CSCE 221 Project Report

***Introduction***

This project tests our knowledge and implementation of eight different sorting algorithms: bubble, selection, insertion, heap, merge, deterministic quick sort, randomized quick sort, and radix sort. These eight are unique in the ways that they navigate through a list of numbers, and re-position (or swap) themselves to sort the data. Each of these previously listed algorithms will be tested and graphed to document their relative efficiencies. Overall, the objective is to see which of the previously listed sorting algorithms runs the fastest, and describe ways one could use them in everyday coding to boost performance.

***Theoretical Analysis***

The first of the sorting algorithms we will analyze is the bubble sort. The bubble sort repeatedly navigates through a list of numbers, compares each pair of adjacent items to one another, and swaps them if they are in the wrong order. In the best case, bubble sort runs in O(n) time, or when the data is nearly sorted. However, in the worst case, it runs in O(n2), when the data is in reversed order. On average, bubble sort runs in O(n2), and we will use this to compare to the others.

Next, insertion sort is another one of the more common sorting algorithms. It builds the final sorted array one item at a time, navigating through a list of data and picking out the smallest number still remaining. This works well because it can dynamically sort data as it comes in, but is only efficient with relatively small data sets. Similar to bubble sort, it runs in O(n) time in the best case, O(n2) in the worst, and on average in O(n2). However, bubble sort only compares the current element to its neighbors, while insertion compares the current element to the entire list of data.

Heap sort is a more abstract way of sorting a list of data; it divides its input into a sorted and an unsorted region, and iteratively shrinks the unsorted region by extracting the largest element and moving it to the sorted region. This improvement of the insertion sort uses the heap data structure rather than a linear-time search to find and remove the maximum element. In all cases, heap sort runs in O(n log n) time.

The merge sort is a divide-and-conquer method that breaks up the list into its smallest unit (1 element), and compares each element to the adjacent list to sort and merge the two adjacent lists. Finally, the elements are all sorted and merged to produce a sorted list of data. Similar in structure to heap sort, it also theoretically runs in the same time, in all cases running in O(n log n). However, merge sort’s unique merging technique distinguishes it from the heap sort, which uses an iterative insertion approach.

We will utilize two different quick sorting algorithms: deterministic and randomized. Deterministic quick sort is similar to merge sort in its nature, where it breaks up the array into two sub arrays at a pivot value, and recursively sorts the two in a divide-and-conquer type algorithm. What differentiates this sort from the merge sort its placement of the pivot point: merge sort is as close to the middle of the array as it can get. The randomized quick sort literally places the pivot at a random spot within the array, and the deterministic sort loops through and selects the spot in which the pivot would make the sort the most efficiently. Both of these run in O(n log n) in the best and the average case, as does the merge sort.

Finally, the last sort we will analyze is the radix sort. This is a non-comparative sorting algorithm that sorts each integer by one place value at a time (ones, tens, hundreds). The deciding factor is how the keys of each element are distributed. The best case for radix sort is that they are taken as consecutive bit patterns. It will run in O(kn) time, which is always the case for radix.

When theoretically analyzing this data, there are three cases we need to consider: an already sorted list of data, an unsorted list of data, and a list that is already sorted in reverse order. When considering an already sorted list, the bubble, insertion and radix will run in the fastest times (O(n)) due to their comparative nature. Following this is both quick sorts, merge, and heap sorts running in O(n log n) with an already sorted list. Finally, the selection sort will run through a sorted list in O(n2) time. Next, a completely unsorted list will cause each algorithm to run at pretty average run times. The fastest of these will be the radix sort in O(kn) time, followed by quick sorts, merge and heap at their pretty constant O(n log n), and finally bubble, selection, and insertion run in O(n2). Lastly, when we analyze sorts on a reversed list of data, radix still runs in its constant O(kn). This is followed by merge and heap, which both run in their constant O(n log n), and the rest of the sorts that run in O(n2). We will show in our analyses the best sorts for each of these cases, each of them having their own respective graphs.

