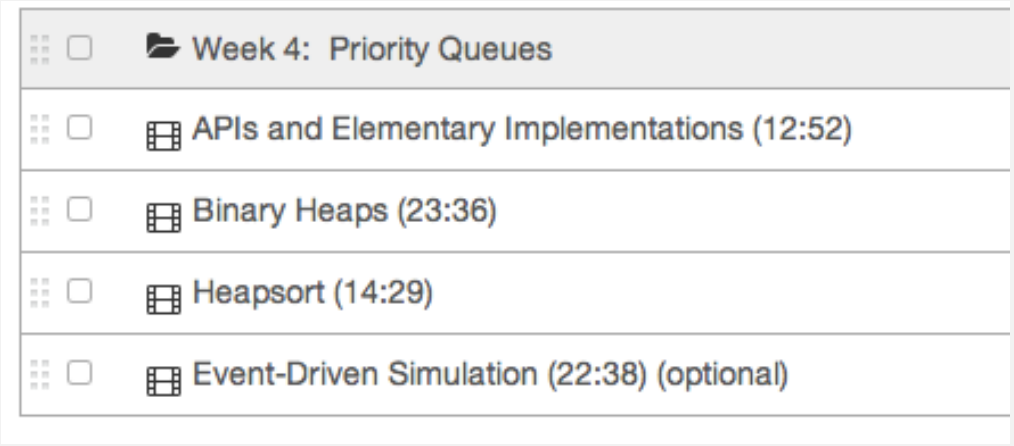
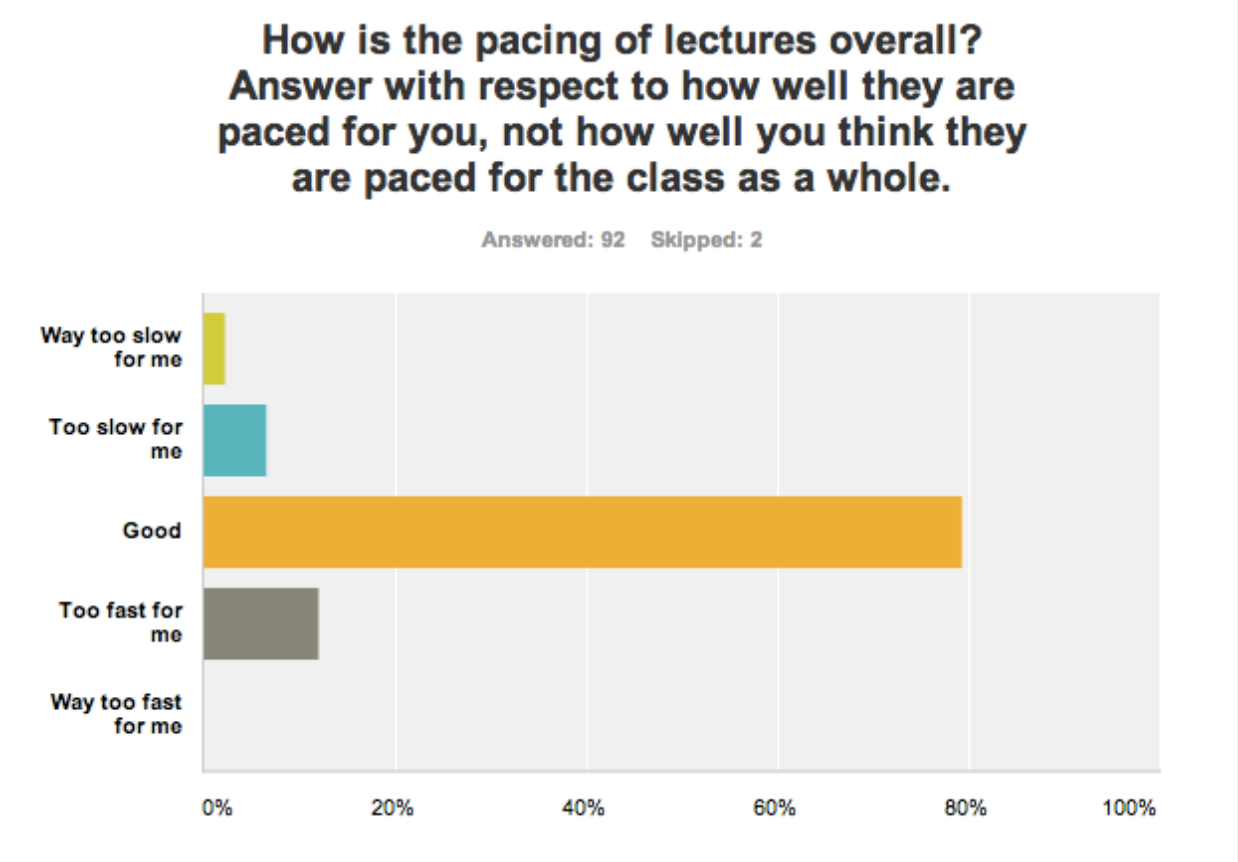
Announcements

Last normal lecture for 3 weeks.

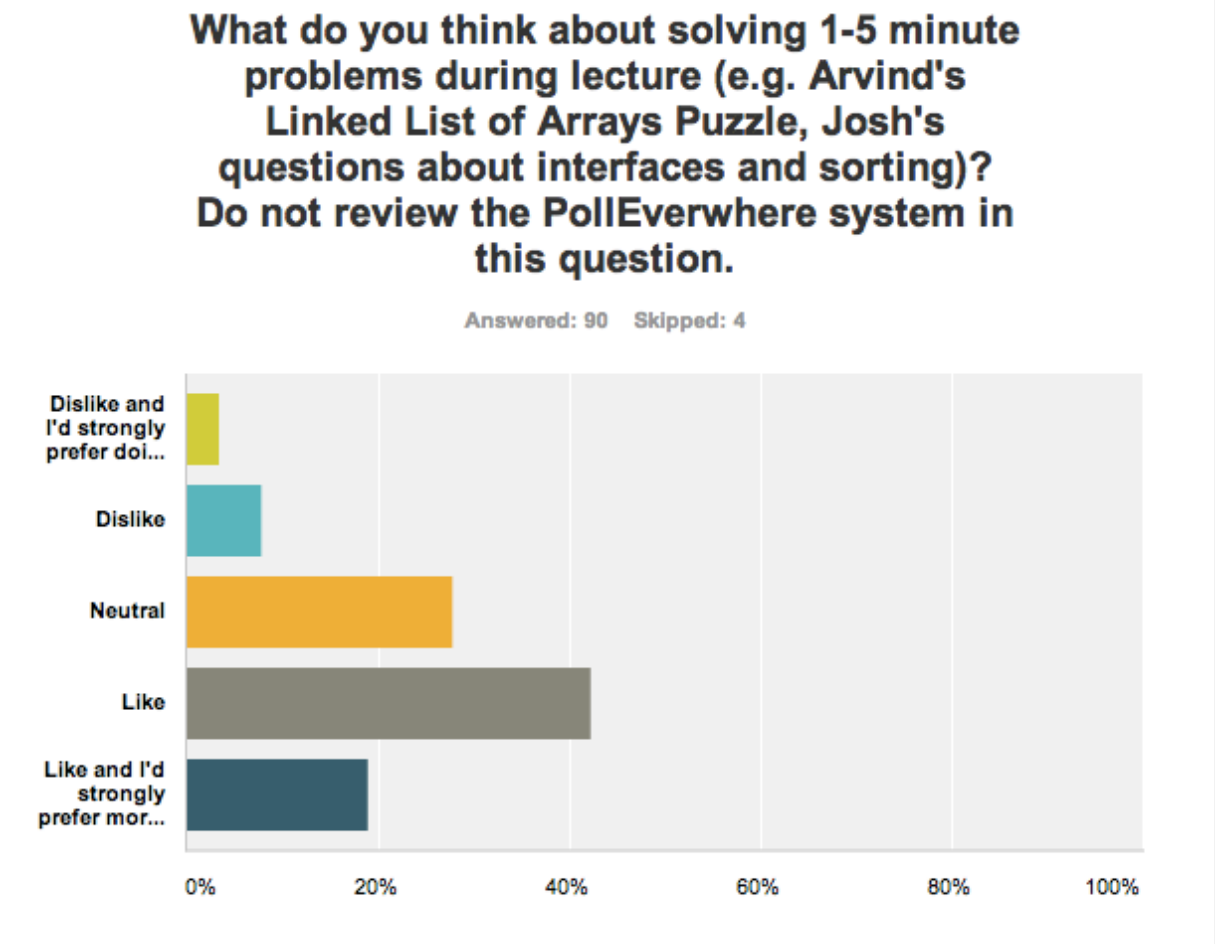
- Sign up for Coursera if you have not already.
- Next week's video lectures will be available tomorrow at 2 PM.
  - Exercises also posted (so you can test comprehension).
- Watch Priority Queues lecture before Monday.
  - 52 minutes. Can be watched at anywhere from 0.5x to 2x speed.
  - If you don't, you will be lost.
- No new material during flipped lecture.
  - Mini-lecture on big picture issues.
  - Solve interesting problems (including old exam problems).
  - Any questions under pinned post on Piazza will be answered.



