

# Types of analyses

---

**Best case.** Lower bound on cost.

- Determined by “easiest” input.
- Provides a goal for all inputs.

**Worst case.** Upper bound on cost.

- Determined by “most difficult” input.
- Provides a guarantee for all inputs.

**Average case.** Expected cost for random input.

- Need a model for “random” input.
- Provides a way to predict performance.

**Ex 1.** Array accesses for brute-force 3-SUM.

Best:  $\sim \frac{1}{2} N^3$

Average:  $\sim \frac{1}{2} N^3$

Worst:  $\sim \frac{1}{2} N^3$

**Ex 2.** Comparisons for binary search.

Best:  $\sim 1$

Average:  $\sim \lg N$

Worst:  $\sim \lg N$

# Types of analyses

---

Best case. Lower bound on cost.

Worst case. Upper bound on cost.

Average case. “Expected” cost.

Actual data might not match input model?

- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.

# Theory of algorithms

---

## Goals.

- Establish “difficulty” of a problem.
- Develop “optimal” algorithms.

## Approach.

- Suppress details in analysis: analyze “to within a constant factor.”
- Eliminate variability in input model: focus on the worst case.

**Upper bound.** Performance guarantee of algorithm for any input.

**Lower bound.** Proof that no algorithm can do better.

**Optimal algorithm.** Lower bound = upper bound (to within a constant factor).

# Commonly-used notations in the theory of algorithms

---

notation	provides	example	shorthand for	used to
<b>Big Theta</b>	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ ⋮	classify algorithms
<b>Big O</b>	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ ⋮	develop upper bounds
<b>Big Omega</b>	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ $N^5$ $N^3 + 22 N \log N + 3 N$ ⋮	develop lower bounds

# Theory of algorithms: example 1

---

## Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 1-SUM = “*Is there a 0 in the array?*”

## Upper bound.

A specific algorithm.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is  $O(N)$ .

## Lower bound.

Proof that no algorithm can do better.

- Ex. Have to examine all  $N$  entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is  $\Omega(N)$ .

## Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is  $\Theta(N)$ .

# Theory of algorithms: example 2

---

## Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM.

**Upper bound.** A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is  $O(N^3)$ .

# Theory of algorithms: example 2

---

## Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM.

## Upper bound. A specific algorithm.

- Ex. Improved algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is  $O(N^2 \log N)$ .

## Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all  $N$  entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is  $\Omega(N)$ .

## Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?

# Algorithm design approach

---

## Start.

- Develop an algorithm.
- Prove a lower bound.

## Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

## Golden Age of Algorithm Design.

- 1970s–.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

## Caveats.

- Overly pessimistic to focus on worst case?
- Need better than “to within a constant factor” to predict performance.

# Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10 N^2$	$10 N^2$ $10 N^2 + 22 N \log N$ $10 N^2 + 2 N + 37$	provide approximate model
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ $N^5$ $N^3 + 22 N \log N + 3 N$	develop lower bounds

Common mistake. Interpreting big-Oh as an approximate model.  
 This course. Focus on approximate models: use Tilde-notation

## Announcements

---

How did HW3 go?

HW4 is will be released later today. Written assignment on sorting. It will be due before class next Thursday. (Doing the homework will be good preparation for the midterm).

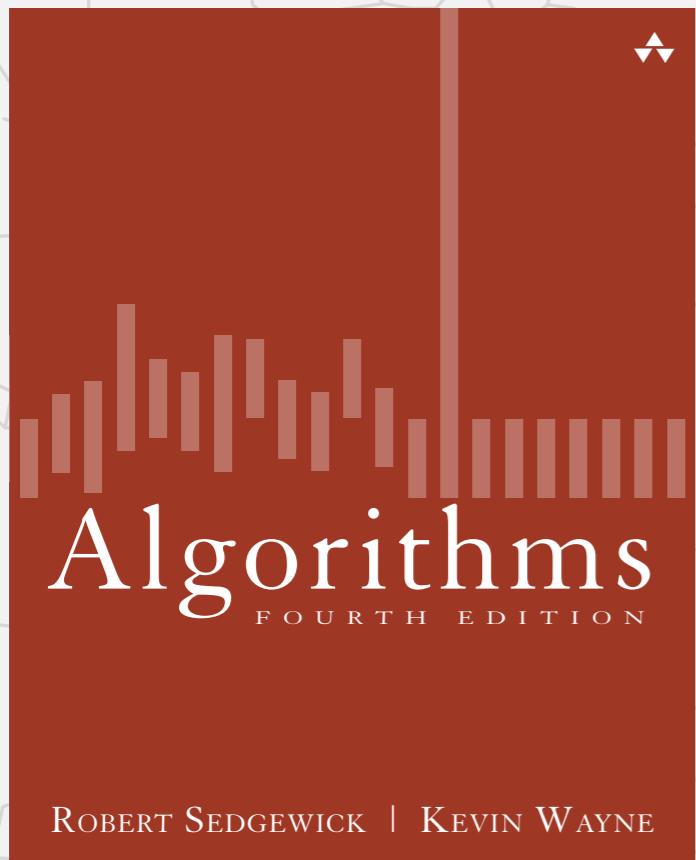
In-class midterm for next Thursday. Closed book. No devices.  
You should study all of the textbook chapters that have been assigned so far.

We will do our very best to grade the midterm before the drop deadline, so that you know where you stand in the class before you have to make that decision.

My office hours are today immediately after class in Levine 506. I will also be in my office tomorrow from noon-1 even with the university closing.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<http://algs4.cs.princeton.edu>

## 2.3 QUICKSORT

---

- ▶ *quicksort*
- ▶ *selection*
- ▶ *duplicate keys*
- ▶ *system sorts*

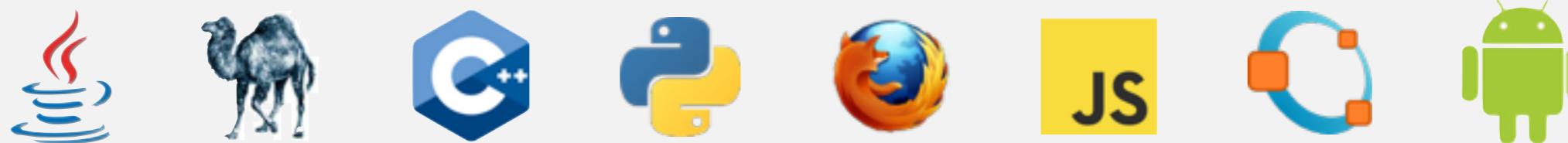
# Two classic sorting algorithms: mergesort and quicksort

---

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20<sup>th</sup> century in science and engineering.

Mergesort. [last lecture]

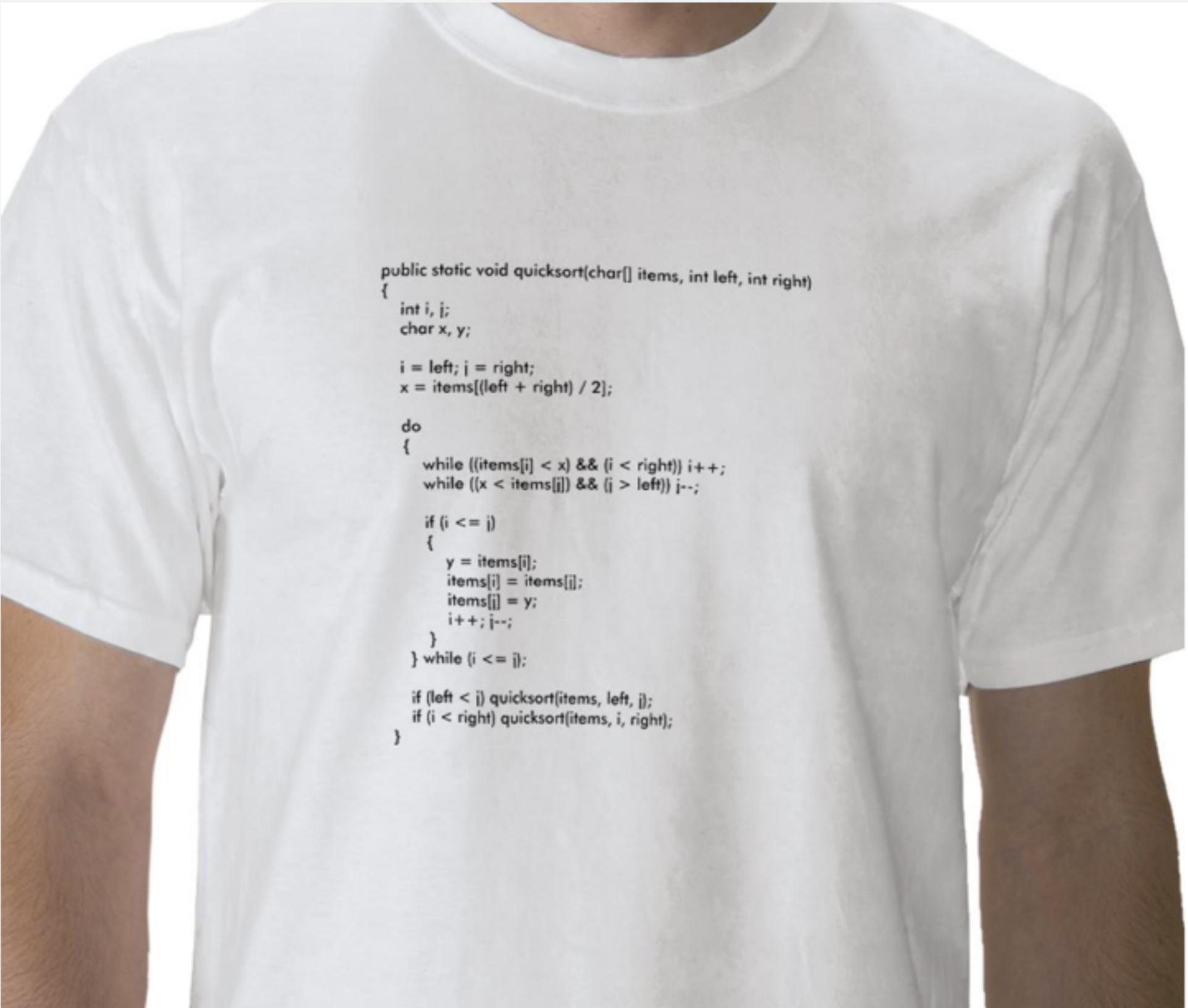


Quicksort. [this lecture]



# Quicksort t-shirt

---



```
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);
}
```



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.3 QUICKSORT

---

- ▶ ***quicksort***
- ▶ *selection*
- ▶ *duplicate keys*
- ▶ *system sorts*

# Quicksort overview

Step 1. Shuffle the array.

Step 2. Partition the array so that, for some j



- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Step 3. Sort each subarray recursively.

<b>input</b>	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
<b>shuffle</b>	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	<i>partitioning item</i>															
<b>partition</b>	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	<i>not greater</i>								<i>not less</i>							
<b>sort left</b>	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
<b>sort right</b>	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
<b>result</b>	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

# Quicksort overview

---

input

Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Quicksort overview

---

Step 1. Shuffle the array.

shuffle

Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Quicksort overview

---

Step 1. Shuffle the array.

shuffle

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Quicksort overview

---

**Step 2.** Partition the array so that, for some  $j$

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

**partition**



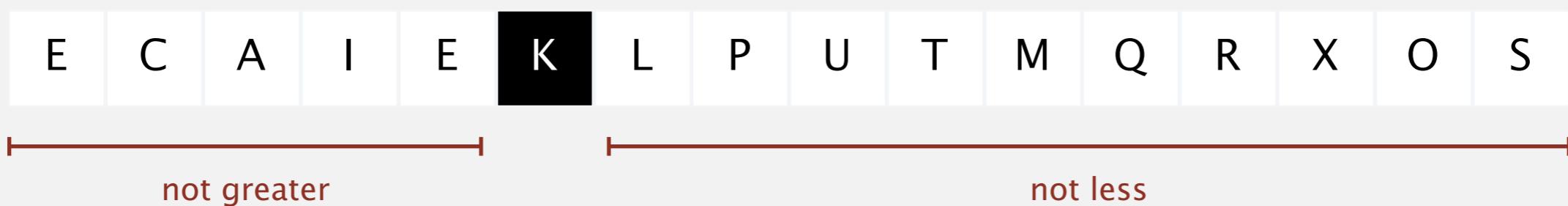
# Quicksort overview

---

**Step 2.** Partition the array so that, for some  $j$

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

**partition**



# Quicksort overview

---

Step 3. Sort each subarray recursively.

