

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

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]



QUICKSORT

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

Quicksort overview

Step 1. Shuffle the array. *//eliminate dependency on input*

Step 2. Partition the array so that, for some j *//split the array (may not even)*

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

Step 3. Sort each subarray recursively.

Choosing the first value in the list for partition (pivot) is straightforward and typically works well

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

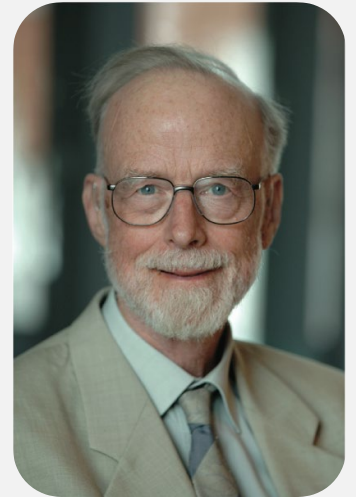
partitioning item (arrow from R to K)

not greater (arrow from A to E)

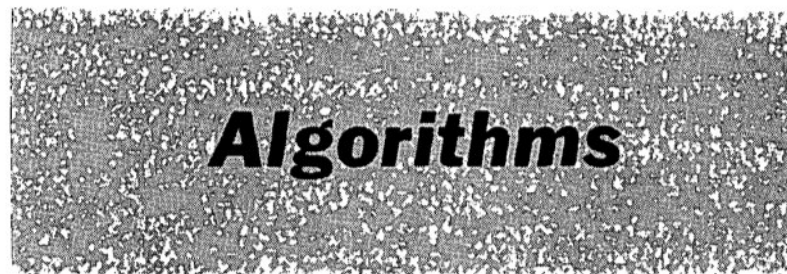
not less (arrow from Q to M)

