

INTRODUCTION TO ALGORITHMS

Lecture 6: Quick Sort Algorithm

Yao-Chung Fan
yfan@nchu.edu.tw

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]



Quicksort t-shirt



QUICKSORT

- ▶ *quicksort*
- ▶ *3ways quick sort*
- ▶ *selection (order statistics)*

Quicksort

Basic plan.

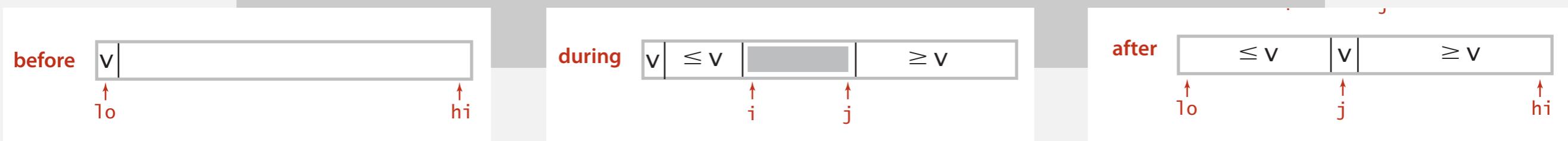
- **Shuffle** the array.
- **Partition** so that, for some j
 - entry $a[j]$ is in place
 - no larger entry to the left of j
 - no smaller entry to the right of j
- **Sort** each subarray recursively.

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
	<i>partitioning item</i>															
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	<i>not greater</i>							<i>not less</i>								
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: 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, if yes , break
        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
```

lo 為 [0]
先從左邊找比[0] 大的
數字
再從右邊找比[0] 小的
數字
再將兩個數字交換
再繼續從 i 位置往左
找比[0] 大的
j 位置往右找比[0] 小
的
直到 i 或 j 走到底 或
者 $i >= j$ 時
最後再將[0] 跟 j 位置
交換



A wonderful exception handler

```
try {  
    something  
} catch(e) {  
    window.location.href =  
        "http://stackoverflow.com/search?q=[js] + "  
        + e.message;  
}
```

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)

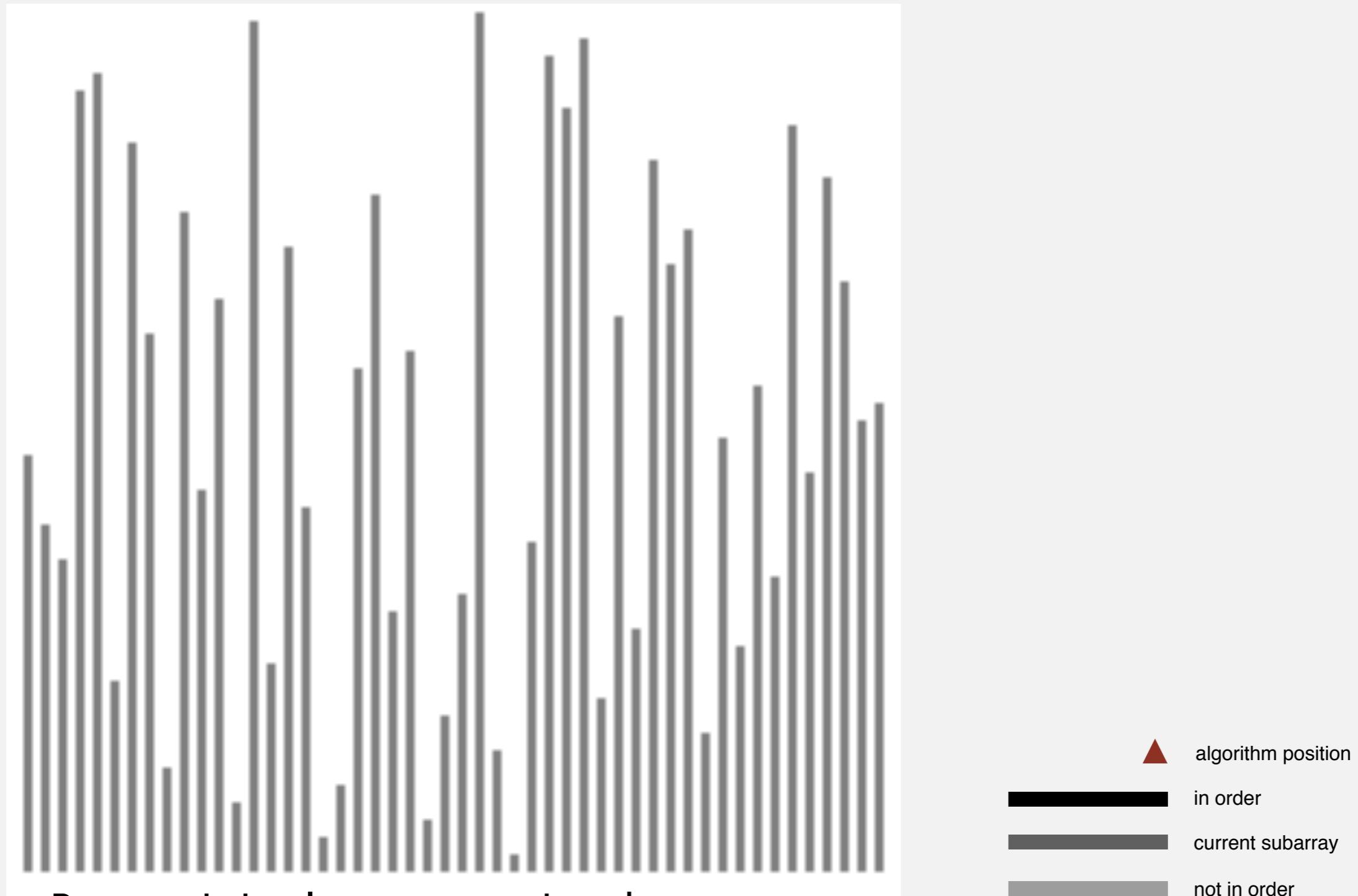
Quicksort trace

	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	0	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		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S
