



<https://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 20th century in science and engineering.

Mergesort. [last lecture]



Quicksort. [this lecture]



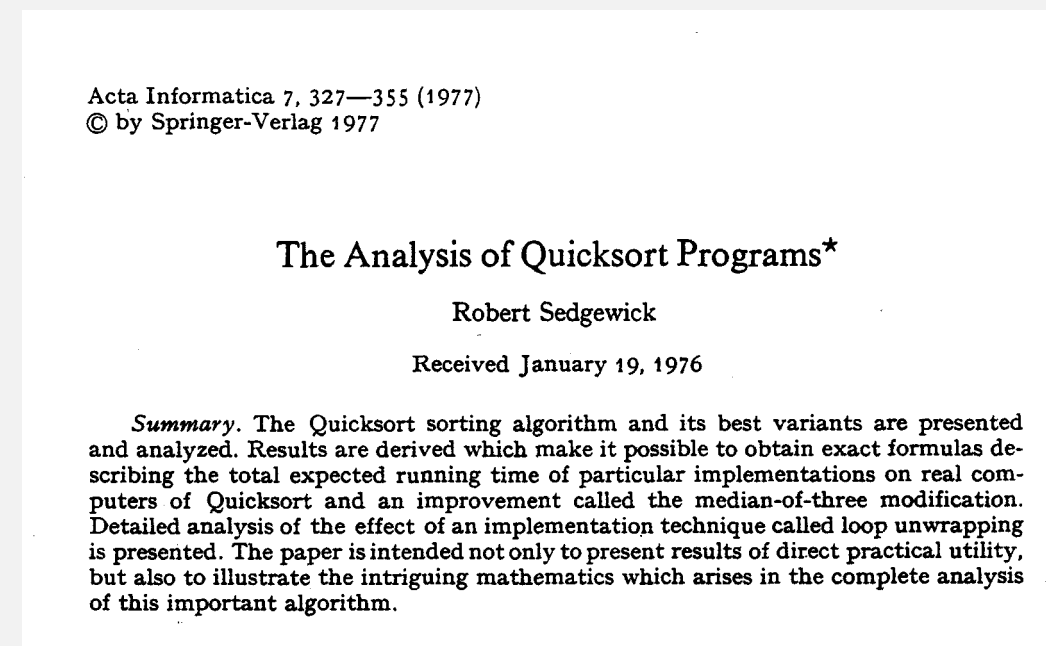
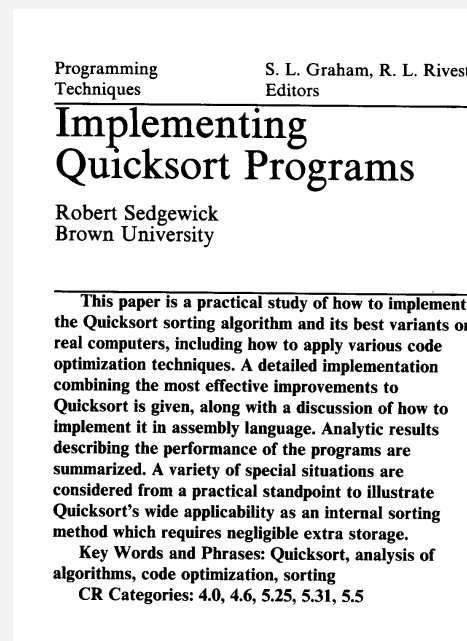
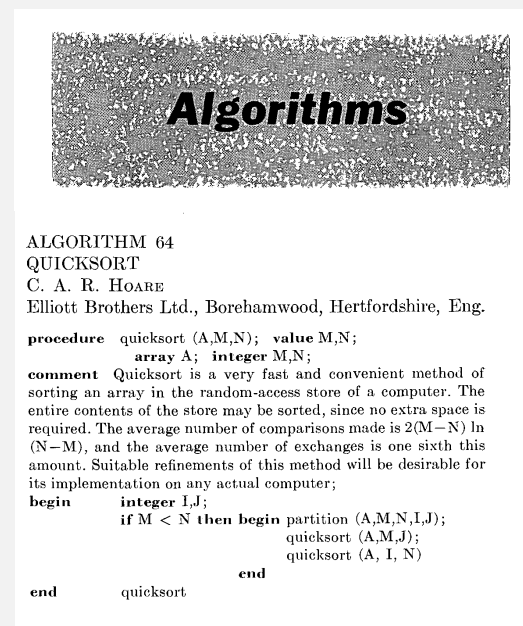
A brief history

Tony Hoare.

- Invented quicksort in 1960 to translate Russian into English.
- Learned Algol 60 (and recursion) to implement it.



Tony Hoare
1980 Turing Award



Bob Sedgewick.

- Refined and popularized quicksort in 1970s.
- Analyzed many versions of quicksort.



Bob Sedgewick



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

2.3 QUICKSORT


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

Quicksort overview



Step 1. Shuffle the array.

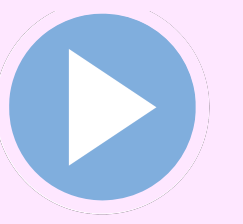
Step 2. Partition the array so that, for some index j :

- Entry $a[j]$ is in place.  “pivot” or “partitioning item”
- No larger entry to the left of j .
- No smaller entry to the right of j .

Step 3. Sort each subarray recursively.

input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	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

