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]

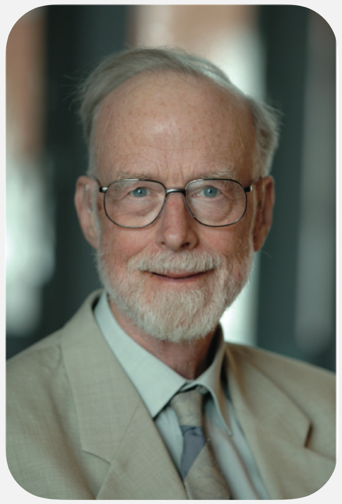
...

Quicksort t-shirt

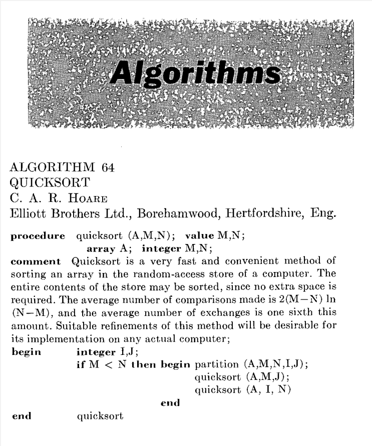
A brief history

Tony Hoare.

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



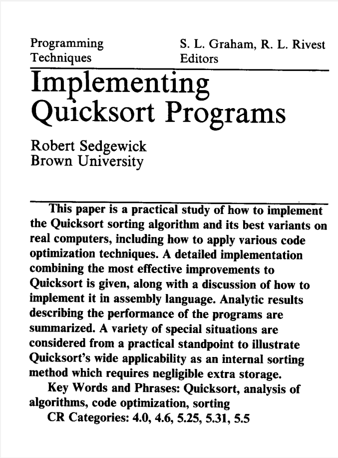
Tony Hoare
1980 Turing Award



Algorithms

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
```



Programming Techniques — S. L. Graham, R. L. Rivest, Editors

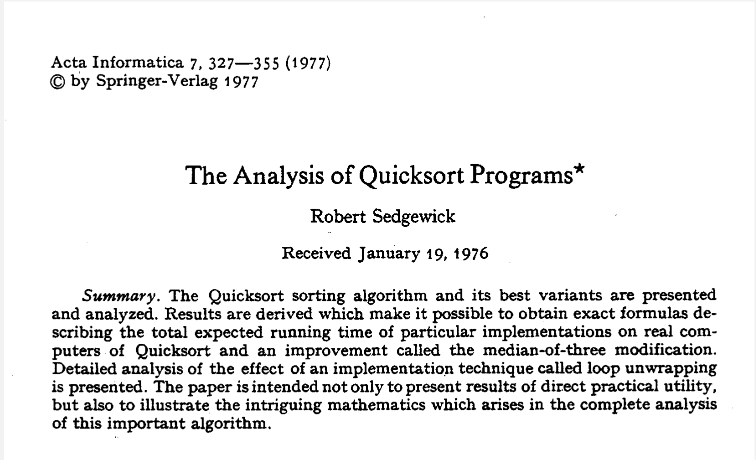
Implementing Quicksort Programs

Robert Sedgewick
Brown University

This paper is a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quicksort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quicksort's wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quicksort, analysis of algorithms, code optimization, sorting

CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5



Acta Informatica 7, 327—355 (1977)
© by Springer-Verlag 1977

The Analysis of Quicksort Programs*

Robert Sedgewick

Received January 19, 1976

Summary. The Quicksort sorting algorithm and its best variants are presented and analyzed. Results are derived which make it possible to obtain exact formulas describing the total expected running time of particular implementations on real computers of Quicksort and an improvement called the median-of-three modification. Detailed analysis of the effect of an implementation technique called loop unwrapping is presented. The paper is intended not only to present results of direct practical utility, but also to illustrate the intriguing mathematics which arises in the complete analysis of this important algorithm.

Bob Sedgewick.

- Refined and popularized quicksort.
- 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 `j`

- Entry `a[j]` is in place.
- 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

partitioning item

not greater

not less