	4		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	0	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	0	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	0	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	0	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	0	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	0	P	Q	R	S	T	U	X
	10	10	10	A	C	E	E	I	K	L	M	0	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	0	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	0	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	0	P	Q	R	S	T	U	X

Quicksort trace (array contents after each partition)

Quicksort animation

50 random items



Reason it is slow: excessive data movement.

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 $\sim N \lg N$.

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

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 $\sim 2N \ln N$.

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

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

partitioning

left right

partitioning probability

- Multiply both sides by N and collect terms:

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

- 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 $N(N+1)$:

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

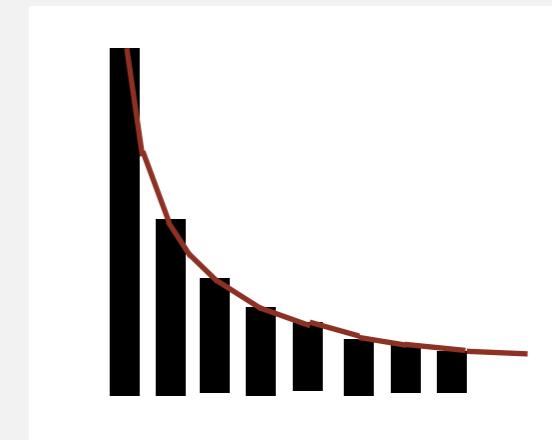
Quicksort: average-case analysis

- Repeatedly apply above equation:

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

- 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 **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.