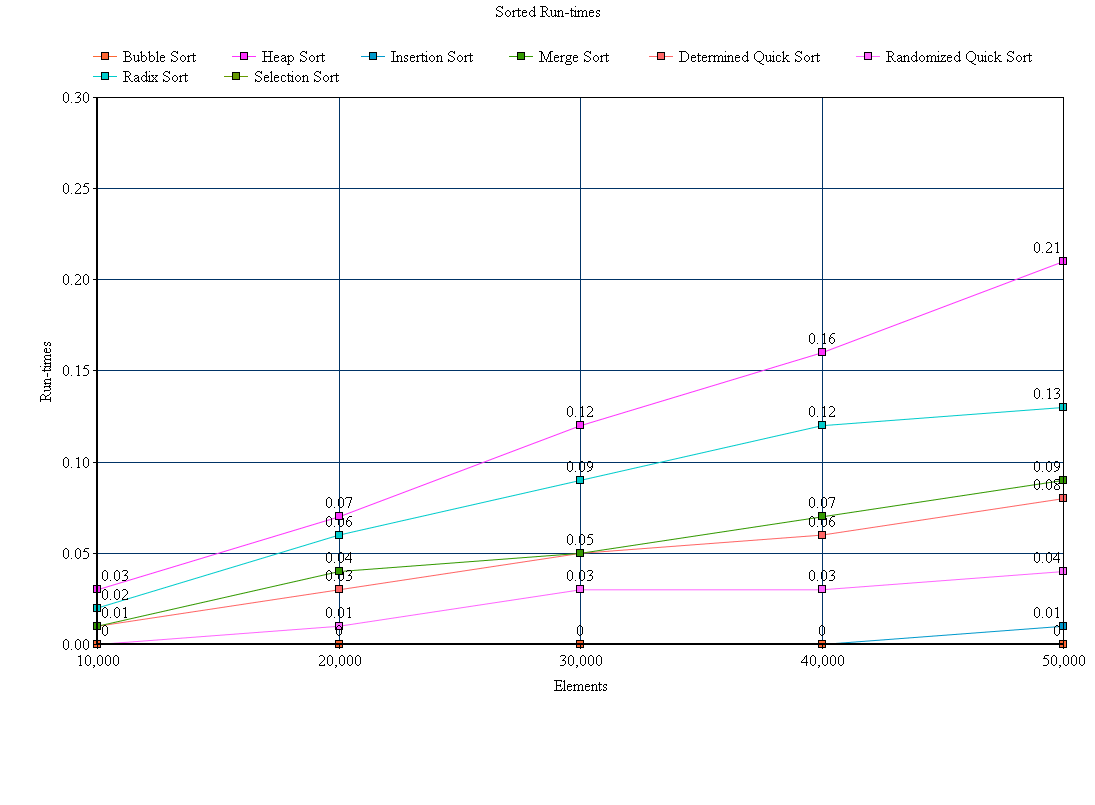
***Experimental Setup***

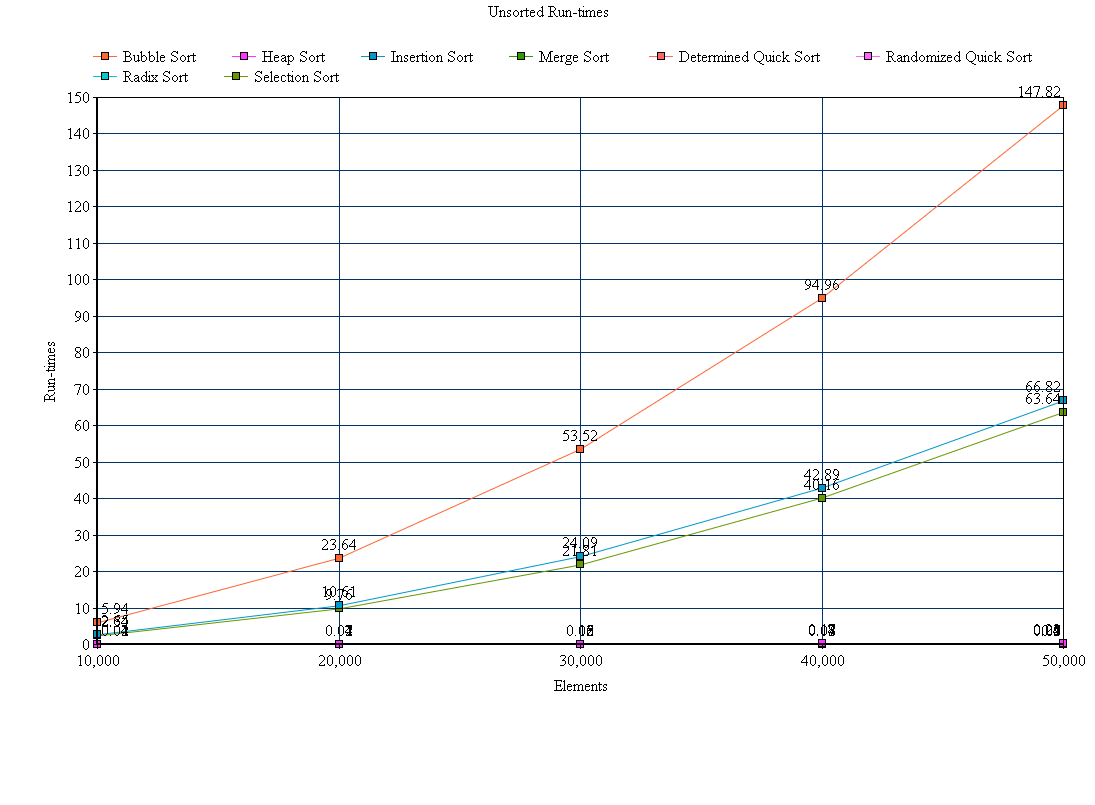
I ran this test on my MacBook Pro (v. 10.10.2) using TextWrangler and the built-in G++ (Linux) compiler.

