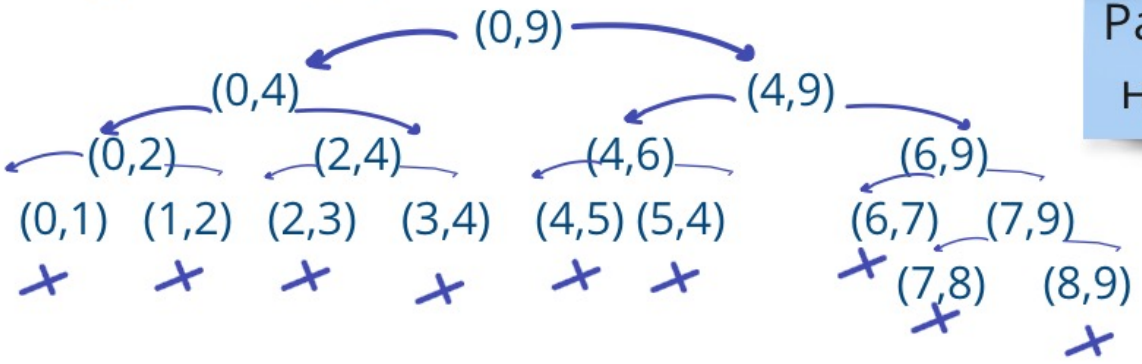
Середина

Левая часть

Правая часть

На возврате объединяем и сортируем подмассивы

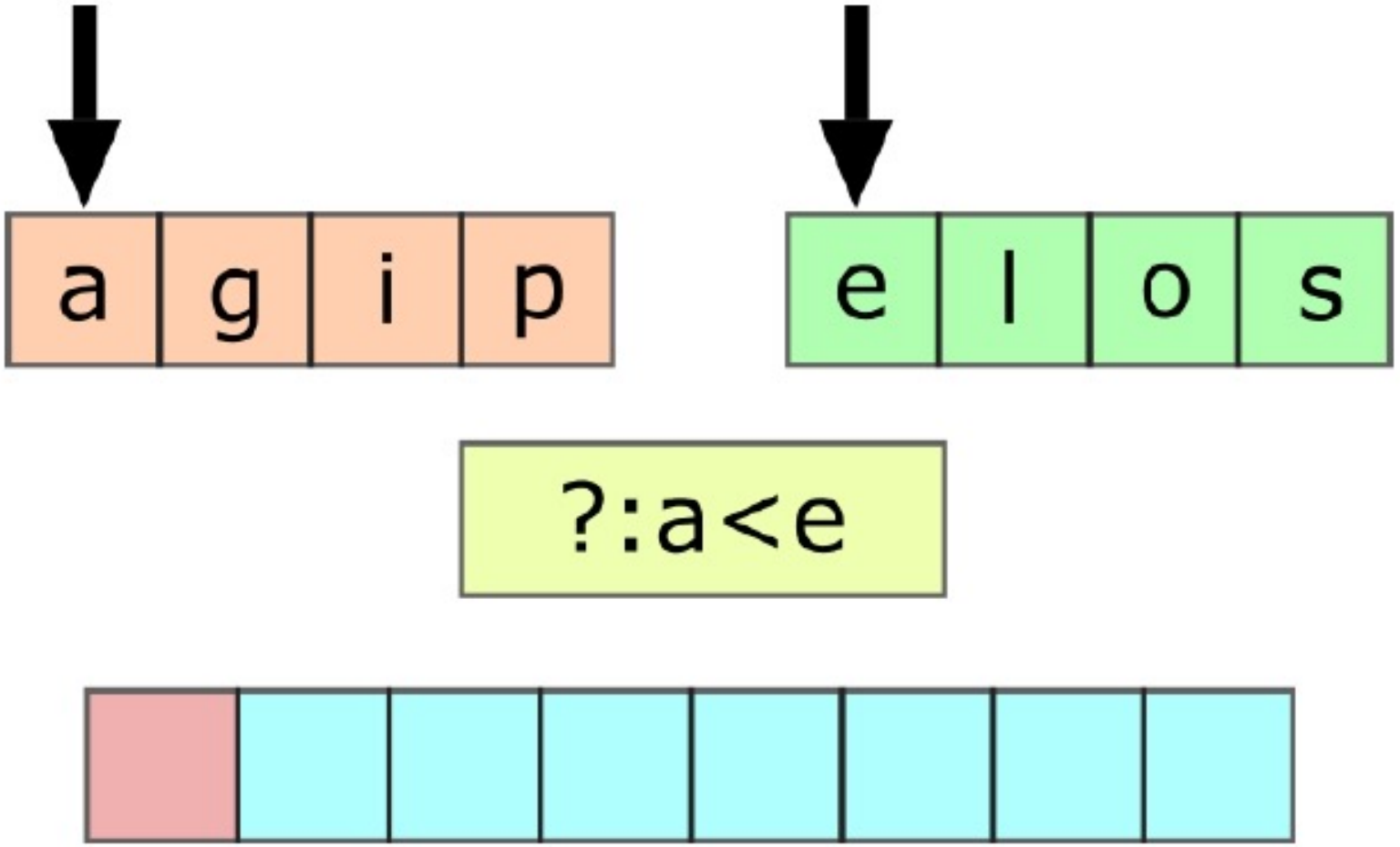
Рекурсивное дерево вызовов



Разбиение на спуске

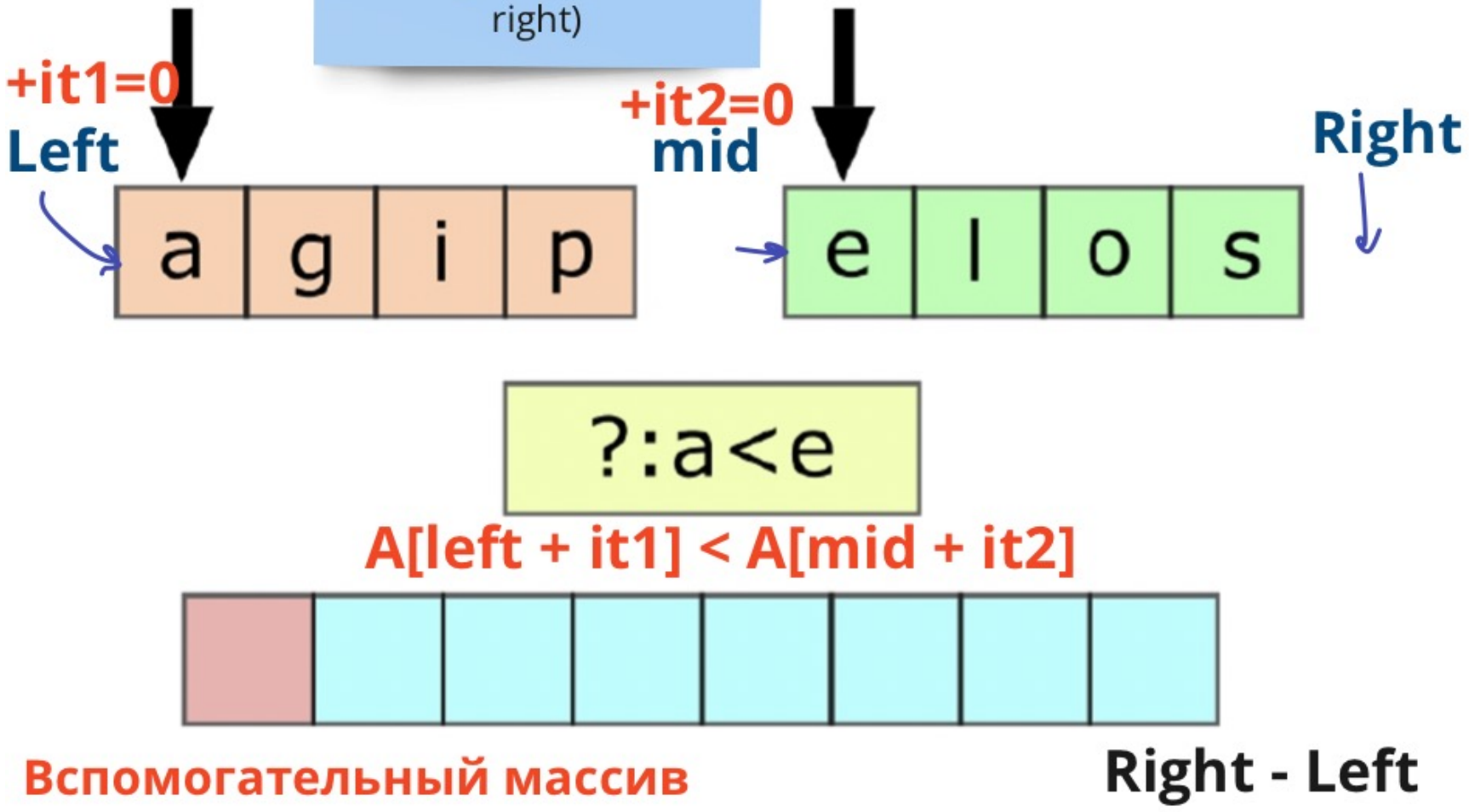
Рассмотрим рекурсивный возврат: процесс сбора общего решения - слияния ***Merge***