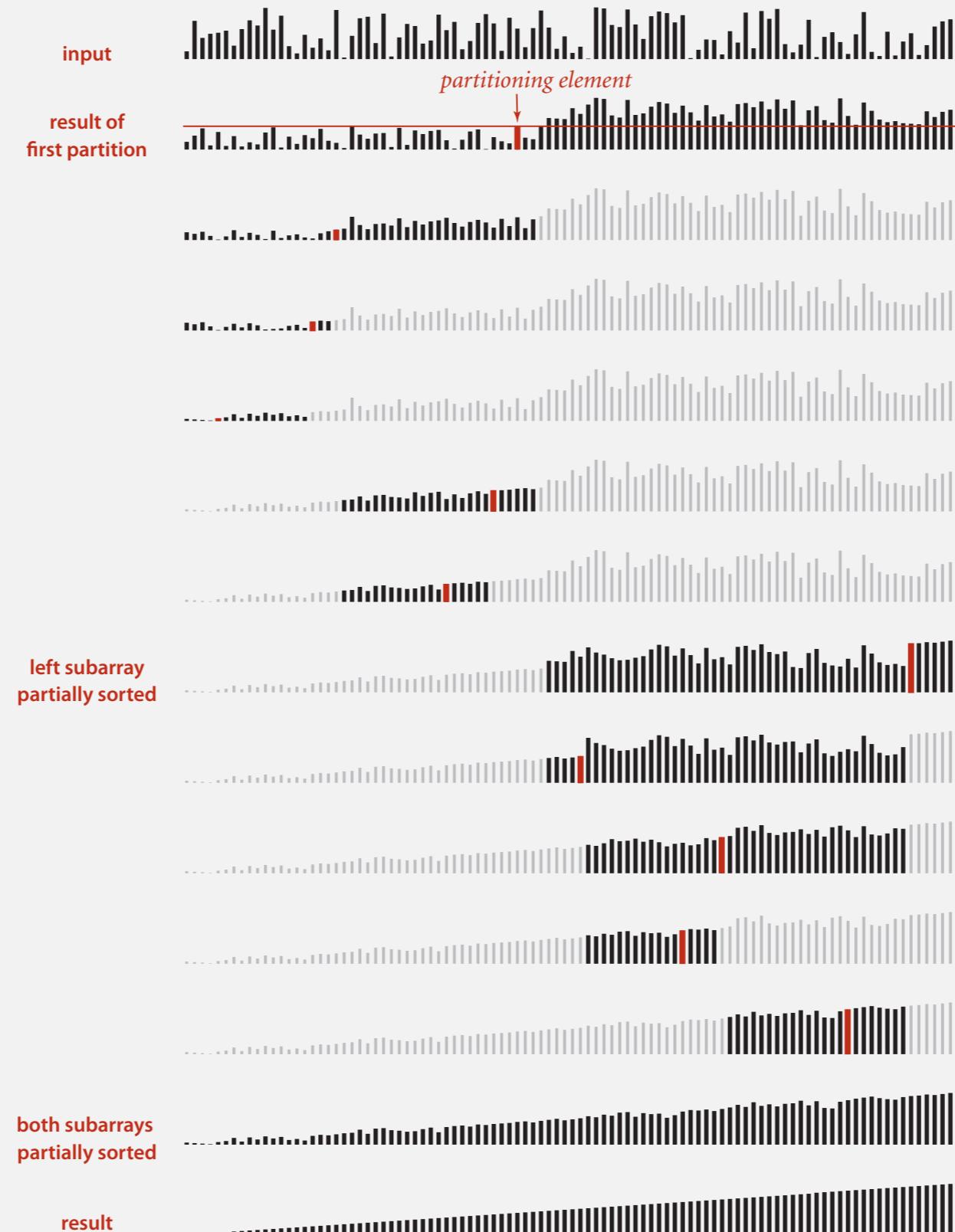
- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.

```
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);
}
```

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



Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier, but is not worth the cost.