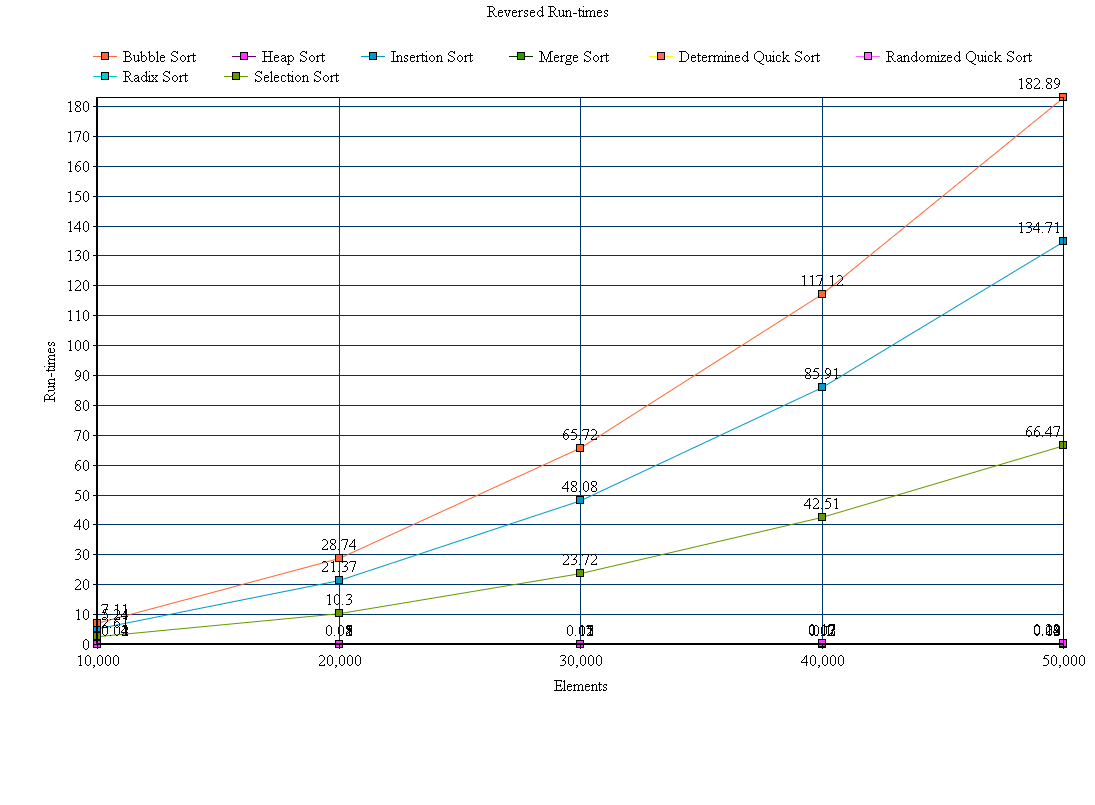
Initially writing and testing our data, we used a simple, reversed, 10-number list of data, just to test the functionality of each algorithm with almost immediate feedback. However, we used a larger set of data actually testing and analyzing the run times of the data. To keep the data uniform across the board, we used the same 3 files for each sort (one sorted, one unsorted, one reversed). Each of these contained the same series of 50,000 randomly generated numbers, where the original file was used as the unsorted. The sorted and reversed lists were generated using modifications of our own sorting algorithms. We ran each test a series of 4 different times, and the points on the graphs are represented by each sort’s average running time.

