http://algs4.cs.princeton.edu

2.3 QUICKSORT

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

A classic sorting algorithm: quicksort

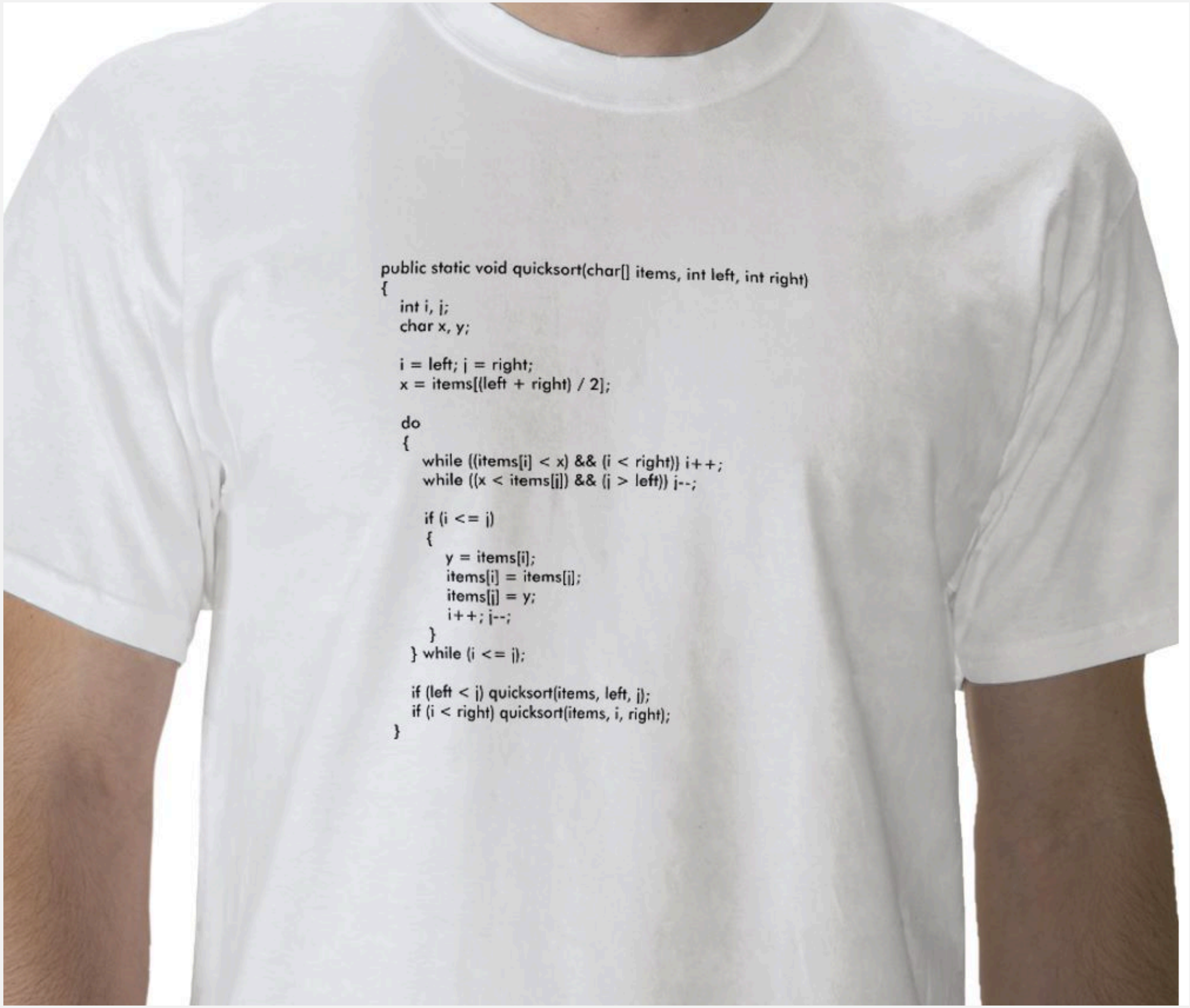
Critical components in the world's computational infrastructure.

- 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.

Quicksort.

…

Quicksort t-shirt



http://algs4.cs.princeton.edu

2.3 Quicksort

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

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.

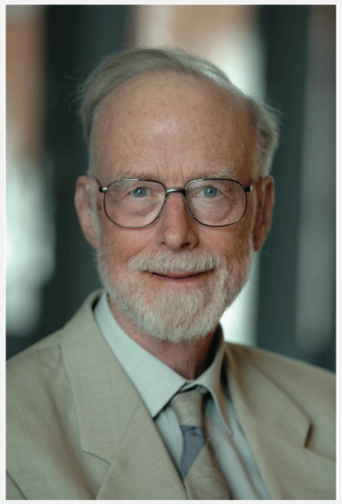
input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

partitioning item

not greater   *not less*

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

Algorithms

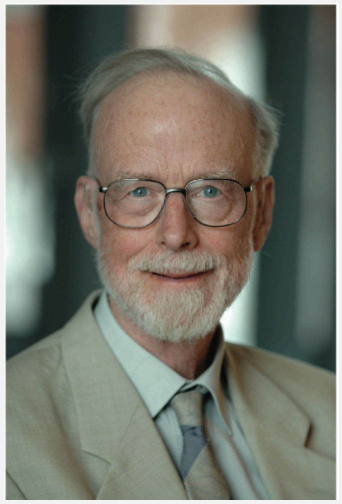
ALGORITHM 64
QUICKSORT
C. A. R. Hoare
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
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
```

Communications of the ACM (July 1961)

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

> " *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.* "

> " *I call it my billion-dollar mistake. It was the invention of the null reference in 1965… 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.* "

Bob Sedgewick

- Refined and popularized quicksort.
- Analyzed quicksort.

Bob Sedgewick

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.

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: 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

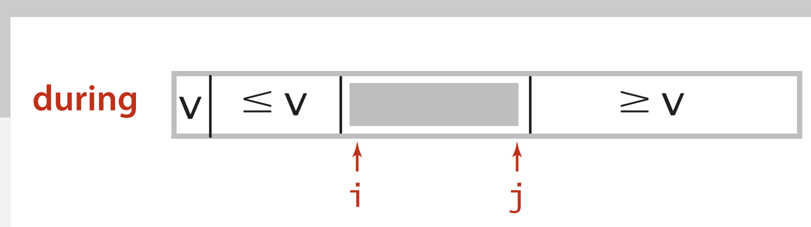
find item on right to swap

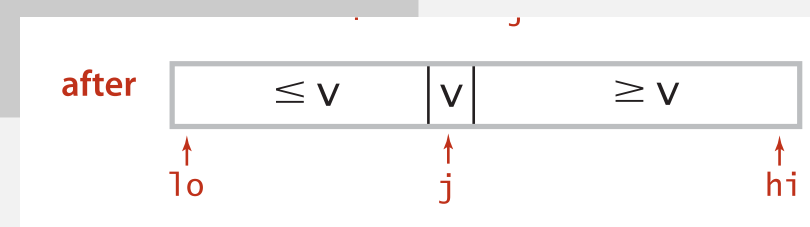
check if pointers cross

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);
  }
}
```

shuffle needed for performance guarantee (stay tuned)

