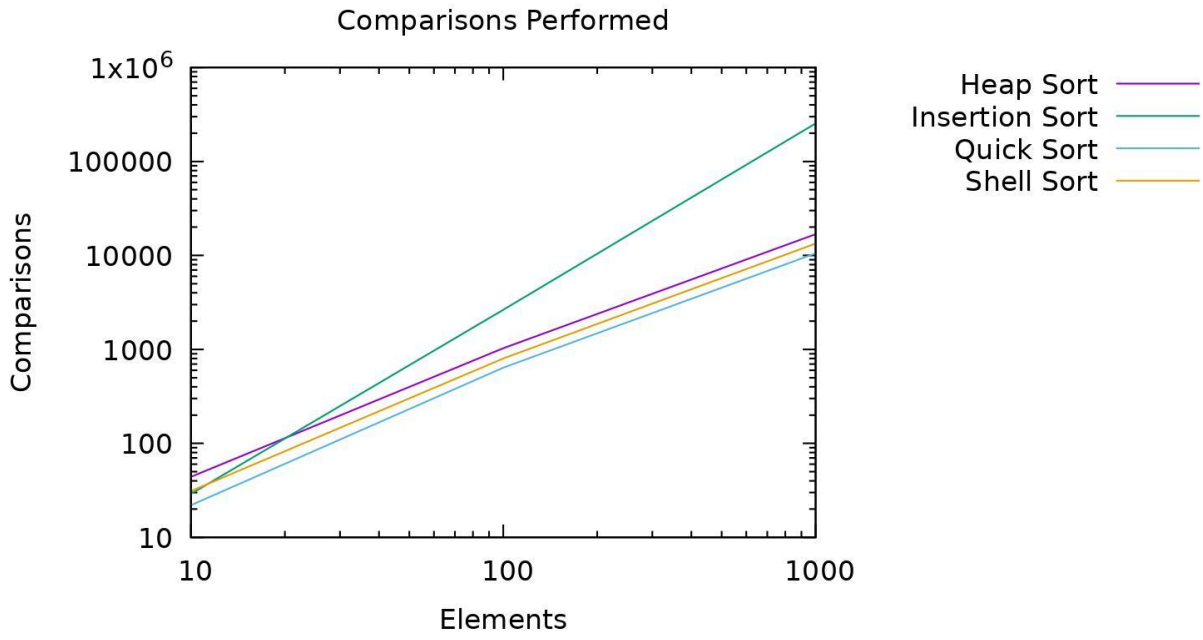
Edison Chen

Assignment 3 - Comparing Sorts: Insert, Heap, Quick, and Shell

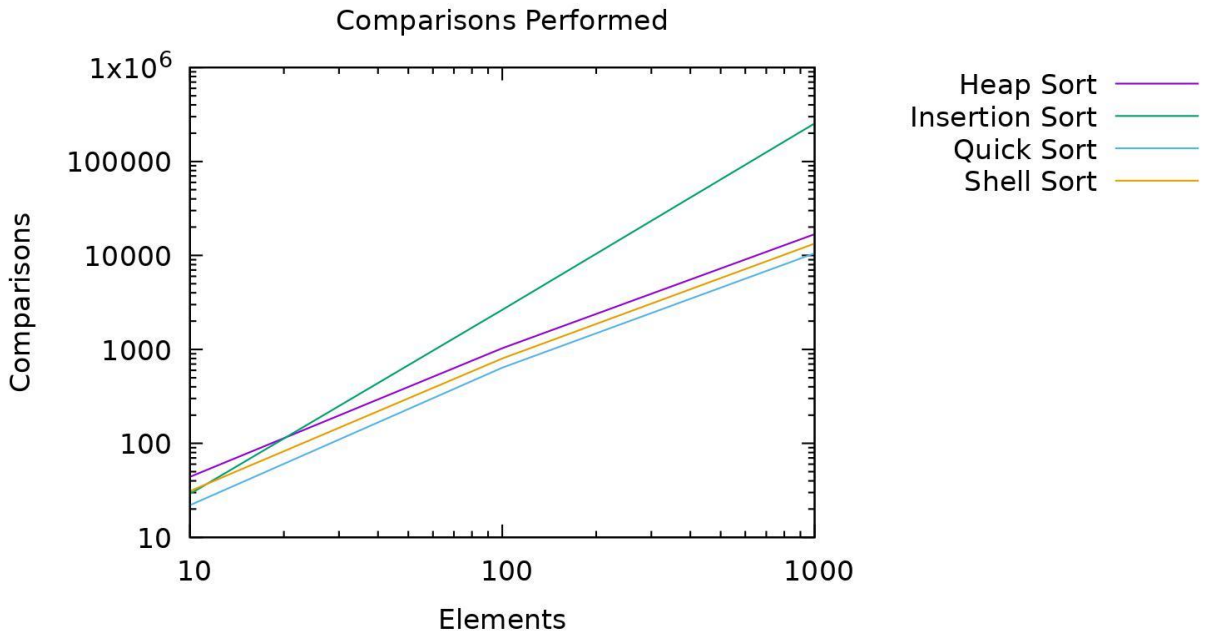
In this assignment, students are required to implement four sorting algorithms: insertion sort, shell sort, heap sort, and quick sort. Statistics such as the number of elements, the number of moves, and the number of comparisons will be implemented and tracked for each sort. This write up will compare the differences between each of the four sorting algorithms by graphing the number of moves and comparisons needed to sort an array of n elements. The performances of the sorting algorithms will be graphed and compared at 100 elements and 1000 elements. The following graphs included in this write up are logarithmically scaled to clearly show and compare the sorts.