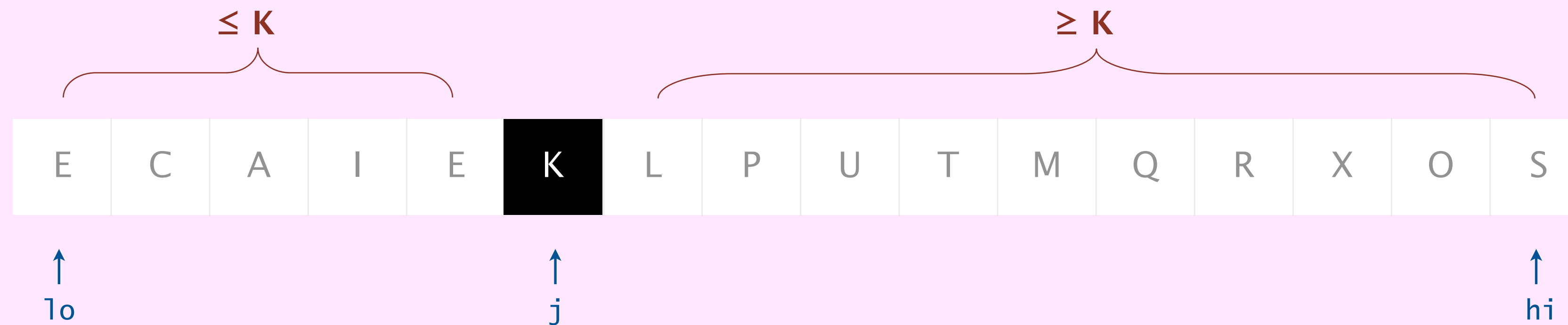
Quicksort partitioning demo



Repeat until 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 partitioning: Java implementation

```
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

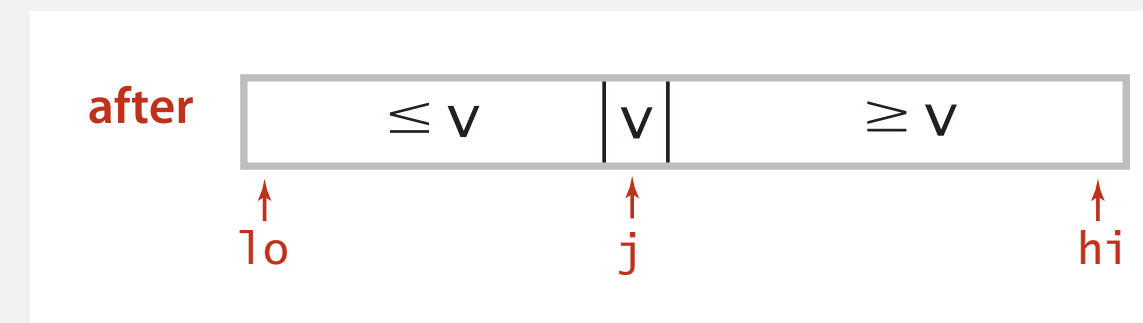
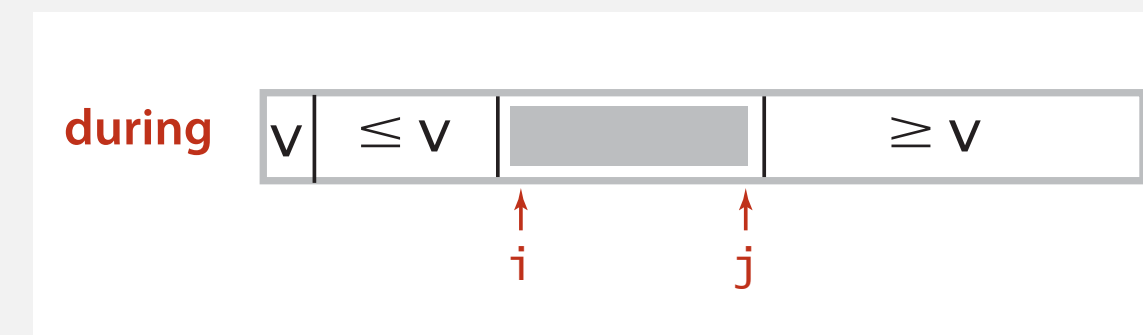
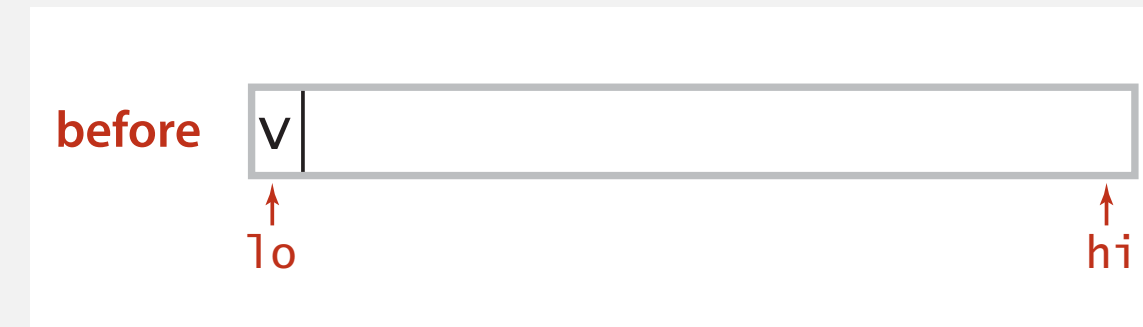
check if pointers cross

swap

swap with pivot

return index of item now known to be in place

<https://algs4.cs.princeton.edu/23quick/Quick.java.html>





In the worst case, how many compares and exchanges does `partition()` make to partition a subarray of length n ?

- A. $\sim \frac{1}{2} n$ and $\sim \frac{1}{2} n$
- B. $\sim \frac{1}{2} n$ and $\sim n$
- C. $\sim n$ and $\sim \frac{1}{2} n$
- D. $\sim n$ and $\sim n$

M	A	B	C	D	E	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10

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); ← shuffle needed for
        sort(a, 0, a.length - 1); performance guarantee
                                (stay tuned)
    }

    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);
    }
}
```