Merging



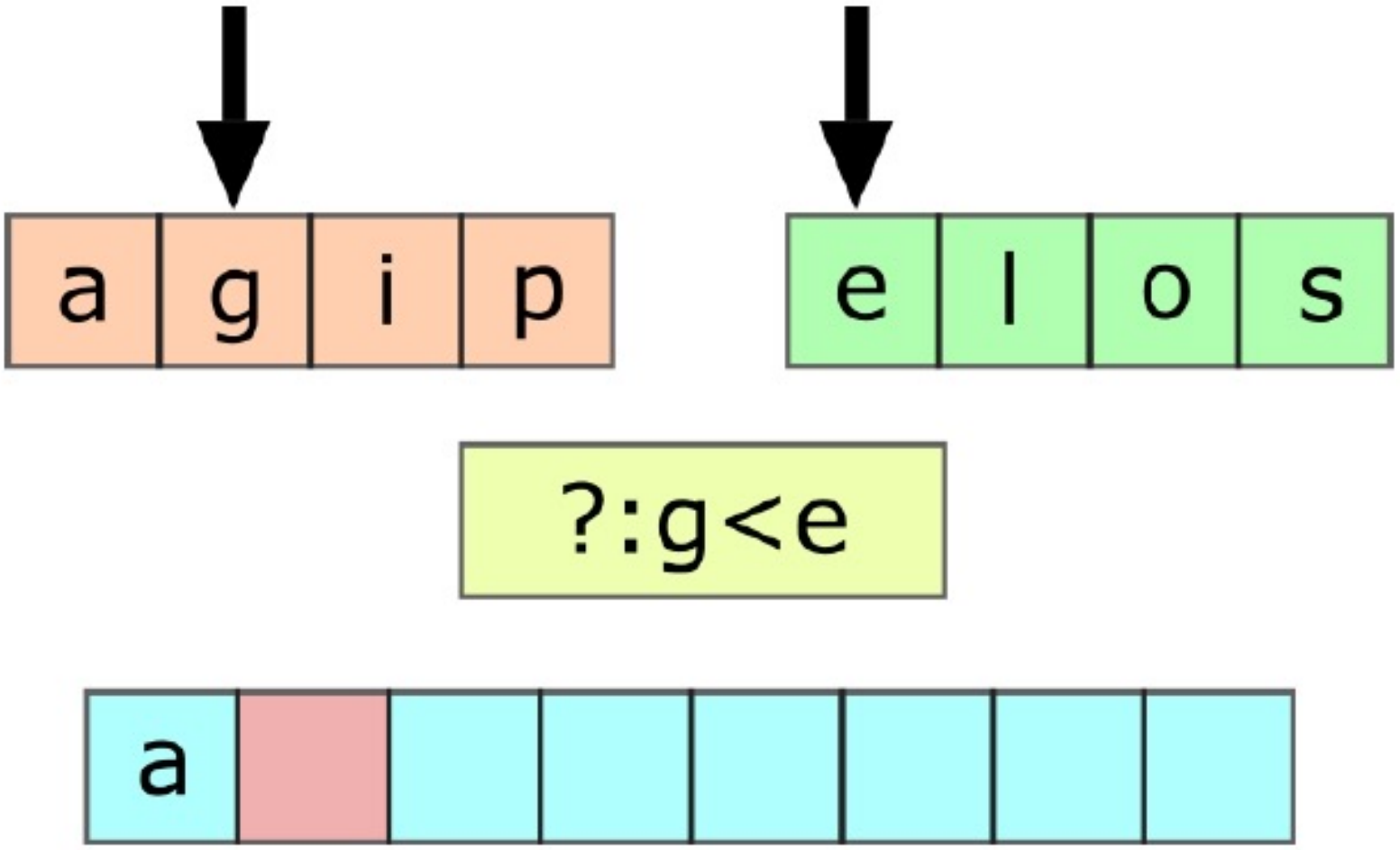
Merging

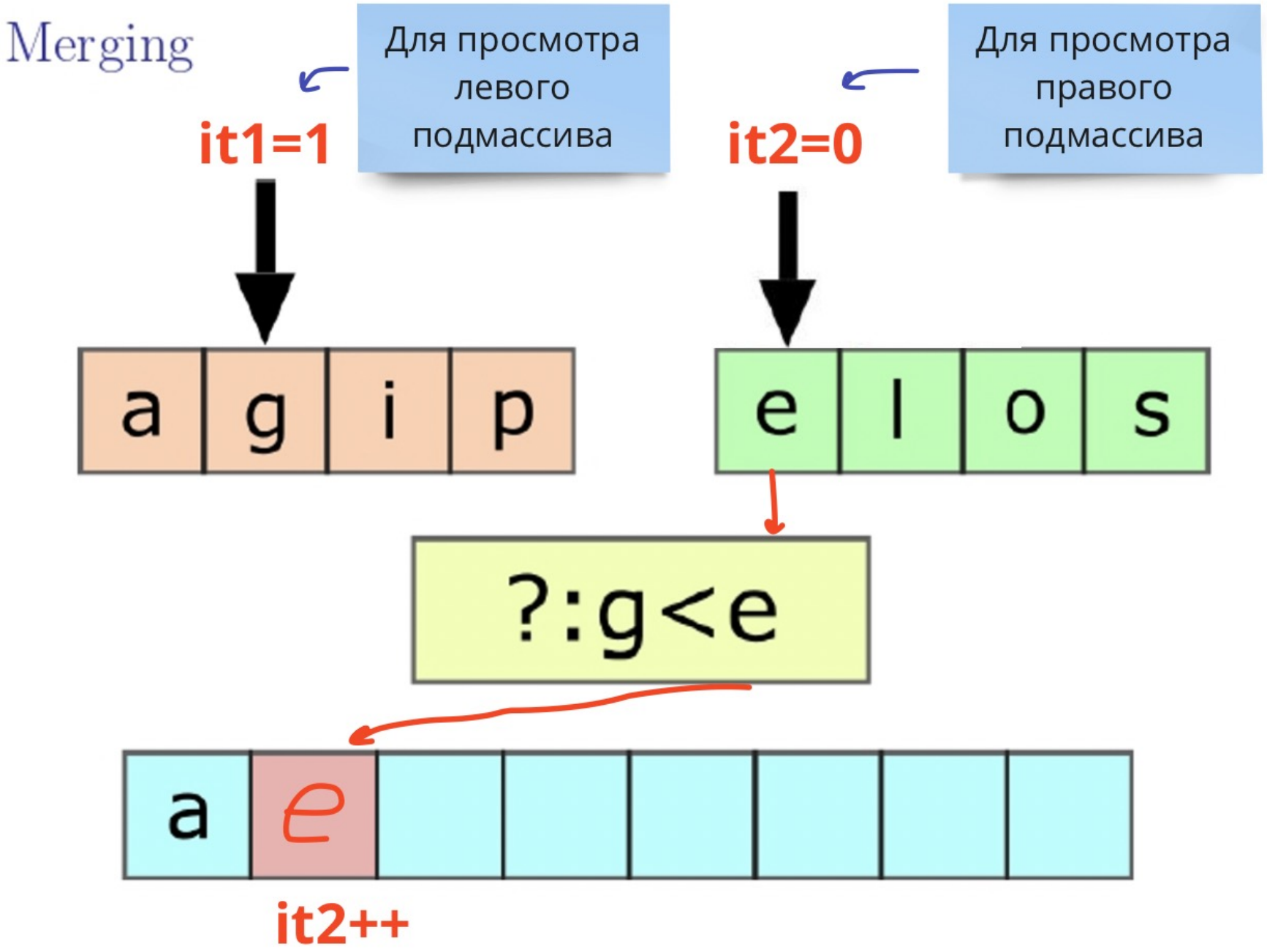
На примере массива символов (char[])
Merge(A, left, mid, right)