Quicksort trace

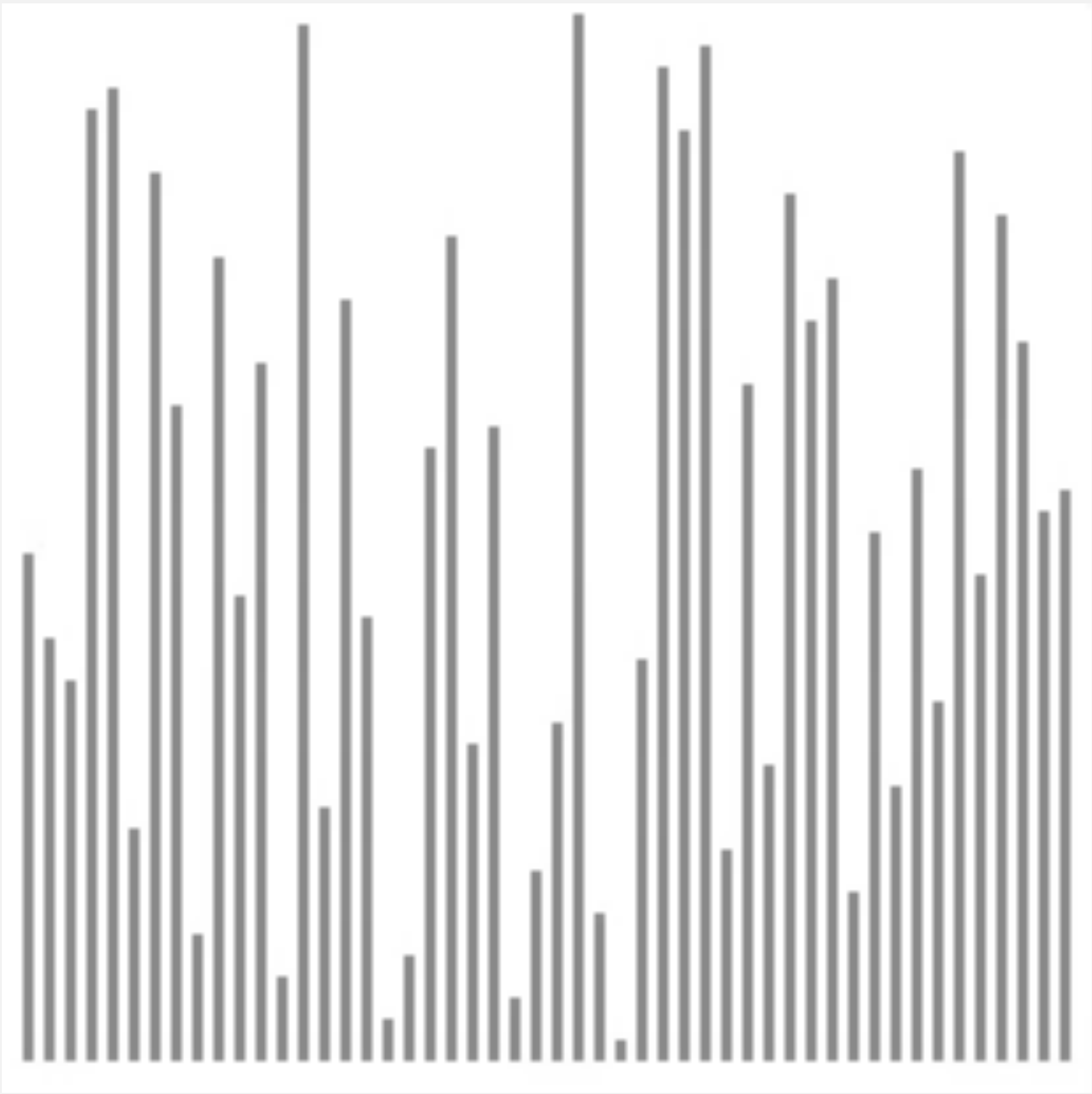
	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	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

no partition for subarrays of size 1

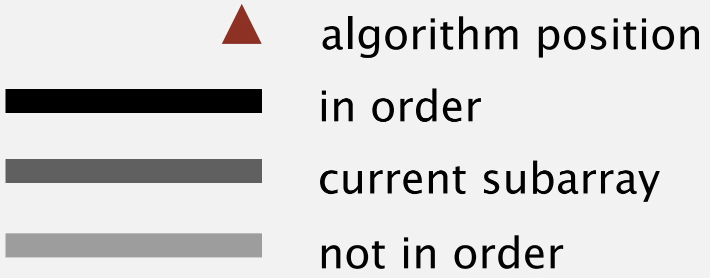
Quicksort trace (array contents after each partition)

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 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.

Quicksort: empirical analysis (1961)

Running time estimates:

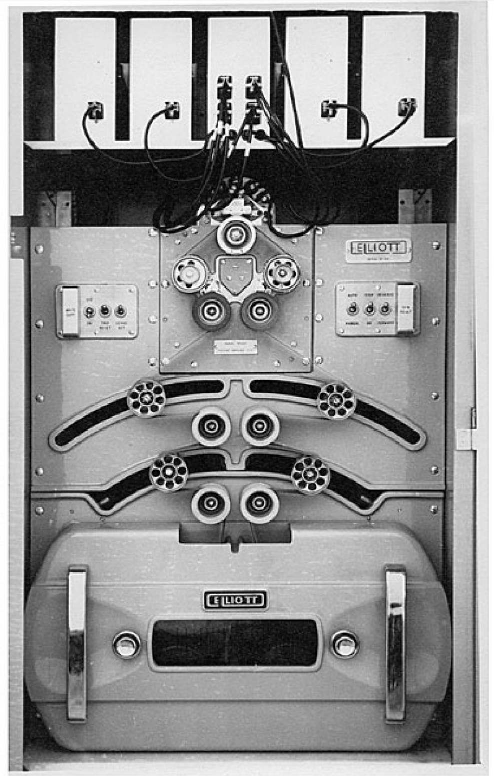
- Algol 60 implementation.
- National-Elliott 405 computer.

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.

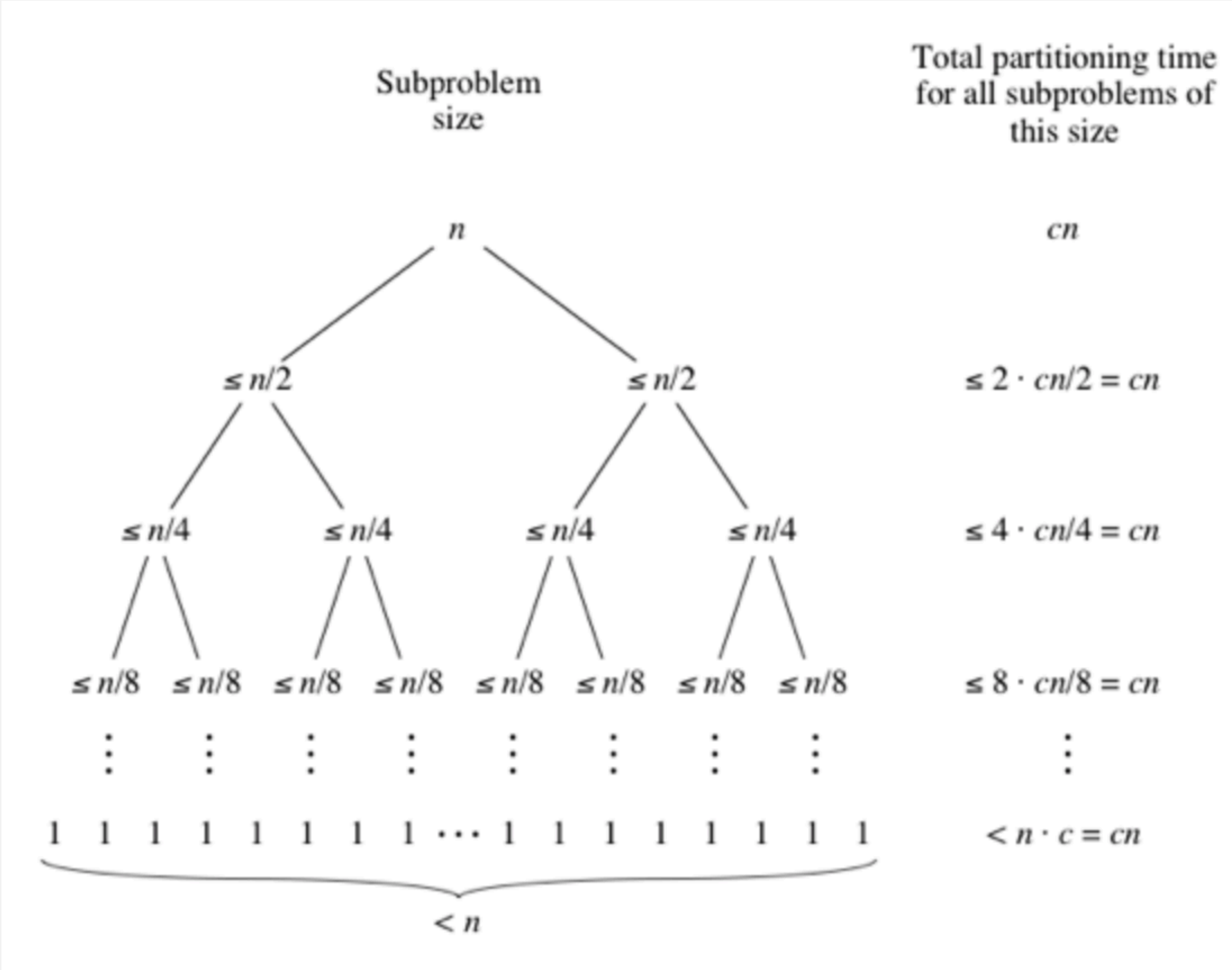
Lesson 2. Great algorithms are better than good ones.