<https://algs4.cs.princeton.edu/23quick/Quick.java.html>

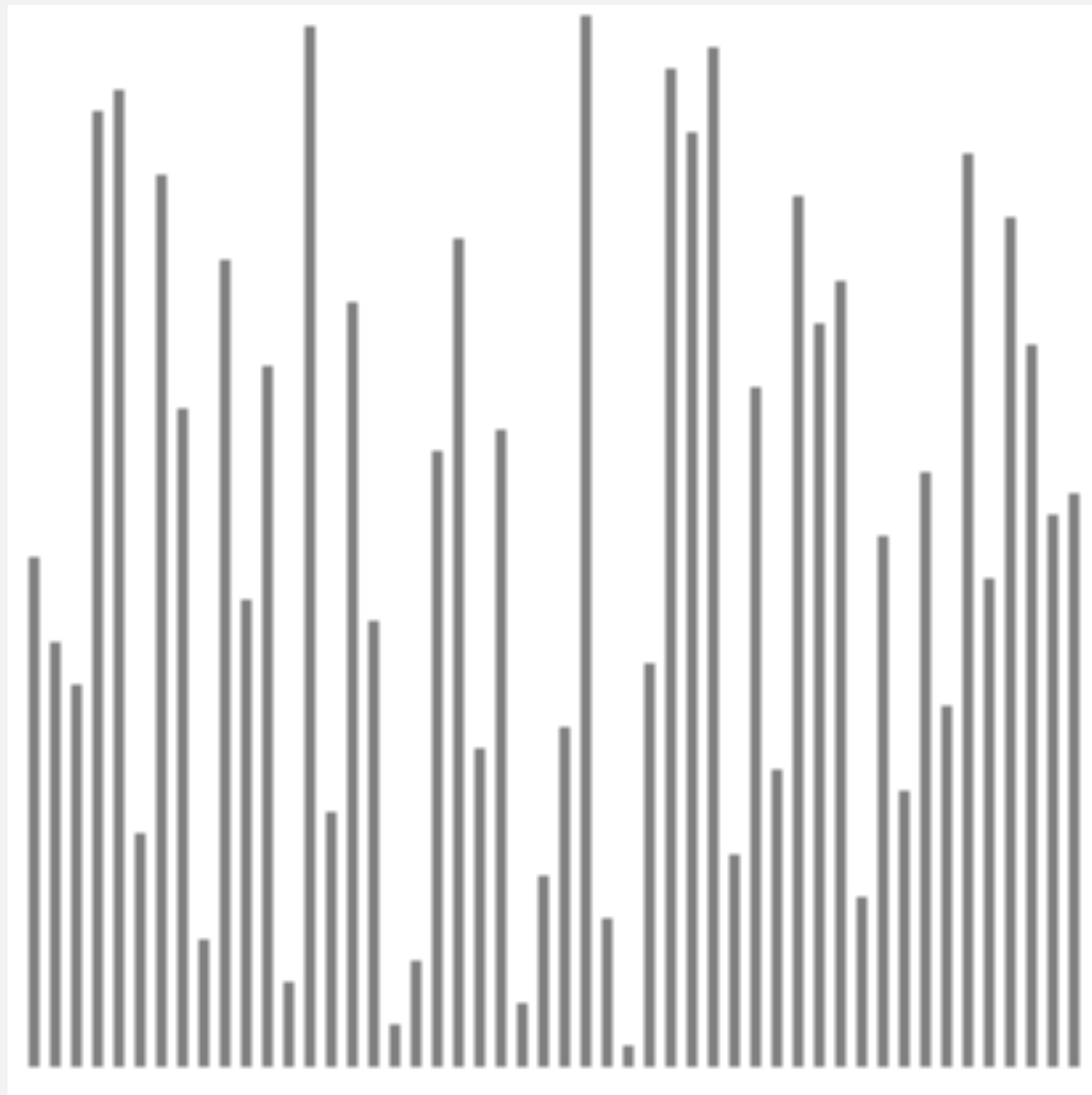
Quicksort trace

			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	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
			4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
			6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
			7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
			7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
			8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
			10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
			10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
			10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
			10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
			14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
			15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result						A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort trace (array contents after each partition)

Quicksort animation

50 random items



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

- ▲ algorithm position
- in order
- current subarray
- not in order

Quicksort: implementation details

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

Loop termination. Terminating the loop is more subtle than it appears.

Equal keys. Handling duplicate keys is trickier than it appears. [stay tuned]

Preserving randomness. Shuffling is needed for performance guarantee.

Equivalent alternative. Pick a random pivot in each subarray.



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.



Why is quicksort typically faster than mergesort in practice?

- A. Fewer compares.
- B. Fewer array accesses.
- C. Both A and B.
- D. Neither A nor B.

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

after random shuffle

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

← after random shuffle

Good news. Worst case for randomized quicksort is mostly irrelevant in practice.

- Exponentially small chance of occurring.
(unless bug in shuffling or no shuffling)
- More likely that computer is struck by lightning bolt during execution.



Quicksort: probabilistic analysis

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

Recall. Any algorithm with the following structure takes $\Theta(n \log n)$ time.

```
public static void f(int n)
{
    if (n == 0) return;
    f(n/2);
    f(n/2);
    linear(n);
}
```

← solve two problems of half the size

← do $\Theta(n)$ work

Intuition. Each partitioning step divides the problem into two subproblems, each of approximately one-half the size.

↑
probabilistically “close enough”

Quicksort: probabilistic analysis

Proposition. The expected 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 $n \geq 2$:

$$C_n = \overset{\text{partitioning}}{\downarrow} (n+1) + \left(\frac{C_0 + C_{n-1}}{n} \right) + \overset{\text{left}}{\downarrow} \left(\frac{C_1 + C_{n-2}}{n} \right) + \overset{\text{right}}{\downarrow} \dots + \left(\frac{C_{n-1} + C_0}{n} \right)$$

↙ partitioning probability

- Multiply both sides by n and collect terms:

$$n C_n = n(n+1) + 2(C_0 + C_1 + \dots + C_{n-1})$$

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

$$n C_n - (n-1) C_{n-1} = 2n + 2 C_{n-1}$$

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

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1}$$

analysis beyond
scope of this course

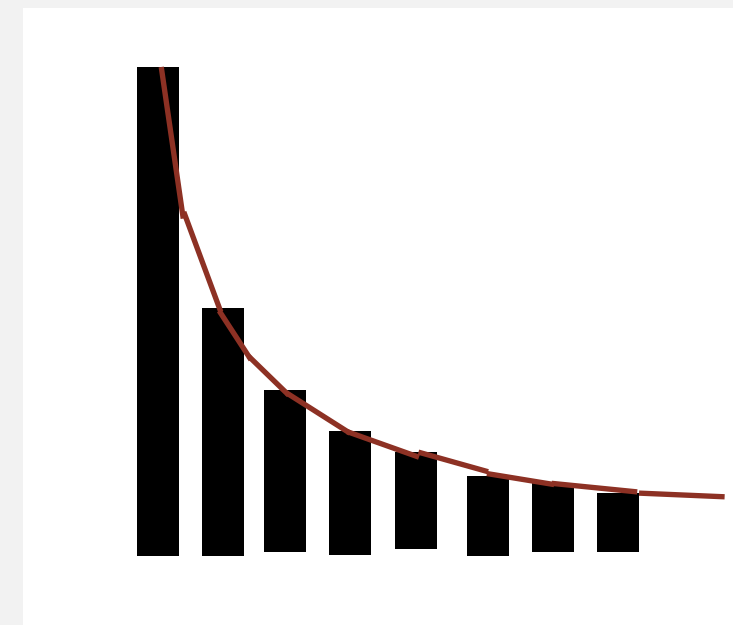
Quicksort: probabilistic analysis

- Repeatedly apply previous equation:

$$\begin{aligned}\frac{C_n}{n+1} &= \frac{C_{n-1}}{n} + \frac{2}{n+1} \\ &= \frac{C_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1} &< \text{substitute previous equation} \\ &= \frac{C_{n-3}}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \dots + \frac{2}{n+1}\end{aligned}$$

- 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:

$$C_n \sim 2(n+1) \ln n \approx 1.39 n \lg n$$

Quicksort properties

Quicksort analysis summary.

- Expected number of compares is $\sim 1.39 n \log_2 n$.
[standard deviation is $\sim 0.65 n$]
39% more than mergesort
- Expected number of exchanges is $\sim 0.23 n \log_2 n$. ← much less than mergesort
- Min number of compares is $\sim n \log_2 n$. ← never less than mergesort
- Max number of compares is $\sim \frac{1}{2} n^2$. ← but never happens

Context. Quicksort is a (Las Vegas) **randomized algorithm**.

- Guaranteed to be correct.
- Running time depends on outcomes of random coin flips (shuffle).



Quicksort properties

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

- Partitioning: $\Theta(1)$ extra space.
- Function-call stack: $\Theta(\log n)$ extra space (with high probability).

can guarantee $\Theta(\log n)$ depth by recurring on smaller subarray before larger subarray (but this 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 ≈ 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.

