CS 171: Introduction to Computer Science II

Quicksort

Welcome back from Spring break!

- Quiz 1 distributed
- Hw3 with 2 late credit is due 3/20 (today)
- Hw4 with 1 late credit is due 3/21, with 2 late credits is due 3/24

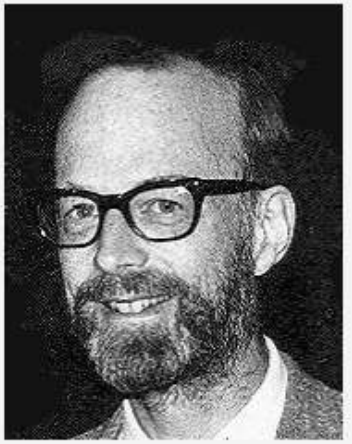
Outline

- Quicksort algorithm (review)
- Quicksort Analysis (cont.)
  - Best case
  - Worst case
  - Average case
- Quicksort improvements
- Sorting summary
- Java sorting methods

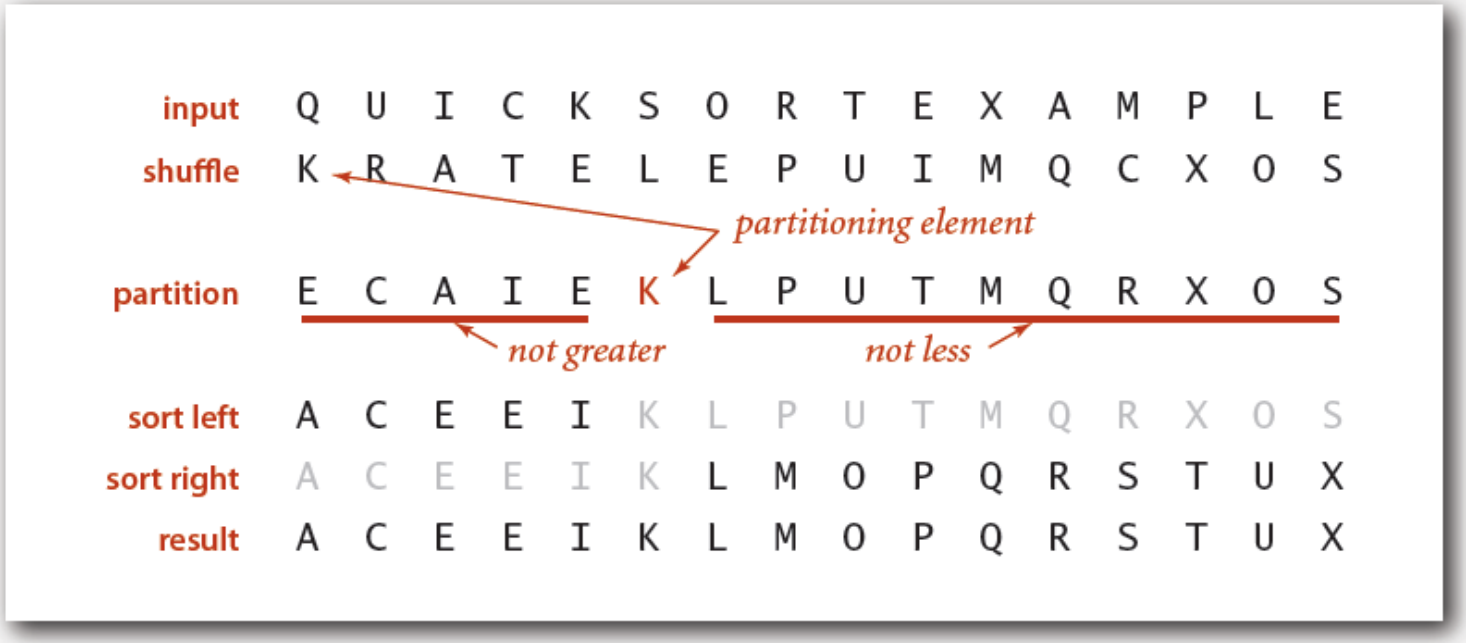
Quicksort

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 piece recursively.



