

Mergesort and Quicksort

Nicholas Moore

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 2.2), Prof. Mhaskar's course slides, and <https://www.cs.princeton.edu/~rs/AlgsDS07/04Sorting.pdf>

Mergesort!

Over the next few lectures, we will study two **Divide and Conquer** algorithms: Mergesort and Quicksort.

- The goal with divide and conquer is to divide a problem into two or more **sub-problems**, which are similar to or the same as the original problem.
- This sub-problem division continues until problems are easy to solve directly.
- Finally, all of the subproblems are combined in such a manner as to produce the desired result.

Mergesort!

The Mergesort divide and conquer algorithm proceeds as follows.

- Given an array to sort, we decide if it is small enough to solve directly (i.e, only one element!).
- If not, we divide the array in half and recursively call Mergesort on both halves in sequence.
- Then, we combine the two halves and pass the sorted array to the calling function.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

Merging on the Highway

Up to now we've only talked about runtime complexity, but we can also examine algorithms in terms of **memory complexity**.

- In general, we are concerned with the amount of memory an algorithm uses *aside from its inputs*.
- Sorts which don't require separate arrays to be instantiated are called "in-place" sorts.
- Because of the use of the `aux[]` array above, mergesort is not an in-place algorithm.
- In-place algorithms require $N + O(\lg N)$ memory.

Mergesort Algorithm: Trace

lo hi
 $merge(a, aux, 0, 0, 1)$
 $merge(a, aux, 2, 2, 3)$
 $merge(a, aux, 0, 1, 3)$
 $merge(a, aux, 4, 4, 5)$
 $merge(a, aux, 6, 6, 7)$
 $merge(a, aux, 4, 5, 7)$
 $merge(a, aux, 0, 3, 7)$
 $merge(a, aux, 8, 8, 9)$
 $merge(a, aux, 10, 10, 11)$
 $merge(a, aux, 8, 9, 11)$
 $merge(a, aux, 12, 12, 13)$
 $merge(a, aux, 14, 14, 15)$
 $merge(a, aux, 12, 13, 15)$
 $merge(a, aux, 8, 11, 15)$
 $merge(a, aux, 0, 7, 15)$

a[]															
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
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

result after recursive call

Polymerization!

Some things of note about the foregoing algorithm:

- Note that, while it's theoretically possible to merge any two subarrays, the two subarrays merged by mergesort are **always adjacent**.
- Arguments to merge:
 - $a[] \Rightarrow$ the array we are sorting
 - $aux[] \Rightarrow$ a pre-declared region of memory for swap space.
 - $lo \Rightarrow$ Beginning of the first half
 - $mid \Rightarrow$ End of the first half
 - $hi \Rightarrow$ End of the second half

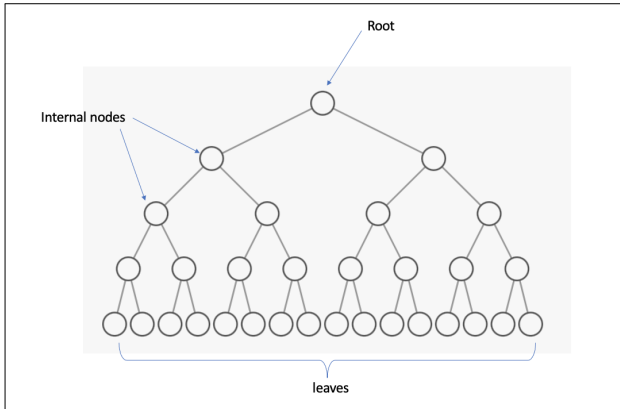
Mergesort analysis: Memory

- How much memory does mergesort require?
 - Original input array = N .
 - Auxiliary array for merging = N .
 - Local variables: constant.
 - Function call stack: $\lg N$.
 - Total = $2N + O(\lg N)$.
- *How much memory do selection sort and insertion sort require?*
- In-place merger is complicated - see [Kronrud, 1969]

