Assignment 5 Writeup:

<u>Overview/Experimentation</u>:

1. *Bubble Sort*:

For bubble sort, we learned that its average time complexity was also its worst time complexity : O(n^2). Although there is a constant, that does have some minor effect on the time complexity, it is ignored because of big O notation. This makes sense logically because when we think about it, in the grand scheme of things, a constant in a situation where the time complexity is described as O(n^2), will not really have a significant enough effect as n increases infinitely, that it becomes anything greater than O(n^2). We can observe this time complexity in action when running the bubble sort algorithm in our code where as the number of elements within our array increases exponentially, the time taken by the algorithm to sort becomes increasingly longer— at some points, it would take so long that it seemed like the program had stopped running altogether.

2. *Heap Sort*:

For heap sort, the best time complexity is equivalent to its average time complexity as well as its worst time complexity-> O(nlog(n)). This seems like an extremely good thing in theory, if the best case, average case, and worst case all have the same time complexity, but in reality, heap sort is not as quick as one would initially expect. In heap sort— though the constant is ignored in big O notation like it is for bubble, it also does have some impact on how the sort performs, especially when compared to the most efficient of sorts: quick sort. When giving heap a variety of different arrays, as expected for the most part, it performs similarly to how quick sort performs. In fact, because of the fact that its time complexity is constant regardless of best case, average case, or worst case, I did not really notice any huge differences when comparing the run time for heap given a reverse array or a large array, etc.

3. *Shell Sort*:

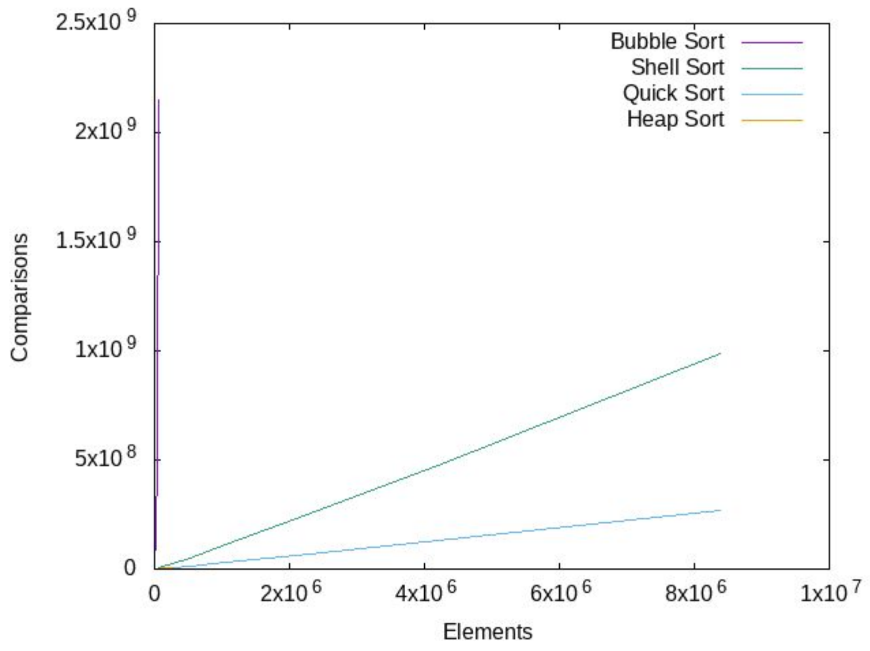
Like heap sort, the best case time complexity is also essentially its average case time complexity: O(nlog(n)). However, one of the main differences between the two sorts is the fact that the worst case time complexity is O(n^2). The main problem, in my point of view, with shell sort is that because the time complexity is completely dependent upon the gap sequence selected, the efficiency of this sorting algorithm is very unstable. This is because if by chance, the wrong gap sequence is selected, it will run with the worst case time complexity, and selecting the wrong sequence is not an occurrence that is very uncommon or unlikely to happen. The likelihood of this happening is actually much higher than desirable and because of this, I personally would not favor this sorting algorithm when choosing what sorting method to use, especially with a larger array. Giving shell arrays of different sizes and reverse arrays therefore, does not really affect its runtime into making it run on a worst case scenario. However, if the gap sequence is altered— for example, a large array is given but the gap sequence is {5, 3, 1} — immediately, the time taken by shell to sort the array is much longer, comparable to the time taken by bubble sort to sort the same array.