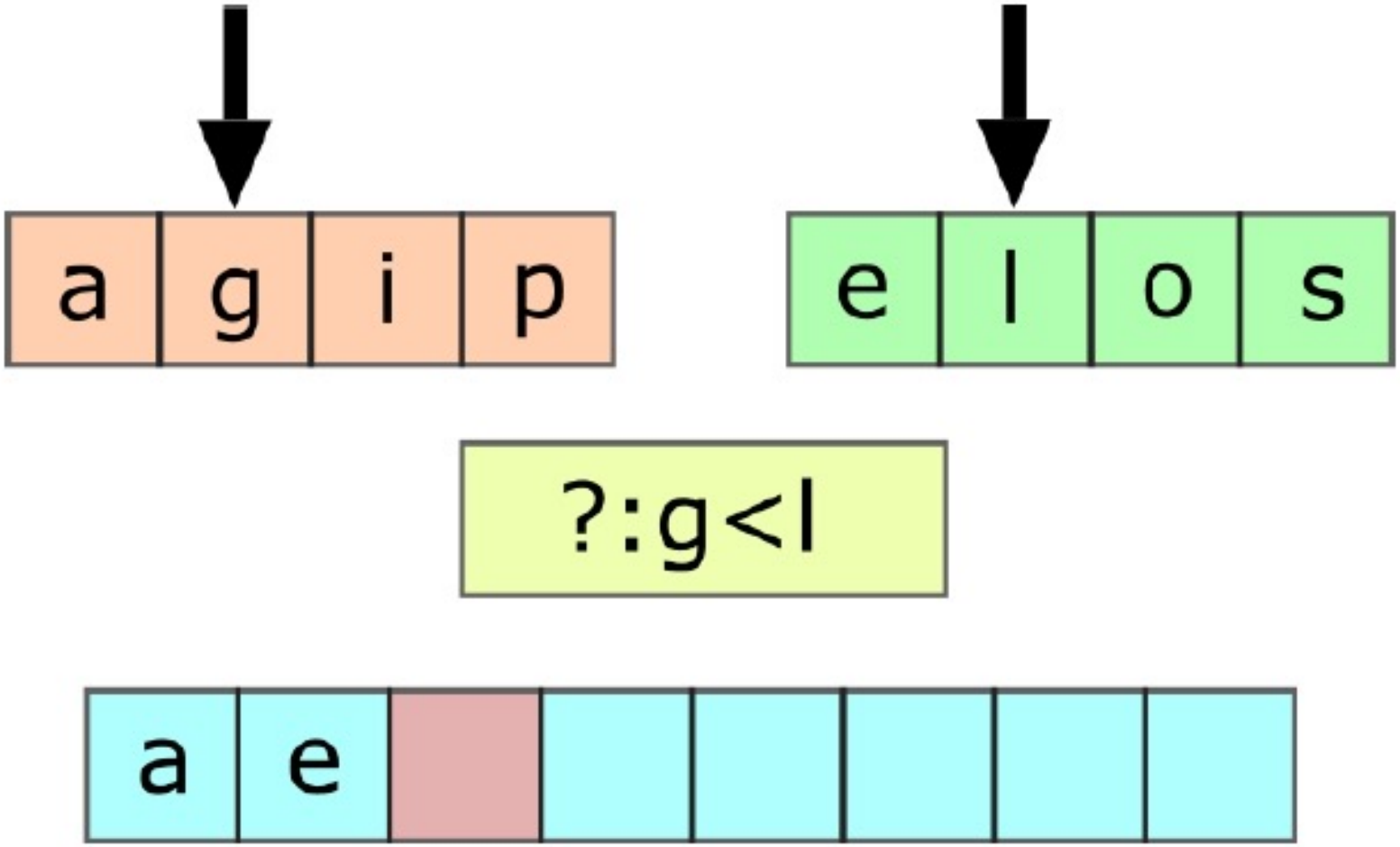
Left и mid граница,
их нужно сохранить

Merging

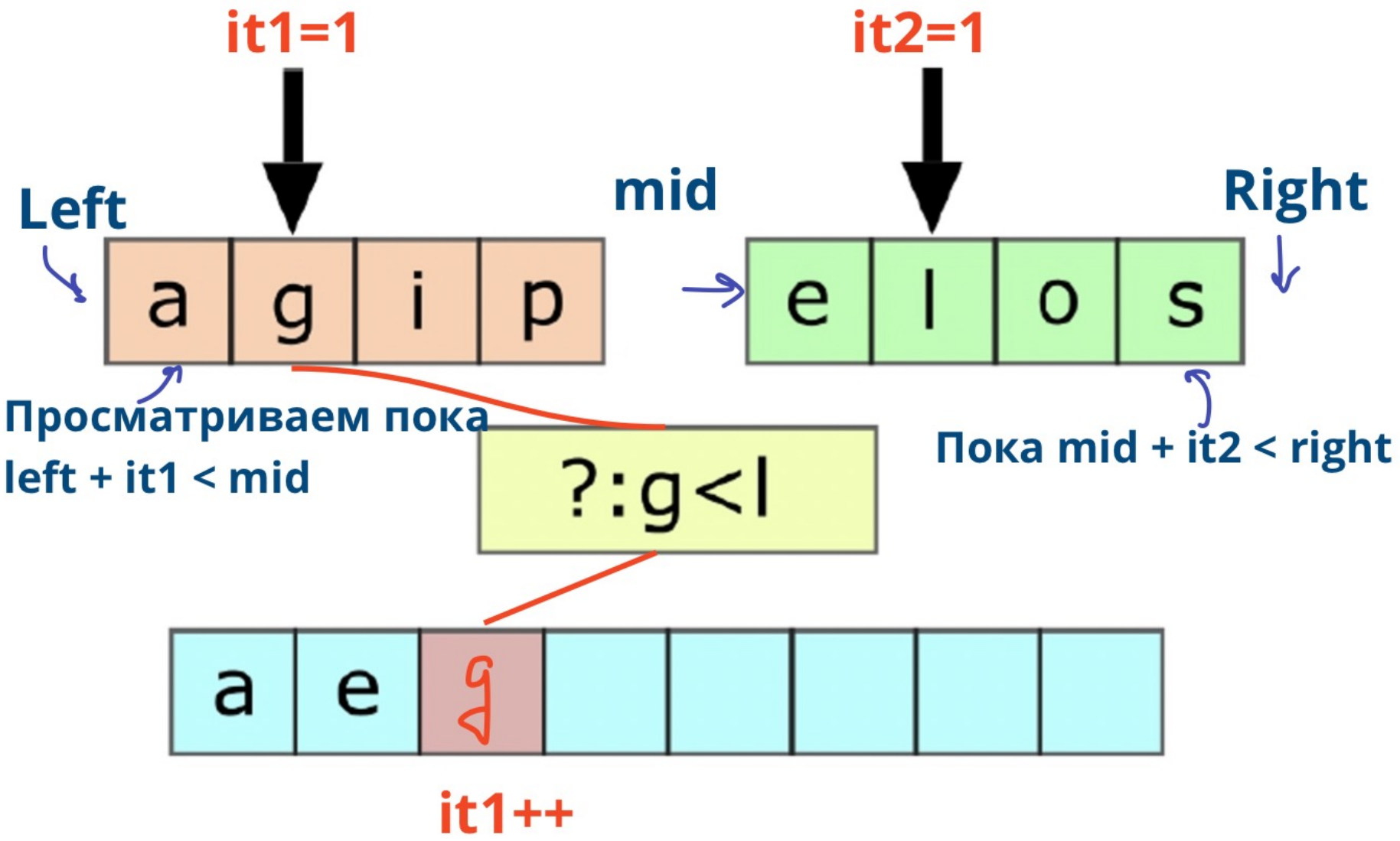



Merging
Для просмотра левого подмассива
Для просмотра правого подмассива
it1=1
it2=0
a g i p
e l o s
?:g<e
a e
it2++