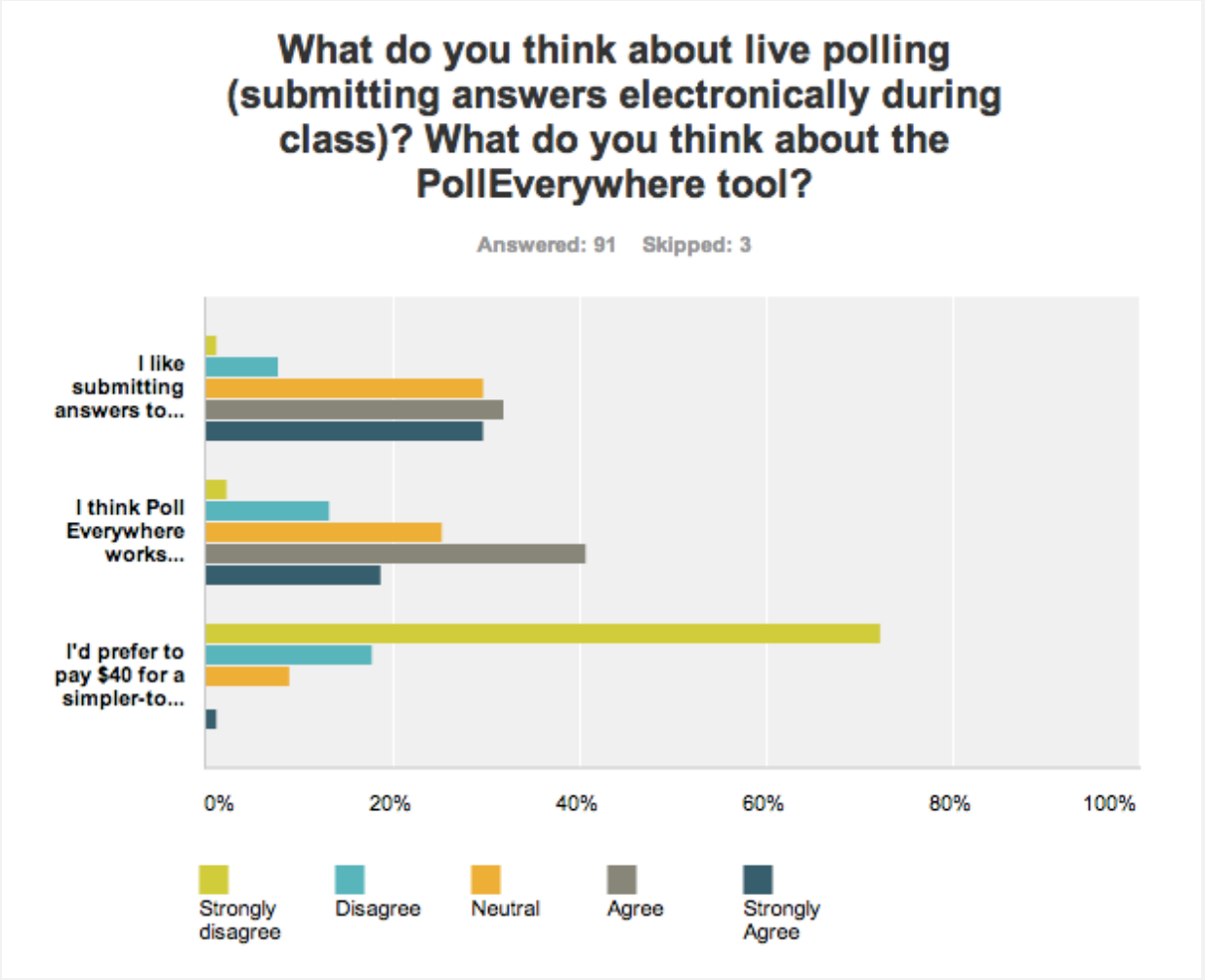
Survey says…


How is the pacing of lectures overall? Answer with respect to how well they are paced for you, not how well you think they are paced for the class as a whole.
Answered: 92 Skipped: 2
Way too slow for me
Too slow for me
Good
Too fast for me
Way too fast for me
0%
20%
40%
60%
80%
100%

Survey says…


What do you think about solving 1-5 minute problems during lecture (e.g. Arvind's Linked List of Arrays Puzzle, Josh's questions about interfaces and sorting)? Do not review the PollEverywhere system in this question.
Answered: 90 Skipped: 4
Dislike and I'd strongly prefer doi...
Dislike
Neutral
Like
Like and I'd strongly prefer mor...
0%
20%
40%
60%
80%
100%

Survey says…


What do you think about live polling (submitting answers electronically during class)? What do you think about the PollEverywhere tool?
Answered: 91 Skipped: 3
I like submitting answers to...
I think Poll Everywhere works...
I'd prefer to pay $40 for a simpler-to...
0%
20%
40%
60%
80%
100%
Strongly disagree
Disagree
Neutral
Agree
Strongly Agree

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 20th century in science and engineering.

Mergesort: 4
Quicksort: 4

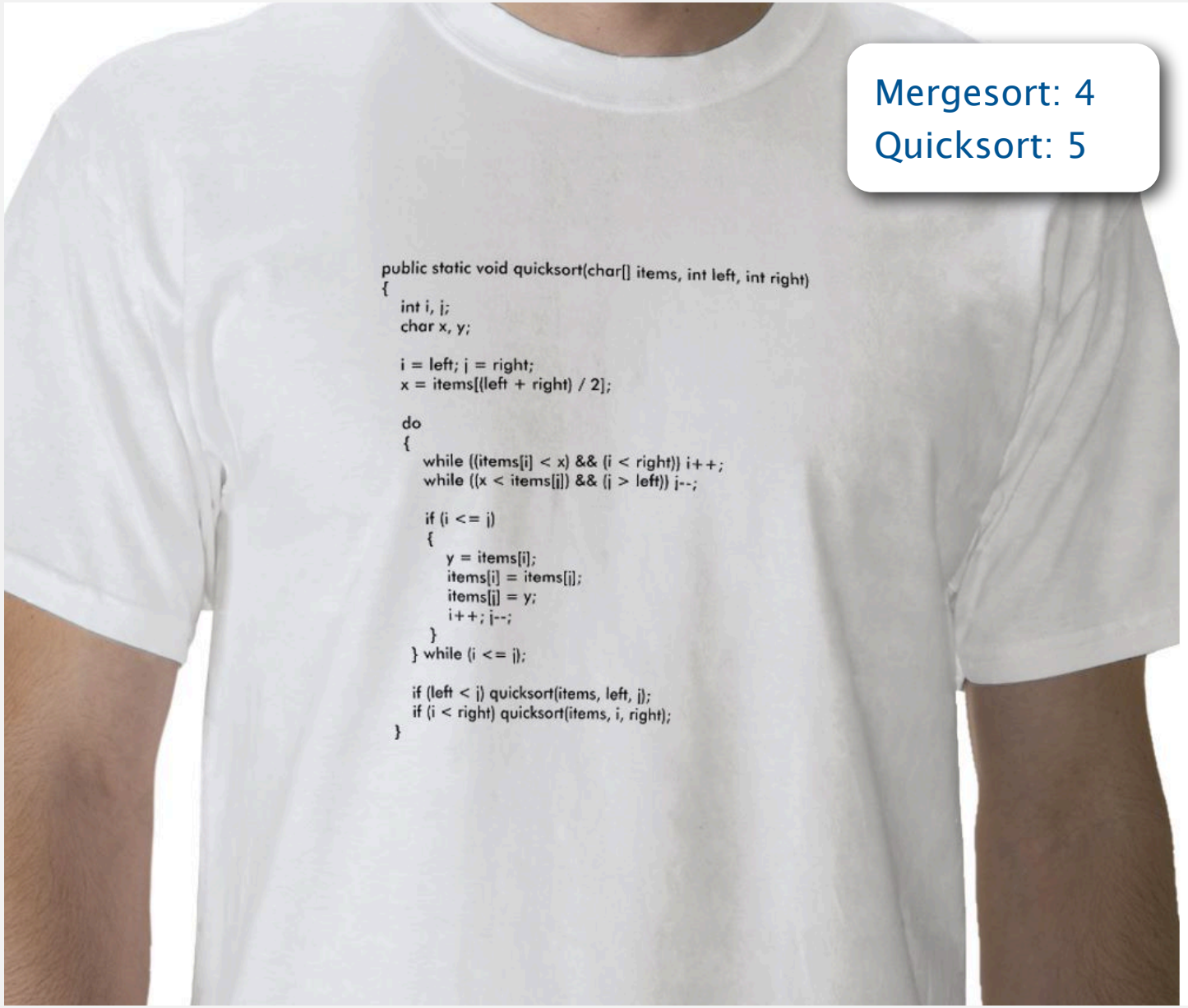
Mergesort. ← last lecture

- 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



http://algs4.cs.princeton.edu

2.3 Quicksort

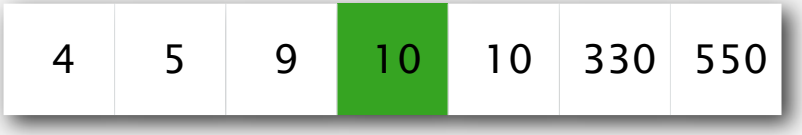
- *quicksort*
- *selection*
- *duplicate keys*
- *system sorts*

Partitioning - the heart of Quicksort

Partitioning.

- To **Partition** an array a[] on element x=a[i] is to rearrange it such that
  - x moves to position j (may be the same as i)
  - All entries to the left of x are <= x.
  - All entries to the right of x are >= x.

A.



B.

C.

D.

pollEv.com/jhug text to **37607**

Q: Which partitions are valid?

A: [405472]

A, B: [405478]

A, B, C: [405479]

A, B, C, D: [405481]

NONE: [405482]

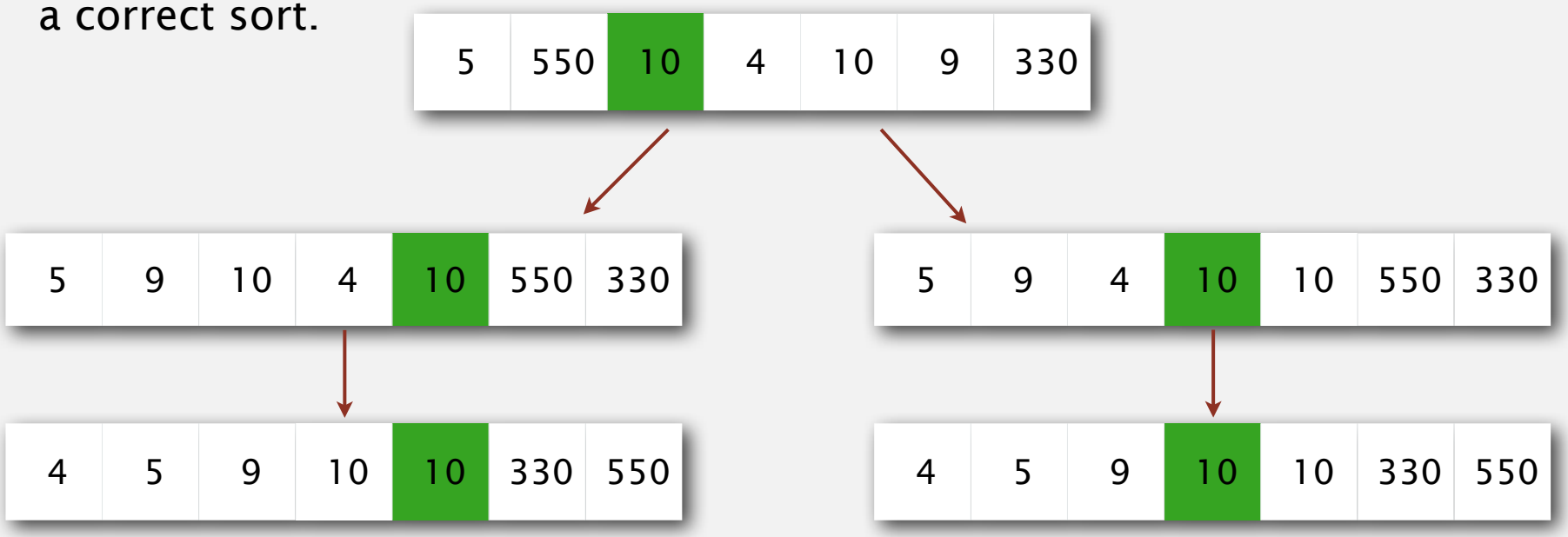
Partitioning

Partitioning.