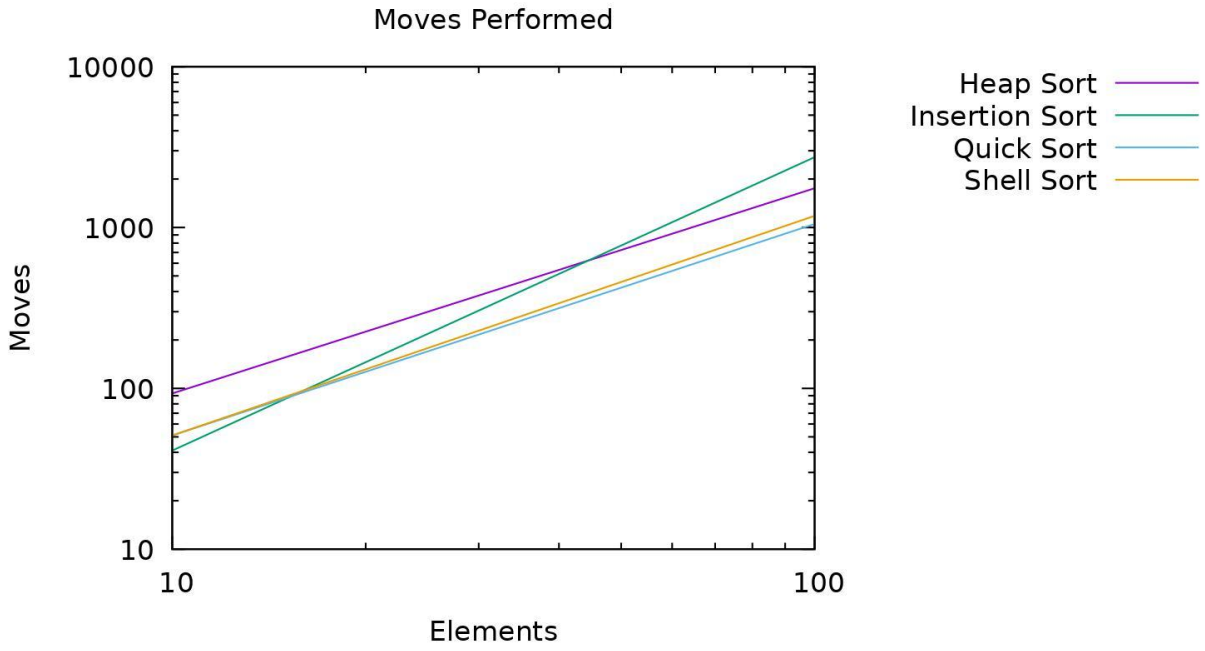
Looking at the graph, it is evident that the performance of heap sort, quick sort, and shell sort are consistently “stable” across 100 elements. What is interesting to see is that the number of comparisons insertion sort makes is initially less than that of shell sort and heap sort at 10

elements. However, as the number of elements increases, so does the number of comparisons insertion sort takes to sort the array, at a much larger rate of increase than the other sorts. This is due to the fact that insertion sort compares from element to element in an array. This means that the next element to be sorted, if smaller than all of the preceding sorted elements before it, would have to be inefficiently compared to all those already sorted elements until it is in the first position. So as the number of elements to be sorted increases it would make sense for insertion sort to become more and more inefficient in terms of comparisons.