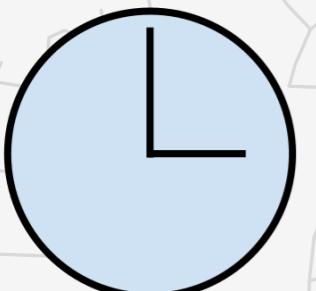
Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

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

Preserving randomness. Shuffling is needed for performance guarantee.

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

Time Consumption



L

Lines of Code



Bubble Sort



Quicksort

QUICKSORT

► *quicksort*

► **3 ways quick sort**

► *selection (order statistics)*

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

Quicksort with duplicate keys. Algorithm can go quadratic unless partitioning stops on equal keys!

S T O P O N E Q U A L K E Y S

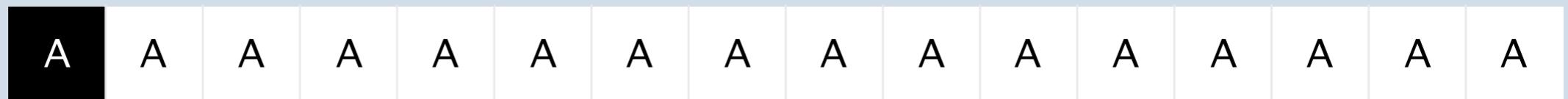
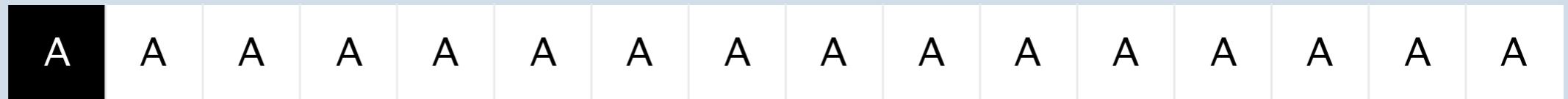
swap

if we don't stop on equal keys

if we stop on equal keys

Partitioning with all equal keys

What is the result of partitioning the following array?



A.



B.



C.

Partitioning an array with all equal keys

Given the st

		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: the problem

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

Consequence. $\sim N \lg N$ compares when all keys equal.

A A B A A **A** B A B A A

A A A A A **A** A A A A A A

Mistake. Don't stop scans on items equal to the partitioning item.

Consequence. $\sim \frac{1}{2} N^2$ compares when all keys equal.

A A B A A A B A B A A

A A A A A A A A A A A

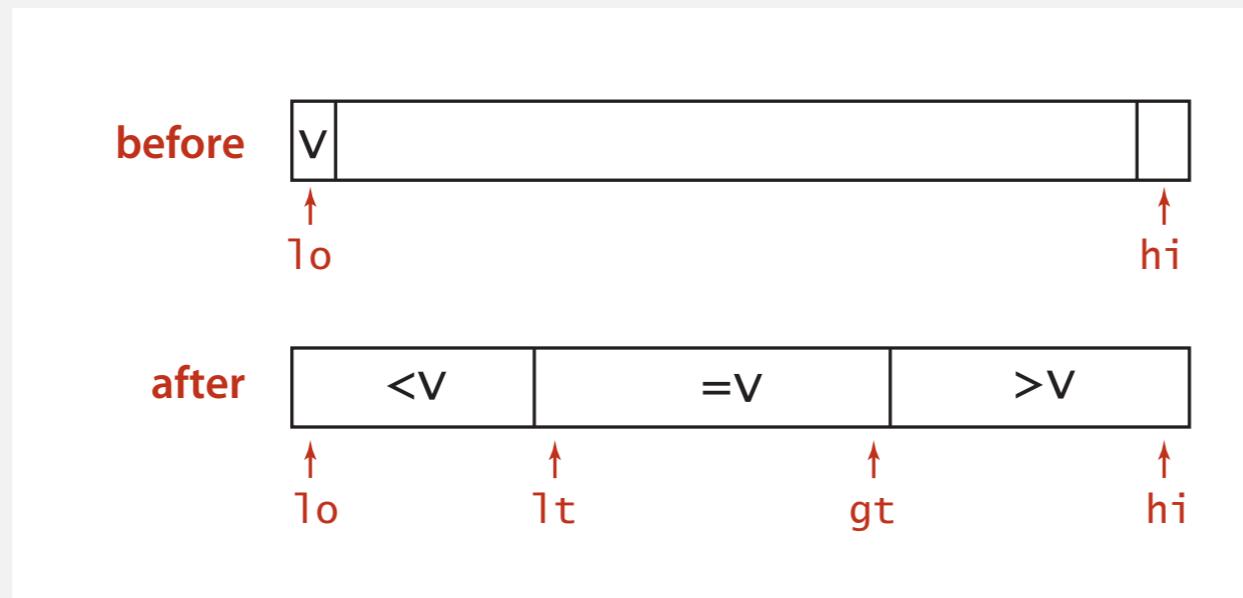
***Algorithm* (noun.)**

Word used by programmers when...
