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 20^{th} century in science and engineering.

Mergesort. [last lecture]

JS ...

Quicksort. [this lecture]

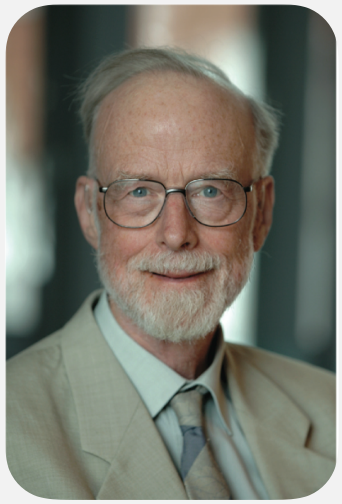
C UNIX JS ...

Quicksort t-shirt

A brief history

Tony Hoare

- Invented quicksort to translate Russian into English.
- [but couldn't explain or implement it!]
- Learned Algol 60 (and recursion).
- Implemented quicksort.



Tony Hoare
1980 Turing Award

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 demo

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

stop i scan because a[i] >= a[lo]

Quicksort quiz 1

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

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

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

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

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

Quicksort partitioning: Java implementation

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))
         if (i == hi) break;

      while (less(a[lo], a[--j]))
         if (j == lo) break;

      if (i >= j) break;
      exch(a, i, j);
   }

   exch(a, lo, j);
   return j;
}
```

