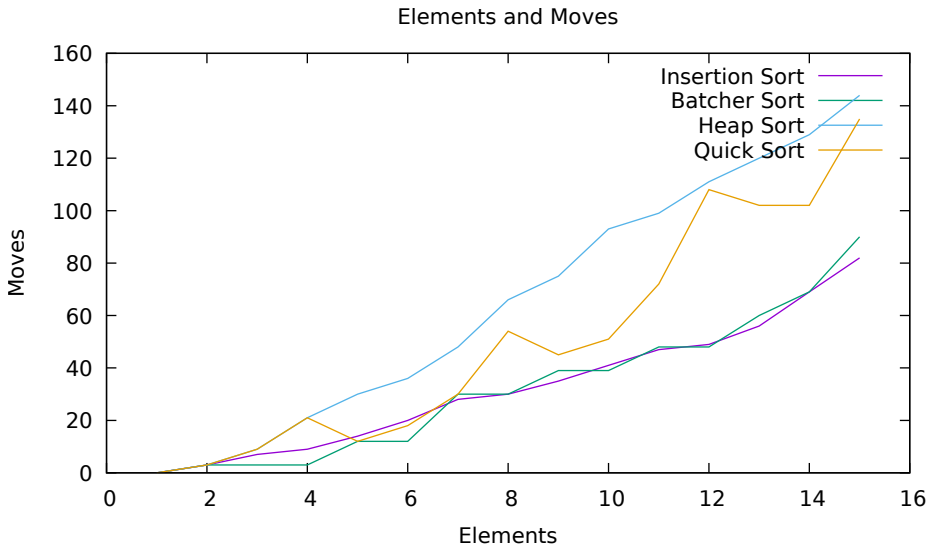
Assignment 3 Write Up

Nicholas Chu

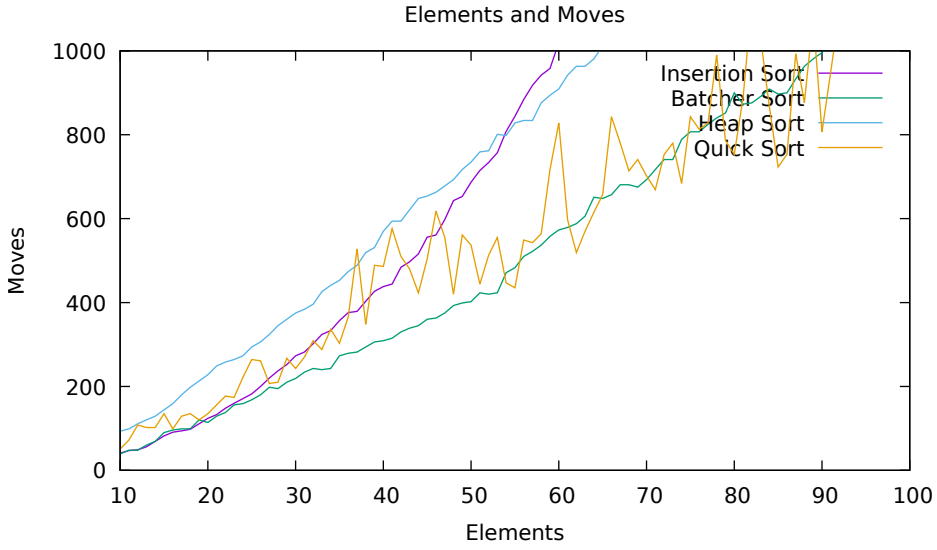
January 26, 2022

1 Conclusion

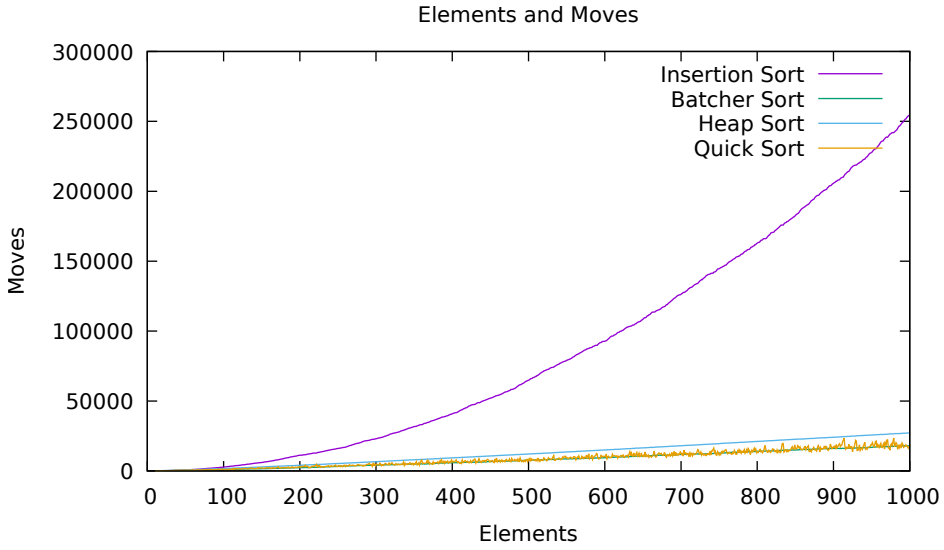
Working through this assignment has introduced me to some common sorting algorithms that can be used to sort data in arrays. The sorts that were introduced were Insertion Sort, Quick Sort, Heap Sort, and Batcher's Odd-Even Merge Sort. The sorts were implemented into functions in separate files and accessed in the main program sorting.c which would use these sorting algorithms to sort pseudo-randomly generated arrays. These sorting functions would also track the amounts of compares and moves of elements within the array when sorting. Below are plots created by comparing how many times elements in an array were moved and adjusted during the sorting process over an array containing a specific number of elements.



