Analysis – Sorting: Putting your affairs in order

Student name: *Serjo Barron*

Course: *CSE13S* – Professor: *Prof. Long*
Due date: *10/17/21*

1 Introduction

Sorting is a fundamental operation used for the sake of giving information to our data, it tells us where to we might want to look to find a certain item. Just like phone books, they are sorted in alphabetical order, and because of that, it certainly makes it easier to find the phone number of a relative of ours simply by knowing their first initial. A world without sorting would disorganized and chaotic.

Since sorting is such an important operation, it is only right that programmers write code based on sorting algorithms to efficiently or even inefficiently sort things. This paper in particular will focus on 4 sorting algorithms – The Insertion Sort, Shell Sort, Heapsort, and Quicksort algorithms. All of which have their strengths, weaknesses, practicality, efficiency, etc. All of this we will be taken into consideration and used to conclude the findings.

2 Methods

The data presented in this paper has been produced and made in the programming C language. The numbers which will be sorted are all 32-bit unsigned integers and have been generated pseudo-randomly using a seed. This means the elements generated to be sorted are random, but since they are made using a seed the pattern in which they are random is repeatable.

All the graphs shown in this paper have been generated and made with Gnuplot. It should be noted, due to the vast amount of numbers to be plotted, the graphs which deal with very large array lengths have been adjusted accordingly to use a proper increment value.

3 Results/Analysis

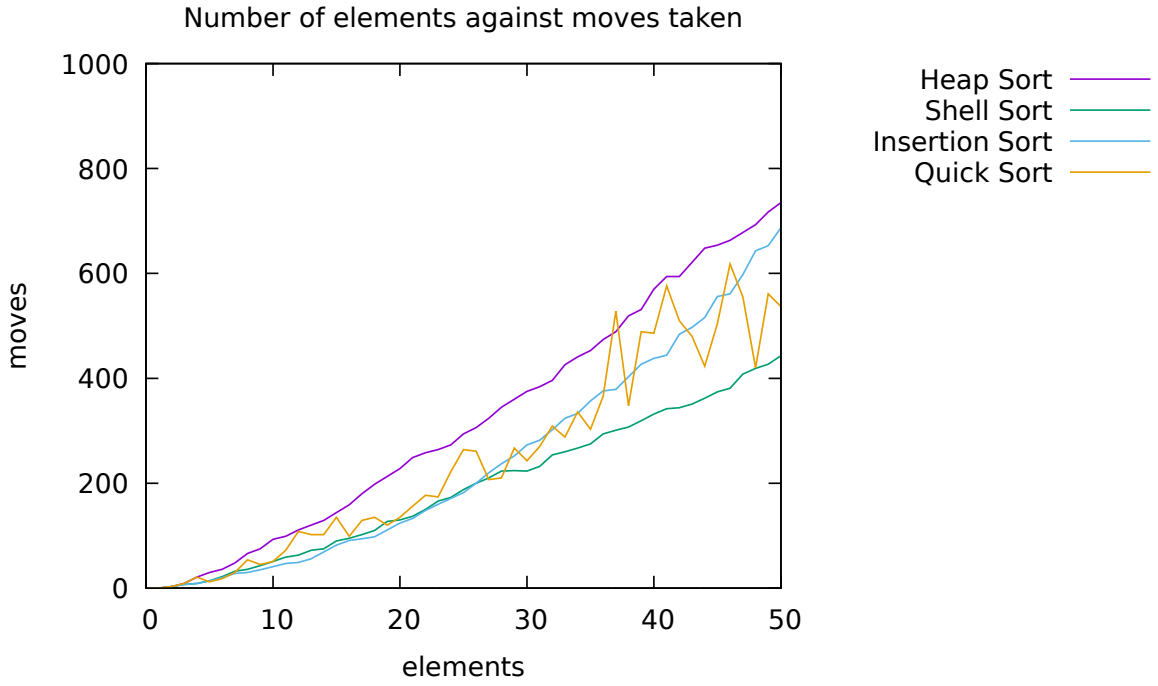
3.1 Array Length Against Moves: Randomly Generated Arrays

This particular subsection will examine various cases, 3 to be exact of how the number of moves each sorting algorithm takes when the array length n is changed. The 3 scenarios in which the sorting algorithms will be tested in will be split into a low,

medium, and high amount of array elements to be sorted.