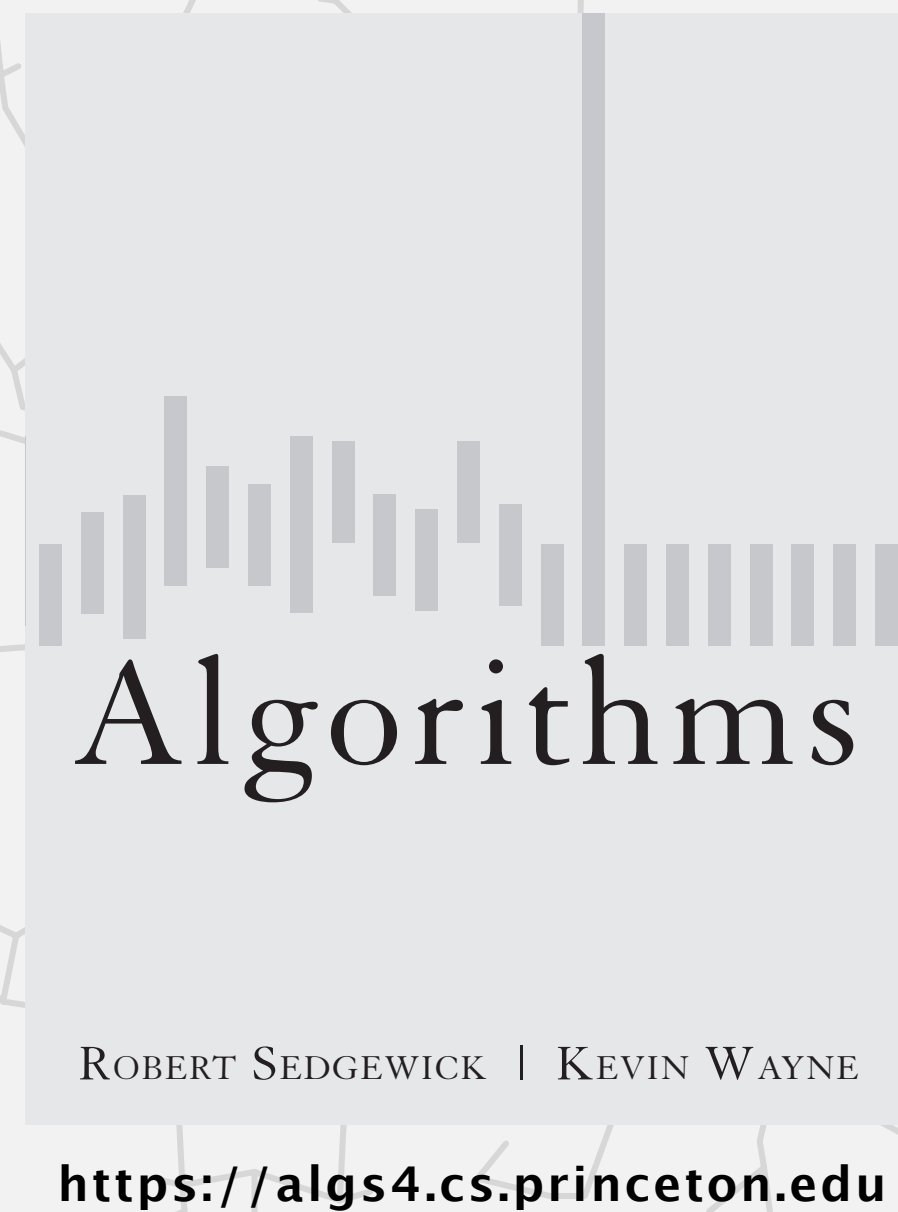
- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.

~ $12/7 \ n \ln n$ 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) >>> 1, hi);
    swap(a, lo, median);

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



2.3 QUICKSORT

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

Selection

Goal. Given an array of n items, find item of **rank** k .

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

Applications.

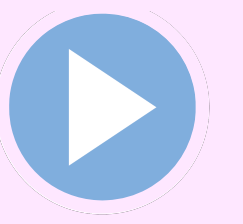
- Order statistics.
- Find the “top k .”

Use complexity theory as a guide.

- Easy $O(n \log n)$ algorithm. How?
- Easy $O(n)$ algorithm for $k = 0$ or 1 . How?
- Easy $\Omega(n)$ lower bound. Why?

Which is true?

- $O(n)$ algorithm? [is there a linear-time algorithm?]
- $\Omega(n \log n)$ lower bound? [is selection as hard as sorting?]



Partition 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 .

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

select element of rank $k = 5$

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

$k = 5$

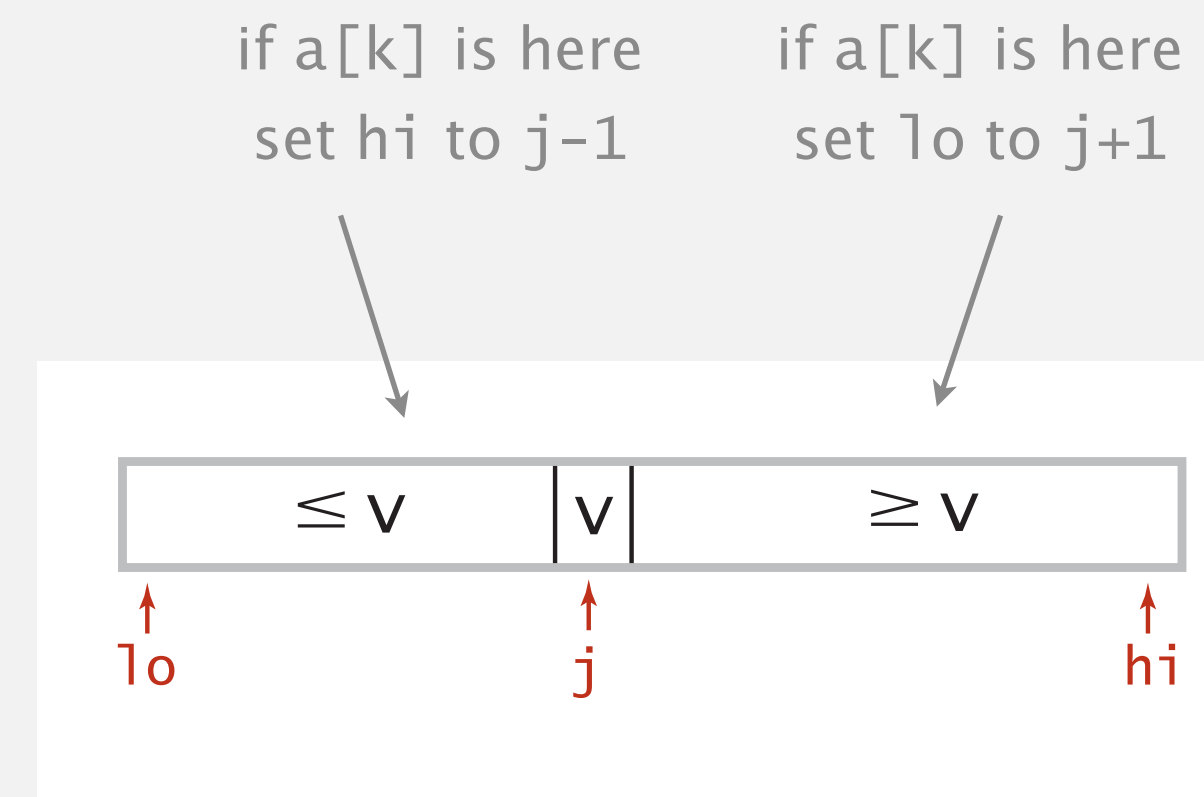
Quickselect

Partition 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 .

Repeat in **one** subarray, depending on j ; stop 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];
}
```



Quickselect: probabilistic analysis

Proposition. The expected number of compares C_n to quickselect the item of rank k in an array of length n is $\Theta(n)$.

← probabilistically “close enough”

Intuition. Each partitioning step approximately halves the length of the array.

Recall. Any algorithm with the following structure takes $\Theta(n)$ time.