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]`.

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
lo	i														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]`.

E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
↑ lo					↑ j										↑ hi

partitioned!

Quicksort quiz 1

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

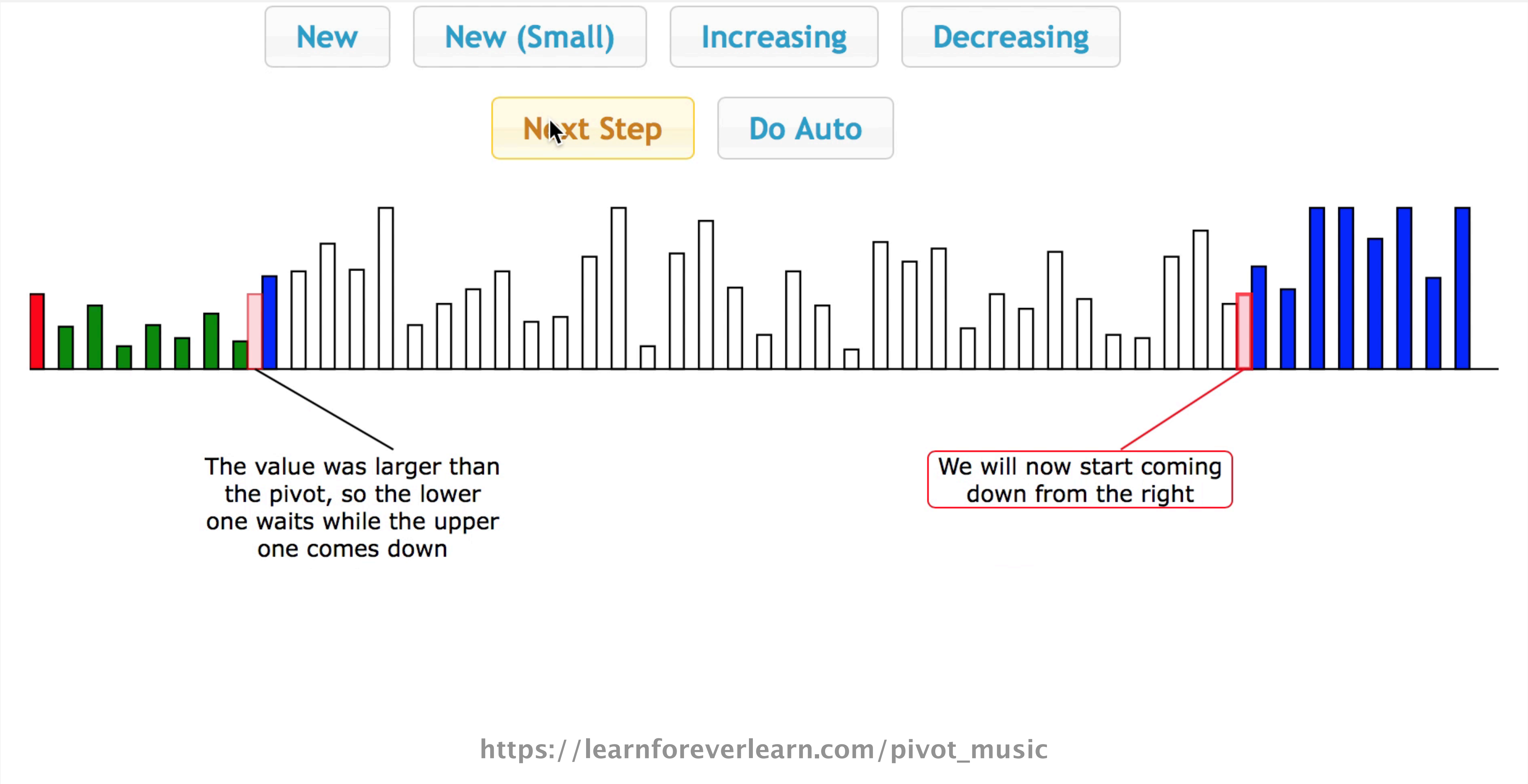
A. $\sim \frac{1}{2} n$ and $\sim \frac{1}{2} n$

B. $\sim \frac{1}{2} n$ and $\sim n$

C. $\sim n$ and $\sim \frac{1}{2} n$

D. $\sim n$ and $\sim n$

The music of quicksort partitioning (by Brad Lyon)



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

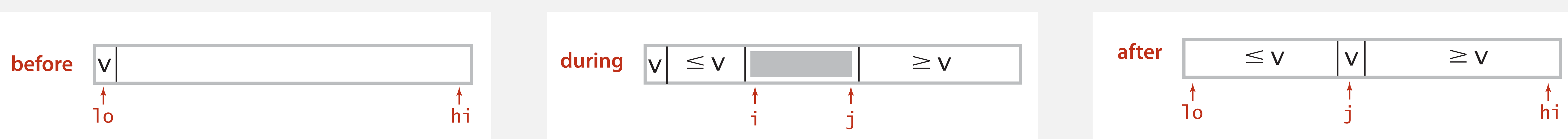
check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place

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



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)

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

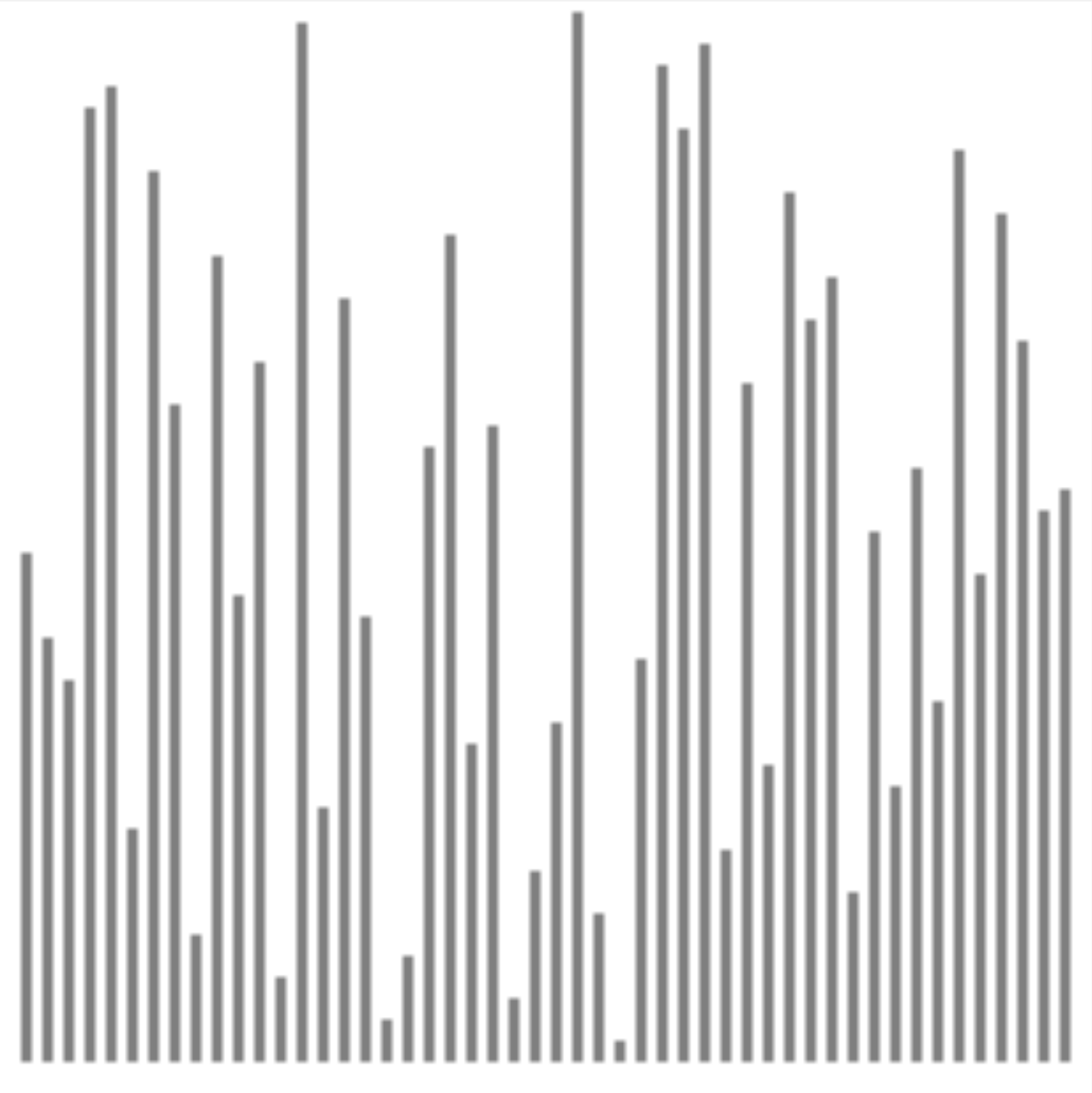
Quicksort trace

	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
no partition for subarrays of size 1	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



▲ algorithm position
in order
current subarray
not in order

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

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 trickier than it appears.

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

Preserving randomness. Shuffling is needed for performance guarantee.
Equivalent alternative. Pick a random partitioning item in each subarray.

Quicksort: empirical analysis (1962)

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: 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

after random shuffle

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

after random shuffle

Good news. Worst case for quicksort is 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 $n \log n$ time.

