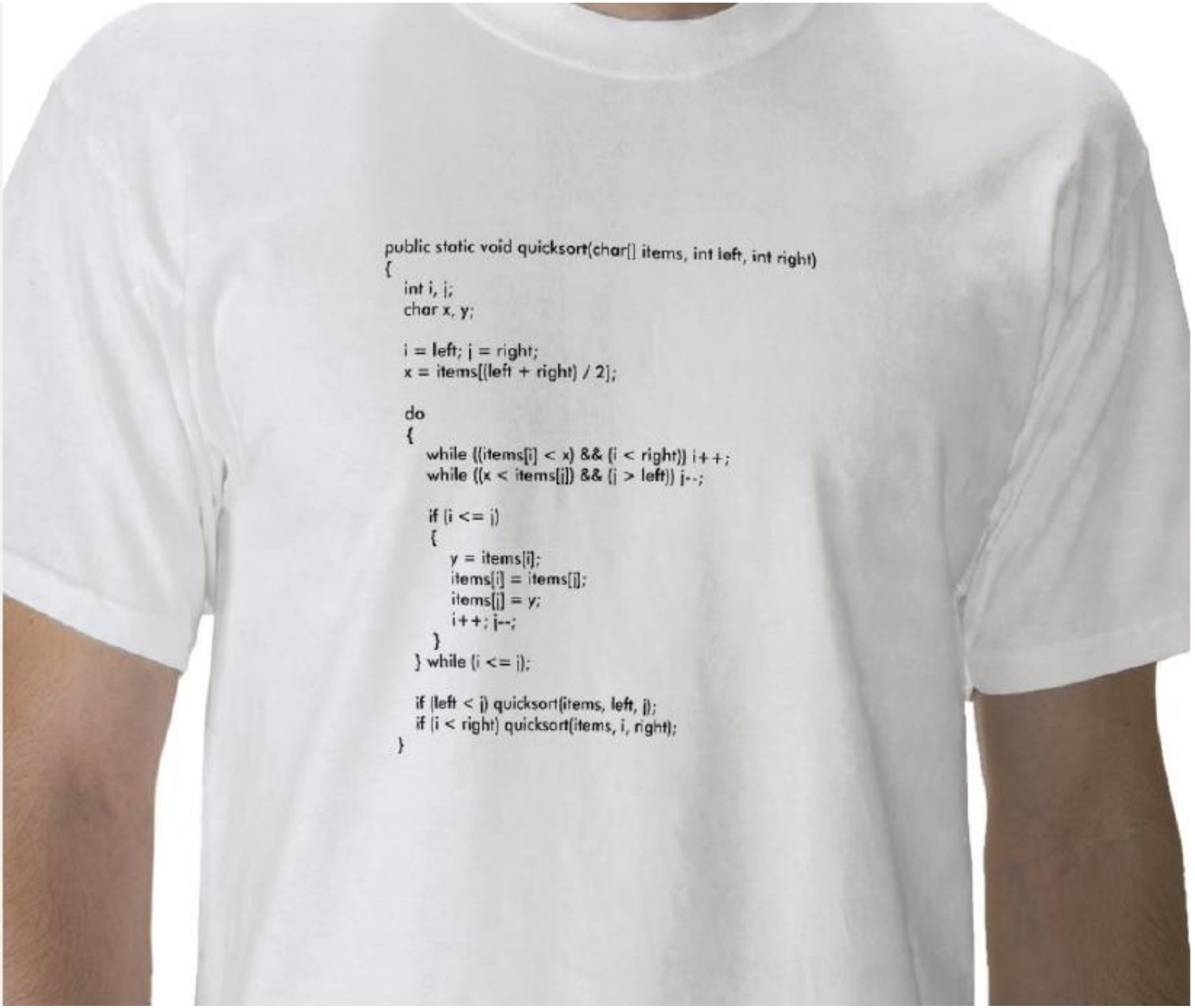
Быстрая сортировка
(Quicksort)

Два классических алгоритма сортировки

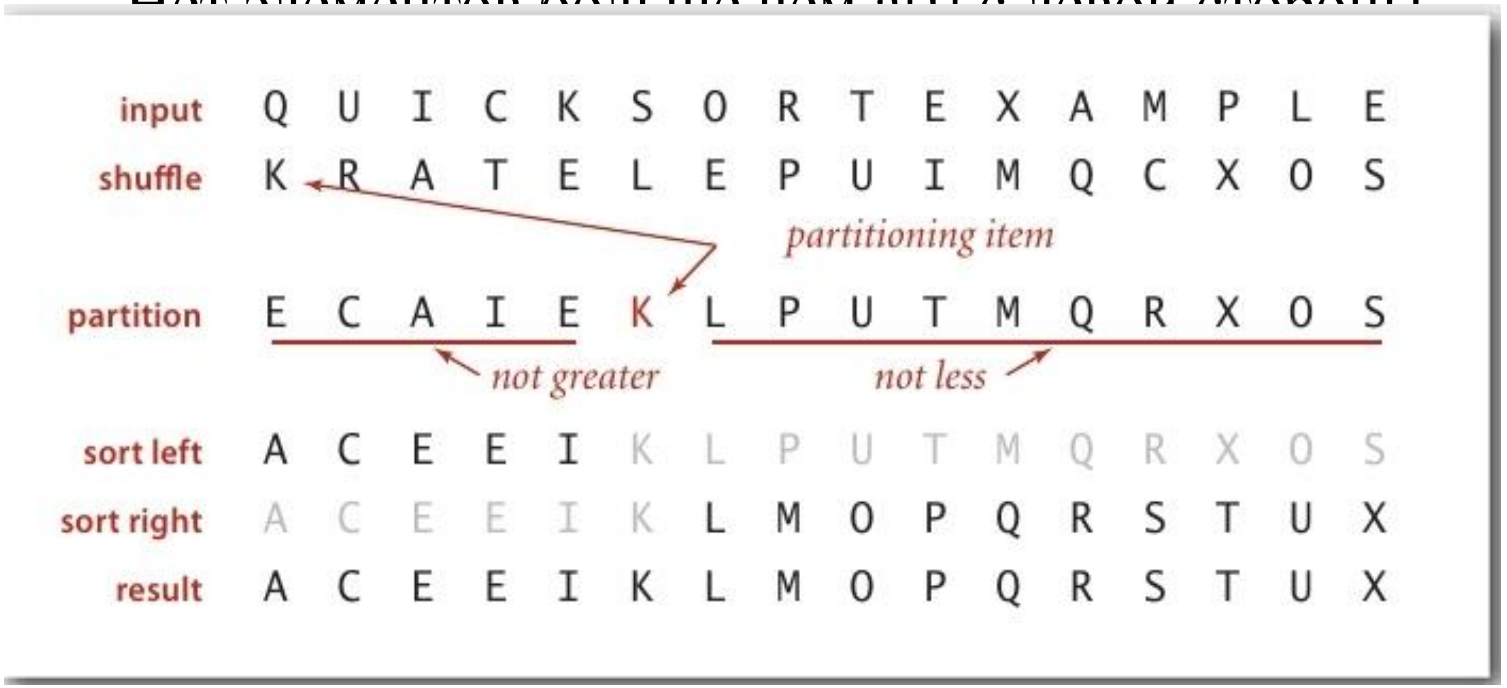
- Критические компоненты в мировой вычислительной инфраструктуре
  - Понимание научных основ этих алгоритмов даст нам возможность разрабатывать прикладные системы для решения реальных задач
  - Быстрая сортировка входит в десятку самых полезных алгоритмов, разработанных в 20-м веке


public static void quicksort(char[] items, int left, int right)
{
int i, j;
char x, y;
i = left; j = right;
x = items[(left + right) / 2];
do
{
while ((items[i] < x) && (i < right)) i++;
while ((x < items[j]) && (j > left)) j--;
if (i <= j)
{
y = items[i];
items[i] = items[j];
items[j] = y;
i++; j--;
}
} while (i <= j);
if (left < j) quicksort(items, left, j);
if (i < right) quicksort(items, i, right);
}

