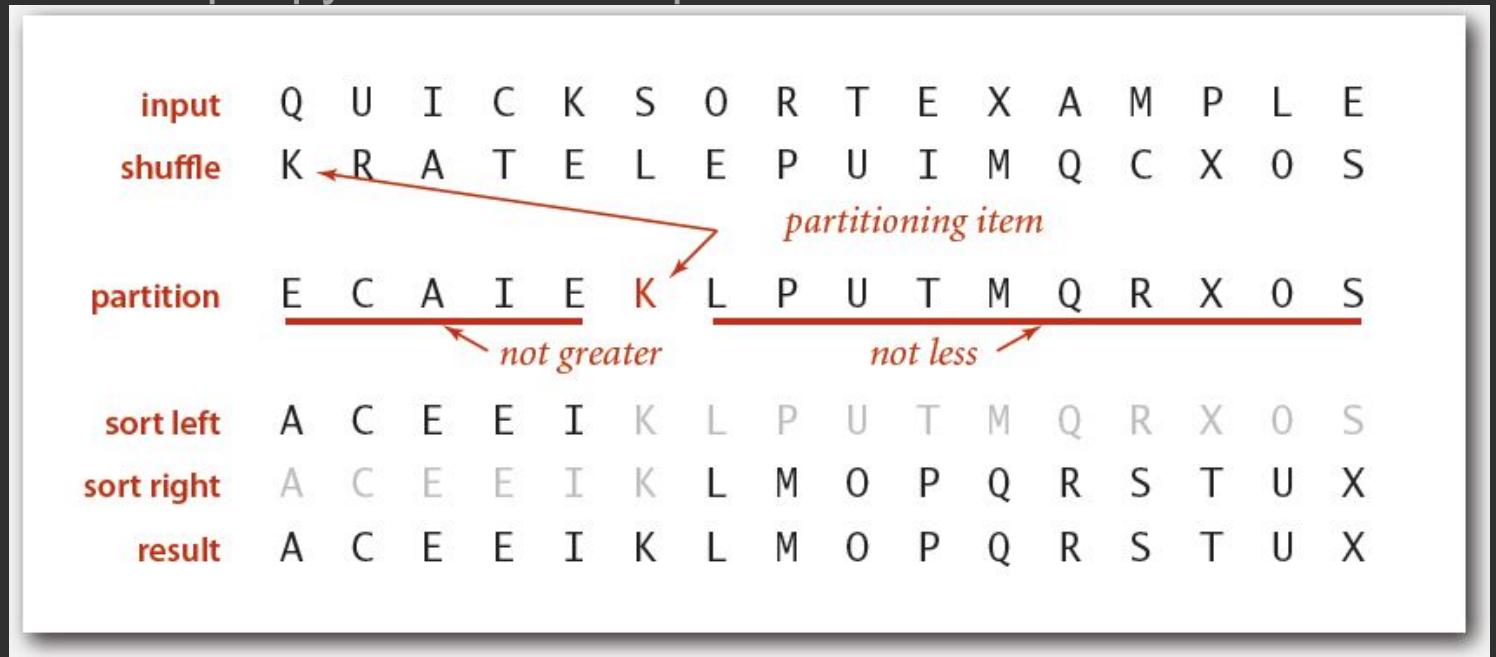
Быстрая сортировка. Quicksort

История про один из самых значимых алгоритмов 20 века

Quicksort (рекурсивен, как и Mergesort)

Основная идея:

- Перетасовываем массив
- Разделяем его так, что для любой j
  - Элемент a[j] находится на правильном месте в массиве
  - Слева от j нет большего элемента
  - Справа от j нет меньшего элемента
- Сортируем каждый участок рекурсивно
- И т.д. – рекурсивно сортируем левые и правые части



Пример quicksort

Повторяем до тех пор, пока указатели i и j не пересекутся.

- Проходим указателем i слева-направо до тех пор, пока вып усл-ие a[i] < a[lo]
- Проходим указателем j справа-налево до тех пор, пока вып усл-ие a[j] > a[lo]
- Меняем местами a[i] и a[j]

Пример quicksort

Выбираем K в качестве разделителя.

Двигаем указатель i слева-направо, до тех пор пока не найдем элемент, больший чем разделитель K.

И двигаем указатель j справа-налево, пока не найдем элемент меньший разделителя K.

В данном примере:

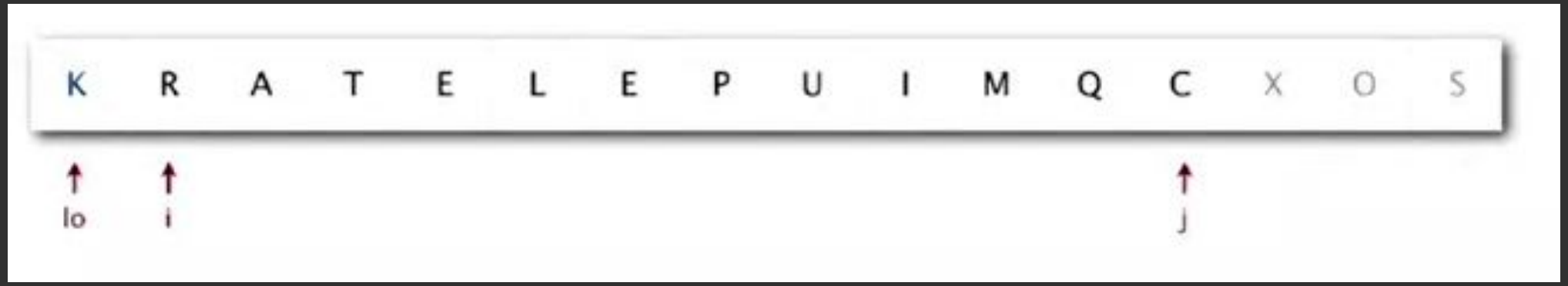
i останавливается сразу, так как K > R и условие a[i]<a[lo] не выполняется

Пример quicksort

Выбираем K в качестве разделителя.

Двигаем указатель i слева-направо, до тех пор пока не найдем элемент, больший чем разделитель K.

И двигаем указатель j справа-налево, пока не найдем элемент меньший разделителя K.



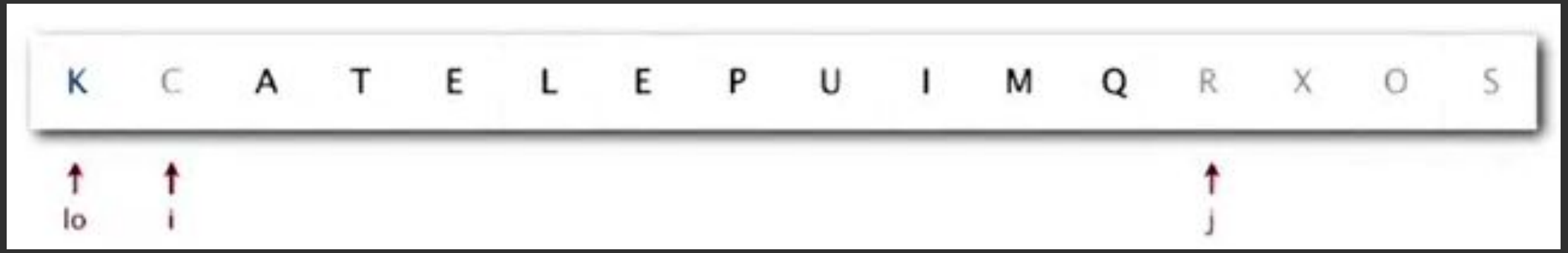
В данном примере:

j останавливается в C, так как C < K и условие a[j]>a[lo] не выполняется

Пример quicksort

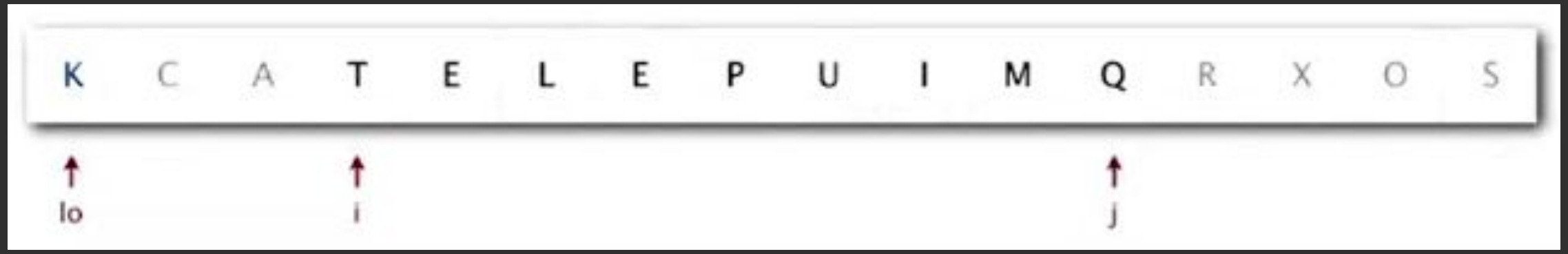
Мы нашли позиции a[i] и a[j], удовлетворяющие условиям относительно a[lo].