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: best-case analysis



~ N lg N

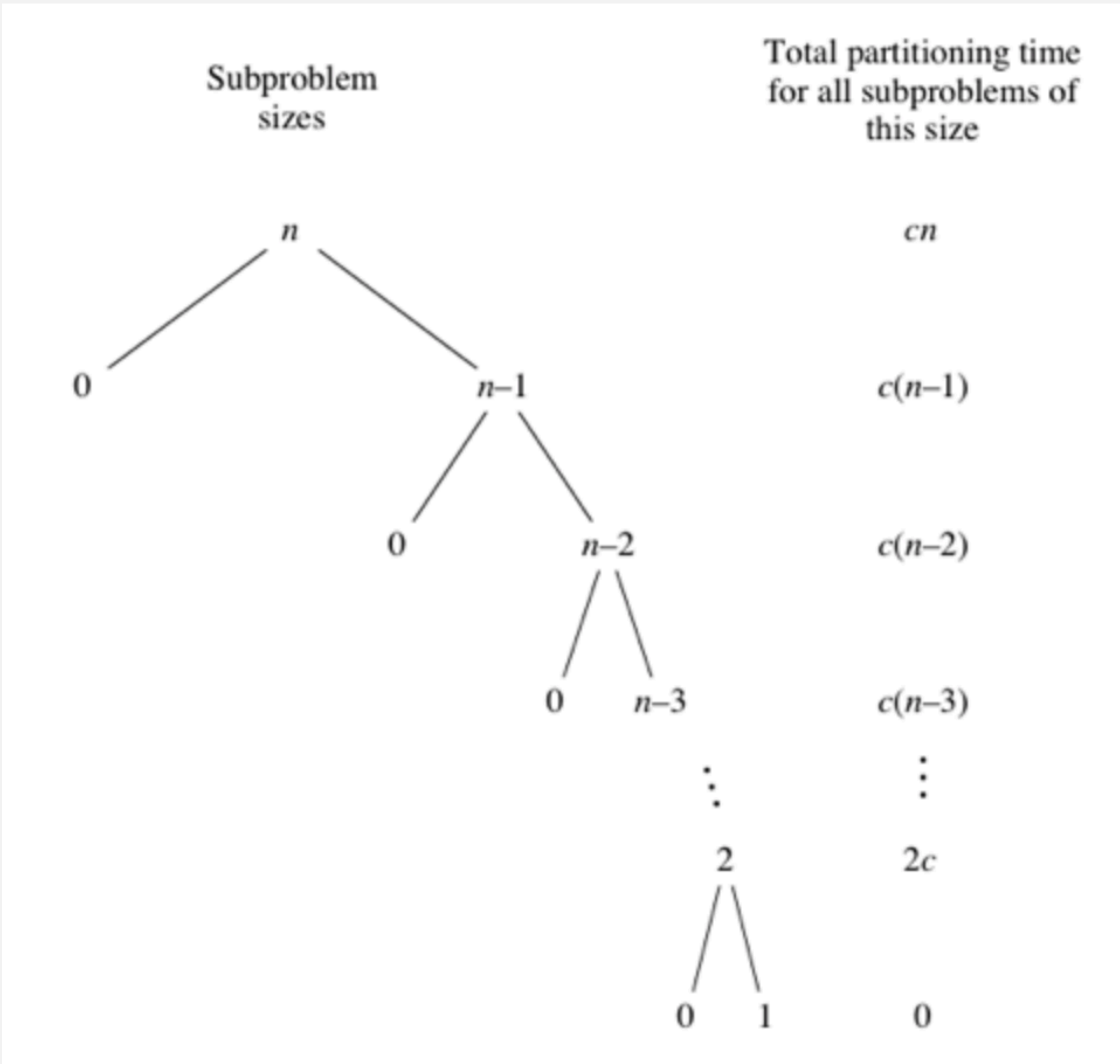
Quicksort: worst-case analysis

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
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: worst-case analysis

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



$$
\begin{aligned}
cn + c(n-1) + c(n-2) + \cdots + 2c &= c(n + (n-1) + (n-2) + \cdot\cdot \\
&= c((n+1)(n/2) - 1) \; .
\end{aligned}
$$

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 $N \geq 2$:

partitioning

left right

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

partitioning 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:

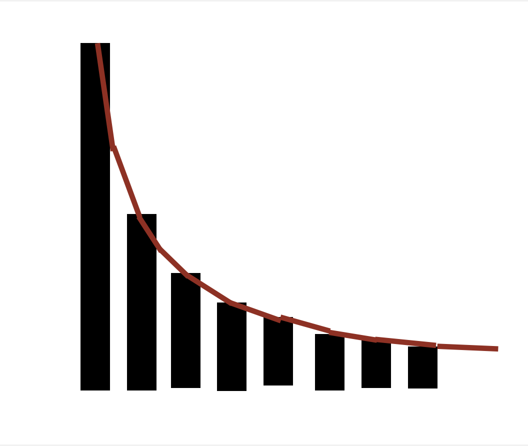
$$
\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\end{aligned}
$$

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:

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

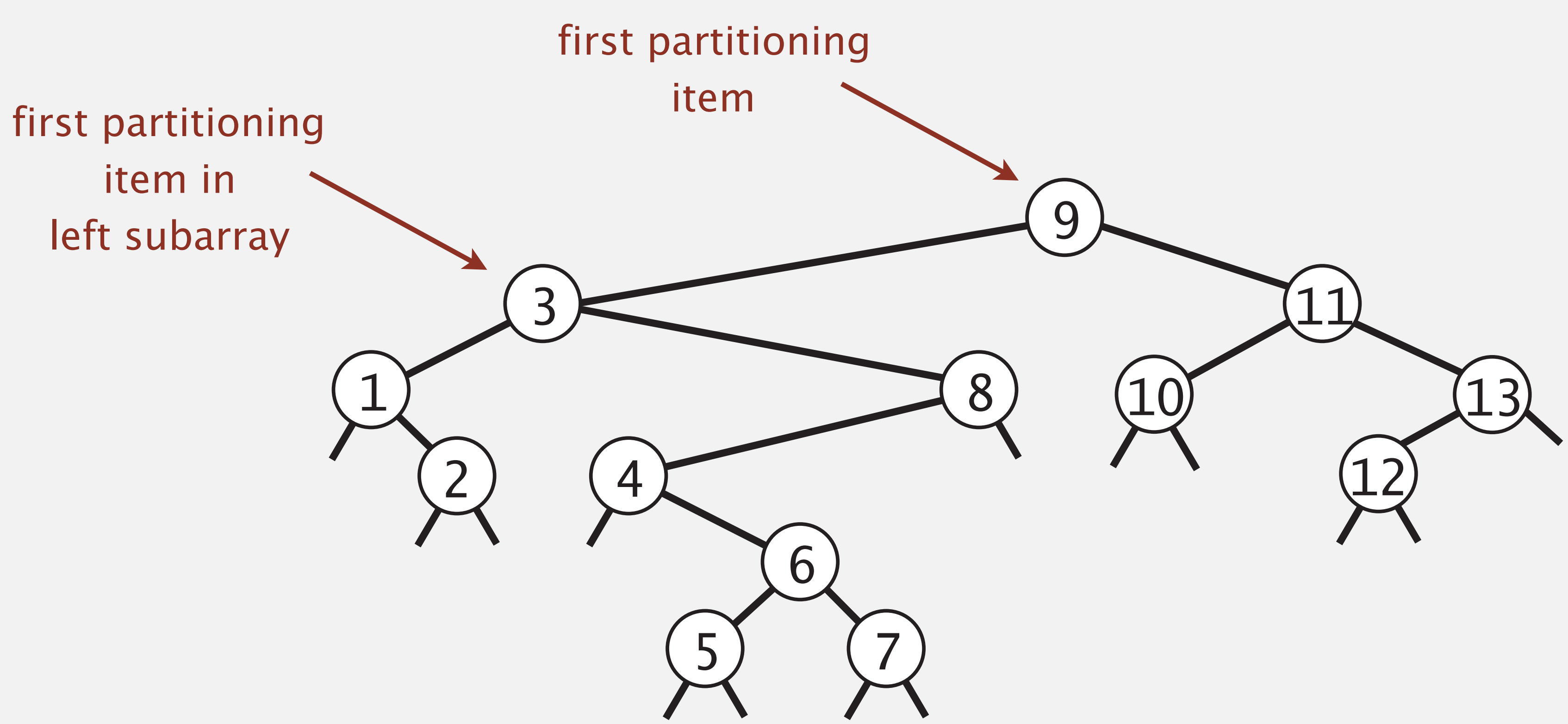
Quicksort: average-case analysis

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 2. Consider BST representation of keys 1 to N.