4. *Quick Sort*:

Quick sort, most often referred to as possible the best and most efficient sorting algorithm one would ever come across in their careers in the field of computer science/engineering, has a best and average case time complexity of O(nlog(n)). It, unfortunately, has a worst case time complexity of O(n^2), the same as the worst case for shell and bubble sort. Once again, the constant in quick sort, which comes about due to where a given array is partitioned, is ignored along with lower degree terms at the end because of how big O notation represents time complexity. Yet still, in general, quick sort is the best sorting algorithm and fastest sorting algorithm in the book. This is mostly due to the fact that although the worst case time complexity is pretty slow being O(n^2), this worst case will not come up very often— in fact, it can actually so unlikely to come up that quick sort is usually considered not to be doomed by it. While quick sort's average time complexity of O(nlog(n)) is the same as heap's average time complexity, the way in which quick sort sorts— dividing in array into two relatively equal halves and sorting each half simultaneously— makes the sorts happen extremely fast, no matter how big the array given to it gets. When experimenting with how different input arrays affect quick, we can observe quite easily why quick sort is the most efficient sorting algorithm since when running this algorithm, there really were no instances in which the sort slowed down for a specific array size or type of array(reverse/regular).

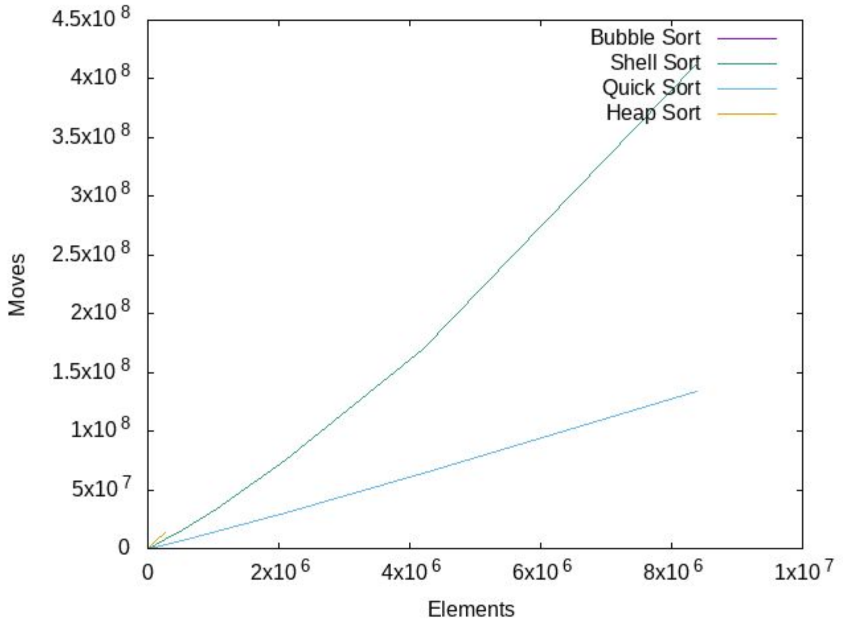
Graphs:

-Comparisons:



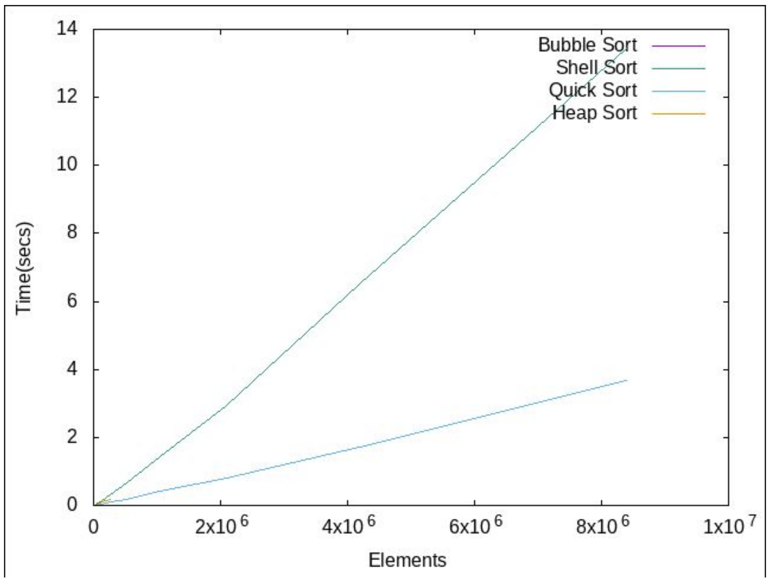
This graph basically depicts and compares the number of comparisons each sorting algorithm made for arrays of different lengths. Due to the fact that the number of elements in the arrays given to bubble sort stops at a smaller number, the number of comparisons it makes are also on the smaller end and thus are shown to be very close to the y-axis. On the other hand, because of the limited memory allocation of the virtual machine, the heap graph only draws the number comparisons up until a certain point before segmentation fault. However, we can see when comparing the graphs for quick and shell that as expected, because of the fact that shell and quick have the same average case time complexity of O(nlog(n)), that the graphs for the two are comparable. We can make out very clearly though, that though similar, quick is a lot more efficient that shell because even as the number of elements gets increased, the number of comparisons made by quick to get the elements into a sorted order increases at a lower rate than shell.

