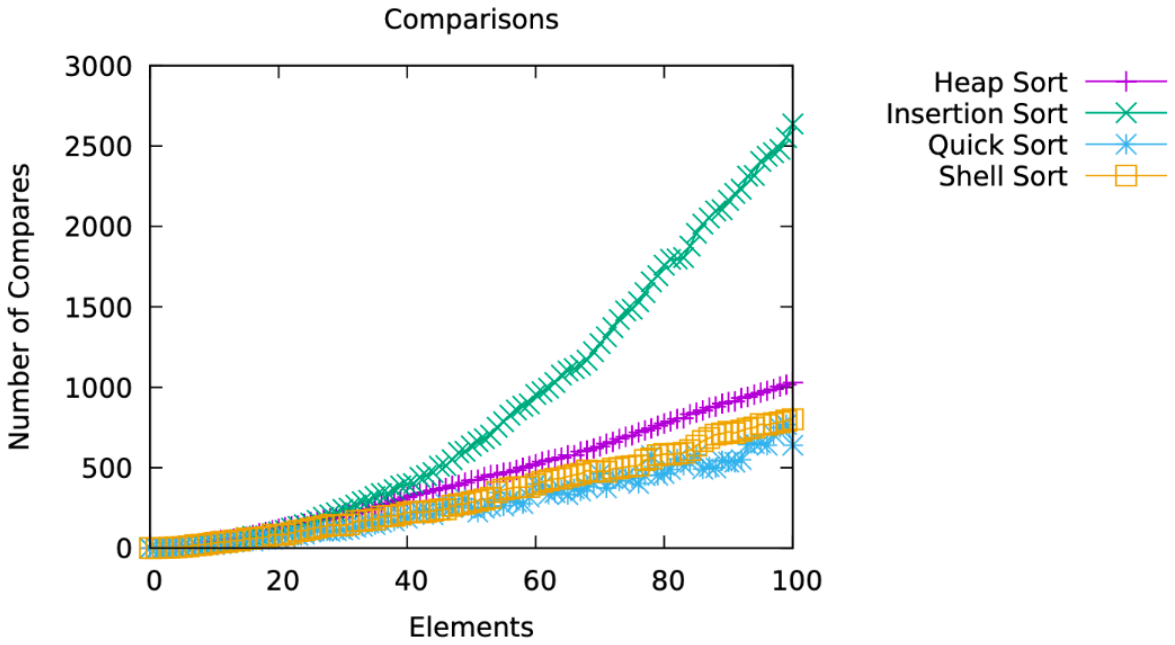
Diwa Ashwini Vittala
CSE 13S - Fall 2021