All of the graphs that will be examined in this subsection have been labeled with the x-axis representing the number of elements, with each value signifying a different array of length n being sorted. The y-axis will be representing the number of moves the sorting algorithm took for the corresponding number of elements.

3.1.1 Elements Against Moves Taken: 50 elements



The first graph to examine is the one above, which displays how the 4 sorting algorithms perform with a small range of array lengths, 0 to 50 in increments of 1.

As seen in the graph, the overall best-performing sorting algorithm appears to be Shell Sort. Though, if we look closer Insertion Sort performs the best out of all the 4 sorts between the range of 0 to around 25 elements, but then later on is overtaken by Heap Sort in fewer moves taken.

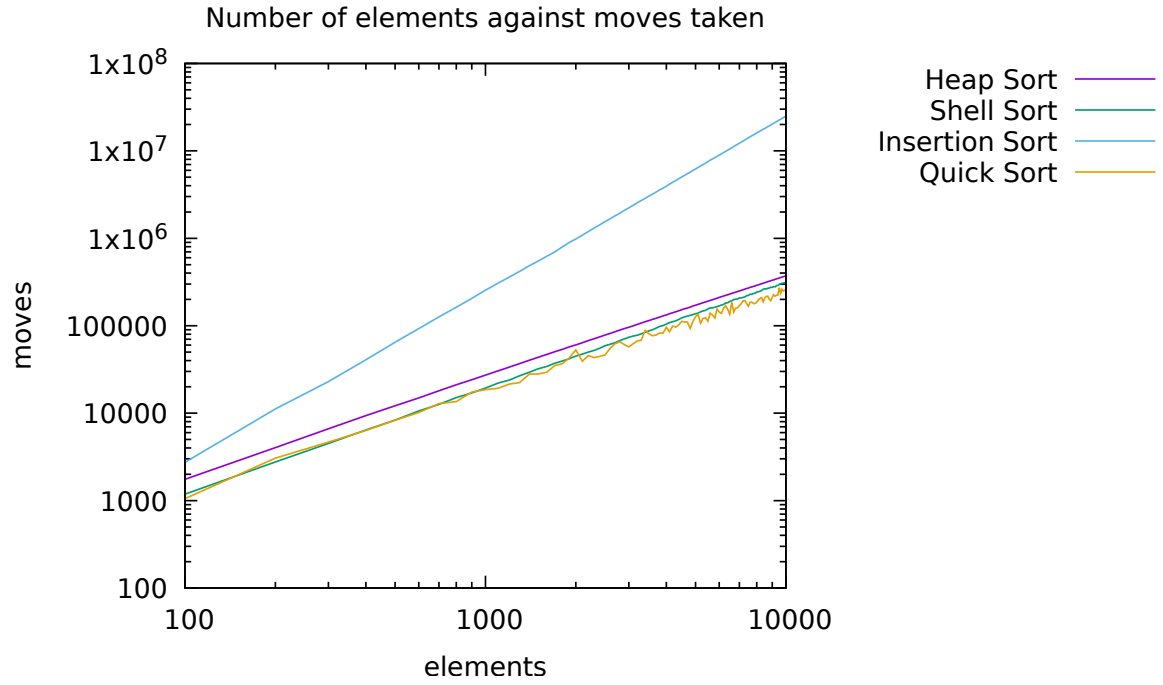
In terms of the behavior of the 4 sorting algorithms, Heap, Shell, and Insertion Sort all appear to be fairly linear with little oscillation. Though as you may have noticed, Quick Sort undeniably has serious oscillations with the way it jumps high and on low on the y-axis as the number of elements inside the array being sorted increases.

The oscillating graph by Quick Sort is likely caused by the algorithm itself, particularly the randomness of its partitions. Since the partitions choose a value (typically the first element) in the array and use that as a pivot to compare to the other elements to partition them as less or greater than the pivot, there is a small chance the pivot value chosen is next to (or is) the lowest or greatest element. So of course, the distribution of the elements would be very skewed to one side or the other (left or right). Would one case of that happening be enough to oscillate that much? Most likely not. The

oscillations, in particular, would be caused by multiple unlucky situations, so in an array of let's say 10,000 elements the pivot value by small chance just happens to be that unlucky number over and over again.