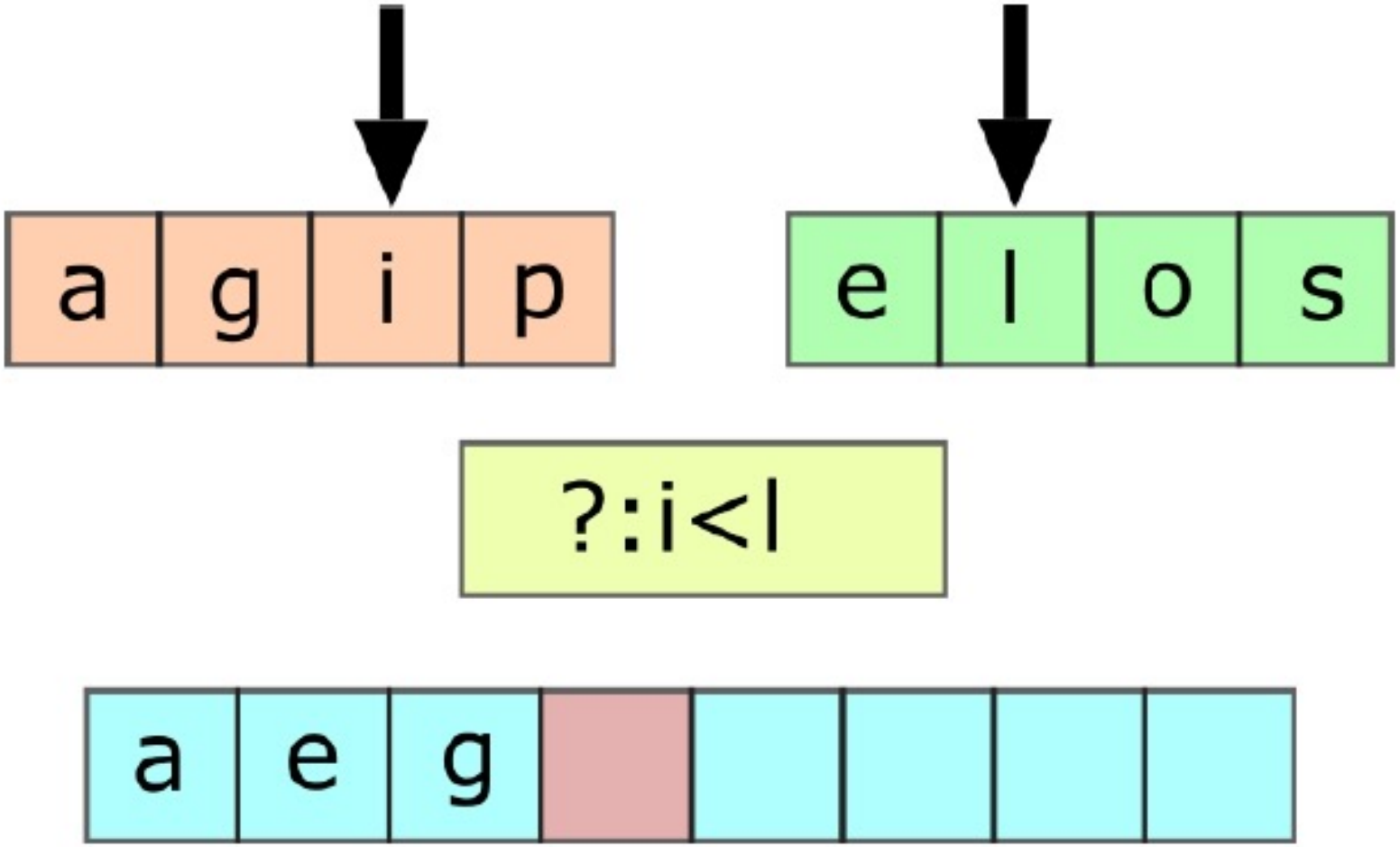
Merging



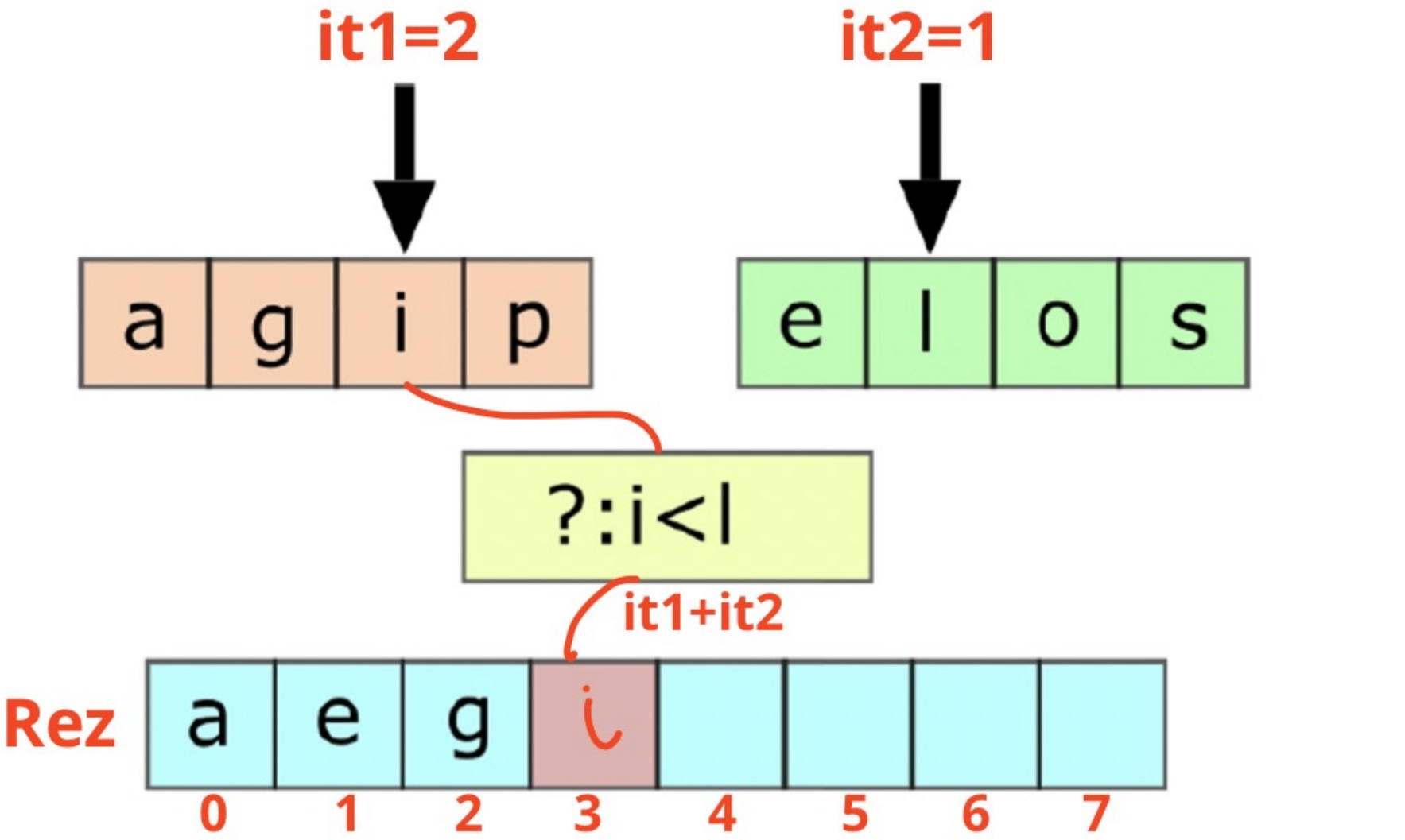
Merging


it1=1
it2=1
Left
mid
Right
a g i p
e l o s
Просматриваем пока
left + it1 < mid
?:g<l
Пока mid + it2 < right
a e g
it1++