```
public static void f(int n)
{
    if (n == 0) return;
    linear(n);      ← do  $\Theta(n)$  work
    f(n/2);        ← solve one subproblem of half the size
}
```

$$n + n/2 + n/4 + \dots + 1 \sim 2n$$

Careful analysis yields:

$$\begin{aligned} C_n &\sim 2n + 2k \ln(n/k) + 2(n-k) \ln(n/(n-k)) \\ &\leq (2 + 2 \ln 2) n \\ &\approx 3.38 n \end{aligned} \quad \left| \begin{array}{l} \leftarrow \text{max occurs for median } (k = n/2) \end{array} \right.$$

Theoretical context for selection

Q. Compare-based selection algorithm that makes $\Theta(n)$ compares in the **worst case**?

A. Yes! [ingenious divide-and-conquer]

Time Bounds for Selection*

MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT,
RONALD L. RIVEST, AND ROBERT E. TARJAN

Department of Computer Science, Stanford University, Stanford, California 94305

Received November 14, 1972

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 $5.4305n$ comparisons are ever required. This bound is improved for extreme values of i , and a new lower bound on the requisite number of comparisons is also proved.

$$T(n) = T(n/5) + T(7n/10) + \Theta(n)$$

↑ ↑
find pivot that eliminates
 30% of items

Caveat. Constants are high \Rightarrow not used in practice.

Use theory as a guide.

- Open problem: **practical** algorithm that makes $\Theta(n)$ compares in the worst case.
- Until one is discovered, use quickselect (if you don't need a full sort).



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

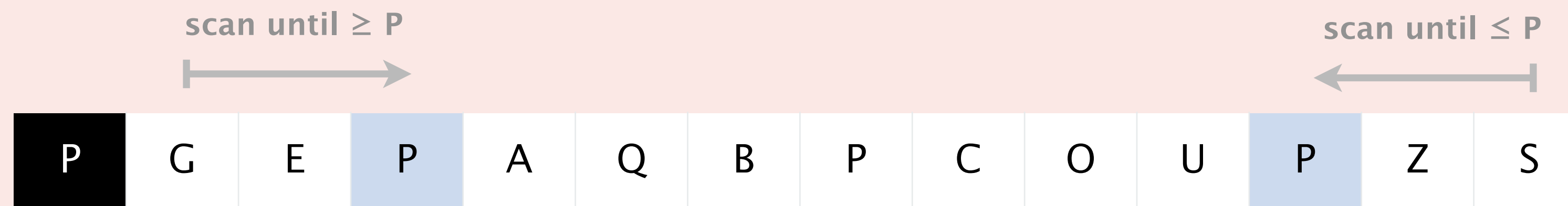


When partitioning, how to handle keys equal to pivot?

A.



B.



C. Either A or B.

Duplicate keys: partitioning strategies

Bad. Don't stop scans on equal keys.

[$\Theta(n^2)$ 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**

Good. Stop scans on equal keys.