Binary Trees

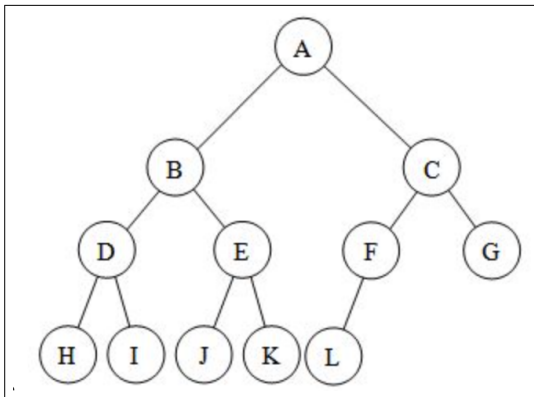
Binary Tree: A tree where each internal node has **at most two children**.

Full Binary Tree: A binary tree where each internal node has exactly two children.



Binary Trees contd..

Complete Binary Tree: A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



<https://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/FullvsComplete.html>

Mergesort Time Complexity

Proof. Sketch.

The number of compares $C(N)$ to mergesort an array of length N satisfies the recurrence:

$$C(N) \leq \underbrace{C(\lceil N/2 \rceil)}_{\text{left half}} + \underbrace{C(\lfloor N/2 \rfloor)}_{\text{right half}} + \underbrace{N}_{\text{merge}} \quad \text{for } N > 1, \text{ with } C(1) = 0,$$

where $\lceil x \rceil$ is the smallest integer $\geq x$, i.e. $\lceil 1.5 \rceil = 2$, $\lceil 3.1 \rceil = 4$,
and $\lfloor x \rfloor$ is the biggest integer $\leq x$, i.e. $\lfloor 1.5 \rfloor = 1$, $\lfloor 3.1 \rfloor = 3$.

We solve the recurrence when N is a power of 2:

$$D(N) = 2D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

The result holds for all N , but general proof is a little bit messy. □

Recursion Tree

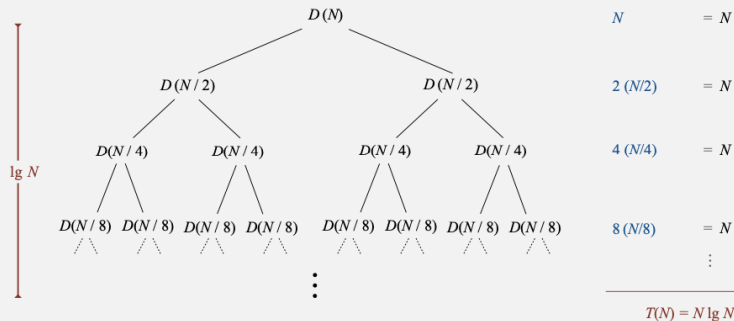
A **recurrence relation** is an equation that defines a sequence based on a rule that gives the next term as a function of the previous terms.

- A **recursion tree** is useful for visualizing what happens when such an equation is iterated, documenting not only the number of recursive calls, but also the amount of work done at each step.
- In a recursion tree, each node represents the size and cost of each subproblem.
- We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

Mergesort Recursion Tree

Proposition. If $D(N)$ satisfies $D(N) = 2 D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]



Mergesort: Explanation of time complexity

Mergesort divides arrays into two equal parts at each step.

- The second level of the recursion tree contains 2^1 subarrays of size $\frac{n}{2^1}$.
- The third level has 2^2 subarrays of size $\frac{n}{2^2}$.

In general, assuming levels are zero-indexed, each level has 2^h arrays of size $\frac{n}{2^h}$.

- We are processing $2^h \cdot \frac{n}{2^h}$ items per level in the tree, so each level is $O(n)$, but *how many levels are there?*
- By construction, Mergesort creates complete binary trees.
- The number of levels in a full binary tree with n nodes in it's deepest level is $\lg(n)$

Therefore, we have n items being processed $\lg(n)$ times, making Mergesort a $O(n \lg n)$ (linearithmic) algorithm!