Меняем a[i] и a[j] местами



Пример quicksort

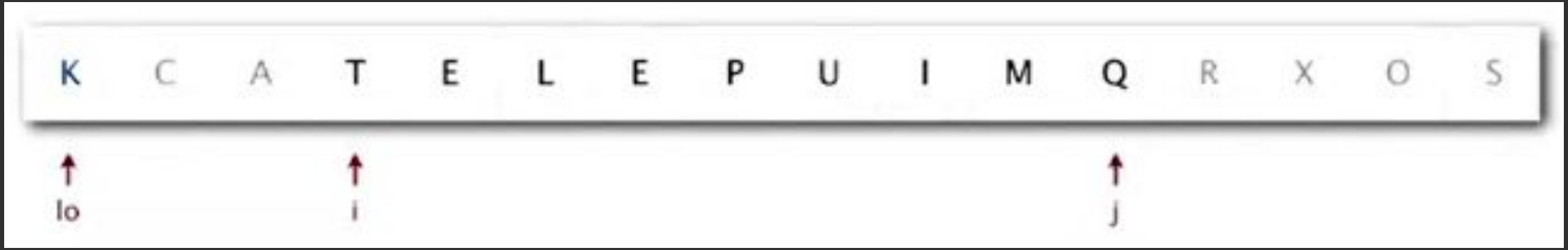
Меняем стартовые позиции - i увеличиваем на 1 и j уменьшаем на 1



Пример quicksort

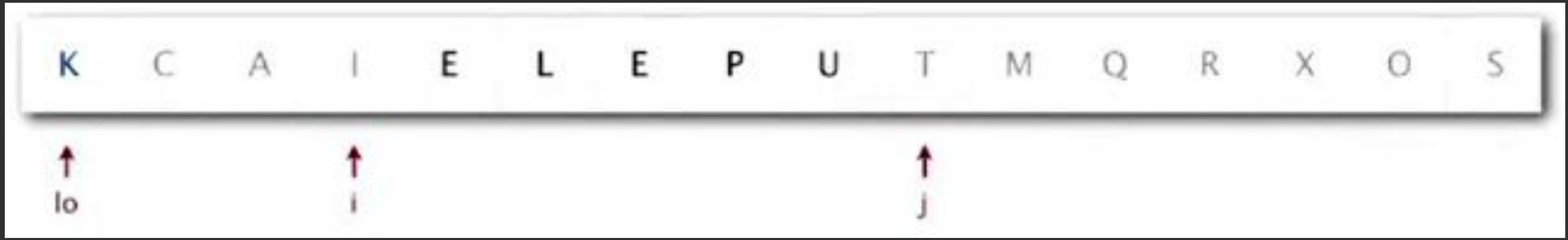
Ищем a[i] и a[j], удовлетворяющие условиям.

Это a[i] = T и a[j] = I



Пример quicksort

Меняем их местами



Пример quicksort

Продолжаем по аналогии

Меняем местами

Пример quicksort

Увеличиваем i и уменьшаем j

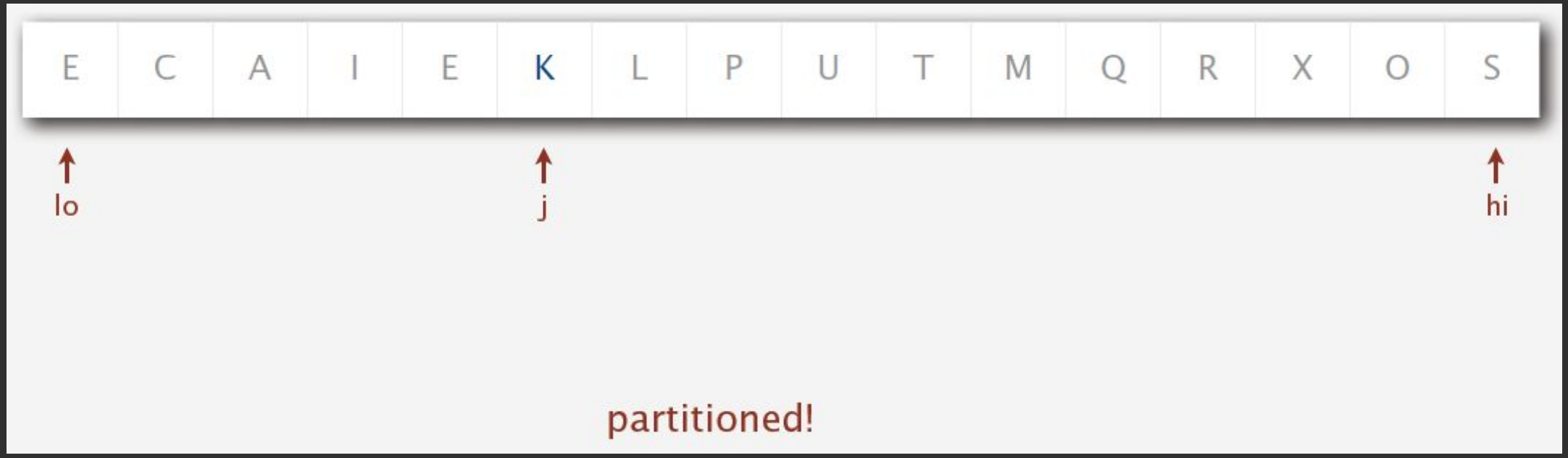
Процесс разделения закончен, так как i и j пересеклись

Пример quicksort

Фаза 2 (указатели пересеклись)

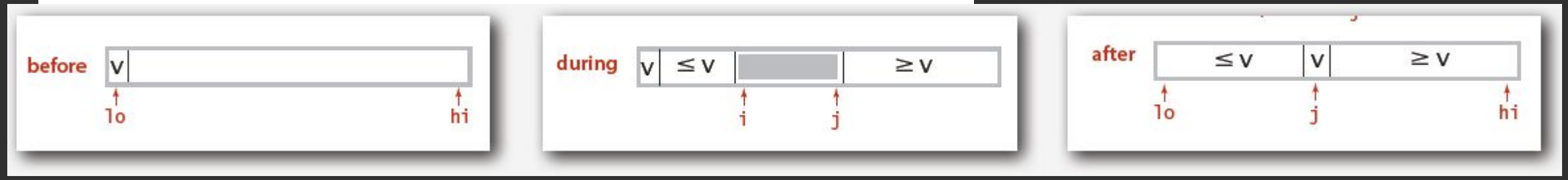
Меняем местами a[lo] и a[j]

Теперь слева от K только меньшие элементы, в справа только большие



Quicksort реализация

```
private static int partition(IComparable[] a, int lo, int hi)
{
    int i = lo, j = hi + 1;
    while (true)
    {
        while (less(a[++i], a[lo]))
            if (i == hi) break;    //ищем элемент слева для перестановки
        while (less(a[lo], a[--j]))
            if (j == lo) break;  //ищем элемент справа для перестановки
        if (i >= j) break;        //проверяем пересечение указателей
        exch(a, i, j);            //переставляем
    }
    exch(a, lo, j);              //переставляем части
    return j;                    //возвращаем индекс элемента
                                 //который теперь находится на своем месте
}
```