- To **Partition** an array a[] on element x=a[i] is to rearrange it such that
  - x moves to position j (may be the same as i)
  - All entries to the left of x are <= x.
  - All entries to the right of x are >= x.

Observations.

- O1. After **Partitioning**, x is 'in place.' No need to move x in order to reach a correct sort.



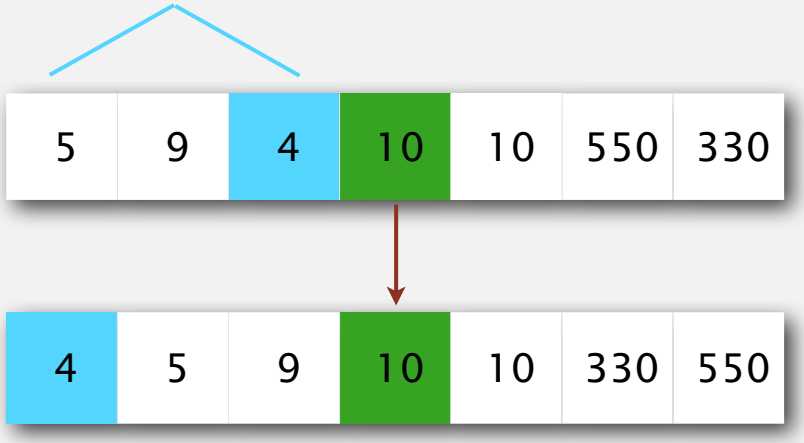
Partitioning

Partitioning.

- To **Partition** an array a[] on element x=a[i] is to rearrange it such that
  - x moves to position j (may be the same as i)
  - All entries to the left of x are <= x.
  - All entries to the right of x are >= x.

Observations.

- O1. After **Partitioning**, our pivot x is 'in place.' No need to move in order to reach a correct sort.
- O2. To **Partition** on a new pivot y, there's no need to look beyond the confines of previously **Partitioned** items.



Partition-Based Sorting

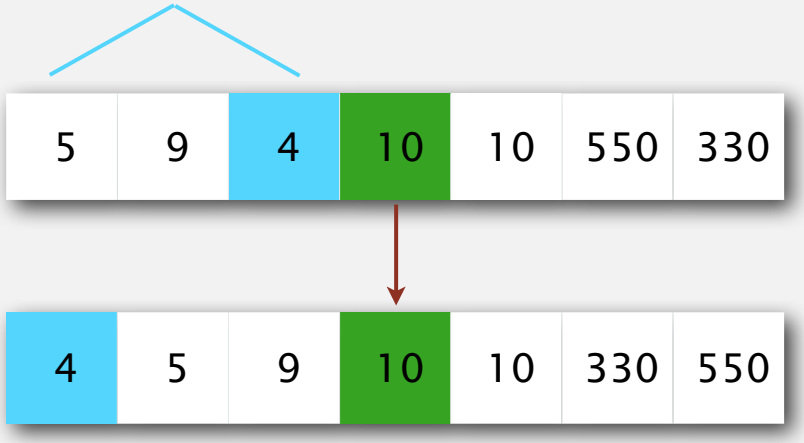
Observations.

- O1. After **Partitioning**, the pivot x is 'in place.' No need to move in order to reach a correct sort.
- O2. To **Partition** on a new pivot y, there's no need to look beyond the confines of previously **Partitioned** items.

Partition-Based Sorting.

- `PBSort(a, lo, hi)`
  - **Partition** on pivot `a[i]`, which ends up in position `j`
  - `PBSort(a, lo, j-1)`
  - `PBSort(a, j+1, hi)`

nothing to do if hi >= lo



Analysis.

- O1. Proof that N partitions yields a sort.
- O2. Suggests that it might be fast.

Quicksort

Partition-Based Sorting.

- `PBSort(a, lo, hi)`
  - **Partition** on pivot `a[i]`, which ends up in position `j`
  - `PBSort(a, lo, j-1)`
  - `PBSort(a, j+1, hi)`

Needed Decisions.

- Which item do we use as a pivot?
- How do we **Partition** items with respect to that pivot?

Quicksort

Partition-Based Sorting.

- `PBSort(a, lo, hi)`
  - **Partition** on pivot `a[i]`, which ends up in position `j`
  - `PBSort(a, lo, j-1)`
  - `PBSort(a, j+1, hi)`

Needed Decisions.

- Which item do we use as a pivot?
- How do we **Partition** items with respect to that pivot?

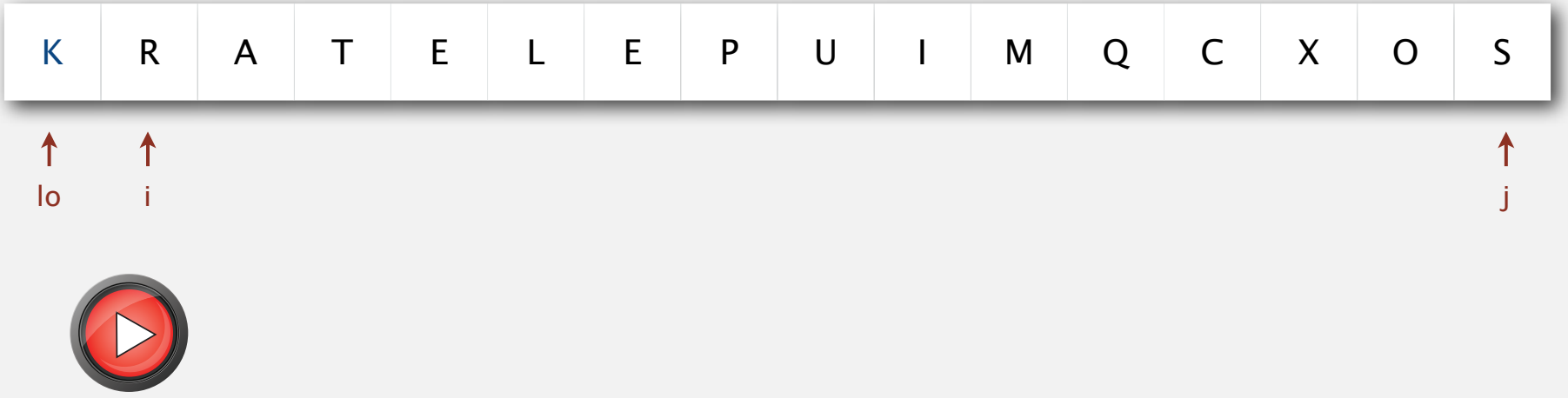
Quicksort (in 226).

- Always use leftmost item in the subarray as pivot.
- Use Quicksort partitioning to perform **Partitioning**.
  - a.k.a. Sedgewick 2-way partitioning.
- Perform a shuffle before only the first `PBSort` call. ← shuffle needed for performance guarantee (stay tuned)

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]`.



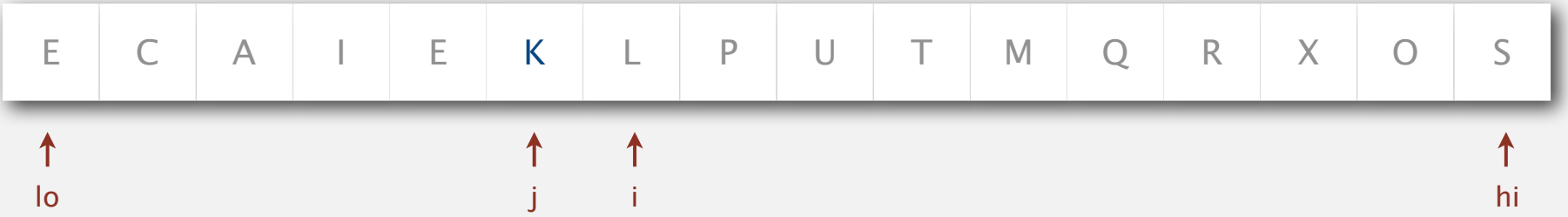
Quicksort partitioning demo

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]`.

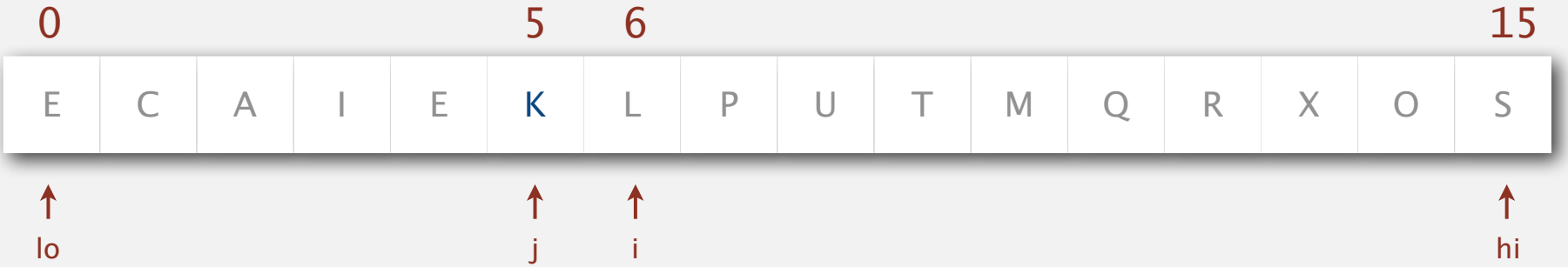


partitioned!

Quicksort 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]`.



pollEv.com/jhug text to **37607**

Q: How many total array **compares** occurred during the partition process?

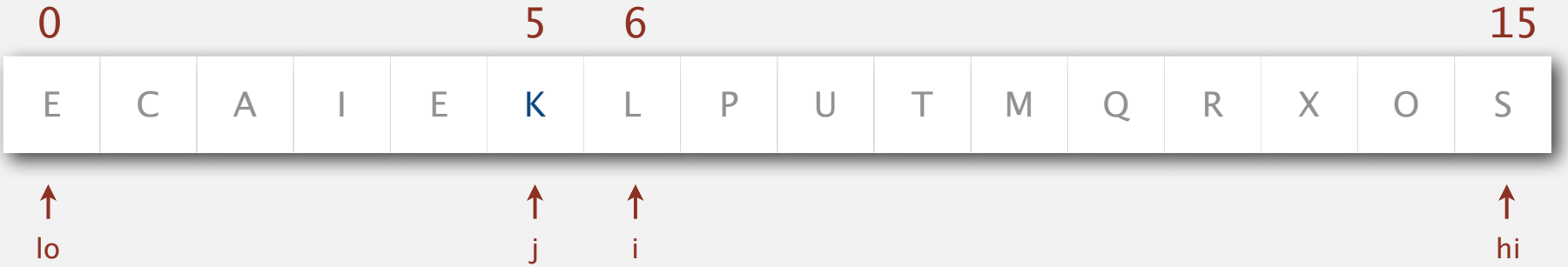
If texting, send "506314 ###", where ### is your answer.