shuffle

9	10	2	5	8	7	6	1	11	12	13	3	4



Quicksort: average-case analysis

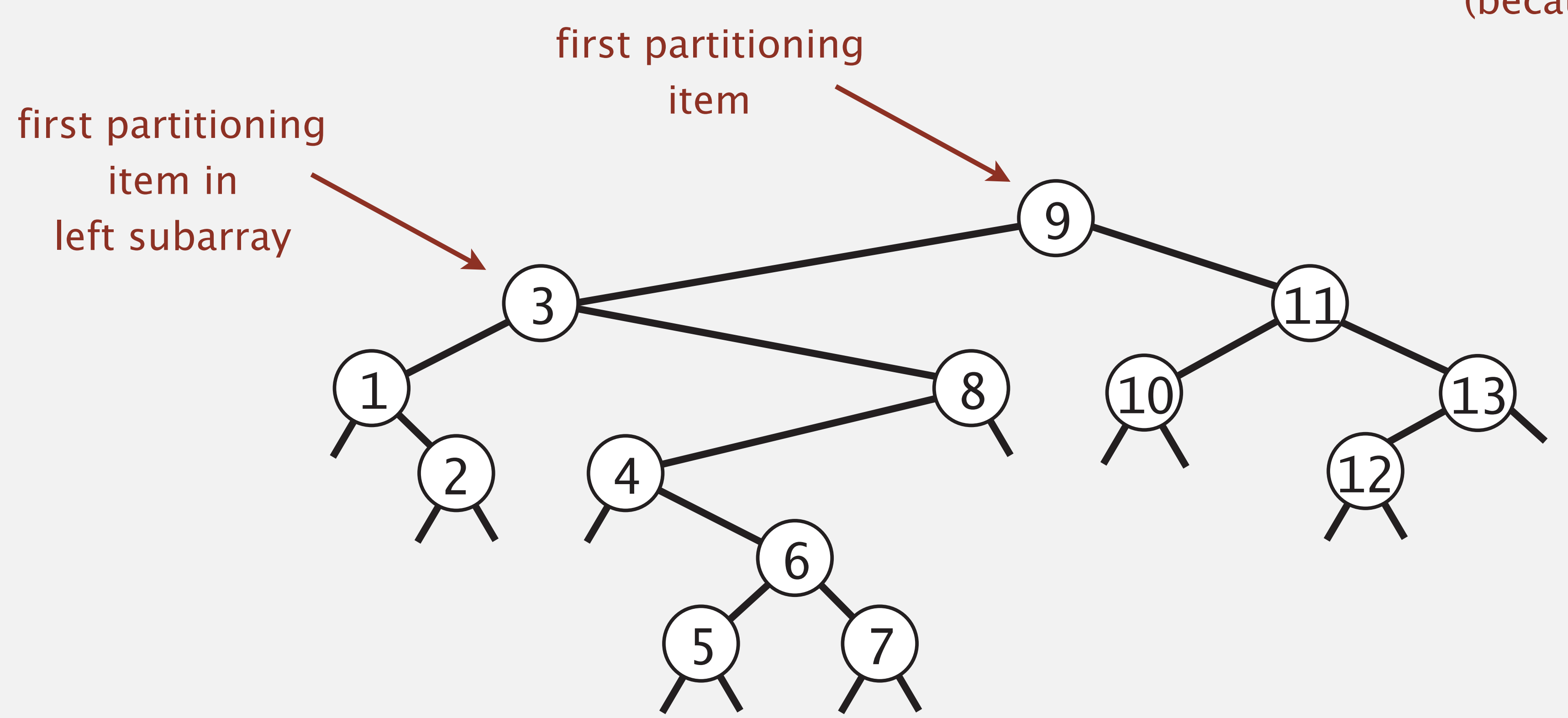
Proposition. The average number of compares C_N to quicksort an array of N distinct keys is $\sim 2N \ln N$ (and the number of exchanges is $\sim \frac{1}{3} N \ln N$).

Pf 2. Consider BST representation of keys 1 to N.

- A key is compared only with its ancestors and descendants.
- Probability i and j are compared equals $2 / |j - i + 1|$.

3 and 6 are compared
(when 3 is partition)

1 and 6 are not compared
(because 3 is partition)



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 2. Consider BST representation of keys 1 to N.

- A key is compared only with its ancestors and descendants.
- Probability i and j are compared equals $2 \,/\, |j - i + 1|$.
- Expected number of compares $= \displaystyle\sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{2}{j-i+1}$

all pairs i and j

$$
\begin{aligned}
&= 2 \sum_{i=1}^{N} \sum_{j=2}^{N-i+1} \frac{1}{j} \\
&\leq 2N \sum_{j=1}^{N} \frac{1}{j} \\
&\sim 2N \int_{x=1}^{N} \frac{1}{x}\, dx \\
&= 2N \ln N
\end{aligned}
$$

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]