```
public static void f(int n)
{
    if (n == 0) return;
    f(n/2);          ← solve two problems
    f(n/2);          ← of half the size
    linear(n);       ← do a linear amount of work
}
```

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

"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 $n \geq 2$:

$$C_n \;=\; \underbrace{(n+1)}_{\text{partitioning}} \;+\; \left(\frac{C_0 + C_{n-1}}{n}\right) \;+\; \left(\frac{\overset{\text{left}}{C_1} + \overset{\text{right}}{C_{n-2}}}{n}\right) \;+\; \ldots \;+\; \left(\frac{C_{n-1} + C_0}{n}\right)$$

partitioning probability

- Multiply both sides by n and collect terms:

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

- 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

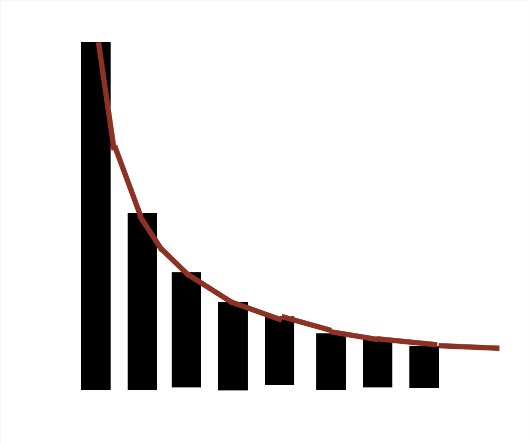
- Repeatedly apply previous equation:

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

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:

$$\boxed{C_n \sim 2\,(n+1)\ln n \approx 1.39\,n \lg n}$$

Quicksort: performance characteristics

Quicksort performance summary.

39% more than mergesort

- Expected number of compares is $\sim 1.39\, n \lg n$.
- Minimum number of compares is $\sim n \lg n$. ← never fewer than mergesort
- Maximum number of compares is $\sim \frac{1}{2} n^2$. ← but never happens
- Faster than mergesort in practice because of less data movement.

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.

Pf.

- Partitioning: constant extra space.
- Function-call stack: 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.

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.

$\sim$ $12/7 \;\; n \ln n$ compares (14% fewer)
$\sim$ $12/35 \; n \ln n$ 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);
}
```

https://algs4.cs.princeton.edu

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 theory as a guide.

- Easy $n \log n$ upper bound. How?
- Easy n upper bound for $k = 0, 1, 2$. 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?]

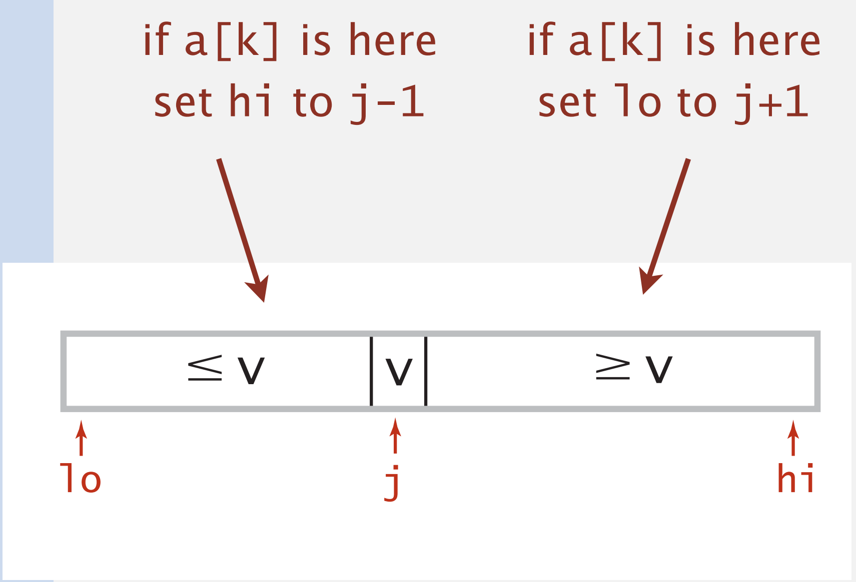
Quick-select

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.

