

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



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

## 2.3 QUICKSORT

---

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

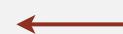
# Two classic sorting algorithms

---

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

- Java sort for objects.
- Perl, C++ stable sort, Python stable sort, Firefox JavaScript, ...

Quicksort.

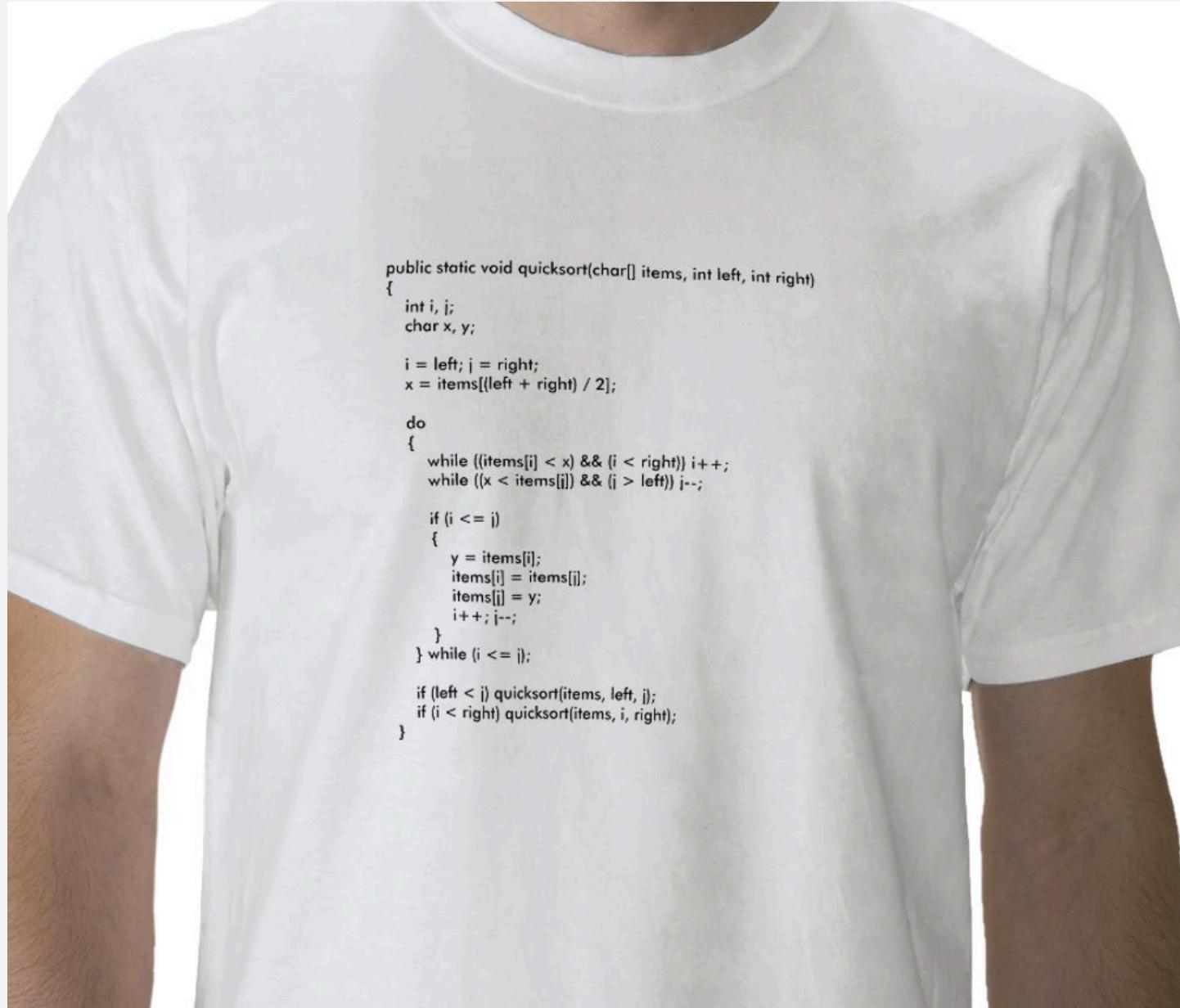


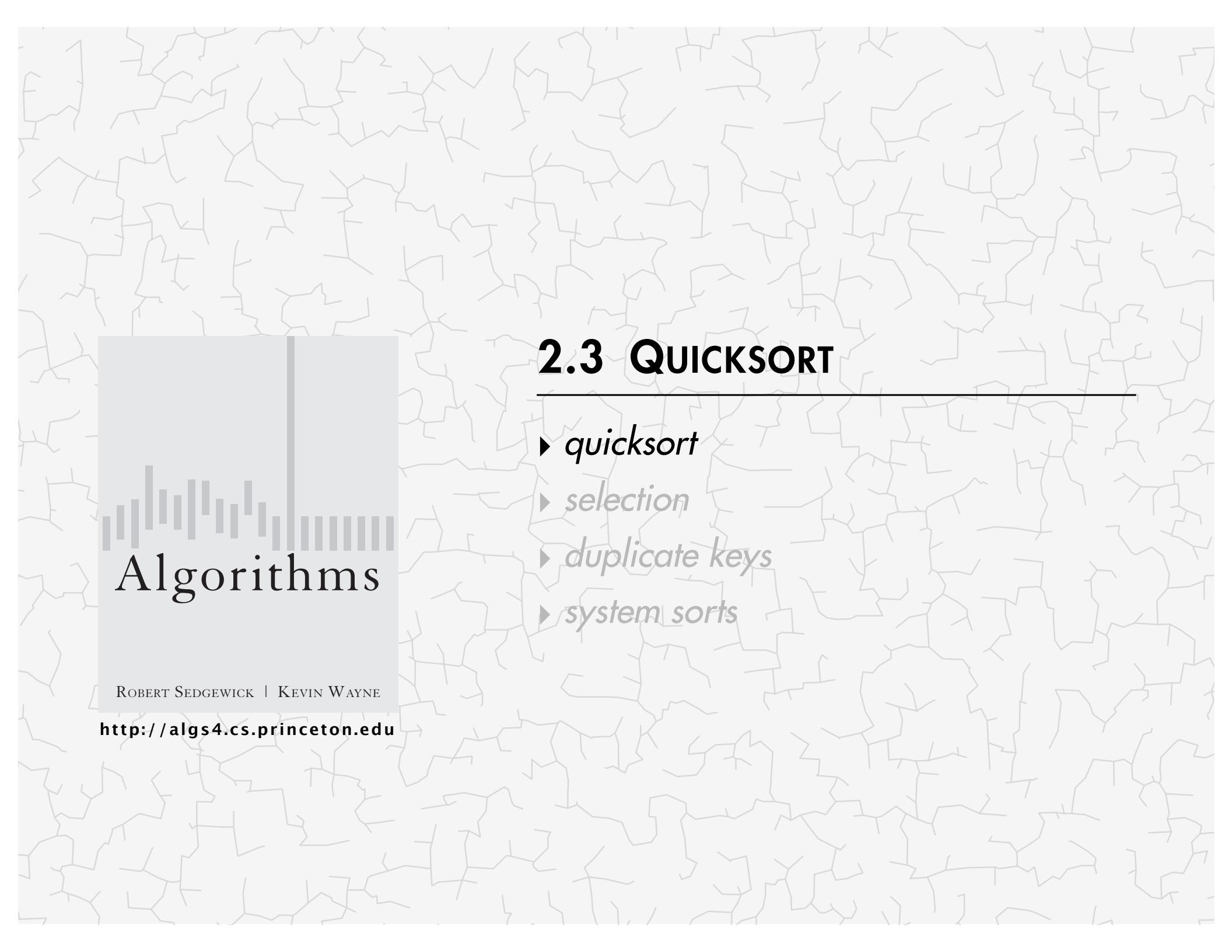
