CS 171: Introduction to Computer Science II

Quicksort

Outline

- MergeSort
  - Recursive Algorithm (top-down)
  - Analysis
  - Improvements
  - Non-recursive algorithm (bottom-up)
- QuickSort
  - Algorithm
  - Analysis
  - Practical improvements

MergeSort

- Merging two sorted array is a key step in merge sort.
- Merge sort uses a divide and conquer approach.
- It repeatedly splits an input array to two sub-arrays, sort each sub-array, and merge the two.
- It requires O(N*logN) time.

- On the downside, it requires additional memory space (the workspace array).

Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1) Insertion.sort(a, lo, hi);
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

Mergesort: practical improvements

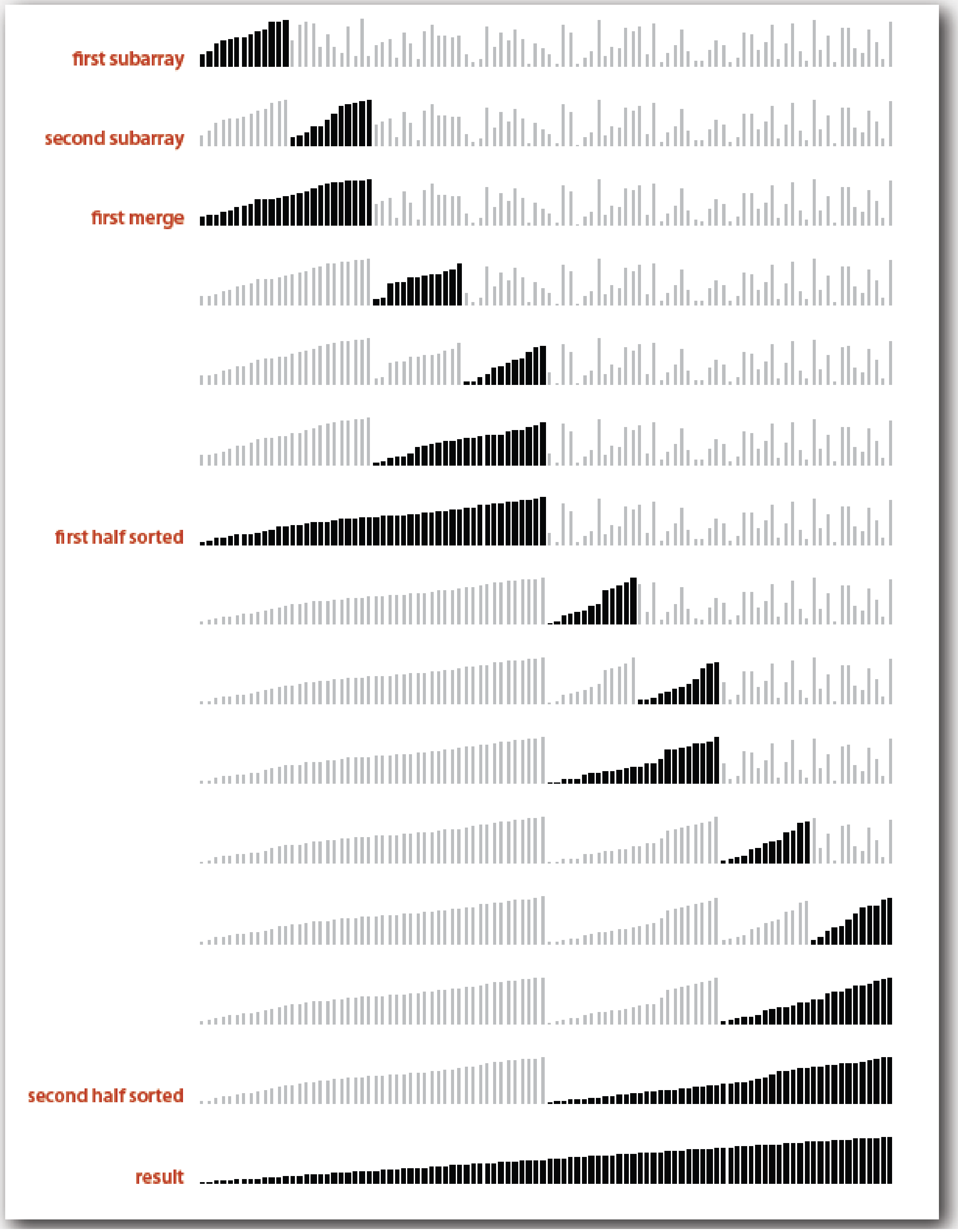
Stop if already sorted.