Recall. Any algorithm with the following structure takes linear time.

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

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

"close enough"

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

Formal analysis.

$$\begin{aligned} C_n &= 2\,n + 2\,k \ln(n/k) + 2\,(n-k)\ln(n/(n-k)) \\ &\le (2 + 2\ln 2)\,n \\ &\approx 3.38\,n \end{aligned}$$

max occurs for median $(k = n/2)$

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***
>
> 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 $5.430\dot{5}\, n$ 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.

Remark. Constants are 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).

https://algs4.cs.princeton.edu

2.3 Quicksort

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

Duplicate keys

Often, purpose of sort is to bring items with equal keys together.

- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.

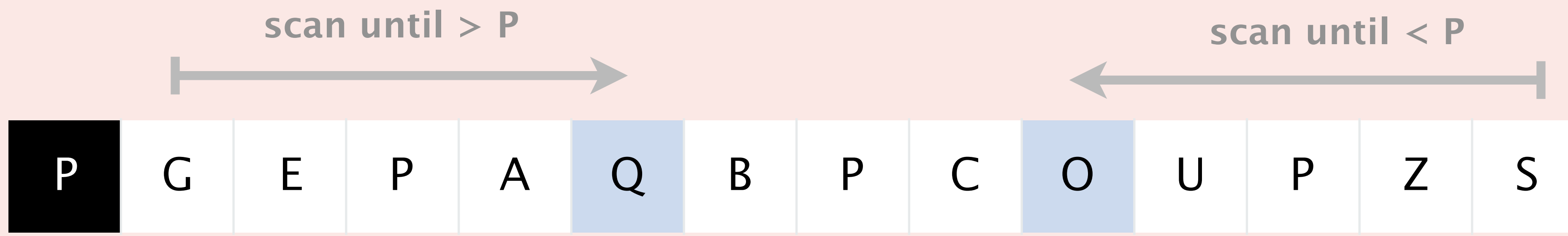
```
Chicago 09:25:52
Chicago 09:03:13
Chicago 09:21:05
Chicago 09:19:46
Chicago 09:19:32
Chicago 09:00:00
Chicago 09:35:21
Chicago 09:00:59
Houston 09:01:10
Houston 09:00:13
Phoenix 09:37:44
Phoenix 09:00:03
Phoenix 09:14:25
Seattle 09:10:25
Seattle 09:36:14
Seattle 09:22:43
Seattle 09:10:11
Seattle 09:22:54
```

key

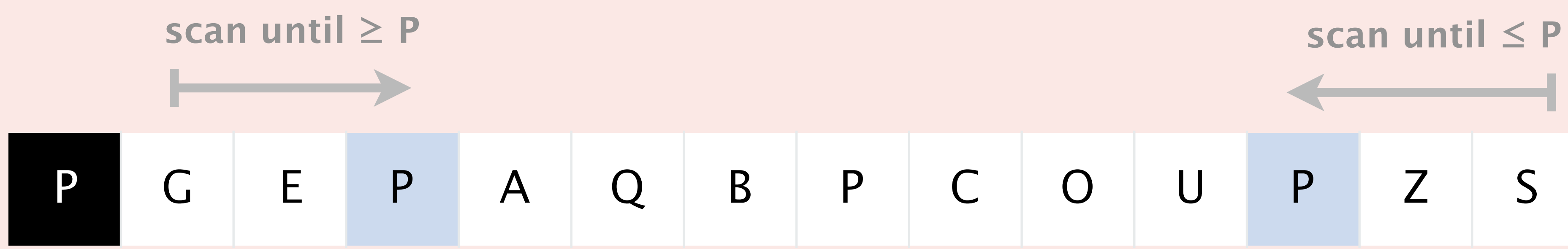
Quicksort: quiz 2

When partitioning, how should we handle keys equal to partitioning key?

A.