Состояние массива до, во время и после

Quicksort реализация

А перетасовывается для того, чтобы гарантировать нормальную производительность

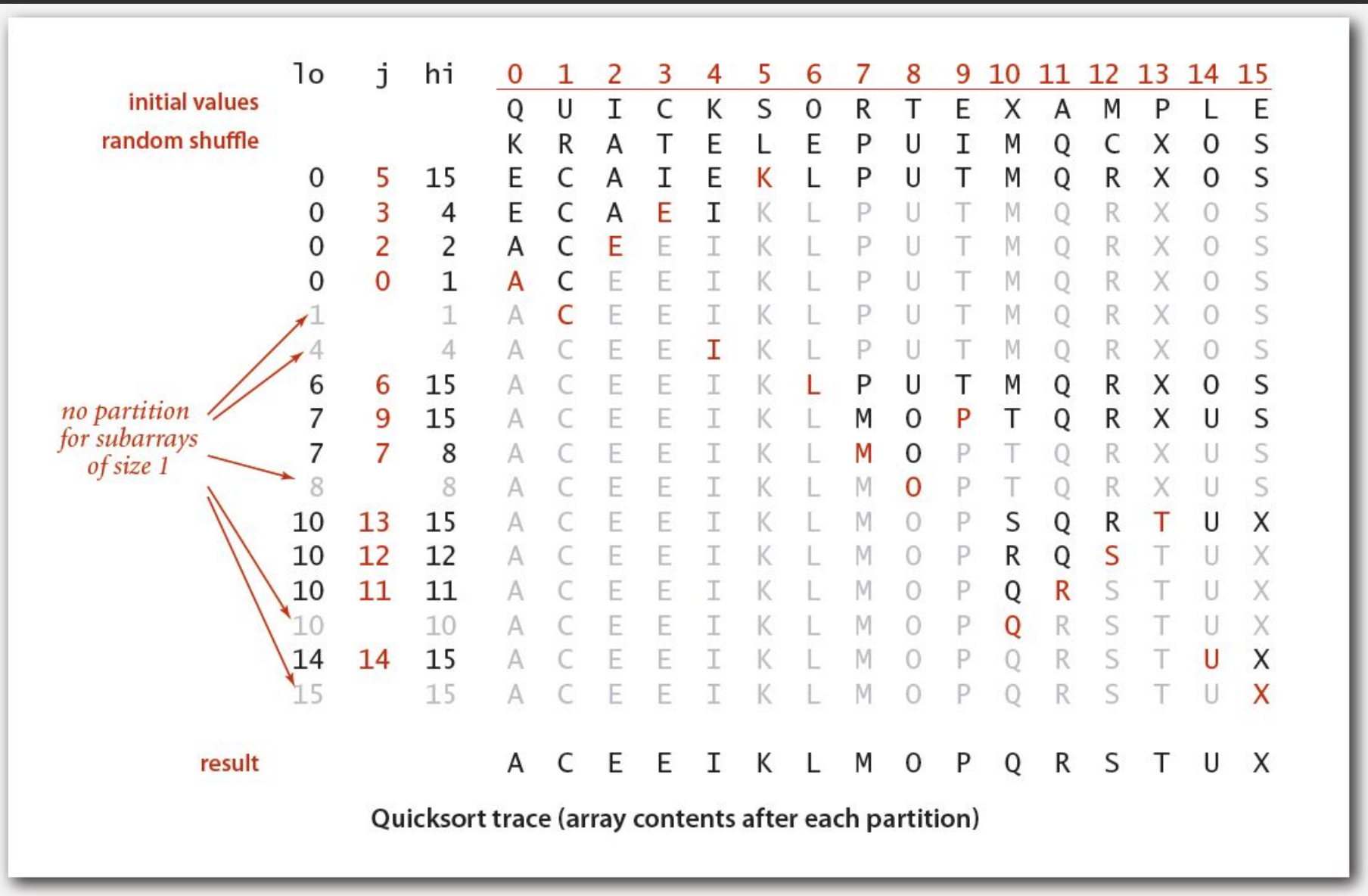
```
class quicksort
{
    private static int partition(IComparable[] a, int lo, int hi)...

    public static void sort(IComparable[] a)
    {
        Shuffle(a);                 //перетасовываем массив a (любым ивестным способом)
        sort(a, 0, a.Length - 1);   //вызываем quicksort
    }

    private static void sort(IComparable[] a, int lo, int hi)
    {                                       //рекурсивно сортируем
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j - 1);
        sort(a, j + 1, hi);
    }

    //далее можно добавить доп методы или брать их из библиотек
    //}
```

Quicksort пошагово



	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

no partition for subarrays of size 1

Quicksort trace (array contents after each partition)

Quicksort особенности реализации

Разделение. Использование дополнительного массива делает разделение более простым и стабильным, но требует доп памяти.

Quicksort, в отличии от Mergesort, использует только 1 массив.

Удаление циклов. Проверка пересечения указателей i и j, немного сложнее, чем кажется.

Остаемся в границах массива. Проверка j == lo не нужна, так как там есть разделитель (a[lo]), но i == hi нужна.

Сохранение случайности элементов. Перетасовка массива необходима для гарантии производительности.

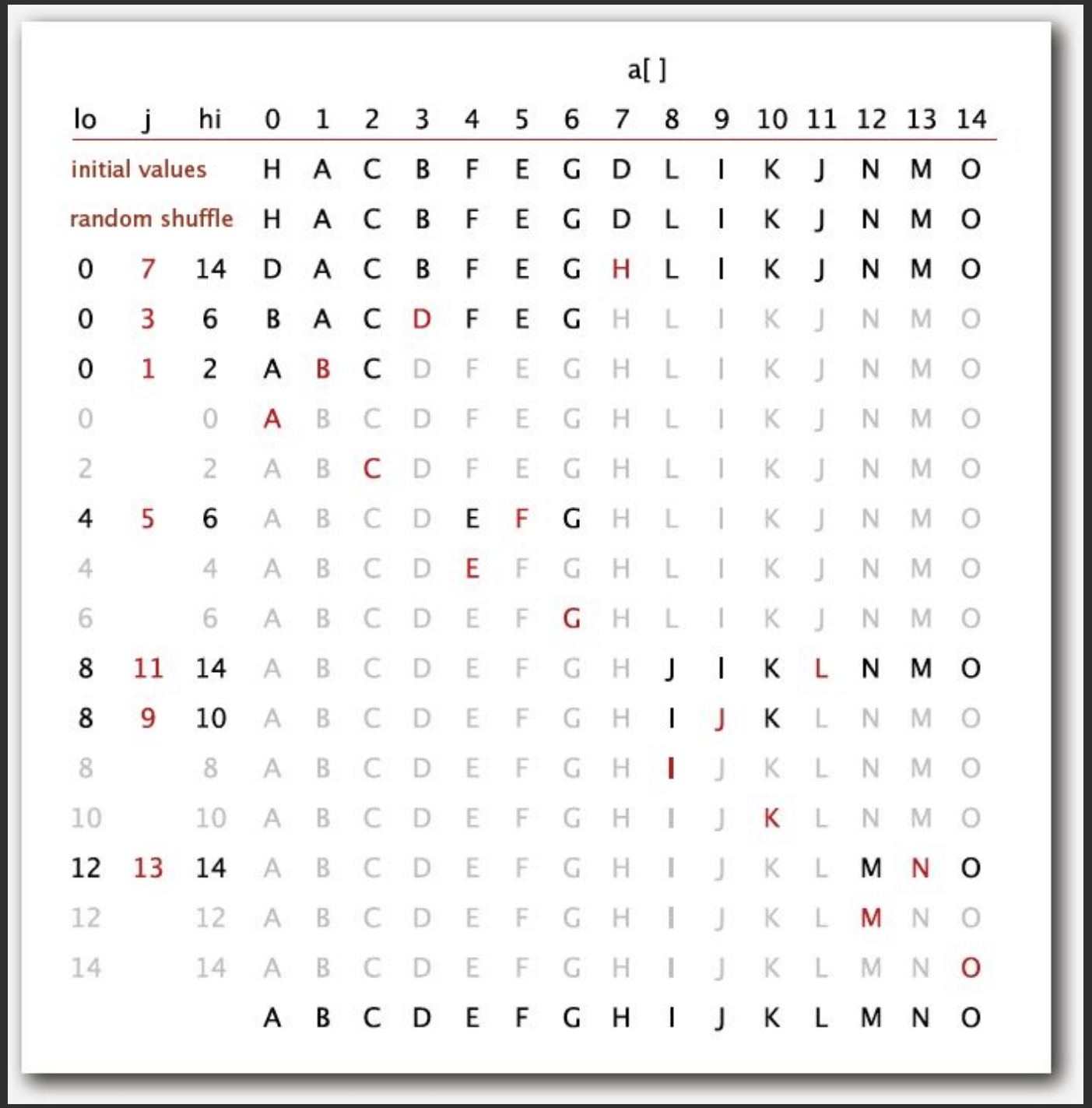
Одинаковые ключи. Когда есть одинаковые элементы, лучше останавливаться на элементах, равных разделителю.

Quicksort эмпирический анализ

Оценки производительности

- Домашний ПК – 10^8 операций/сек
- Суперкомпьютер – 10^12 операций/сек

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant



			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort анализ в лучшем случае

Лучший случай. Число сравнений ~ N*lgN

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort анализ в худшем случае

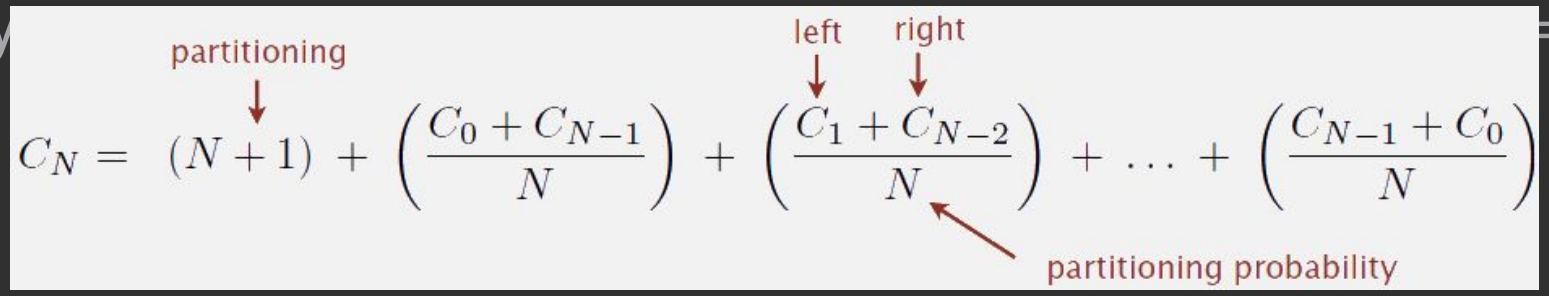
Лучший случай. Число сравнений ~ ½ $*N^2$

Если мы предварительно отсортируем массив, то такой случай маловероятен.