-Moves:



This graph shows and compares the number of moves that each sorting algorithm makes for arrays of increasing lengths. Like with the number of comparisons made, because of the smaller number of elements in the array given to bubbles and the limited memory allocated to the virtual machine, the graphs for bubble and heap are not very visible. When comparing the graphs for shell and quick sort however, we can see that like with comparisons, the number of moves for the two methods compare in the same way. Where the number of moves for shell increases rapidly as the number of elements in the array increases, the number of moves for quick sort increases at a slower, steadier pace. This is, as we know, due to the fact that quick sort simultaneously sorts either side of the pivot point, and therefore, uses less number of moves overall when compared to shells sort which can be compared to bubble sort but over a greater range/interval.

-Times(referred to @mglapagr's script):



This graph shows the amount of time that each sorting algorithm took to sort arrays of exponentially increasing lengths. Because of the fact that the limit on the number of elements in the arrays given to bubble sort was a lot less than the the element limit in the arrays given to the other sorting algorithms, the amount of time taken by the bubble sorting method for an array of the greatest length(within the limit specified) is so little that it is not very visible on the graph outputted. Likewise, due to the segmentation fault that occurs for heap because of the limited memory allocated to the virtual machine during set up, the amount of time taken for the last array sorted by heap is too little to show obviously on the graph with all four sorts. When looking at the time taken by the shell and quick sorts, this is where it becomes extremely clear that quick is the most efficient method for a reason. Even at the max number of elements, quick sort takes almost ten seconds less than shell sort to finish sorting the array in ascending order. Even when dealing with reverse arrays, the times taken by both sorts are similar to what is shown in the graph because shell only has a worst case time complexity of $O(n^2)$ if the gap sequence that it uses is not of an appropriate size for the given array, and quick only has a worst case time complexity of $O(n^2)$ if the partition happens to be at either of the array ends. In this way, we can see and conclude that quick sort is generally the best sorting algorithm to use.

Overall:

From this assignment, I learned a great deal about the different types of sorting methods that are used by people. Before this assignment, the only sorting algorithm I knew about and had used previously, especially in first CS class I took freshman year(Intro to Java), was the bubble sort method. I had never heard about the shell sorting algorithm or the heap sorting method before and even quicksort I only vaguely knew. After experimenting with these four methods, I learned and actually was able to see how these other methods of sorting were a lot more efficient than bubble sort which was introduced to many of us in our very first coding class. Specifically, I understand why quick sort is widely considered to be the most efficient of the sorting algorithms and now that we coded it, I get why it is efficient because obviously, if an array is split into two parts and sorted *simultaneously*, it will take a lot less time to sort the whole thing than if you were just taking it as one whole array and sorting it two elements at a time. With shell, though I get why it may be a more efficient algorithm to use when compared to bubble, I do not necessarily think that this would be the method that anyone would *tend* to use especially because if the appropriate gap sequence is not provided, shell would sort the array only slightly more efficiently than bubble. Lastly, in regards to heap sort, I think that though it is advantageous to have a stable time complexity across the board— regardless of worst case, best case, or average case— because of *how* it sorts in relation to how quick sort sorts, I think it is typically better and more efficient to utilize quick over heap, particularly when one needs to sort a large number of elements at once. I think however, that it would be cool to perhaps combine algorithms because in the case that the worst case *does* occur for quick sort, it might be beneficial to switch to a heapsort algorithm since its worst case is only O(nlog(n)). I was considering this because sometimes even the best of things have its own downfalls, and similar to how hybrid cars switch from electric to gas when its battery runs out, if it was possible to switch from quick to heap if the need arises would be very beneficial.