find item on left to swap

find item on right to swap

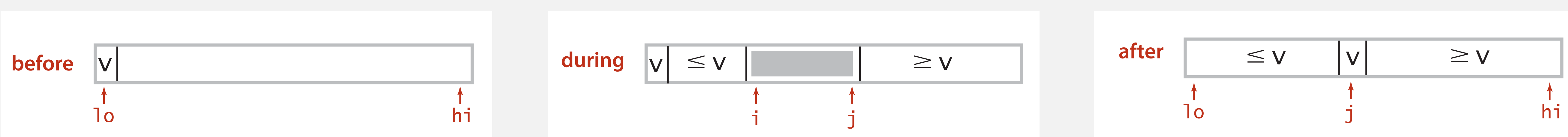
check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place

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

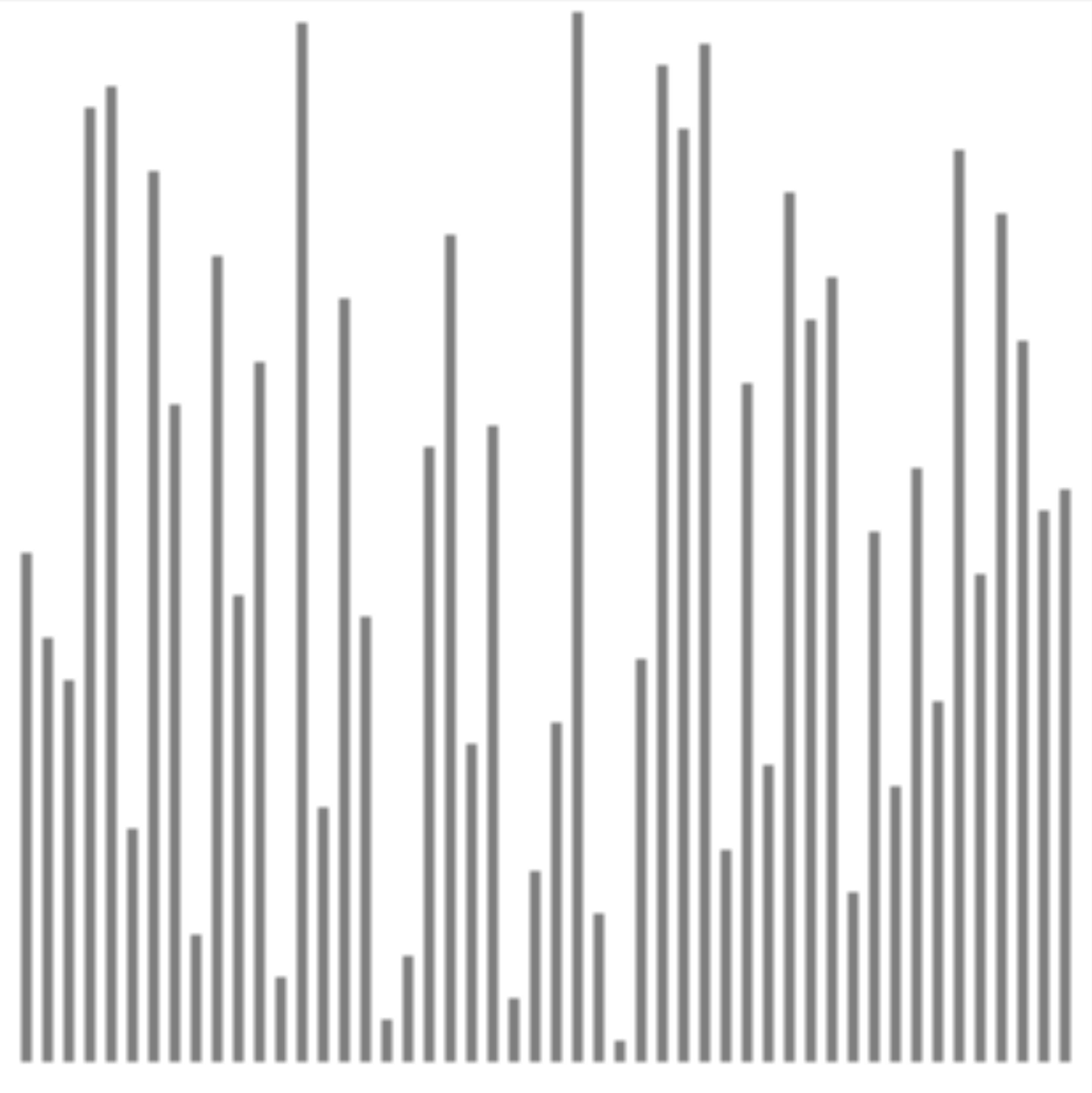
Quicksort trace

	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



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

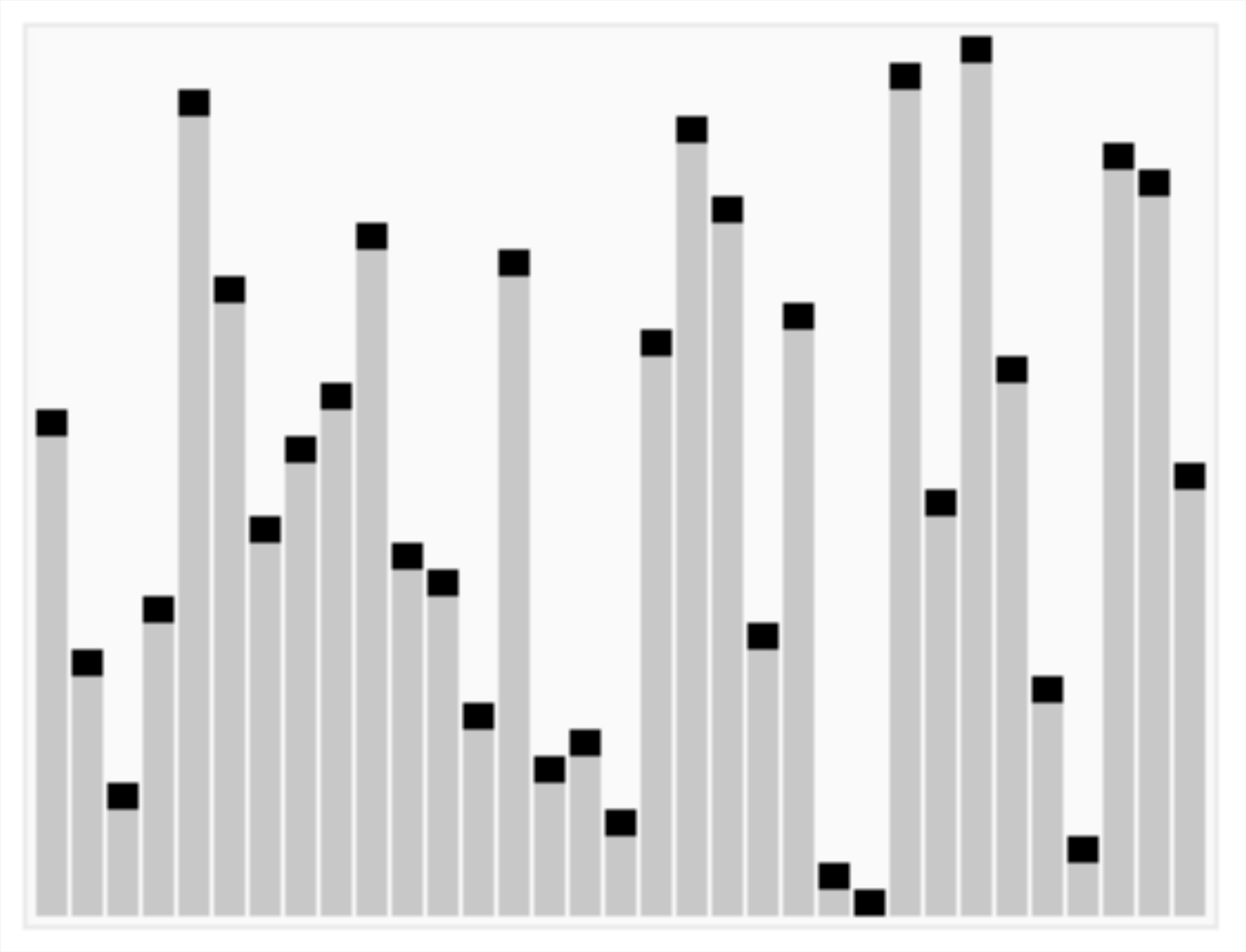
▲ algorithm position

in order

current subarray

not in order

Another quicksort animation



https://en.wikipedia.org/wiki/Quicksort

Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but it is not worth the cost.

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

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

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

Quicksort: empirical analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (n^2)			mergesort (n log n)			quicksort (n log n)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Quicksort: worst-case analysis

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

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

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

after random shuffle