Quicksort. Анализ с среднестатистическом случае

Предположение. Среднее число сравнений C_N для сортировки quicksort массива N различных элементов ~ 2*N*lnN (и число перестановок ~ 1/3*N*lnN)

Док-во. C_N у = 0 и N>=2:



Умножаем обе

$$NC_N = N(N+1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

(I)

Записываем (I) для

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

Quicksort. Анализ с среднестатистическом случае

Док-во. Переставляем слагаемые и делим всё на N*(N+1)

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Рекурсивно подставляем данное соотношение:

$$\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\end{aligned}$$

previous equation

substitute previous equation

Quicksort. Анализ с среднестатистическом случае

Док-во. Переставляем слагаемые и делим всё на N*(N+1)

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Рекурсивно подставляем данное соотношение:

$$\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\end{aligned}$$

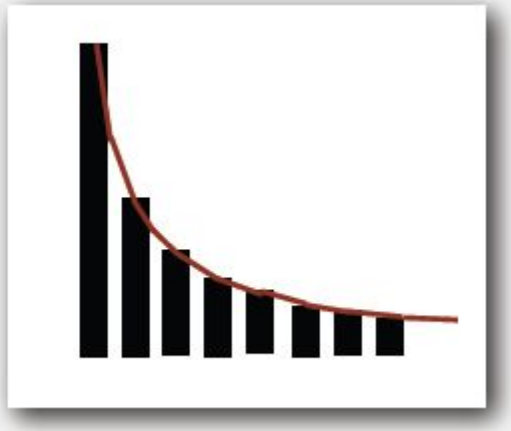
previous equation

substitute previous equation

Quicksort. Анализ с среднестатистическом случае

Док-во. Вычисляем сумму с помощью интеграла

$$
\begin{aligned}
C_N &= 2(N+1)\left(\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\cdots\frac{1}{N+1}\right) \\
&\sim 2(N+1)\int_3^{N+1}\frac{1}{x}\,dx
\end{aligned}
$$



Получаем желаемый результат:

$$C_N \sim 2(N+1)\ln N \approx 1.39N\lg N$$

Quicksort. Выводы по производительности.

Худший случай. Число сравнений квадратично

- $N + (N - 1) + (N - 2) + \ldots + 1 \sim \frac{1}{2} N^2$.
- Такой случай маловероятен, из-за предварительной случайной перетасовки

Среднестатистический случай. Число сравнений ~ 1.39*N*lgN

- На 39% больше сравнений чем в Mergesort
- Но быстрее Mergesort из-за меньшей работы с памятью

Случайная сортировка.

- Случайность спасает от худшего случая
- Основа для мат модели, которая может быть оценена с помощью экспериментов.

Стоит учитывать что! Многие книжные реализации работают квадратично, если:

- Массив отсортирован (по убыванию или возрастанию)
- Имеет много повторяющихся элементов (даже если случайно перетасован)

Quicksort. Свойства

Предположение. Quicksort сортирует «на месте», т.е. не использует доп массивов.

Док-во. 1) Разделение. Требует фиксированной доп памяти

2) Глубина рекурсии. Логарифмическая доп память (с высокой вероятностью)

Предположение. Quicksort нестабилен.

Док-во.

i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

Quicksort. Свойства

Предположение. Quicksort сортирует «на месте», т.е. не использует доп массивов.

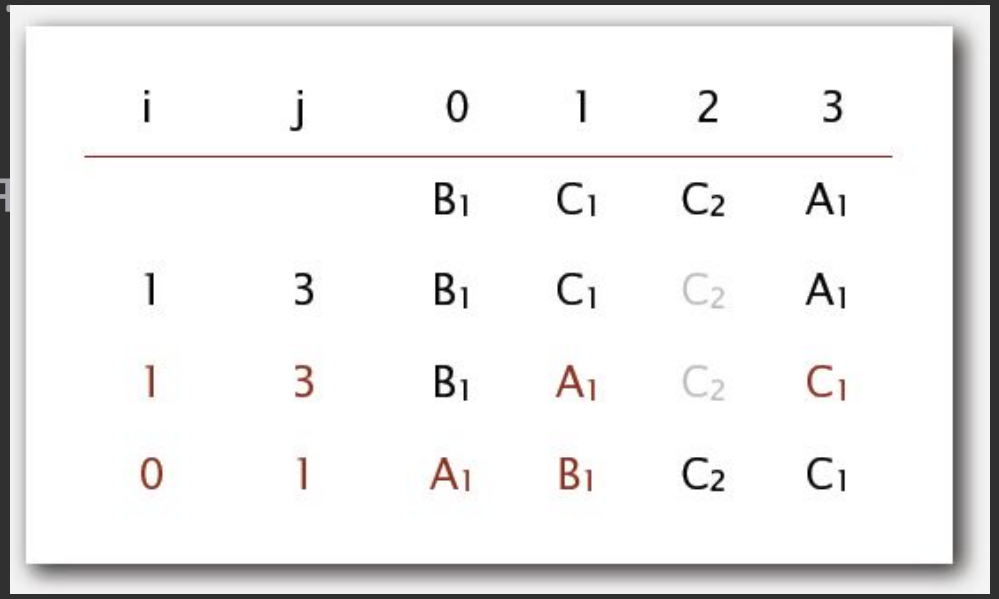
Док-во. 1) Разделение. Требует фиксированной доп памяти

2) Глубина рекурсии. Логарифмическая доп память (с высокой вероятностью)

Предположение. Quicksort нестабилен.

Док-во. Разделение совершает

Перестановки на большие расстояния



i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

Quicksort. Возможные улучшения

Сортировка со вставкой для маленьких подмассивов

- Quicksort жрет слишком много памяти для маленьких массивов
- Предел для использования Quicksort – от 10 элементов

Java-code. For C# change Comparable[] to IComparable[]

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

Quicksort. Возможные улучшения

Медиана

- Предполагаем, что разделитель находится примерно в середине.
- Берем несколько элементов и определяем их медиану
- Медиана 3 (случайных) элементов, немного снижает число сравнений, но увеличивает число перестановок

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;

   int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
   swap(a, lo, m);

   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

ПРОСЫПЕМСЯ!

Предполагаемое время работы алгоритма quicksort с случайной перетасовкой на уже отсортированном массиве:

Линейно

N*lgN

Квадратично

Экспоненциально

Выборка

Дан массив N элементов, найти k-ый наибольший элемент

Прим. Минимум (k=0), максимум (k=N-1), медиана (k = N/2)

Приложения:

- Статистика
- Найти наибольшие k элементов

Теоретические знания для реализации алгоритмва:

- N*logN верхняя граница – сортируем массив, макс элт – последний, мин - первый
- N верхняя граница для k=1,2,3 – если k мало, то время работы проп-но N
- N нижняя граница – достаточно просмотреть весь массив

Из всего этого можно сделать вывод – нужен линейный алгоритм.

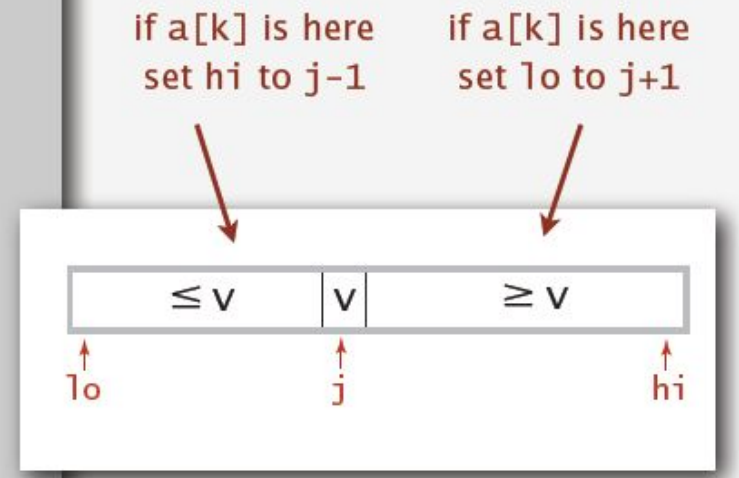
Существует ли линейный алгоритм или выборка так же сложна как сортировка.

Quick-select. Быстрая выборка

Разделяем массив, так что:

- a[j] находится на месте
- Слева от j нет больших элементов
- Справа от j нет меньших элементов
- Повторяем в одном подмассиве, в зависимости от j; заканчиваем когда j ра

```
public static Comparable select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
       int j = partition(a, lo, hi);
       if      (j < k) lo = j + 1;
       else if (j > k) hi = j - 1;
       else            return a[k];
    }
    return a[k];
}
```



Quick-select: Математический анализ

Предположение. Quick-select в среднем работает линейное время.

Док-во. На каждом шаге разделения массив делится примерно пополам:

N + N/2 + N/4 + … + 1 ~ 2N сравнений.

Формальный анализ, подобный тому, что был в quicksort:

$$C_N = 2N + k \ln (N / k) + (N - k) \ln (N / (N - k))$$

(2 + 2 ln 2) N to find the median

P.S. Quick-select использует ~ ½ N^2 сравнений в худшем случае, но (как с quicksort) случайная перетасовка дает некую защиту от этого.