this lecture

- Java sort for primitive types.
- C qsort, Unix, Visual C++, Python, Matlab, Chrome JavaScript, ...

# Quicksort t-shirt

---





# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## 2.3 QUICKSORT

---

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

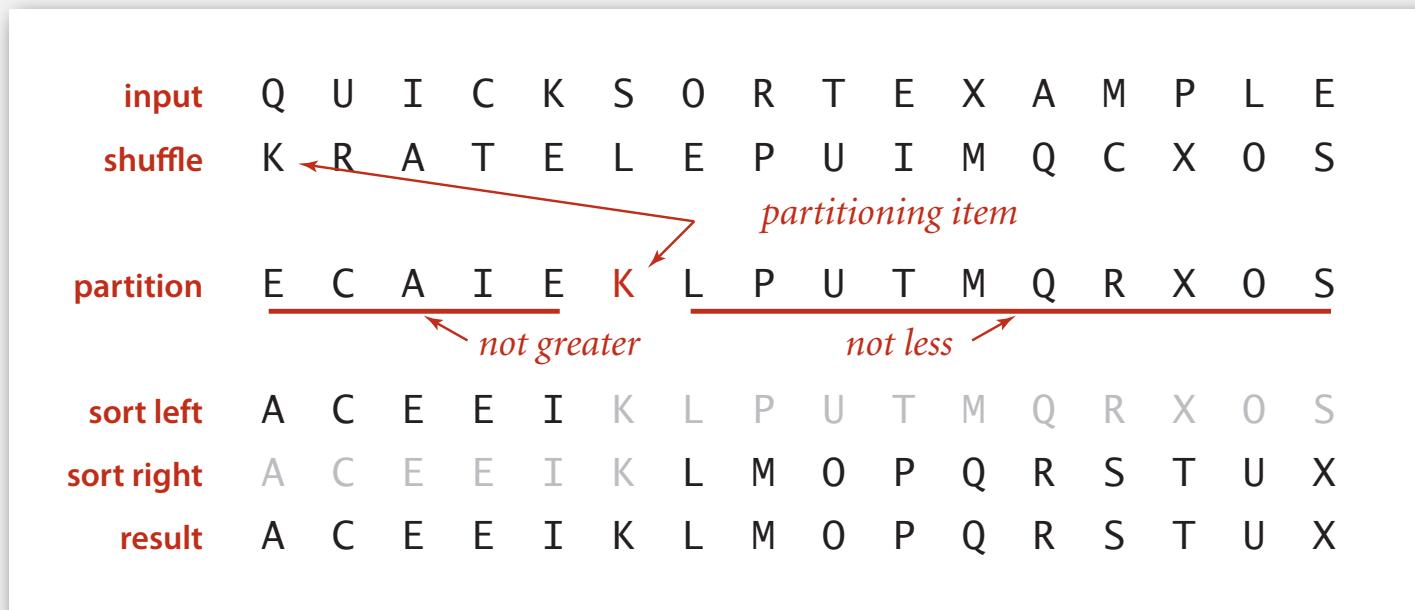
# Quicksort

Basic plan.

- **Shuffle** the array.
- **Partition** 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  $j$
- **Sort** each piece recursively.