Example: **506314 28** would mean that you think the answer is 28.

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]`.



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

- 6 compares: i pointer checked C A I E K L.
- 11 compares: j pointer checked S O X R Q M T U P L K.
- Total compares: 17
- More generally: N + 1 compares to partition

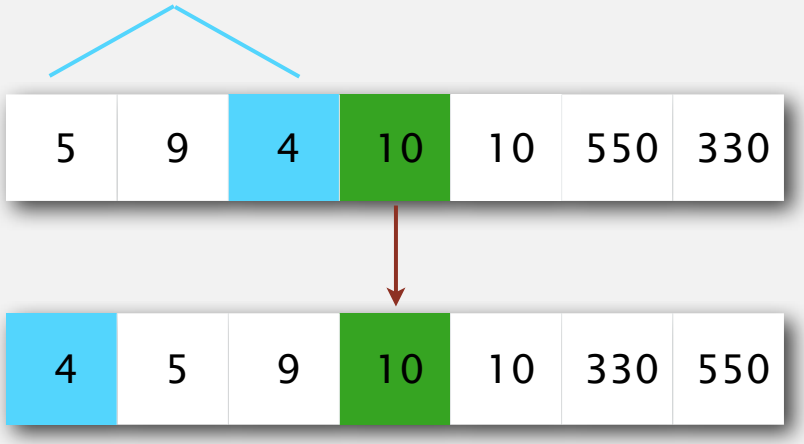
The Wages of Partitioning

Observations.

- O1. After **Partitioning**, the pivot x is 'in place.' No need to move in order to reach a correct sort.
- O2. To **Partition** on a new pivot y, there's no need to look beyond the confines of previously **Partitioned** items.

Analysis by Example.

- If **Partitioning** = Sedgewick 2-way partitioning, on length 7 array:
  - First partition: 8 compares
  - Left partition: 4 compares
- Each partition takes less time!
- Ideally want item to end up in the middle.



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]))
         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

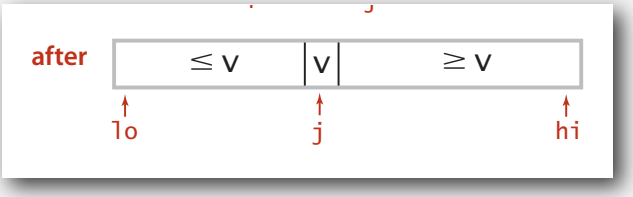
find item on right to swap

check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place



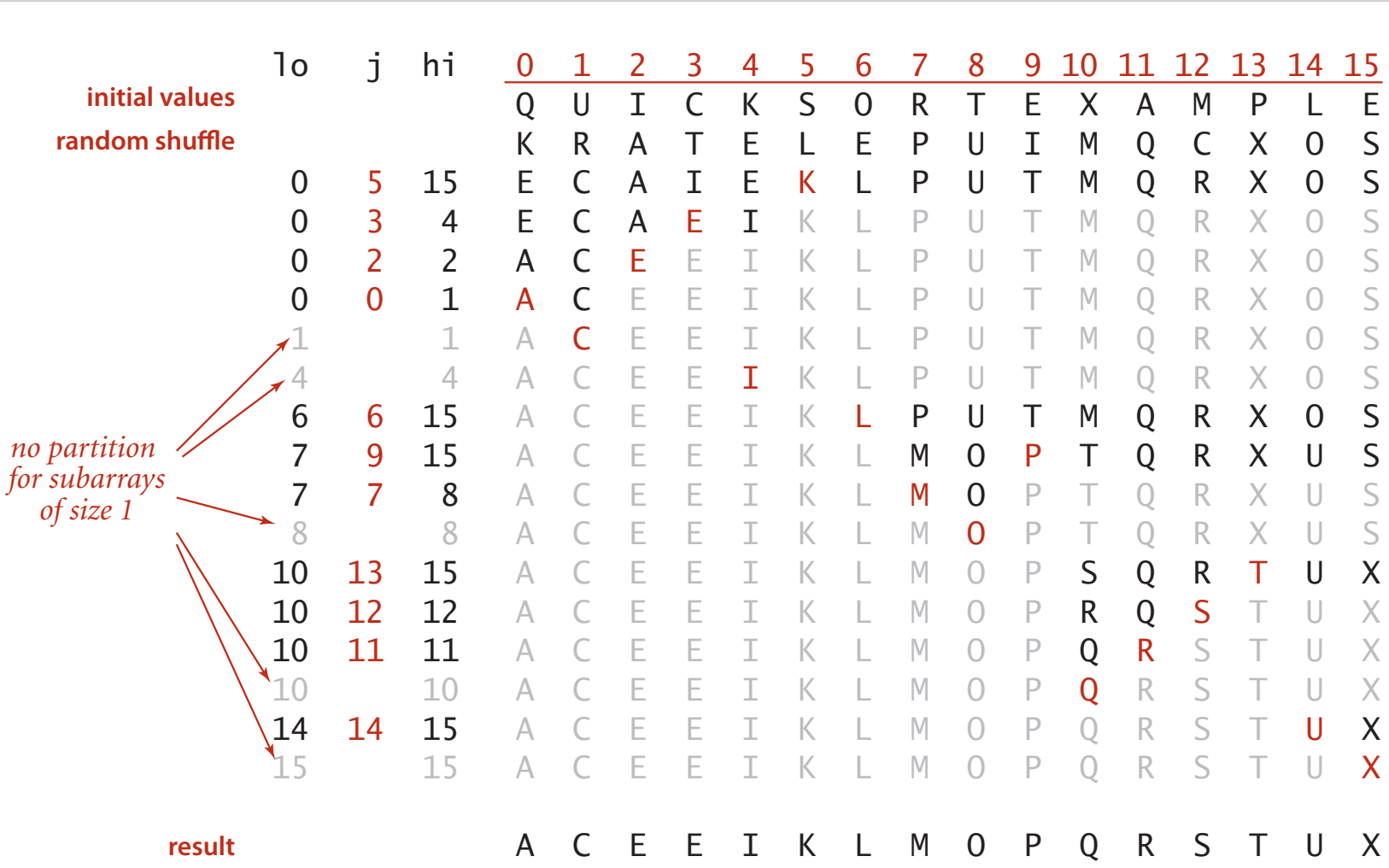
Quicksort: Java implementation

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

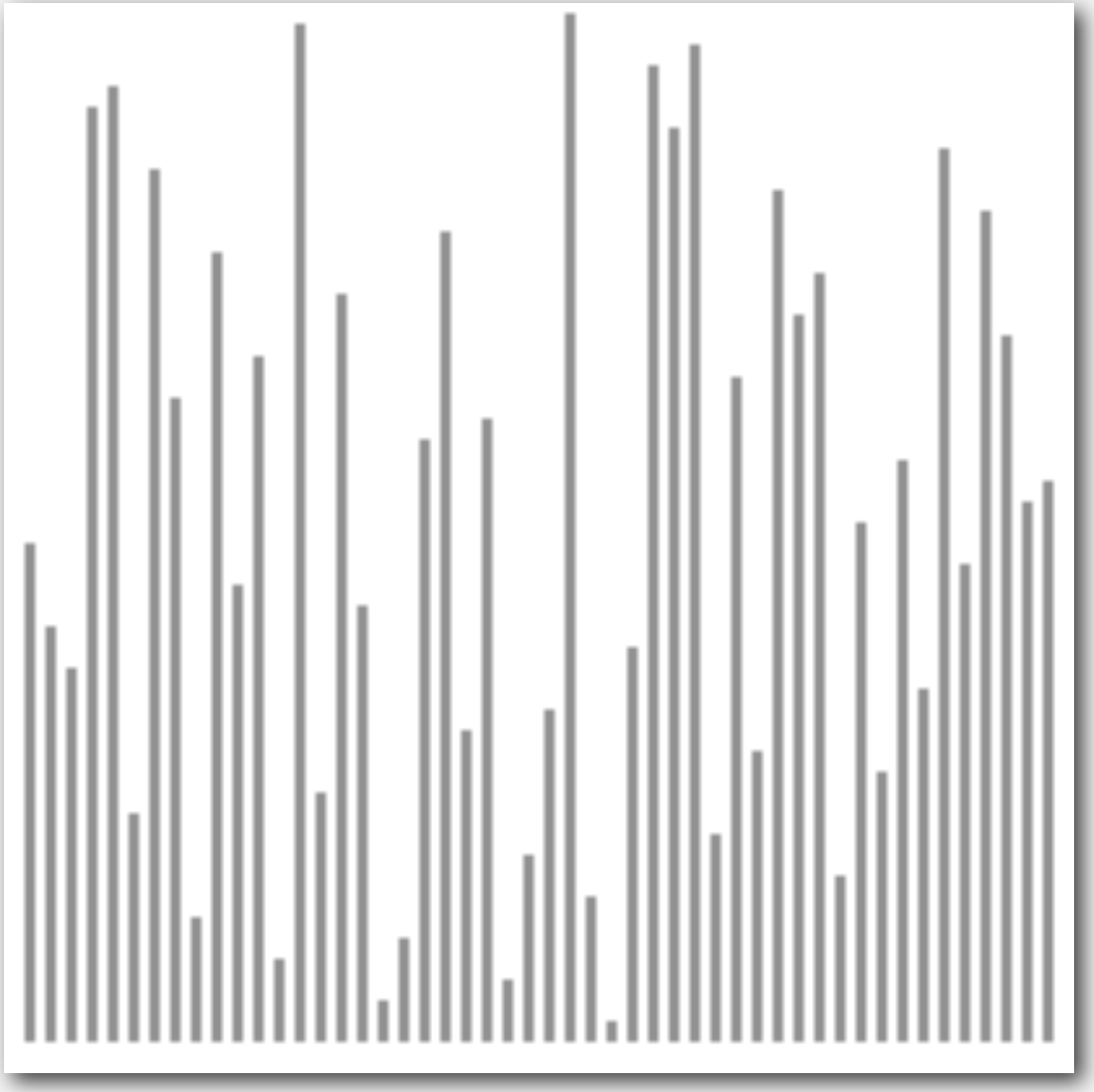
Quicksort trace



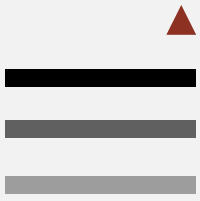
Quicksort trace (array contents after each partition)

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.