sort the left subarray

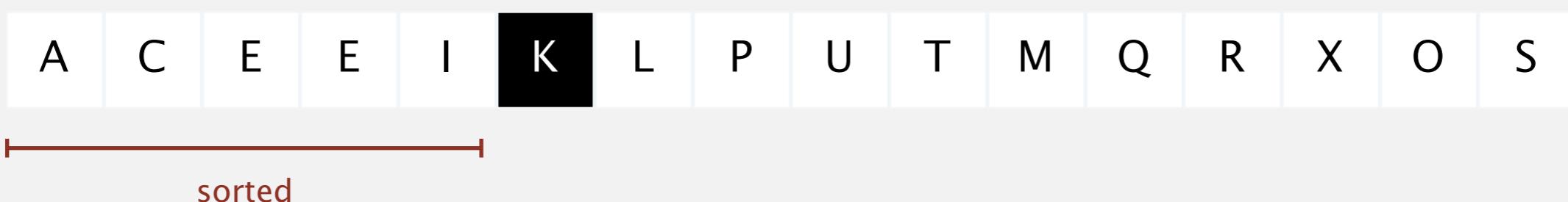


# Quicksort overview

---

Step 3. Sort each subarray recursively.

sort the left subarray

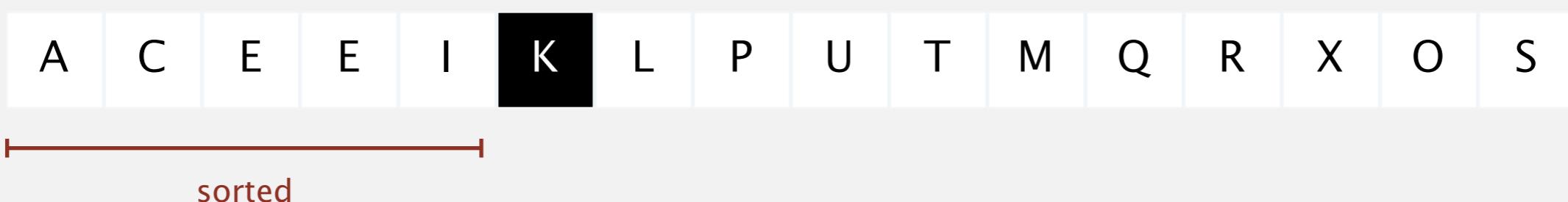


# Quicksort overview

---

Step 3. Sort each subarray recursively.

sort the right subarray

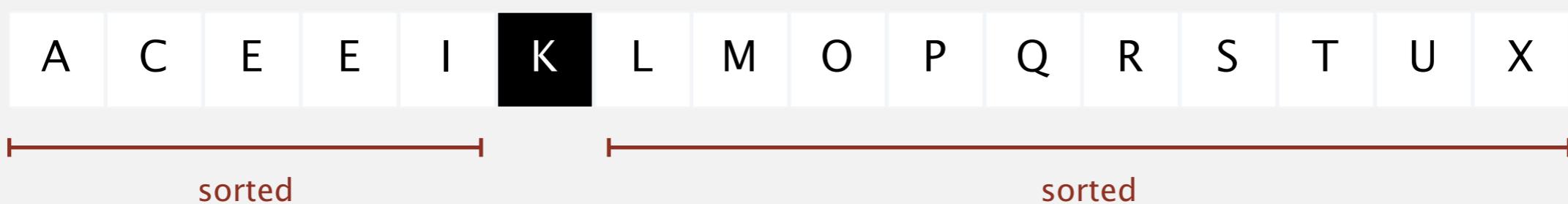


# Quicksort overview

---

Step 3. Sort each subarray recursively.

sort the right subarray



# Quicksort overview

---

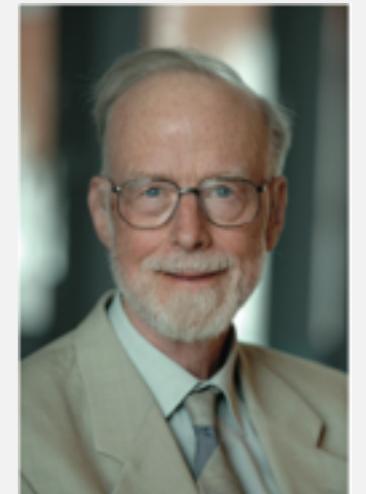
sorted array

A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Tony Hoare

---

- Invented quicksort to translate Russian into English.  
[ but couldn't explain his algorithm or implement it! ]
- Learned Algol 60 (and recursion).
- Implemented quicksort.



**Tony Hoare**  
**1980 Turing Award**

The image shows the cover of a technical document. At the top, the word "Algorithms" is written in a large, bold, serif font. Below it, the title "ALGORITHM 64" is in a smaller serif font. Underneath that, "QUICKSORT" and the author's name "C. A. R. HOARE" are listed. The text continues with the algorithm's implementation in Algol 60, followed by a detailed explanatory comment and the pseudocode for the quicksort procedure.

ALGORITHM 64  
QUICKSORT  
C. A. R. HOARE  
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

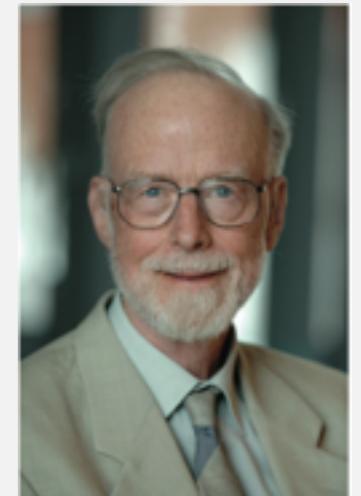
**procedure** quicksort (A,M,N); **value** M,N;  
    **array** A; **integer** M,N;  
**comment** Quicksort is a very fast and convenient method of sorting an array in the random-access store of a computer. The entire contents of the store may be sorted, since no extra space is required. The average number of comparisons made is  $2(M-N) \ln(N-M)$ , and the average number of exchanges is one sixth this amount. Suitable refinements of this method will be desirable for its implementation on any actual computer;  
**begin**     **integer** I,J;  
        **if** M < N **then begin** partition (A,M,N,I,J);  
            quicksort (A,M,J);  
            quicksort (A, I, N)  
        **end**  
    **end**     quicksort

**Communications of the ACM (July 1961)**

# Tony Hoare

---

- Invented quicksort to translate Russian into English.  
[ but couldn't explain his algorithm or implement it! ]
- Learned Algol 60 (and recursion).
- Implemented quicksort.



*“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”*

**Tony Hoare**  
**1980 Turing Award**

*“I call it my billion-dollar mistake. It was the invention of the null reference in 1965... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”*

# Bob Sedgewick

- Refined and popularized quicksort.
- Analyzed many versions of quicksort.



Bob Sedgewick

Programming Techniques      S. L. Graham, R. L. Rivest  
Editors

## Implementing Quicksort Programs

Robert Sedgewick  
Brown University

This paper is a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quicksort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quicksort's wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quicksort, analysis of algorithms, code optimization, sorting

CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5

Acta Informatica 7, 327—355 (1977)  
© by Springer-Verlag 1977

### The Analysis of Quicksort Programs\*

Robert Sedgewick

Received January 19, 1976

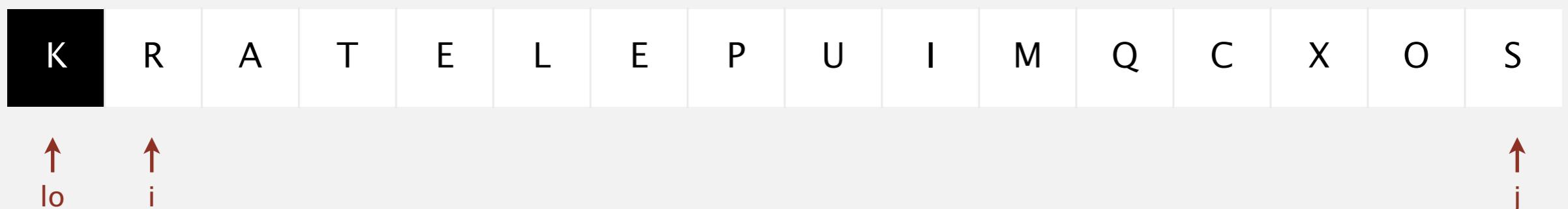
*Summary.* The Quicksort sorting algorithm and its best variants are presented and analyzed. Results are derived which make it possible to obtain exact formulas describing the total expected running time of particular implementations on real computers of Quicksort and an improvement called the median-of-three modification. Detailed analysis of the effect of an implementation technique called loop unwrapping is presented. The paper is intended not only to present results of direct practical utility, but also to illustrate the intriguing mathematics which arises in the complete analysis of this important algorithm.

# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

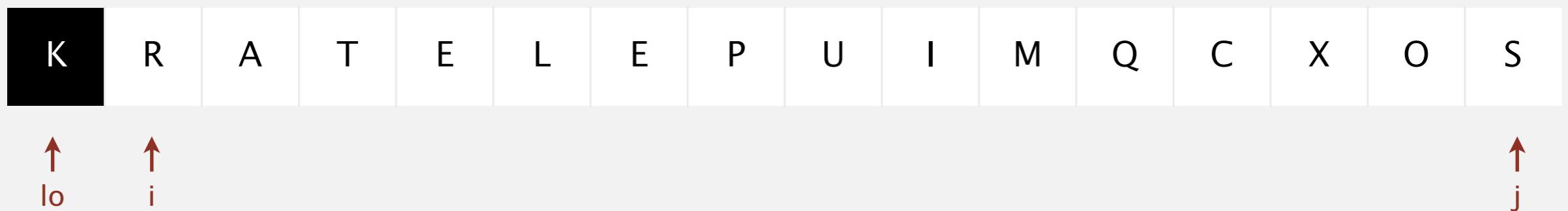


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



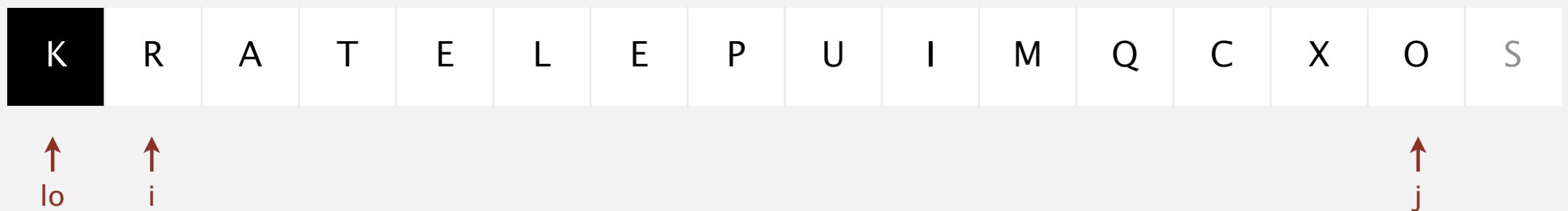
stop i scan because  $a[i] \geq a[lo]$

# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

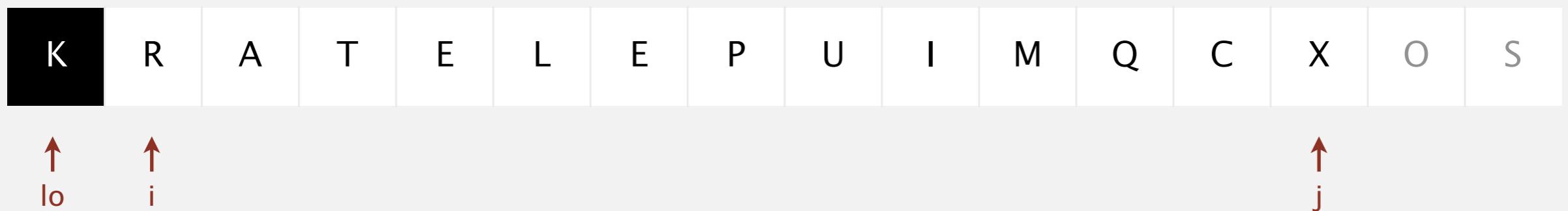


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

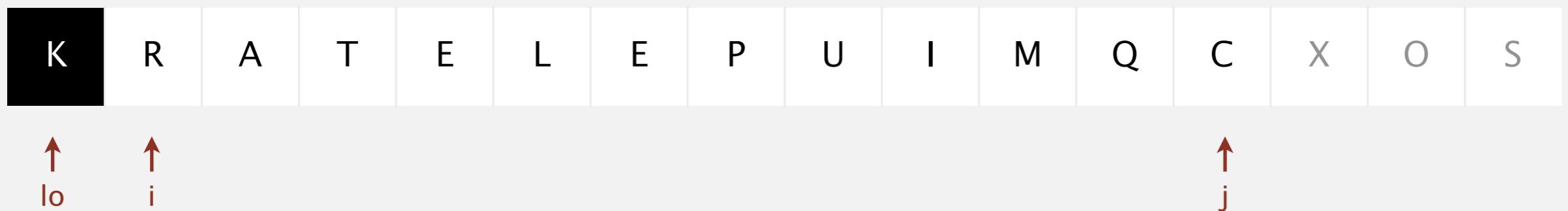


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



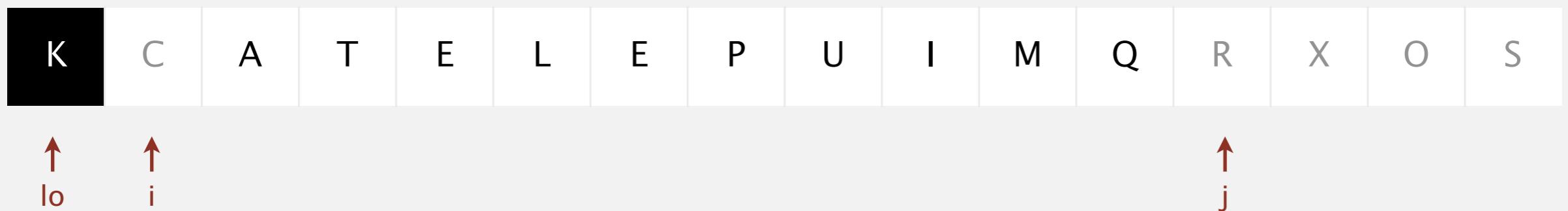
stop j scan and exchange  $a[i]$  with  $a[j]$

# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

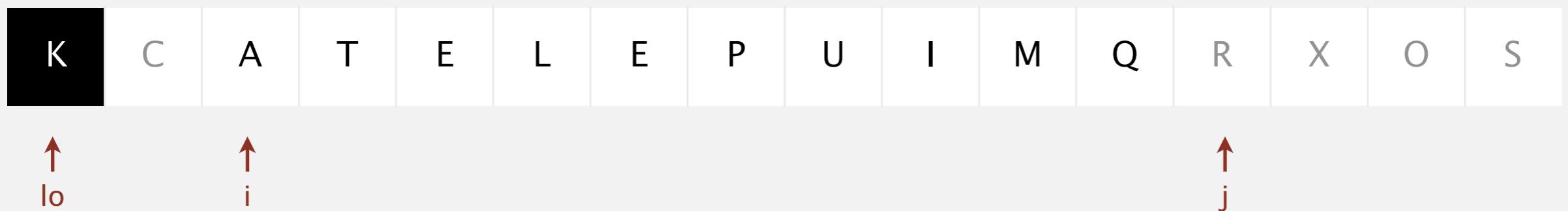


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

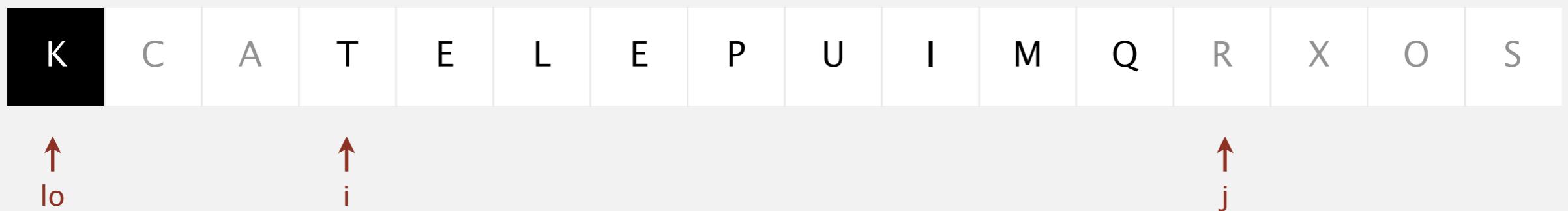


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



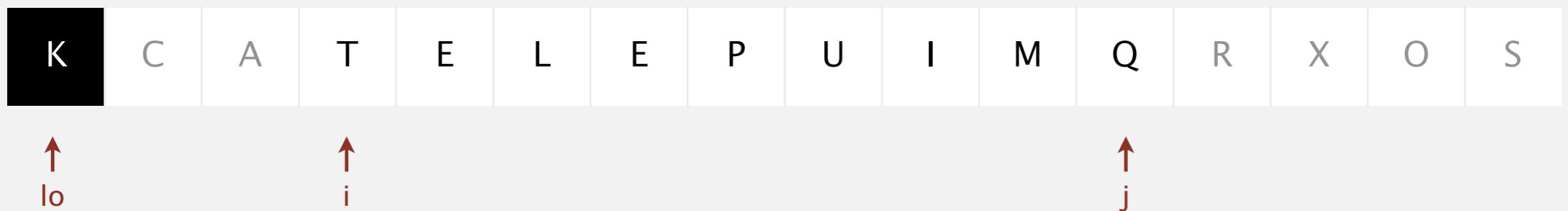
stop i scan because  $a[i] \geq a[lo]$

# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

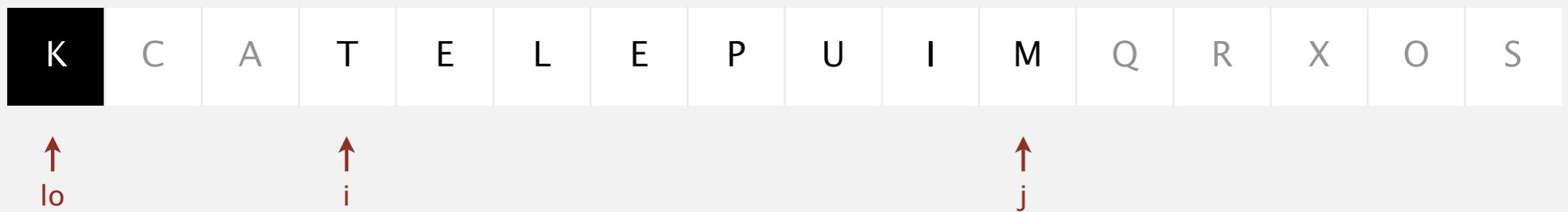


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

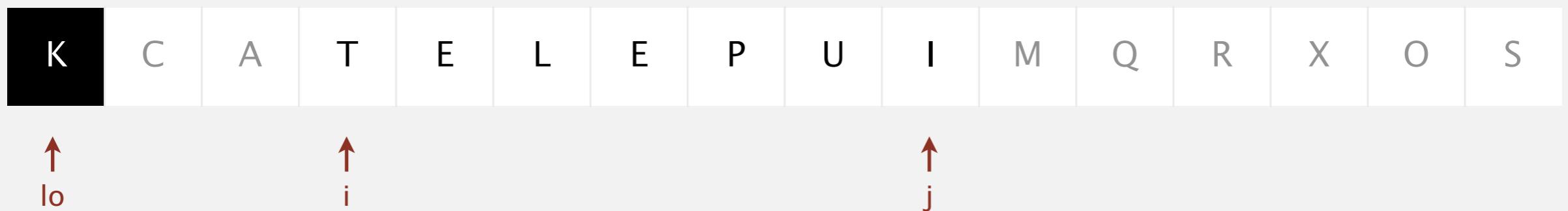


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



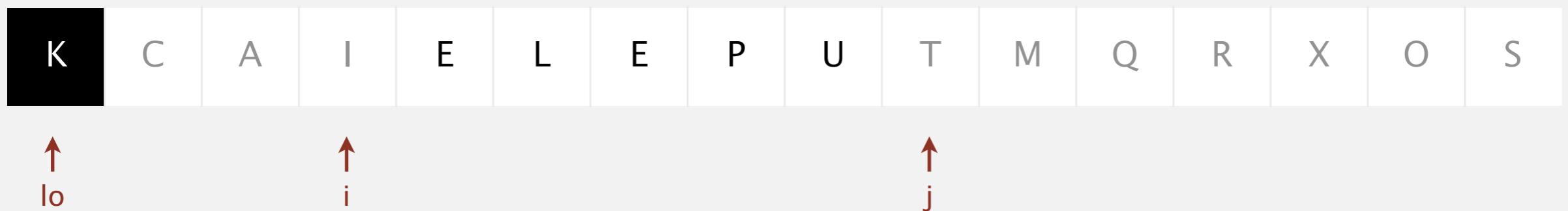
stop j scan and exchange  $a[i]$  with  $a[j]$

# Quicksort partitioning demo

---

Repeat until  $i$  and  $j$  pointers cross.

- Scan  $i$  from left to right so long as  $(a[i] < a[lo])$ .
- Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
- Exchange  $a[i]$  with  $a[j]$ .

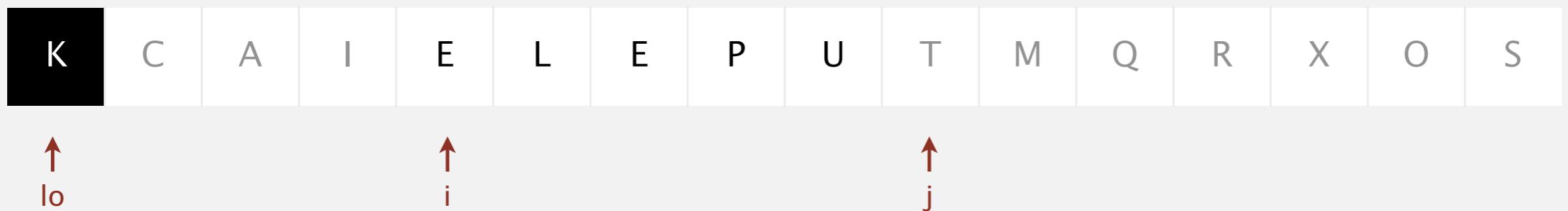


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

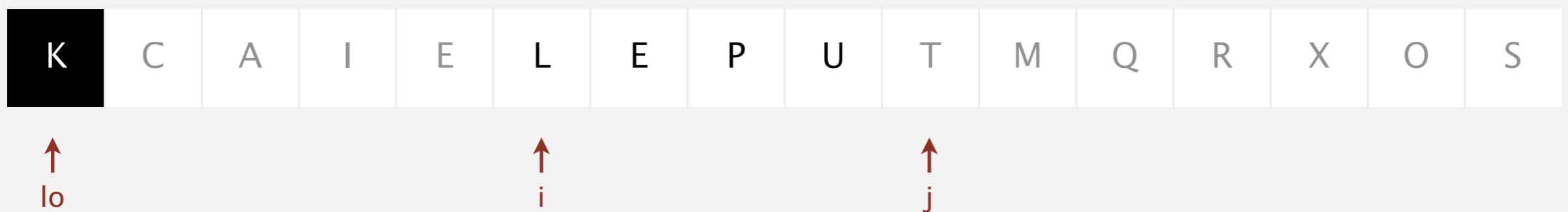


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



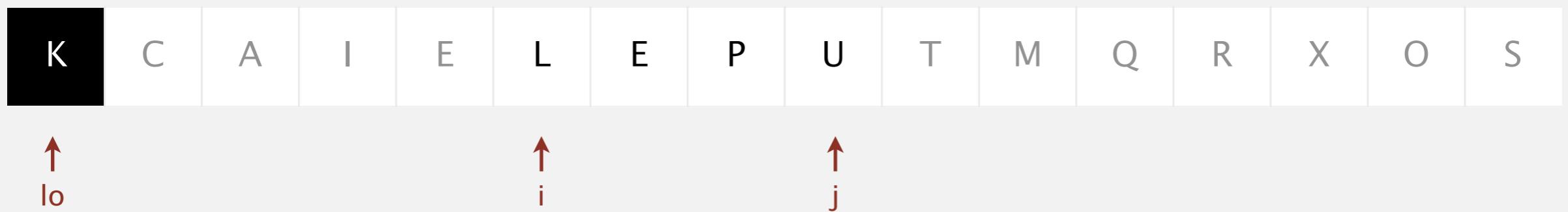
stop i scan because  $a[i] \geq a[lo]$

# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

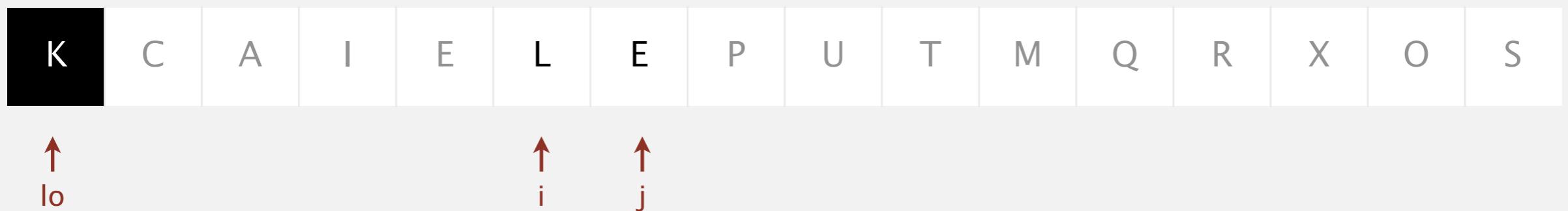


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



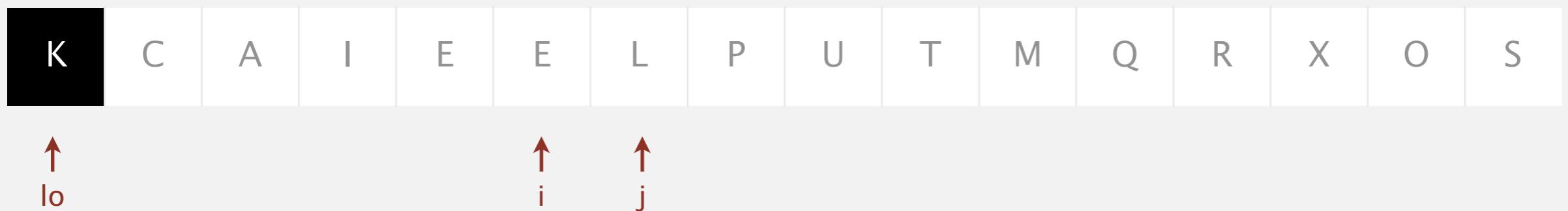
stop j scan and exchange  $a[i]$  with  $a[j]$

# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

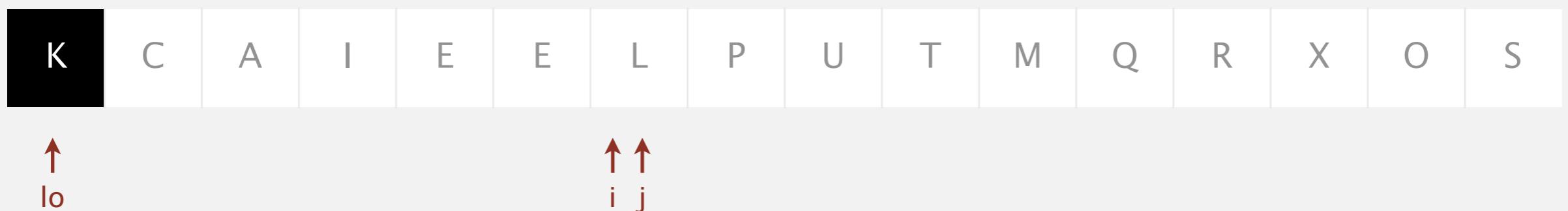


# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



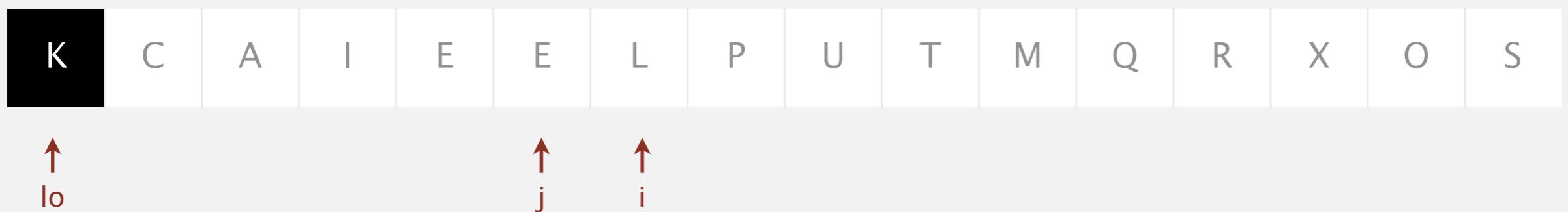
stop i scan because  $a[i] \geq a[lo]$

# Quicksort partitioning demo

---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .



stop j scan because  $a[j] \leq a[lo]$

# Quicksort partitioning demo

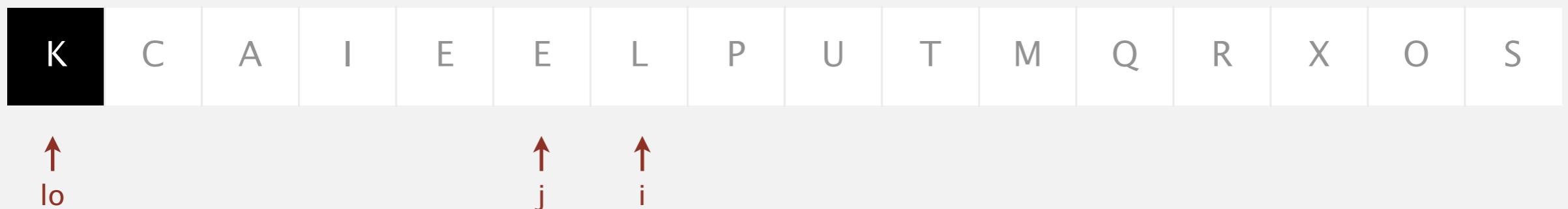
---

Repeat until i and j pointers cross.

- Scan i from left to right so long as ( $a[i] < a[lo]$ ).
- Scan j from right to left so long as ( $a[j] > a[lo]$ ).
- Exchange  $a[i]$  with  $a[j]$ .

When pointers cross.

- Exchange  $a[lo]$  with  $a[j]$ .



# Quicksort partitioning demo

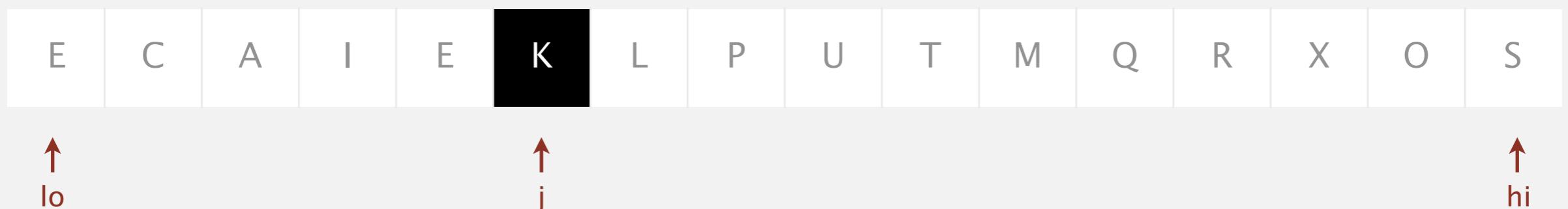
---

Repeat until  $i$  and  $j$  pointers cross.

- Scan  $i$  from left to right so long as  $(a[i] < a[lo])$ .
- Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
- Exchange  $a[i]$  with  $a[j]$ .

When pointers cross.

- Exchange  $a[lo]$  with  $a[j]$ .



partitioned!

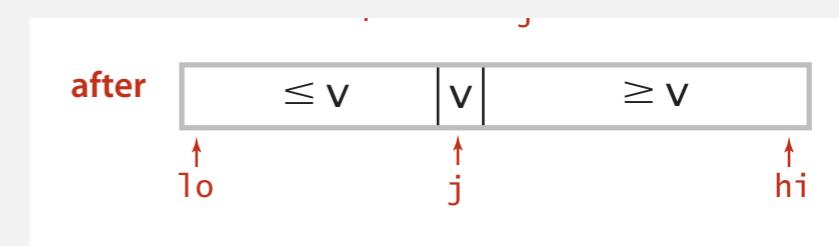
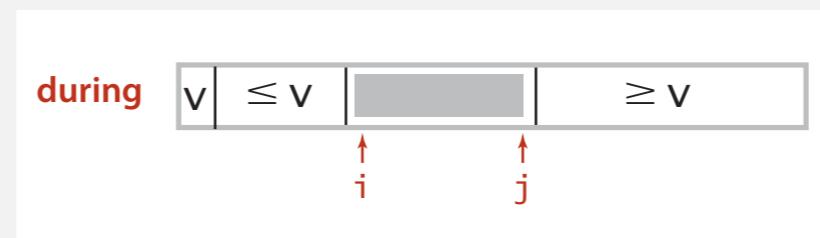
# Quicksort: Java code for partitioning

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

        while (less(a[lo], a[--j]))           find item on right to swap
            if (j == lo) break;

        if (i >= j) break;                  check if pointers cross
        exch(a, i, j);                   swap
    }

    exch(a, lo, j);                  swap with partitioning item
    return j;                        return index of item now known to be in place
}
```



# Quicksort: Java implementation

```
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)

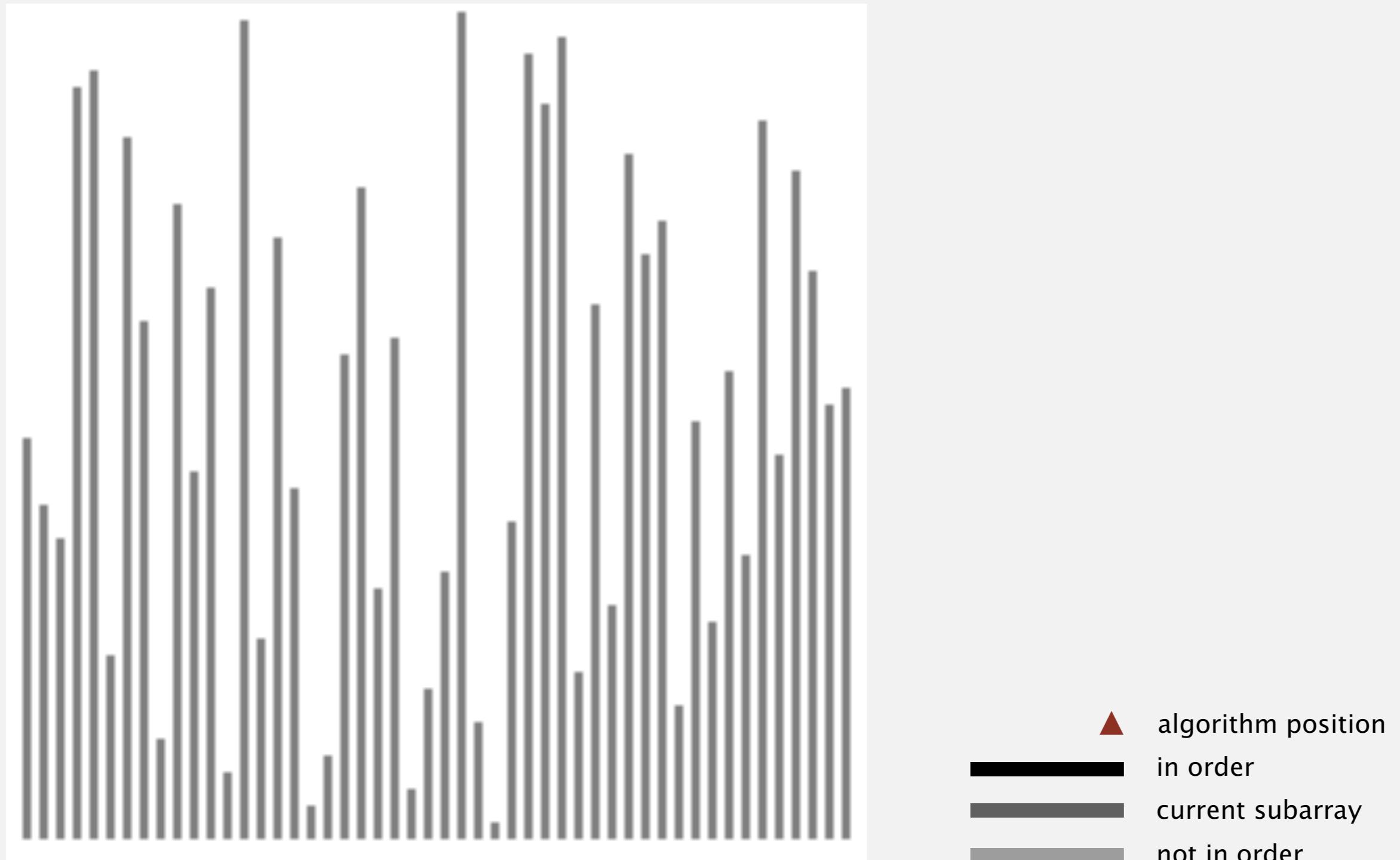
# Quicksort trace

	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	0	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	0	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	0	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S
	1			A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S
	4			A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	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			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	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			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

Quicksort trace (array contents after each partition)

# Quicksort animation

## 50 random items



<http://www.sorting-algorithms.com/quick-sort>

## Quicksort: implementation details

---

**Partitioning in-place.** Using an extra array makes partitioning easier (and stable), but is not worth the cost.

**Terminating the loop.** Testing whether the pointers cross is trickier than it might seem.

**Equal keys.** When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item's key. ← stay tuned

**Preserving randomness.** Shuffling is needed for performance guarantee.

**Equivalent alternative.** Pick a random partitioning item in each subarray.

# Quicksort: empirical analysis (1961)

## Running time estimates:

- Algol 60 implementation.
- National-Elliott 405 computer.