Solving Recurrence Equations

Consider the size of the sub arrays at each iteration.

- If the algorithm creates **equal sub-arrays**, then every path of the recursion tree has the same height h .
- However, if the algorithm creates **unequal sub-arrays**, this property no longer holds. To determine the tree height we must examine the longest path.
- At each division, the sub-array with the most items will take longest to be sub-divided down to arrays of one item.
- Therefore, longest path always follows the largest sub-arrays, and this path is the height of the tree.

Solving Recurrence Equations II

- To determine the height of this longest path, we formulate an equation $n/k^h = 1$, where k is the fraction by which n is divided at each level, to get the largest sub array.
- Computing the time taken at each level and multiplying it with h will give you the order of the total complexity of the algorithm.

Solving a Particular Recurrence Equation

The equation $T(n) = T(n/10) + T(9n/10) + cn$ is divided unevenly.

- The largest sub-arrays dictate tree height.
- At the second level, the largest subarray is of size $\frac{9n}{10}$.
- At the next level, the largest subarray is of size $\frac{9^2n}{10^2}$, and so on.
- At maximum depth, $\frac{n}{(10/9)^h} = 1$, where h is our maximum tree depth.
- Solving the above equation yields

$$h = \log_{10/9} n$$

- Each level of the tree still takes cn , so with h levels the total time taken is $O(n \log_{10/9} n)$.

Mergesort: Bottom-up

- Pass through array, merging as we go to double size of sorted subarrays.
- Keep performing the passes and merging subarrays, until you do a merge that encompasses the whole array.

Mergesort: Bottom-up Java

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

Trace of merge results for bottom-up mergesort

Min Top-Down Comparisons

Proposition: Top-down mergesort uses between $1/2N \lg N$ and $N \lg N$ comparisons.

Let the total number of comparisons be $C(N)$:

$$C(N) \leq \underbrace{C(\lceil N/2 \rceil)}_{\text{left recursion}} + \underbrace{C(\lfloor N/2 \rfloor)}_{\text{right recursion}} + \underbrace{cN}_{\text{comparisons at this level}}$$

- The smallest number of comparisons made by MERGE is $\frac{N}{2}$.
 - If one sub-array contains all the smallest elements, we only walk that array before appending the other.
- If you sort a sorted array, this would be the case at every level.
- Solving the above recursion with $c = \frac{1}{2}$ yields $C(N) = 1/2N \lg N$.

Max Top-down Comparisons

- Similarly, the maximum number of comparisons is made when both sub-arrays must be fully examined.
- If the input array is of size N , then at most N comparisons are made.
- If the above happens at every level of the recursion, the total number of comparisons at each level is at most N .
- By solving the same recurrence equation with $c = 1$, we get $C(N) = N \lg N$.

Max Top down Accesses

Proposition. Top-down mergesort uses at most $6N \lg N$ array accesses to sort an array of length N .

- Each merge uses at most $6N$ array accesses
 - $2N$ to copy the sub arrays initially
 - $2N$ to put the values back (in order)
 - At most N comparisons, each accessing two array elements ($2N$)
- Hence the total number of array accesses after solving the recurrence is most $6N \lg N$.

Min/Max Bottom Up Comparisons/Accesses

Proposition. Bottom-up mergesort uses between $1/2N \lg N$ and $N \lg N$ compares and at most $6N \lg N$ array accesses to sort an array of length N .

- The number of passes through the array is precisely $\lg N$.
- For each pass, by the same argument made as in the case of the top-down mergesort approach, the number of array accesses is exactly $6N$ and the number of compares is at most N and no less than $N/2$.
- Therefore, bottom-up mergesort uses between $1/2N \lg N$ and $N \lg N$ compares and at most $6N \lg N$ array accesses to sort an array of length N .