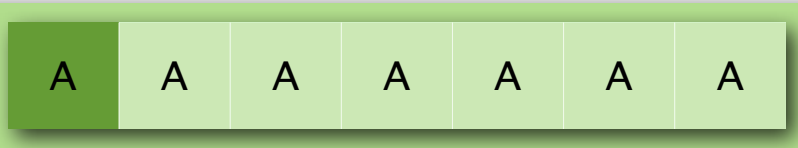
Preserving randomness. Shuffling is needed for performance guarantee.

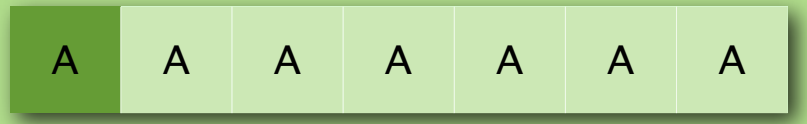
Equal keys. When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item's key.

Equal Keys.

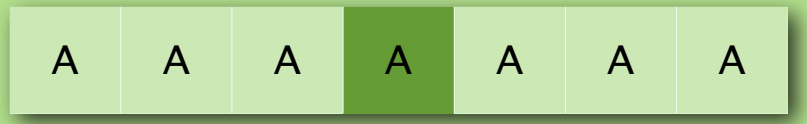
pollEv.com/jhug text to **37607**

Q: What is the result of partitioning:

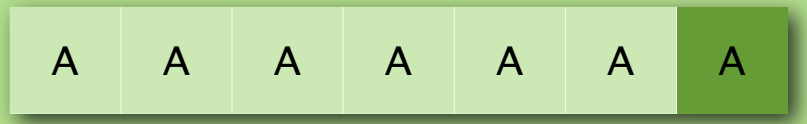




[547848]



[547849]



[547850]

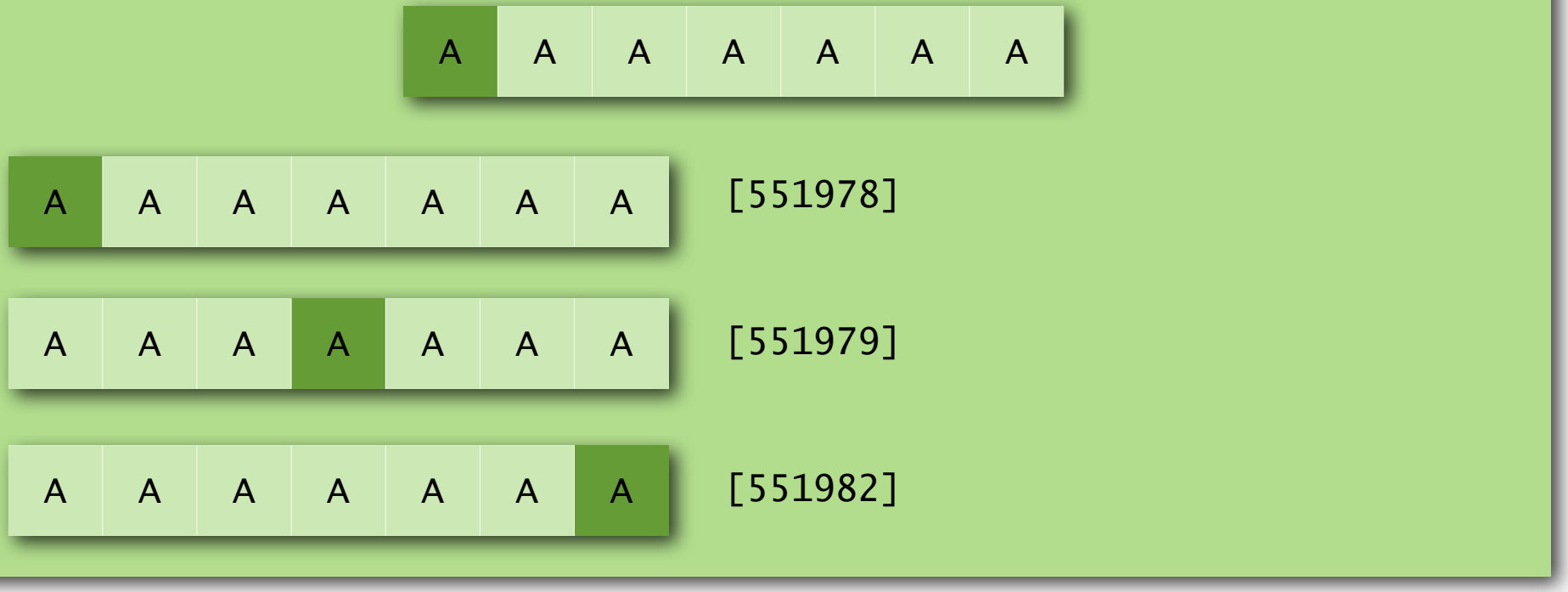
Correct Answer: The middle one!

- 8 total compares
- 3 swaps involving non-pivot As
  - {1, 6}, {2, 5}, {3, 4}
- 1 swap involving the pivot

Equal Keys.

pollEv.com/jhug text to **37607**

Q: What is the result of partitioning if we do not stop on equal values?



Correct Answer: The top one!

- 8 total compares
- 1 swap involving the pivot and itself

Total running time?

- Order of growth: N^2

Quicksort: worst-case analysis

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

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

Quicksort: best-case analysis

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

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

Quicksort: 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 $N \geq 2$:

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

partitioning

left right

partitioning probability

- Multiply both sides by N and collect terms:

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

- 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:

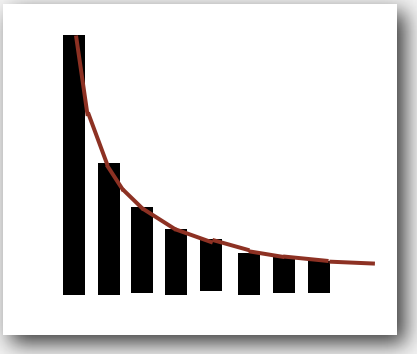
$$\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\end{aligned}$$

previous equation

substitute previous equation

- Approximate sum by an integral:

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



- Finally, the desired result:

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

Quicksort: 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 $N \geq 2$:

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

partitioning

left right

partitioning probability

Shuffling.

- Ensures that this recurrence approximately applies.
- Just how close it gets is a difficult analysis problem (Sedgewick, 1975).

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

Mergesort: 4
Quicksort: 6

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.

- $N + (N - 1) + (N - 2) + \ldots + 1 \sim \frac{1}{2} N^2$.
- 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).

Mergesort: 4
Quicksort: 7

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray

Proposition. Quicksort is not stable.

Pf.

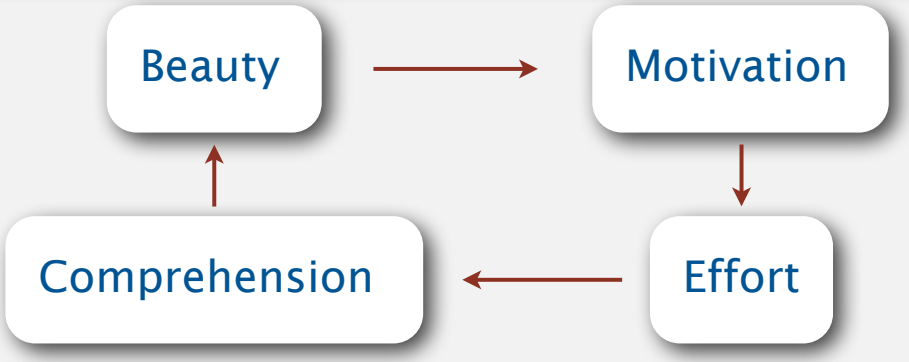
i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

COS226: Quicksort vs. Mergesort

algorithm	Mergesort	Quicksort
Recursion	Before doing work	Do work first
Deterministic	Yes	No
Compares (worst)	N lg N	$N^2 / 2$
Compares (average)		1.39 N lg N
Exchanges (average)	N/A	0.23 N lg N
Stable	Yes	No
Memory Use	N	Constant
Overall Performance	Worse	Better
Meaningless Score	4	7

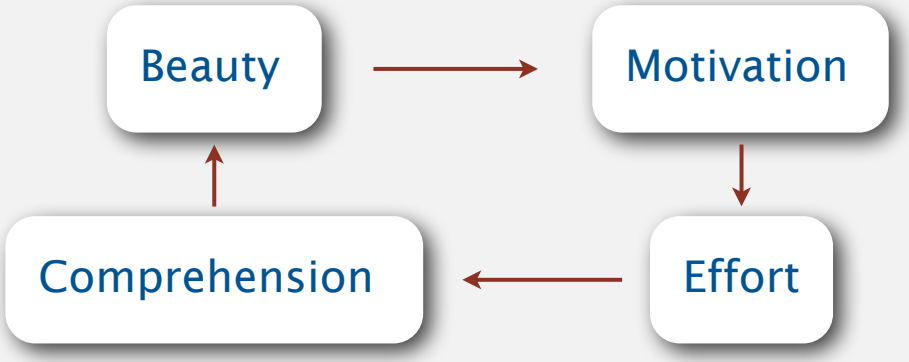
COS226: Quicksort vs. Mergesort

Mergesort	Quicksort
	



COS226: Quicksort vs. Mergesort

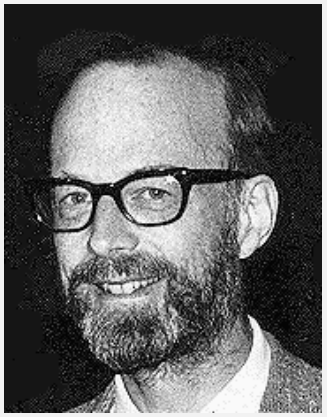
"Mathematics, rightly viewed, possesses not only truth, but supreme beauty — a beauty cold and austere, like that of sculpture, without appeal to any part of our weaker nature, without the gorgeous trappings of painting or music, yet sublimely pure, and capable of a stern perfection such as only the greatest art can show. The true spirit of delight, the exaltation, the sense of being more than Man, which is the touchstone of the highest excellence, is to be found in mathematics as surely as poetry" — Bertrand Russell (The Study of Mathematics. 1919)



Quicksort Inventor.

Tony Hoare.