Sir Charles Antony Richard Hoare  
1980 Turing Award

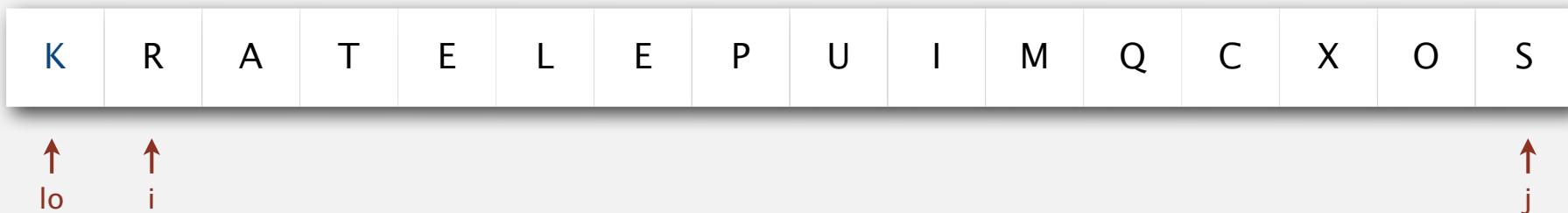


# Quicksort partitioning demo

---

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

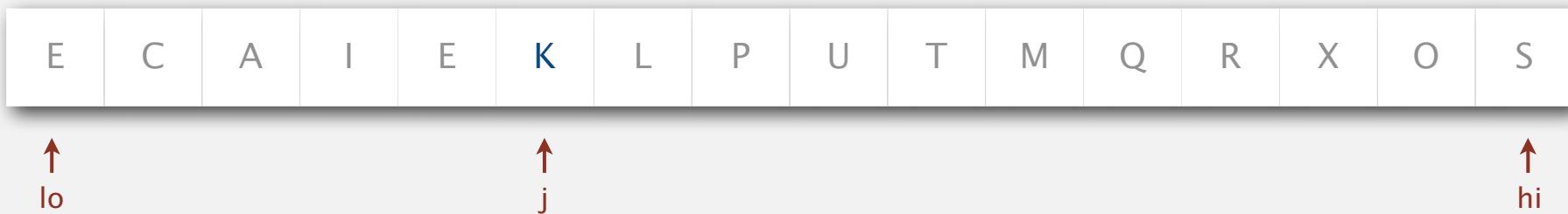
---

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  $a[j]$ .

When pointers cross.

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



partitioned!

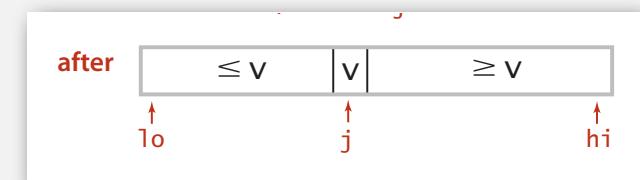
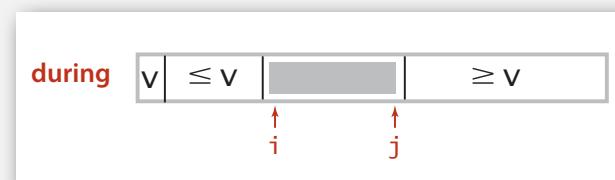
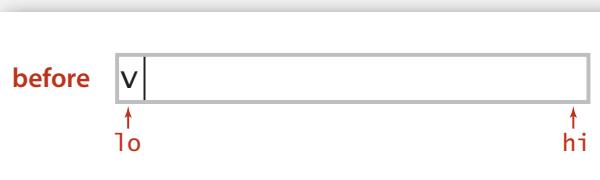
# Quicksort: Java code for partitioning

```
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)

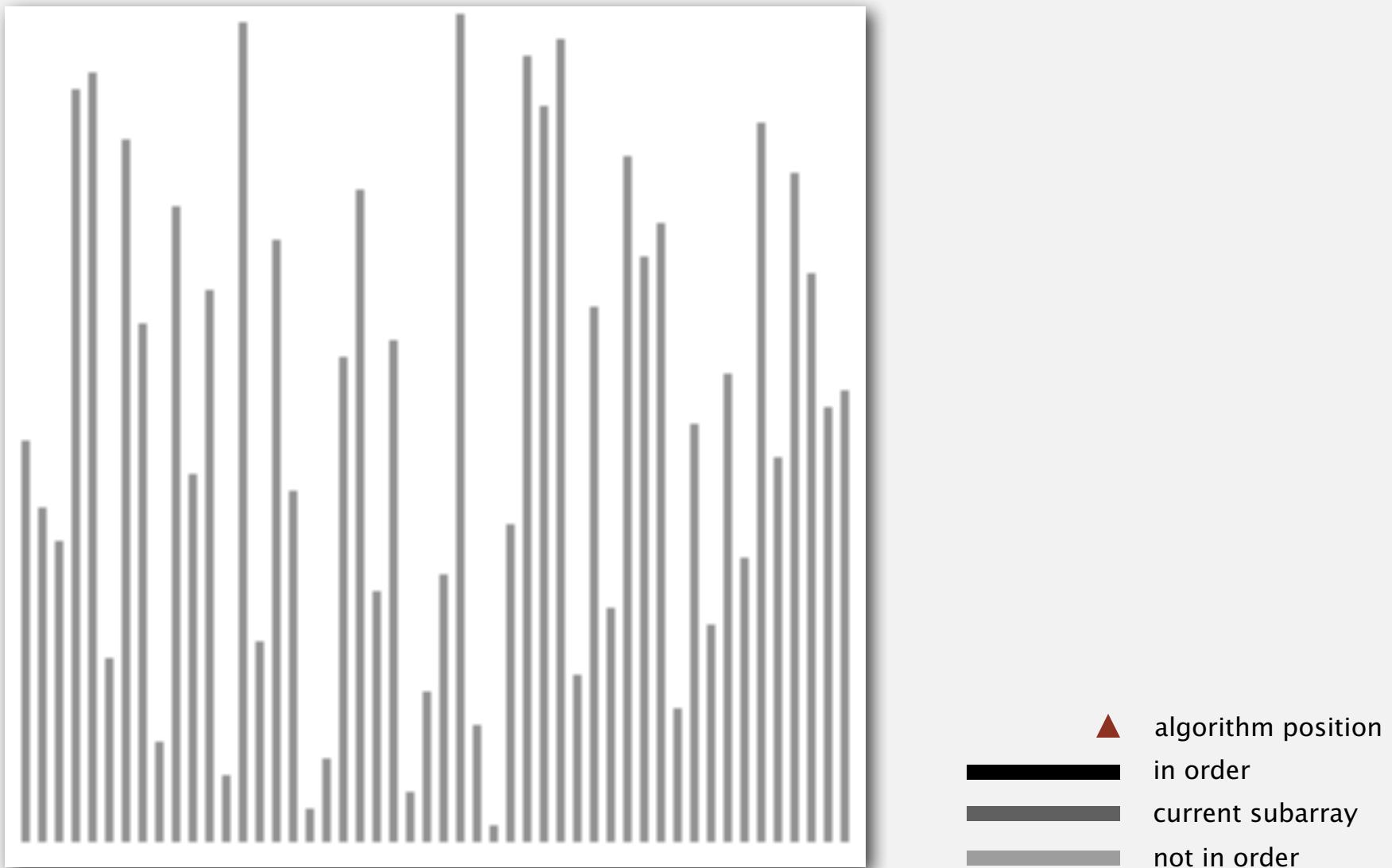
# Quicksort trace

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	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	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S	
4	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	M	O	P	T	Q	R	X	U	S
8	8	8	A	C	E	E	I	K	L	M	M	O	P	T	Q	R	X	U	S
10	13	15	A	C	E	E	I	K	L	M	M	O	P	S	Q	R	T	U	X
10	12	12	A	C	E	E	I	K	L	M	M	O	P	R	Q	S	T	U	X
10	11	11	A	C	E	E	I	K	L	M	M	O	P	Q	R	S	T	U	X
10	10	10	A	C	E	E	I	K	L	M	M	O	P	Q	R	S	T	U	X
14	14	15	A	C	E	E	I	K	L	M	M	O	P	Q	R	S	T	U	X
15	15	15	A	C	E	E	I	K	L	M	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 a bit trickier than it might seem.

**Staying in bounds.** The  $(j == lo)$  test is redundant (why?), but the  $(i == hi)$  test is not.

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

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

# 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  $\sim N \lg N$ .

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
			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
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  $N \geq 2$ :

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

- 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 this from 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

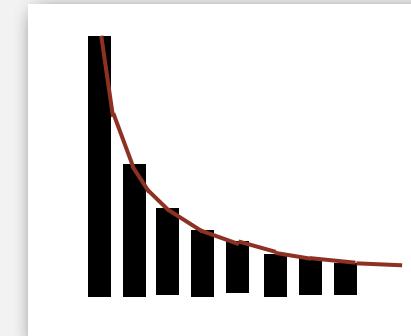
- Repeatedly apply above equation:

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

← previous equation

- 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

---

**Worst case.** Number of compares is quadratic.

- $N + (N - 1) + (N - 2) + \dots + 1 \sim \frac{1}{2} N^2$ .
- More likely that your computer is struck by lightning bolt.

**Average case.** Number of compares is  $\sim 1.39 N \lg N$ .

- 39% more compares than mergesort.
- **But** faster than mergesort in practice because of less data movement.

**Random shuffle.**

- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

**Caveat emptor.** Many textbook implementations go **quadratic** if array

- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!)

## 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

Proposition. Quicksort is **not stable**.

Pf.

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.
- Note: could delay insertion sort until one pass at end.