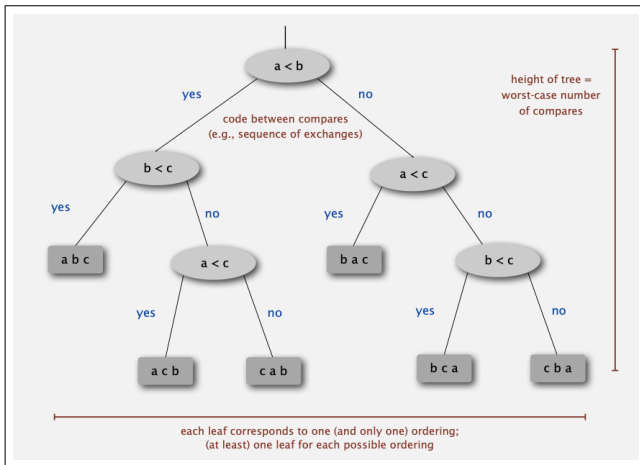
Practicality!

A few small improvements

- Mergesort has too much overhead for tiny subarrays.
Therefore, use insertion sort arrays of < 10 items.
- You can quickly check if two arrays are already sorted using $a[mid] \leq a[mid + 1]$. If so, nothing more needs to be done. Stop if already sorted; that is, do not call merge if the sub arrays are sorted. This can be done by a simple check

Lower Bound for Sorting: Decision Tree

Recall that there are $n!$ permutations of n elements. The comparison decisions which result in these permutations can be represented as a decision tree.



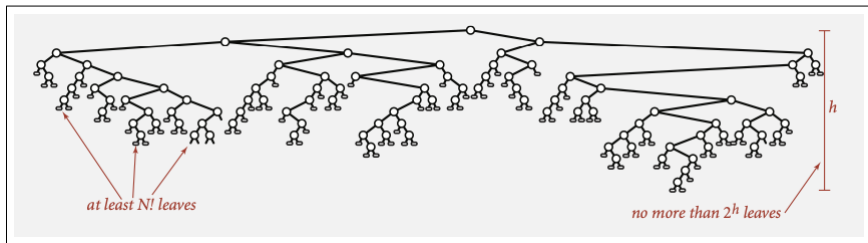
Lower Bound for Sorting

Proposition: Any compare-based sorting algorithm must use at least $N \lg N$ compares; that is, $T(N) = \Omega(N \lg N)$.

Proof Sketch:

- Assume array consists of N distinct values a_1 through a_N .
- Worst case dictated by height h of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N!$ different orderings \Rightarrow at least $N!$ leaves.

Lower Bound for Sorting



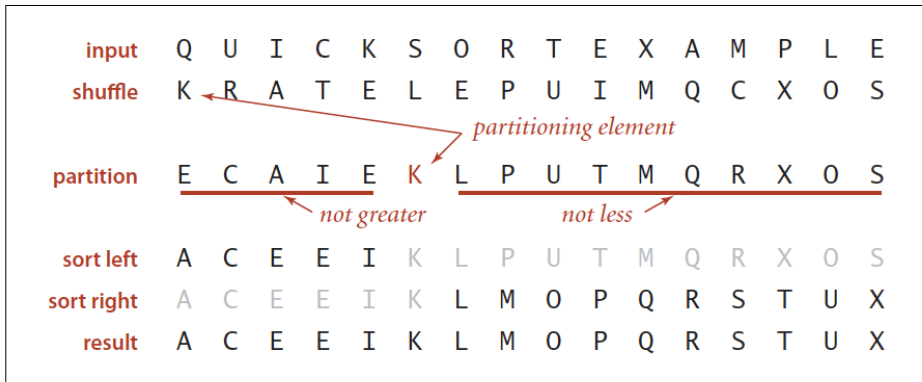
- $2^h \geq N! \Rightarrow h \geq \lg(N!)$, by Stirling's approximation we have $\lg(N!) \approx N \lg N$ https://en.wikipedia.org/wiki/Stirling%27s_approximation
- Hence $h \geq N \lg N$. Therefore, any compare-based sorting algorithm must use at least $N \lg N$ compares in the worst-case.