- Is biggest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.

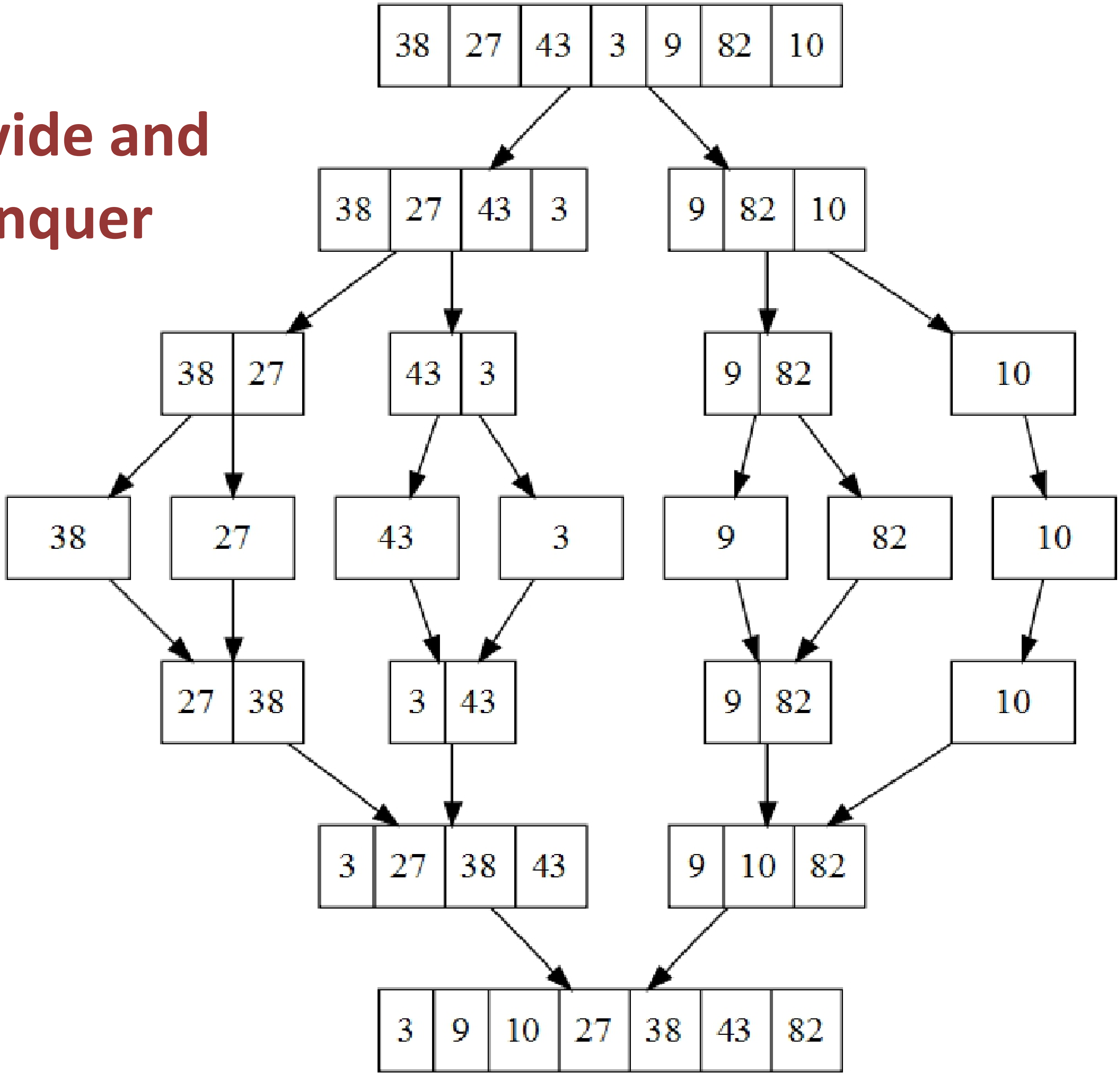
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