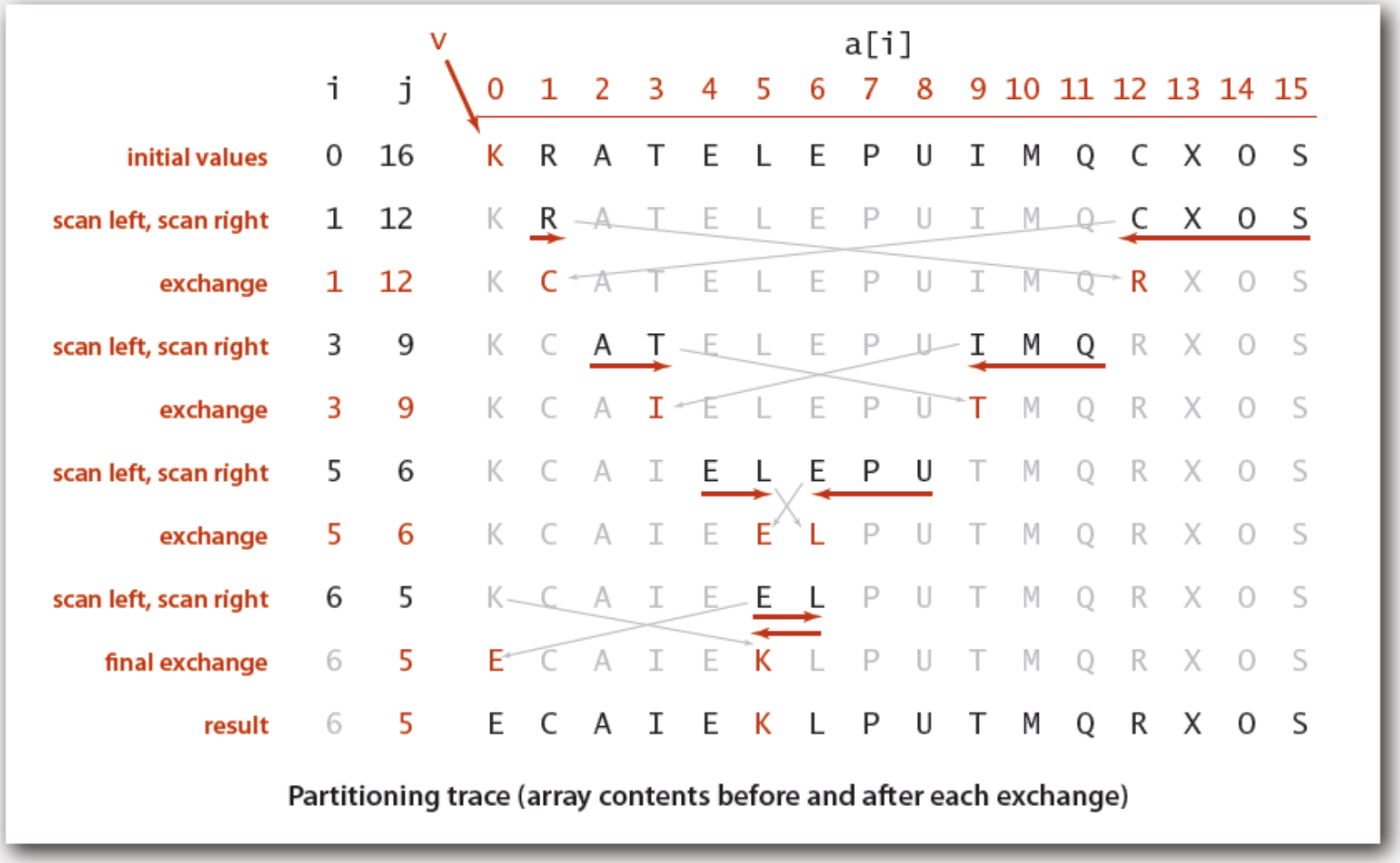
Sir Charles Antony Richard Hoare
1980 Turing Award



Quicksort partitioning

Basic plan.