ПРОСЫПАЕМСЯ!!!

Каково ожидаемое время работы для поиска медианы используя quick-select со случайной перетасовкой?

Постоянно

Логарифмично

Линейно

N*lgN

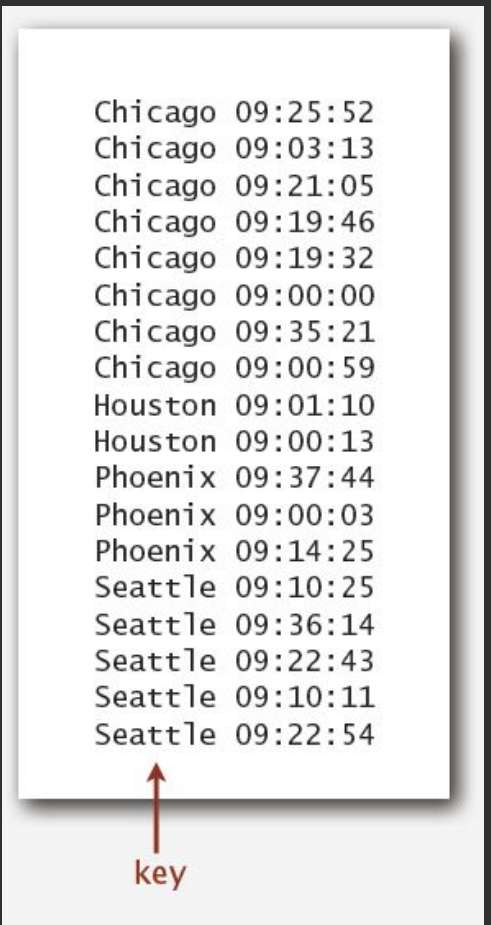
Повторяющиеся ключи (элементы)

Часто, задачей сортировки является группировка записей с одинаковыми ключами

- Сортировка по возрасту
- Удаление одинаковых писем из ящика
- Прибытий автобусов на остановки

Типичные свойства подобных приложений:

- Большие массивы данных
- Маленькое число ключей

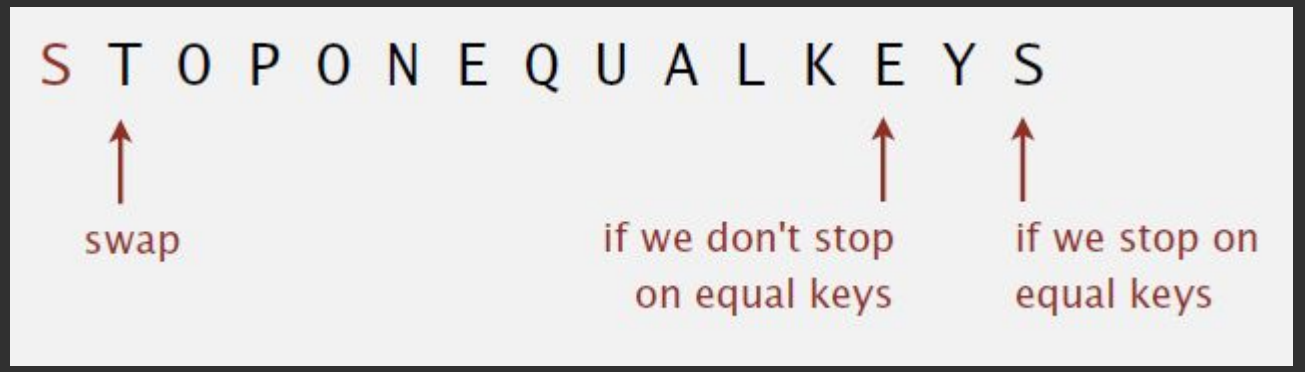


Повторяющиеся ключи

Mergesort для повторяющихся ключей. Число сравнений между ½*N*lgN и N*lgN

Quicksort для повторяющихся ключей:

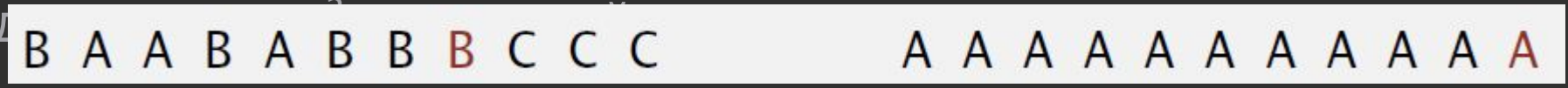
- Алгоритм квадратичен, но не в случае когда разделение останавливается на одинаковых ключах!



Повторяющиеся ключи. Проблема

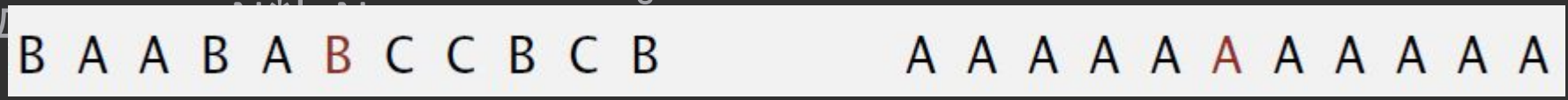
Ошибка. Все элементы, равные разделителю, размещаются с одной стороны

Послед

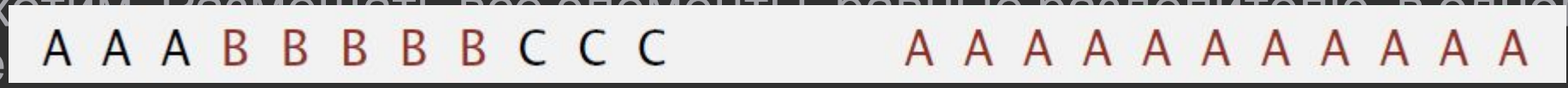


Рекомендуется. Останавливать поиск на элементах равных разделителю.

Послед



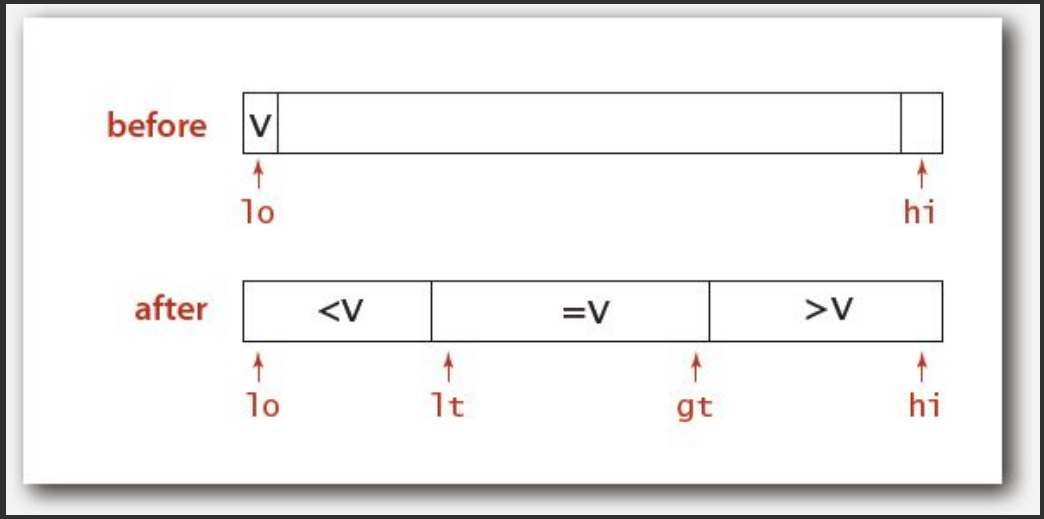
Чего хотим. Разместить все элементы, равные разделителю, в одном месте



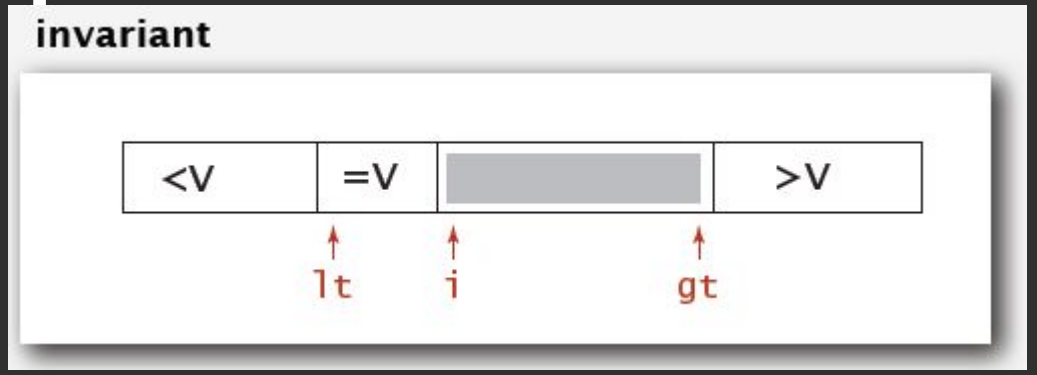
Разделение в 3 направлениях

Цель. Разделять массив на 3 части таким образом, что:

- Элементы между lt и gt равны разделителю v
- Слева от lt нет больших элементов
- Справа от gt нет меньших элементов



Разделение в 3 направлениях. пример



Пусть v разделитель a[lo]

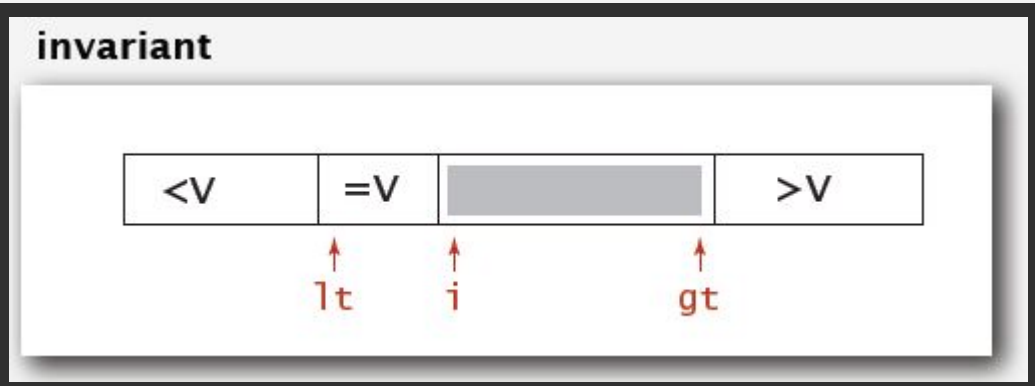
Сканируем i слева-направо.

- (a[i] < v): меняем местами a[lt] и a[i]; увеличиваем lt и i
- (a[i] > v): меняем местами a[gt] и a[i]; уменьшаем gt
- (a[i] == v): увеличиваем i

Разделение в 3 направлениях. пример

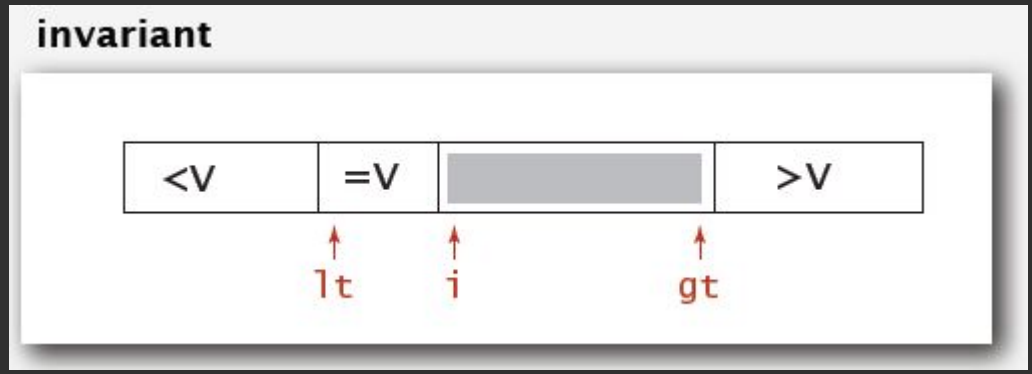
Указатель i находится слева от неизвестной (не просмотренной области).

Слева от lt всё меньшее v, между lt и i – равное v, между i и gt – неизвестное, больше gt – большее v



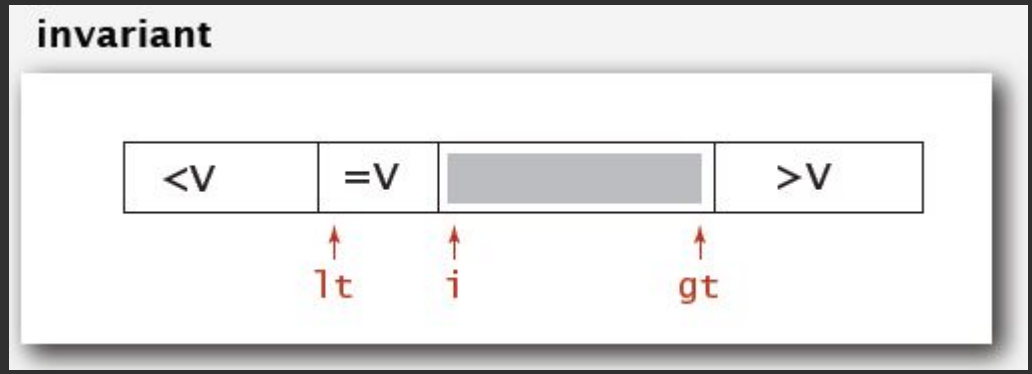
Разделение в 3 направлениях. пример

Увеличиваем i. А меньше чем P, поэтому переносим её за lt. При этом увеличиваем lt и i



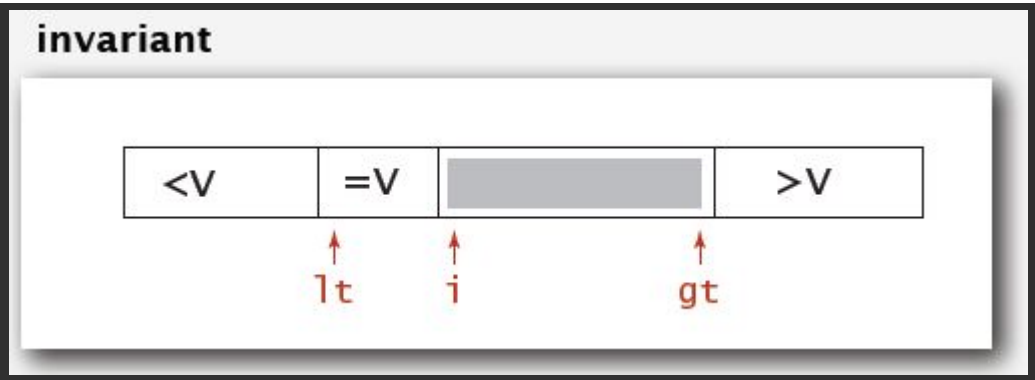
Разделение в 3 направлениях. пример

Теперь i указывает на B, B тоже меньше чем P, переносим элемент. Увеличиваем lt и i



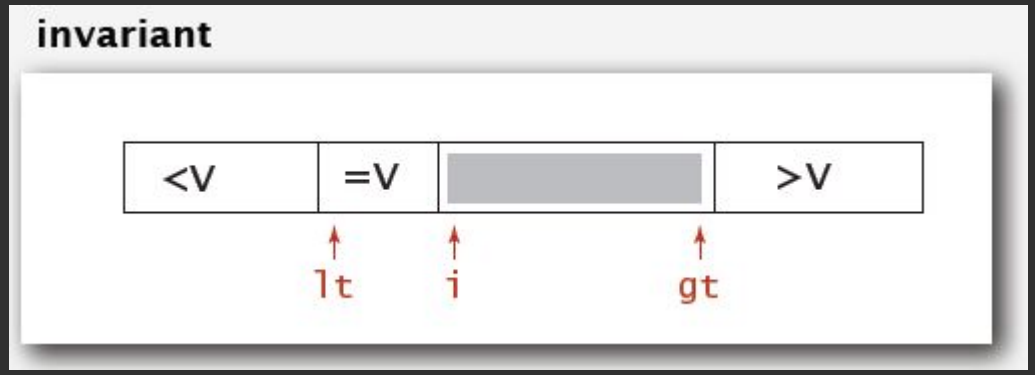
Разделение в 3 направлениях. пример

X больше чем P, меняем его местами с gt, уменьшаем gt.