Mergesort: visualization



Divide and Conquer



Bottom-up MergeSort

1. Every element itself is trivially sorted;
2. Start by merging every two adjacent elements;
3. Then merge every four;
4. Then merge every eight;
5. …
6. Done.

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ...

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Bottom line. No recursion needed!

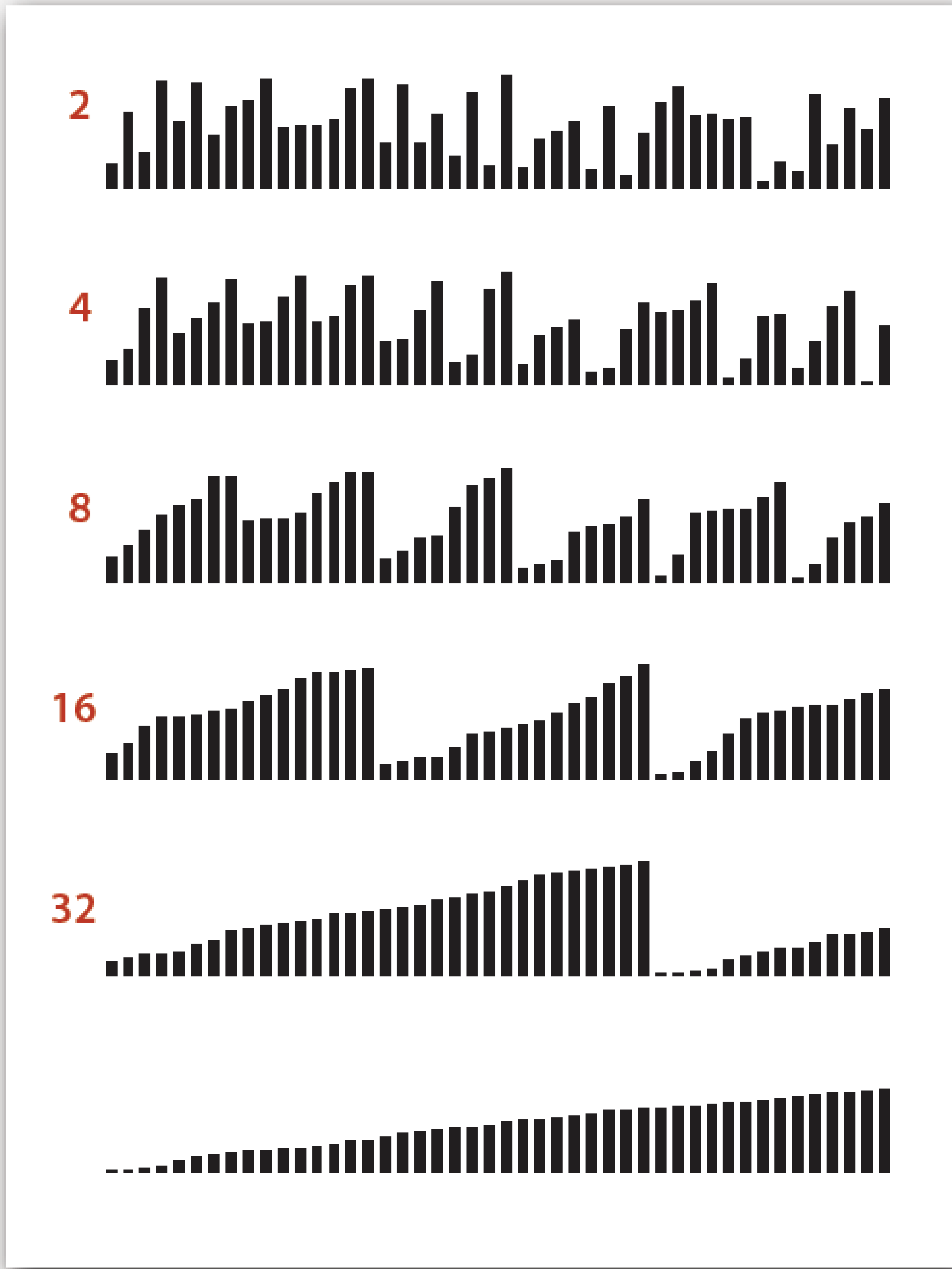
Bottom-up mergesort: Java implementation

```
public class MergeBU
{
   private static Comparable[] aux;

   private static void merge(Comparable[] a, int lo, int mid, int hi)
   {  /* as before */  }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

Bottom-up mergesort: visual trace



Summary of mergesort

- Divide and conquer: split an input array to two halves, sort each half recursively, and merge.
- Can be converted to a non-recursive version.

- O(N*logN) cost
- Requires additional memory space.

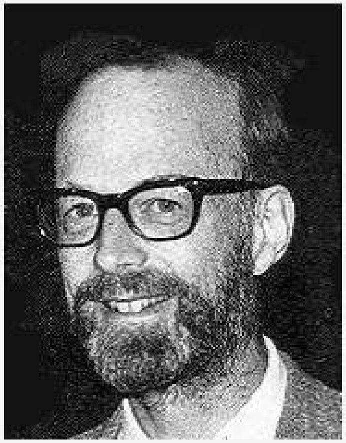
Quick Sort

- The most popular sorting algorithm.
- Divide and conquer.
- Uses recursion.
- Fast, and sort ‘**in-place**’ (i.e. does not require additional memory space)

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

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 element