***Experimental Results***

First, we analyzed the run times for each of the respective search algorithms, using sorted, unsorted, and reversed sequences of data. We used sequences of 50,000 numbers 4 times, and the numbers provided are averages of the 4 trials. The data for each sort is graphed below, with respect to their run times and the total number of elements.

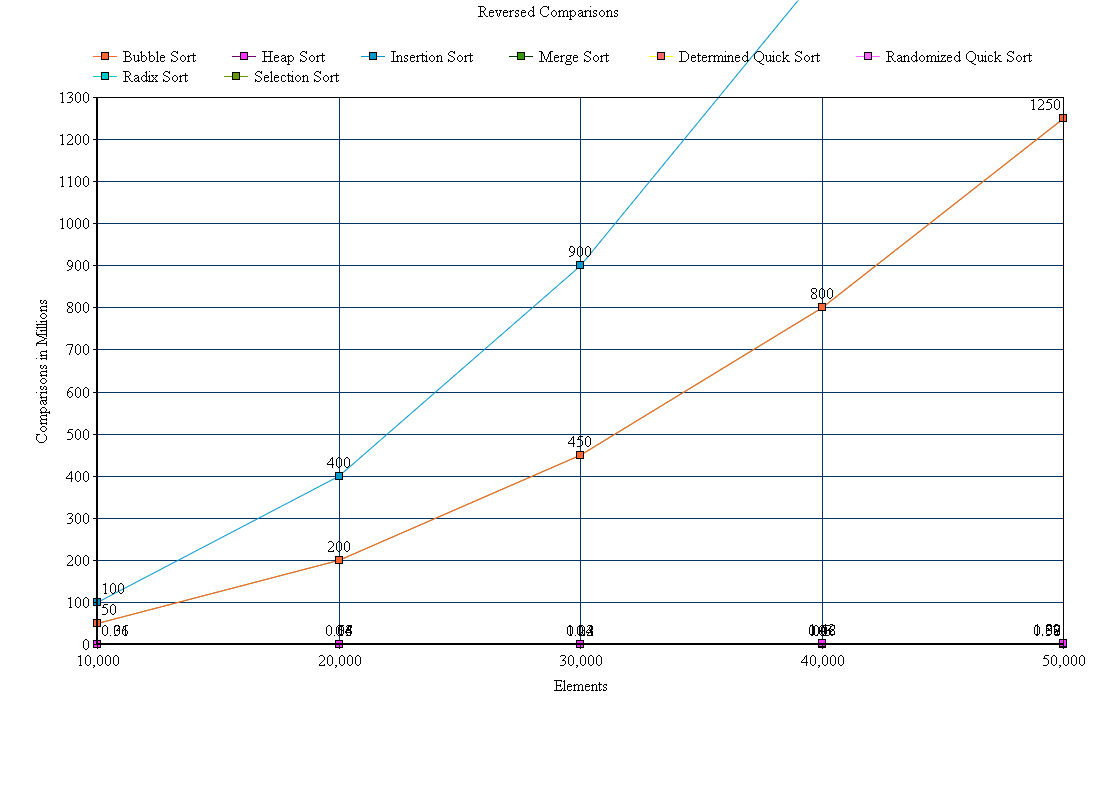


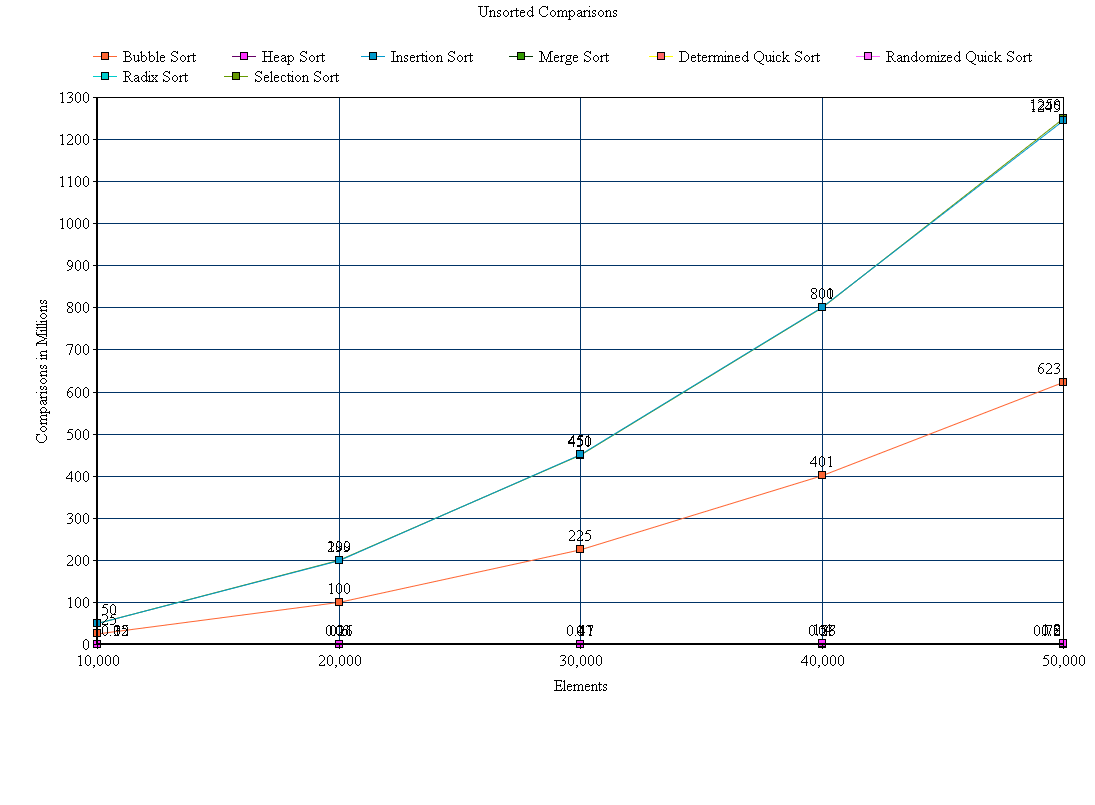


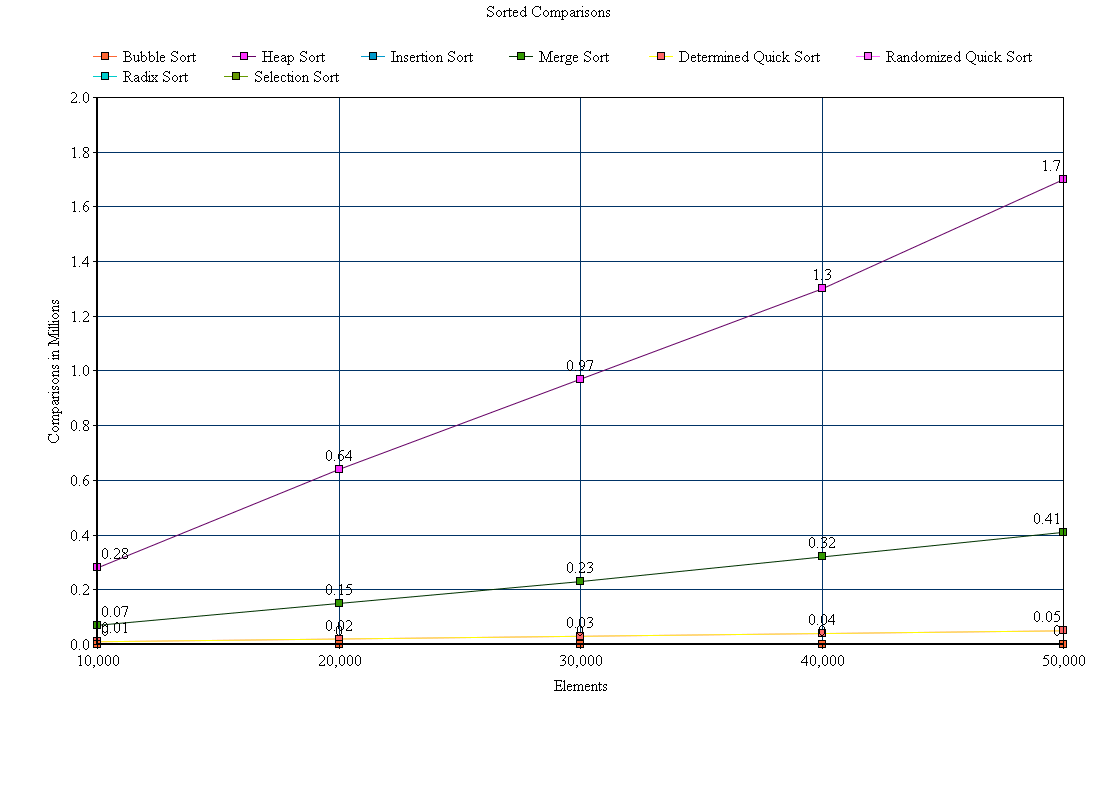


As anticipated, our experimental run times are very similar to those of our theoretical run times. When we ran the random, unsorted sequence of data, the bubble took by far the longest time to execute. The insertion and selection sorts ran slightly faster, followed by each of the remaining sorts that ran too quickly to be shown discretely on the plot (heap, merge, both quick, and radix). This proved that when sorting a large-scale, random list of numbers, it is best to look at the big picture rather than at each individual number and its immediate neighbors. The already sorted sequence of data proved that the least efficient was the heap, followed by radix, selection, both quick, and insertion. Due to the simplicity of the bubble sort’s algorithm, it actually ran through this sorted list on O(n) time, making it the most efficient to run on an already sorted sequence of data. This is primarily because it only compares each element to its direct neighbors, and when they are already sorted, it only scans over each of them once. We