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



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
```

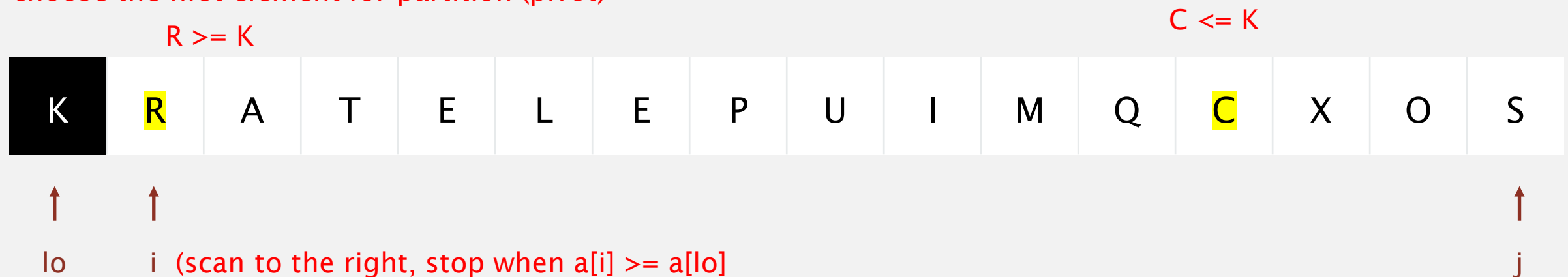
Communications of the ACM (July 1961)

Quicksort partitioning demo

Repeat until i and j pointers cross.

- Scan i from left to right so long as $(a[i] < a[lo])$. //stop when $a[i] \geq a[lo]$
- Scan j from right to left so long as $(a[j] > a[lo])$. //stop when $a[j] \leq a[lo]$
- **Exchange** $a[i]$ with $a[j]$.

choose the first element for partition (pivot)

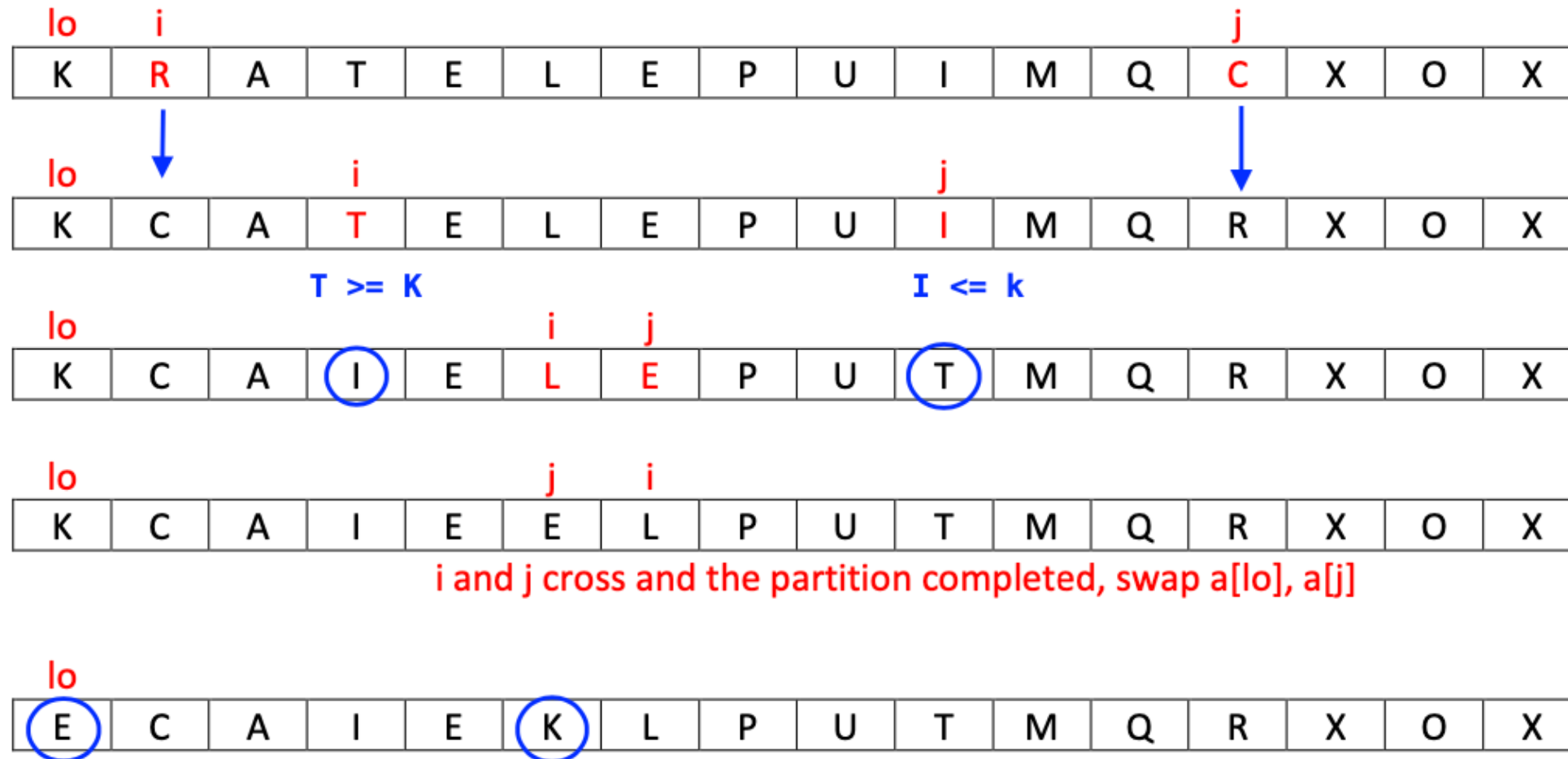


scan to the left, stop when $a[j] \leq a[lo]$



swap $a[i]$, $a[j]$

Partition completed



after the partition with pivot value K, all values to the left of K are no greater than ($\leq$) K,
and all values to the right of K are no smaller than ($\geq$) K

Quicksort partitioning demo

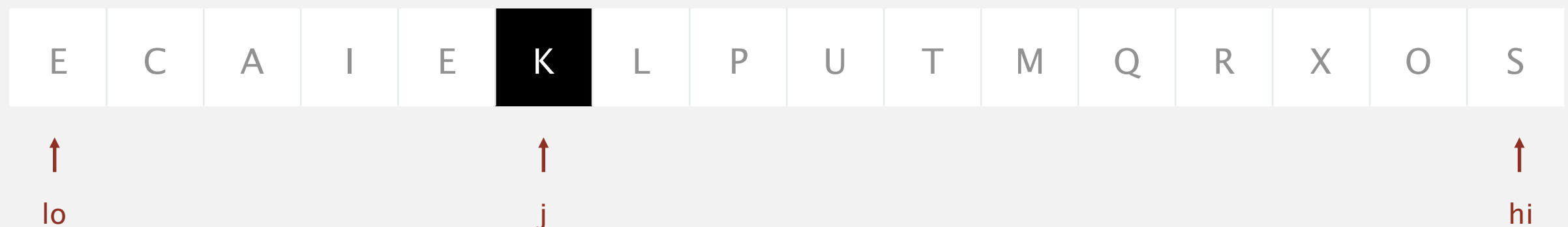
Repeat until i and j pointers cross.

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

When pointers cross.

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

NOTE: 2 subarrays are not in equal size after the partition



partitioned!

Quicksort quiz 1

In the worst case, how many compares and exchanges to partition an array of length n ?

Each exchange puts two new items into proper position, so only $\sim n/2$ exchanges needed in worst case (1 in best case).

- 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 $\sim n$

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

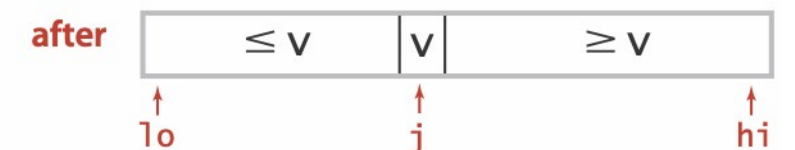
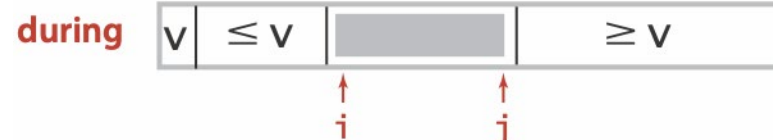
find item on right to swap

check if pointers cross

swap

swap with partitioning item

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; //no partition with array size 1
        int j = partition(a, lo, hi); //split at index j
        sort(a, lo, j-1); //recursively sort the left subarray
        sort(a, j+1, hi); //recursively sort the right subarray
    }
}
```