not greater

not less

Partition (Split)

- A **key step** in quicksort
- Given an input array, and a **pivot value**
- Partition the array to two groups: all elements smaller than the pivot are on the left, and those larger than the pivot are on the right

- Example: K R A T E L E P U I M Q C X O S

  pivot: K

Partition (Split)

- How to write code to accomplish partitioning?
- Think about it for a while.

1. Assume you are allowed additional memory space.
2. Assume you must perform in-place partition (i.e. no additional memory space allowed).

Quicksort uses in-place partitioning

Partition (Split)

- If <u>additional memory space is allowed</u> (using a workspace array)

Loop over the input array, copy elements smaller than the pivot value to the left side of the workspace array, copy elements larger than the pivot value to the right hand side of the array, and put the pivot value in the “middle”

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.

			v								a[i]							
	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

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

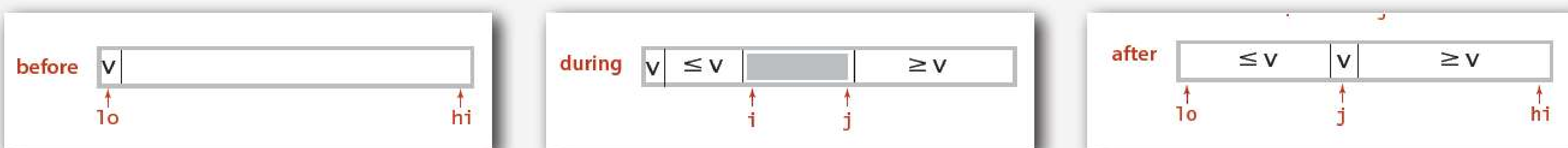
find item on right to swap