they do not want to explain what they did.

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.

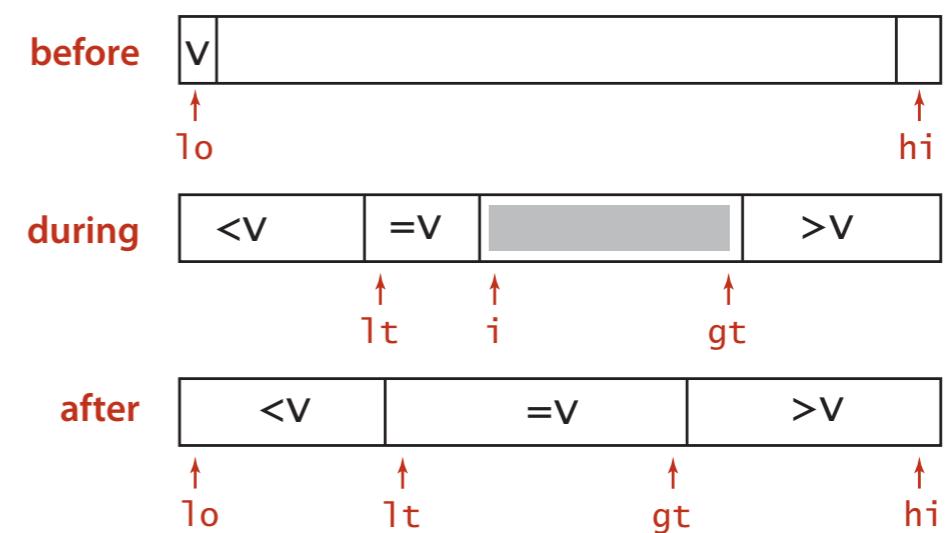
Dijkstra's 3-way partitioning: trace

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

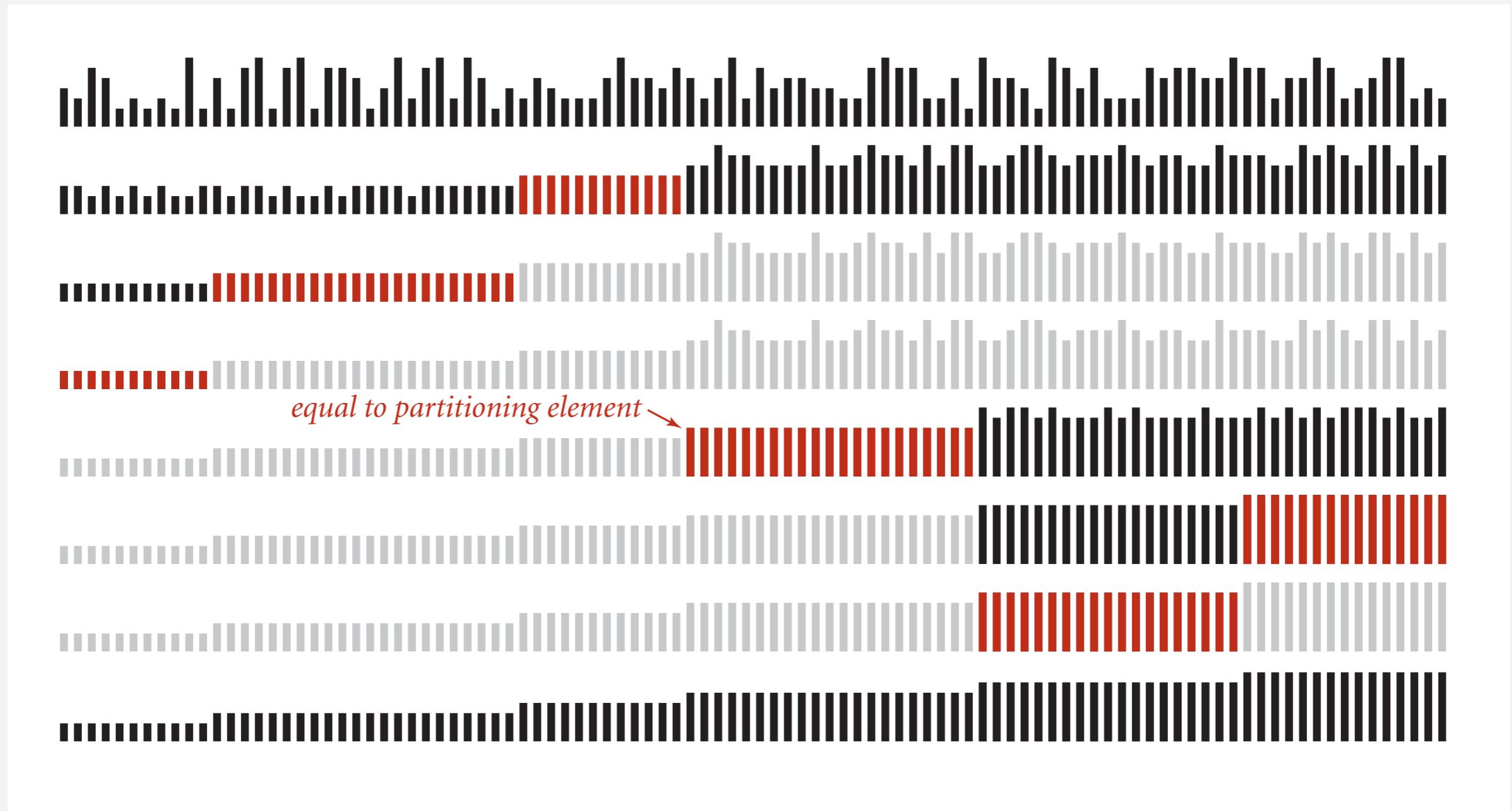
3-way partitioning trace (array contents after each loop iteration)