```
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 (slightly fewer)  
~  $12/35 N \ln N$  exchanges (slightly more)

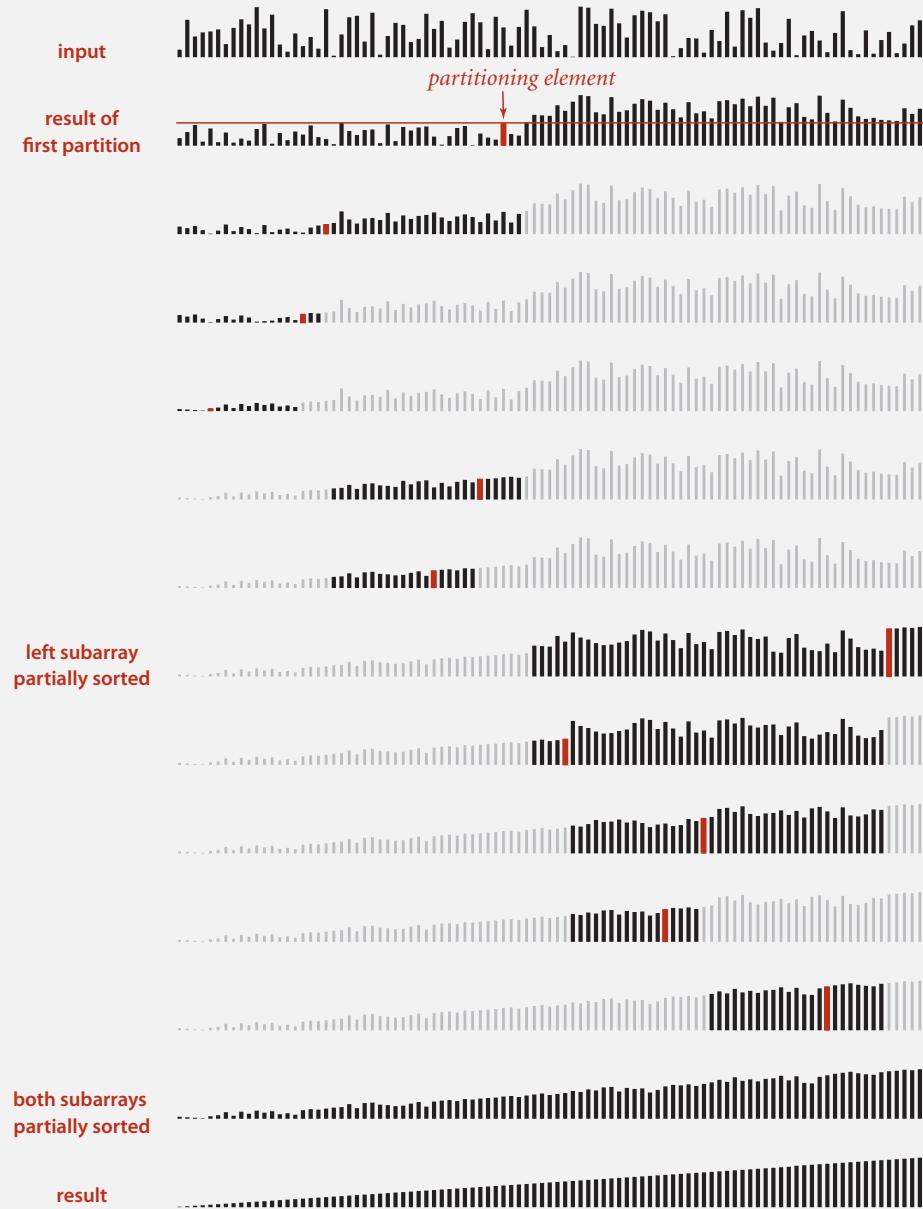
```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;

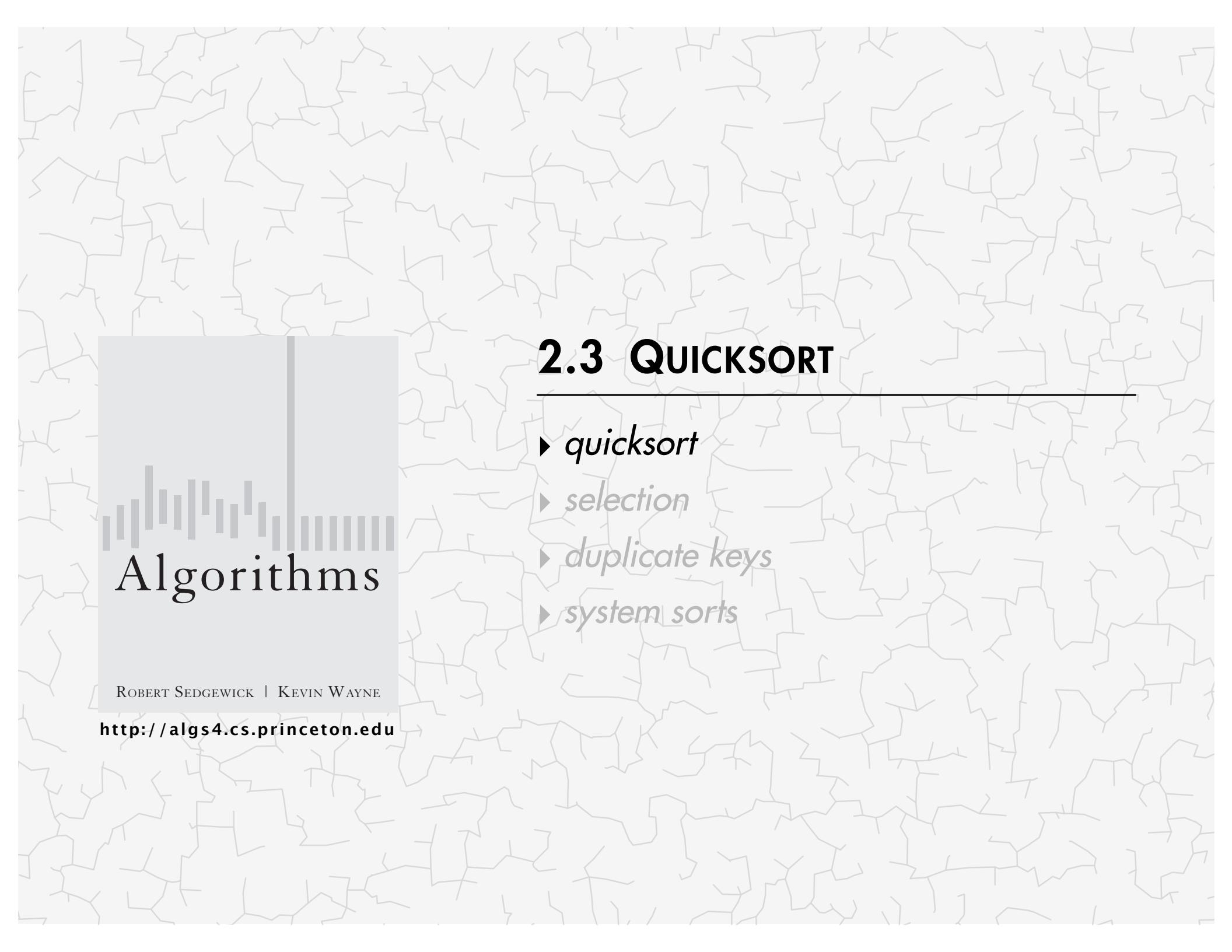
    int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, m);

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

# Quicksort with median-of-3 and cutoff to insertion sort: visualization

---





# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## 2.3 QUICKSORT

---

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

# 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 a  $k^{\text{th}}$  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 for each  $k$ ?

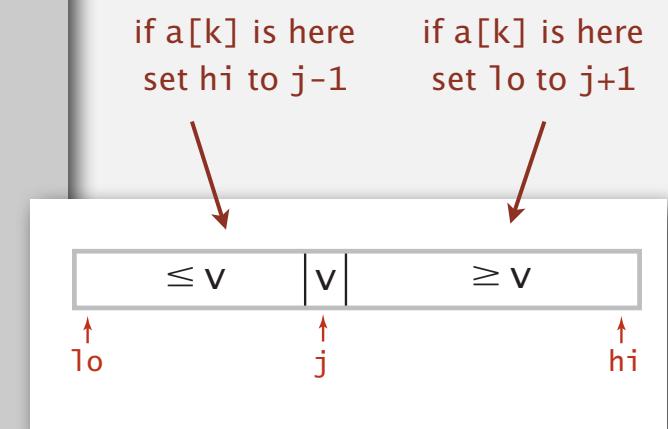