[$\sim n \log_2 n$ 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

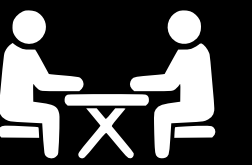
Better. Put all equal keys in place. How?

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

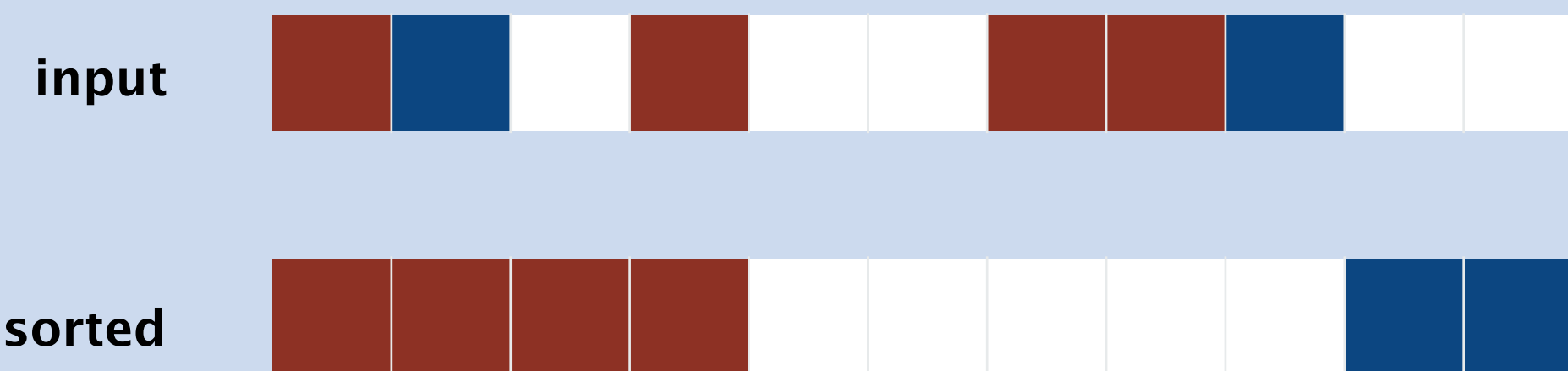
A A A **B B B B B** C C C

A A A A A A A A A A A

DUTCH NATIONAL FLAG PROBLEM



Problem. [Edsger Dijkstra] Given an array of n buckets, each containing a red, white, or blue pebble, sort them by color.



Operations allowed.

- $swap(i, j)$: swap the pebble in bucket i with the pebble in bucket j .
- $getColor(i)$: determine the color of the pebble in bucket i .

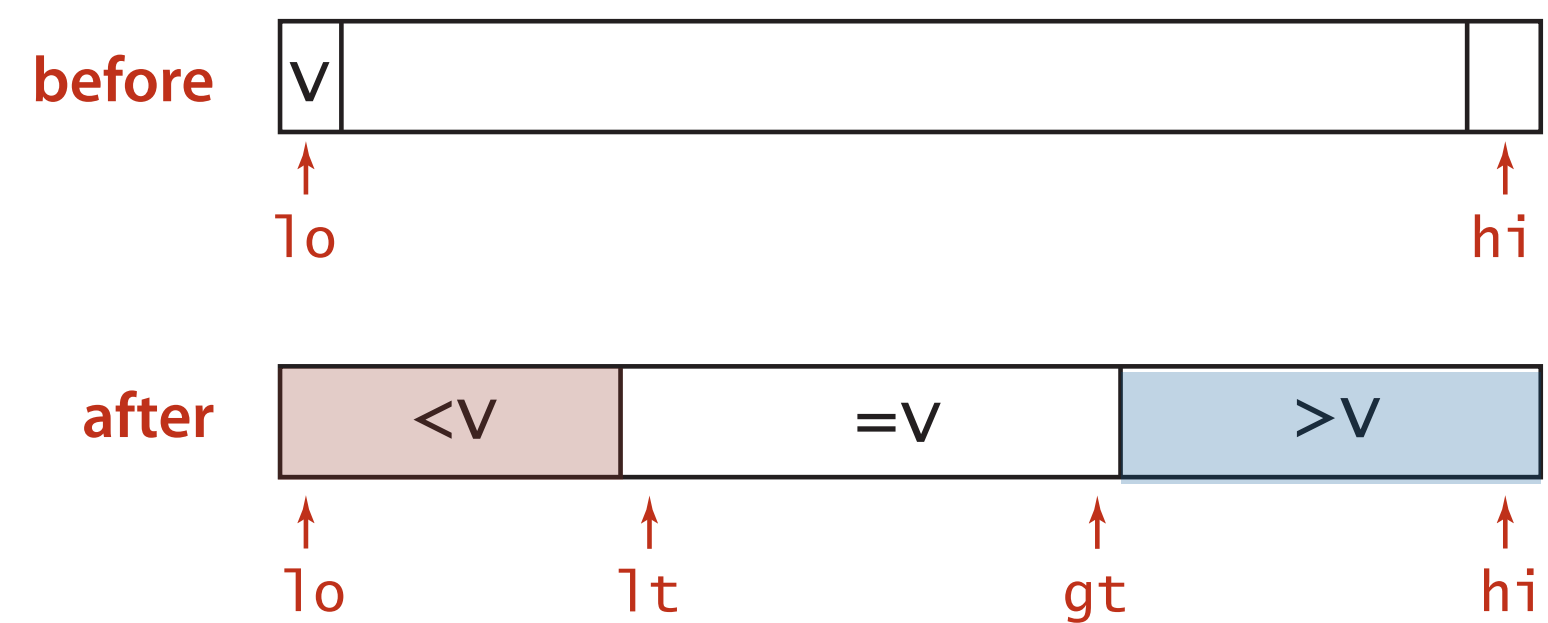
Performance requirements.

- Exactly n calls to $getColor()$.
- At most n calls to $swap()$.
- $\Theta(1)$ extra space.

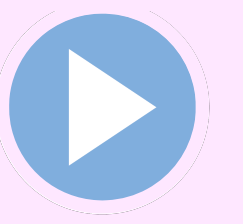
3-way partitioning

Goal. Use pivot $v = a[l_o]$ to partition array into **three** parts so that:

- Red: smaller entries to the left of l_t .
- White: equal entries between l_t and gt .
- Blue: larger entries to the right of gt .



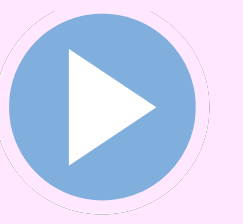
Dijkstra's 3-way partitioning algorithm: demo



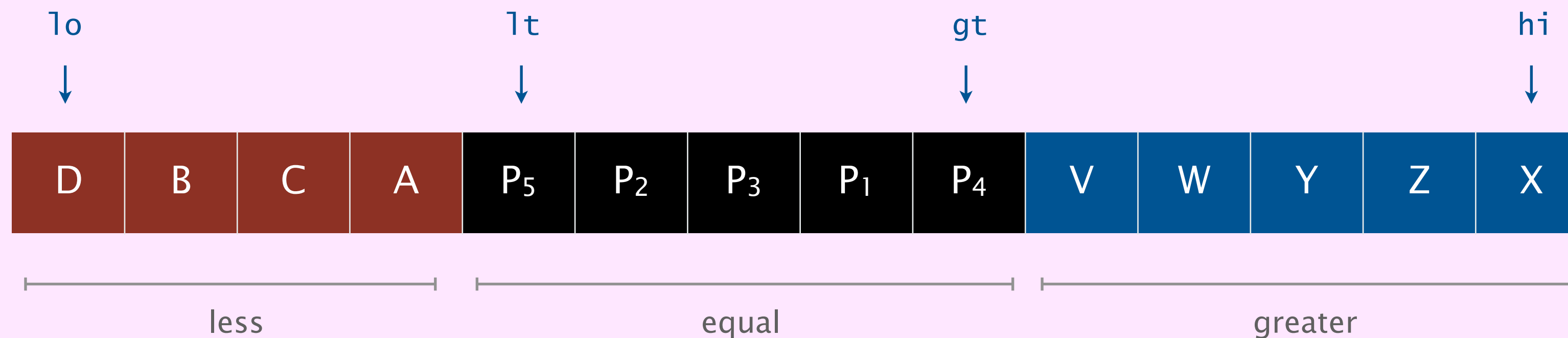
- Let $v = a[l_o]$ be pivot.
- Scan i from left to right and compare $a[i]$ to v .
 - less: exchange $a[i]$ with $a[l_t]$; increment both l_t and i
 - greater: exchange $a[i]$ with $a[gt]$; decrement gt
 - equal: increment i



Dijkstra's 3-way partitioning algorithm: demo



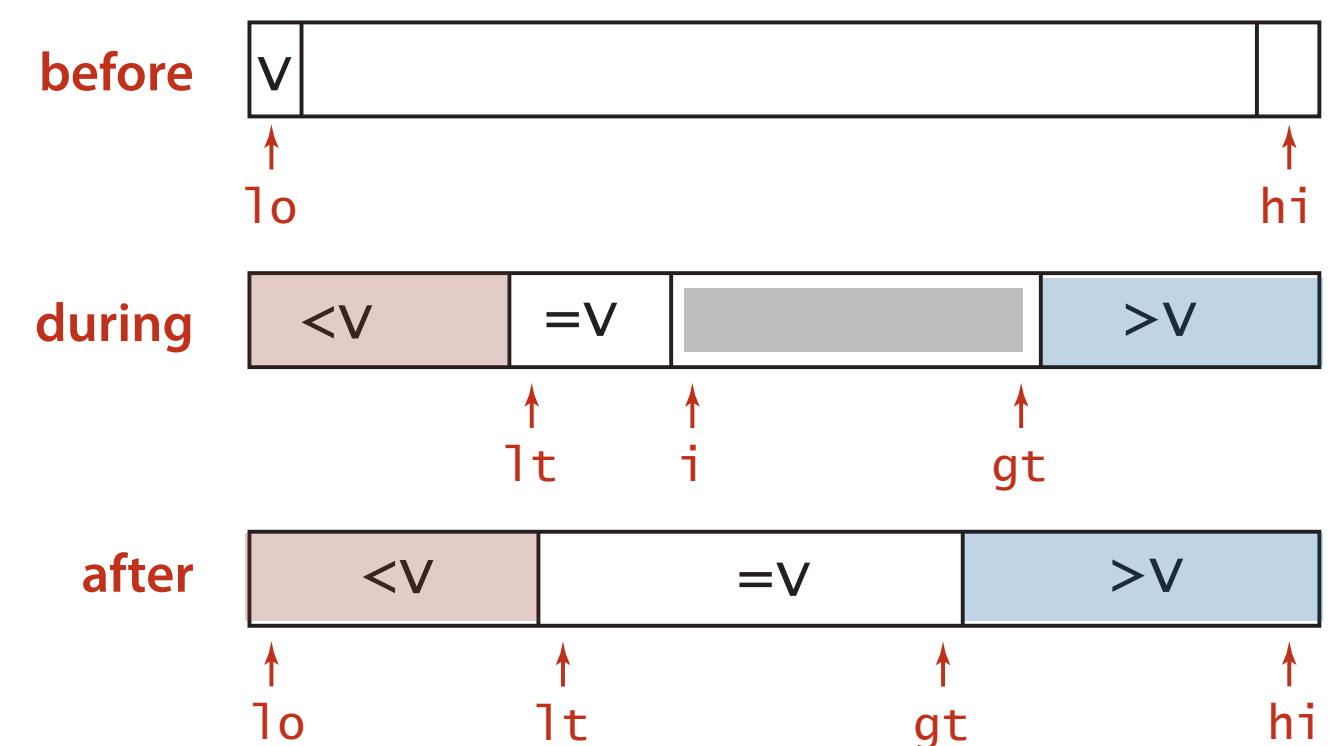
- Let $v = a[l_o]$ be pivot.
- Scan i from left to right and compare $a[i]$ to v .
 - less: exchange $a[i]$ with $a[l_t]$; increment both l_t and i
 - greater: exchange $a[i]$ with $a[gt]$; decrement gt
 - equal: increment i

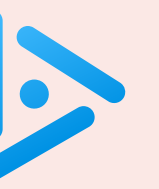


3-way quicksort: Java implementation