- Scan `i` from left for an item that belongs on the right.
- Scan `j` from right for an item that belongs on the left.
- Exchange `a[i]` and `a[j]`.
- Repeat until pointers cross.



Partitioning trace (array contents before and after each exchange)

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

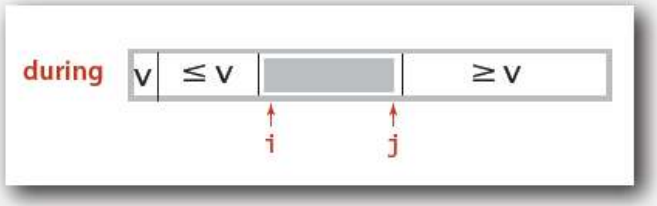
find item on right to swap

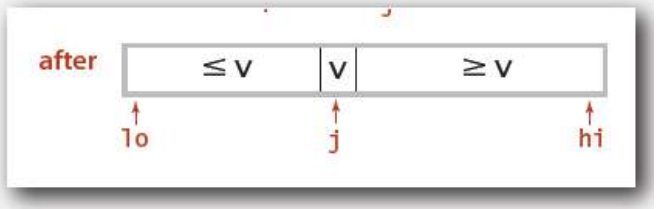
check if pointers cross

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 Cost Analysis

- Depends on the partitioning
  - What's the best case?
  - What's the worst case?
  - What's the average case?

Quicksort: best-case analysis

			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 Cost Analysis – Best case

- The best case is when each partition splits the array into two equal halves
- Overall cost for sorting N items
  - Partitioning cost for N items: N+1 comparisons
  - Cost for recursively sorting two half-size arrays
- Recurrence relations
  - $C(N) = 2\ C(N/2) + N + 1$
  - $C(1) = 0$

Quicksort Cost Analysis – Best case

- Simplified recurrence relations
  - $C(N) = 2\ C(N/2) + N$
  - $C(1) = 0$
- Solving the recurrence relations
  - $N = 2^k$
  - $C(N) = 2\ C(2^{k-1}) + 2^k$

    $= 2\ (2\ C(2^{k-2}) + 2^{k-1}) + 2^k$

    $= 2^2\ C(2^{k-2}) + 2^k + 2^k$

    $= \ldots$

    $= 2^k\ C(2^{k-k}) + 2^k + \ldots\ 2^k + 2^k$

    $= 2^k + \ldots\ 2^k + 2^k$

    $= k * 2^k$

    $= O(N\log N)$

Quicksort: worst-case analysis

			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 Cost Analysis – Worst case

- The worst case is when the partition does not split the array (one set has no elements)
- Ironically, this happens when the array is sorted!
- Overall cost for sorting N items
  - Partitioning cost for N items: N+1 comparisons
  - Cost for recursively sorting the remaining (N-1) items
- Recurrence relations
  - $C(N) = C(N-1) + N + 1$
  - $C(1) = 0$

Quicksort Cost Analysis – Worst case

- Simplified Recurrence relations

  $C(N) = C(N-1) + N$

  $C(1) = 0$

- Solving the recurrence relations

$$
\begin{aligned}
C(N) &= C(N-1) + N \\
&= C(N-2) + N - 1 + N \\
&= C(N-3) + N-2 + N-1 + N \\
&= \ldots \\
&= C(1) + 2 + \ldots + N-2 + N-1 + N \\
&= O(N^2)
\end{aligned}
$$

Quicksort Cost Analysis – Average case

- Suppose the partition split the array into 2 sets containing k and N-k-1 items respectively (0<=k<=N-1)
- Recurrence relations
  - C(N) = C(k) + C(N-k-1) + N + 1
- On average,
  - C(k) = C(0) + C(1) + … + C(N-1) /N
  - C(N-k-1) = C(N-1) + C(N-2) + … + C(0) /N
- Solving the recurrence relations (not required for the course)
  - Approximately, C(N) = 2NlogN

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: 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.