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



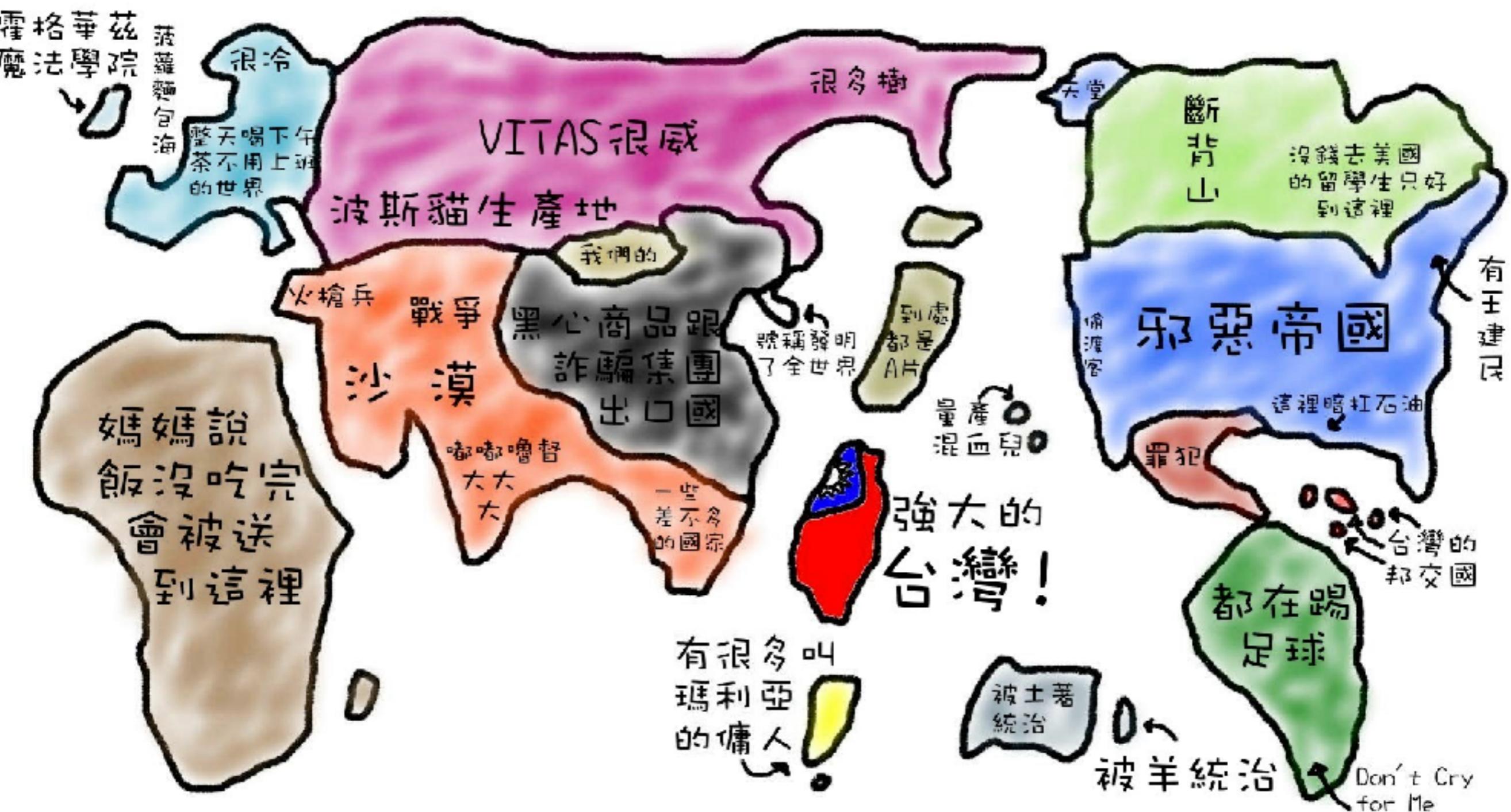
Sorting summary

	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

Which sorting algorithm to use?

Applications have diverse attributes.

- Stable?
- Parallel?
- In-place?
- Deterministic?
- Duplicate keys?
- Multiple key types?
- Large or small items?
- Randomly-ordered array?
- Guaranteed performance?



System sort in Java 7

`Arrays.sort()`.

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



Algorithms.

- Quicksort for primitive types.
- Timsort for reference types.

QUICKSORT

- ▶ *quicksort*
- ▶ *3 way partition*
- ▶ *selection*

Selection

Goal. Given an array of N items, find the k^{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?