Быстрая сортировка

- Основной план
  - Перемешать элементы случайным об
  - Разбиение для элемента j
    - a[j] оставить на месте
    - Нет элементов меньше чем a[j] с правой стороны
    - Нет элементов больше чем a[j] с левой стороны
  -

Sir Charles Antony Richard Hoare
1980 Turing Award



Быстрая сортировка

- Повторять до тех пор, пока i и j не пересекутся
  - Проверять i-ые элементы до тех пор пока a[i] < a[lo]
  - Проверять j-ые элементы до тех пор пока a[j] > a[lo]
  - Поменять местами a[i] и a[j]
- Видео 1

Быстрая сортировка: реализация разбиения на Java

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))
         if (i == hi) break;

      while (less(a[lo], a[--j]))
         if (j == lo) break;

      if (i >= j) break;
      exch(a, i, j);
   }

   exch(a, lo, j);
   return j;
}
```

find item on left to swap

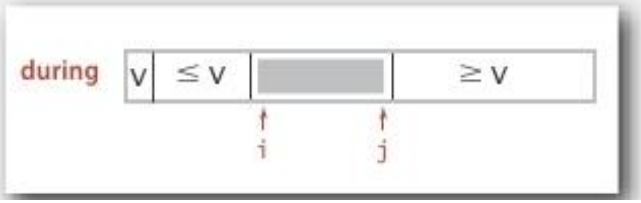
find item on right to swap

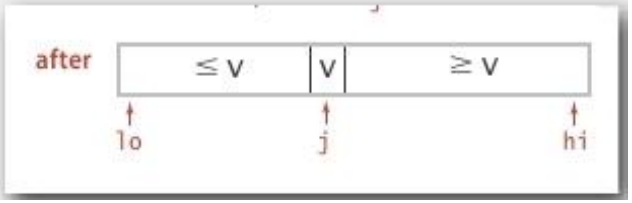
check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place





Быстрая сортировка: реализация на Java

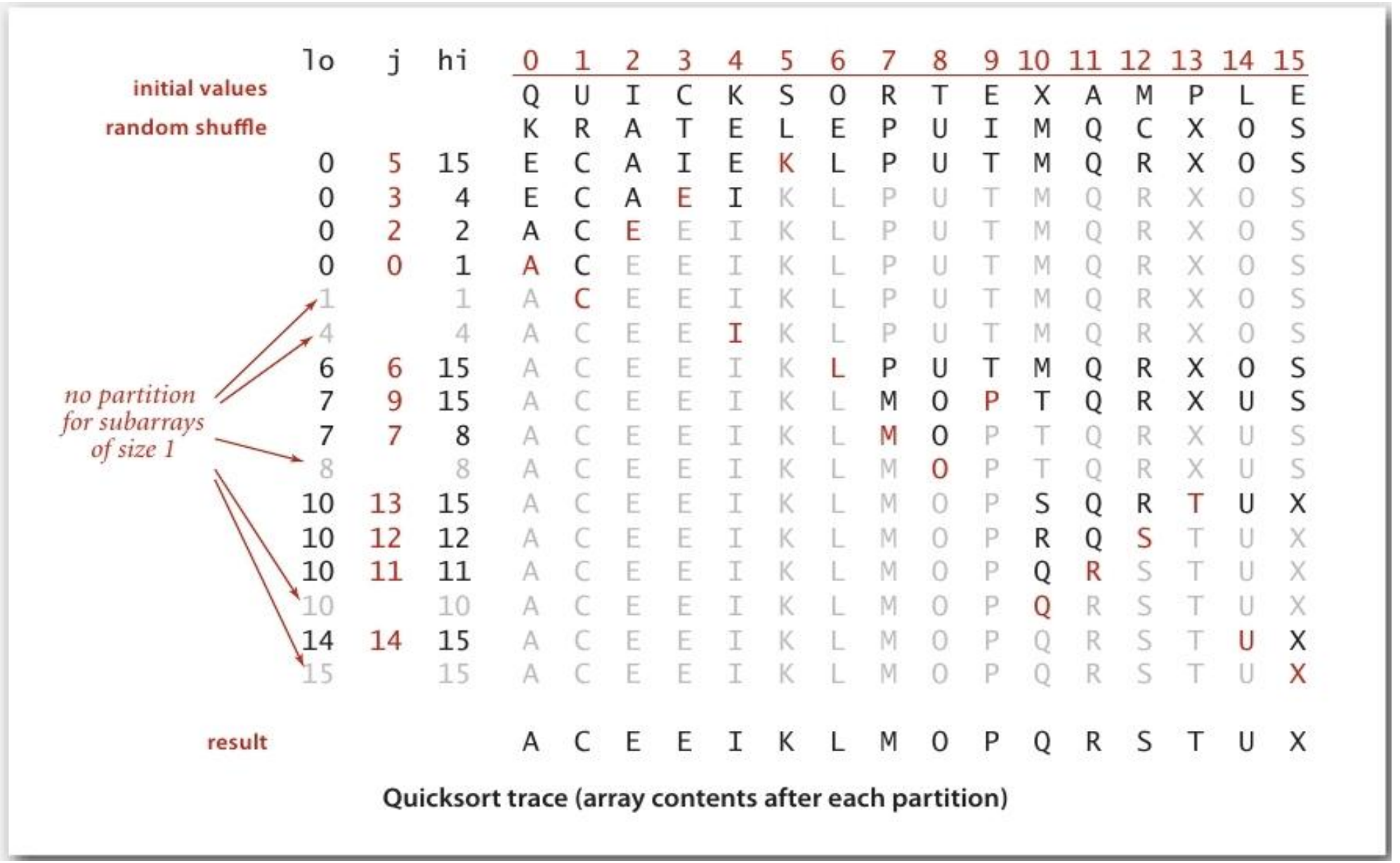
```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