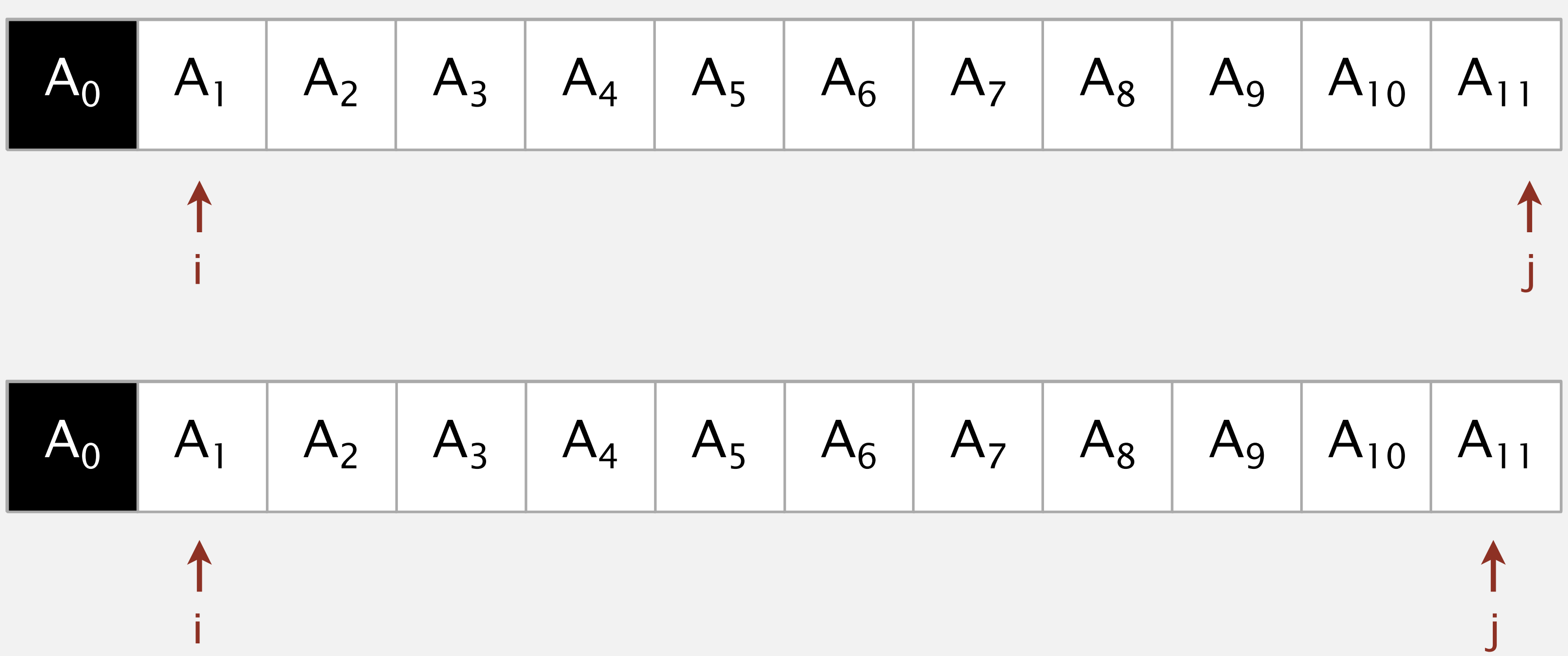
B.



C. Either A or B.

War story (system sort in C)

Bug. A `qsort()` call in C that should have taken seconds was taking minutes to sort a random array of 0s and 1s.



skip over equal keys

stop on equal keys

Duplicate keys: partitioning strategies

Bad. Don't stop scans on equal keys.

[quadratic number of 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

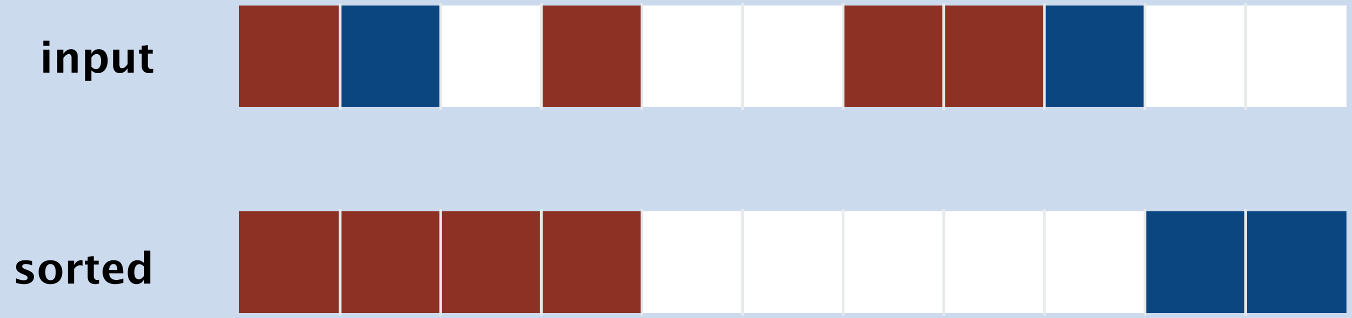
Better. Put all equal keys in place. How?

[$\sim n$ 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

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.
- $color(i)$: color of pebble in bucket i.

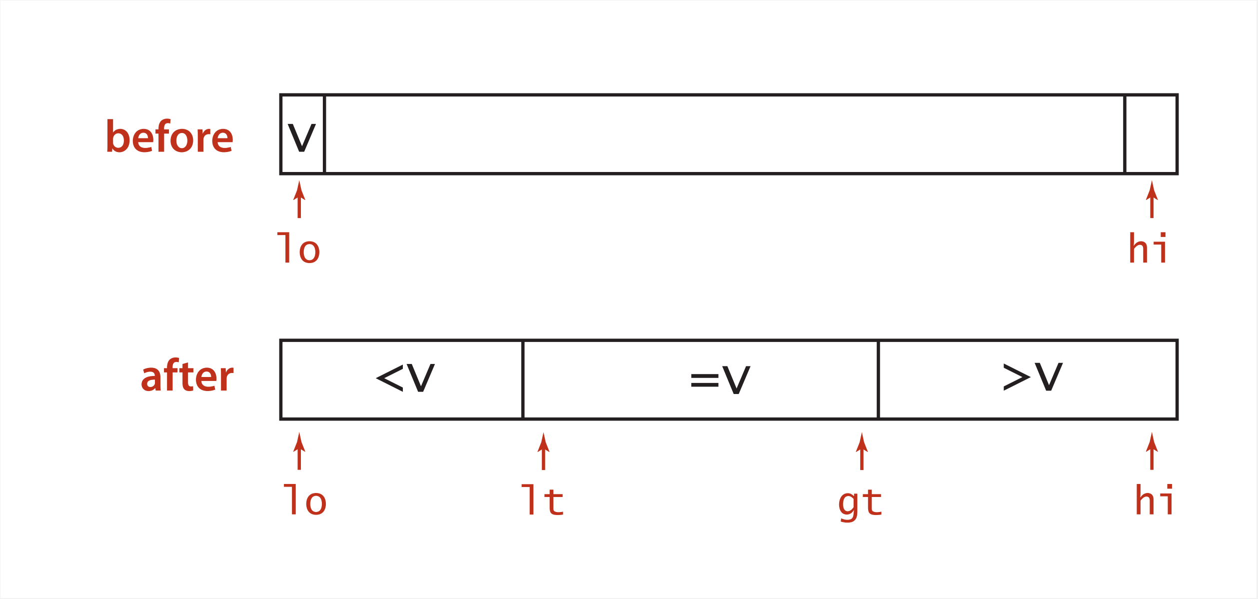
Requirements.