Summary

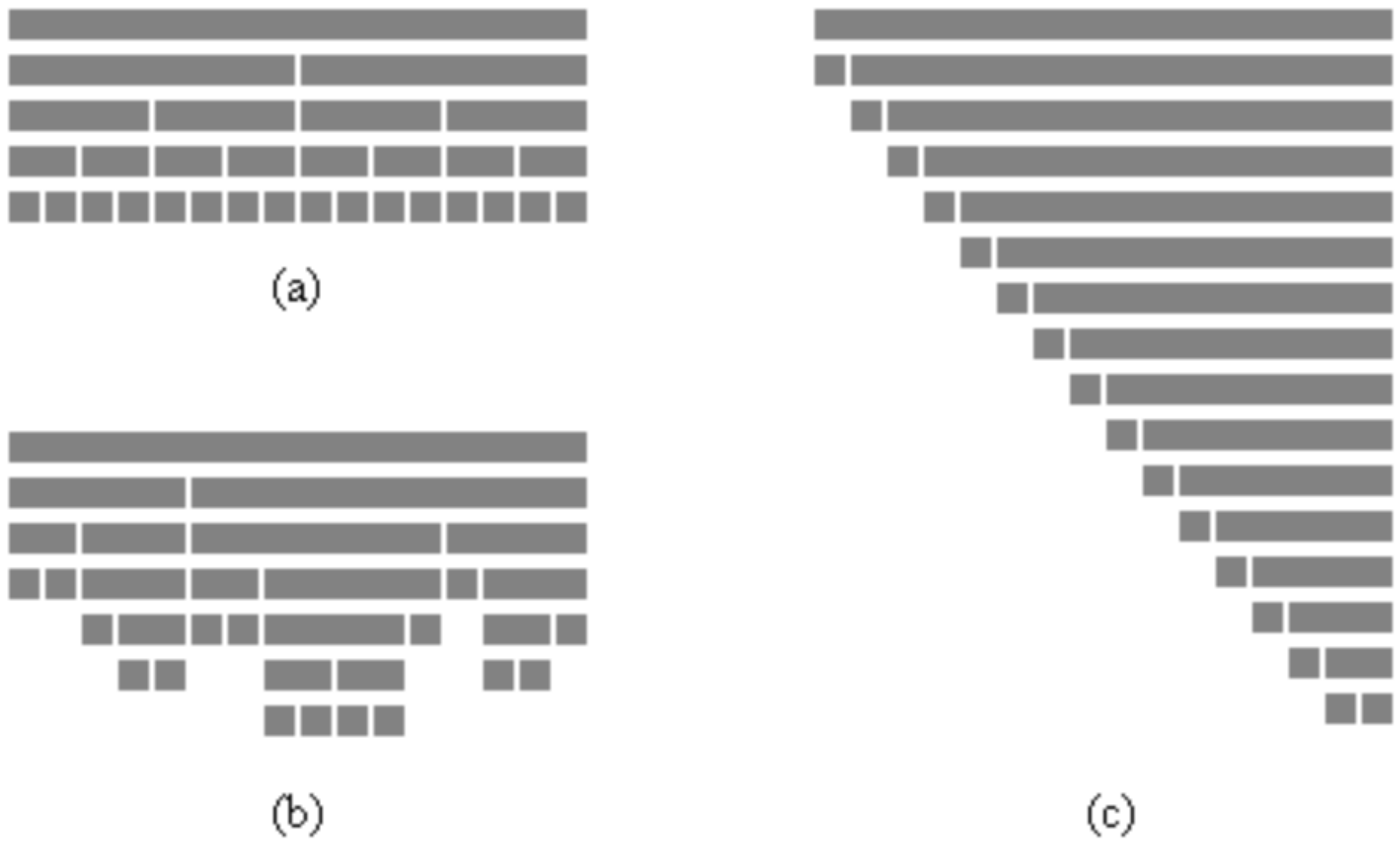


Figure 2: Recursion depth of quicksort: a) best case, b) average case, c) worst case

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

can guarantee logarithmic depth by recurring
on smaller subarray before larger subarray
(requires using an explicit stack)

Proposition. Quicksort is not stable.

Pf. [by counterexample]

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

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

Quicksort: practical improvements

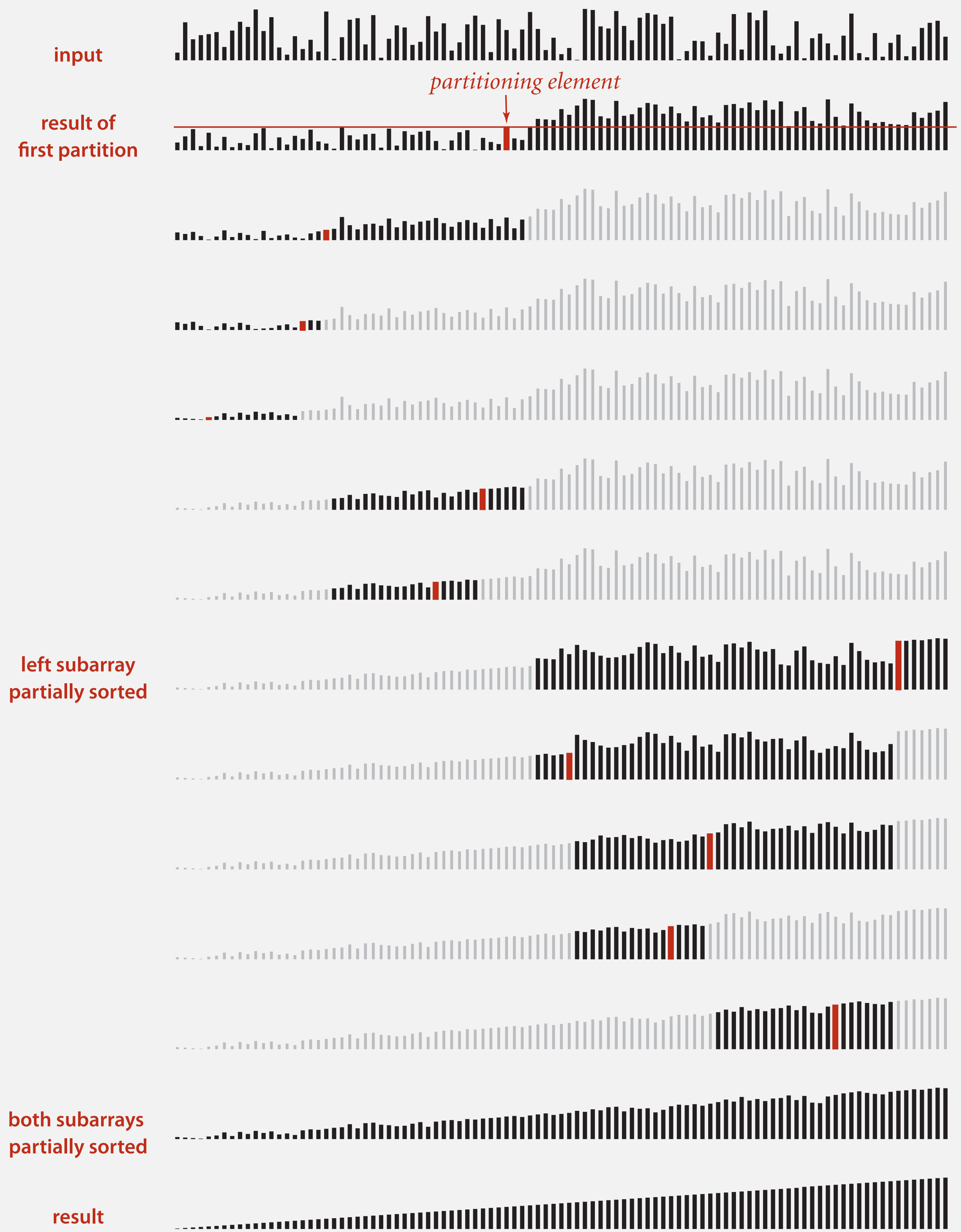
Median of sample.

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

~ 12/7 N ln N compares (14% less)
~ 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 with median-of-3 and cutoff to insertion sort: visualization



http://algs4.cs.princeton.edu

2.3 QUICKSORT

- *quicksort*
- *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

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

Caveat emptor. Some textbook (and commercial) implementations go quadratic when many duplicate keys.

What is the result of partitioning the following array?

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

A.

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

B.

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

C.

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

Partitioning an array with all equal keys: stop on 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

Partitioning an array with all equal keys: do not stop on equal keys

		a[ ]														
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
0	0	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
	0	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.

B A A B A B C C B C B 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.

B A A B A B B B C C C A A A A A A A A A A A

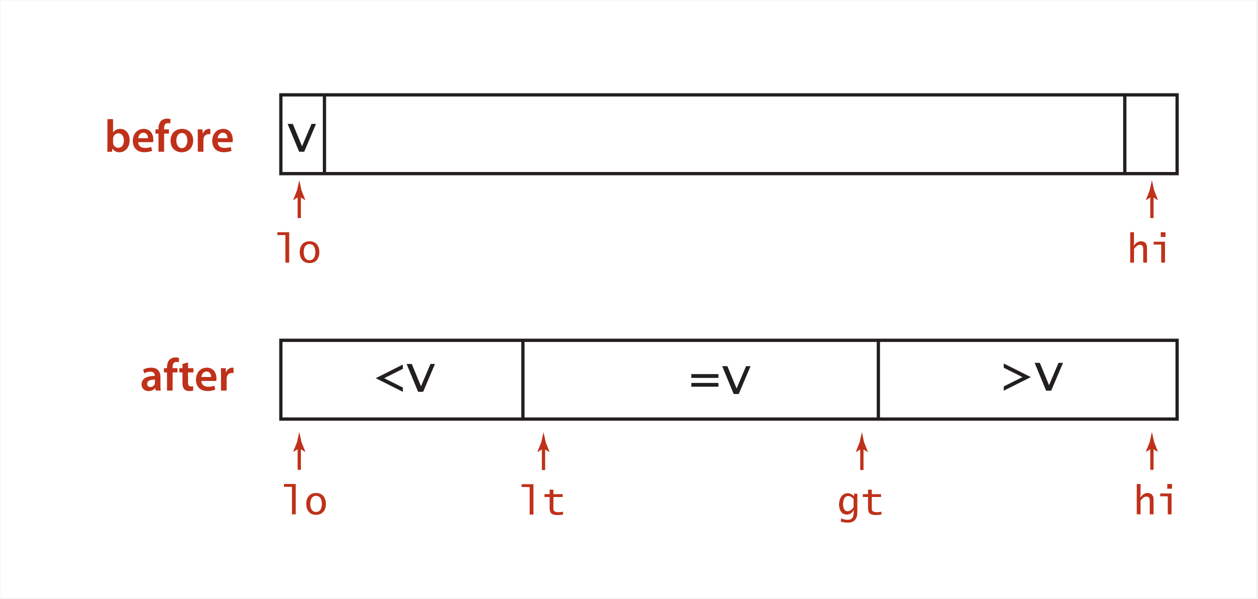
Desirable. Put all items equal to the partitioning item in place.