Good news. Worst case analysis of quicksort is irrelevant for practical purposes.

Worst case exponentially unlikely to occur (unless bug in shuffling method.)
More likely that lightning strikes computer during execution.

Quicksort: average-case 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 $\sim \frac{1}{3}\, n \ln n$).

Intuition. Each partitioning step splits array approximately in half.

Recall: Any algorithm with the following structure takes $\Theta(n \log n)$ time.

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

For quicksort, the two problems aren't exactly half the size, but close enough.

Quicksort: average-case 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 $n \geq 2$:

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:

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

- 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 $n\,(n+1)$:

$$\frac{C_n}{n+1} \;=\; \frac{C_{n-1}}{n} \;+\; \frac{2}{n+1}$$

Interesting but out of scope

Quicksort: average-case analysis

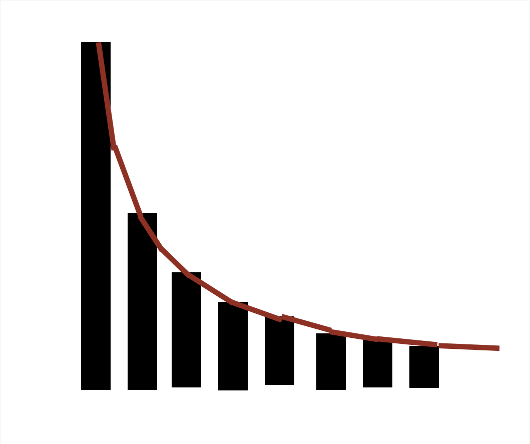
- Repeatedly apply previous equation:

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

