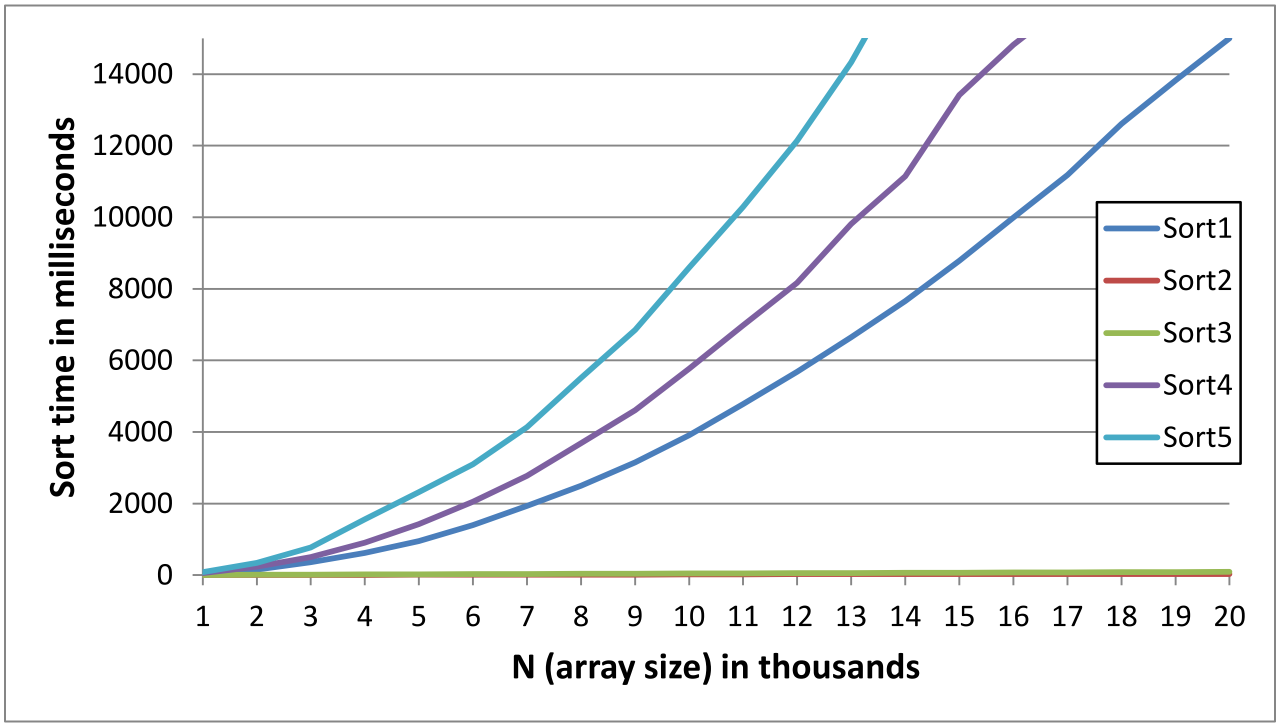
Brian Scoles – July 2014
Solutions for Stanford CS106b Assignment 5 Part A

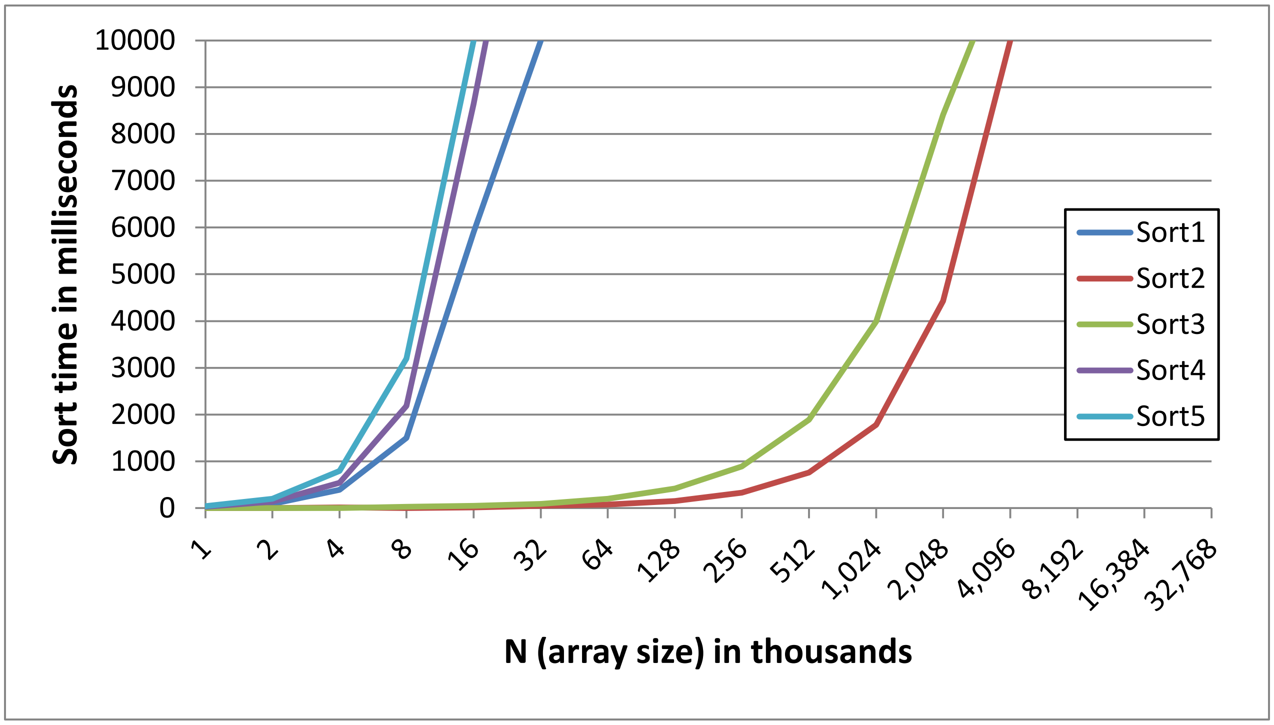
Solutions:

MysterySort1:	Insertion sort
MysterySort2:	Quicksort
MysterySort3:	Merge sort
MysterySort4:	Selection sort
MysterySort5:	Bubble sort

Explanation:
I began by running each sort algorithm through a battery of timed trials (sorting integers of random values between 0 and 100,000) and graphing the results. Here are the results when I linearly increase the size of the array to be sorted, starting at N=1,000 and going up to N=20,000.



You can see that Sort1, Sort4, and Sort5 are increasing logarithmically: this implies they run in O(N^2) time. Sort 2 and Sort 3 are clearly the efficient ones of this bunch; their runtime remains basically 0 for N <= 20,000. Let's try dramatically, nay, exponentially increasing N!



Sort2 and Sort3 begin slowing down more noticeably as N approaches 1 million. At this point, just by looking at the graph, I can hazard some guesses:

- Sort5 is the absolute slowest, so it's probably Bubble sort.
- Sort4 and Sort1 are almost as slow (and they are quadratic, as observed above), so they are probably Selection sort and Insertion sort. Can't yet say which is which.
- Sort2 is the absolute fastest, so I'm going to guess that it's Quicksort.
- That would mean Sort3 must be Merge sort, which seems plausible: it runs much faster than the quadratic algorithms, and almost as fast as Quicksort.

So let's put these guesses to the test!

Quicksort (when it always uses the first element as its pivot value, as is the case in this exercise) is unique in that its worst-case scenario is sorting an already-sorted list. So I fed a sorted list of 20,000 elements to each of the sorting algorithms, and observed that every algorithm handled this array just fine, except for Sort2, whose sort time ballooned massively! It took 5.9 seconds for Sort2 to finish the pre-sorted array, versus the 0.034 seconds it required to sort a scrambled array of the same size.

Confirmed: MysterySort2 = Quicksort!

That wasn't the only interesting result of testing with an already-sorted array. Sort5 and Sort1 both completed this test in 0 milliseconds, versus a time of >15 seconds for a randomly ordered array of the same size. This should mean that they are Bubble sort and Insertion sort, because those are the two algorithms that will stop after 1 iteration through the list if the list is found to be already in order. By elimination, this means that Sort4 must be Selection sort.

A rigorous way to examine each algorithm is to interrupt it partway through its sort. This allows us to see in what order it has been rearranging the data. Note that most of the Input and Yield values shown below have been simplified for readability. Red bars mark the "seam" between different patterns in the output stream.

Sort1 interruption:

Input:	[9, 2, 8, 5, 7, 0, 4, 6, 1, 3]
Yield:	[2, 5, 8, 9, | 7, 0, 4, 6, 1, 3]

Notice that before the seam in the output array, all numbers are in sorted order, and after the seam, the sequence is still in its original state. This is the work of Insertion sort: it works towards the right, taking one element at a time and inserting it into its correct place in the already sorted section to the left of the seam. The elements to the right of the seam are still in their original order because the algorithm hasn't touched them yet.

Confirmed: MysterySort1 = Insertion sort!

Sort3 interruption:

Input:	[2,000 ints, random order, range 1-50]
Simplified Yield:	[1, 2, 4, 11, 15, 21, 32, 45, | 1, 17, 28, 45 | 37, 12, 19, 4, 22, 34, 50, 26, …]

Here you can observe that the data is being sorted section by section. The leftmost section is the largest, and it is followed by smaller sorted sets. The rightmost section of the array is still in its original state. This is the work of Merge sort, which breaks the array down recursively into smaller and smaller pieces and then merges the pieces back together, sorting as it goes.

Confirmed: MysterySort3 = Merge sort!

Sort4 interruption:

Input:	[9, 2, 8, 5, 7, 0, 4, 6, 1, 3]
Yield:	[0, 1, 2, 3, | 9, 8, 5, 7, 4, 6]

All of the very smallest values appear in the correct order on the left-hand side of the seam. The rest of the data is still in its original relative order. This is the work of Selection sort, which iterates through the whole unsorted portion of the array, finds the minimum value, and moves that value into the rightmost position of the sorted section.

Confirmed: MysterySort4 = Merge sort!

Sort5 interruption:

Input:	[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Yield:	[7, 6, 5, 4, 3, 2, 1, 0, | 8, 9]

The highest values have been shifted to their proper locations at the very end of the array, and everything else is still in its original relative order. This is the work of Bubble sort, because the simple swapping behavior causes high values to “bubble up” to the end of the list.

Confirmed: MysterySort5 = Bubble sort!