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 three parts so that:

- Entries between `lt` and `gt` equal to the partition item.
- No larger entries to left of `lt`.
- No smaller entries to right of `gt`.



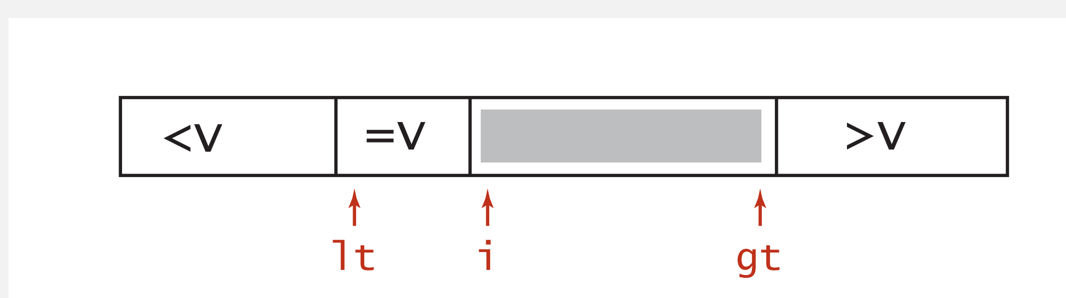
Dutch national flag problem. [Edsger Dijkstra]

- Conventional wisdom until mid 1990s: not worth doing.
- Now incorporated into C library `qsort()` and Java 6 system sort.

Dijkstra 3-way partitioning demo

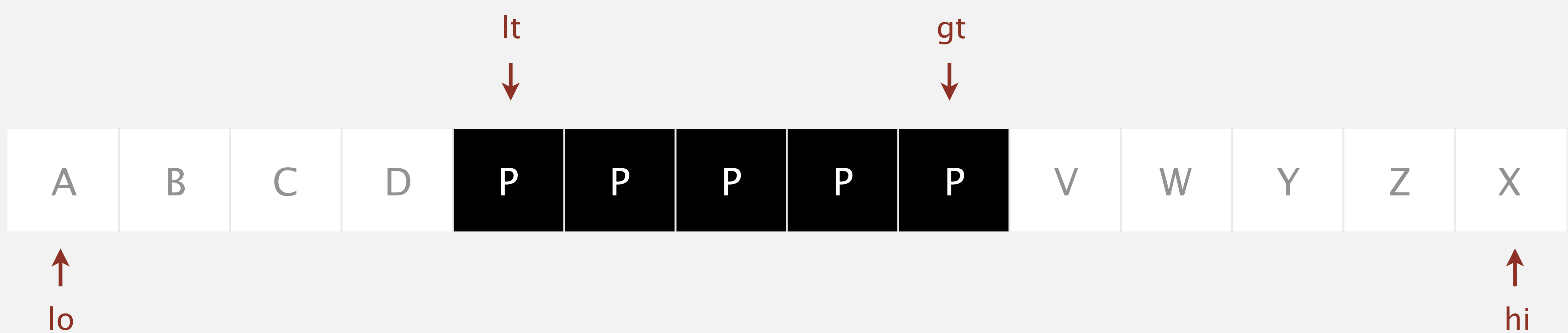
- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