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: summary of performance characteristics

Quicksort is a 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]

Three different types of average-case complexity

	Cost is averaged over...	Example	Impact of worst case
Amortized	Sequence of operations	Stacks and queues using resizing arrays	Some operations take (far) longer than amortized running time
Expected	Internal randomness of implementation	Quicksort	Irrelevant
Average case	Possible inputs	Quicksort without shuffling	Worst case may occur if our model of "average" input is wrong

Frequent source of performance bugs in practice

In this course, for simplicity, we'll ignore the distinction between average case and expected complexity.

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

- Partitioning: constant extra space.
- Function-call stack: logarithmic extra space (with high probability).

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 improvement

Insertion sort small subarrays.

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

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 $(k = n / 2)$.

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?

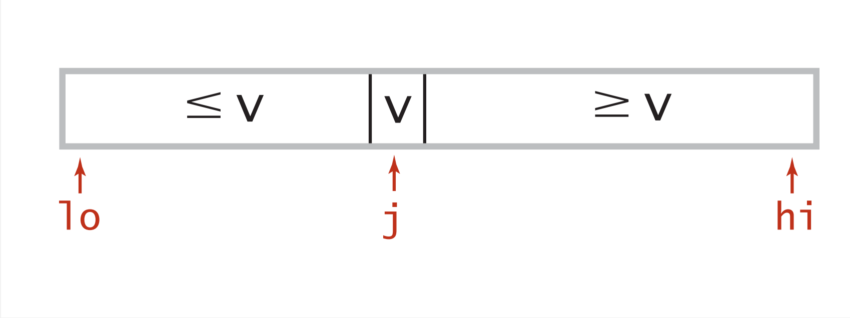