- QuickSort invented in 1960 at age 26
  - Used to help with machine translation project
- Also invented the **null-pointer**
- 4 honorary doctorates
- 1 real doctorate
- Knight



Sir Charles Antony Richard Hoare
1980 Turing Award

> *" I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. "* — *Tony Hoare (2009)*

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 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.

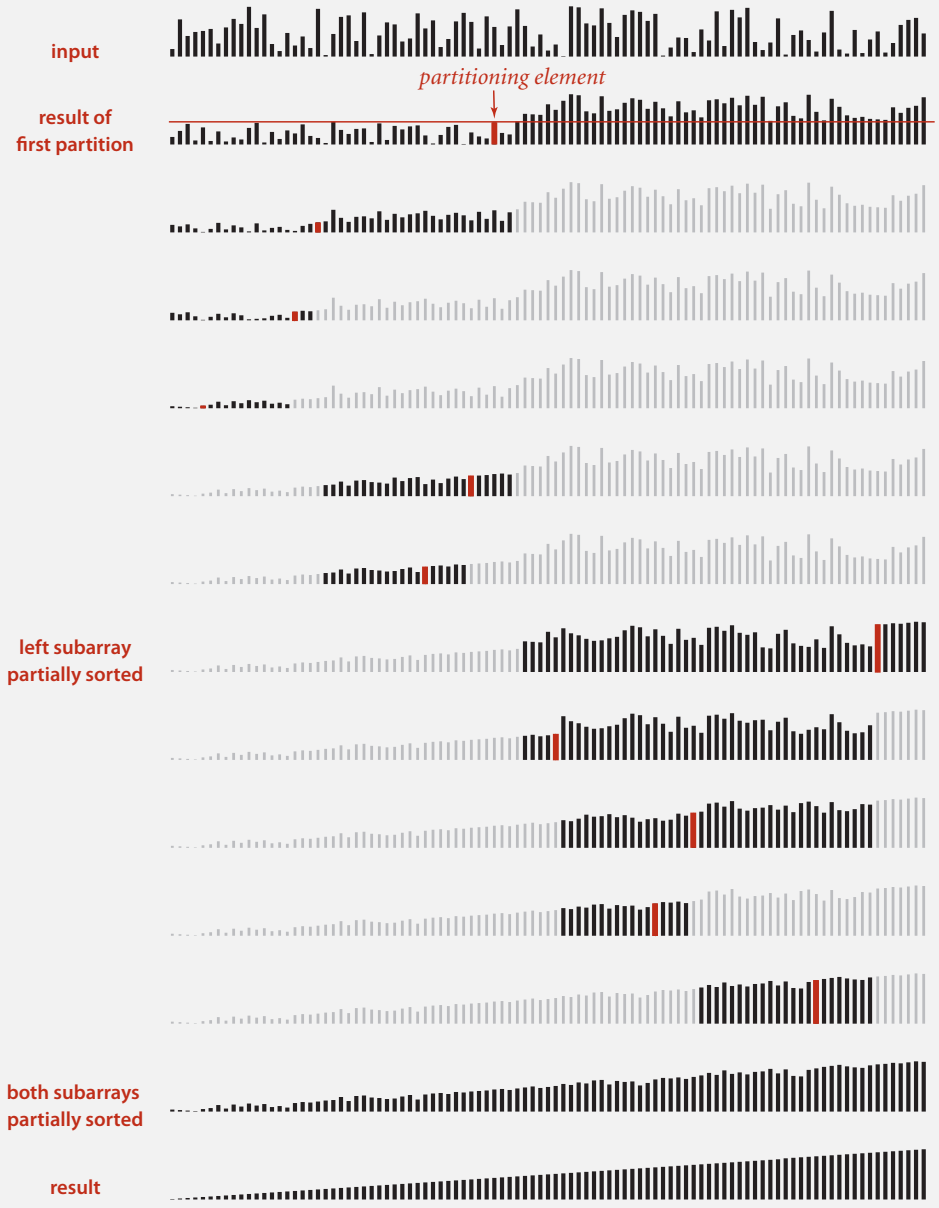
~ 12/7 N ln N compares (slightly fewer)
~ 12/35 N ln N exchanges (slightly more)

```
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



http://algs4.cs.princeton.edu

2.3 Quicksort

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

Selection

Goal. Given an array of N items, find the k^{th} largest.

Ex. Min $(k = N - 1)$, max $(k = 0)$, 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?

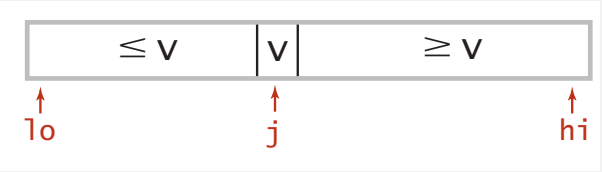
Which is true?

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

Thought Experiment and Consciousness Check

Suppose.

- We have an unsorted array a[] of length 37
- We want to find the median
- We partition on a[0] and it ends up at j=14



pollEv.com/jhug text to **37607**

Q: Where in the array is the median now?

A. Between 0 and 13 [494513]
B. At 14 [494535]
C. Between 15 and 36 [494536]
D. Somewhere that we can find in constant time [494539]

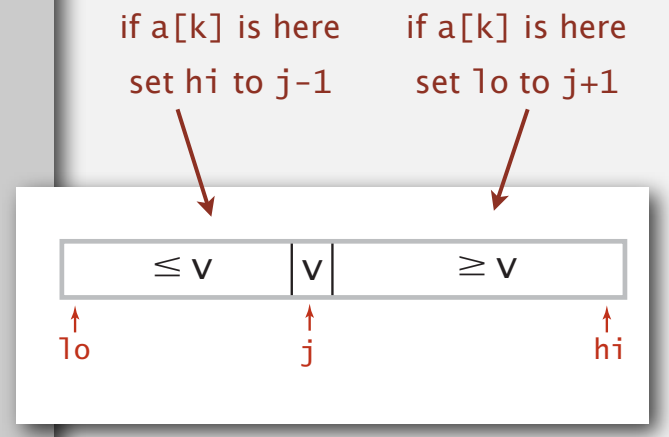
Quick-select

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: $N + N/2 + N/4 + \ldots + 1 \sim 2N$ compares.
- Formal analysis similar to quicksort analysis yields:

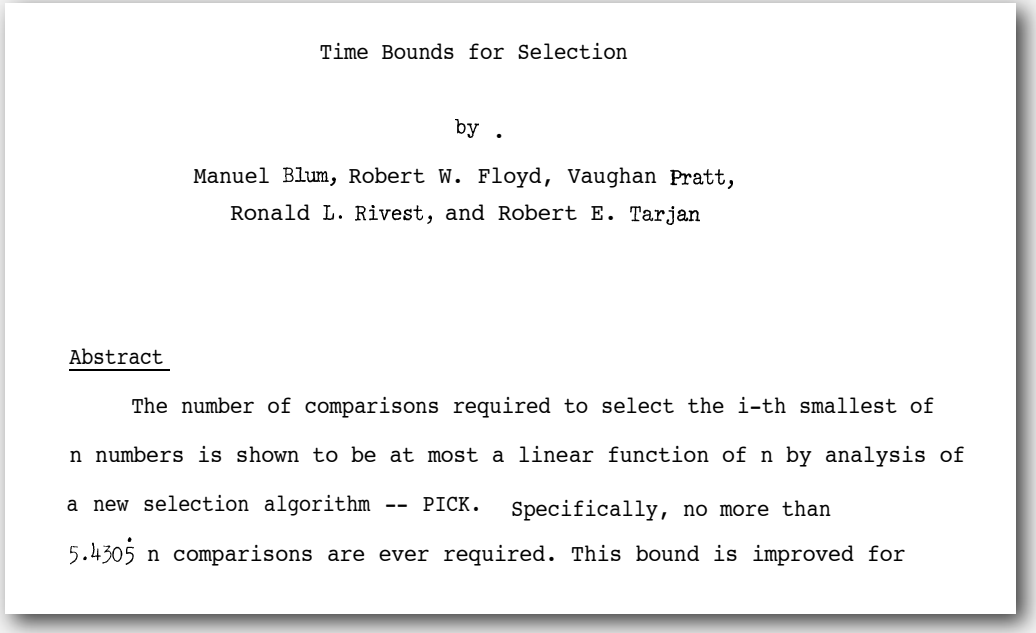
$$C_N = 2N + 2k \ln (N/k) + 2(N-k) \ln (N/(N-k))$$

(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.



Time Bounds for Selection

by .

Manuel Blum, Robert W. Floyd, Vaughan Pratt,
Ronald L. Rivest, and Robert E. Tarjan

Abstract

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm -- PICK. Specifically, no more than 5.4305 n comparisons are ever required. This bound is improved for

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.

http://algs4.cs.princeton.edu

2.3 QUICKSORT

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

War story (C qsort function)

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

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

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

Here are the timings on our machine:
$ time a.out 2000
real    5.85s
$ time a.out 4000
real   21.64s
$time a.out 8000
real   85.11s
```

War story (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.

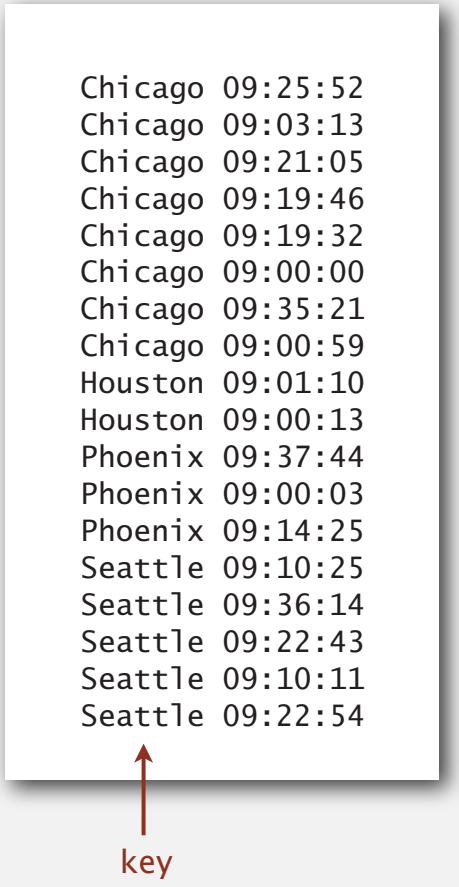
Duplicate keys

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.
- Place children in magical residential colleges.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.

