# Analysis – Array Sorting Algorithms

Brian Nguyen

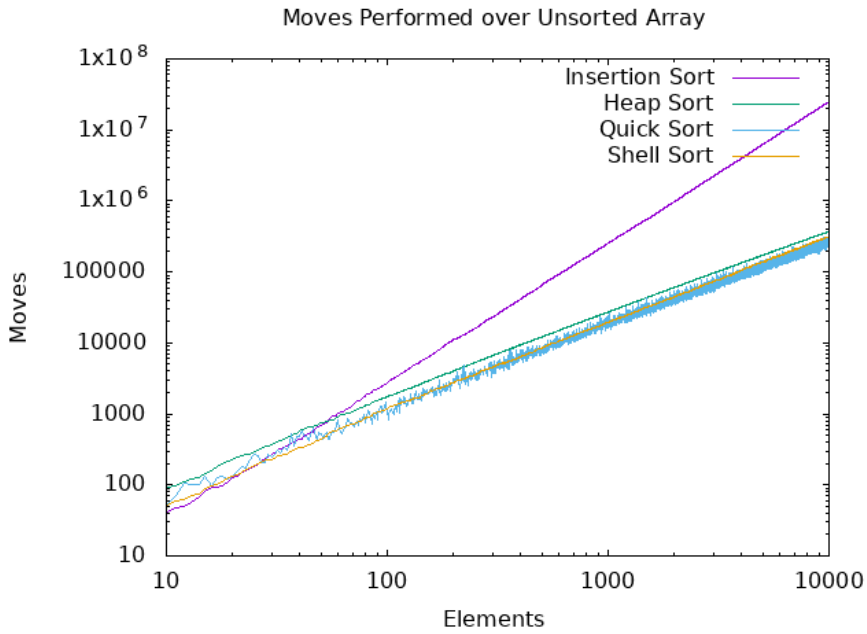
October 2021

## 1 Introduction

In this sorting library, I created 4 sorting functions (Insertion, Heap, Shell, and Quick Sort). The 4 sorting methods were implemented in C to sort a psuedorandomly generated Array of 30-bit unsigned integer elements from least to greatest. Because all these sorting methods use different algorithms to sort an array from iterative to partitioning, they will perform differently and vary with their moves and comparisons they have to make to sort the same array. In this write-up, I will analyze these differences in each of the sort's performances to see which sort is slowest or fastest at sorting a random n-element array via data supplied in the graphs.

## 2 Analysis of Differences

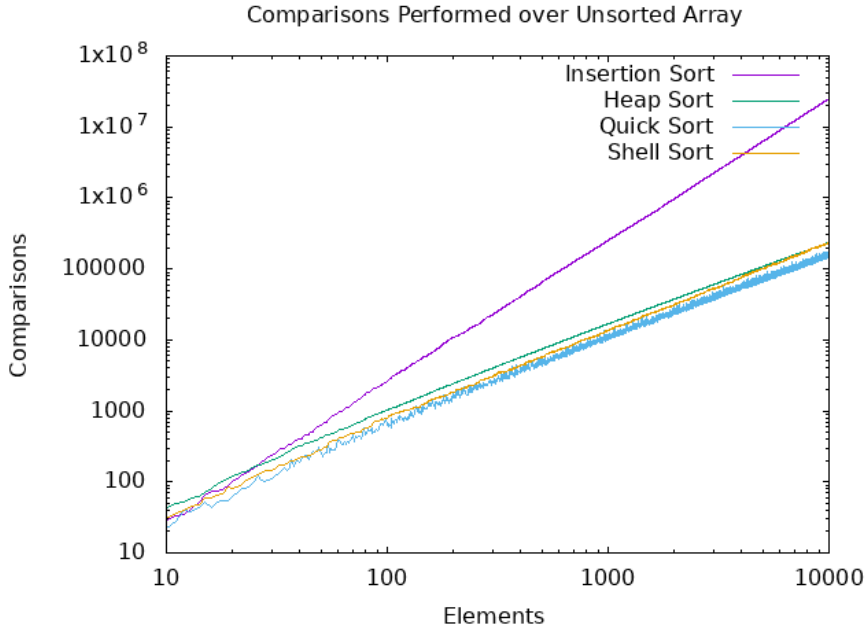
### 2.1 Moves Graph




This graph shows the number of moves each sort has to make to sort the same n-element ar-

ray depicted in the y and x axis respectively. All functions have a steady increase in moves as elements goes up obviously so as the elements needed to move are tied to size. Looking at the graph it seems that because of Insertion Sort's iterative nature it does slightly better than other sorts when it comes to low element arrays as other sorts have to section or divide up the array before doing anything to the list. However, once it gets to anything past around 100 or 200 elements, it is clear the other methods are better because of their logarithmic or exponential nature verses Insertion still having to check each element individually which adds up to being slower. All the other 3 sorts have take a similar amount of moves to sort the array because of their similar time complexity of n*log(n), but it seems like heap is consistently slower in moves than shell or quick. This is due to the amount of swaps Heap does based on its algorithm, having to do more than quick and shell would do which increases the amount of moves. It also seems that quick sort is generally faster as elements get larger although still being relatively the same as shell because it does not have to swap any elements but just move them while still maintaining the same n*log(n) time complexity. Quick is however being sporadic and sometimes slower than shell sort with its worst case of $n^2$ verses the others of n*log(n) due to the algorithm used in quick sort of partition arrays and using pivots which would affect its time based on the location of the pivot.

## 2.2 Comparisons Graph




This graph shows the number of comparisons each sort has to make to sort the same n-element array depicted in the y and x axis respectively. Again, there is the same trend of Insertion being the slowest with a big array size and the other 3 sorts being around the same with Heap sort taking slightly more comparisons than shell or quick sort. Quick sort seems to have less sporadic results for comparing elements which is most likely due to the pivot location not affecting the

number of comparisons as much with it being directly tied to the main cause of the sporadic behavior being the partition function and therefore tied to the pivot as well. Shell is also again very similar to quick as it is with moves. Overall, the comparisons graph depicts the same trends and results the moves graph does besides the differences listed.

# 3 Conclusion

Based on the analysis of the graphs from the implemented sorting algorithms, it seems expected that the results closely correlate to the time complexity each sort has which explains their performances and differences between how fast and slow they are based on moves and comparisons. It seems that Insertion sort is the only very slow sort when it comes to big sized arrays, while all the other 3 have very similar performance with the quickest average being quick or shell sort depending on the array. These graphs have provided a great visual on the big O or time complexity in action between these sorts.