```
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 + 1;
    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++;
    }

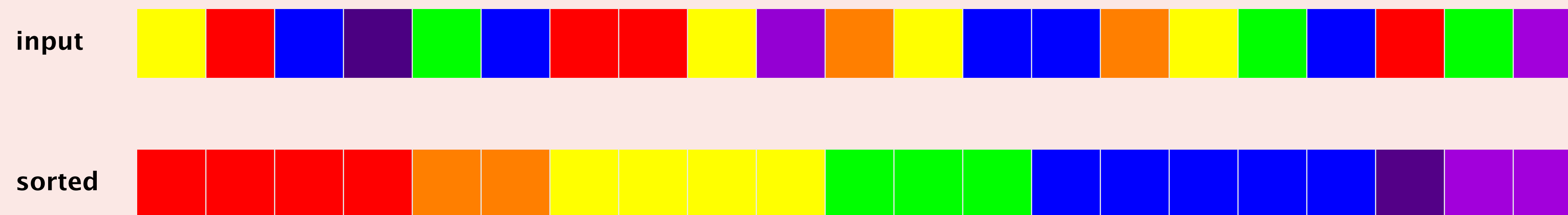
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
```





What is the worst-case number of compares to 3-way quicksort an array of length n containing only 7 distinct values?

- A. $\Theta(n)$
- B. $\Theta(n \log n)$
- C. $\Theta(n^2)$
- D. $\Theta(n^7)$



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 sorted arrays
merge		✓	$\frac{1}{2} n \log_2 n$	$n \log_2 n$	$n \log_2 n$	$\Theta(n \log n)$ guarantee; stable
timsort		✓	n	$n \log_2 n$	$n \log_2 n$	improves mergesort when pre-existing order
quick	✓		$n \log_2 n$	$2 n \ln n$	$\frac{1}{2} n^2$	$\Theta(n \log n)$ probabilistic guarantee; fastest in practice
3-way quick	✓		n	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
?	✓	✓	n	$n \log_2 n$	$n \log_2 n$	holy sorting grail

number of compares to sort an array of n elements



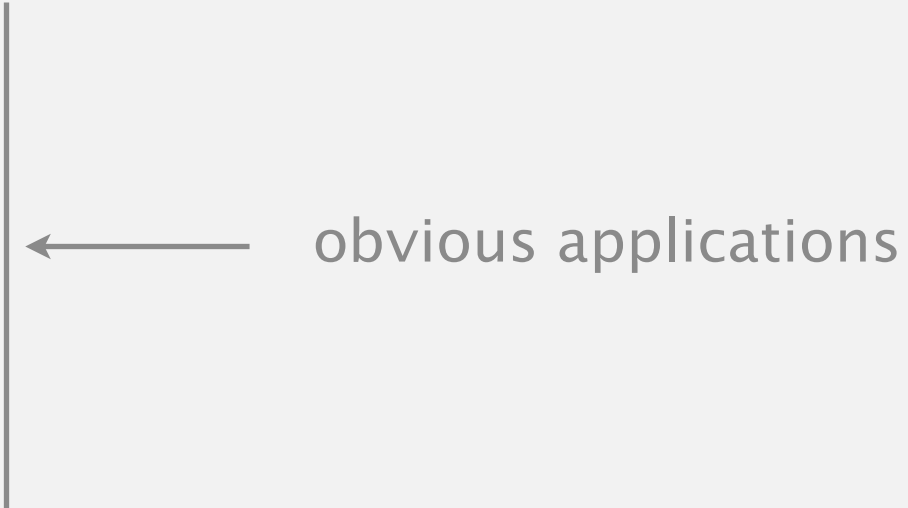
<https://algs4.cs.princeton.edu>

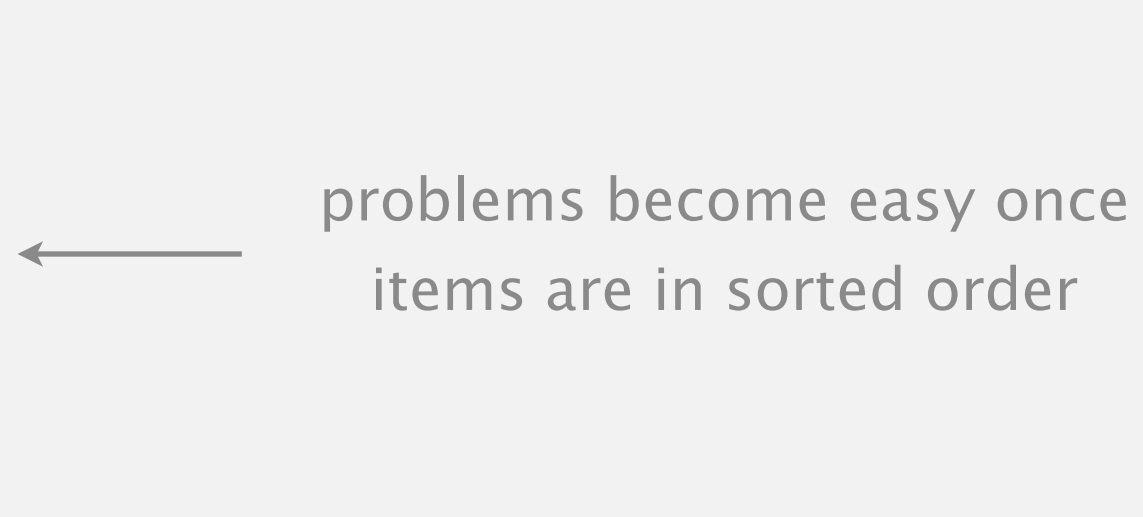
2.3 QUICKSORT

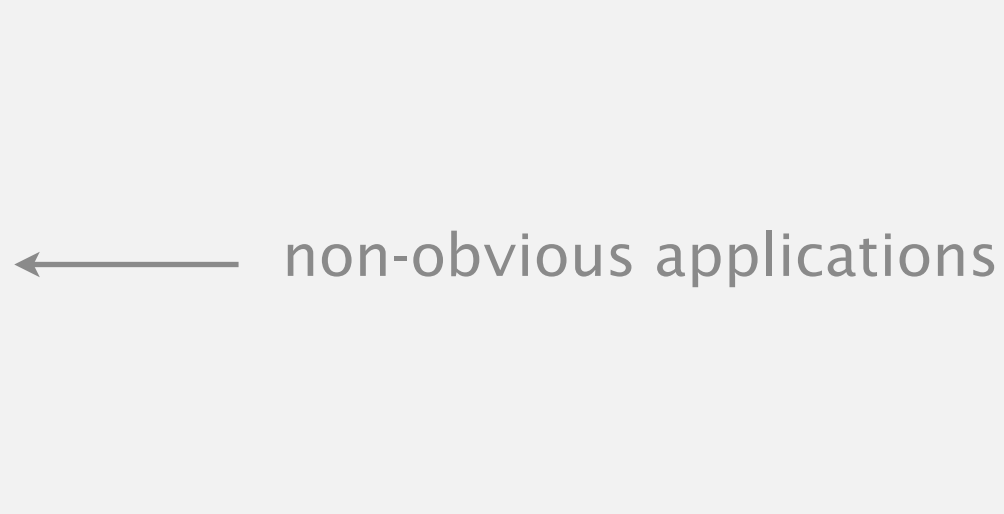
- ▶ *quicksort*
- ▶ *selection*
- ▶ *duplicate keys*
- ▶ *system sorts*

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.
- 

...

Engineering a system sort (in 1990s)


Bentley–McIlroy quicksort.

- Cutoff to insertion sort for small subarrays.
- Pivot selection: 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.

In the wild. C, C++, Java 6,

A Java 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 $P1 \leq P2$, 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]

...

Another Java mailing list post (Yaroslavskiy–Bloch–Bentley)

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/t1/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

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

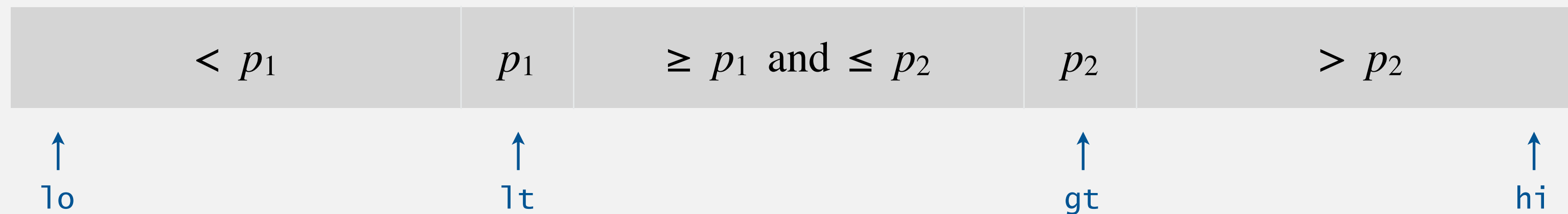
+ src/share/classes/java/util/DualPivotQuicksort.java

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

Dual-pivot quicksort

Use **two** pivots 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 (skip middle subarray if $p_1 = p_2$).

↑
degenerates to Dijkstra's 3-way partitioning

In the wild. Java 8, Java 11, Python unstable sort, Android, ...



**Suppose you are the lead architect of a new programming language.
Which sorting algorithm(s) would you use for the system sort? Defend your answer.**

System sorts in Java 8 and Java 11

`Arrays.sort()` and `Arrays.parallelSort()`.

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



Algorithms.

- **Timsort** for reference types.
- **Dual-pivot quicksort** for primitive types.
- Parallel mergesort for `Arrays.parallelSort()`.

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

Bottom line. Use the system sort!