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

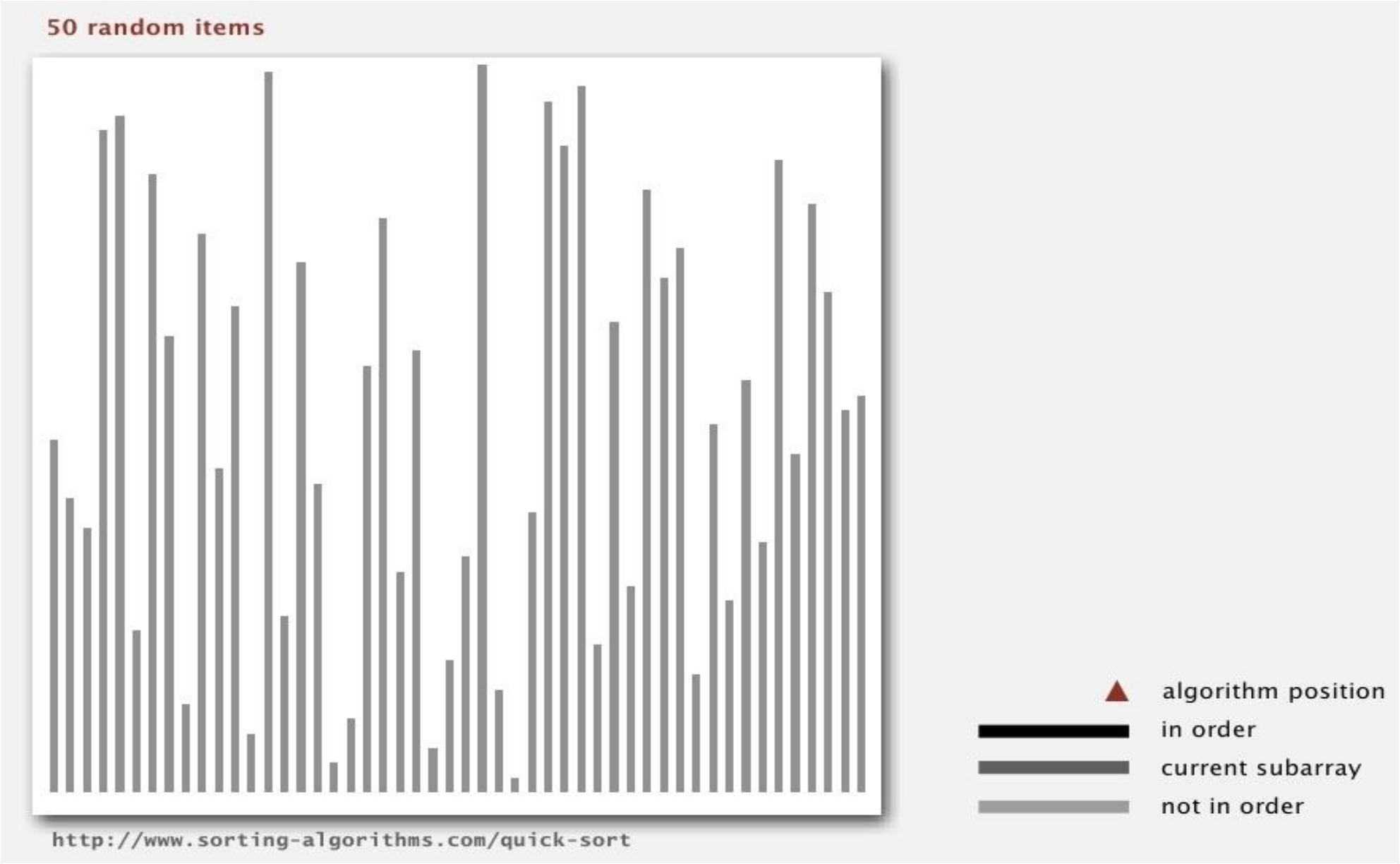
shuffle needed for performance guarantee (stay tuned)

Быстрая сортировка



Quicksort trace (array contents after each partition)

Быстрая сортировка



Быстрая сортировка: реализация

- Не требует дополнительной памяти
- Выход из циклов. Обращайте особое внимание на условия выхода из циклов
- Ограничения. Проверка j == lo излишняя, но i == hi нет
- Рандомизация. Перетасовка нужна чтобы обеспечить гарантии производительности
- Равные ключи. Если присутствуют дубликаты, то можно использовать другой вариант алгоритма

Быстрая сортировка: эмпирический анализ

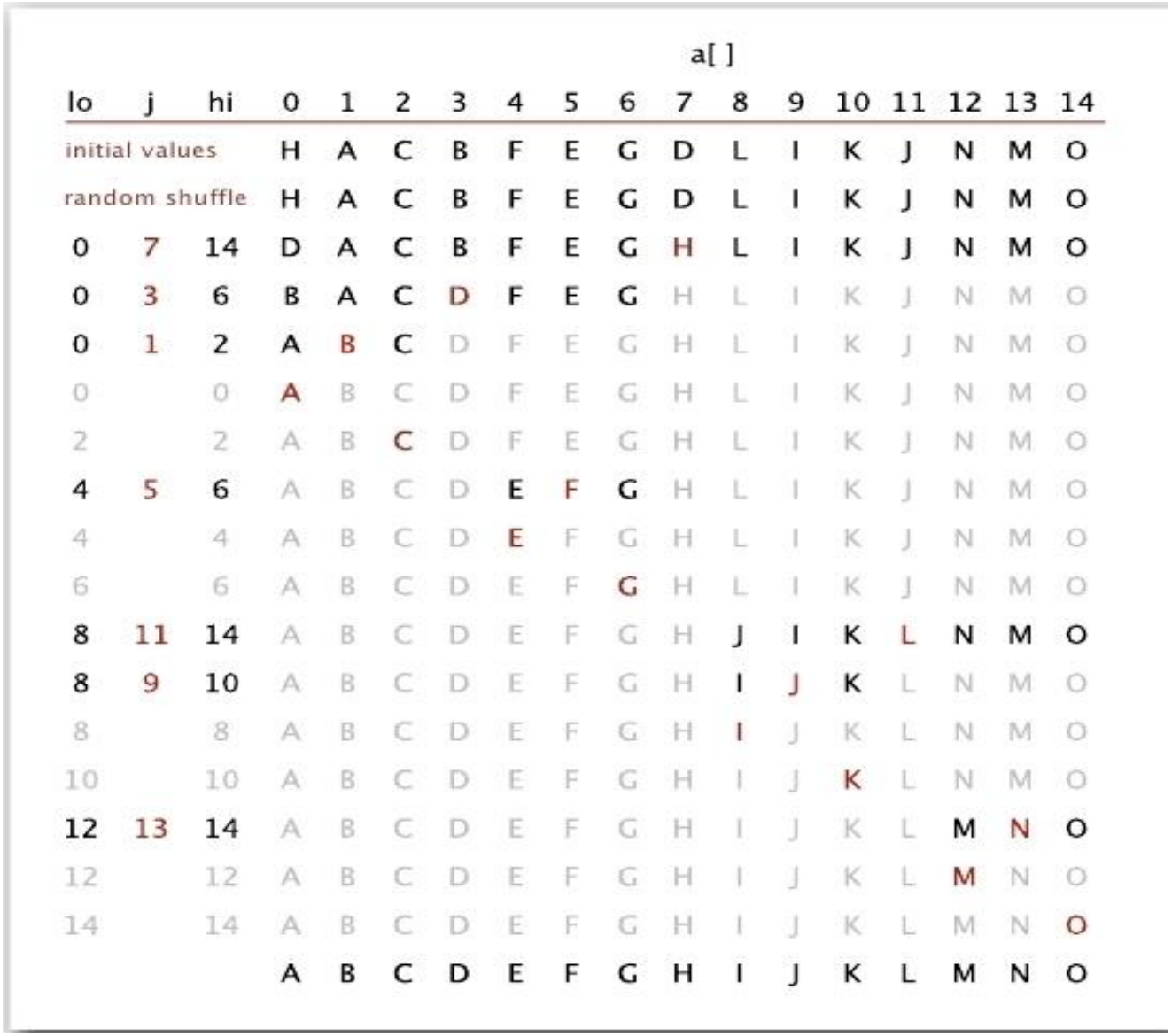
- Оценка времени выполнения:
  - На ПК 10^8 сравнений/секунду
  - На суперкомпьютере 10^{12} сравнений/секунду

	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

- Вывод 1. Хорошие алгоритмы лучше, чем суперкомпьютер

Быстрая сортировка: лучший случай

- Лучший случай. Количество сравнений ~ $N \log_2 N$



			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

Быстрая сортировка: худший случай

- Худший случай. Количество сравнений ~ ½ N^2

			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

Быстрая сортировка: средний случай

- Repeatedly apply above equation:

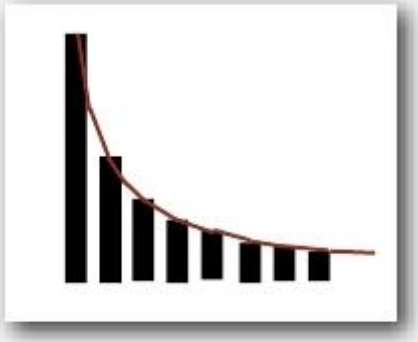
$$\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\end{aligned}$$

previous equation

substitute previous equation

- Approximate sum by an integral:

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



- Finally, the desired result:

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

Быстрая сортировка: свойства

- Худший случай. Квадратичное количество сравнений
  - $N + (N-1) + (N-2) + \ldots + 1 \sim \frac{1}{2} N^2$
- Средний случай. Количество сравнений ~ 1,39 $N \log_2 N$
  - На 39% сравнений больше, чем у сортировки слиянием
  - Но, на практике, быстрее сортировки слиянием, потому что меньше перемещаются данные
- Перетасовка
  - Гарантирует отсутствия худшего случая
- Предупреждение. Часть реализаций быстрой

Быстрая сортировка: свойства

- Утверждение. Быстрая сортировка не требует дополнительной памяти
- Доказательство
  - Разделение требует дополнительную память р
  - Р логарифм
- Быс ва

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

Быстрая сортировка: реализация

- Используйте сортировку вставками для маленьких подмассивов
  - Быстрая сортировка очень дорогая для маленьких подмассивов
  - Подмассивы для сортировки вставками ~ 10

```
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);
}
```

Быстрая сортировка: реализация

- Разбиение по медиане
  - Поиск медианы из небольшой выборки элементов
  - Медиана из 3-х случайных элементов

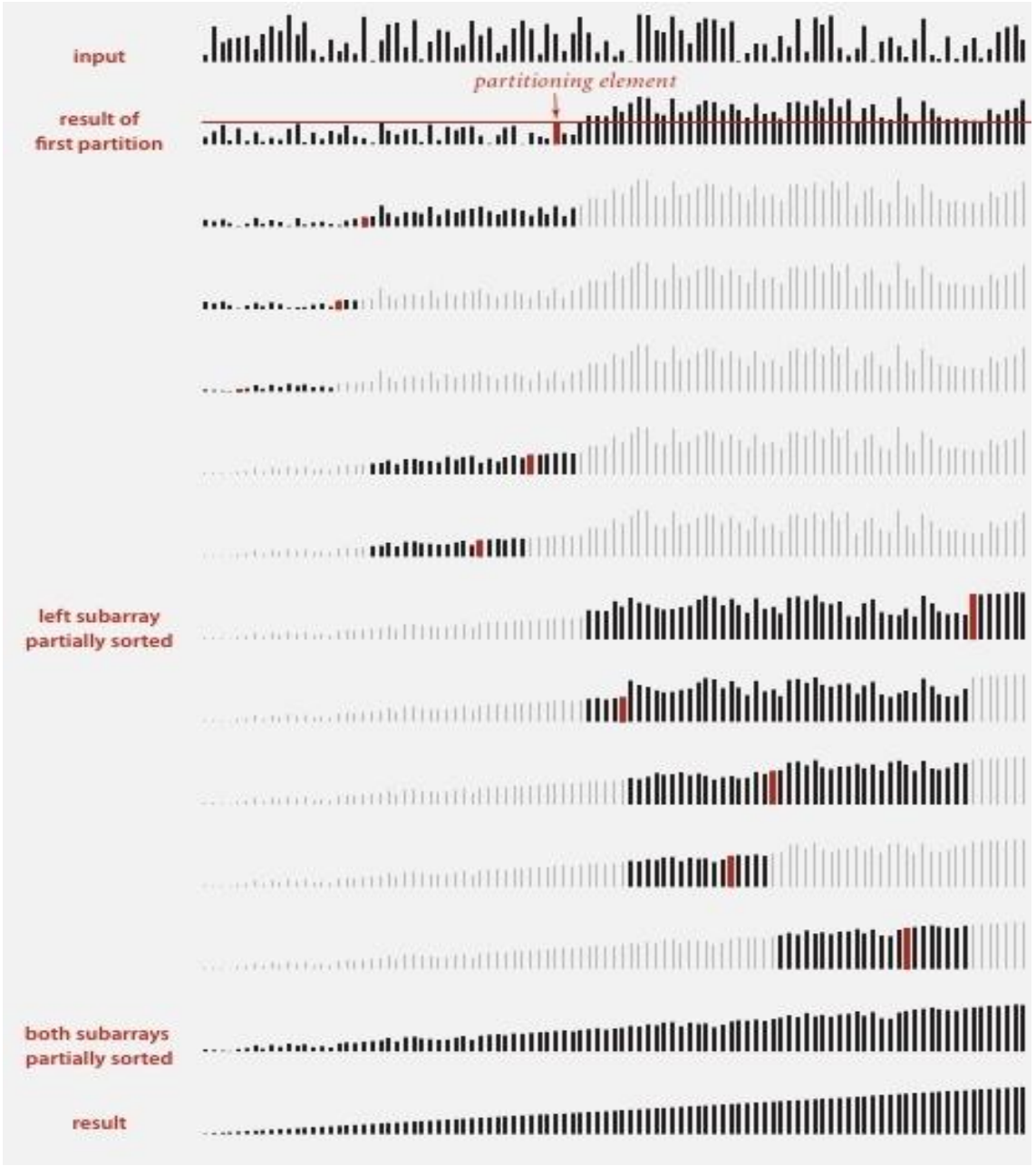
~ 12/7 N ln N compares (slightly fewer)
~ 12/35 N ln N exchanges (slightly more)

```
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);
}
```

Быстрая сортировка с сортировкой вставками и медианой из 3-х



Выбор
(Selection)

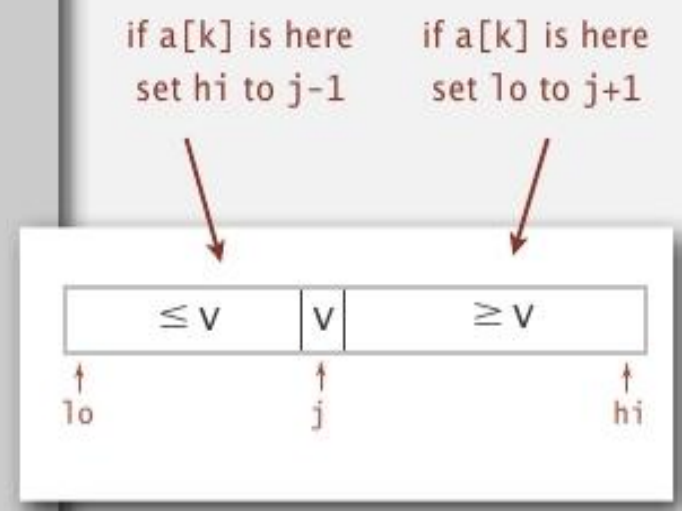
Выбор

- Цель. В массиве размером N, найти k-й наименьший элемент
- Пример. Min (k = 0), max (k = N-1), median (k = N / 2)
- Применение
  - Порядковая статистика
  - Поиск наибольшего элемента
- Руководствуйтесь теорией
  - NlogN верхняя граница
  - N верхняя граница для k = 1, 2, 3.
  - N нижняя граница

Быстрый выбор (Quick-select)

- Разделение массива
  - a[j] остается на месте
  - Слева нет элемента больше j
  - Справа нет элемента меньше j

```
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];
}
```