**Table 1**

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

\* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

sorting N 6-word items with 1-word keys



**Elliott 405 magnetic disc  
(16K words)**

# Quicksort: empirical analysis

---

## Running time estimates:

- Home PC executes  $10^8$  compares/second.
- Supercomputer executes  $10^{12}$  compares/second.

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

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

# Quicksort: best-case analysis

Best case. Number of compares is  $\sim N \lg N$ .

			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

# Quicksort: worst-case analysis

Worst case. Number of compares is  $\sim \frac{1}{2} N^2$ .

			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

# Quicksort: average-case analysis

**Proposition.** The average number of compares  $C_N$  to quicksort an array of  $N$  distinct keys is  $\sim 2N \ln N$  (and the number of exchanges is  $\sim \frac{1}{3}N \ln N$ ).

Pf.  $C_N$  satisfies the recurrence  $C_0 = C_1 = 0$  and for  $N \geq 2$ :

$$C_N = \underset{\text{partitioning}}{(N+1)} + \left( \frac{C_0 + C_{N-1}}{N} \right) + \left( \frac{C_1 + C_{N-2}}{N} \right) + \dots + \left( \frac{C_{N-1} + C_0}{N} \right)$$

↓      ↓  
left      right  
↑

- Multiply both sides by  $N$  and collect terms: partitioning probability

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

- Subtract from this equation the same equation for  $N - 1$ :

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

- Rearrange terms and divide by  $N(N+1)$ :

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

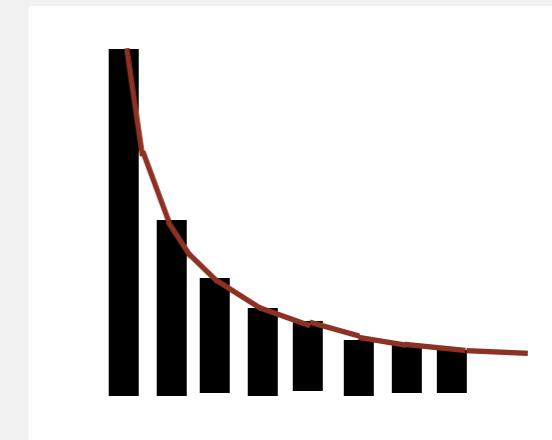
## Quicksort: average-case analysis

- Repeatedly apply previous equation:

$$\begin{aligned}\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \quad \leftarrow \text{substitute previous equation} \\ &= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \dots + \frac{2}{N+1}\end{aligned}$$

- Approximate sum by an integral:

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



- Finally, the desired result:

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

## Quicksort: summary of performance characteristics

---

Quicksort is a (Las Vegas) randomized algorithm.

- Guaranteed to be correct.
- Running time depends on random shuffle.

Average case. Expected number of compares is  $\sim 1.39 N \lg N$ .

- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

Best case. Number of compares is  $\sim N \lg N$ .

Worst case. Number of compares is  $\sim \frac{1}{2} N^2$ .

[ but more likely that lightning bolt strikes computer during execution ]



# Quicksort properties

---

Proposition. Quicksort is an **in-place** sorting algorithm.

Pf.

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

can guarantee logarithmic depth by recurring  
on smaller subarray before larger subarray  
(but requires using an explicit stack)

Proposition. Quicksort is **not stable**.

Pf. [ by counterexample ]

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: practical improvements

---

## Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for  $\approx 10$  items.

```
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: practical improvements

---

## Median of sample.

- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.



~  $12/7 N \ln N$  compares (14% fewer)  
~  $12/35 N \ln N$  exchanges (3% more)

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

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

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

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.3 QUICKSORT

---

- ▶ *quicksort*
- ▶ ***selection***
- ▶ *duplicate keys*
- ▶ *system sorts*

# Selection

---

**Goal.** Given an array of  $N$  items, find the  $k^{\text{th}}$  smallest item.

**Ex.** Min ( $k = 0$ ), max ( $k = N - 1$ ), median ( $k = N/2$ ).

## Applications.

- Order statistics.
- Find the "top  $k$ ."

## Use theory as a guide.

- Easy  $N \log N$  upper bound. How?
- Easy  $N$  upper bound for  $k = 1, 2, 3$ . How?
- Easy  $N$  lower bound. Why?

## Which is true?

- $N \log N$  lower bound?  is selection as hard as sorting?
- $N$  upper bound?  is there a linear-time algorithm?

# Quick-select

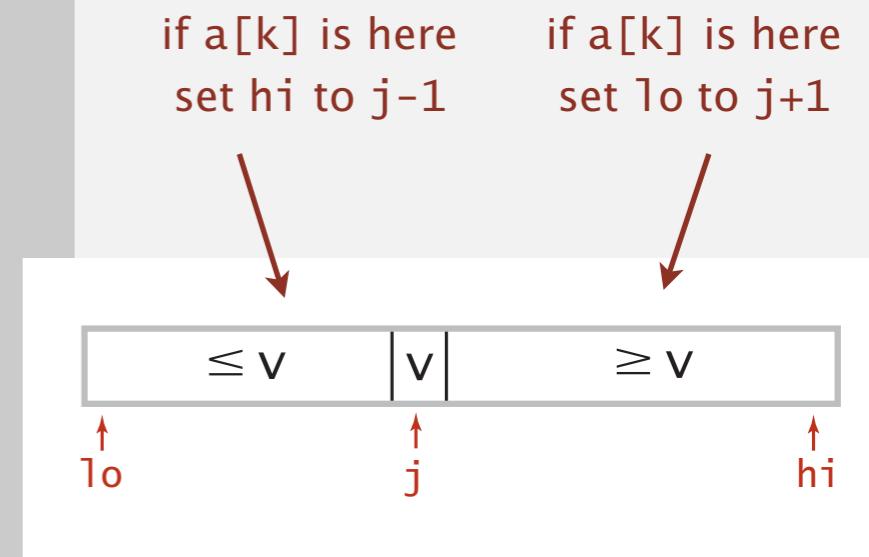
Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .



Repeat in **one** subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

```
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 demo

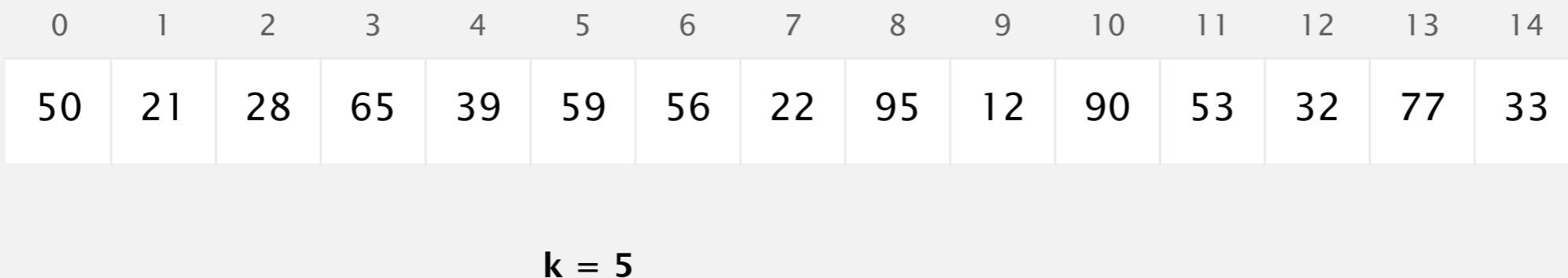
---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

**select element of rank  $k = 5$**



# Quick-select demo

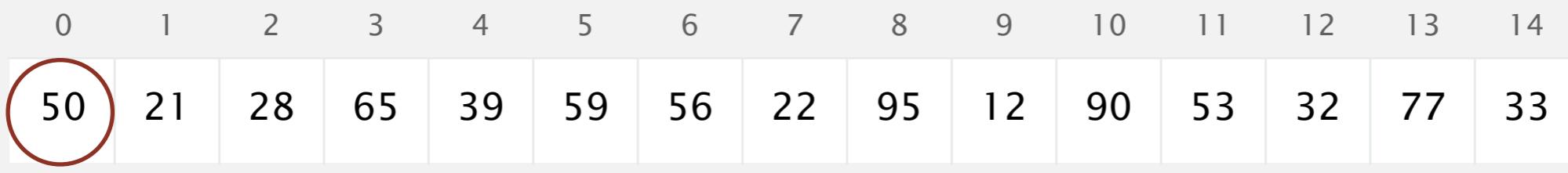
---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

partition on leftmost entry



$k = 5$

# Quick-select demo

---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

partitioned array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
22	21	28	33	39	32	12	50	95	56	90	53	59	77	65

$k = 5$

# Quick-select demo

---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

can safely ignore right subarray

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
22	21	28	33	39	32	12	50	95	56	90	53	59	77	65

$k = 5$

# Quick-select demo

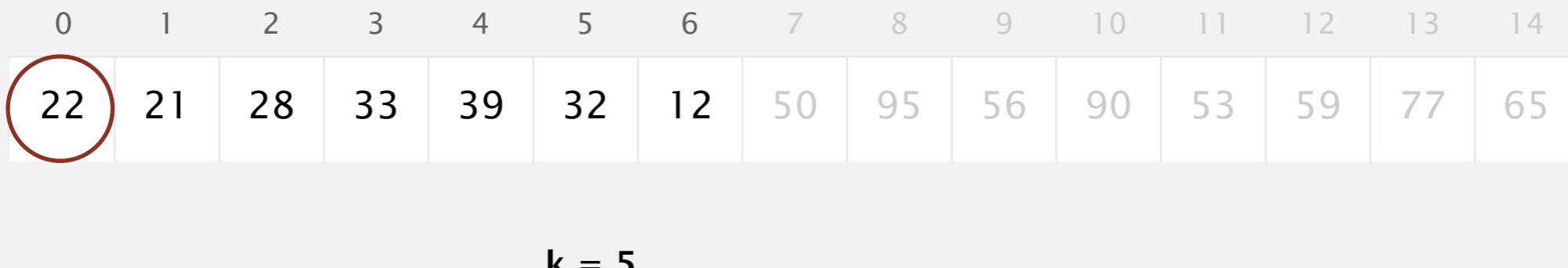
---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

partition on leftmost entry



# Quick-select demo

---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

partitioned array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	21	22	33	39	32	28	50	95	56	90	53	59	77	65

$k = 5$

# Quick-select demo

---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

can safely ignore left subarray



$k = 5$

# Quick-select demo

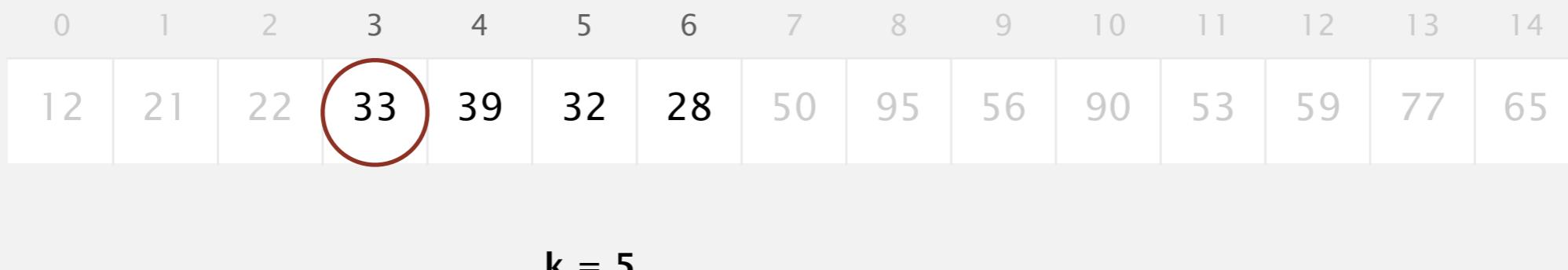
---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

partition on leftmost entry



# Quick-select demo

---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

partitioned array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	21	22	32	28	33	39	50	95	56	90	53	59	77	65

$k = 5$

# Quick-select demo

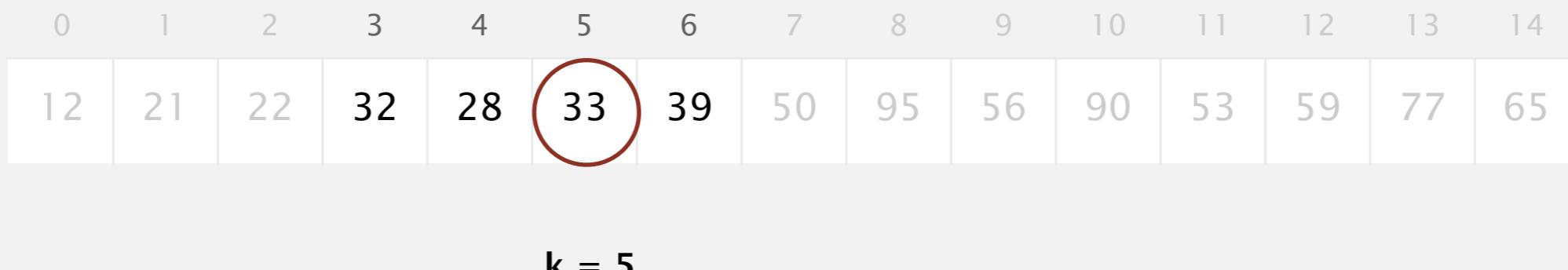
---

Partition array so that:

- Entry  $a[j]$  is in place.
- No larger entry to the left of  $j$ .
- No smaller entry to the right of  $j$ .