Merging

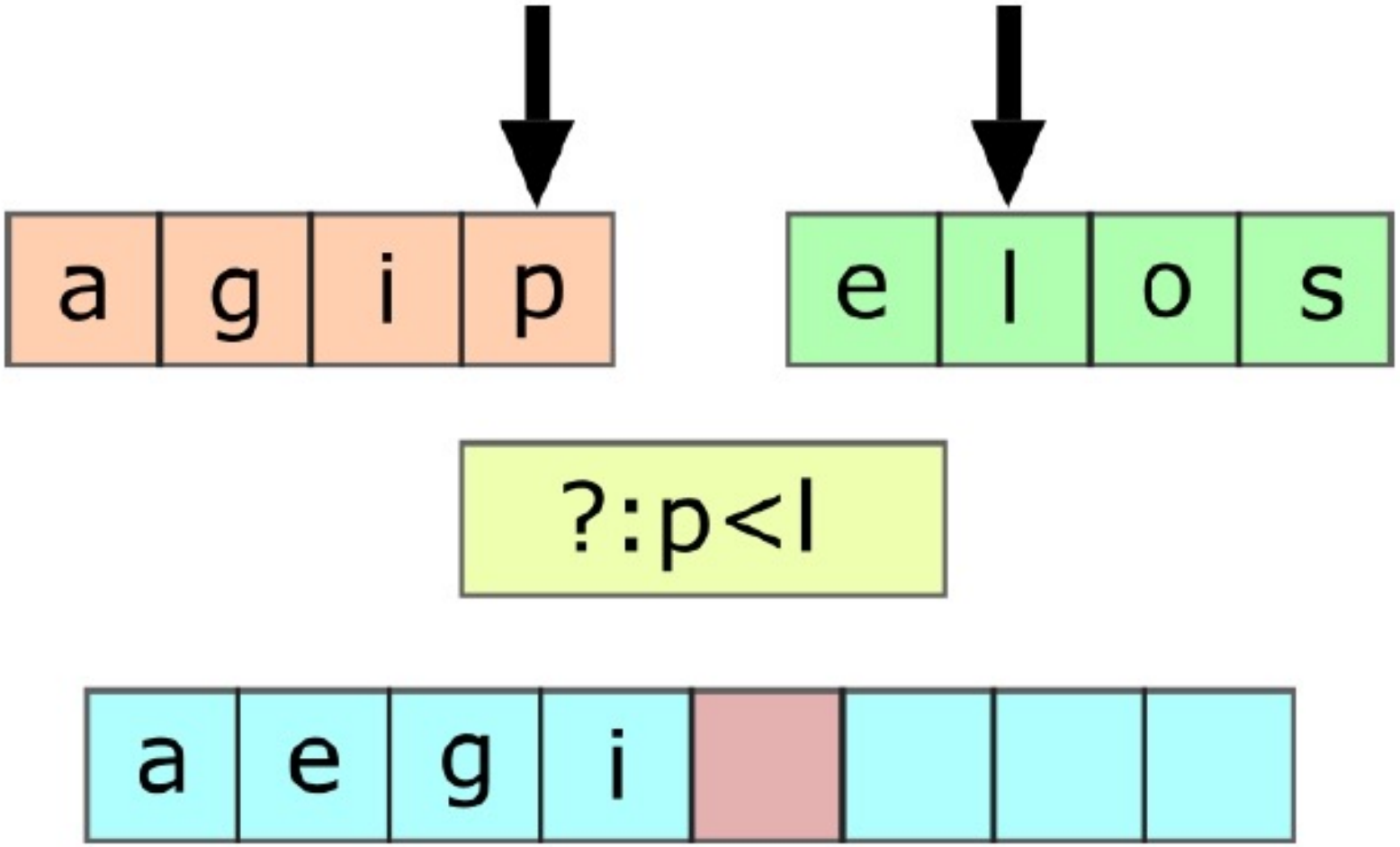


Merging

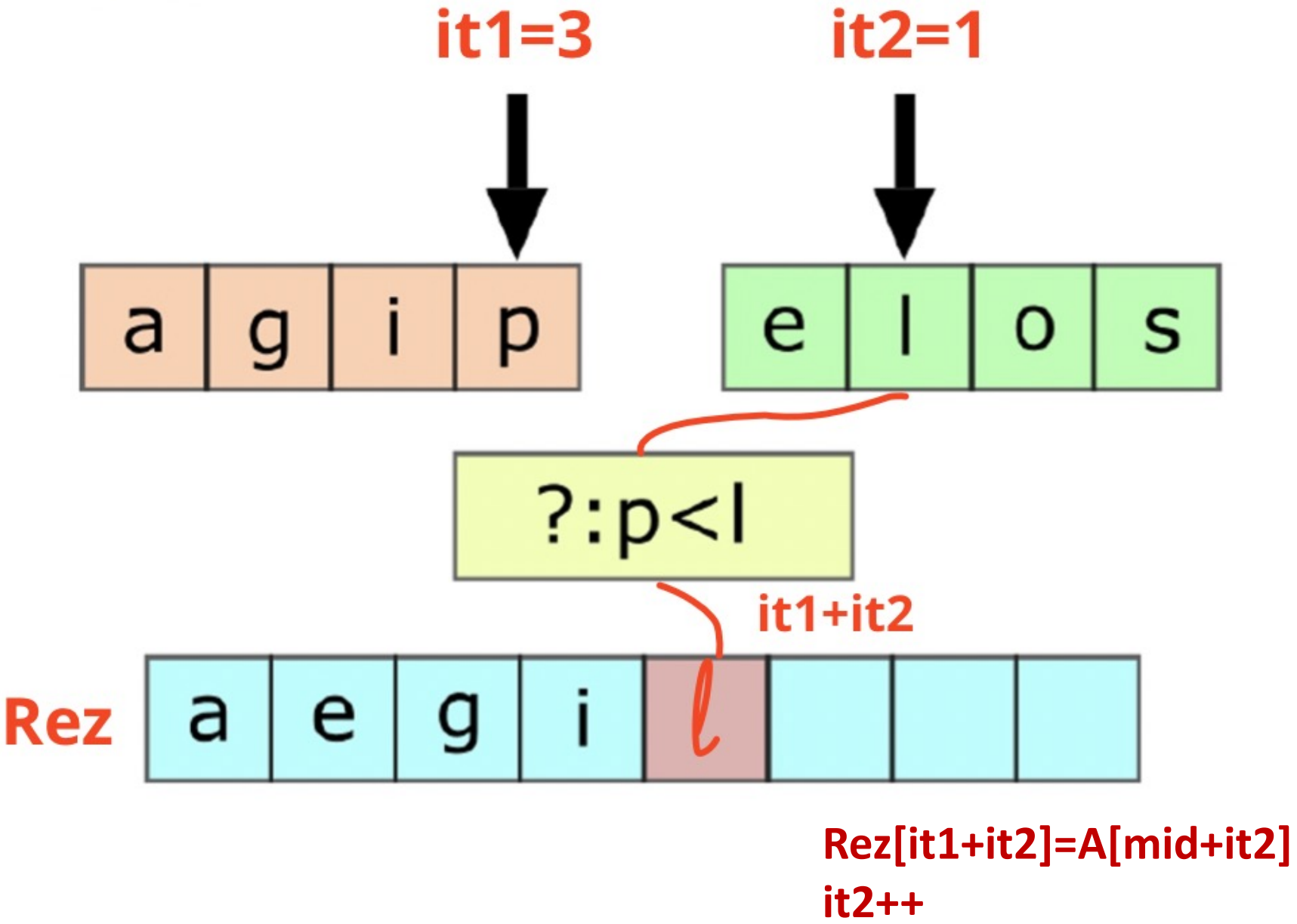


Rez[it1+it2]=A[left+it1]
it1++

Merging

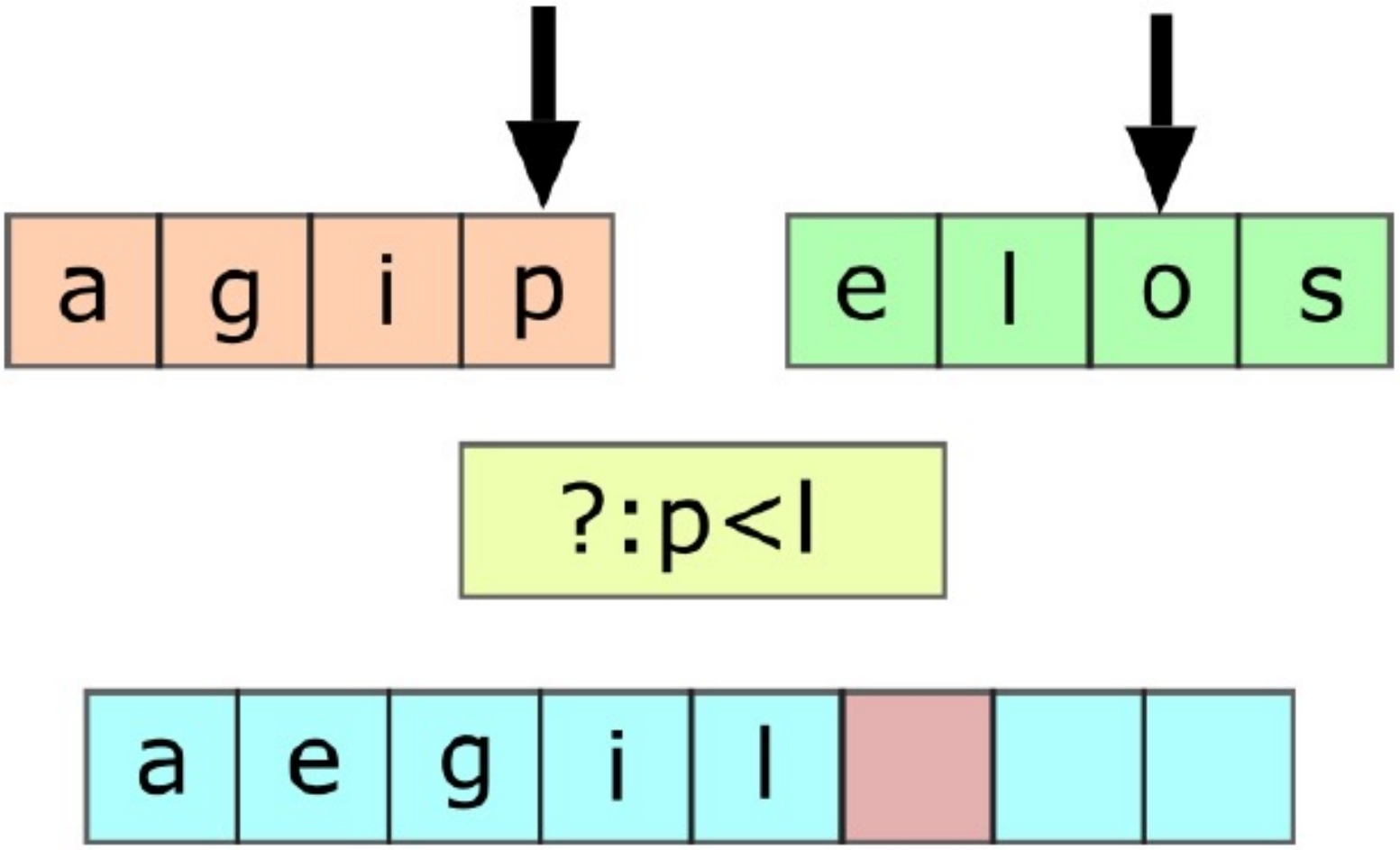


Merging

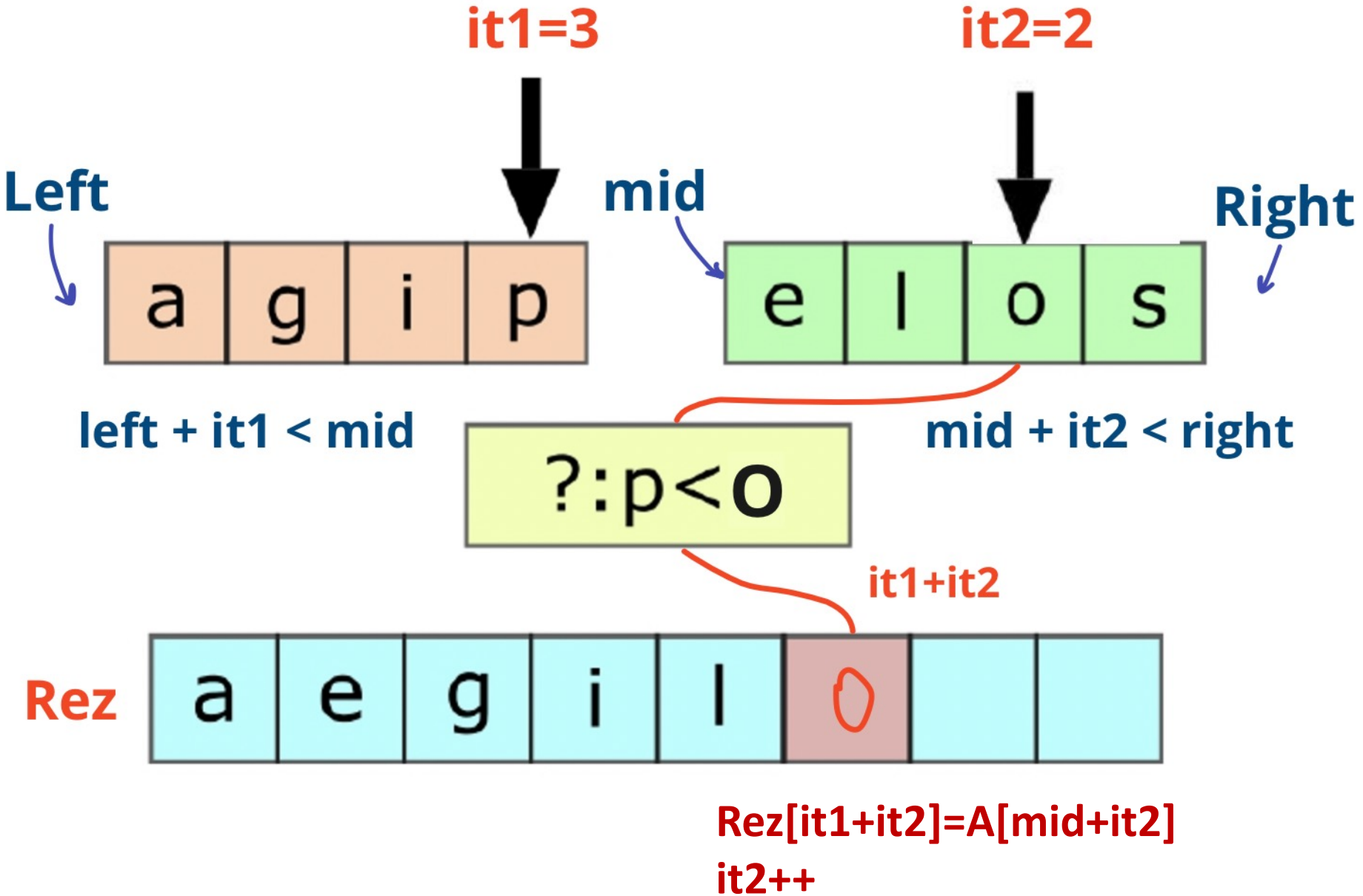


Rez[it1+it2]=A[mid+it2]