```

k) lo = i + 1; else return a[i]; } return a[lo]; } p
mpareTo(w) < 0); } private static void exch(Object[] a,
private static boolean isSorted(Comparable[] a) { return
ted(Comparable[] a, int lo, int hi) { for (int i = lo + 1;
n true; } private static void show(Comparable[] a) { for (in
public static void main(String[] args) { String[] a = StdIn.rea
or (int i = 0; i < a.length; i++) { String ith = (String) Quick
public class Quick { public static void sort(Comparable[] a) { St
static void sort(Comparable[] a, int lo, int hi) { if (hi <= lo)
(a, lo, j-1); sort(a, j+1, hi); } private boolean isSorted(a, lo, hi);
lo, int hi) { int i = lo; while (less(a[i], a[j])) i++; Comparable v = a[i]
ak; while (less(v, a[j])) j++; exch(a, i, j); if (i >= j) break; if (i >= j)
ic static Comparable select(Comparable[] a, int k) { if (k < 0 || k > a.length)
ected element out of bounds; StdRandom.shuffle(a); int lo = 0, hi = a.length - 1;
ition(a, lo, hi); if (i == k) return a[i]; else if (i < k) lo = i + 1; else if (i > k) hi = i - 1; }
boolean less(Comparable v, Comparable w) { return (v.compareTo(w) < 0); }
int j) { Object swap = a[i]; a[i] = a[j]; a[j] = swap; } private static boolean isSorted(a, lo, hi) { for (int i = lo + 1; i <= hi; i++) if (less(a[i], a[i-1])) return false; return true; } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) { StdOut.println(a[i]); } } public static void main(String[] args) { String[] a = StdIn.readStrings(); Quick.sort(a); show(a); StdOut.println("Sorted array:"); for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
ring) Quick.select(a, i); StdOut.println(ith); } }
ndom.shuffle(a); sort(a, 0, a.length - 1); } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
return; int j = partition(a, lo, hi); sort(a, lo, j-1); sort(a, j+1, hi); } }
static int partition(Comparable[] a, int lo, int hi) { int i = lo, j = hi + 1; Comparable v = a[lo]; while (less(a[j], v)) j++; exch(a, lo, j); while (less(v, a[i])) i++; exch(a, i, j); return j; } }
) { while (less(a[i], a[j])) i++; exch(a, i, j); } } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
a, i, j); } } private static boolean isSorted(Comparable[] a, int lo, int hi) { for (int i = lo + 1; i <= hi; i++) if (less(a[i], a[i-1])) return false; return true; } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
th) { throw new RuntimeException("Illegal arguments: " + lo + " and " + hi + " = a.length - 1; } }
else return a[i]; } } private static boolean isSorted(Comparable[] a, int lo, int hi) { for (int i = lo + 1; i <= hi; i++) if (less(a[i], a[i-1])) return false; return true; } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
mpareTo(w) < 0); } private static void exch(Object[] a, int i, int j) { Object swap = a[i]; a[i] = a[j]; a[j] = swap; } private static boolean isSorted(Comparable[] a, int lo, int hi) { for (int i = lo + 1; i <= hi; i++) if (less(a[i], a[i-1])) return false; return true; } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
private static boolean isSorted(Comparable[] a, int lo, int hi) { for (int i = lo + 1; i <= hi; i++) if (less(a[i], a[i-1])) return false; return true; } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
ted(Comparable[] a, int lo, int hi) { for (int i = lo + 1; i <= hi; i++) if (less(a[i], a[i-1])) return false; return true; } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
n true; } private static void show(Comparable[] a) { for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
public static void main(String[] args) { String[] a = StdIn.readStrings(); Quick.sort(a); show(a); StdOut.println("Sorted array:"); for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
or (int i = 0; i < a.length; i++) StdOut.println(a[i]); } }
public class Quick { public static void sort(Comparable[] a, int lo, int hi) { if (hi <= lo) return; exch(a, lo, partition(a, lo, hi)); sort(a, lo, j-1); sort(a, j+1, hi); } }
static void sort(Comparable[] a, int lo, int hi) { if (hi <= lo) return; exch(a, lo, partition(a, lo, hi)); sort(a, lo, j-1); sort(a, j+1, hi); } }
(a, lo, j-1); sort(a, j+1, hi); } }

```