Repeat in one subarray, depending on  $j$ ; finished when  $j$  equals  $k$ .

stop: partitioning item is at index  $k$



## Quick-select: mathematical analysis

---

Proposition. Quick-select takes linear time on average.

Pf sketch.

- Intuitively, each partitioning step splits array approximately in half:  
 $N + N/2 + N/4 + \dots + 1 \sim 2N$  compares.
- Formal analysis similar to quicksort analysis yields:

$$\begin{aligned} C_N &= 2N + 2k \ln(N/k) + 2(N-k) \ln(N/(N-k)) \\ &\leq (2 + 2 \ln 2)N \end{aligned}$$

- Ex:  $(2 + 2 \ln 2)N \approx 3.38N$  compares to find median ( $k = N/2$ ).

# Theoretical context for selection

---

**Proposition.** [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] Compare-based selection algorithm whose worst-case running time is linear.

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  $5.430\dot{5} n$  comparisons are ever required. This bound is improved for

**Remark.** Constants are high  $\Rightarrow$  not used in practice.

**Use theory as a guide.**

- Still worthwhile to seek **practical** linear-time (worst-case) algorithm.
- Until one is discovered, use quick-select (if you don't need a full sort).

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.3 QUICKSORT

---

- ▶ *quicksort*
- ▶ *selection*
- ▶ *duplicate keys*
- ▶ *system sorts*

# Duplicate keys

---

Often, purpose of sort is to bring items with equal keys together.

- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.

Chicago	09:25:52
Chicago	09:03:13
Chicago	09:21:05
Chicago	09:19:46
Chicago	09:19:32
Chicago	09:00:00
Chicago	09:35:21
Chicago	09:00:59
Houston	09:01:10
Houston	09:00:13
Phoenix	09:37:44
Phoenix	09:00:03
Phoenix	09:14:25
Seattle	09:10:25
Seattle	09:36:14
Seattle	09:22:43
Seattle	09:10:11
Seattle	09:22:54

↑  
key

## War story (system sort in C)

---

A beautiful bug report. [Allan Wilks and Rick Becker, 1991]

We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

```
main (int argc, char**argv) {
    int n = atoi(argv[1]), i, x[100000];
    for (i = 0; i < n; i++)
        x[i] = i;
    for ( ; i < 2*n; i++)
        x[i] = 2*n-i-1;
    qsort(x, 2*n, sizeof(int), intcmp);
}
```

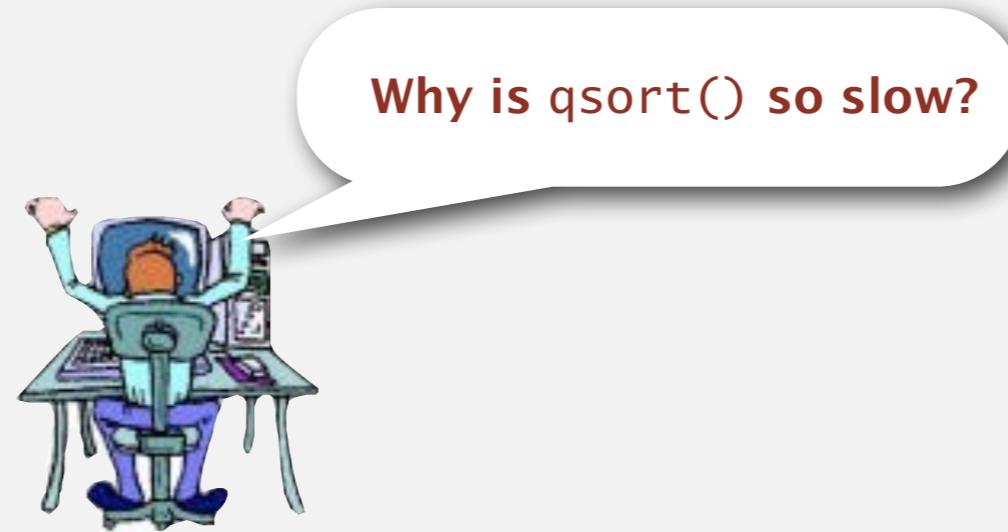
Here are the timings on our machine:

```
$ time a.out 2000
real    5.85s
$ time a.out 4000
real   21.64s
$time a.out 8000
real   85.11s
```

## War story (system sort in C)

---

Bug. A qsort() call that should have taken seconds was taking minutes.



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

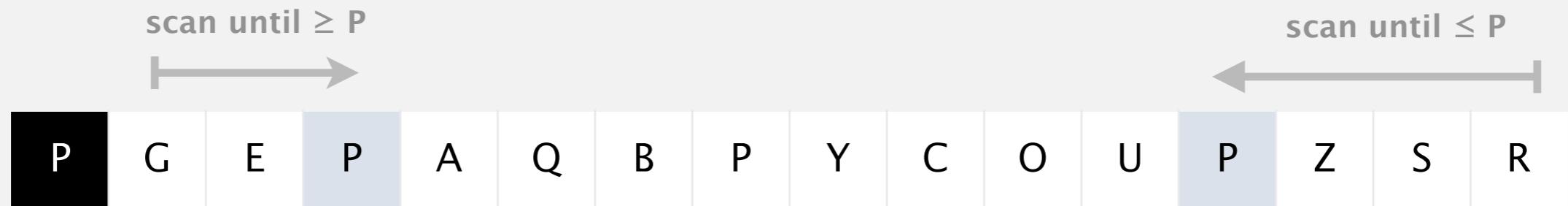
- Version 7 Unix (1979): quadratic time to sort organ-pipe arrays.
- BSD Unix (1983): quadratic time to sort random arrays of 0s and 1s.



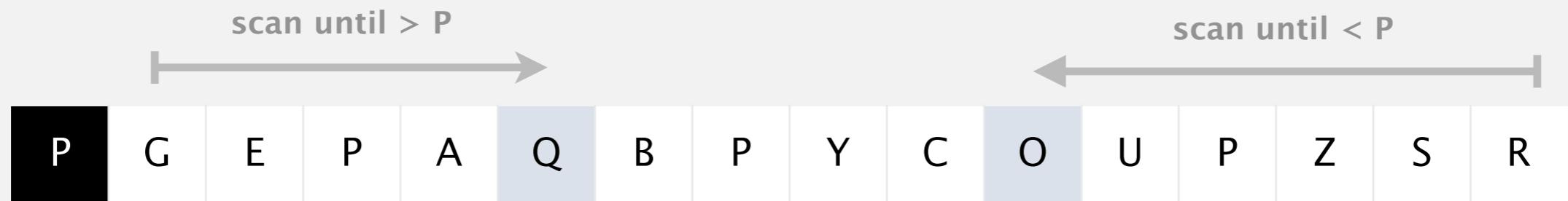
# Duplicate keys: stop on equal keys

---

Our partitioning subroutine stops both scans on equal keys.



Q. Why not continue scans on equal keys?



# Partitioning an array with all equal keys

---

		a[ ]															
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	

## Duplicate keys: partitioning strategies

---

Bad. Don't stop scans on equal keys.

[  $\sim \frac{1}{2} N^2$  compares when all keys equal ]

B A A B A B B B C C C

A A A A A A A A A A A A

Good. Stop scans on equal keys.

[  $\sim N \lg N$  compares when all keys equal ]

B A A B A B C C B C B

A A A A A A A A A A A A

Better. Put all equal keys in place. How?

[  $\sim N$  compares when all keys equal ]

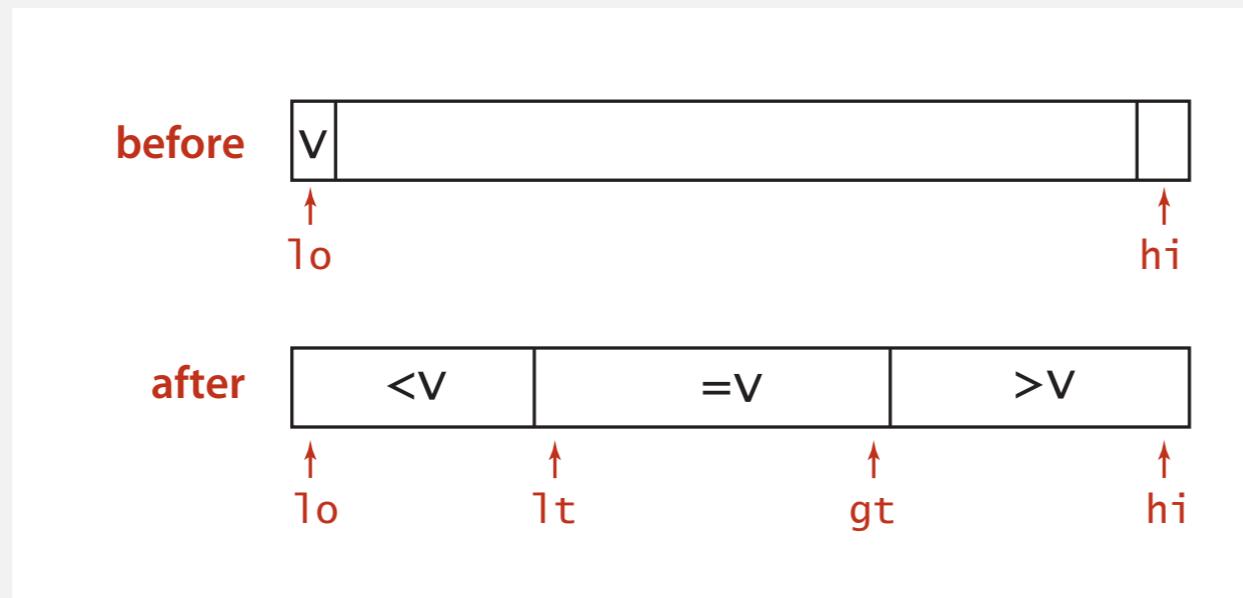
A A A B B B B C C C

A A A A A A A A A A A A

# 3-way partitioning

**Goal.** Partition array into **three** parts so that:

- Entries between  $l_t$  and  $g_t$  equal to the partition item.
- No larger entries to left of  $l_t$ .
- No smaller entries to right of  $g_t$ .



**Dutch national flag problem.** [Edsger Dijkstra]

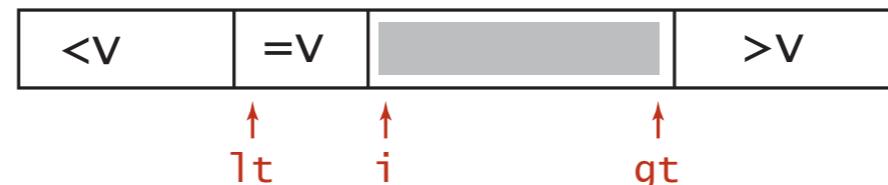
- Conventional wisdom until mid 1990s: not worth doing.
- Now incorporated into C library `qsort()` and Java 6 system sort.