it2++

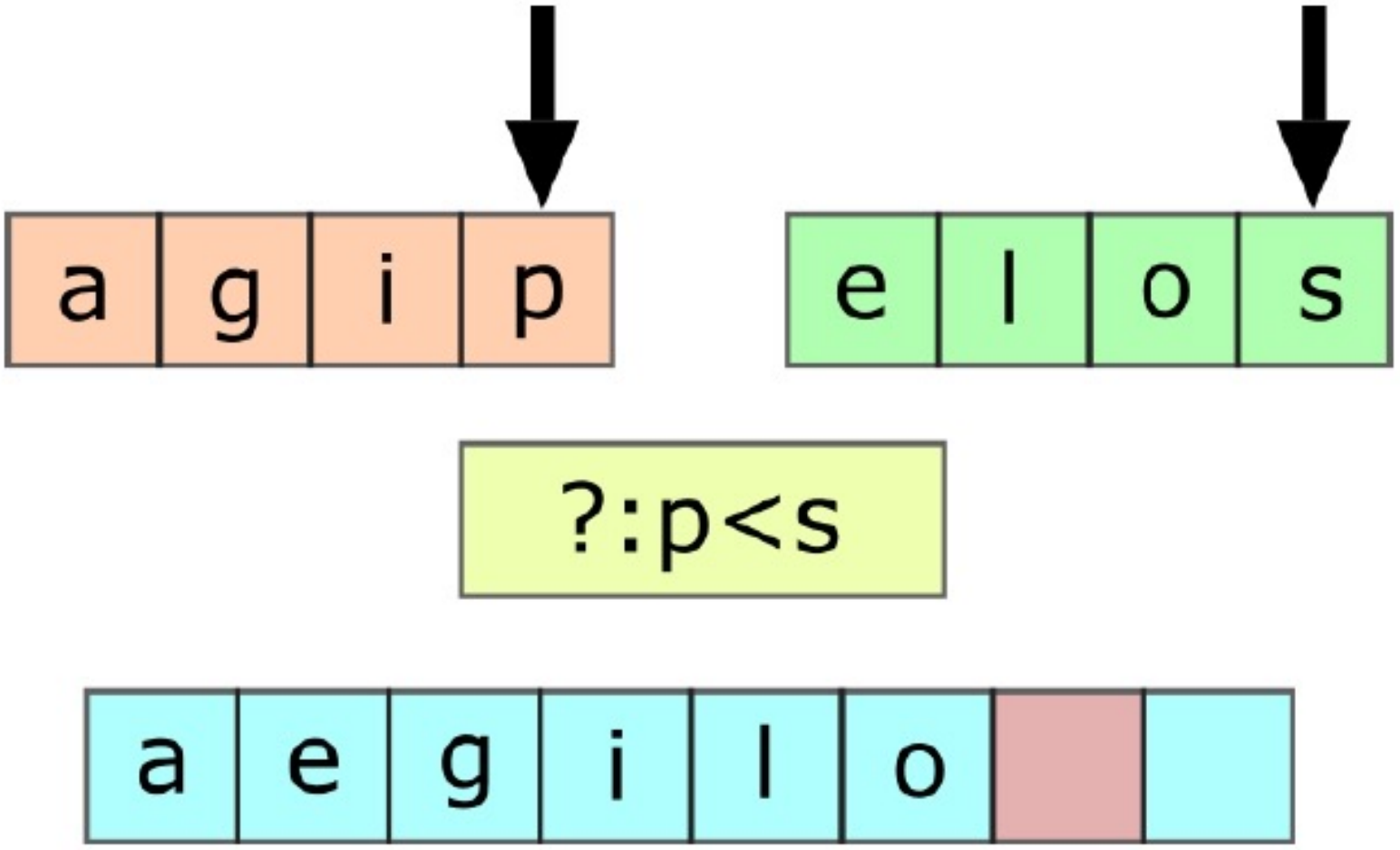
Merging



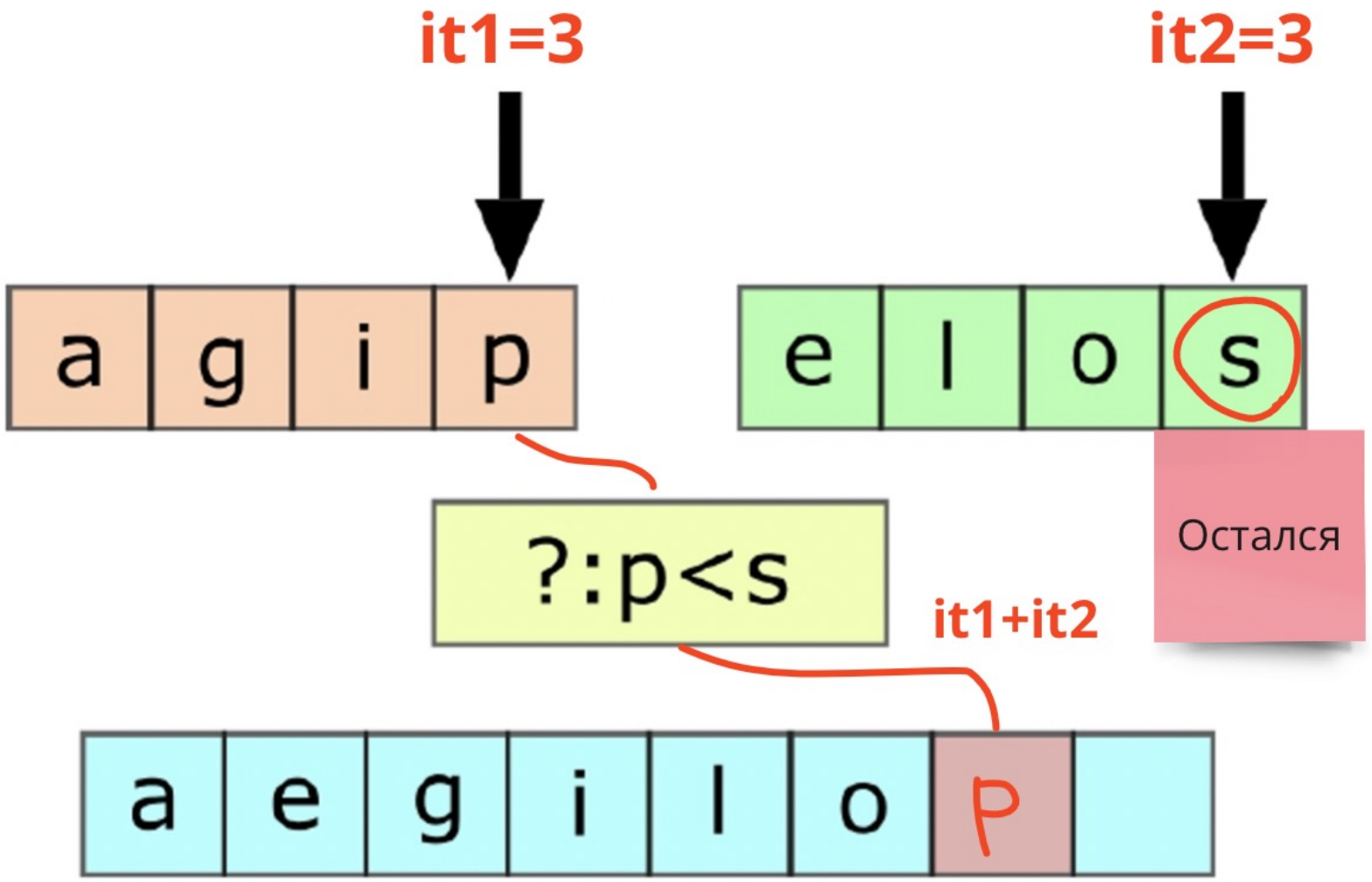
Merging


it1=3
it2=2
Left
mid
Right
a g i p
e l o s
left + it1 < mid
mid + it2 < right
?:p<o
it1+it2
Rez
a e g i l
Rez[it1+it2]=A[mid+it2]
it2++