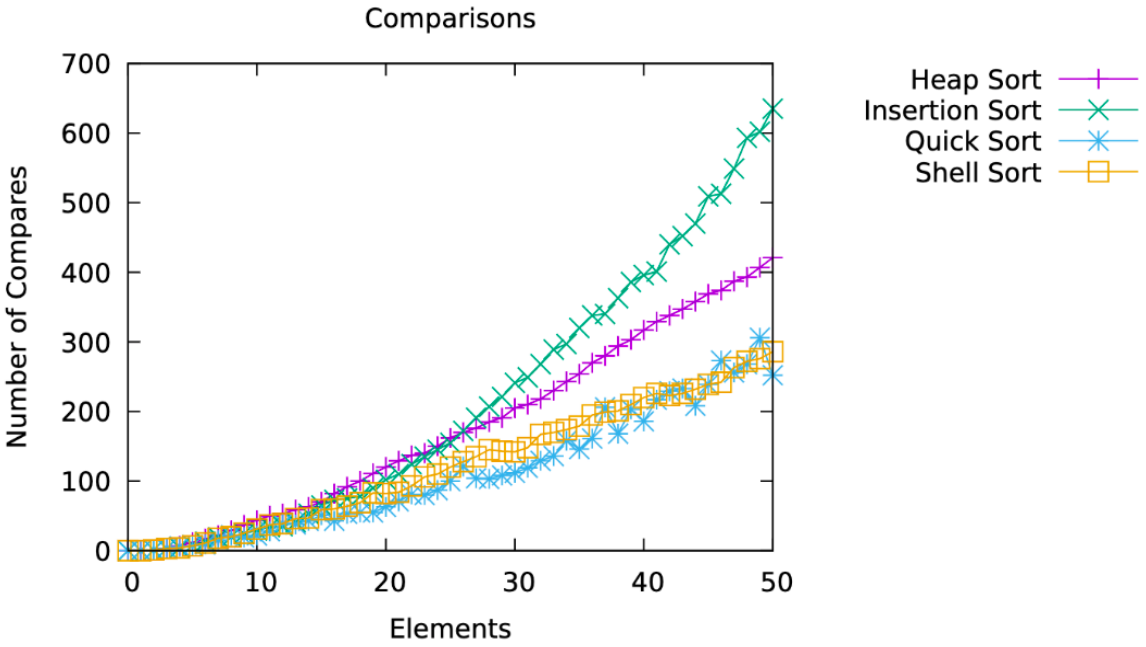
⁰❀ Assignment 3: Writeup ❀。

SORTING: PUTTING YOUR AFFAIRS IN ORDER

Compares

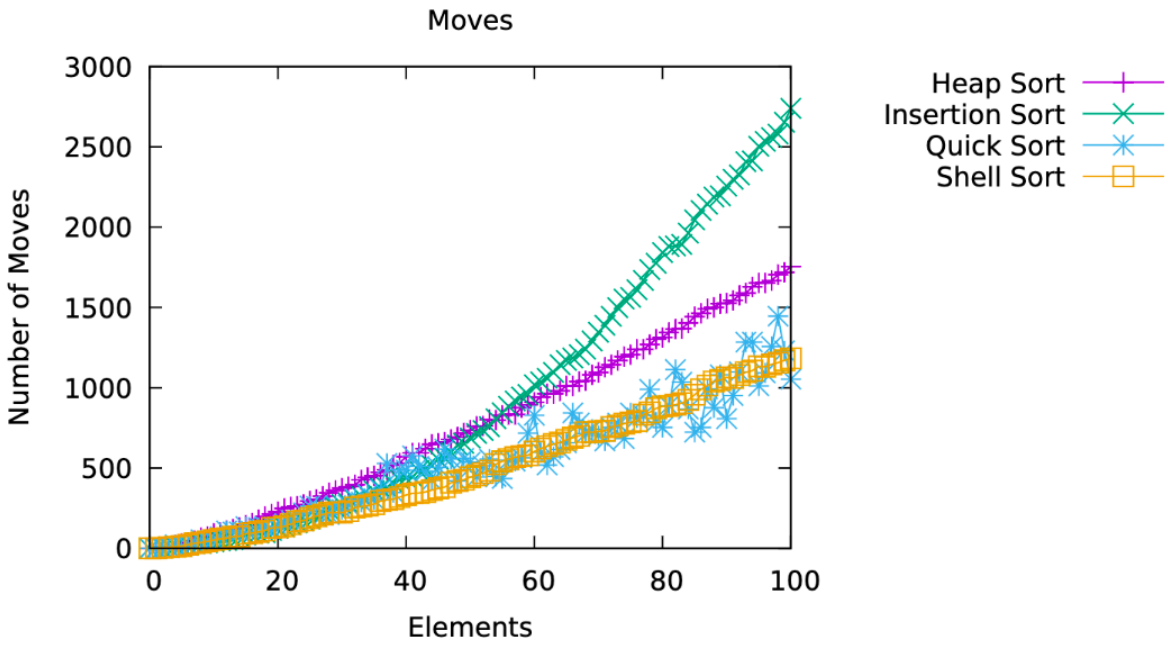


The graph above shows the number of comparisons made by my Heap Sort, Insertion Sort, Quick Sort, and Shell Sort functions over 0 to 100 elements. From the graph, we can see that Insertion Sort takes the most number of comparisons to sort through the array when there are more than 20 elements involved. On the other hand, Quicksort was able to sort through the elements the fastest, even when the number of elements were large. Since Quick Sort is an order nlog(n) function on average (O(nlogn)), while Insertion sort is an order n^2 function (O(n^2)), it makes sense that Insertion sort takes longer to move through the entire array. Quick sort in general takes less comparisons than all the other functions, perhaps because it efficiently uses recursion to split the array to smaller and smaller parts. Heap Sort and Shell Sort, perhaps due to the way they both “restructure” the array in order to sort through them (heap sort structures the array into a heap and sorts through comparing children, shell sort compares two terms in the array using a gap), are both consistent and almost linear in the number of comparisons performed over number of elements.