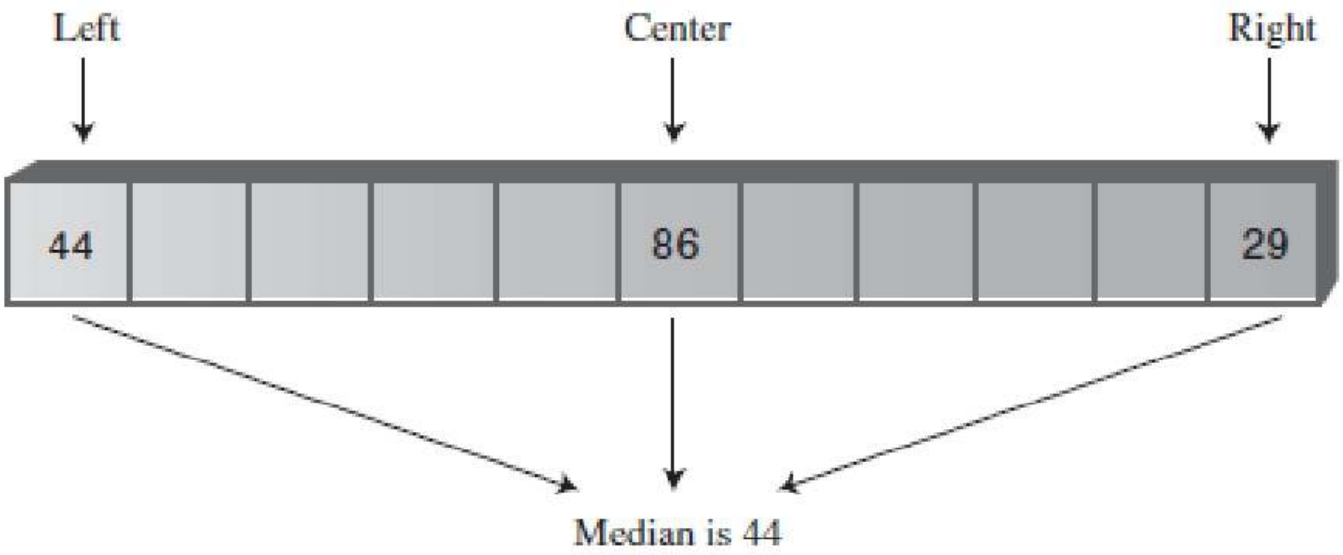
```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

QuickSort: practical improvement

- The basic QuickSort uses the first (or the last element) as the pivot value
- What's the best choice of the pivot value?
- Ideally the pivot should partition the array into two equal halves

Median-of-Three Partitioning