# Dijkstra 3-way partitioning demo

- Let  $v$  be partitioning item  $a[lo]$ .
- Scan  $i$  from left to right.
  - $(a[i] < v)$ : exchange  $a[lt]$  with  $a[i]$ ; increment both  $lt$  and  $i$
  - $(a[i] > v)$ : exchange  $a[gt]$  with  $a[i]$ ; decrement  $gt$
  - $(a[i] == v)$ : increment  $i$

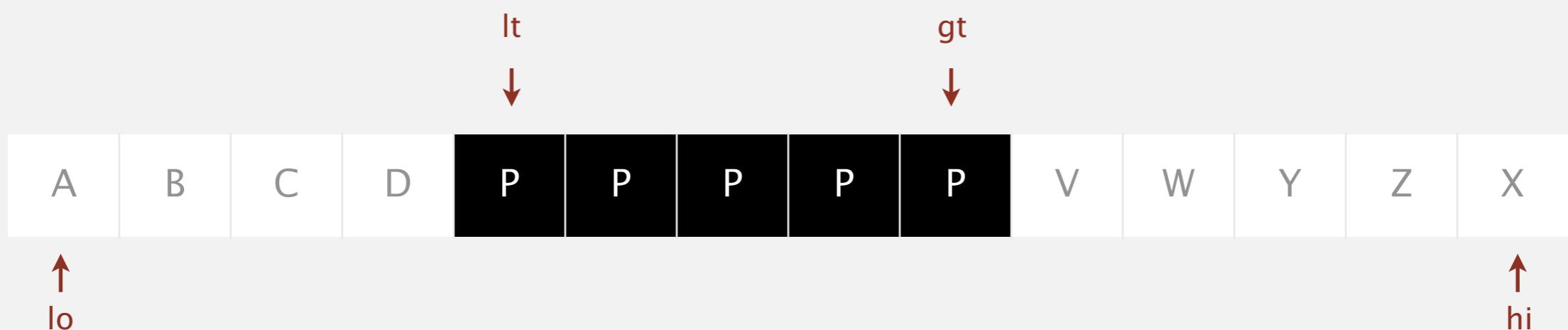


invariant

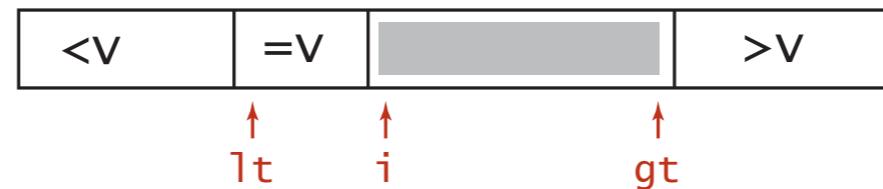


# Dijkstra 3-way partitioning demo

- Let  $v$  be partitioning item  $a[lo]$ .
- Scan  $i$  from left to right.
  - $(a[i] < v)$ : exchange  $a[lt]$  with  $a[i]$ ; increment both  $lt$  and  $i$
  - $(a[i] > v)$ : exchange  $a[gt]$  with  $a[i]$ ; decrement  $gt$
  - $(a[i] == v)$ : increment  $i$

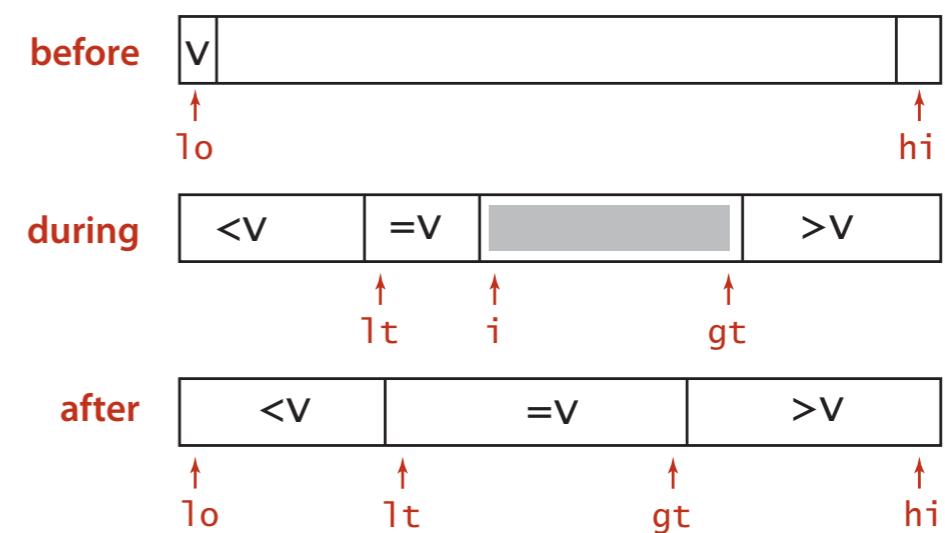


invariant



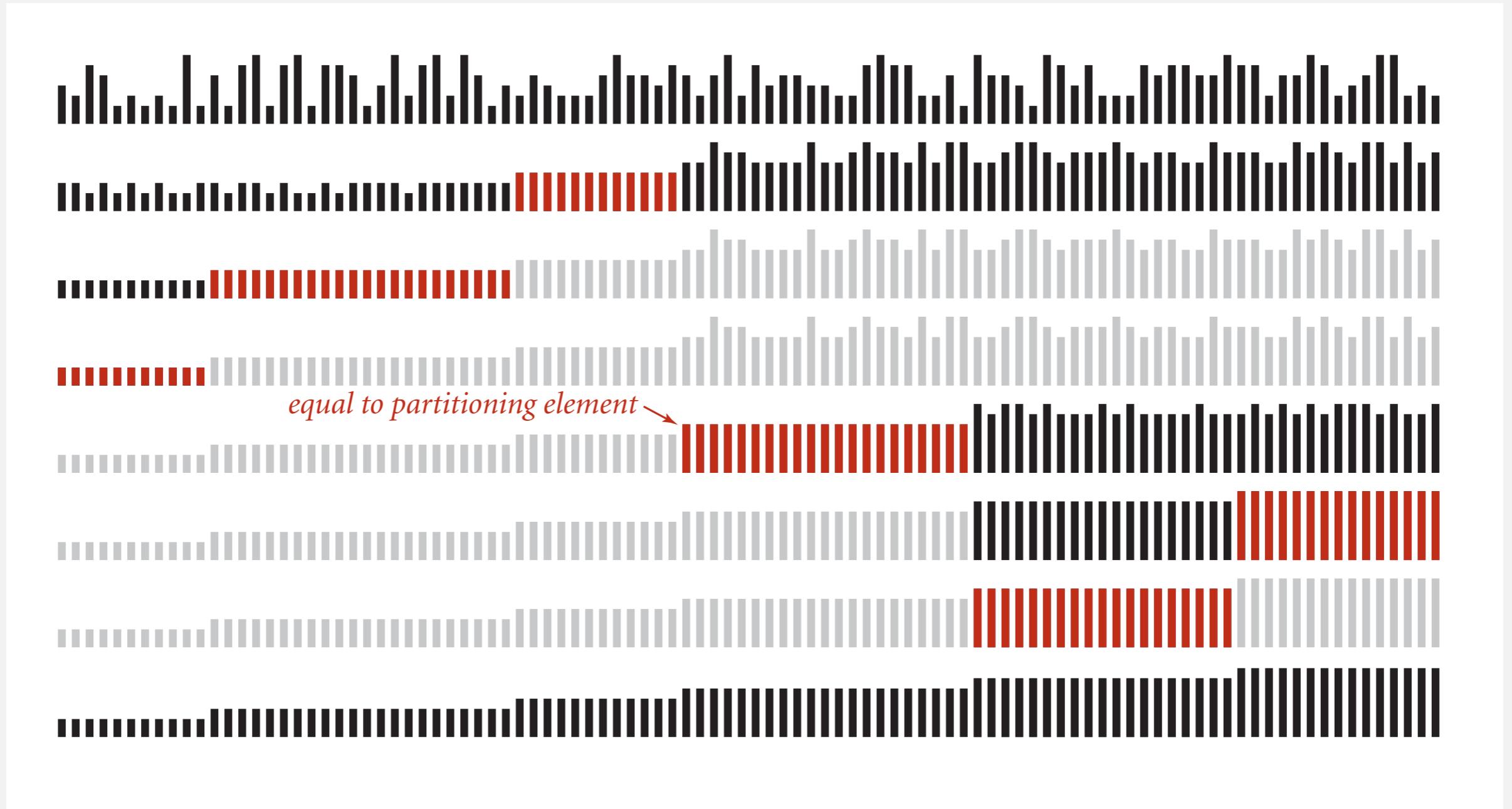
# 3-way quicksort: Java implementation

```
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);
}
```



## 3-way quicksort: visual trace

---



## Duplicate keys: lower bound

---

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

$$\lg \left( \frac{N!}{x_1! x_2! \cdots x_n!} \right) \sim - \sum_{i=1}^n x_i \lg \frac{x_i}{N} \quad \begin{matrix} \leftarrow \\ N \lg N \text{ when all distinct;} \\ \text{linear when only a constant number of distinct keys} \end{matrix}$$

compares in the worst case.

**Proposition.** The expected number of compares to 3-way quicksort an array is **entropy optimal** (proportional to sorting lower bound).

**Pf.** [beyond scope of course]

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

# Sorting summary

---

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$N$ exchanges
insertion	✓	✓	$N$	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small $N$ or partially ordered
shell	✓		$N \log_3 N$	?	$c N^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
timsort		✓	$N$	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
quick	✓		$N \lg N$	$2 N \ln N$	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
3-way quick	✓		$N$	$2 N \ln N$	$\frac{1}{2} N^2$	improves quicksort when duplicate keys
?	✓	✓	$N$	$N \lg N$	$N \lg N$	holy sorting grail

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.3 QUICKSORT

---

- ▶ *quicksort*
- ▶ *selection*
- ▶ *duplicate keys*
- ▶ ***system sorts***

# Sorting applications

---

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.
- 
- Find the median.
  - Identify statistical outliers.
  - Binary search in a database.
  - Find duplicates in a mailing list.
- Data compression.
  - Computer graphics.
  - Computational biology.
  - Load balancing on a parallel computer.
- ...  
obvious applications
- problems become easy once items are in sorted order
- non-obvious applications

# Engineering a system sort (in 1993)

## Bentley-McIlroy quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning item: median of 3 or Tukey's ninther.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning.

sample 9 items  
similar to Dijkstra 3-way partitioning  
(but fewer exchanges when not many equal keys)

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.

Very widely used. C, C++, Java 6, ....

# A beautiful mailing list post (Yaroslavskiy, September 2009)

---

## Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Hello All,

I'd like to share with you new **Dual-Pivot Quicksort** which is faster than the known implementations (theoretically and experimental). I'd like to propose to replace the JDK's Quicksort implementation by new one.

...

The new Dual-Pivot Quicksort uses **\*two\*** pivots elements in this manner:

1. Pick an elements P1, P2, called pivots from the array.
2. Assume that P1 <= P2, otherwise swap it.
3. Reorder the array into three parts: those less than the smaller pivot, those larger than the larger pivot, and in between are those elements between (or equal to) the two pivots.
4. Recursively sort the sub-arrays.

The invariant of the Dual-Pivot Quicksort is:

[ < P1 | P1 <= & <= P2 } > P2 ]

...

# A beautiful mailing list post (Yaroslavskiy-Bloch-Bentley, October 2009)

---

## **Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort**

Date: Thu, 29 Oct 2009 11:19:39 +0000

Subject: Replace quicksort in java.util.Arrays with dual-pivot implementation

Changeset: b05abb410c52

Author: alanb

Date: 2009-10-29 11:18 +0000

URL: <http://hg.openjdk.java.net/jdk7/tl/jdk/rev/b05abb410c52>

6880672: Replace quicksort in java.util.Arrays with dual-pivot implementation

Reviewed-by: jjb

Contributed-by: vladimir.yaroslavskiy at sun.com, joshua.bloch at google.com, jbentley at avaya.com

! make/java/java/FILES\_java.gmk

! src/share/classes/java/util/Arrays.java

+ src/share/classes/java/util/DualPivotQuicksort.java

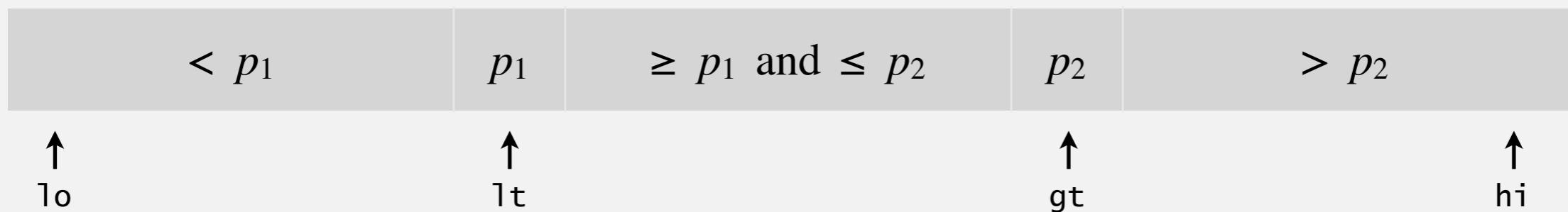
<http://mail.openjdk.java.net/pipermail/compiler-dev/2009-October.txt>

## Dual-pivot quicksort

---

Use **two** partitioning items  $p_1$  and  $p_2$  and partition into three subarrays:

- Keys less than  $p_1$ .
- Keys between  $p_1$  and  $p_2$ .
- Keys greater than  $p_2$ .