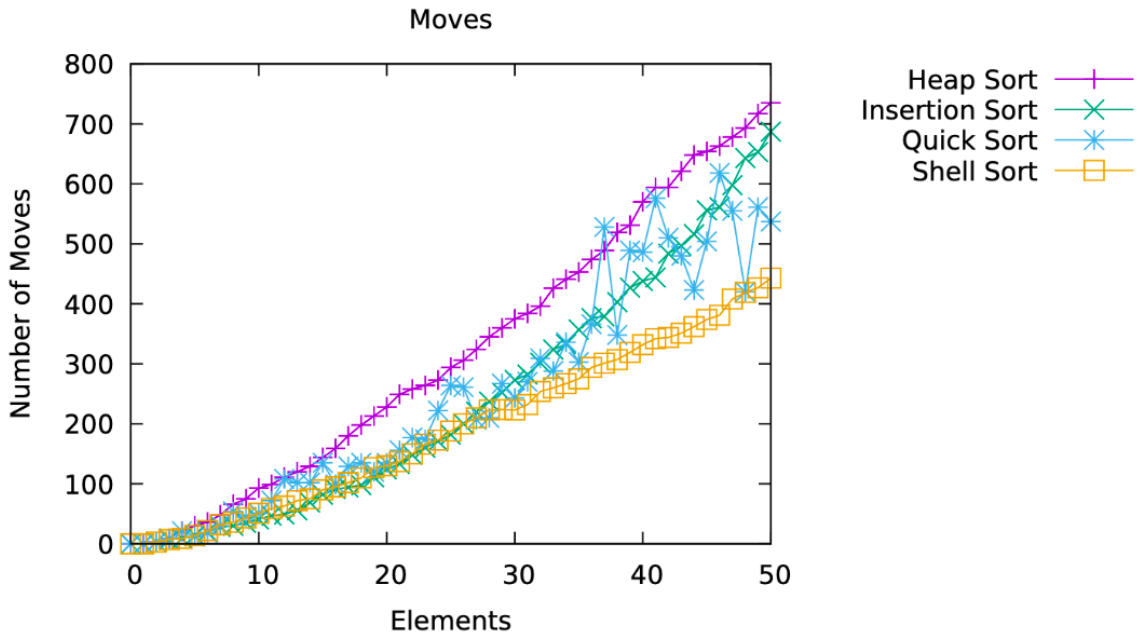


Adding on, when we zoom into the functions as they make comparisons over 0 to 50 elements, we see that all the functions perform relatively the same number of comparisons until about 15 elements. From here, Heap Sort performs the most comparisons until we input 25 elements, but then Insertion Sort performs the most comparisons (and becomes progressively inefficient) as we input 25 or more elements. This means that it is best to use sorting functions such as Insertion Sort when the size of our array is relatively small with around 20-25 elements or less, while it is best to use functions such as Quick Sort or Shell Sort when we have more elements when we are trying to optimize the number of comparisons used.

Moves



The graph above shows the number of moves made by my Heap Sort, Insertion Sort, Quick Sort, and Shell Sort functions over 0 to 100 elements. Interestingly, the Quick Sort function is less linear than it was in the Comparisons chart. Since Quicksort is the only function that uses recursion to sort through elements, it may contribute to the reason why some array sizes use up more moves than others even though they are so close to each other. When looking at the graph of Insertion Sort in comparison to Heap Sort, we see that Heap Sort takes more moves to sort through elements than Insertion Sort until around 50 elements where Insertion Sort becomes the function that takes the most moves 50 elements and beyond. This tells us that we should use Insertion Sort when we want to sort through arrays with less than 50 elements while heap sort would be better to use for arrays with greater than 50 elements while optimizing the number of moves used.



Zooming into the functions as they perform over 0 to 50 elements gives us similar results but we can see more variation in the way that the sorts perform. Shell Sort is an order n^(5/3) function and Heap Sort is an order nlog(n) function, which may explain why Heap Sort takes the most moves than others when the number of elements in the array are smaller. It would be better to use Shell Sort if we are trying to optimize the number of moves performed with an array that has a smaller number of elements. The Quick Sort algorithm seems to output a wide range of moves, even exceeding Heap Sort at some points, which may be due to the fact that it can be an order n^2 function at its worst case scenario or an order nlog(n) function depending on the input.

What I learned about the different sorting algorithms:

As I mentioned before in the analysis, different sorting algorithms perform better or worse depending on the number of elements in the array. For example, if we were trying to optimize the number of moves and comparisons that we use for our sorting algorithm to be the smallest possible value, it would be more ideal to use our insertion sort algorithm for elements smaller than 50. Similarly, heap sort uses less moves and comparisons than insertion sort only when the array size is greater than 50. If we want to use a sorting algorithm that outputs a consistent (linear) number of moves and comparisons as we add more elements in the array, we should use shell sort rather than quick sort.