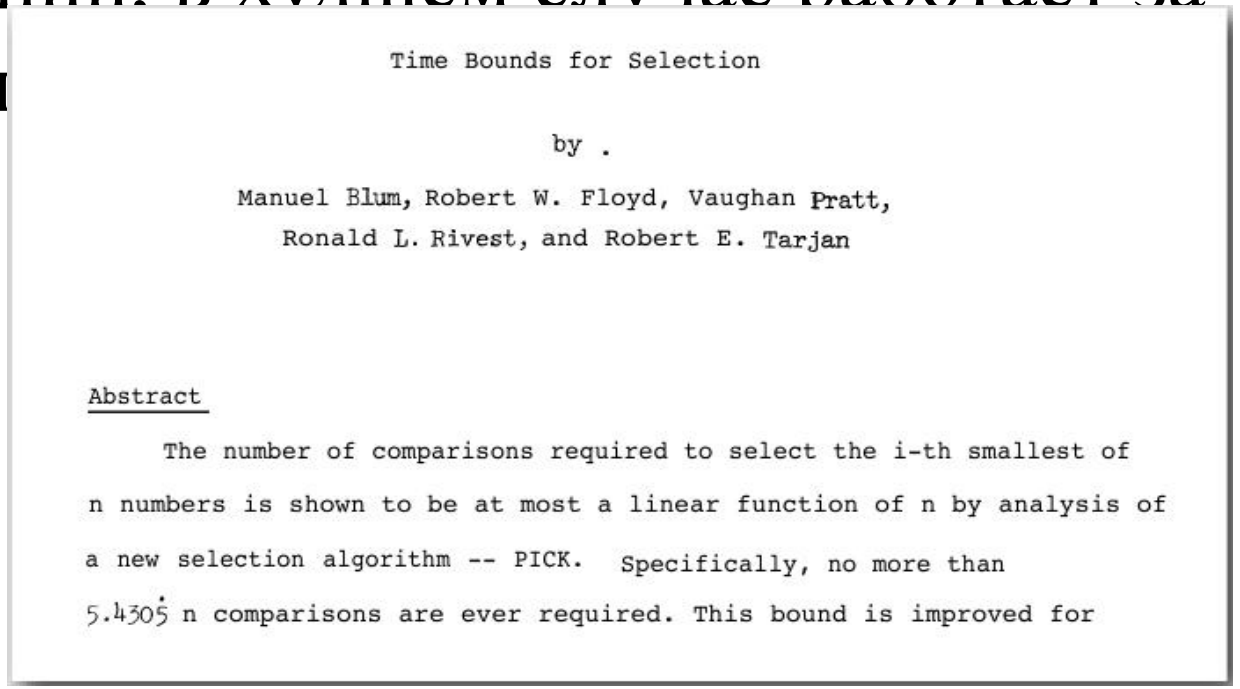


Быстрый выбор: математический анализ

- Утверждение. Quick-select в среднем работает за линейное время
- Доказательство
  - Каждый шаг делит массив пополам: $N + N/2 + N/4 + \ldots + 1 \sim 2N$ сравнений
  - Формула анализа похожа на Q-sort
  - $C_N = 2N + 2k \ln(N/k) + 2(N-k) \ln(N/(N-k))$
- Замечание. Q-select использует $\sim \frac{1}{2} N^2$ сравнений в худшем случае, но (как и для Q-sort) перемешивание дает вероятностную гарантию

Быстрый выбор

- Утверждение. Алгоритм выбора, основанный на сравнении, в худшем случае работает за линейн



Time Bounds for Selection

by .

Manuel Blum, Robert W. Floyd, Vaughan Pratt,
Ronald L. Rivest, and Robert E. Tarjan

Abstract

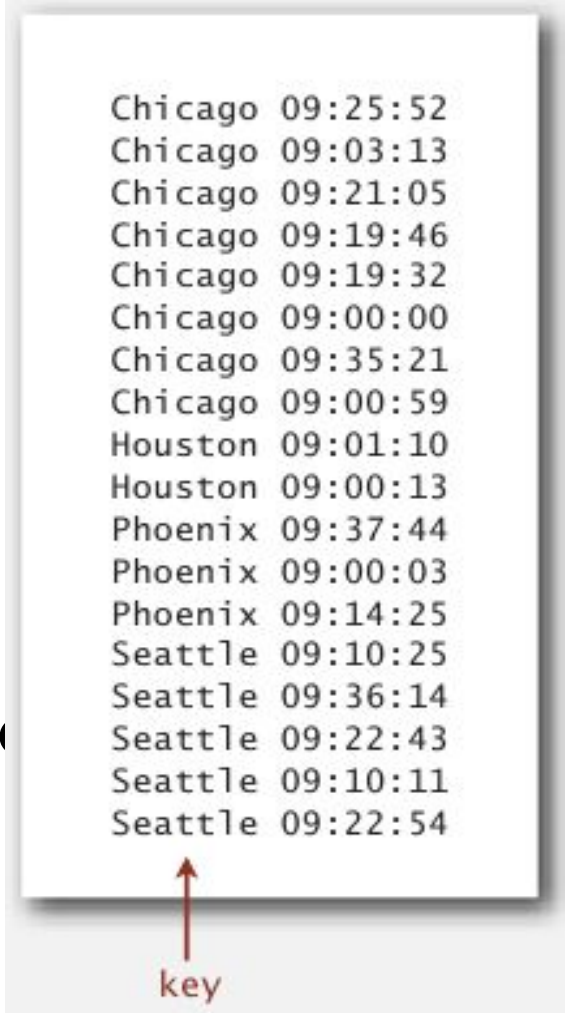
The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm -- PICK. Specifically, no more than 5.4305 n comparisons are ever required. This bound is improved for

- Замечание. Константа слишком велика => на практике не используется
- Руководствуйтесь теорией
  - Все еще требуется найти алгоритм, работающий в

Повторяющиеся ключи

Повторяющиеся ключи

- Часто приходится сортировать данные с повторяющимися ключами
  - По возрасту людей
  - Удалять повторяющиеся письма
  - По профессии или должности
- Обычно в таких случаях
  - Огромный массив данных
  - Небольшое количество значений клю



Быстрый выбор (Quick-select)

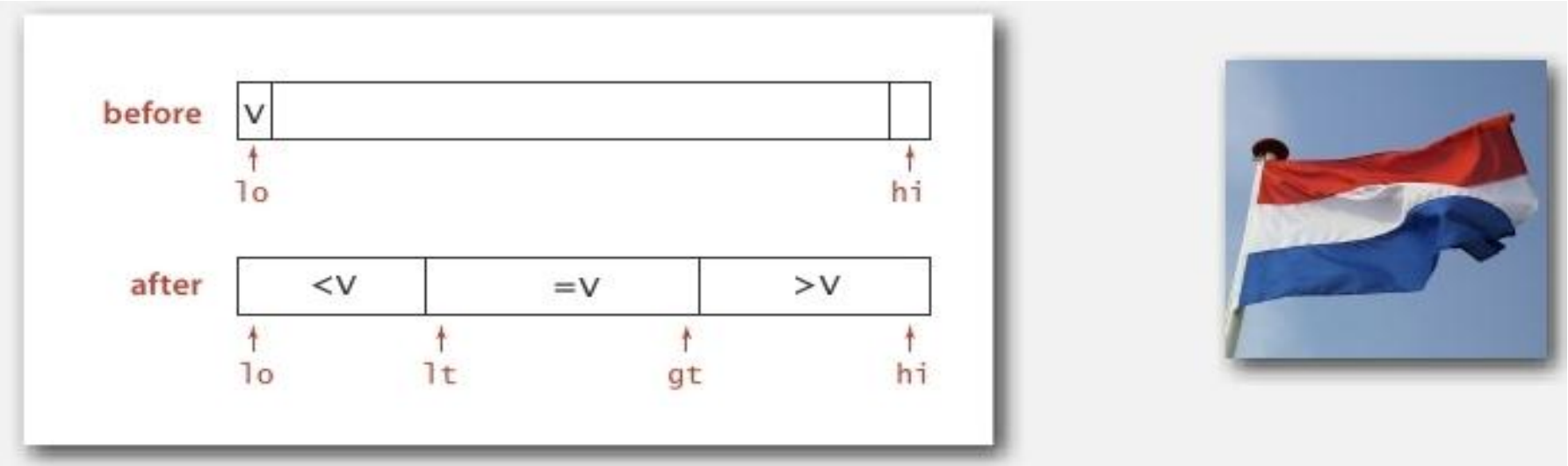
- Сортировка слиянием для массива с дубликатами. От ½ N $\log_2 N$ до N $\log_2 N$ сравнений
- Q-sort для массива с дубликатами
  - Алгоритм выполняется за квадратичное время, если не происходит остановки на элементе равном текущему
  - В 199 в qsort()