theoretically discussed the best-case running times for each of these sorts, and we extrapolated data very similar to the results we expected. Finally, we analyzed the results for the reversed sequence of data. As anticipated, the bubble took the longest, followed by insertion and selection. Since these took so long to complete (around 3 minutes in the worst case), each of the remaining sorts all ran too quickly to be discretely shown on the plots. The reason radix an heap took so long on the sorted, but ran so quickly on the reversed and unsorted, is because they are designed to run in a relatively constant time for any sort of sequence. These operate by looking at the whole sequence in a hierarchical fashion, rather than a linear, directly comparative approach. To add, we proved that the bubble sort is the least efficient when sorting anything but an already sorted list. Overall, we got to see each of these sorts run in their best (already sorted), worst (reversed), and average (random sequence) run times, and the results we anticipated in our theoretical analyses matched up to the results almost exactly.

Next, we extrapolated the average number of comparisons in each of the cases, for each of the respective sorting algorithms. The results are graphed below, with respect to the millions of comparisons and the number of elements.







As the results above show, the number of comparisons that each sort completes doesn’t necessarily effect the run time of the sort. In the unsorted sequence of data, the radix and merge sorts took on over twice as many comparisons as did the other algorithms. However, the bubble still had a longer running time, despite almost half of the comparisons. Since these three sorts had so many comparisons in the unsorted sequence of data, the 5 remaining used so few comparisons that they weren’t discretely shown on the graph. In the sorted array of data, the heap sort actually had the most comparisons, due to its hierarchical system of sorting. Then came the merge, and then each of the others with too few comparisons to be discretely shown on the plot. Again, this proves that the number of comparisons doesn’t necessarily affect the run time. Other than the heap that ran in the longest time, each of the remaining sorts actually ran in an order completely opposite to that of the number of comparisons. Finally, the bubble and insertion had exponentially the most comparisons when sorting the reversed sequence, followed by all of the others that used less than 1% of the number of bubble’s comparisons. This data once again illustrates the lack of correlation between the run times and the comparisons, making it safe to assume that the run time is just an indicator of the strength of the algorithm. A sorting algorithm can use virtually no comparisons while taking a very long time to run, and just as much the opposite. Since number of comparisons don’t indicate the run time, it is important to analyze the strength of the algorithm and measure the run time when judging efficiencies of sorting algorithms.