The same trend can be seen with 1000 elements.

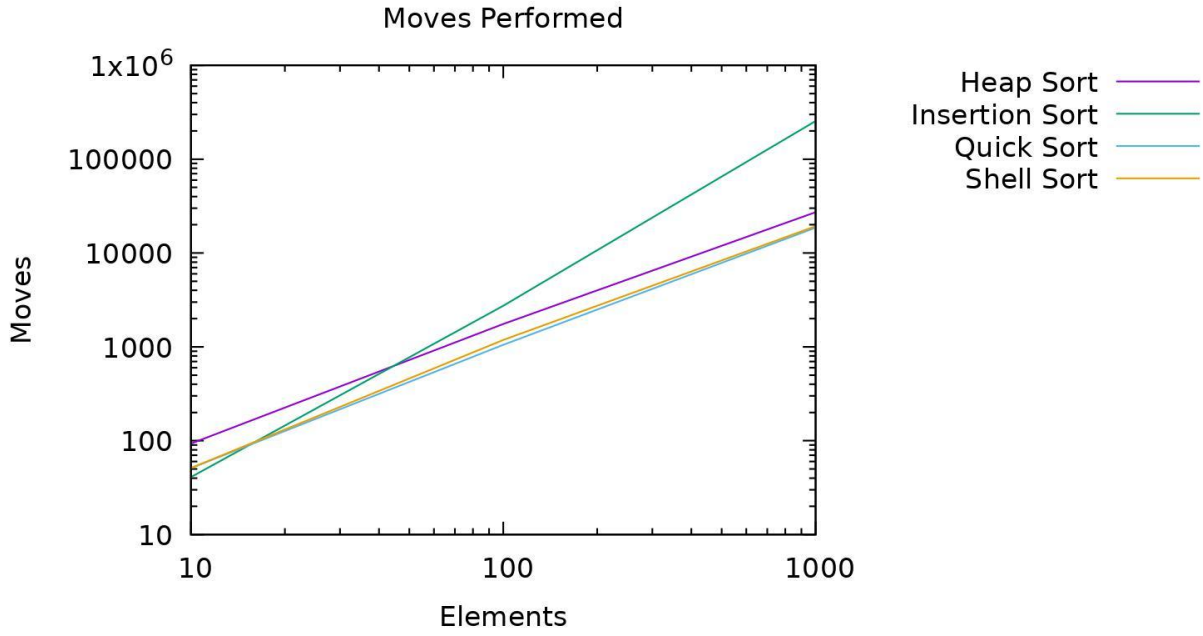


What about the number of moves with 100 elements?



Insertion sort requires the least amount of moves compared to all the other sorts when sorting arrays up to 15 elements, until it begins losing to quick and shell sort. Insertion sort continues to require more moves at a much faster rate than that of the other sorts as it passes heap sort at around 45 elements. This is likely due to the nature of insertion sort, similar to the case of comparisons. Though highly efficient with smaller elements, the method of insertion sort gradually becomes ineffective as more and more of the same comparisons and moves need to be made as smaller elements at the end of the array are compared/moved from end to end.

What happens to moves with a 1000 elements?



Interestingly, much of the same trend can be seen as in the graph of moves with 100 elements. However, there is a slight difference in the number of moves required by quick sort and shell sort seen at the middle of the graph. A noticeable gap can be found between the number of moves required for quick sort and shell sort that starts at around 20 elements to 300 elements before overlapping. This gap can likely be explained by the best case time complexity of quick sort and shell sort, which is O(n*log(n)) for both. That explains the close similarities of the number of moves and comparisons between quick and shell sort. Though they are close, it is clear to see from the graphs of both moves and comparisons that shell sort has always performed better than quick sort. This can likely be explained by their worst case time complexities. Quick sort has a worst case time complexity of O(n^2). Shell sort has a worst case time complexity of O(n*log^2n). This can explain the differences in performances and the gap, in which quick sort likely had bad cases.