# Quick-select

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: 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:

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

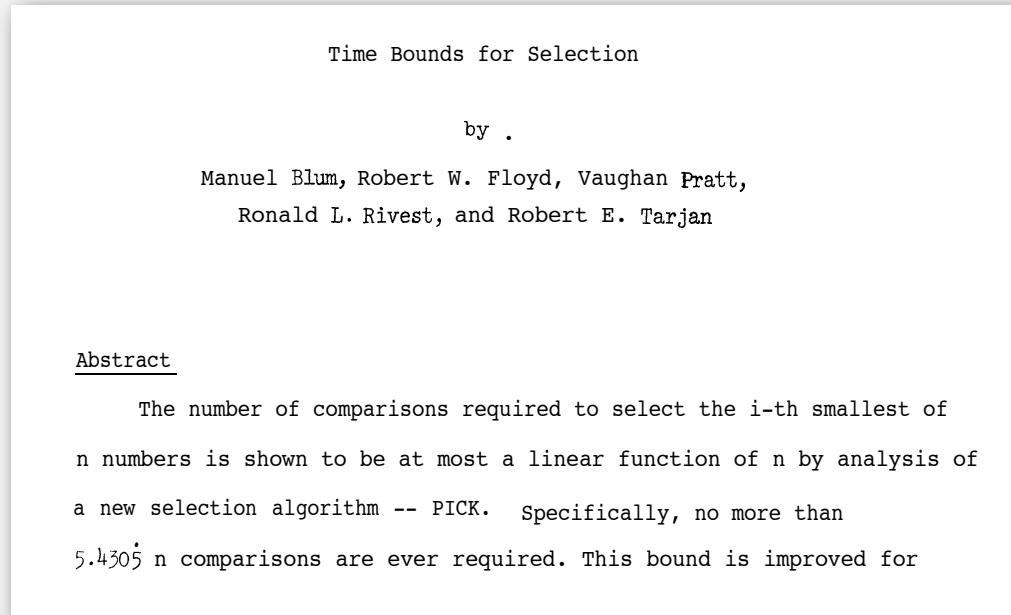
  
 $(2 + 2 \ln 2)N$  to find the median

Remark. Quick-select uses  $\sim \frac{1}{2}N^2$  compares in the worst case, but (as with quicksort) the random shuffle provides a probabilistic guarantee.

# Theoretical context for selection

---

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



**Remark.** But, constants are too 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*

# 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

## Duplicate keys

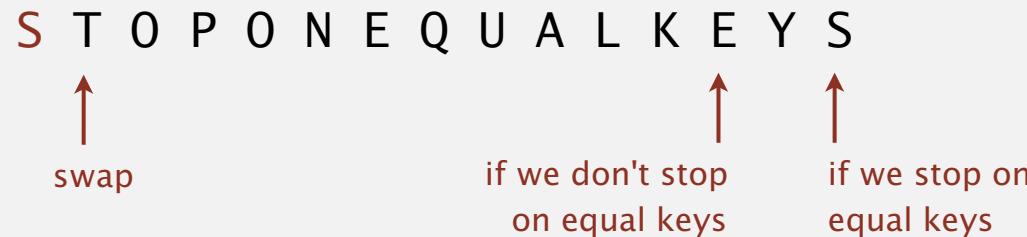
---

Mergesort with duplicate keys. Between  $\frac{1}{2}N\lg N$  and  $N\lg N$  compares.

### Quicksort with duplicate keys.

- Algorithm goes **quadratic** unless partitioning stops on equal keys!
