Comparison of Efficiency and Speed of Different Sorting Algorithms and their Methods

Introduction

In this scientific writeup, we will be observing and comparing the performance of 4 popular sorting algorithms, used in everyday computing and computer science projects. These sorting algorithms are: Insertion Sort, Shell Sort, Heap Sort, and Quick Sort. Each is unique and preferred for its own characteristics, however we want to see how these sorting algorithms compare with each other in terms of items moved/items compared, based on parameters we specify such as the size of the list to be sorted, the seed for the randomness of these lists, etc.

The way a sorting algorithm is judged is based on its calculated efficiency. This is labeled as Big O, or O(), or in other words, the time complexity/space complexity of the sort. This characteristic measures how long the sorting algorithm takes the more items you sort, and can also measure the amount of space needed in memory for the sort when adding more items. Ideally you would want a sorting algorithm that can sort an unsorted list in O(1), meaning no matter how long your list is, it will sort it in the same amount of time. However this is not possible, so we try to go for the next best thing. Here are the O() functions from best to worse:

$O(1)$
$O(log(n))$
$O(n)$
$O(nlog(n))$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$

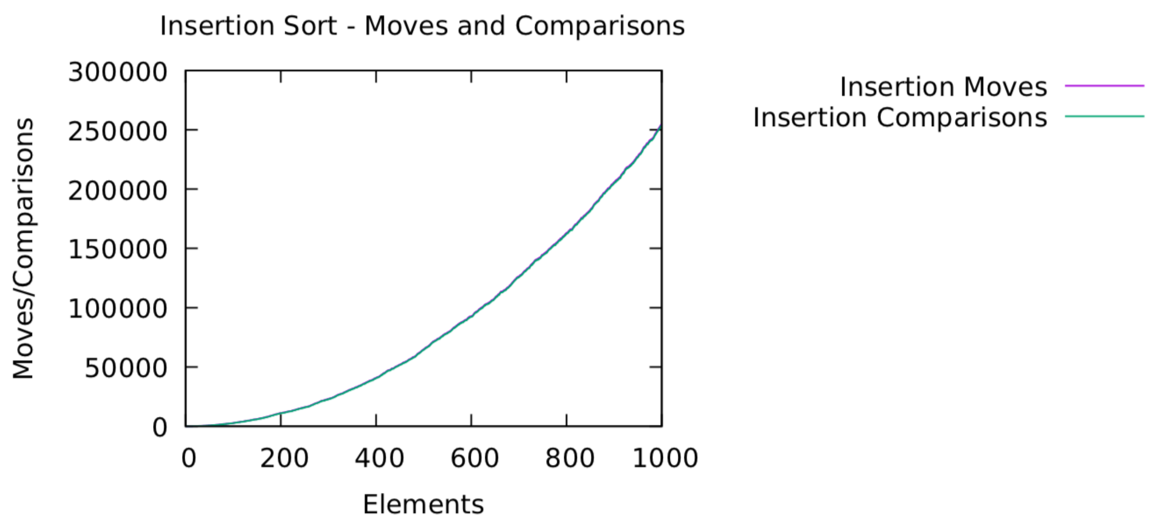
Materials and Methods

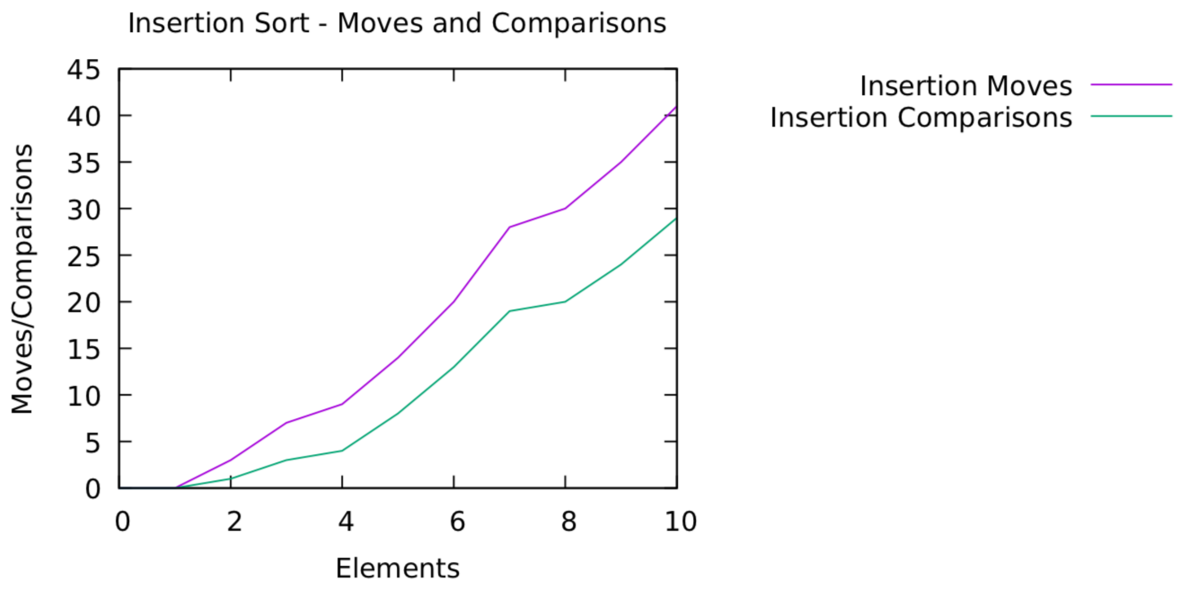
For this study, we were given pseudocode as a means of implementing our sorting algorithms in C. We were also given a Stats structure, that was used to facilitate the gathering of statistics about our sorts without using local variables. Lastly, a test harness was created as a way to input which sort(s) you wanted to view, the amount of elements you wished to be in the list to be sorted, and the seed to generate the pseudorandom numbers to be imputed into our list to be sorted.

Results

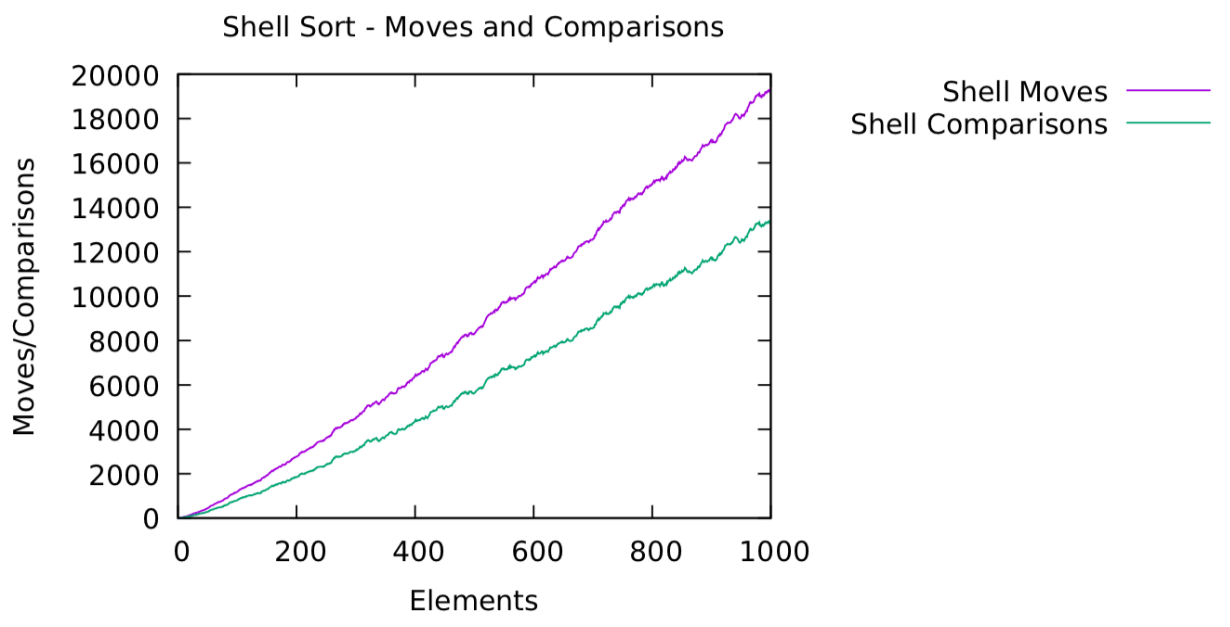
The results are displayed in the order: Insertion Sort, Shell Sort, Heap Sort, Quick Sort.