Проблема повторяющихся ключей

- Ошибка. Все элементы остаются на одной стороне
- Результат. ~ ½ N^2 сравнений, когда все ключи равны
- B A A B A B B B C C C A A A A A A A A A A A
- Рекомендация. Останавливать сканирование, если элемент равен центральному элементу
- Результат. ~ N $\log_2 N$ сравнений, когда все ключи равны
- B A A B A B C C B C B A A A A A A A A A A A
- Желаемый случай. Оставлять элементы равные центральному элементу на месте

Трехчастное разбиение

- Цель. Делим массив на 3 части
  - Элементы между lt и gt, равные центральному элементу v

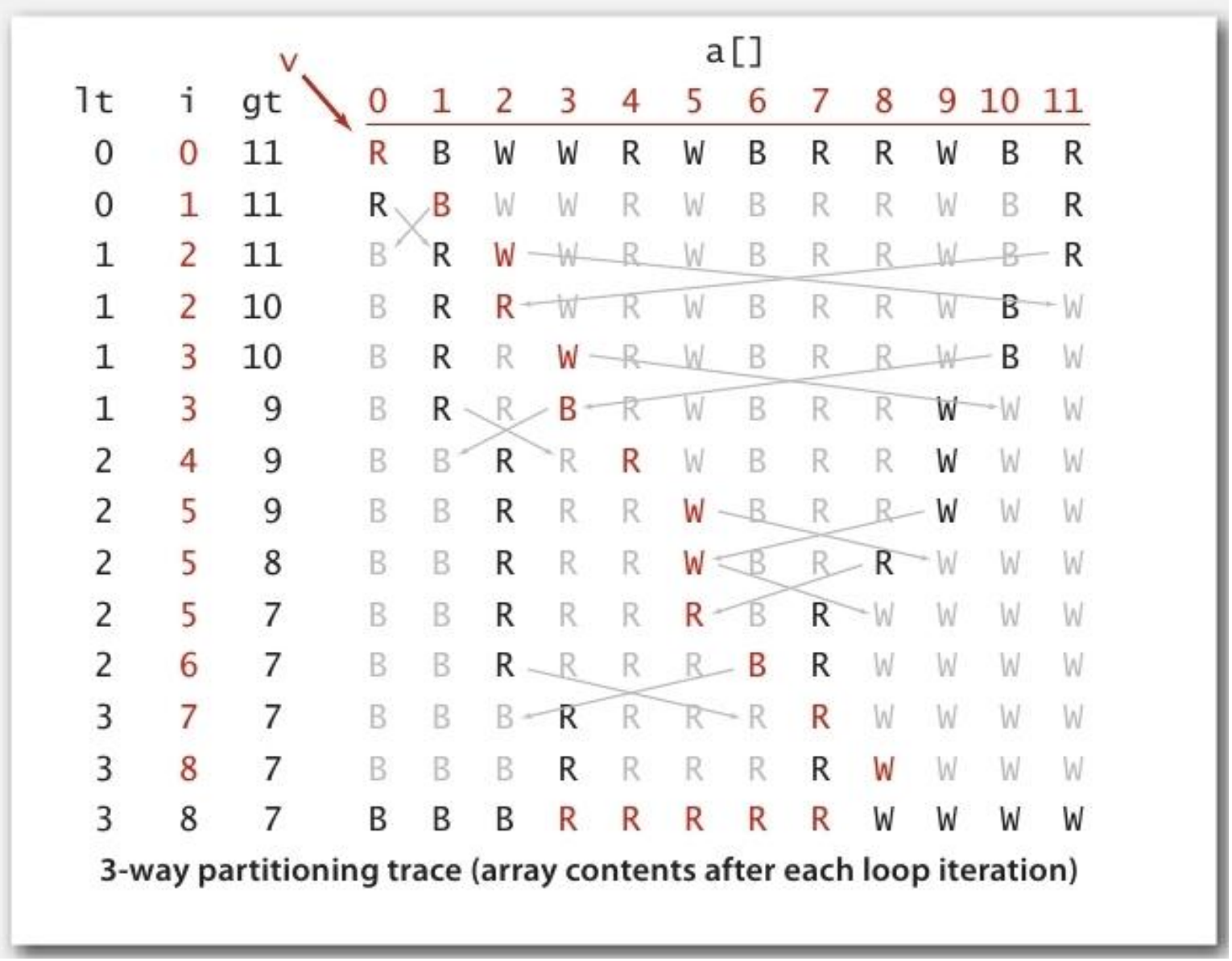


- Проблема нидерландского флага
  - Всеобщая мудрость до середины 90-х: ничего не делать
  - Ошибка найдена и исправлена в библиотеке C —

Трехчастное разбиение Дейкстры

- Берем v равное a[lo]
- Сканируем i слева на право
  - (a[i] < v): меняем местами a[lt] и a[i]; инкремент lt и i
  - (a[i] > v): меняем местами a[gt] и a[i]; декремент gt
  - (a[i] == v): инкремент i
- Видео 3

Трехчастное разбиение Дейкстры

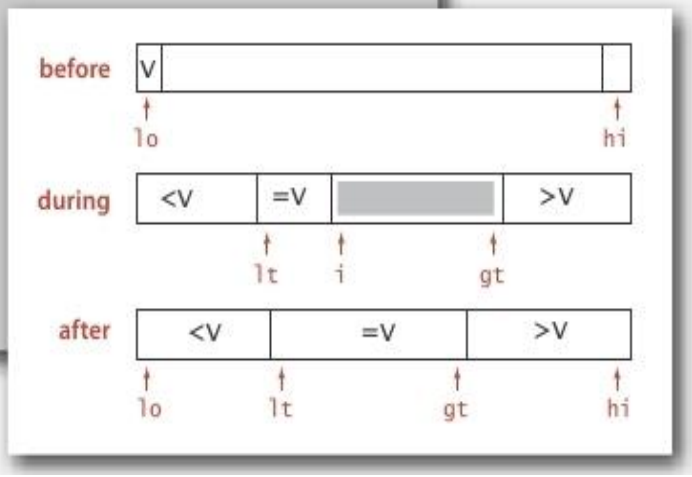


3-way partitioning trace (array contents after each loop iteration)