- 1990s C user found this defect in `qsort()`.

several textbook and system  
implementation also have this defect



## Duplicate keys: the problem

---

Mistake. Put all items equal to the partitioning item on one side.

Consequence.  $\sim \frac{1}{2} N^2$  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**

Recommended. Stop scans on items equal to the partitioning item.

Consequence.  $\sim N \lg N$  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

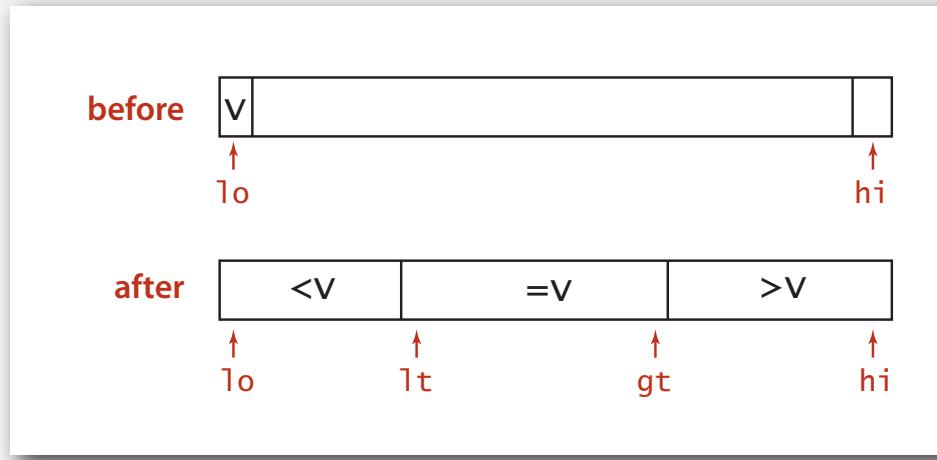
Desirable. Put all items equal to the partitioning item in place.

A A A **B** B B B C C C      **A** A A A A A A A A A A A A

## 3-way partitioning

**Goal.** Partition array into 3 parts so that:

- Entries between  $lt$  and  $gt$  equal to partition item  $v$ .