Recursively sort three subarrays.

Note. Skip middle subarray if  $p_1 = p_2$ .

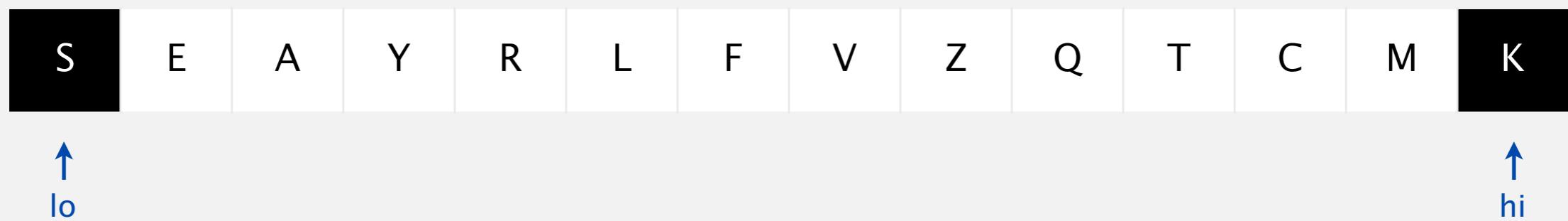
degenerates to Dijkstra's 3-way partitioning

# Dual-pivot partitioning demo

---

## Initialization.

- Choose  $a[lo]$  and  $a[hi]$  as partitioning items.
- Exchange if necessary to ensure  $a[lo] \leq a[hi]$ .

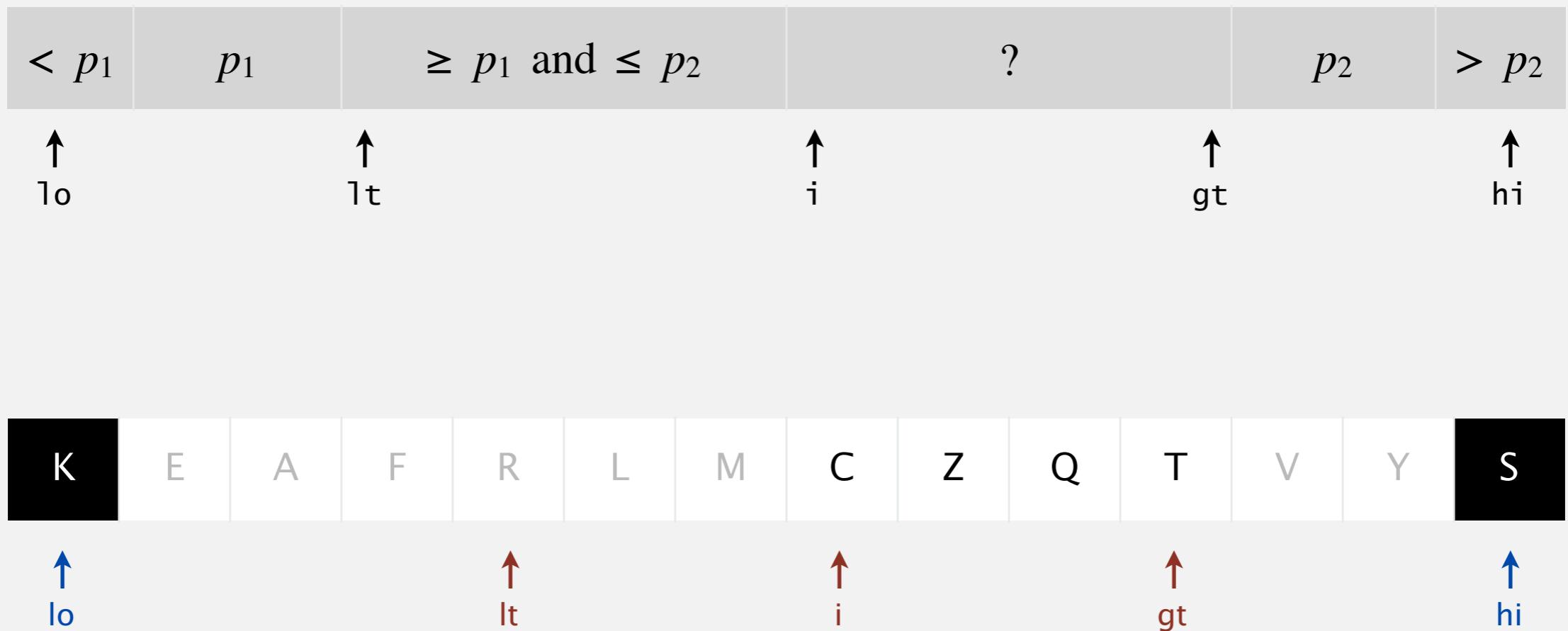


**exchange  $a[lo]$  and  $a[hi]$**

# Dual-pivot partitioning demo

Main loop. Repeat until  $i$  and  $gt$  pointers cross.

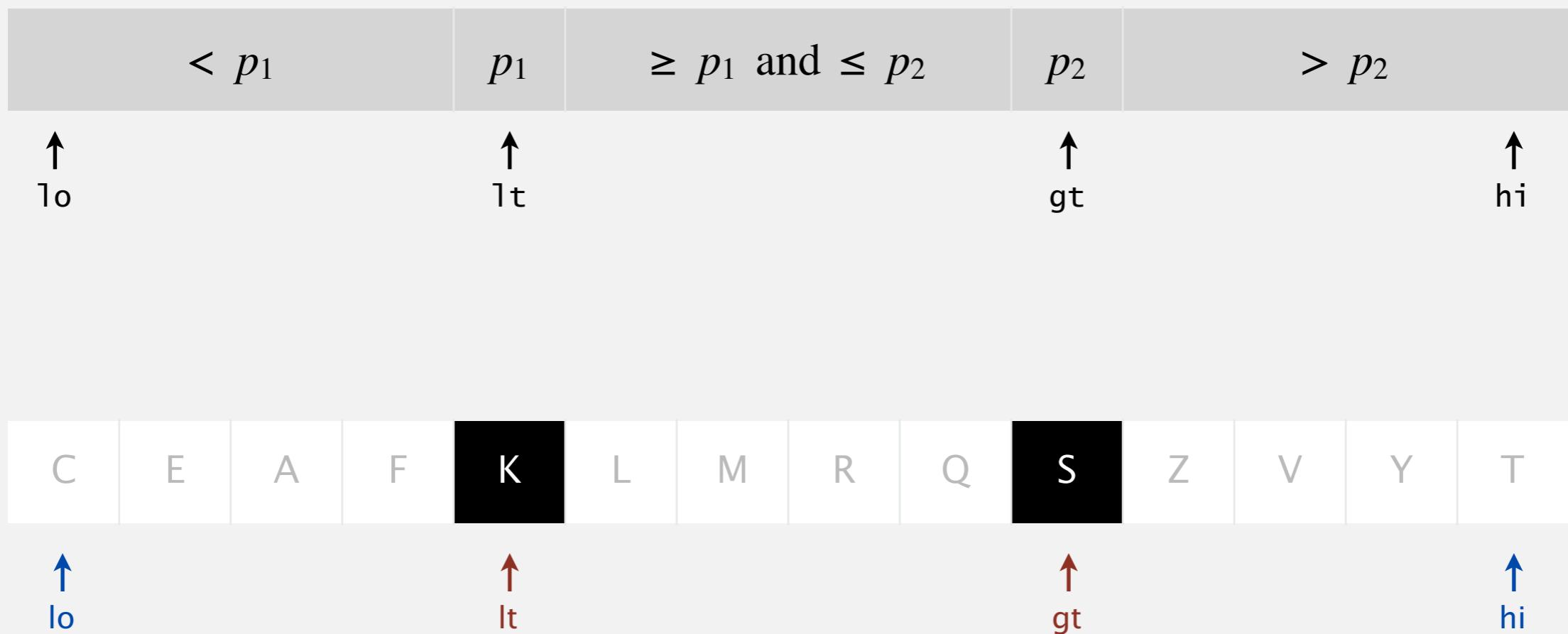
- If  $(a[i] < a[lo])$ , exchange  $a[i]$  with  $a[lt]$  and increment  $lt$  and  $i$ .
- Else if  $(a[i] > a[hi])$ , exchange  $a[i]$  with  $a[gt]$  and decrement  $gt$ .
- Else, increment  $i$ .



# Dual-pivot partitioning demo

Finalize.

- Exchange  $a[lo]$  with  $a[--lt]$ .
- Exchange  $a[hi]$  with  $a[++gt]$ .



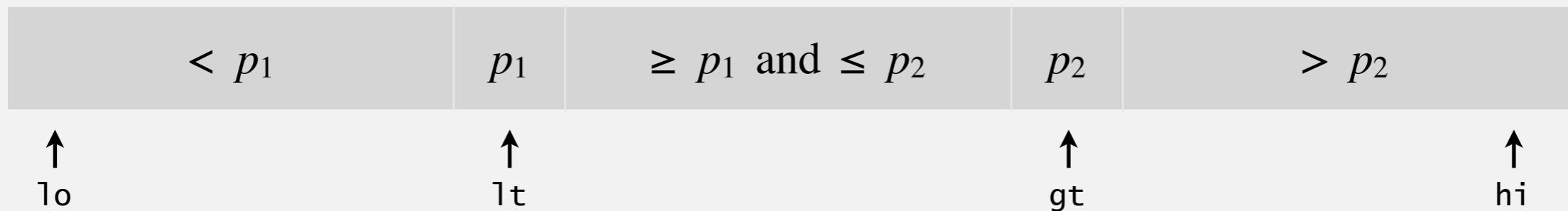
3-way partitioned

## Dual-pivot quicksort

---

Use **two** partitioning items  $p_1$  and  $p_2$  and partition into three subarrays:

- Keys less than  $p_1$ .
- Keys between  $p_1$  and  $p_2$ .
- Keys greater than  $p_2$ .

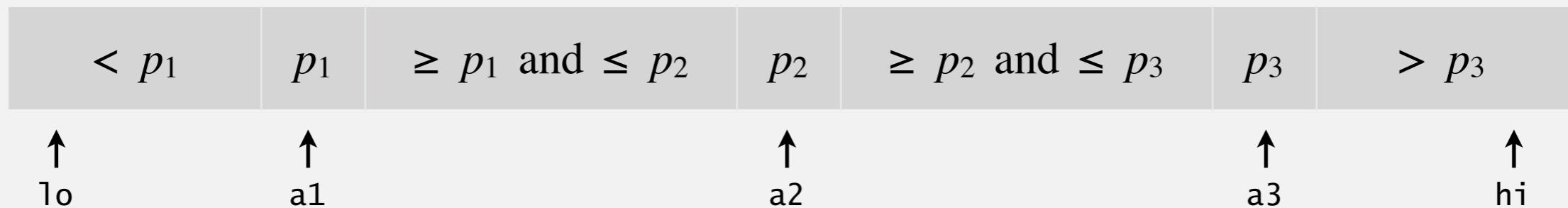


Now widely used. Java 7, Python unstable sort, Android, ...

# Three-pivot quicksort

Use **three** partitioning items  $p_1$ ,  $p_2$ , and  $p_3$  and partition into four subarrays:

- Keys less than  $p_1$ .
- Keys between  $p_1$  and  $p_2$ .
- Keys between  $p_2$  and  $p_3$ .
- Keys greater than  $p_3$ .



## Multi-Pivot Quicksort: Theory and Experiments

Shrinu Kushagra  
[skushagr@uwaterloo.ca](mailto:skushagr@uwaterloo.ca)  
University of Waterloo

Alejandro López-Ortiz  
[alopez-o@uwaterloo.ca](mailto:alopez-o@uwaterloo.ca)  
University of Waterloo

J. Ian Munro  
[jmunro@uwaterloo.ca](mailto:jmunro@uwaterloo.ca)  
University of Waterloo

Aurick Qiao  
[a2qiao@uwaterloo.ca](mailto:a2qiao@uwaterloo.ca)  
University of Waterloo

# System sort in Java 7

---

## Arrays.sort().

- Has one method for objects that are Comparable.
- Has an overloaded method for each primitive type.
- Has an overloaded method for use with a Comparator.
- Has overloaded methods for sorting subarrays.



## Algorithms.

- Dual-pivot quicksort for primitive types.
- Timsort for reference types.

Q. Why use different algorithms for primitive and reference types?

Bottom line. Use the system sort!