invariant

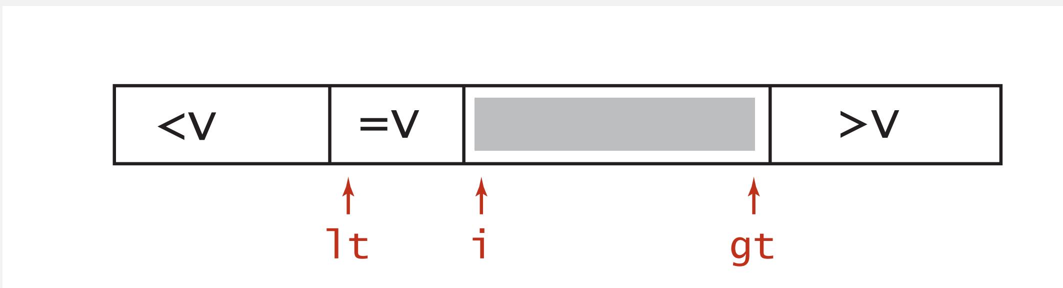


Dijkstra 3-way partitioning demo

- 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



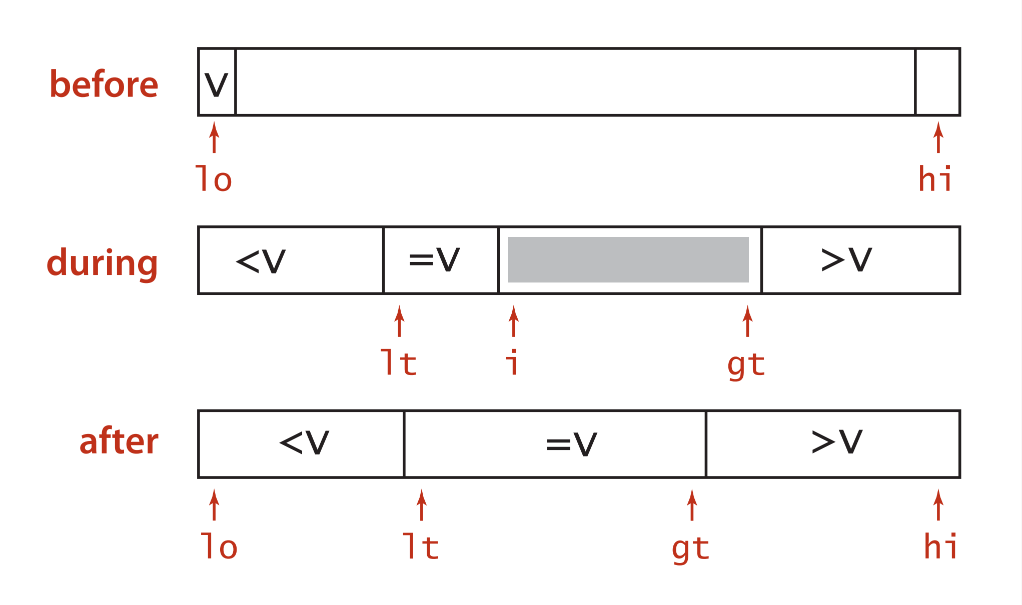
Dijkstra's 3-way partitioning: trace

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

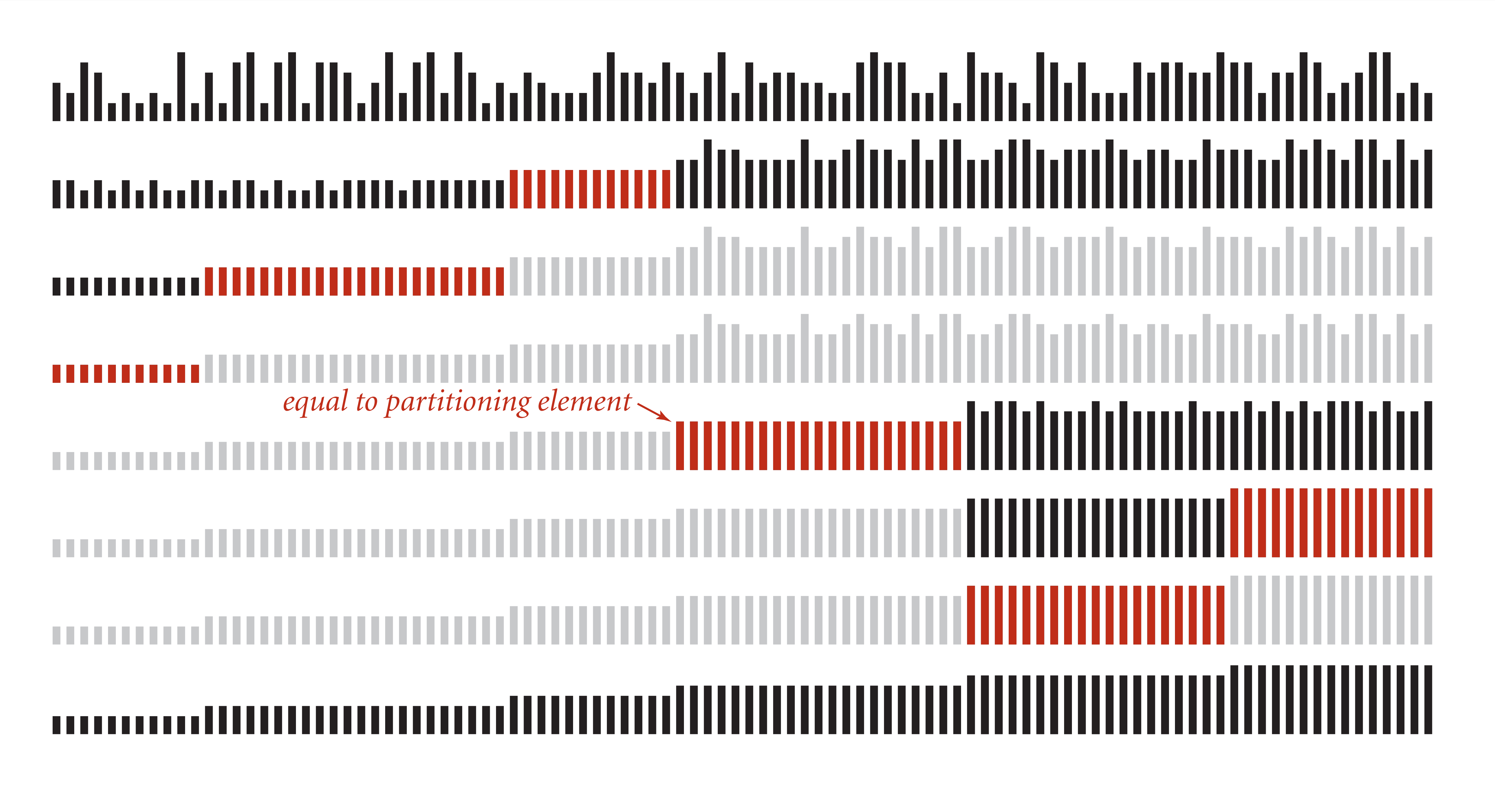
3-way partitioning trace (array contents after each loop iteration)

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



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.
Pf. [beyond scope of course]

proportional to lower bound

Bottom line. Quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.

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

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

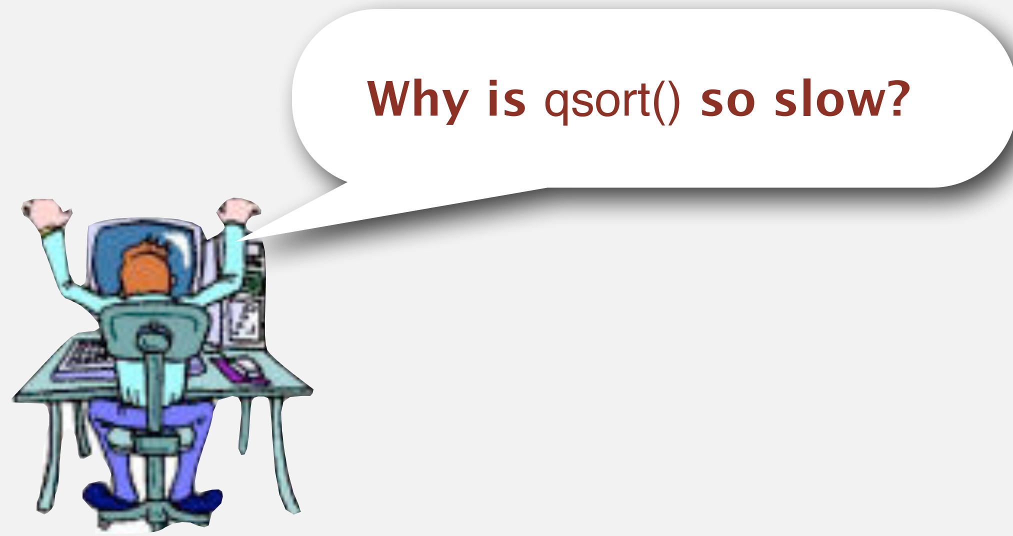
War story (system sort in C)

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 (system sort in C)

Bug. 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.

Engineering a system sort (in 1993)

Basic algorithm for sorting primitive types = quicksort.

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

samples 9 items

similar to Dijkstra 3-way partitioning
(but fewer exchanges when not many equal keys)

Engineering a Sort Function

JON L. BENTLEY
M. DOUGLAS McILROY
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

SUMMARY

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

Very widely used. C, C++, Java 6, ….

A beautiful mailing list post (Yaroslavskiy, September 2011)