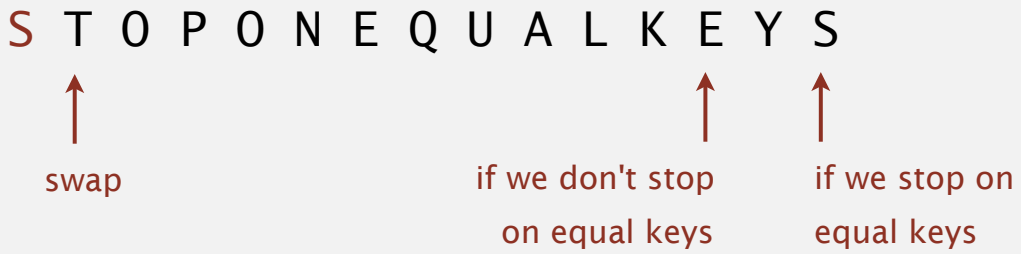


Duplicate keys

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!

which is why ours does!
(but many textbook implementations do not)



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

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

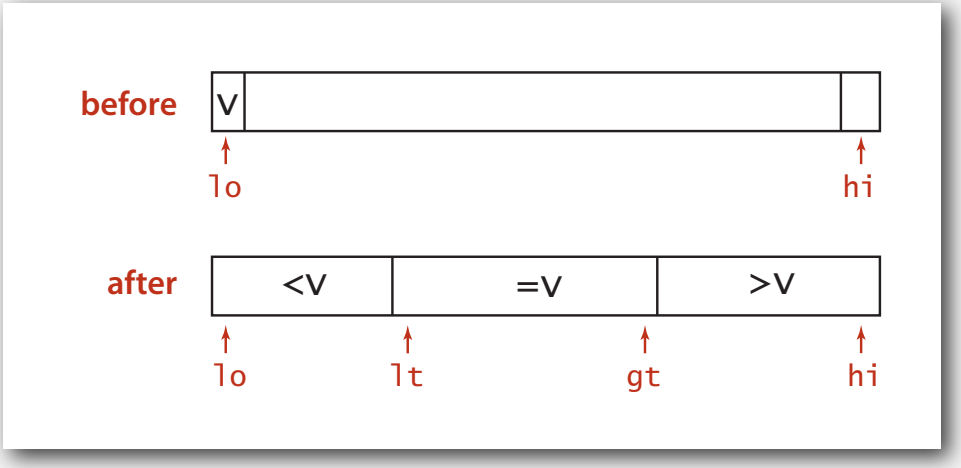
A A A B B B B B C C C

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`.



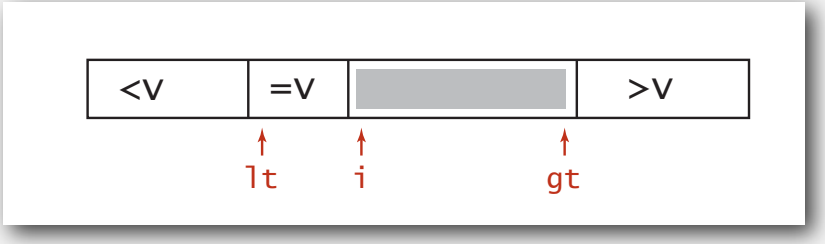
Dutch national flag problem. [Edsger Dijkstra]

- 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

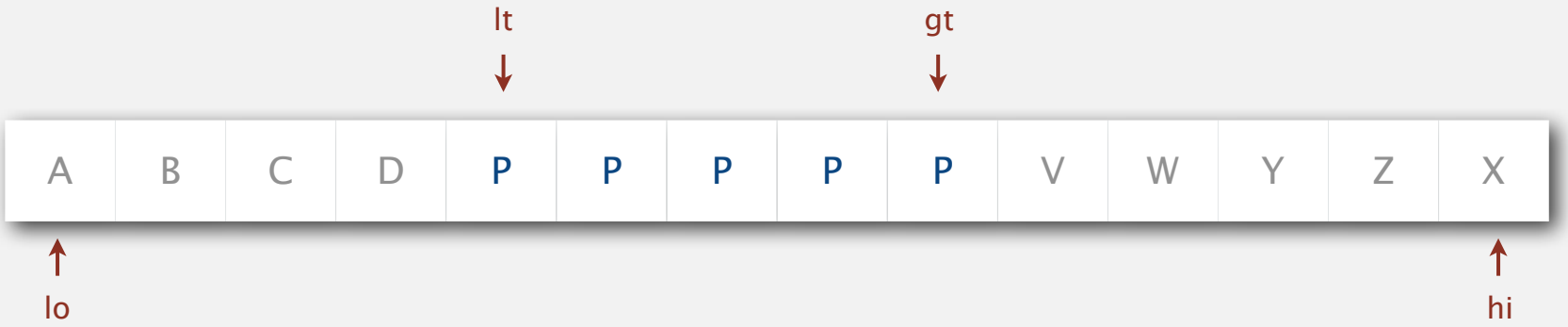
- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i] < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i] > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

invariant

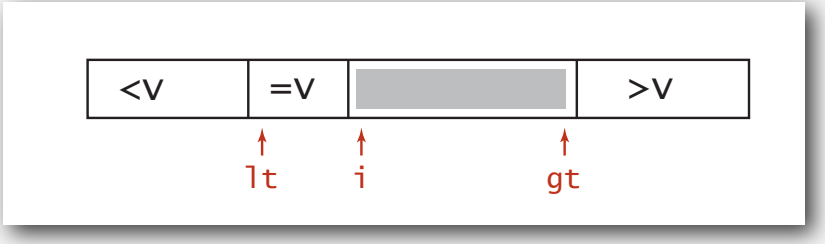


Dijkstra 3-way partitioning demo

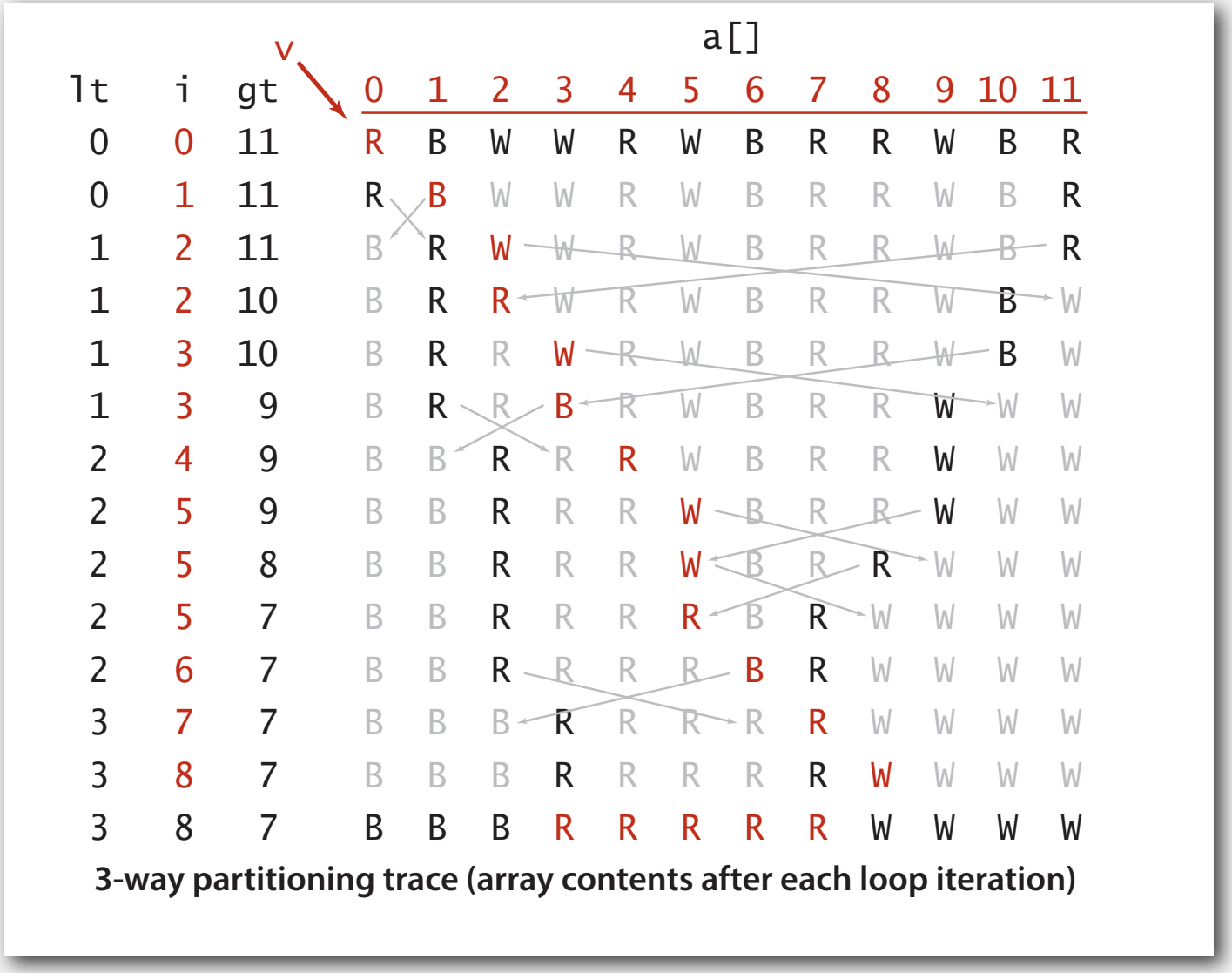
- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i] < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i] > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`



invariant



Dijkstra's 3-way partitioning: trace

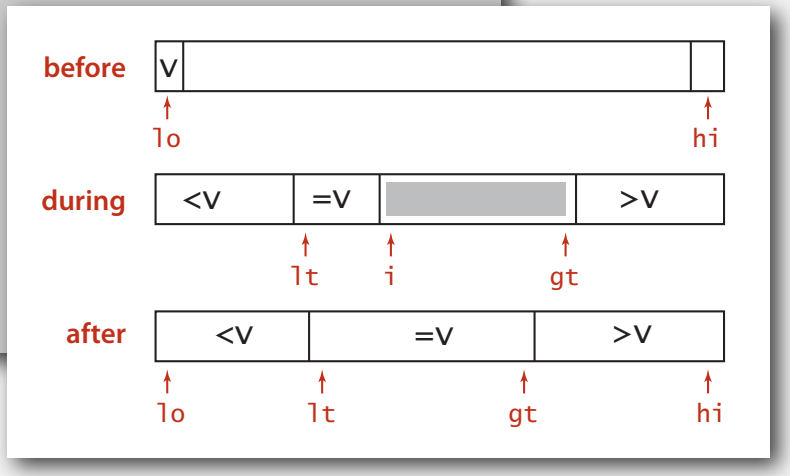


3-way partitioning trace (array contents after each loop iteration)