- We don't know the median, but let's approximate it by the median of three elements in the array: the first, last, and the center.
- This is **fast**, and has a good chance of giving us something **close to** the real median.



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.

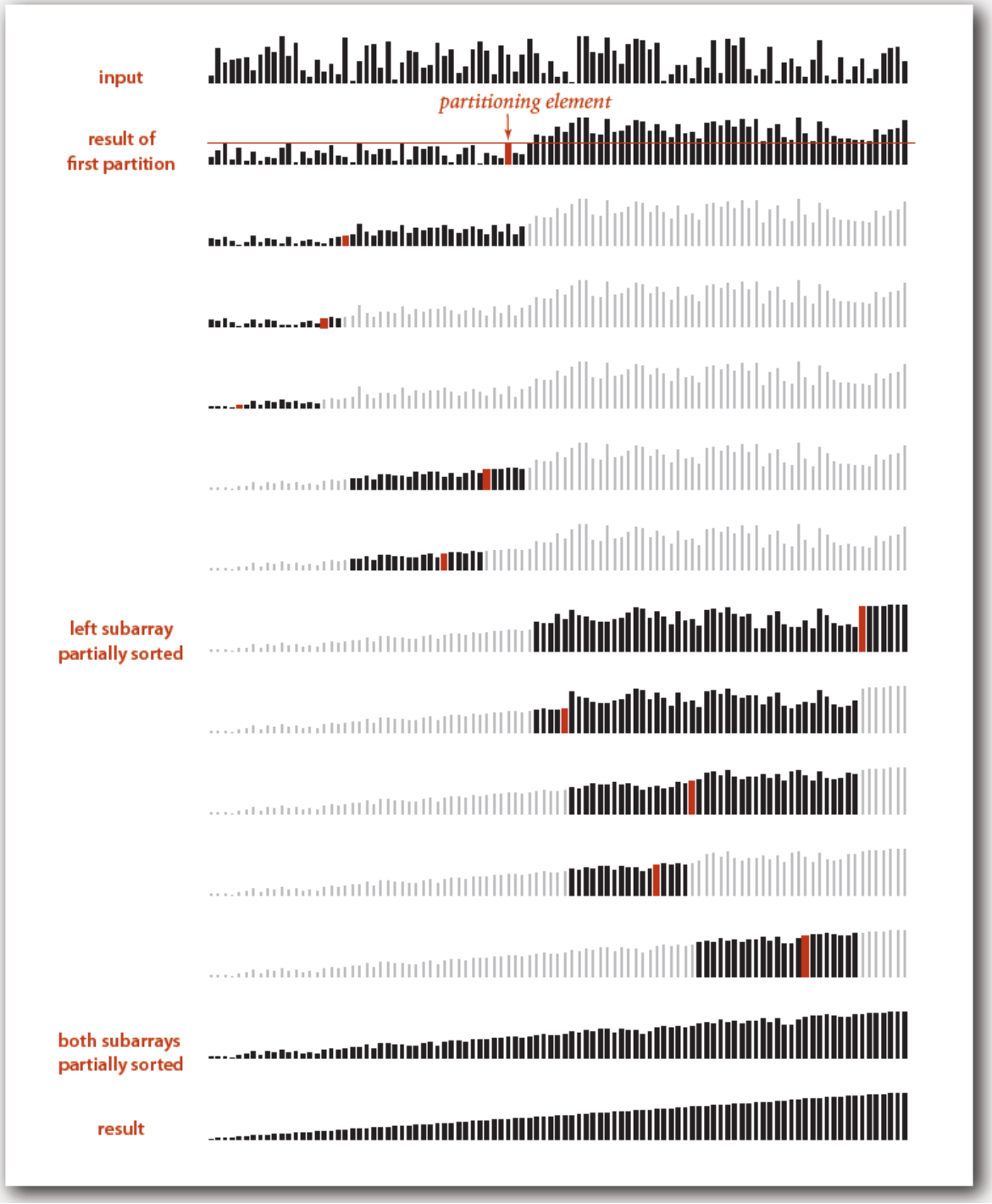
~ 12/7 N ln N compares (slightly fewer)
~ 12/35 N ln N exchanges (slightly more)