- Exactly n calls to $color()$.
- At most n calls to $swap()$.
- Constant extra space.

3-way partitioning

Goal. Partition array into three parts so that:

- White: entries between `lt` and `gt` equal to the partition item.
- Red: smaller entries to left of `lt`.
- Blue: larger entries to the right of `gt`.



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 algorithm: demo

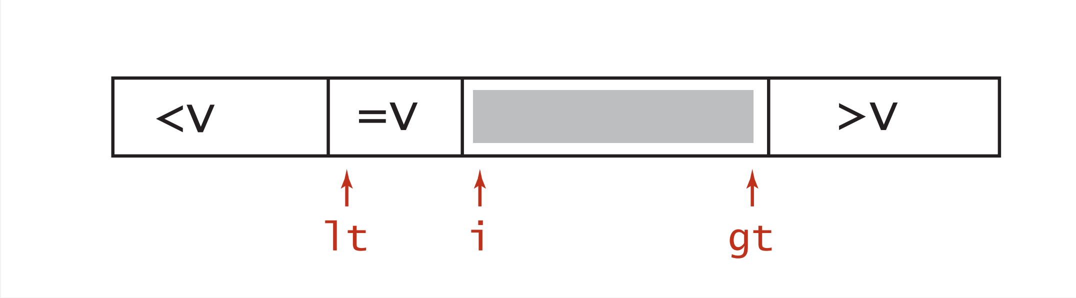
- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