- No larger entries to left of  $lt$ .
- No smaller entries to right of  $gt$ .



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

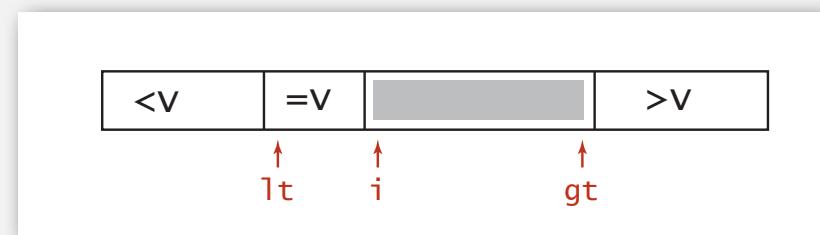
- Conventional wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in C library `qsort()`.
- Now incorporated into `qsort()` and Java system `sort`.

# 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  $i$

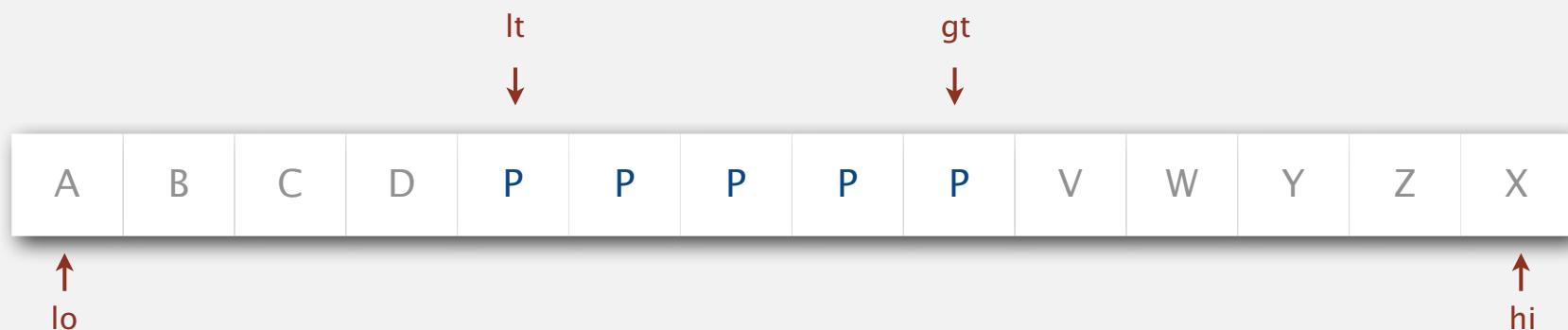


invariant

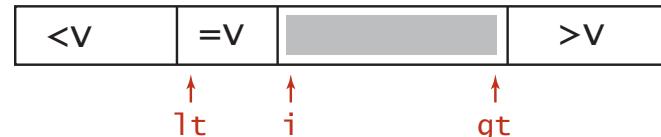


# Dijkstra 3-way partitioning demo

- 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



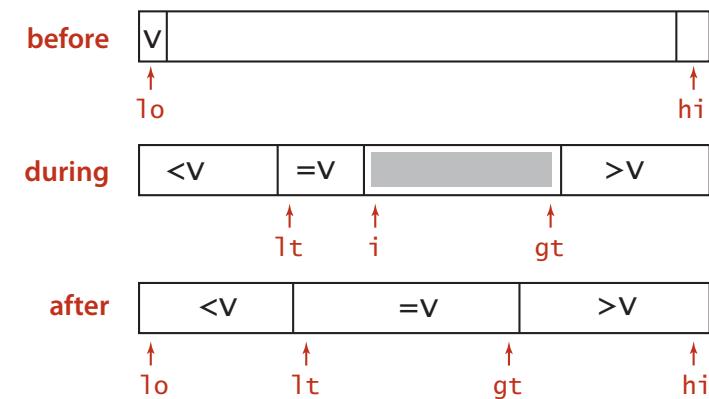
# Dijkstra's 3-way partitioning: trace

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

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

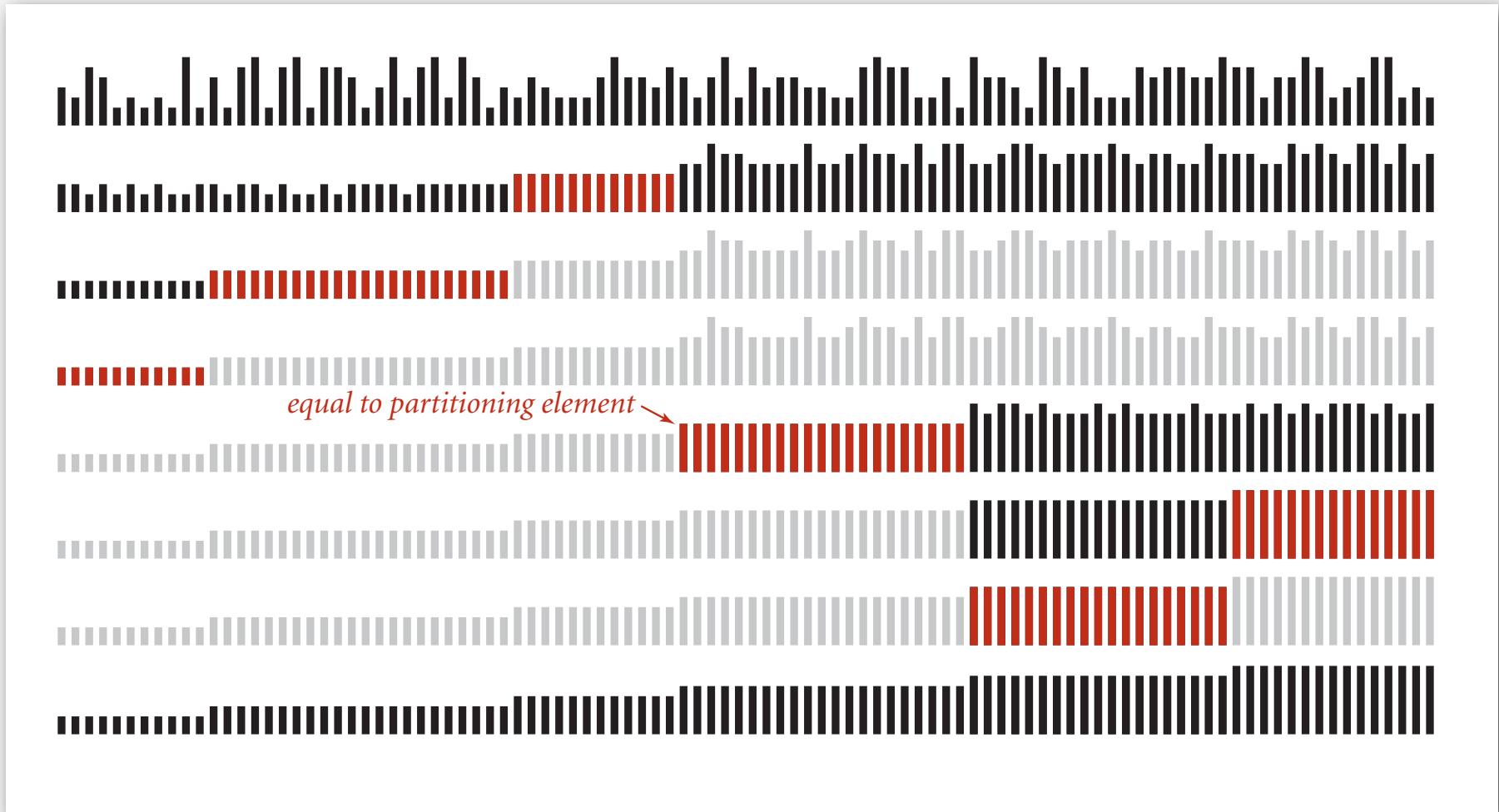
## 3-way quicksort: Java implementation