← shuffle needed for performance guarantee (stay tuned)

Quicksort trace //array contents after each partition

	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: 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

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: 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	n + 1 compares
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	n compares
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	n - 1 compares
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	

after random shuffle

- every partition divides array in one subarray of length 0 and one of length n-1
- Here, we assume random shuffle leaves array in same order as original

- every partition divides array in one subarray of length 0 and one of length n-1
- Here, we assume random shuffle leaves array in same order as original

Quicksort: best-case analysis

Best case. Number of compares is $\sim n \lg n$. //split into equal halves every partition

			a[]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
			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	I	J	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

after random shuffle

Quicksort: summary of performance characteristics

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

- Guaranteed to be correct.
- Running time depends on random shuffle. *//ideally, evenly split the array*

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 $\sim n \lg n$. *//evenly split every partition*

Worst case. Number of compares is $\sim \frac{1}{2} n^2$. *//sorted array*

[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. *//for the swap*
- **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**. *//swapping elements, including equal keys*

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 - lo)/2, hi);
    swap(a, lo, median);

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

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 theory as a guide.

- Easy $n \log n$ upper bound. How? //just sort the array and get rank k
- Easy n upper bound for $k = 0, 1, 2$. How? //if k is small, scan the array
- Easy n lower bound. Why? //need to scan the whole array to get the item of rank k

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

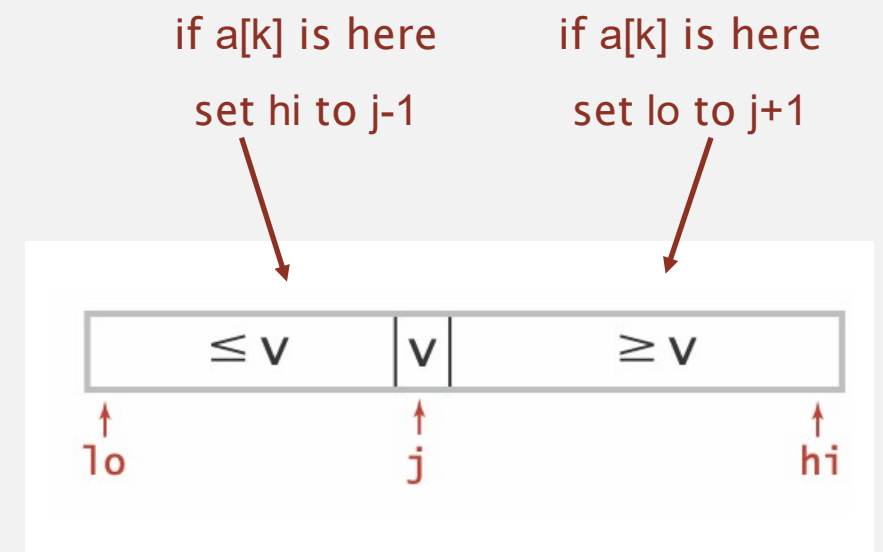
Partition array so that:

- 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; //select for more items in the right subarray
        else if (j > k) hi = j - 1; //select items in the left subarray
        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:

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

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

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.

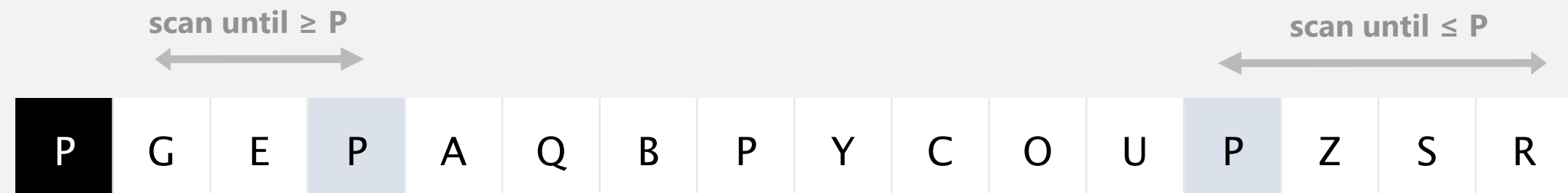
- Huge array.
- Small number of key values. *//many duplicates*

```
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
```

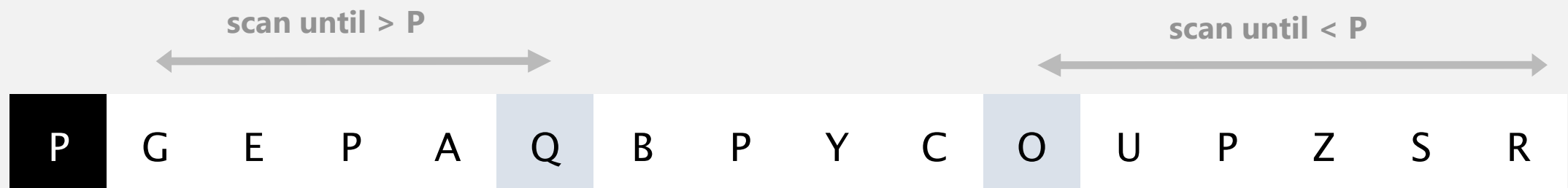
↑
key

Duplicate keys: stop on equal keys

Our partitioning subroutine stops both scans on equal keys.



Q. Why not continue scans on equal keys?

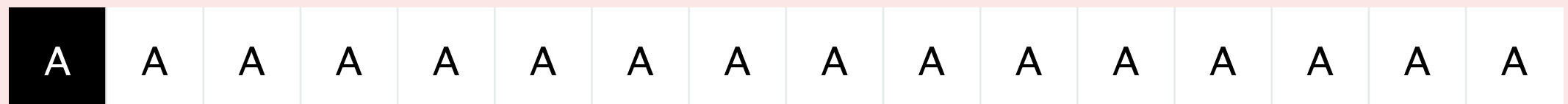


Quicksort: quiz 2

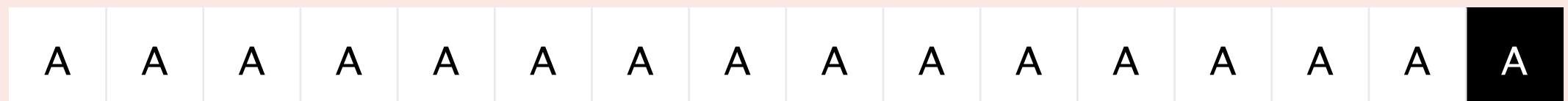
What is the result of partitioning the following array (**skip over equal keys**)?