Insertion Sort

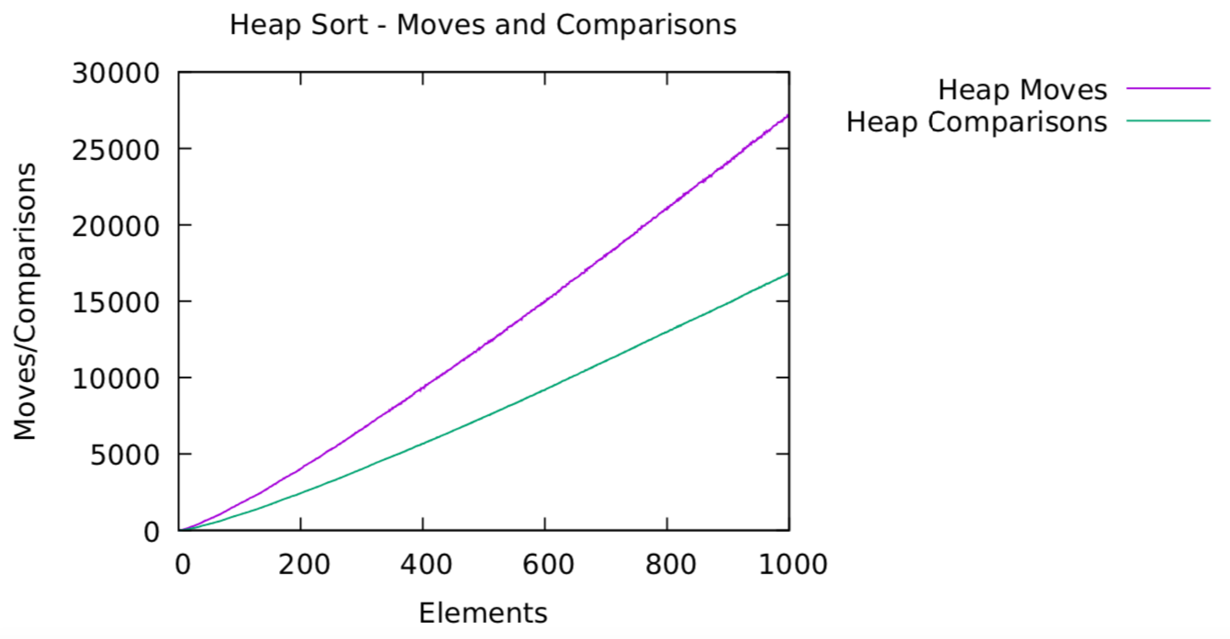




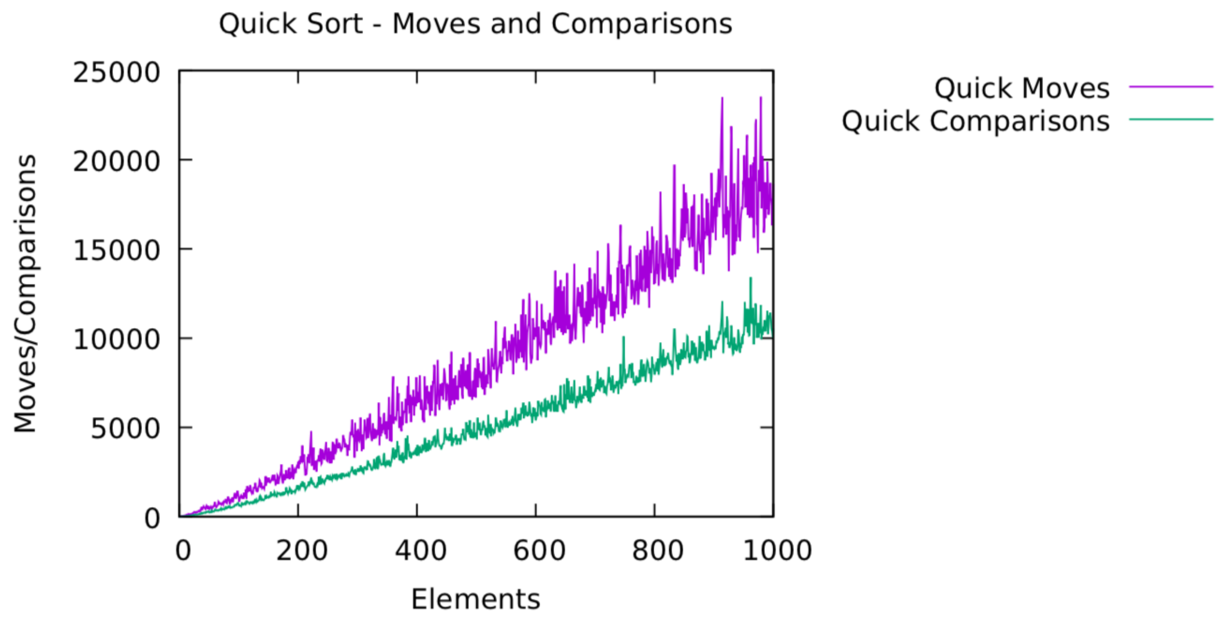
Shell Sort



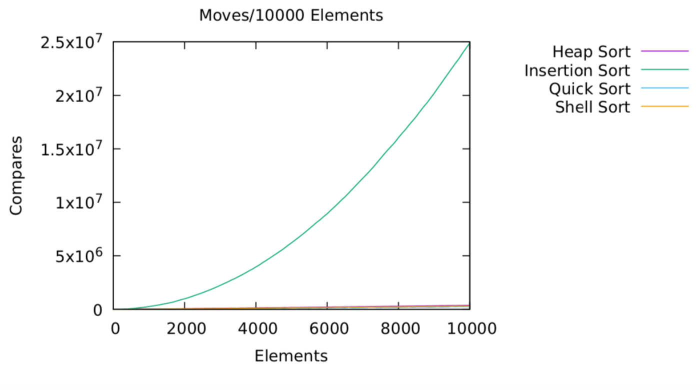
Heap Sort

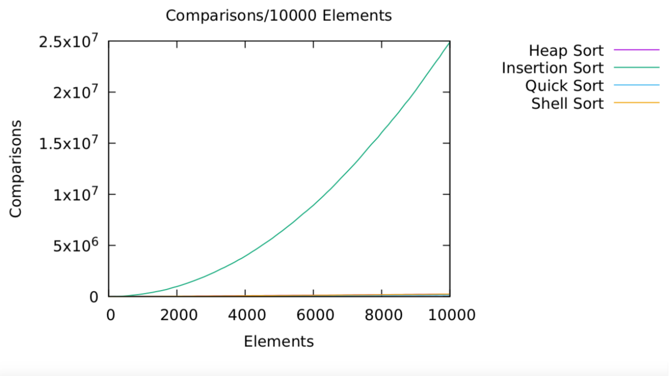


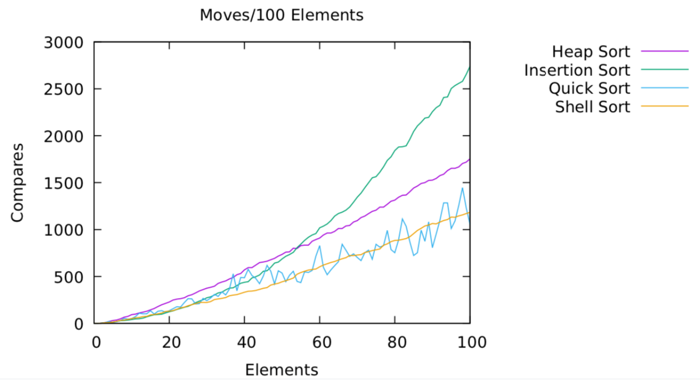
Quick Sort

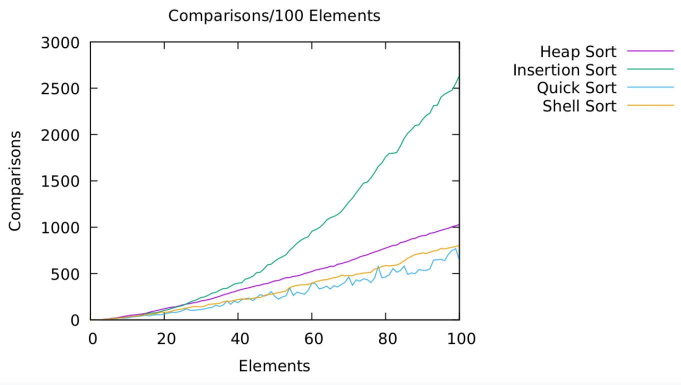


All Sorts, 10000 Element range vs 100 Element range









Analysis

It is always interesting to observe how different sorting algorithms perform compared to one another. With this data, it is easy to observe which algorithm is best for the task at hand. Let us take a look first at our two pairs of last graphs. Both pairs display the amount of compares and the amount of steps each sort does over a range of list lengths. The first pair is over a range from 0 to 10 thousand list lengths, and the second pair is over a range from 0 to 100 list lengths. We can already tell that insertion sort, arguably the easiest to implement, performs the worst overall for large amounts of data. The Big O for insertion sort is $O(n)$ in the best case and $O(n^2)$ in the worst case. On a scale of 10 thousand elements sorted, other than insertion sort, our other sorts perform similarly. To compare them clearly, we must look at our second pair of graphs, since these observe the sorts over a smaller range of length of sorted elements. For lists with the number of elements under 40, all sorts perform the same way, having comparison and move counts either at 500 or below. After this we see