lt i gt

P_1	D	B	X	W	P_2	P_3	V	P_4	A	P_5	C	Y	Z

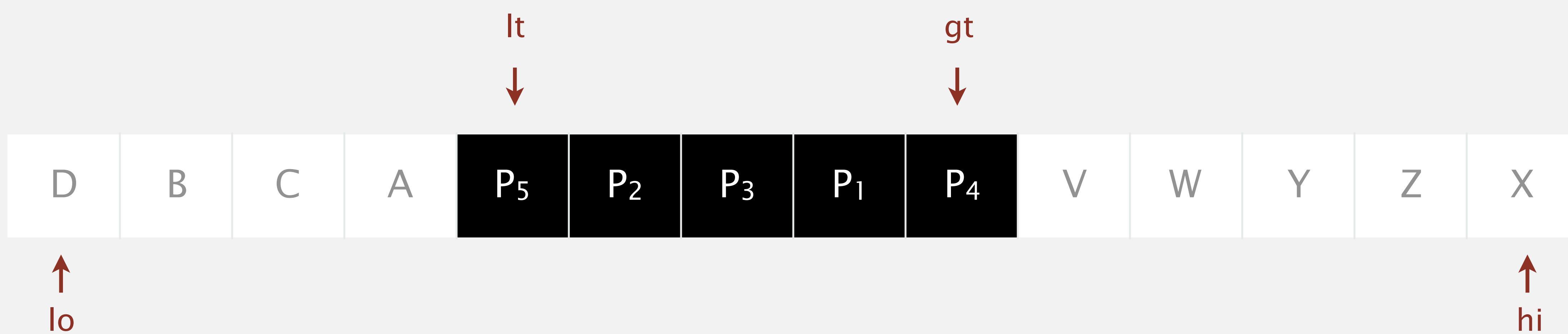
lo hi

invariant

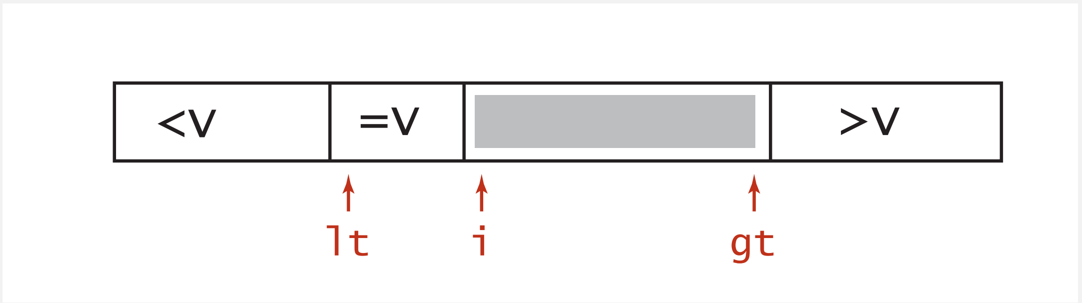


Dijkstra's 3-way partitioning algorithm: demo

- 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`

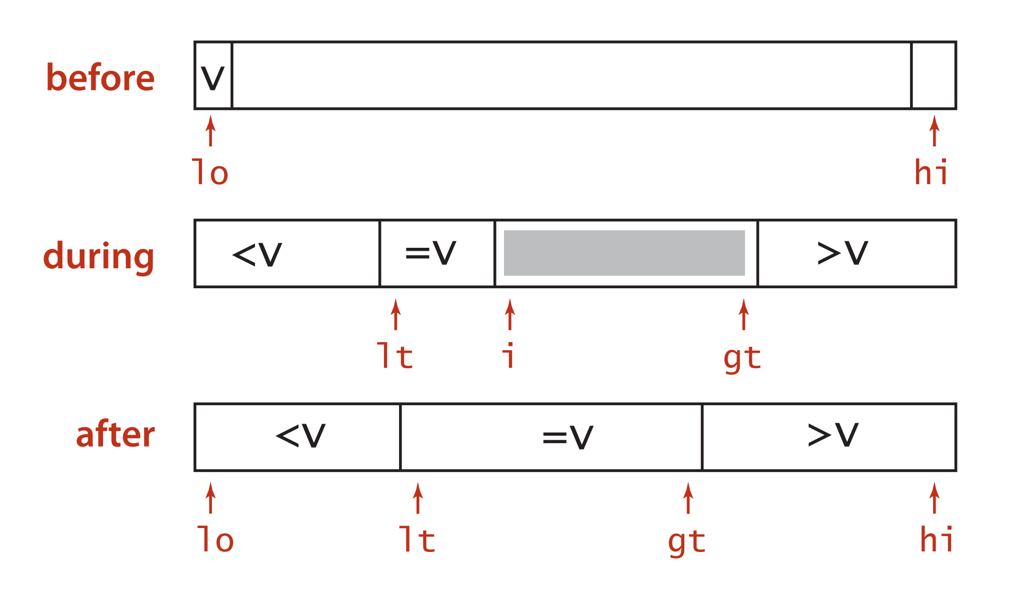


invariant

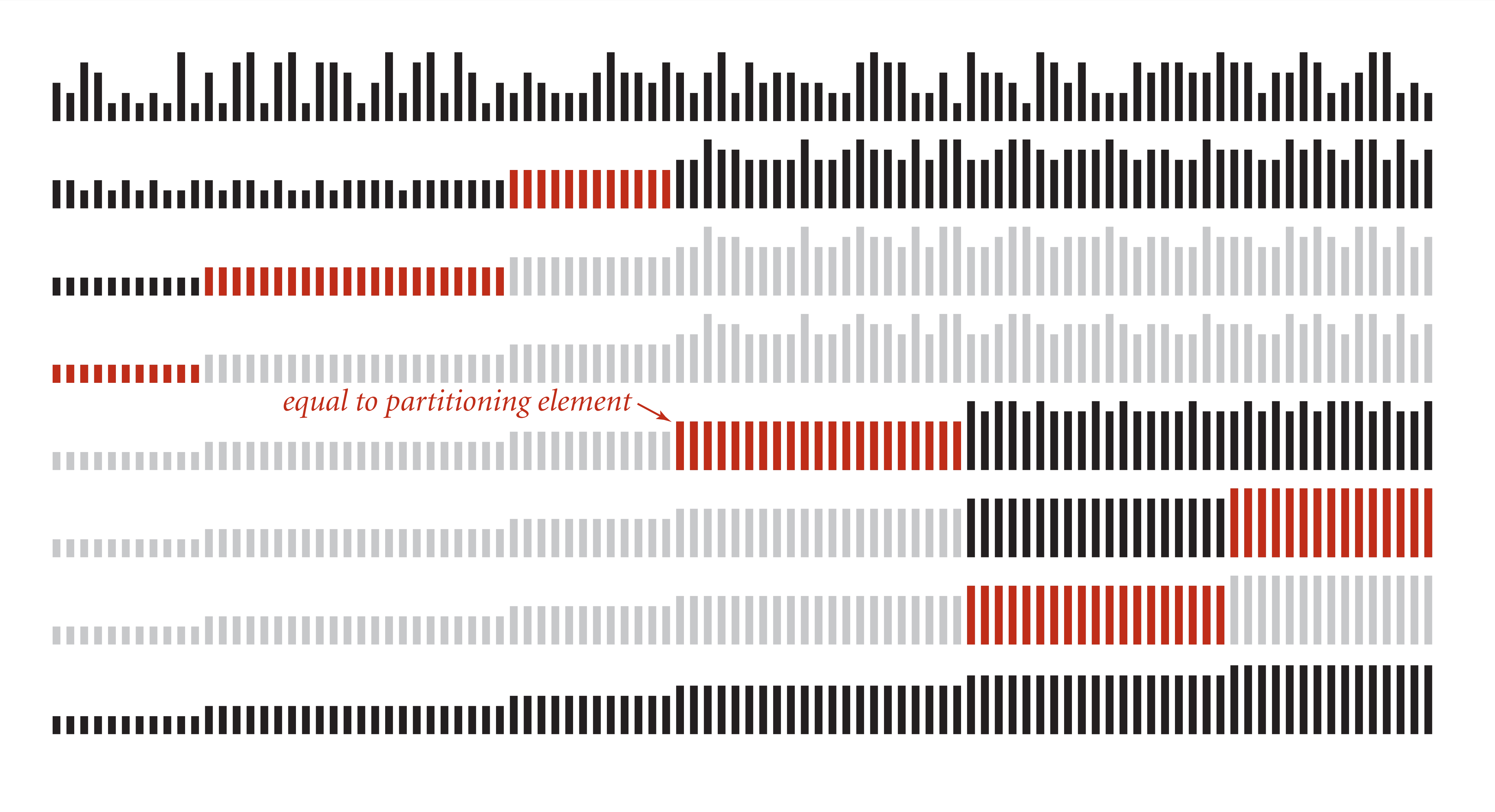


3-way quicksort: Java implementation

```
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);
}
```



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 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 when duplicate keys
?	✔	✔	n	$n \lg n$	$n \lg n$	holy sorting grail

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.