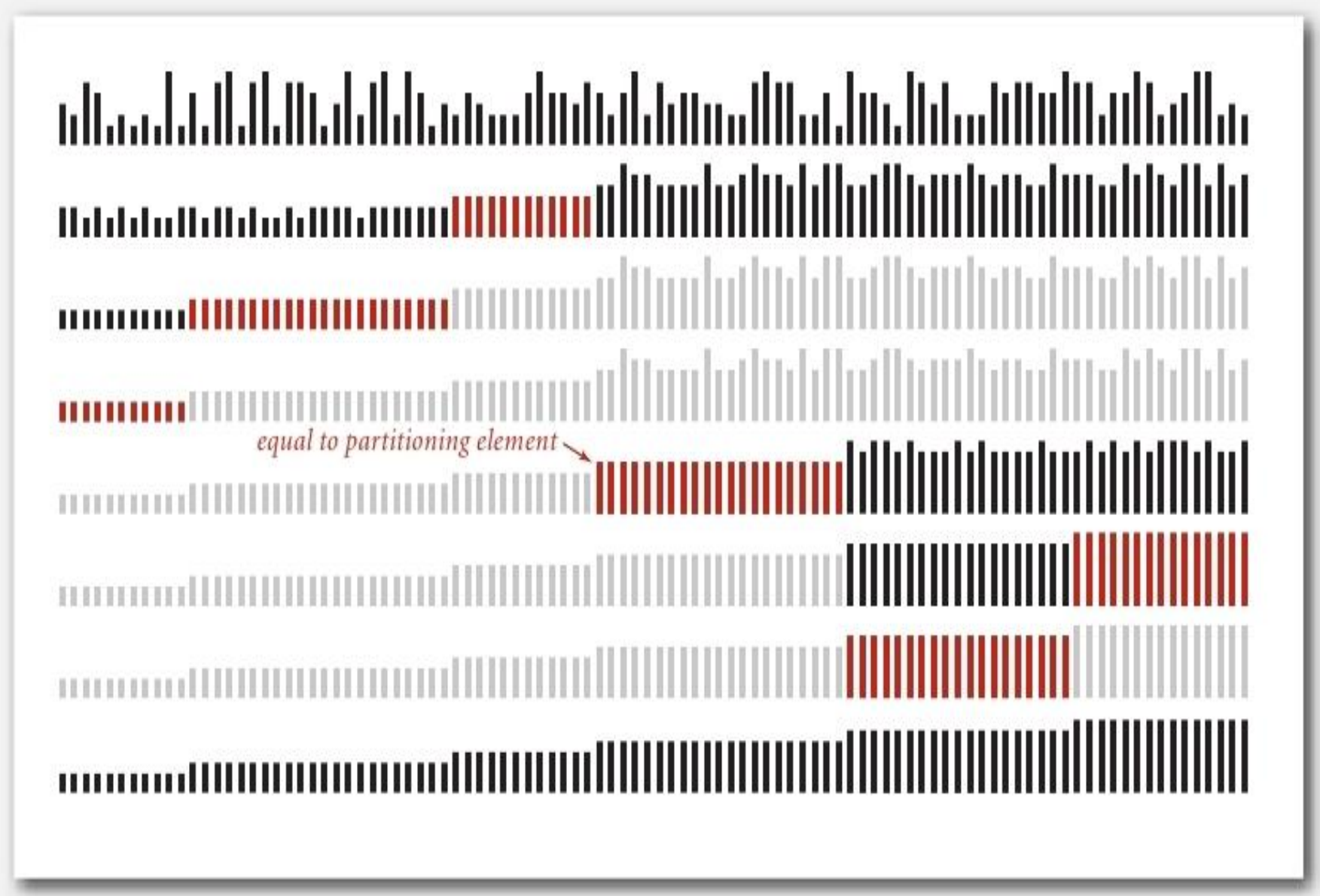
Трехчастное разбиение Дейкстры: реализация на Java

```
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);
}
```



Трехчастное разбиение Дейкстры: трассировка



Повторяющиеся ключи: нижняя граница

Sorting lower bound. If there are n distinct keys and the i^{th} one occurs x_i times, any compare-based sorting algorithm must use at least

$$\lg \left(\frac{N!}{x_1!\, x_2! \cdots x_n!} \right) \sim -\sum_{i=1}^{n} x_i \lg \frac{x_i}{N}$$

$N \lg N$ when all distinct;
linear when only a constant number of distinct keys

compares in the worst case.

Proposition. [Sedgewick-Bentley, 1997]

proportional to lower bound

Quicksort with 3-way partitioning is entropy-optimal.

Pf. [beyond scope of course]

Bottom line. Randomized quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.

Применение сортировок

Применение сортировок

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS feed in reverse chronological order.

obvious applications

- Find the median.
- Identify statistical outliers.
- Binary search in a database.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

- Data compression.
- Computer graphics.
- Computational biology.
- Load balancing on a parallel computer.

. . .