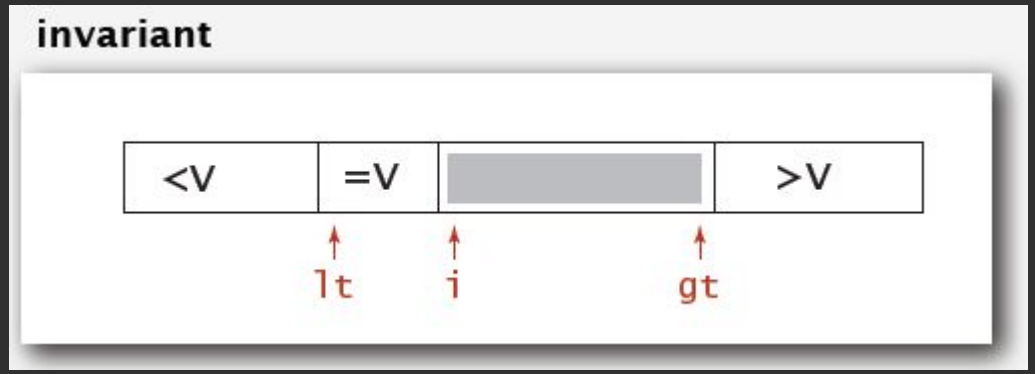
Разделение в 3 направлениях. пример

Та же ситуация, меняем местами с gt, при этом уменьшаем gt на 1



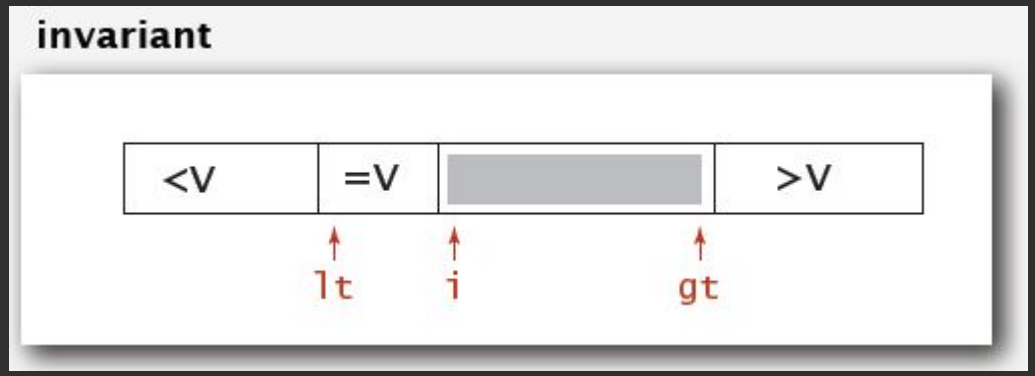
Разделение в 3 направлениях. пример

То же самое



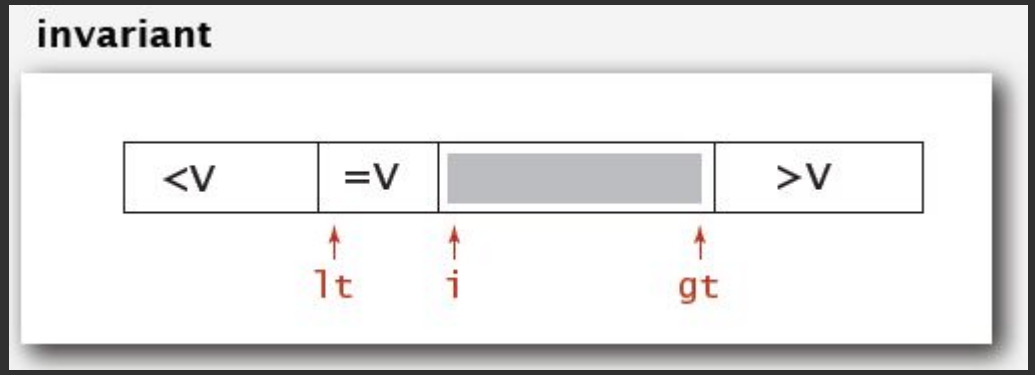
Разделение в 3 направлениях. пример

C < P, уносим за lt



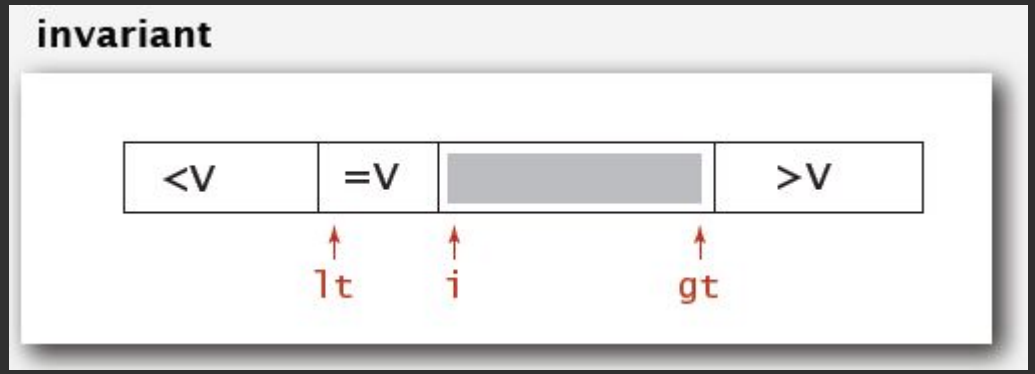
Разделение в 3 направлениях. пример

P > W, меняем местами с gt.



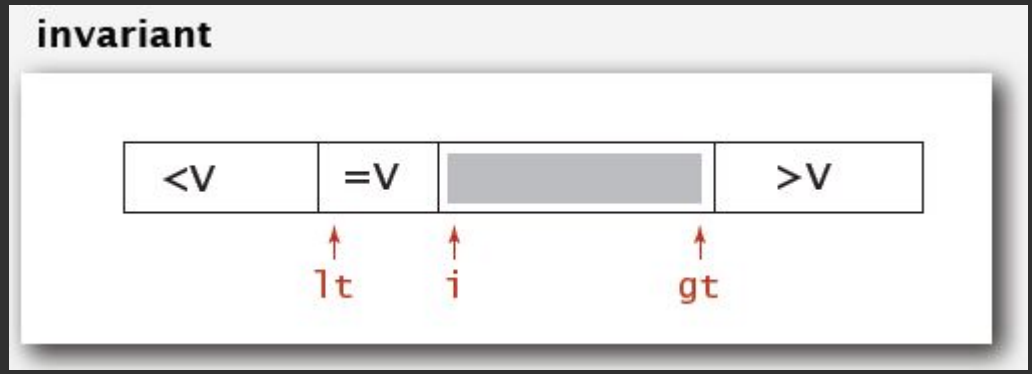
Разделение в 3 направлениях. пример

Теперь i указывает на элемент, равный разделителю. Просто увеличиваем i

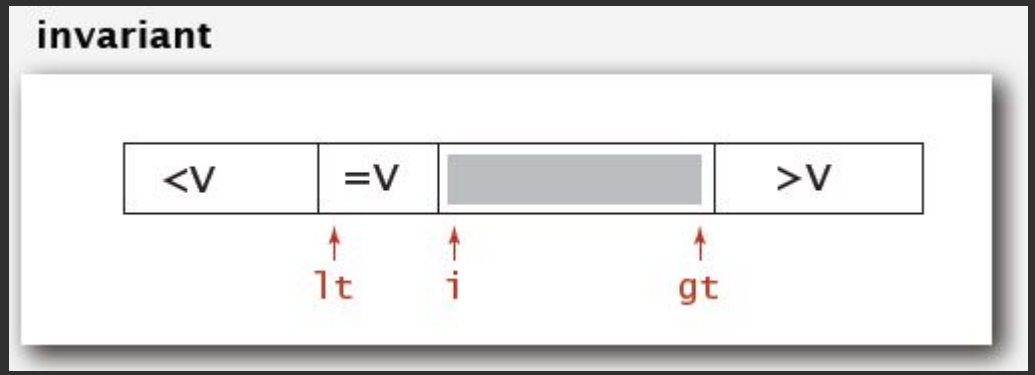


Разделение в 3 направлениях. пример

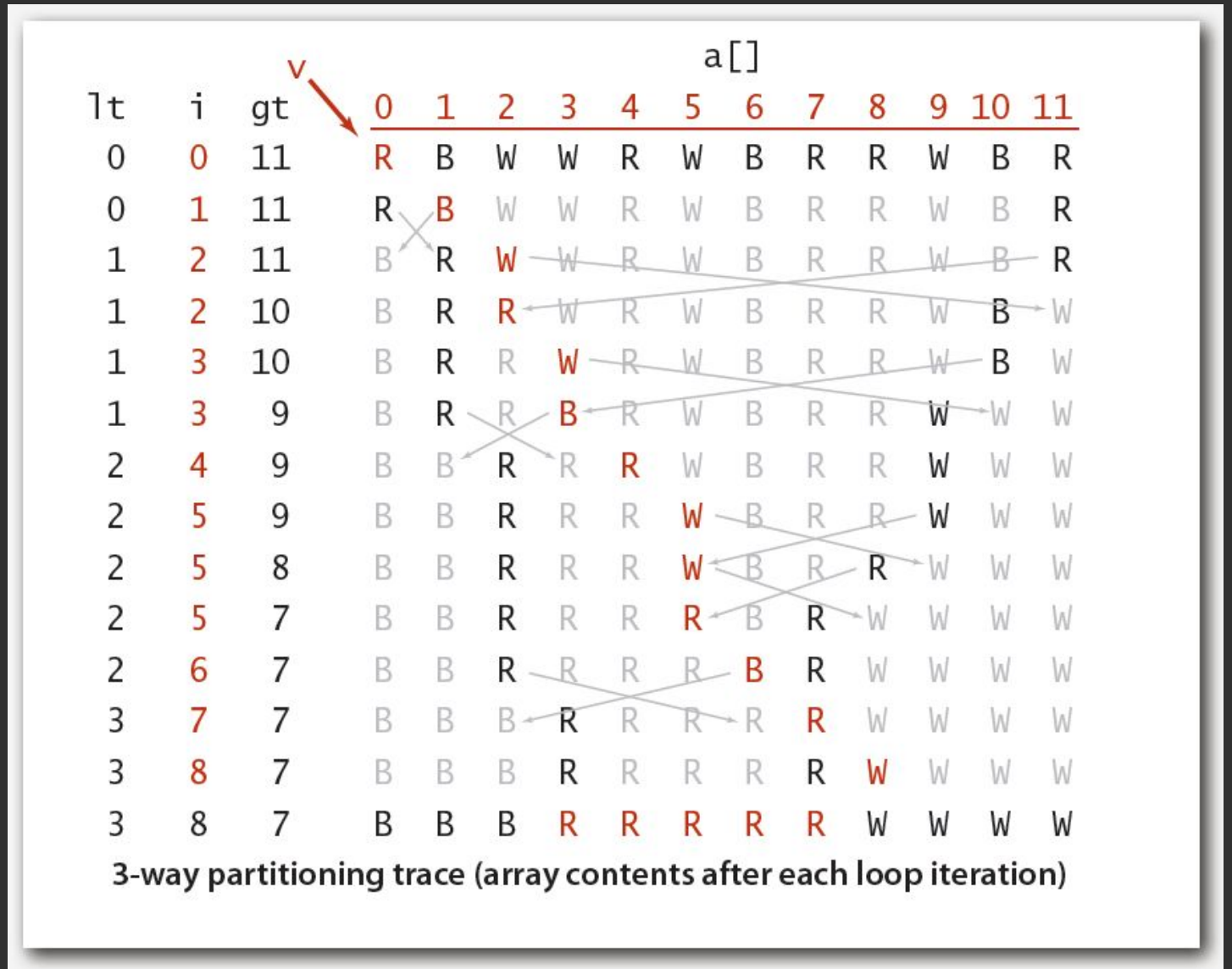
Снова P = P, делаем всё аналогично, пока не получим более-менее отсортированный массив.