Merging



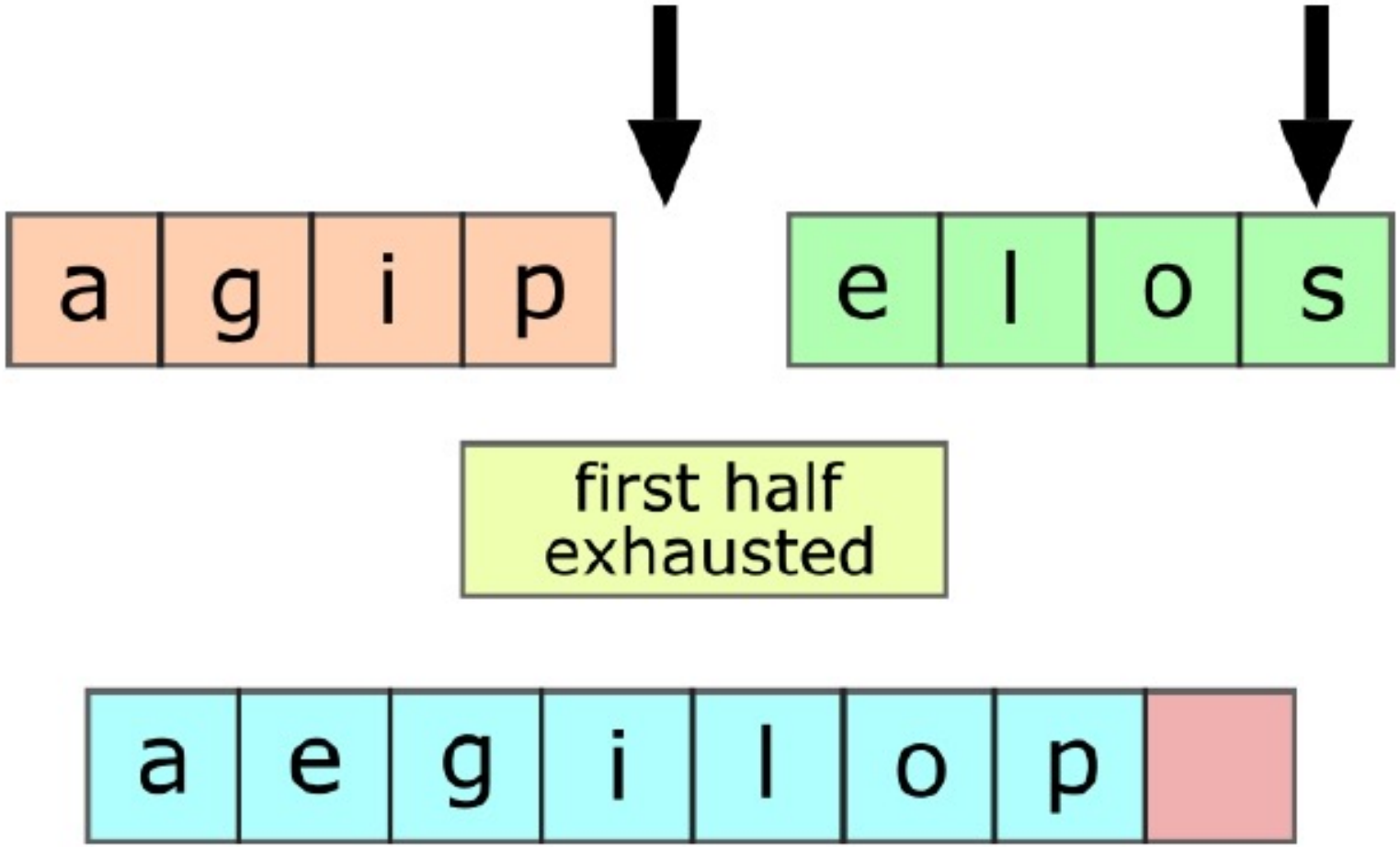
Merging



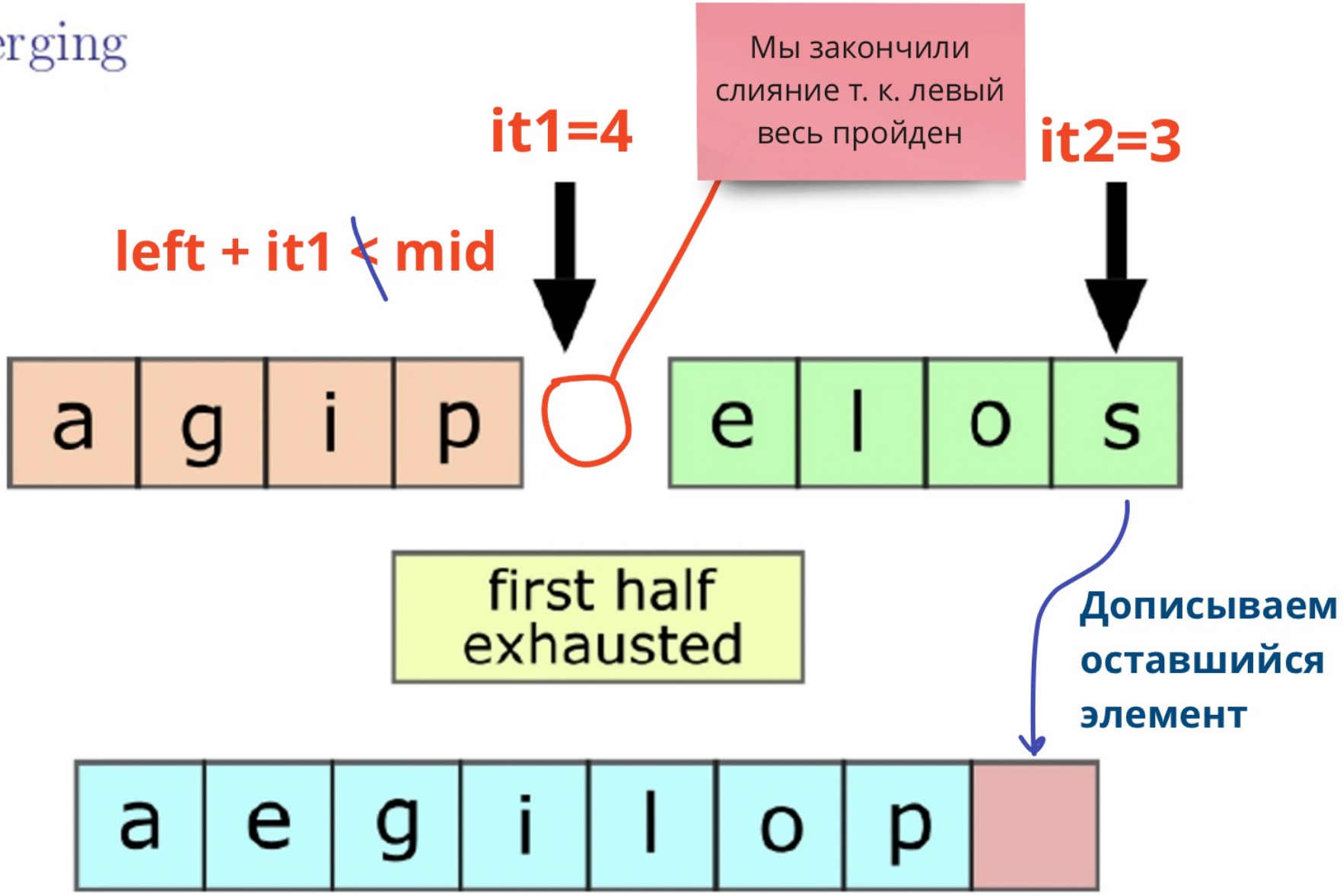
Rez[it1+it2]=A[mid+it1]

it1++ => left + it1 = mid

Merging



Merging



А если останется несколько?

Доливаем хвост

0	1	2	3	4
1	4	11	13	22

5	6	7	8	9
5	**23**	**34**	**35**	**56**

0	1	2	3	4	5	6	7	8	9
1	4	5	11	13	22	**23**	**34**	**35**	**56**

Доливаем хвост

Left **it1=5** **mid** **it2=1** **Right**

0	1	2	3	4
1	4	11	13	22

5	6	7	8	9
5	**23**	**34**	**35**	**56**

Доливаем

it1+it2

Пока mid + it2 < right
it2++

Rez

0	1	2	3	4	5	6	7	8	9
1	4	5	11	13	22	23	34	35	56

```
{Rez[it1 + it2] =
A[mid + it2];
it2++;}
```

Код Merge

```
function merge(a : int[n]; left, mid, right : int)
    it1 = 0
    it2 = 0
    result : int[right - left]

    while left + it1 < mid and mid + it2 < right
        if a[left + it1] < a[mid + it2]
            result[it1 + it2] = a[left + it1]
            it1 += 1
        else
            result[it1 + it2] = a[mid + it2]
            it2 += 1

    while left + it1 < mid
        result[it1 + it2] = a[left + it1]
        it1 += 1

    while mid + it2 < right
        result[it1 + it2] = a[mid + it2]
        it2 += 1

    for i = 0 to it1 + it2
        a[left + i] = result[i]
```

Код Merge

```
function merge(a : int[n]; left, mid, right : int)
    it1 = 0
    it2 = 0
    result : int[right - left]

    while left + it1 < mid and mid + it2 < right
        if a[left + it1] < a[mid + it2]
            result[it1 + it2] = a[left + it1]
            it1 += 1
        else
            result[it1 + it2] = a[mid + it2]
            it2 += 1

    while left + it1 < mid
        result[it1 + it2] = a[left + it1]
        it1 += 1

    while mid + it2 < right
        result[it1 + it2] = a[mid + it2]
        it2 += 1

    for i = 0 to it1 + it2
        a[left + i] = result[i]
```

Для просмотра левого и правого подмассивов

Вспомогательный массив

1 2

1 Пока не закончится левый

2 Пока не закончится правый

Записываем из левого

Записываем из правого

Доливаем левый хвост, если есть

Правый хвост

Из вспомогательного массива переписываем в изначальный

6 5 3 1 8 7 2 4

Сортировка СЛИЯНИЯМИ

ПО ВРЕМЕНИ

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

В данном случае алгоритм всегда работает одинаково!

ПО ПАМЯТИ

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

В данном случае алгоритм всегда работает одинаково!

Сортировка СЛИЯНИЯМИ

ПО ВРЕМЕНИ

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

В данном случае алгоритм всегда работает одинаково!

На слияние

В рекурсии на разбиение

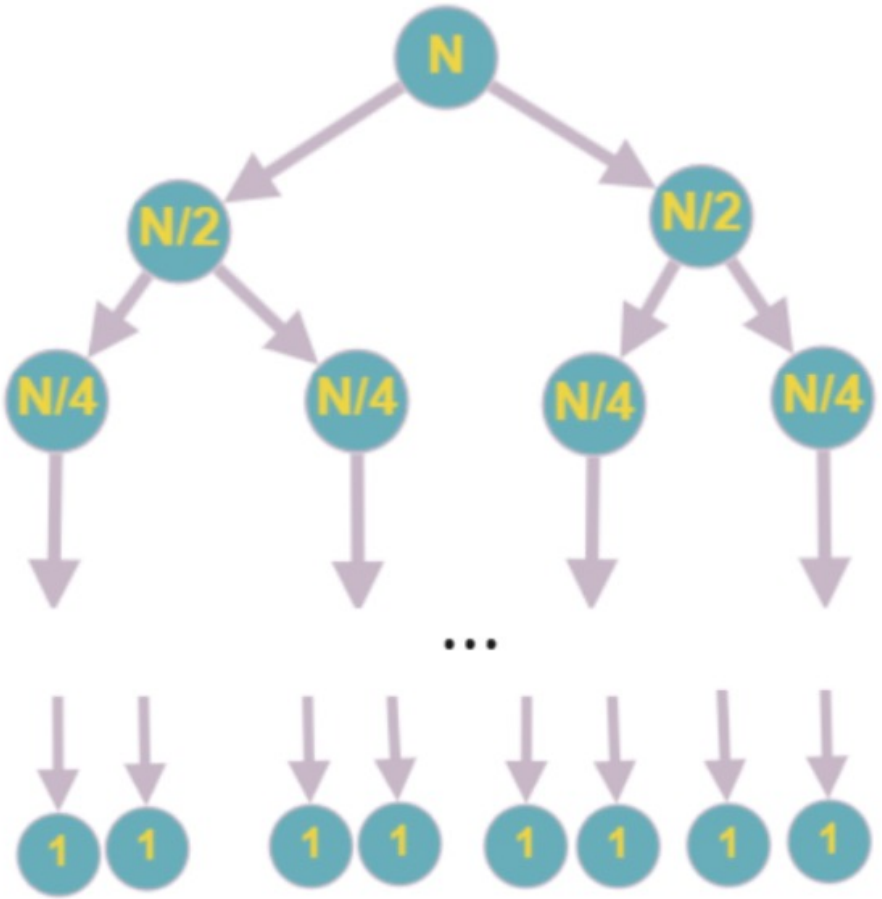
ПО ПАМЯТИ

Лучший	Средний	Худший
O(n)	O(n)	O(n)

В данном случае алгоритм всегда работает одинаково!