- $n \log n$ lower bound? ← is selection as hard as sorting?
- n upper bound? ← is there a linear-time algorithm?

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`.



Proposition. Quick-select takes linear time on average.

Intuition:

Each partitioning step splits array approximately in half:

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

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

Duplicate keys: stop on equal keys

Our partitioning subroutine stops both scans on equal keys.

scan until ≥ P

scan until ≤ P

P	G	E	P	A	Q	B	P	Y	C	O	U	P	Z	S	R

Q. Why not continue scans on equal keys?

scan until > P

scan until < P

P	G	E	P	A	Q	B	P	Y	C	O	U	P	Z	S	R

Quicksort: quiz 2

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

scan until > A → ← scan until < A

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

D. *I don't know.*

Quicksort: quiz 3

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

scan until ≥ A

scan until ≤ A

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

D. *I don't know.*

Partitioning an array with all equal keys

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
1	15	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
2	14	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
3	13	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
4	12	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
5	11	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
6	10	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
7	9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
	8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
	8	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

Duplicate keys: partitioning strategies

Bad. Don't stop scans on equal keys.

[$\sim \frac{1}{2} n^2$ compares when all keys equal]

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

Good. Stop scans on equal keys.

[$\sim n \lg n$ compares when all keys equal]

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

Better. Put all equal keys in place. How?

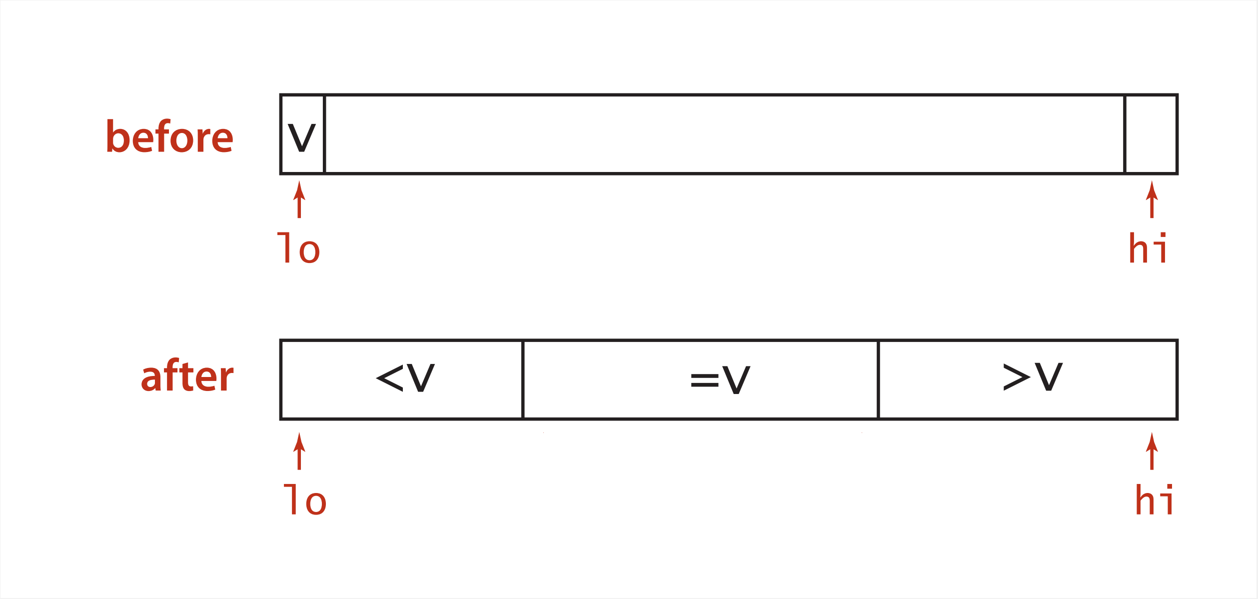
[$\sim n$ compares when all keys equal]

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

3-way partitioning

Goal. Partition array into three parts so that:

- Entries in the left part are less than the partitioning item.
- Entries in the left part are equal to the partitioning item.