Quicksort - Idea

Quicksort was rated one of the top 10 algorithms of the 20th century. Here's the overall approach:

- Shuffle the array to eliminate dependence on input.
- Select first element of array as **pivot**.
- Create two sub arrays from remaining elements via in-place swapping.
 - One containing all the elements less than (or equal to) the pivot.
 - The other containing all the elements greater (or equal to) than the pivot.
- Recurse on both subarrays.
- Profit!

Quicksort Visualized



Quicksort Full 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 Java I

ALGORITHM 2.5 Quicksort

```
public class Quick
{
    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);           // Eliminate dependence on input.
        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);    // Partition (see page 291).
        sort(a, lo, j-1);                 // Sort left part a[lo .. j-1].
        sort(a, j+1, hi);                 // Sort right part a[j+1 .. hi].
    }
}
```

Quicksort Java II

Partition returns the index of the pivot.

Quicksort partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{ // Partition into a[lo..i-1], a[i], a[i+1..hi].
  int i = lo, j = hi+1;           // left and right scan indices
  Comparable v = a[lo];           // partitioning item
  while (true)
  { // Scan right, scan left, check for scan complete, and exchange.
    while (less(a[++i], v)) if (i == hi) break;
    while (less(v, a[--j])) if (j == lo) break;
    if (i >= j) break;
    exch(a, i, j);
  }
  exch(a, lo, j);                 // Put v = a[j] into position
  return j;                       // with a[lo..j-1] <= a[j] <= a[j+1..hi].
}
```


Quicksort Partitioning

- First, we select the item we are partitioning on (v).
- We then set up a left and a right scan.
 - We scan from the right until we hit a value greater than v .
 - Then we scan from the left until we hit a value less than v .
 - we swap these two values and then continue the scan.
- The two scans will eventually meet in the middle, and that's where we put v .
- The two partitions are guaranteed to be lower/higher than v , but not guaranteed to be sorted, so we recursively sort them.

Quicksort Partitioning Example

	i	j	v	a[]															
				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		5		E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Quicksort Analysis

- Using an auxillary array eliminates the need to perform the scan partition, simplifying the algorithm, but is not worth the cost.
- Shuffling is key for or performance guarantee. An alternate way to preserve randomness is to choose the pivot randomly.
- When the scan hits an element identical to the pivot, it is (counter-intuitively) best to stop the scan. This aids in avoiding quadratic running time in certain applications.

Quicksort Performance Characteristics

Quick Sort is considered the fastest sorting when input is highly disordered and ($N > 15$).

- Half of all cases execute **very** quickly.
- The inner loop is very short.
 - All we're doing is incrementing an index and comparing an array entry to a fixed value.
 - It is difficult to have less going on in the inner loop.
 - Both mergesort and shellsort move data within their inner loops.

Quicksort Best/Worst Cases

- **Best case:** Number of comparisons is $\approx N \lg N$; that is, $\Omega(N \lg N)$
 - Partitioning always creates equal size subarrays.
 - Recurrence relation is $T(n) = 2T(n/2) + cn$
 - Therefore, $T(n) \in \Omega(N \lg N)$.
- **Worst case:** Number of comparisons is $\approx 1/2N^2$; that is, $O(N^2)$
 - Partitioning always results in one partition of size zero.
 - The recurrence relation is $T(n) = T(n-1) + T(0) + cn$,
 - Therefore $T(n) \in O(N^2)$.

Quicksort running time Analysis

- **Average case:** Expected number of comparisons is $\approx 1.39N \log N$; that is, $\Theta(N \log N)$.
- Although mergesort also has the same running time and quicksort does 39% more compares, quicksort is typically faster as it does much less data movement.

Quicksort: Practical improvements

All the below are suggestions validated with refined math models and experiments.

- Best choice of pivot element = median.
- Even quicksort has too much overhead for tiny subarrays. Therefore, use insertion sort for < 15 items.
- There is a non-recursive version using stacks which always sorts the smaller half first.

Sorting Summary

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort ($N \log N$)

thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

Quicksort ($N \log N$)

thousand	million	billion
instant	0.3 sec	6 min
instant	instant	instant