Нужен вспомогательный массив

Выведем временную сложность



1

2

4

...

N

0 уровень рекурсии

2^0

2^1

2^2

2^k

k уровень рекурсии

=> На каждом уровне рекурсии идет разбиение на 2^i частей

=> Разбиение идеи до одноэлементных массивов, всего разбиений К, К = $\log_2 n$, - глубина рекурсии

```
function mergeSortRecursive(a : int[n]; left, right : int):
    if left + 1 >= right
        return
    mid = (left + right) / 2
    mergeSortRecursive(a, left, mid)
    mergeSortRecursive(a, mid, right)
    merge(a, left, mid, right)
```

$T(n) = \begin{cases} O(1), & \text{при } n=1 \\ 2\,T(n/2) + O(n), & \text{при } n>1 \end{cases}$

```
function mergeSortRecursive(a : int[n]; left, right : int):
    if left + 1 >= right
        return
    mid = (left + right) / 2
    mergeSortRecursive(a, left, mid)
    mergeSortRecursive(a, mid, right)
    merge(a, left, mid, right)
```

O(1) при завершении

T(n/2)

T(n/2)

O(n) -> [0 N/2] + [N/2 N]

T(n)

пройдем весь массив

Для вычисления используем реккурентные соотношения

$T(n) = \begin{cases} O(1), при\ n=1 \\ 2\,T(n/2) + O(n), при\ n>1 \end{cases}$

Выведем по аналогии =>

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

$$=> f(n-1) = f(n-2) + 2$$

$$f(n-3) = f(n-4) + 2$$

$$f(n-4) = f(n-5) + 2$$

$$\dots$$

$$f(3) = f(2) + 2 = 2 * 2 + 2$$

$$= 2 * 3$$

$$f(2) = f(1) + 2 = 2 + 2 = 2 * 2$$

$$f(1) = 2$$

F(n) = n*2

$$T(n) = \begin{cases} O(1), n = 1 \\ 2T\left(\frac{n}{2}\right) + O(n), n > 1 \end{cases}$$

Тогда $T(\frac{n}{2}) = 2\,T(\frac{n}{2}/2) + O(\frac{n}{2}) = 2T(\frac{n}{4}) + O(\frac{n}{2})$

$2O(n)$

$\Rightarrow T(n) = 2 * (2\,T(\frac{n}{4}) + O(\frac{n}{2})) + O(n) = 4T(\frac{n}{4}) + 2O(\frac{n}{2}) + O(n)$

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + O(\frac{n}{4})$

$3O(n)$

$\Rightarrow 4*(2\,T(\frac{n}{8}) + O(\frac{n}{4})) + 2O(n) = 8T(\frac{n}{8}) + 4O(\frac{n}{4}) + 2O(n) \Rightarrow$

$= n = O(1)$

$\Rightarrow T(n) = 2^k\,T(\frac{n}{2k}) + kO(n) = 2^{\log_2 n}\,T(1) + \log_2 n * O(n) =$

$= n*O(1) + O(n)\log_2 n \sim O(n\log n)$

К – число разбиений в рекурсии, т.к. мы бьем массив пополам до одноэлементных массивов, то логично, что

$2^k = n$

$k = \log_2 n$, т.к. это глубина рекурсии, т.е. бьем ровно $\log_2 n$ раз наш массив пополам

$2^k = 2^{\log_2 n} = n$

Выведем сложность по памяти

```
function merge(a : int[n]; left, mid, right : int)
    it1 = 0
    it2 = 0
    result : int[right - left]

    while left + it1 < mid and mid + it2 < right
        if a[left + it1] < a[mid + it2]
            result[it1 + it2] = a[left + it1]
            it1 += 1
        else
            result[it1 + it2] = a[mid + it2]
            it2 += 1

    while left + it1 < mid
        result[it1 + it2] = a[left + it1]
        it1 += 1

    while mid + it2 < right
        result[it1 + it2] = a[mid + it2]
        it2 += 1

    for i = 0 to it1 + it2
        a[left + i] = result[i]
```

Выведем сложность по памяти

```
function merge(a : int[n]; left, mid, right : int)
    it1 = 0 +1
    it2 = 0 +1
    result : int[right - left]

    while left + it1 < mid and mid + it2 < right
        if a[left + it1] < a[mid + it2]
            result[it1 + it2] = a[left + it1]
            it1 += 1
        else
            result[it1 + it2] = a[mid + it2]
            it2 += 1

    while left + it1 < mid
        result[it1 + it2] = a[left + it1]
        it1 += 1

    while mid + it2 < right
        result[it1 + it2] = a[mid + it2]
        it2 += 1

    for i = 0 to it1 + it2
        a[left + i] = result[i]
```

На следующем шаге возврата

T(n) ~O(n)

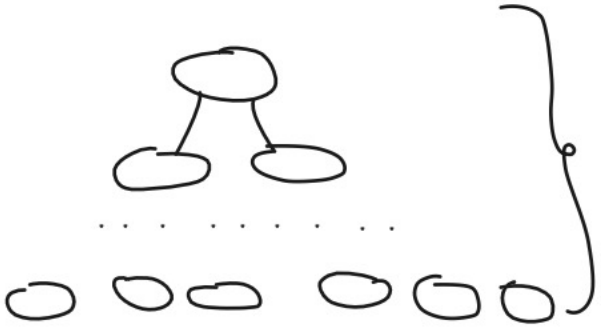
~O(n)

Result = a => +n

M(n) ~O(n)

Но все ли мы учли?

рекурсия



$n = \log_2 n$

=> Нужно хранить $\log_2 n$ точек возврата на самом глубоком уровне рекурсии

$M(n) = n + 2 + \log_2 n \sim O(n)$

Свойства O

1) Сложность суммы
2) Сложность произведения
3) Умножение на константу
4) Сумма с константой
5) Теорема о связи

Свойства O

C - константа

$T_1(n) = O(g_1(n))$ $T_2(n) = O(g_2(n))$

1) **Сложность суммы** $T_1+T_2 = O(\max(g_1(n), g_2(n)))$

Большая из функций порядка роста

2) **Сложность произведения** $T_1 * T_2 = O(g_1(n) * g_2(n))$

Произведение порядков

3) **Умножение на константу** $C * T_1 = O(g_1(n))$

При произведении на константу ее можно отбросить => не берем в расчет

4) **Сумма с константой** $T1 + C = O(g1(n))$

При сумме с константой –> ее можно отбросить

5) **Теорема о связи** θ, Ω, O

$$T(n) \sim \theta(n) \Leftrightarrow \begin{cases} T(n) \sim \Omega(g(n)) \\ T(n) \sim O(g(n)) \end{cases}$$

Тогда оценка ∃ ⇔ когда совпадает с оценкой снизу и сверху

Устойчивость ?

Исходный массив: 1 33_1 7_1 6 0 33_2 8 7_2 4 7_3

После сорт1: 0 1 4 6 7_3 7_1 7_2 8 33_2 33_1 (Не устойчивая)

Не сохраняется порядок

После сорт2: 0 1 4 6 7_1 7_2 7_3 8 33_1 33_2 (Устойчивая)

Порядок сохраняется

Устойчивая (стабильная) сортировка –

сортировка, которая не меняет относительный порядок сортируемых элементов, имеющих одинаковые ключи, по которым происходит сортировка

Зачем нужна устойчивость?

Вспомним:

- Сортируем не просто значения, а объекты, которые обладают данными характеристиками
- Потребность сортировки по двум критериям: например, по группе и по алфавиту (внутри групп по алфавиту)
- Данные могут иметь много повторяющихся значений ключа сортировки

Устойчивая

М3101
Ф1И1 59
Ф2И2 57
Ф2И3 45

М3102
ФИ 58
ФИ 57

ИЛЛЮСТРАЦИЯ проблемы: Вторая сортировка, изменяя порядок объектов с ключами с одинаковыми значениями - меняет порядок объектов, упорядоченных первой сортировкой

Не устойчивая

М3101 Ф1И1 59
М3102 ФИ 58
М3102 ФИ 57
М3101 Ф2И2 57
М3101 Ф2И3 45
Нарушена сортировка по группе

Устойчивость сортировки ВСТАВКАМИ

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

Устойчивая —когда идет процесс вставки элемента в отсортированную часть в строгий знак сравнения не дает поменять местами элементы с одинаковыми значениями.

Устойчивость сортировки ВСТАВКАМИ

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

Устойчивая —когда идет процесс вставки элемента в отсортированную часть в строгий знак сравнения не дает поменять местами элементы с одинаковыми значениями.

Если бы было A[i] >= key => не была бы устойчивой
То есть элементы с одинаковыми значениями менялись бы местами