non-obvious applications

Сортировки в Java

- Arrays.sort()
  - Есть особые методы для примитивных типов
  - Методы для типов данных реализованных с помощью Comparable
  - Методы использующие Comparator
  - Исполь... ...итивных типов;

```
import java.util.Arrays;

public class StringSort
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readStrings());
      Arrays.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

War story (C qsort function)

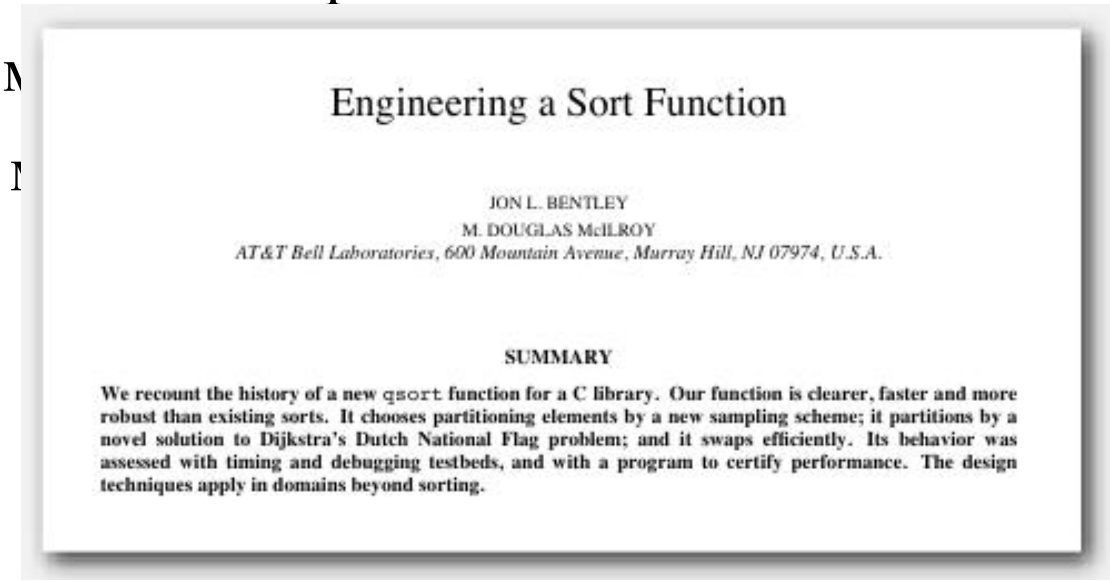
AT&T Bell Labs (1991). Allan Wilks and Rick Becker discovered that a `qsort()` call that should have taken seconds was taking minutes.

At the time, almost all `qsort()` implementations based on those in:

- Version 7 Unix (1979): quadratic time to sort organ-pipe arrays.
- BSD Unix (1983): quadratic time to sort random arrays of 0s and 1s.

Применение сортировок на практике

- Основной алгоритм — Q-sort
  - Сортировка вставками для маленьких подмассивов
  - Трехчастное разбиение
  - Разбиение
    - Маленький массив: средний элемент
    - Средний м
    - Большой м



Engineering a Sort Function

JON L. BENTLEY
M. DOUGLAS McILROY
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

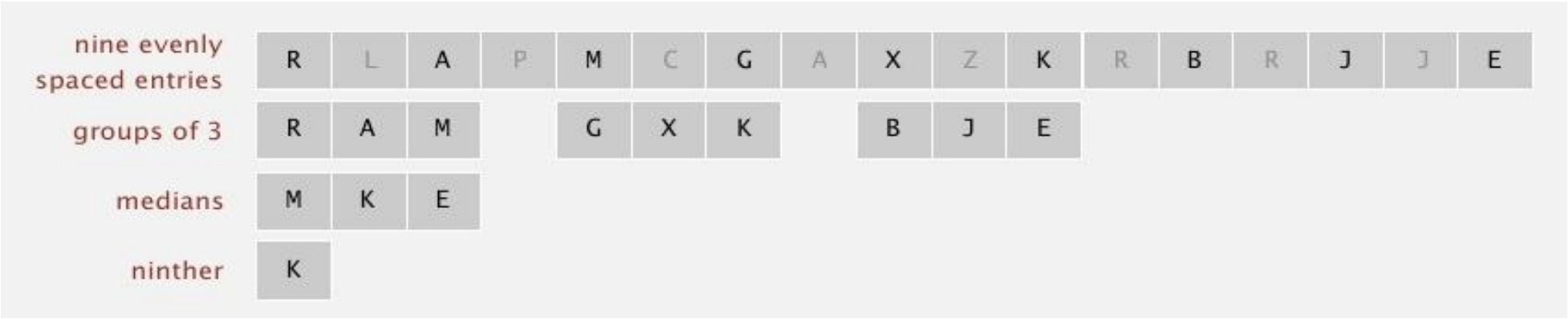
SUMMARY

We recount the history of a new `qsort` function for a C library. Our function is clearer, faster and more robust than existing sorts. It chooses partitioning elements by a new sampling scheme; it partitions by a novel solution to Dijkstra's Dutch National Flag problem; and it swaps efficiently. Its behavior was assessed with timing and debugging testbeds, and with a program to certify performance. The design techniques apply in domains beyond sorting.

- Сейчас широко используются в C, C++, Java ...

Девятки Тьюки

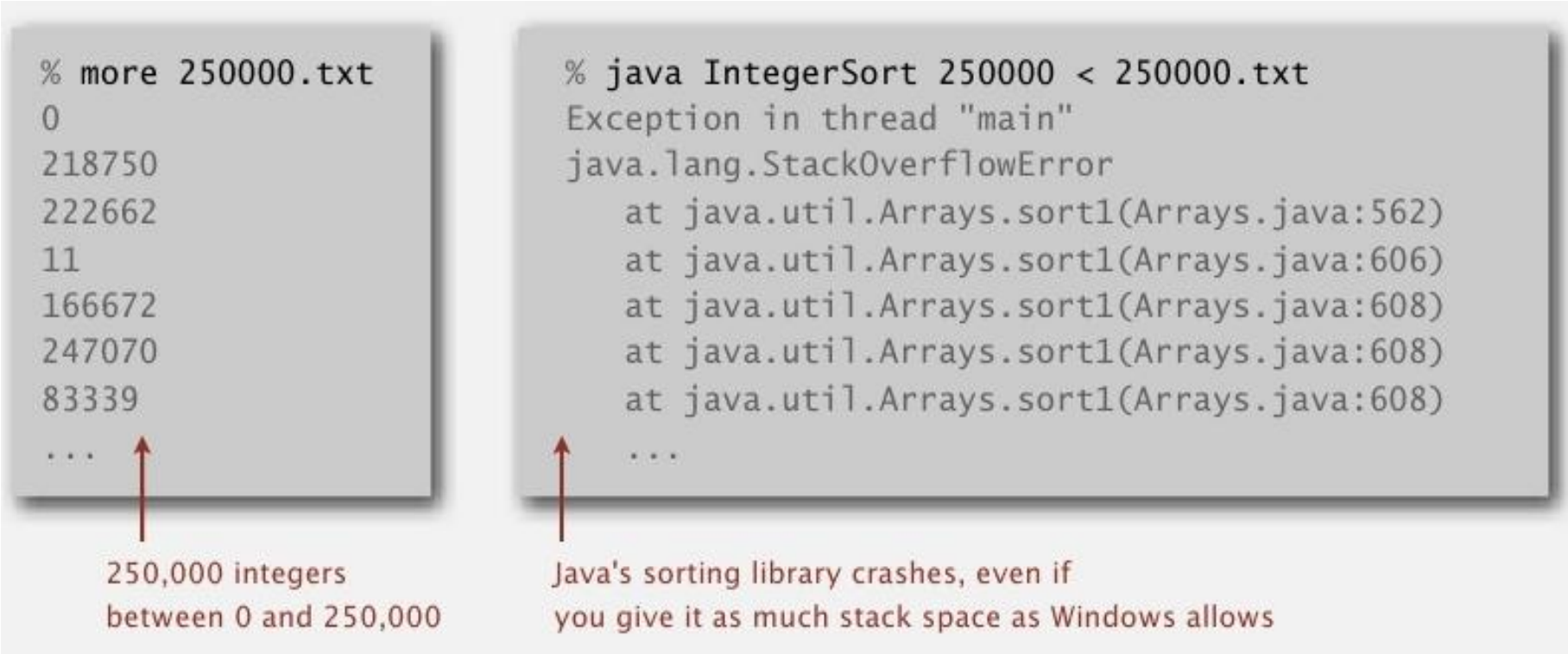
- Медиана из трех медиан из трех
  - Аппроксимация медианы из 9-ти
  - Использует не более 12 сравнений



- Лучше разбивает массив, чем случайный выбор центрального элемента: малая стоимость

Переполнение стека в Java

- Переполнение стека рекурсии в Java рушит программу
- Выполнение программы за квадратичное время



System sort: Which algorithm to use?

Many sorting algorithms to choose from:

Internal sorts.

- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splaysort, Yaroslavskiy sort, psort, ...

External sorts. Poly-phase mergesort, cascade-merge, oscillating sort.

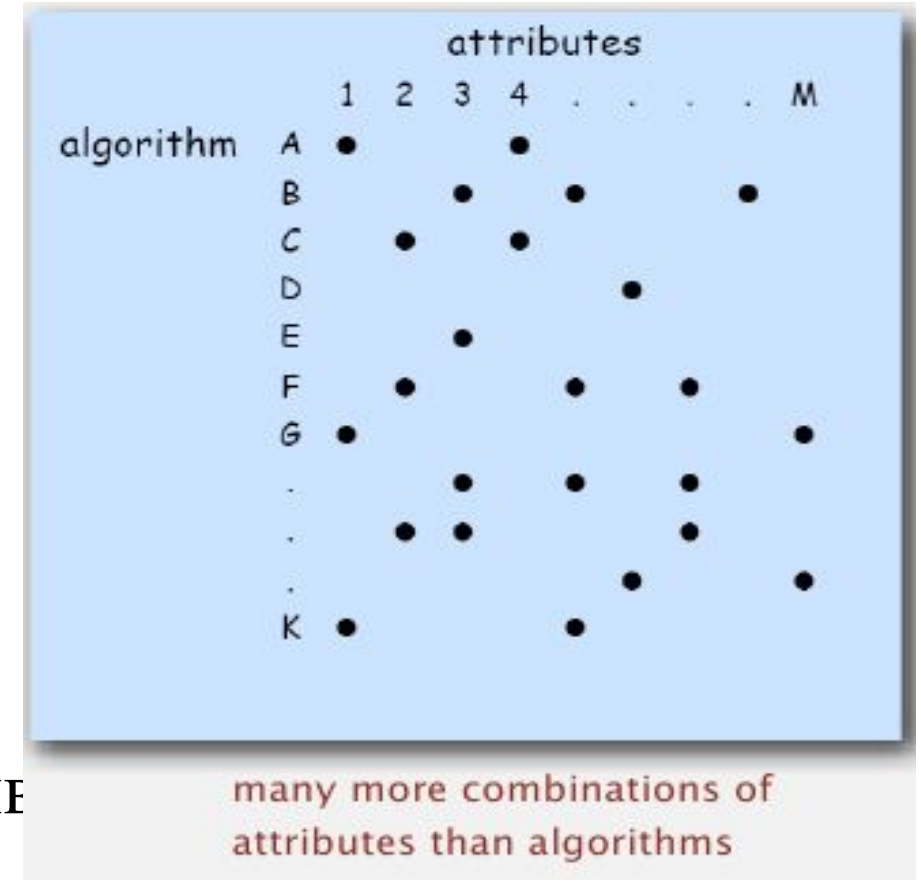
String/radix sorts. Distribution, MSD, LSD, 3-way string quicksort.

Parallel sorts.

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

Какую сортировку выбрать?

- Атрибуты
  - Стабильность
  - Параллелизм
  - Детерминированность
  - Дубликаты
  - Типы ключей
  - Связный список или массив
  - Количество элементов
  - Упорядоченность в массиве
  - Гарантии производительности
- Комбинирование сортировок



Сортировки. Выводы

	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