We can observe through this graph that from sorting arrays ranging from 1 to 15 elements long that the two most sorts in this range that took the least amount of moves were Insertion Sort and Batcher Sort, with the least amount of moves. Heap Sort and Quick Sort both generally take more moves than the other two, and Heap Sort taking the greatest number of moves to sort the smaller sized arrays.



When zooming out the graph to show the range of 1 to 100, we begin to see changes in the relationship of the number of moves needed to sort an array of a given length. We can observe that as the number of elements in the arrays begin to increase, the number of moves that the Insertion Sorts needs to make becomes much larger compared to the other sorts.



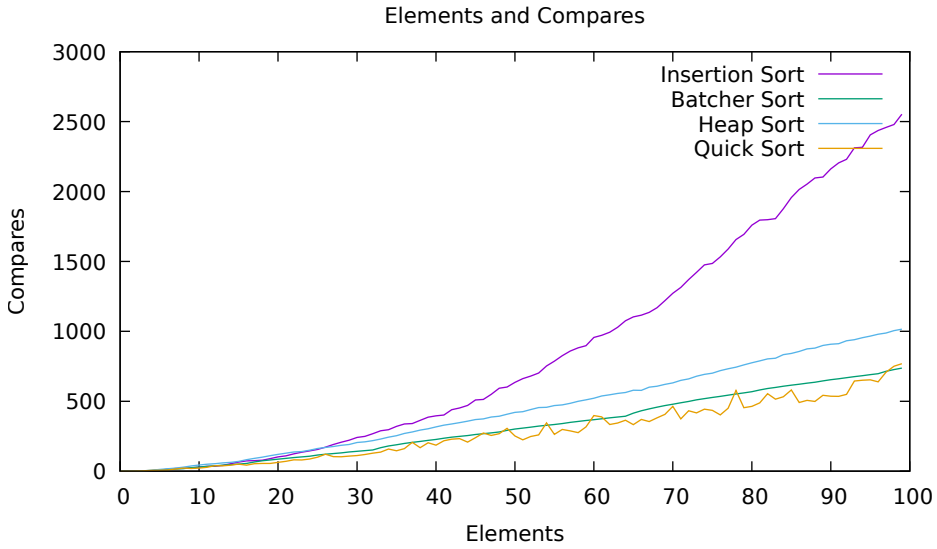
By plotting the sorting algorithms and having them sort arrays with up to 1000 elements, we can get a better perspective on the relationship between the number of compares and array size. Some of the notable features of this graphs that we can observe are:

- Insertion sort has the largest and most notable slope, which we can fairly assume is exponential. This reflects its complexity of $O(n^2)$ and that the number of compares needed to sort arrays of growing sizes will grow exponentially. This is because Insertion sort iterates over

every element in the array twice, for each element moving it across the elements in the array until it can be inserted into the appropriate position. This would naturally get out of hand as when the arrays get larger the number of moves needed to sort sees a sharp increase, showing that this sorting algorithm is not a very efficient way to sort large arrays.

- Quick Sort has a very interesting jagged shaped slope which can be attributed to the way the algorithm is structured. Quick Sort works by dividing the array by a pivot into smaller arrays, sorting sorting those, and recombining to get a sorted array. The jagged structure can most likely be attributed to the way the pivots are found, as the sort needs to swap around elements in the array to varying degrees, like if the pivot is the largest or smallest element of the array, before it can find an appropriate pivot to split the array at.

The relationships on the graph help reflect this, as naturally these algorithms will have different levels of efficiency when sorting different sized arrays, and thus cause different complexities and run times. This is also reflected when comparing elements with the number of compares done throughout a sort.



We see that the relationship between number of compares relative to array size also follows the same trend like with the number of moves. The slopes are very similar for each algorithm and reflect their complexity. As mentioned above, Insertion Sort has a complexity of $O(n^2)$ because it needs to go over all arrays twice. This is reflected in the moves and compares slopes which are both exponential and show it is a bad sorting algorithm to use then sorting large arrays due to the large amount of compares and moves it has to do. Heap Sort has a complexity of $O(nlog(n))$ as it first needs to organize the array into the heap and take all the elements from the heap in order and convert it to a sorted array. The same can be said with Quick Sort since instead of iterating over a heap, it does so over split arrays. This is reflected in their slopes as they all look very similar to a nlog(n) graph, which looks similar to a linear slope with there existing a very small increase as the elements increase. This shows that both the necessary amount of moves and compares both these functions need to sort scale fairly well as elements increase and in are better options to choose when sorting large arrays. Batcher's Odd-Even Merge Sort has a complexity of $O(nlog(n)^2)$ because it

sorts all the elements by comparing sets over multiple gaps and swapping them when appropriate. In the graph we can observe that its slope is similar to Heap Sort's and Quick Sort's except that Batcher Sort's slope has a slightly higher increase, reflecting the additional log(n).

2 Takeaways

One of the biggest takeaways from this assignment is the importance of choosing the appropriate algorithm when writing a program. From just testing and plotting the results of each algorithm it became apparent how much slower Insertion Sort was, especially when dealing with larger arrays, when compared to the other sorts. While the algorithms all produced the same results, the other sorts would have been finished in a fraction of the time that insertion would take. This could have great effects on the usability and efficiency of the programs they are used in. However, this does not completely write-off the viability of algorithms like Insertion Sort, as when looking at the number of compares and moves it made when dealing with small arrays like of lengths around 1 through 20, it was either just as or more efficient than the other algorithms. This shows the importance of first addressing the task and choosing the most effective algorithm for said task, as one algorithm does not fit all. Other takeaways from this assignment were learning about sets and how they can be creatively used like in the case of storing command line arguments, as well as masking and memory allocation for when generating random arrays.