```
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  $i^{\text{th}}$  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.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

## 2.3 QUICKSORT

---

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

# 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. obvious applications
- Display Google PageRank results.
- List RSS feed in reverse chronological order.
  
- Find the median.
- Identify statistical outliers. problems become easy once items are in sorted order
- Binary search in a database.
- Find duplicates in a mailing list.
  
- Data compression.
- Computer graphics. non-obvious applications
- Computational biology.
- Load balancing on a parallel computer.
  
- . . .

## Java system sorts

---

### [Arrays.sort\(\)](#).

- Has different method for each primitive type.
- Has a method for data types that implement Comparable.
- Has a method that uses a Comparator.
- Uses tuned quicksort for primitive types; tuned mergesort for objects.

```
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]);
    }
}
```

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

## 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.



# Engineering a system sort

---

Basic algorithm = quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning.
- Partitioning item.
  - small arrays: middle entry
  - medium arrays: median of 3
  - large arrays: Tukey's ninther [next slide]

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.

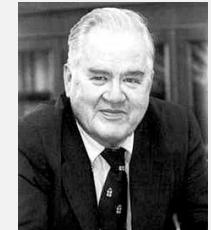
Now widely used. C, C++, Java 6, ....

# Tukey's ninther

---

Tukey's ninther. Median of the median of 3 samples, each of 3 entries.

- Approximates the median of 9.
- Uses at most 12 compares.



Q. Why use Tukey's ninther?

A. Better partitioning than random shuffle and less costly.

# Achilles heel in Bentley-McIlroy implementation (Java system sort)

Q. Based on all this research, Java's system sort is solid, **right?**

A. No: a killer input.

- Overflows function call stack in Java and crashes program.
- Would take quadratic time if it didn't crash first.

```
% more 250000.txt
```

```
0
```

```
218750
```

```
222662
```

```
11
```

```
166672
```

```
247070
```

```
83339
```

```
...
```



250,000 integers  
between 0 and 250,000

```
% java IntegerSort 250000 < 250000.txt
```

```
Exception in thread "main"
```

```
java.lang.StackOverflowError
```

```
at java.util.Arrays.sort1(Arrays.java:562)
```

```
at java.util.Arrays.sort1(Arrays.java:606)
```

```
at java.util.Arrays.sort1(Arrays.java:608)
```

```
at java.util.Arrays.sort1(Arrays.java:608)
```

```
at java.util.Arrays.sort1(Arrays.java:608)
```

```
...
```

Java's sorting library crashes, even if  
you give it as much stack space as Windows allows

# 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.

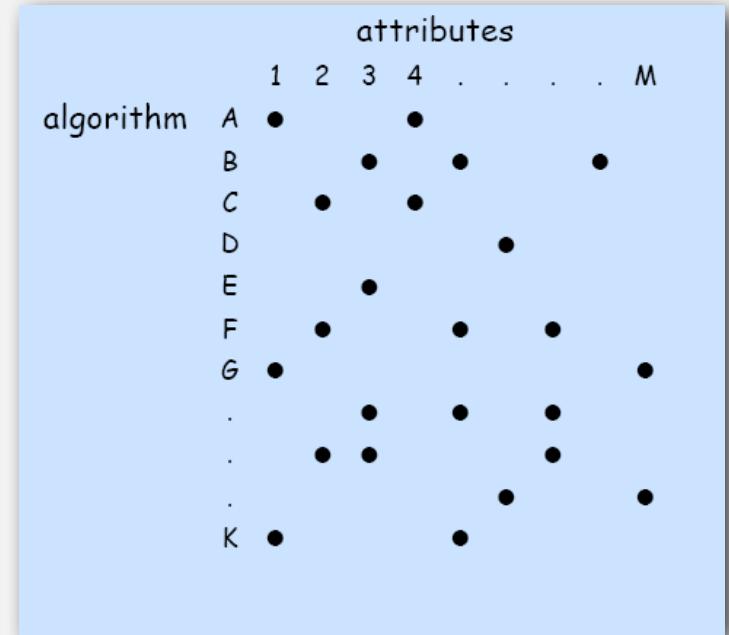
## Parallel sorts.

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

# System sort: Which algorithm to use?

Applications have diverse attributes.

- Stable?
- Parallel?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Is your array randomly ordered?
- Need guaranteed performance?



many more combinations of attributes than algorithms

Elementary sort may be method of choice for some combination.

Cannot cover **all** combinations of attributes.

Q. Is the system sort good enough?

A. Usually.

# Sorting summary

---

	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	✓		$N^2 / 2$	$2N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	✓		$N^2 / 2$	$2N \ln N$	$N$	improves quicksort in presence of duplicate keys
???	✓	✓	$N \lg N$	$N \lg N$	$N$	holy sorting grail

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

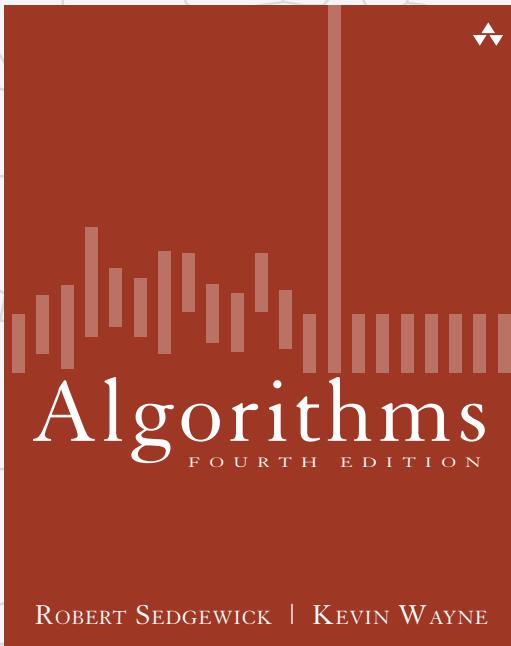
## 2.3 QUICKSORT

---

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

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



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

## 2.3 QUICKSORT

---

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