check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place



Partition (Split)

Some observations:

- The array is not necessarily partitioned in half.
  - This depends on the pivot value.
- The array is by no means sorted.
  - But we are getting closer to that goal.

- What’s the cost of partition?

Quick Sort

- Partition is the key step in quicksort.
- Once we have it, quicksort is pretty simple:
  - Partition (this splits the array into two: left and right)
  - Sort the left part, and sort the right part (how? What's the base case?)
  - What about the element at the partition boundary?

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

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

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 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,
  - $C(k) = C(0) + C(1) + \ldots + C(N-1) /N$
  - $C(N-k-1) = C(N-1) + C(N-2) + \ldots + C(0) /N$
- Solving the recurrence relations (not required for the course)
  - Approximately, $C(N) = 2N\log N$

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 $\approx$ 10 items.
- Note: could delay insertion sort until one pass at end.

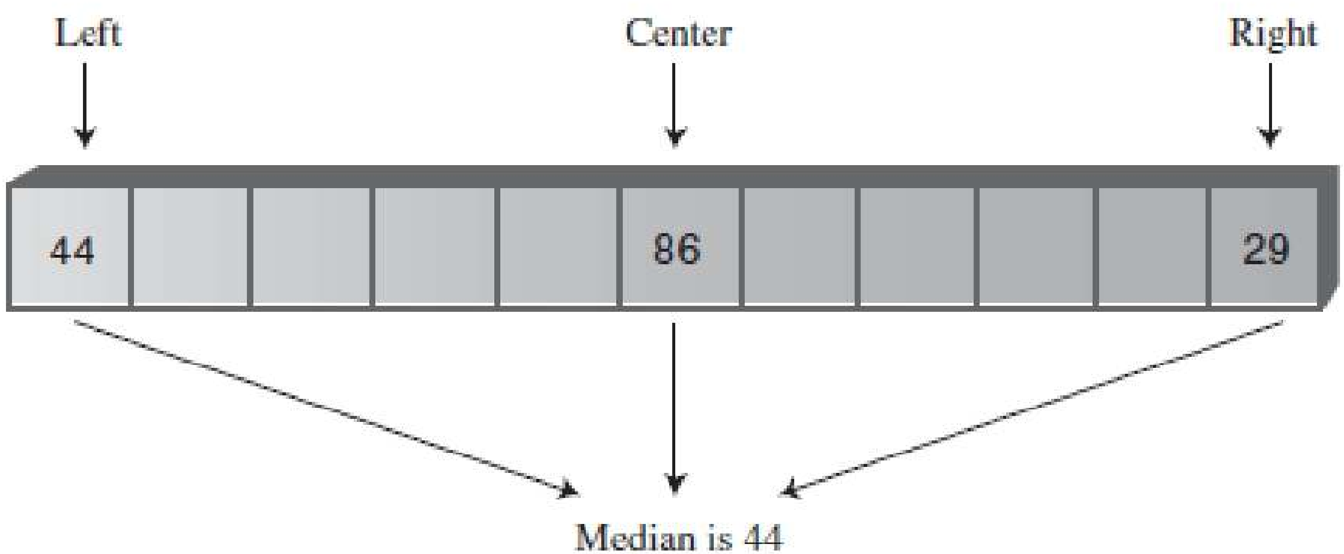
```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```