insertion sort take off and peel away from the general trend of the other algorithms. Heap sort also seems to be a bit less efficient, but not by much. However what is interesting is Quick sort and Heap sort, both performing around the same (also accounting for quicksort's fluctuation/variability).

Now we will look at the individual graphs of the sorts.

With insertion sort, both the moves and the compares follow the same trend closely. So close that a second graph showing a range from 0 to 100 was needed to see how close the curves follow each other. The larger range graph seems to depict an x^2 function. It is still the worst performing sort (in worst case with a Big o of $O(n^2)$, and on average is $O(n^2)$, though at least the number of moves and the number of comparisons are somewhat predictable if we know one of the values. It is as if the curve for the number of moves was the curve for the number of compares, just shifted up 10 units.

Shell instead, is a generalized version of insertion sort, and uses much fewer swaps than the later does. Thus, we can expect it to perform much better than our first counterpart. Shell sort has a best case big O of $O(nlogn)$ worst case $O((nlogn)^2)$, and an average of $O((nlogn)^2)$. Over a range of 1000 sorted elements, it's comparisons and moves curves are much more linear, show more separation compared to insertion sort (at 1000 list length the difference is about 10 thousand units), but they do show a bit more fluctuation.

Heap Sort is most likely the most technical to implement and understand. It has an average Big O of $O(nlogn)$, which seems to remain constant for either best or worst case sorting. It displays similar graph characteristics as shell sort, however it lacks the variability/slight fluctuation that shell sort's comparisons and moves have.

Our last sort is quick sort, which is a sort that uses recursion. With an average big O of $O(nlogn)$, but at worst case of $O(n^2)$, it has the same time complexity that heap sort has. The graphs show the most variation seen with any sorting algorithm's moves and compares.

Overall, there is no way to choose which sorting algorithm is best overall. It all depends on a case by case situation. If you have a list of 100 unsorted elements, quicksort and shell sort perform the best compared to the other 2 while insertion sort is the worst, however with lists of elements 10 or smaller, insertion sort's simplicity makes it marginally the best one. It is also completely possible to merge different sorting algorithms together to get the best time complexity. One example could be using quick sort for 90 percent of the sort, and once the list is almost finished insertion sort comes in and finishes the last few moves and compares, which could shave off a few microseconds from the sorting time. In the end, there are many more sorts to choose from than these 4, and only the situation for its use will be the last decider on which will perform the best.