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

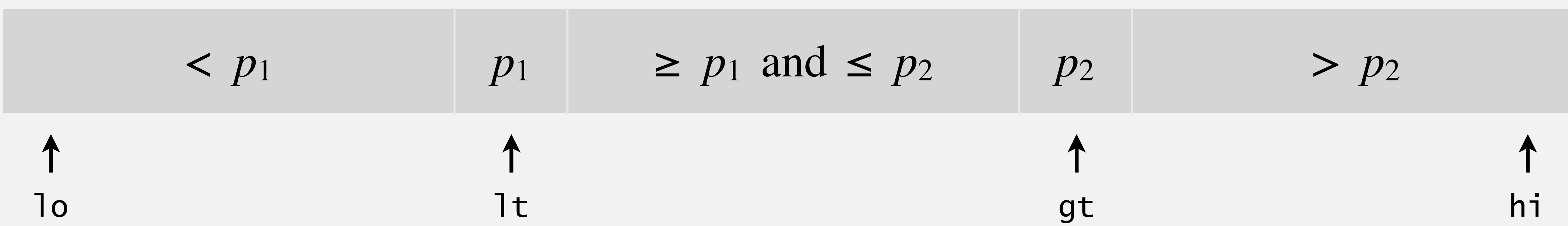
```
Hello All,

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

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.

degenerates to Dijkstra's 3-way partitioning

Note. Skip middle subarray if $p_1 = p_2$.

Dual-pivot partitioning demo

Initialization.

- Choose a[lo] and a[hi] as partitioning items.
- Exchange if necessary to ensure a[lo] ≤ a[hi].

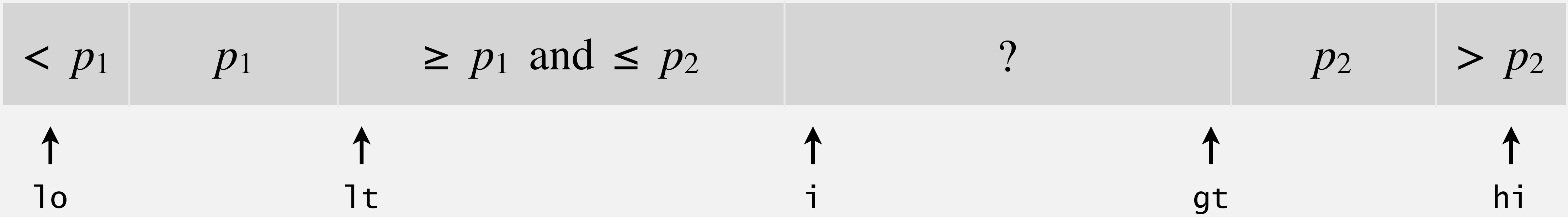
S	E	A	Y	R	L	F	V	Z	Q	T	C	M	K
↑ lo													↑ hi

exchange a[lo] and a[hi]

Dual-pivot partitioning demo

Main loop. Repeat until i and gt pointers cross.

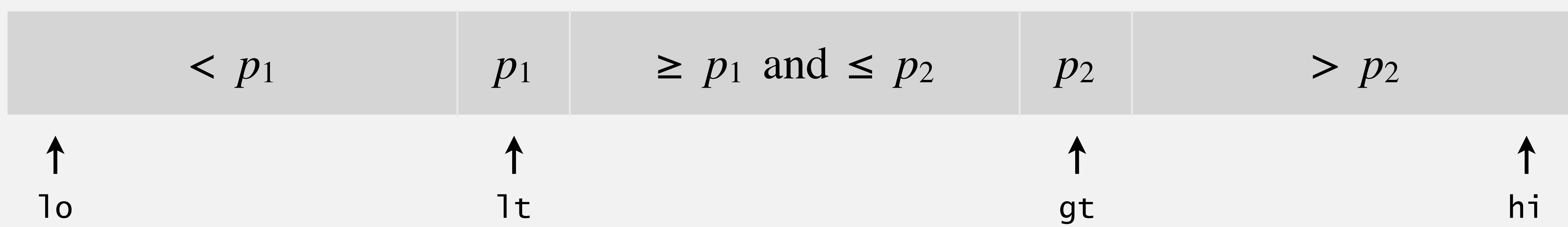
- If (a[i] < a[lo]), exchange a[i] with a[lt] and increment lt and i.
- Else if (a[i] > a[hi]), exchange a[i] with a[gt] and decrement gt.
- Else, increment i.

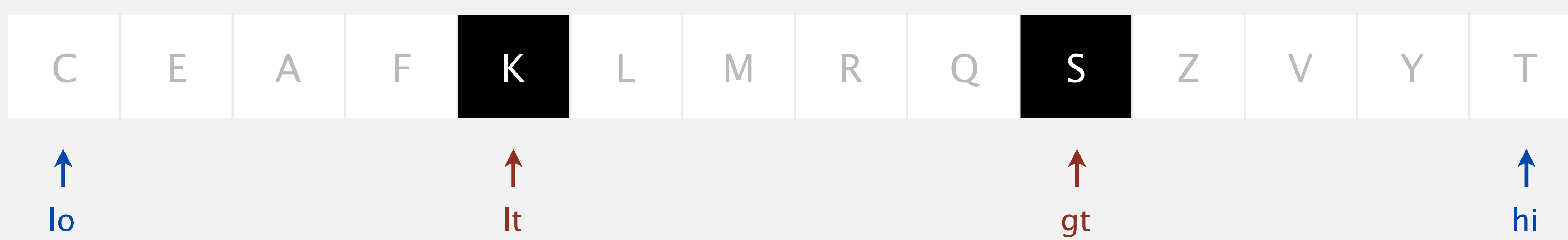


Dual-pivot partitioning demo

Finalize.

- Exchange a[lo] with a[--lt].
- Exchange a[hi] with a[++gt].



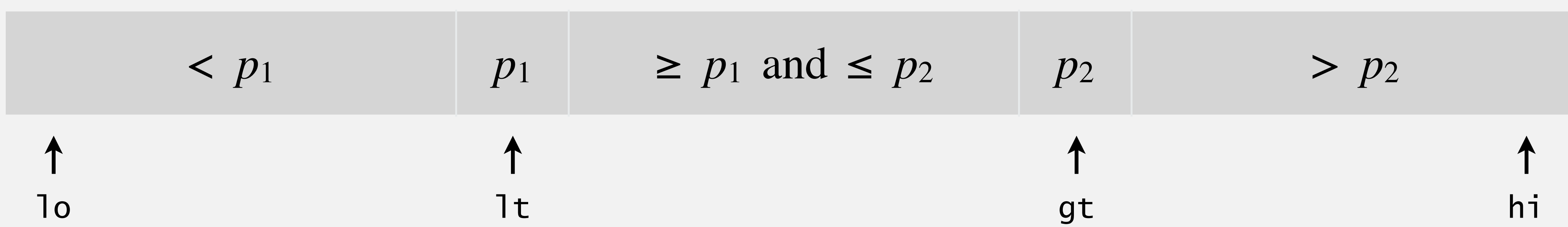


3-way partitioned

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.



Now widely used. Java 7, Python unstable sort, …

Three-pivot quicksort

Use three partitioning items p_1, p_2, and p_3 and partition into four subarrays:

- Keys less than p_1.
- Keys between p_1 and p_2.
- Keys between p_2 and p_3.
- Keys greater than p_3.

$< p_1$	p_1	$\geq p_1$ and $\leq p_2$	p_2	$\geq p_2$ and $\leq p_3$	p_3	$> p_3$
↑ lo	↑ a1		↑ a2		↑ a3	↑ hi

Multi-Pivot Quicksort: Theory and Experiments

Shrinu Kushagra
skushagr@uwaterloo.ca
University of Waterloo

Alejandro López-Ortiz
alopez-o@uwaterloo.ca
University of Waterloo

J. Ian Munro
imunro@uwaterloo.ca
University of Waterloo

Aurick Qiao
a2qiao@uwaterloo.ca
University of Waterloo

Performance

Q. Why do 2-pivot (and 3-pivot) quicksort perform better than 1-pivot?

A. ~~Fewer compares?~~

A. ~~Fewer exchanges?~~

A. Fewer cache misses.

partitioning	compares	exchanges	cache misses
1-pivot	$2\,N \ln N$	$0.333\,N \ln N$	$2\,\frac{N}{B}\ln\frac{N}{M}$
median-of-3	$1.714\,N \ln N$	$0.343\,N \ln N$	$1.714\,\frac{N}{B}\ln\frac{N}{M}$
2-pivot	$1.9\,N \ln N$	$0.6\,N \ln N$	$1.6\,\frac{N}{B}\ln\frac{N}{M}$
3-pivot	$1.846\,N \ln N$	$0.616\,N \ln N$	$1.385\,\frac{N}{B}\ln\frac{N}{M}$

beyond scope of this course

Bottom line. Caching can have a significant impact on performance.

Which sorting algorithm to use?

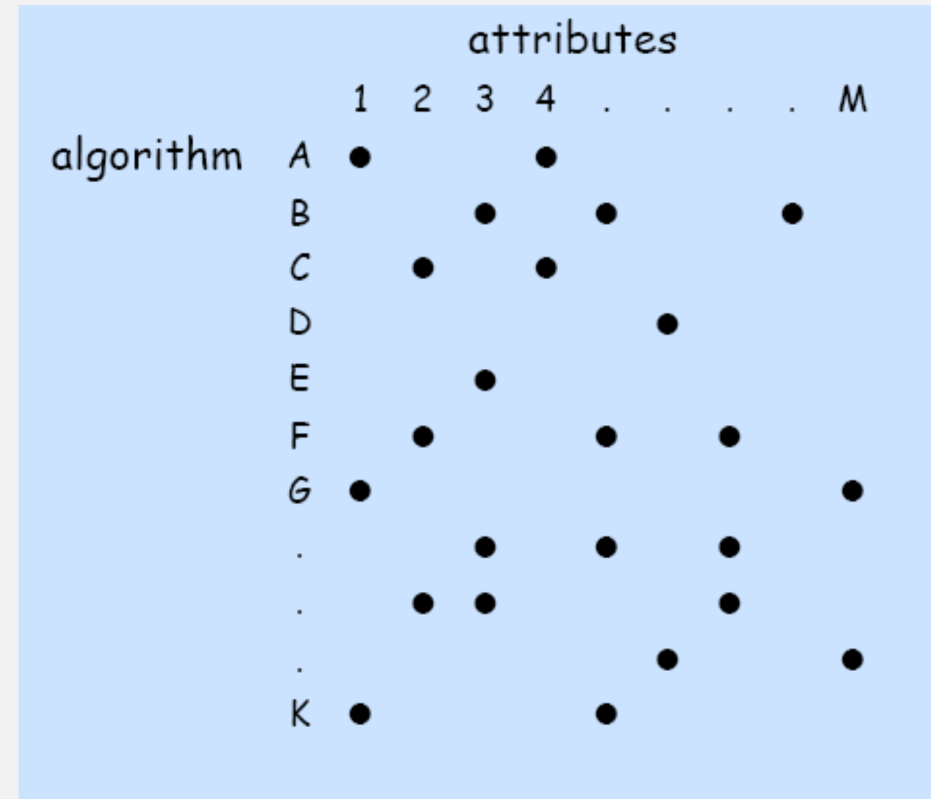
Many sorting algorithms to choose from:

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?



many more combinations of attributes than algorithms

Q. Is the system sort good enough?

A. Usually.

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.

- Dual-pivot quicksort for primitive types.
- Timsort for reference types.

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

Ineffective sorts

http://xkcd.com/1185