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.

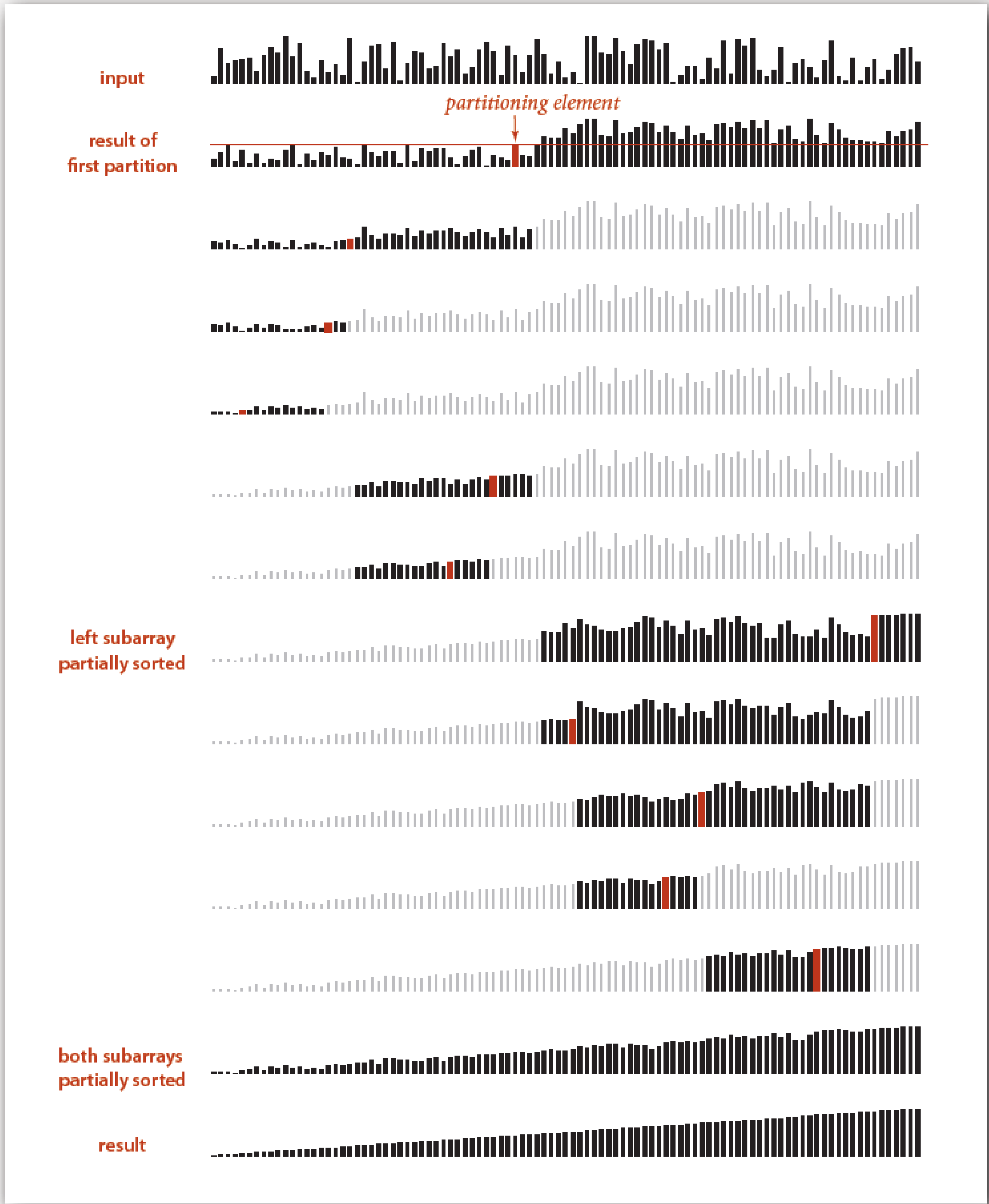
~ 12/7 N ln N compares (slightly fewer)
~ 12/35 N ln N exchanges (slightly more)

```
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



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.

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 Arrays.sort() Methods

- In Java, Arrays.sort() methods use **mergesort** or a tuned **quicksort** depending on the data types
  - Mergesort for objects
  - Quicksort for primitive data types
- switch to **insertion sort** when fewer than seven array elements are being sorted

Reminders

- Hw3 with 1 late credit is due today
- Hw4 is due Friday
- Enjoy your Spring break!