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

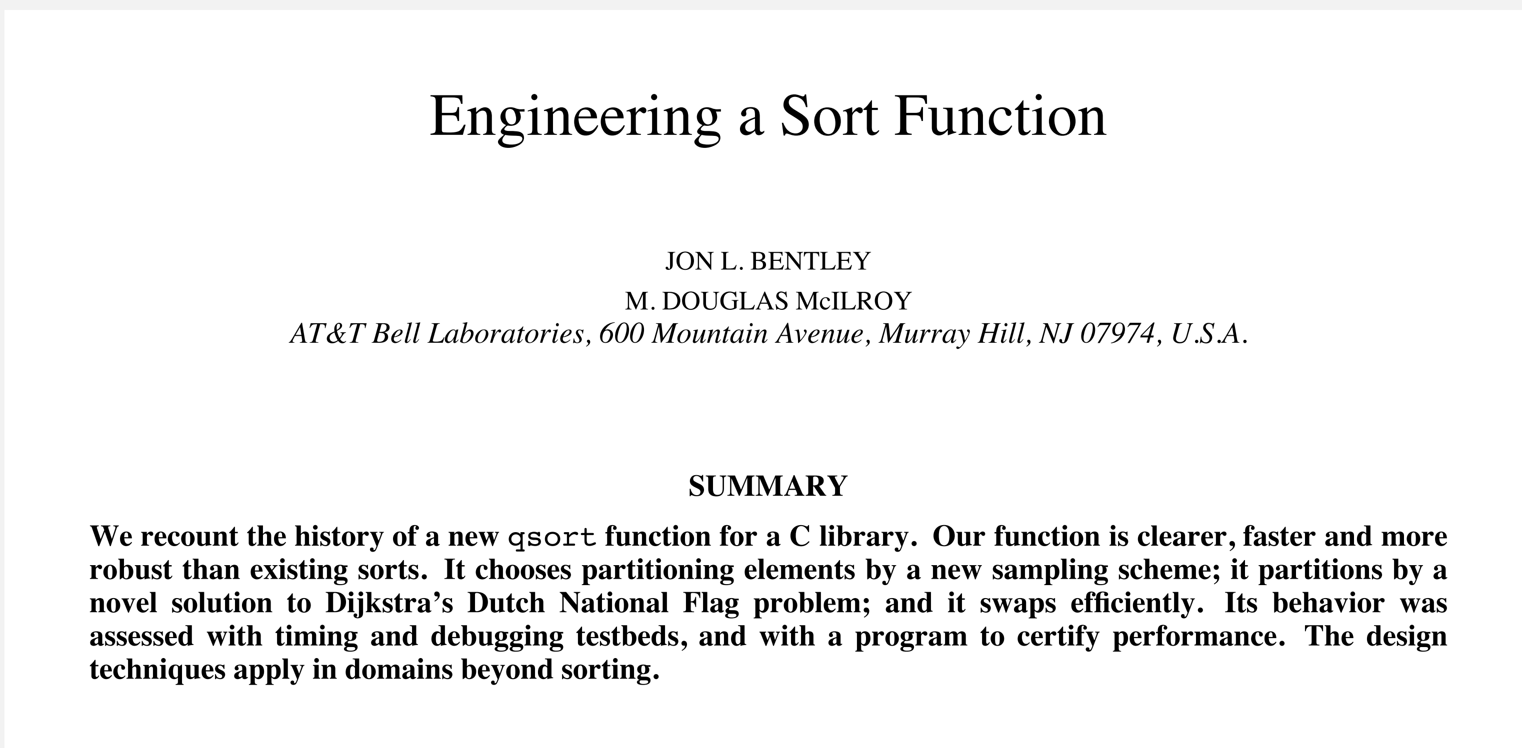
Engineering a system sort (in 1993)

Bentley–McIlroy quicksort.

sample 9 items

- Cutoff to insertion sort for small subarrays.
- Partitioning item: median of 3 or Tukey's ninther.
- Partitioning scheme: Bentley–McIlroy 3-way partitioning.

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.

Very widely used. 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 <= 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 ]

...
```

http://mail.openjdk.java.net/pipermail/core-libs-dev/2009-September/002630.html

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

Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

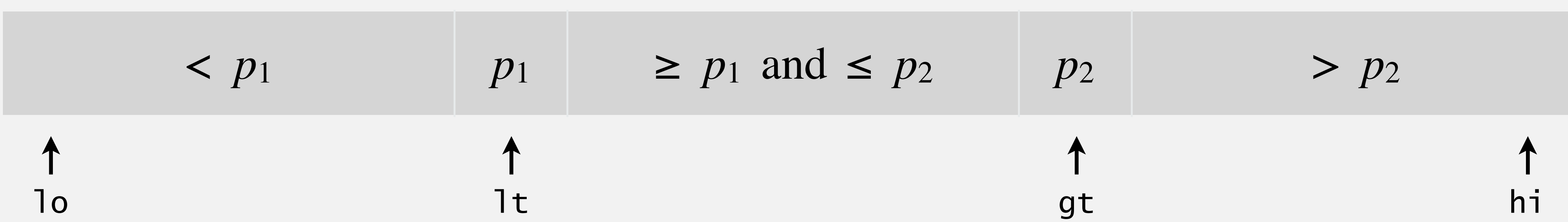
```
Date: Thu, 29 Oct 2009 11:19:39 +0000
Subject: Replace quicksort in java.util.Arrays with dual-pivot implementation

Changeset: b05abb410c52
Author:    alanb
Date:      2009-10-29 11:18 +0000
URL:       http://hg.openjdk.java.net/jdk7/tl/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

! make/java/java/FILES_java.gmk
! src/share/classes/java/util/Arrays.java
+ src/share/classes/java/util/DualPivotQuicksort.java
```

http://mail.openjdk.java.net/pipermail/compiler-dev/2009-October.txt

Dual-pivot quicksort

Use two partitioning items 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

Now widely used. Java 8, Python unstable sort, Android, …

Quicksort quiz 3

Why does 2-pivot quicksort perform better than 1-pivot?

A. Fewer compares.

B. Fewer exchanges.

C. Both A and B.

D. Neither A nor B.

System sort in Java 8–11

`Arrays.sort()`, `Arrays.parallelSort()`.

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

Algorithms.

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

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

Bottom line. Use the system sort!