Quick-select

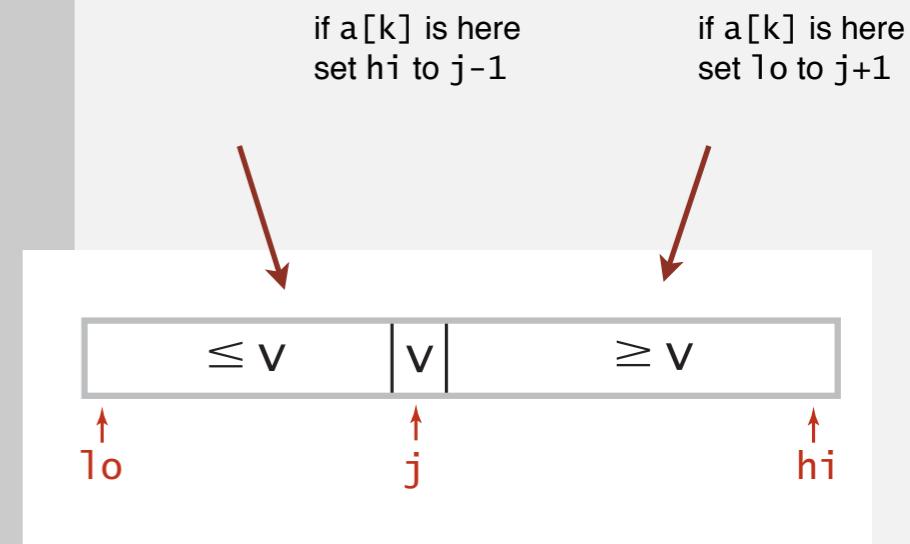
Partition array so that:

- 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 $k = 5$

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

$k = 5$

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

$k = 5$

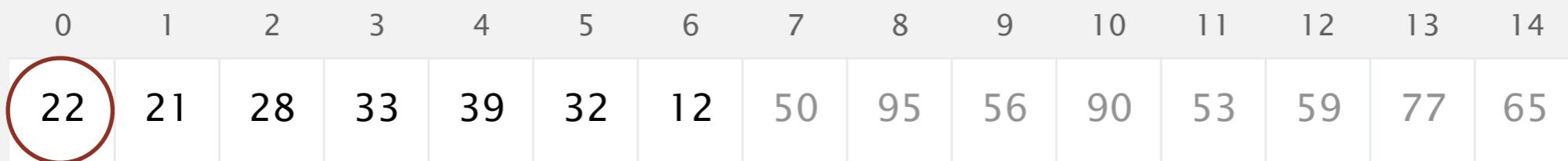
Quick-select demo

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



$k = 5$

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
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

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 .

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
12	21	22	32	28	33	39	50	95	56	90	53	59	77	65

$k = 5$

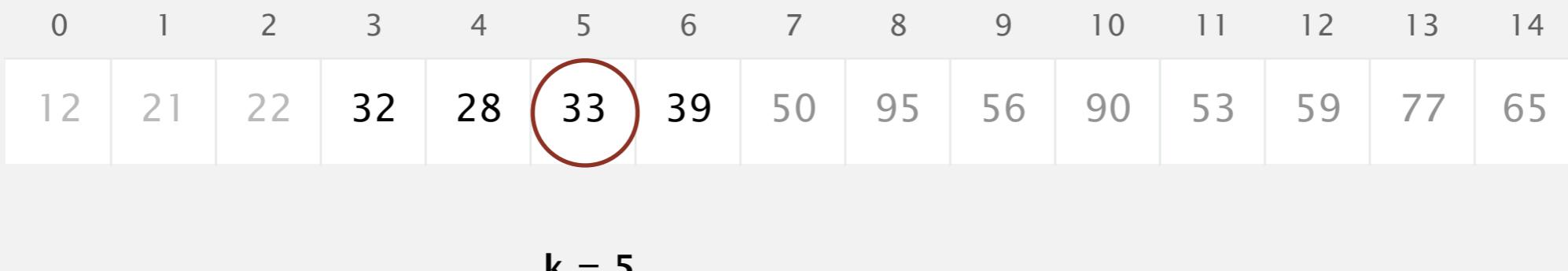
Quick-select demo

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:

$$N + N/2 + N/4 + \dots + 1 \sim 2N \text{ compares.}$$