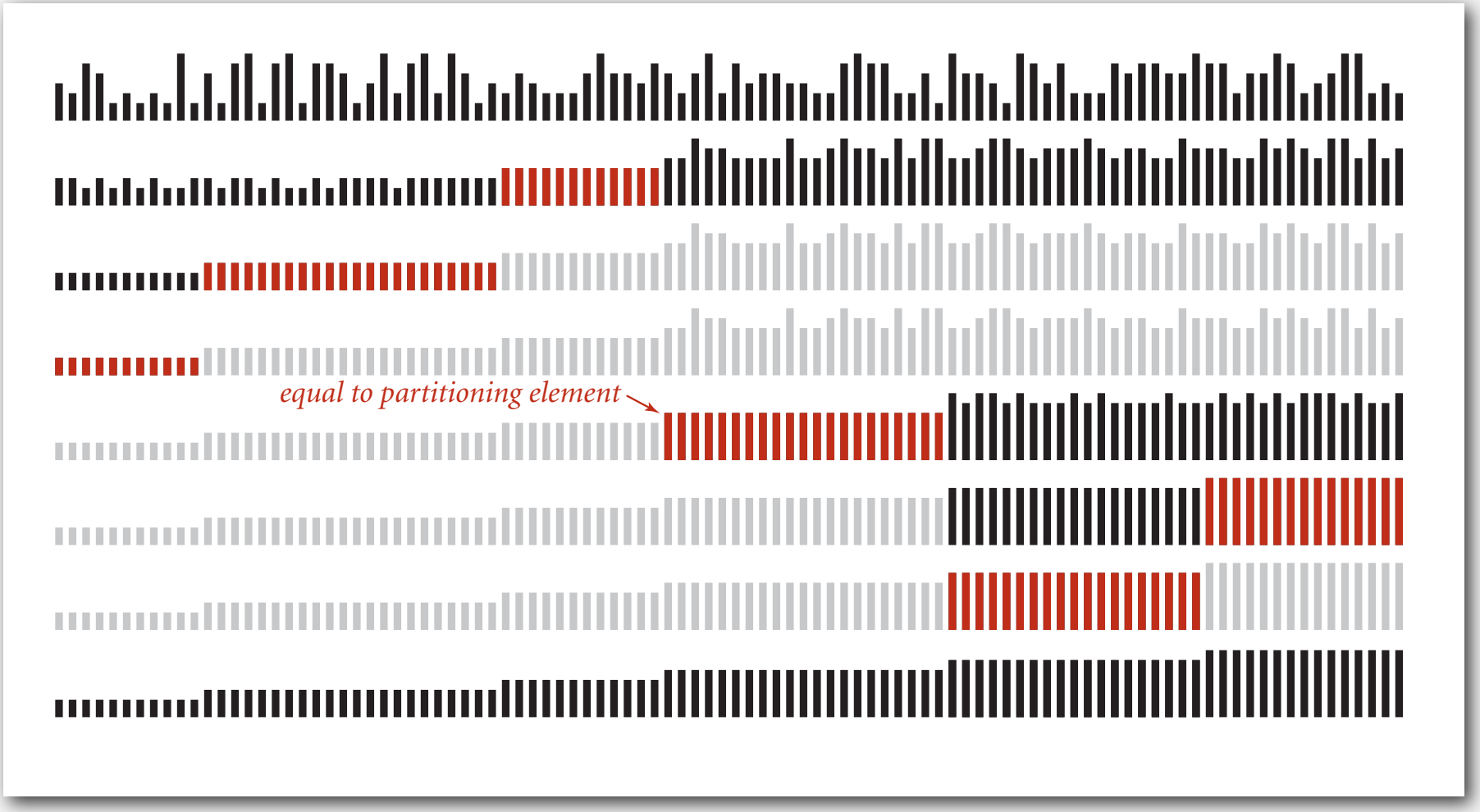
3-way quicksort: Java implementation

```
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^{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]
Quicksort with 3-way partitioning is entropy-optimal.

proportional to lower bound

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.

http://algs4.cs.princeton.edu

2.3 Quicksort

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

Sorting applications

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS feed in reverse chronological order.

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

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]);
   }
}
```

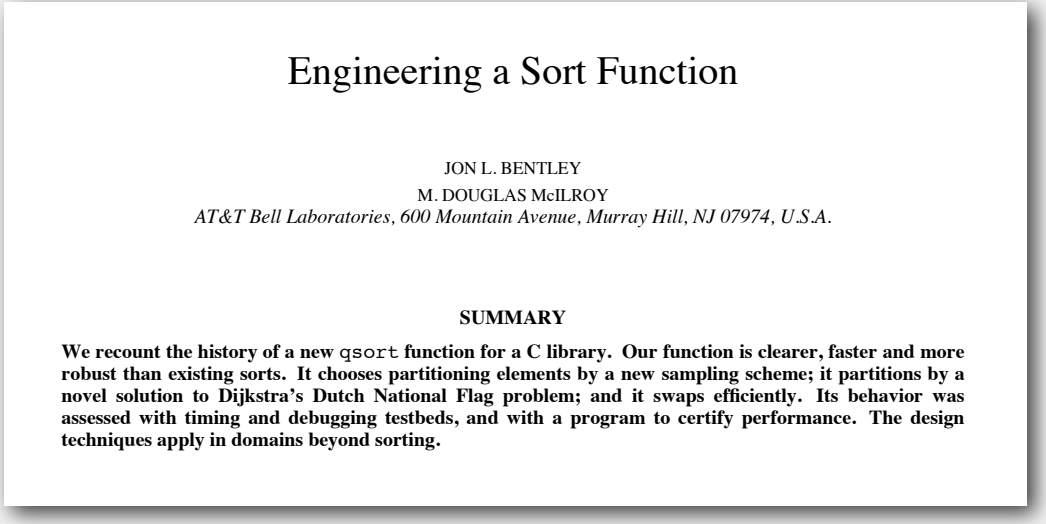
Q. Why use different algorithms for primitive and reference types?

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]

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

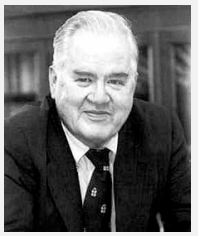
We recount the history of a new `qsort` function for a C library. Our function is clearer, faster and more robust than existing sorts. It chooses partitioning elements by a new sampling scheme; it partitions by a novel solution to Dijkstra's Dutch National Flag problem; and it swaps efficiently. Its behavior was assessed with timing and debugging testbeds, and with a program to certify performance. The design techniques apply in domains beyond sorting.

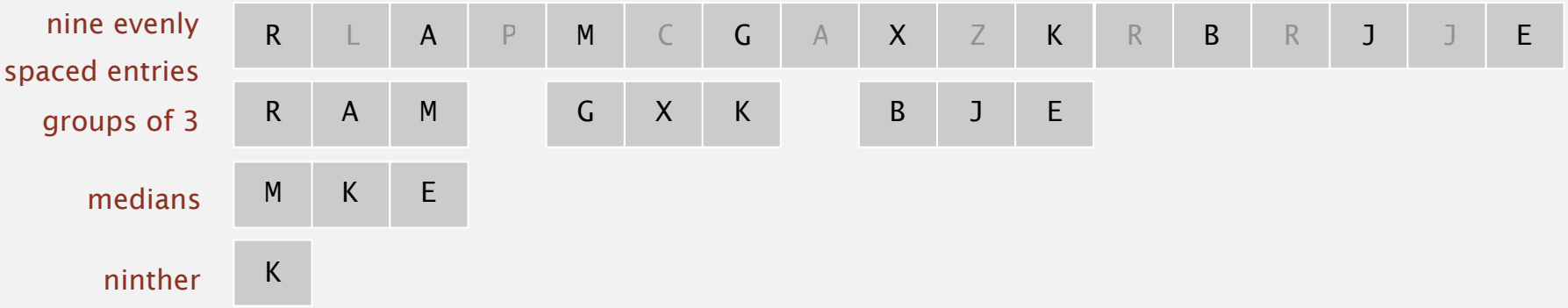
Now very widely used. C, C++, Java 6, ….

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)

McIlroy's devious idea. [A Killer Adversary for Quicksort]

- Construct malicious input on the fly while running system quicksort, in response to the sequence of keys compared.
- Make partitioning item compare low against all items not seen during selection of partitioning item (but don't commit to their relative order).
- Not hard to identify partitioning item.

Consequences.

- Can be used to generate worst case inputs for deterministic sorts.
- Algorithmic complexity attack: you enter linear amount of data; server performs quadratic amount of work.

Good news. Attack is not effective if `sort()` shuffles input array.

Q. Why do you think `Arrays.sort()` is deterministic?

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

Which sorting algorithm to use?

Many sorting algorithms to choose from:

Internal (in-memory) sorts.

- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splaysort, dual-pivot quicksort, timsort, ...

External sorts. Poly-phase mergesort, cascade-merge, psort, ...

String/radix sorts. Distribution, MSD, LSD, 3-way string quicksort.

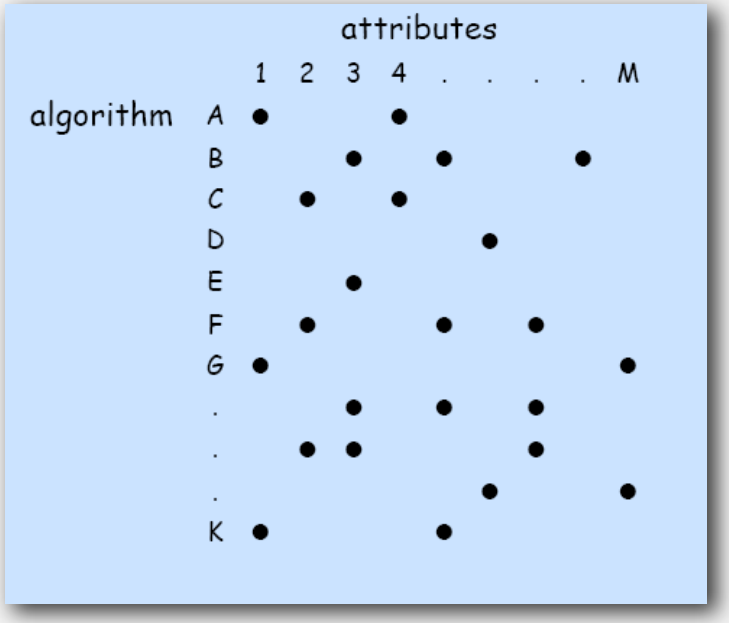
Parallel sorts.

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

Which sorting algorithm to use?

Applications have diverse attributes.

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



many more combinations of attributes than algorithms

Elementary sort may be method of choice for some combination but cannot cover all combinations of attributes.

Q. Is the system sort good enough?

A. Usually.