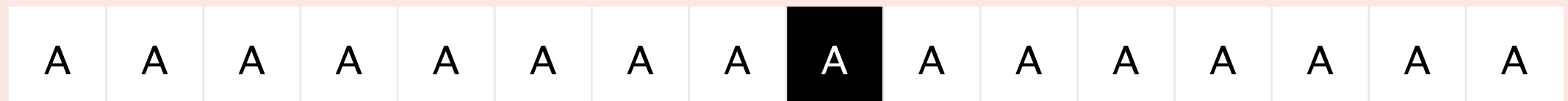
A.



B.



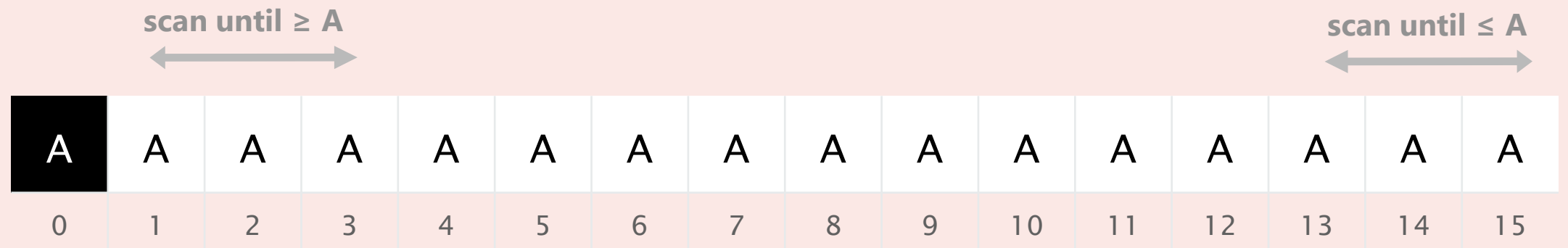
C.



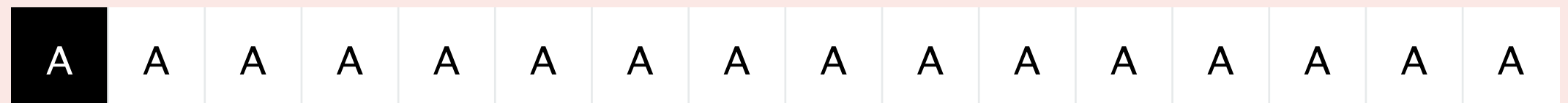
D. *I don't know.*

Quicksort: quiz 3

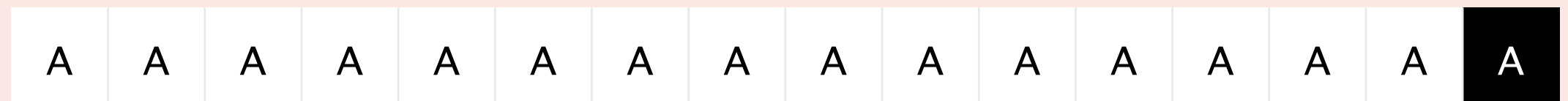
What is the result of partitioning the following array (**stop on equal keys**)?



A.



B.



C.



D. *I don't know.*

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	A
1	15	A	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	A
2	14	A	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	A
3	13	A	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	A
4	12	A	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	A
5	11	A	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	A
6	10	A	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	A
7	9	A	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	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

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

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**

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
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 pre-existing 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 ^[1] when duplicate keys
?	✓	✓	n	$n \lg n$	$n \lg n$	holy sorting grail

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

Which sorting algorithm to use?

Many sorting algorithms to choose from:

sorts	algorithms
elementary sorts	insertion sort, selection sort, bubblesort, shaker sort, ...
subquadratic sorts	quicksort, mergesort, heapsort, shellsort, samplesort, ...
system sorts	dual-pivot quicksort, timsort, introsort, ...
external sorts	Poly-phase mergesort, cascade-merge, psort,
radix sorts	MSD, LSD, 3-way radix quicksort, ...
parallel sorts	bitonic sort, odd-even sort, smooth sort, GPUsort, ...

Which sorting algorithm to use?

Applications have diverse attributes.

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

		attributes								
		1	2	3	4	.	.	.	.	M
algorithm	A	●			●					
	B			●		●			●	
	C		●		●					
	D						●			
	E			●						
	F		●			●		●		
	G	●								●
	.			●		●		●		
	.		●	●				●		
	.						●			●
	K	●				●				

many more combinations of
attributes than algorithms

Q. Is the system sort good enough?

A. Usually.

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

QUICKSORT

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