```
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



Quicksort Summary

- Quicksort partition the input array to two sub-arrays, then sort each subarray recursively.
- It sorts in-place.
- O(N*logN) cost, but faster than mergesort in practice
- These features make it the most popular sorting algorithm.

Outline

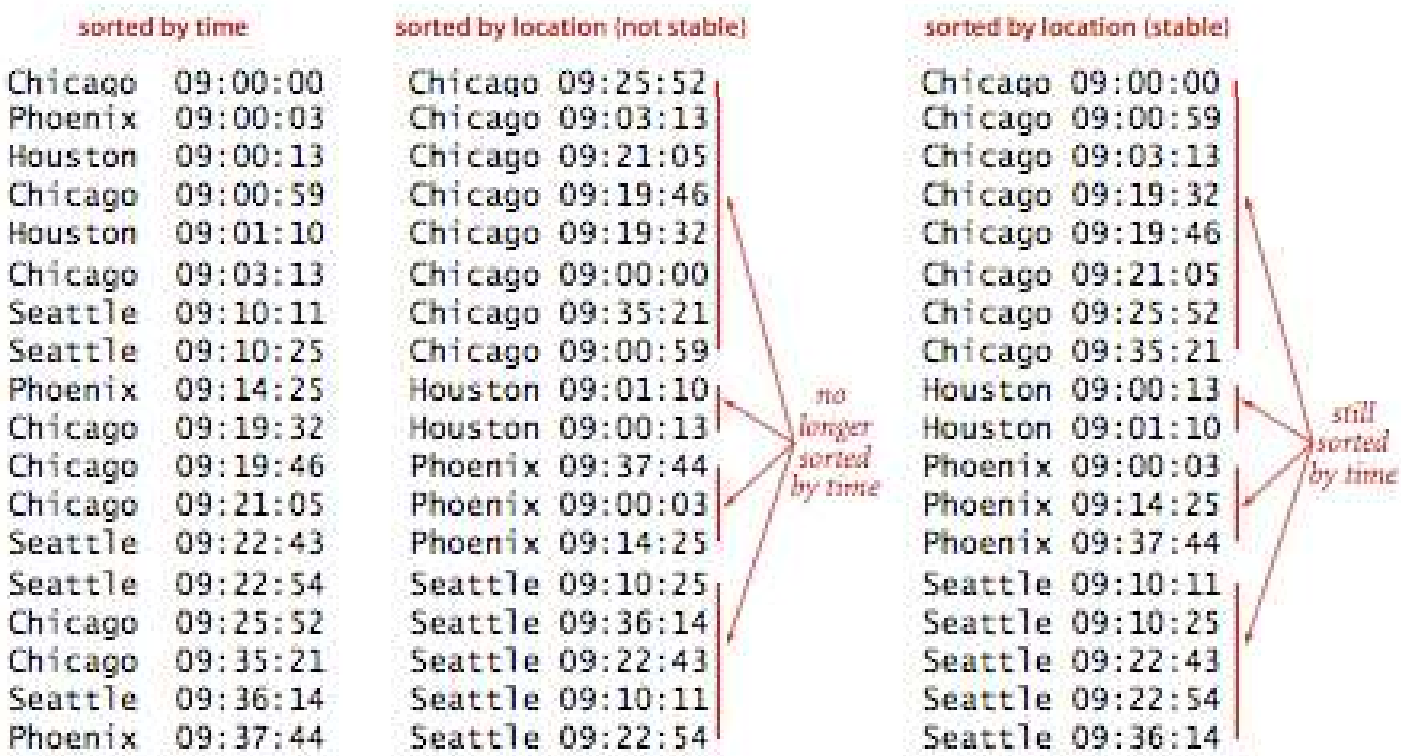
- Quicksort algorithm (review)
- Quicksort Analysis (cont.)
  - Best case
  - Worst case
  - Average case
- Quicksort improvements
- Sorting summary
- Java sorting methods

Sorting Summary

- Elementary sorting algorithms
  - Bubble sort
  - Selection sort
  - Insertion sort
- Advanced sorting algorithms
  - Merge sort
  - Quicksort
- Performance characteristics
  - Runtime
  - Space requirement
  - Stability

Stability

- A sorting algorithm is stable if it preserves the relative order of equal keys in the array
- Stable: insertion sort and mergesort
- Unstable:: selection sort, quicksort



Stability when sorting on a second key

Sorting summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	*N* exchanges
insertion	x	x	$N^2 / 2$	$N^2 / 4$	N	use for small *N* or partially ordered
shell	x		?	?	N	tight code, subquadratic
merge		x	N lg N	N lg N	N lg N	*N* log *N* guarantee, stable
quick	x		$N^2 / 2$	2 N ln N	N lg N	*N* log *N* probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	2 N ln N	N	improves quicksort in presence of duplicate keys
???	x	x	N lg N	N lg N	N lg N	holy sorting grail

Java system sort method

- Arrays.sort() in the java.util library represents a collection of overloaded methods:
  - Methods for each primitive type
    - e.g. sort(int[] a)
  - Methods for data types that implement Comparable.
    - sort(Object[] a)
  - Method that use a Comparator
    - sort(T[] a, Comparator<? super T> c)
- Implementation
  - quicksort (with 3-way partitioning) to implement the primitive-type methods (speed and memory usage)
  - mergesort for reference-type methods (stability)

Example

- Sorting transactions
  - Who, when, transaction amount
- Use Arrays.sort() methods
- Implement Comparable interface for a transaction
- Define multiple comparators to allow sorting by multiple keys

- Transaction.java

http://algs4.cs.princeton.edu/25applications/Transaction.java.html