- Entries in the left part are greater than the partitioning item.

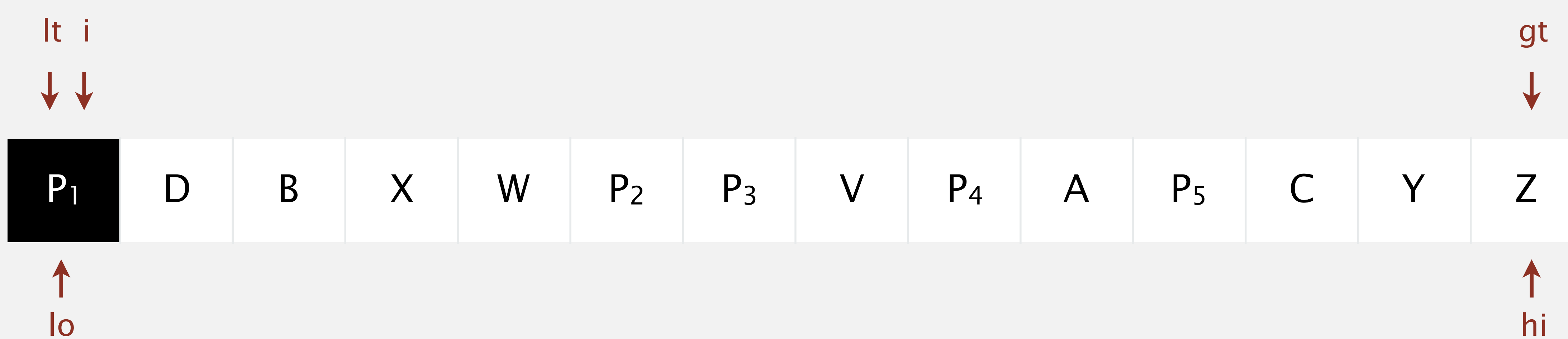


3-way partitioning algorithm. [Edsger Dijkstra]

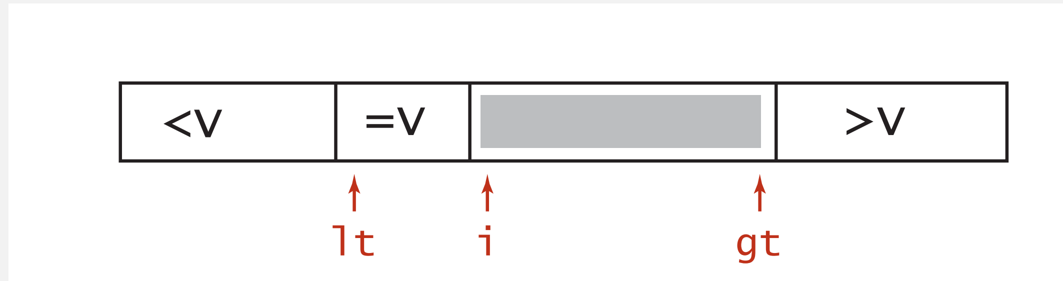
- Now incorporated into C library `qsort()` and Java 6 system sort.

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`

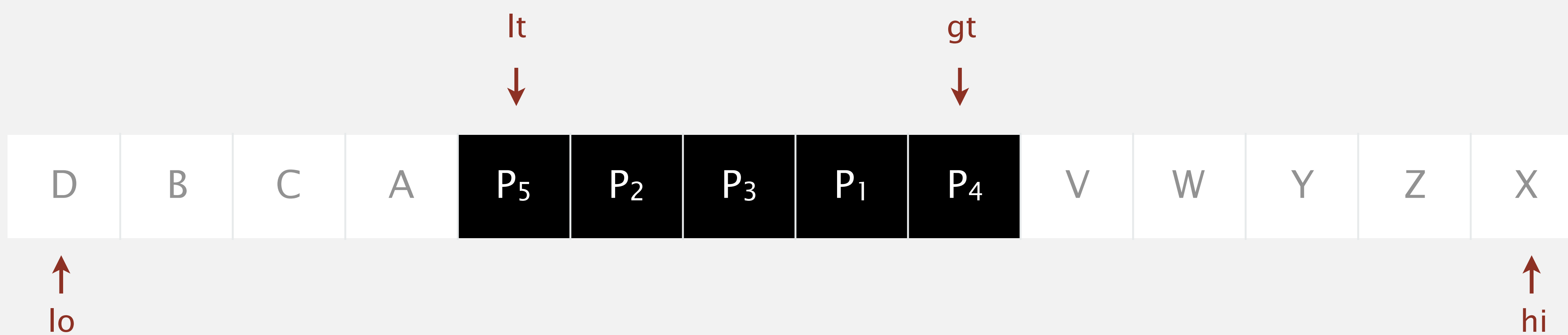


invariant

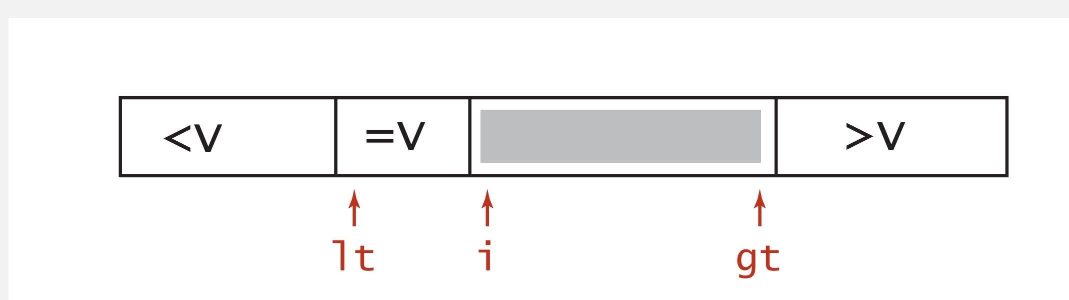


Dijkstra's 3-way partitioning algorithm: demo

- 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



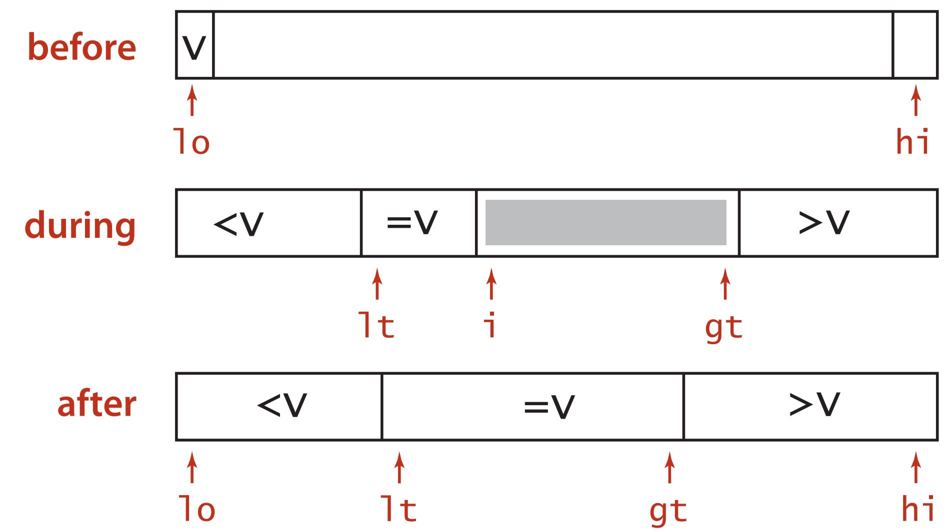
invariant



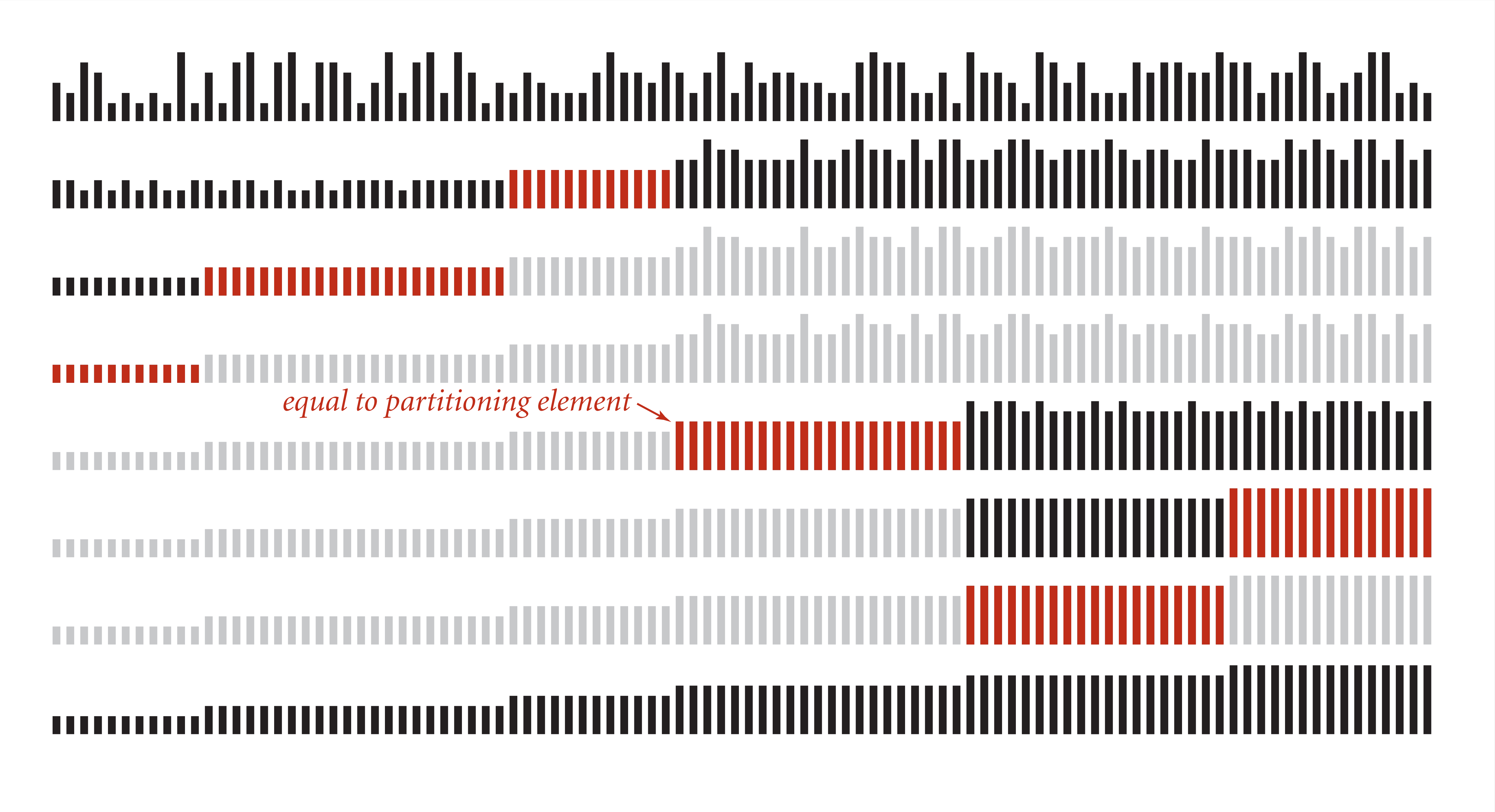
3-way quicksort: Java implementation

```
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
merge		✔	$\frac{1}{2}\, n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
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

System sort in Java 8

`Arrays.sort()`.

- 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.

Use two pivots for partitioning; recursively sort three subarrays

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

Optimized mergesort

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

Q. Why so many overloaded methods?

Bottom line. Use the system sort!