Разделение в 3 направлениях. пример



Разделение в 3 направлениях. Пошагово



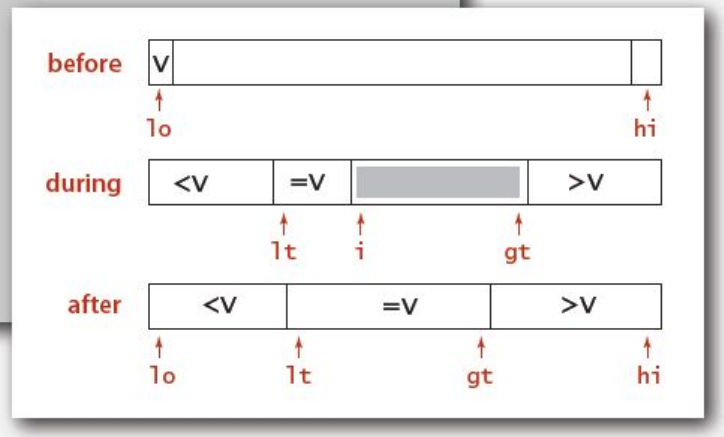
lt	i	gt	v	a[] 0	1	2	3	4	5	6	7	8	9	10	11
0	0	11		R	B	W	W	R	W	B	R	R	W	B	R
0	1	11		R	B	W	W	R	W	B	R	R	W	B	R
1	2	11		B	R	W	W	R	W	B	R	R	W	B	R
1	2	10		B	R	R	W	R	W	B	R	R	W	B	W
1	3	10		B	R	R	W	R	W	B	R	R	W	B	W
1	3	9		B	R	R	B	R	W	B	R	R	W	W	W
2	4	9		B	B	R	R	R	W	B	R	R	W	W	W
2	5	9		B	B	R	R	R	W	B	R	R	W	W	W
2	5	8		B	B	R	R	R	W	B	R	R	W	W	W
2	5	7		B	B	R	R	R	R	B	R	W	W	W	W
2	6	7		B	B	R	R	R	R	B	R	W	W	W	W
3	7	7		B	B	B	R	R	R	R	R	W	W	W	W
3	8	7		B	B	B	R	R	R	R	R	W	W	W	W
3	8	7		B	B	B	R	R	R	R	R	W	W	W	W

3-way partitioning trace (array contents after each loop iteration)

Разделение в 3 направлениях. реализация

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int lt = lo, gt = hi;
   Comparable v = a[lo];
   int i = lo;
   while (i <= gt)
   {
      int cmp = a[i].compareTo(v);
      if      (cmp < 0) exch(a, lt++, i++);
      else if (cmp > 0) exch(a, i, gt--);
      else              i++;
   }

   sort(a, lo, lt - 1);
   sort(a, gt + 1, hi);
}
```



Повторяющиеся ключи. Нижняя граница

Вывод – Quicksort со случайным перетасовыванием и разделением в 3 направлениях становится линейным алгоритмом, а не N*lgN

ПРОСЫПАЕМСЯ!

Использование разделения в 3 направлениях с Quicksort наиболее эффективно в случаях кгода входные данные имеют следующее свойство

Все элементы различны

Есть несколько различных элементов

Элементы расположены в порядке возрастания

Элементы расположены в порядке убывания

Сортировки в повседневной жизни

Алгоритмы сортировки распространены в огромном количестве приложений:

- Сортировка имен
- Библиотеке песен
- Выдаче google
- RSS-ленте, для сортировки новостей

- Поиск медианы
- Бинарный поиск в БД
- Поиск повторяющихся email’ов

- Сжатие данных
- Компьютерная графика
- Вычислительная биология

Какой алгоритм использовать?

Алгоритмов больше, чем мы рассмотрели:

Внутренняя сортировка:

- Со вставкой, с выбором, пузырёк, shaker sort
- Quicksort, mergesort, heapsort, samplesort, сортировка Шелла.
- Solitaire sort, red-black sort, splaysort, Yaroslavskiy sort, psort,…

Внешнаяя сортировка. Poly-phase mergesort, cascade-merge, oscillating sort.

Строчные сортировки. Distribution, MSD, LSD, 3-way string quicksort.

Параллельные сортировки.

- Bitonic sort, batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPUsort

Какой алгоритм использовать?

Приложения имеют множество параметров/особенно

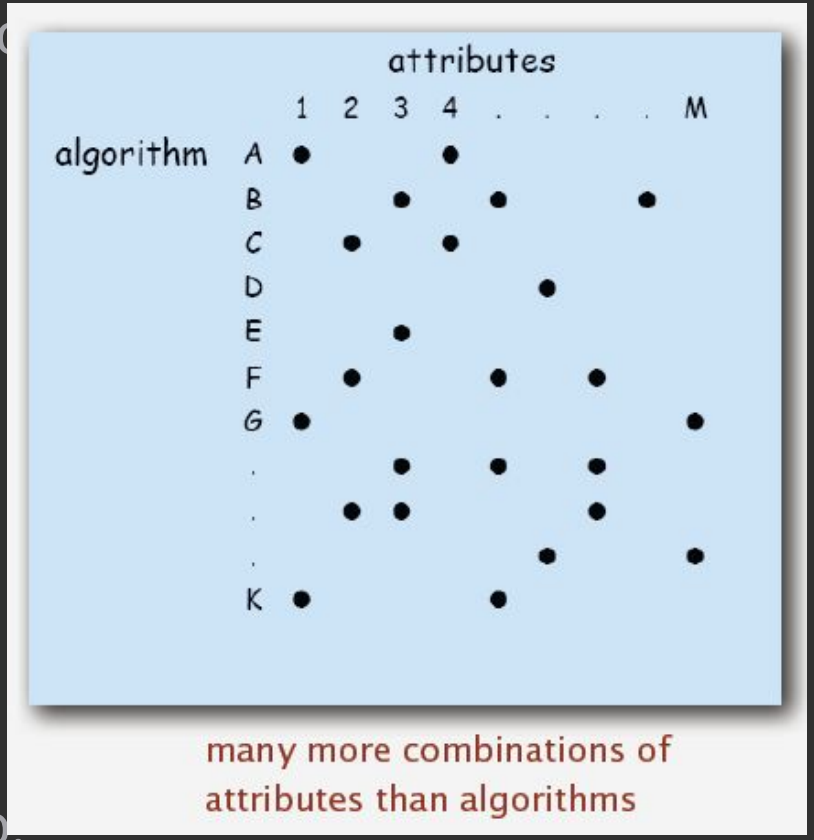
- Стабильность
- Параллельность
- Определенность
- Различие ключей
- Различие типов ключей
- Массив или ссылочный список
- Случайность расположения элементов

И т.д.

Можно использовать одну сортировку или комбинацию.

НО – учесть все параметры и особенности практически невозможно

Вывод – нет единого «правильного» метода сортировки!



Вывод по сортировкам

	inplace?	stable?	worst	average	best	remarks
selection	✔		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	*N* exchanges
insertion	✔	✔	$N^2 / 2$	$N^2 / 4$	N	use for small *N* or partially ordered
shell	✔		?	?	N	tight code, subquadratic
merge		✔	N lg N	N lg N	N lg N	*N* log *N* guarantee, stable
quick	✔		$N^2 / 2$	2 N ln N	N lg N	*N* log *N* probabilistic guarantee fastest in practice
3-way quick	✔		$N^2 / 2$	2 N ln N	N	improves quicksort in presence of duplicate keys
???	✔	✔	N lg N	N lg N	N	holy sorting grail