That being said, the worst-performing algorithm with a low amount of elements is undoubtedly Heap Sort. Though it performs the worst of the 4 sorts in this graph, we will examine how that changes in the next graph scenario.

3.1.2 Elements Against Moves Taken: 10,000 elements



This second graph examines the same 4 sorting algorithms as analyzed in **3.1.1**, but this time we will examine how the 4 sorting algorithms perform with 10,000 elements. The graph above starts with array lengths of 100 to 10,000, with increments of 100.

The first noticeable change between the previous graph and the current graph is the behavior of Insertion Sort. As examined previously, Insertion Sort did quite well in a range of 0 to 50 array lengths, but now that we are dealing with array lengths in the thousands, Insertion Sort has undoubtedly lost in this race and is by far the most inefficient sorting algorithm for large sets of data.

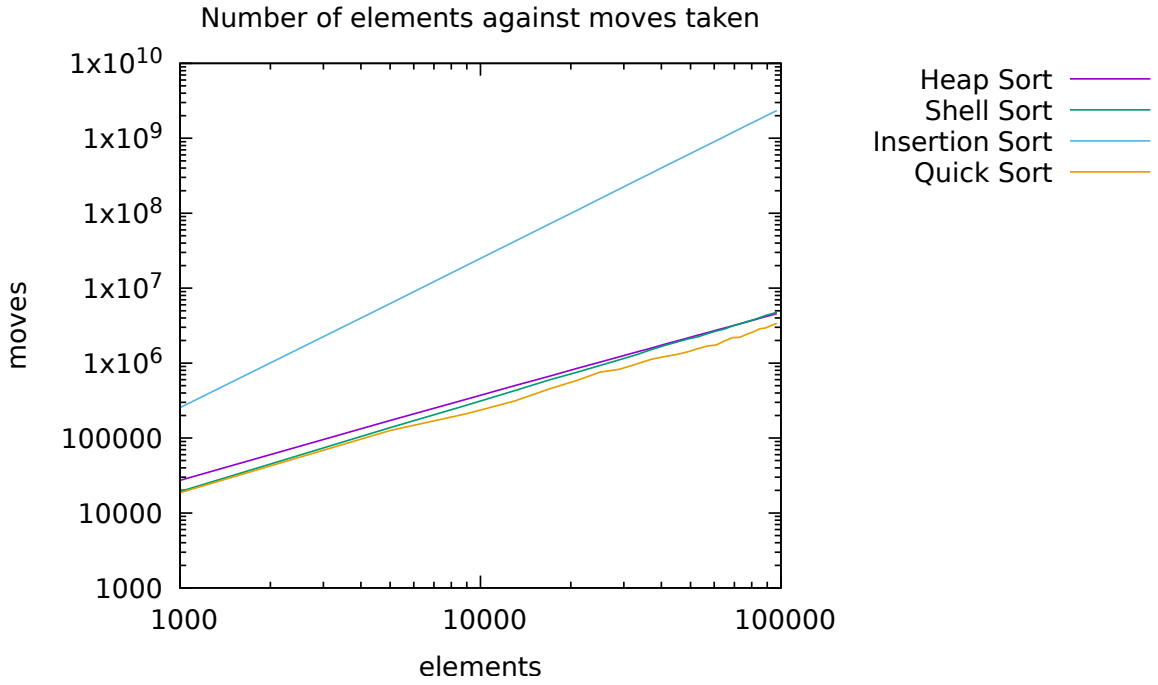
On the other hand, the other 3 sorting algorithms: Heap Sort, Shell Sort, and Quick Sort are all fairly close to each other and without a doubt perform far better than Insertion Sort. Heap Sort and Quick Sort in particular made a big improvement in this graph as they were among the worst-performing algorithms when it came to small array lengths.

Quick Sort has now become the best performing sorting algorithm despite its still noticeable oscillations. Because the graph displays far more cases of array lengths the

oscillating behavior has tamed down a lot, but what's being shown is an average performance of Quick Sort.

Shell Sort though has amazingly performed quite well in the first two graphs, in both dealing with small array lengths and large array lengths. The versatility of a sorting algorithm is not something to be shrugged off lightly, as it has already been seen that Heap Sort, Insertion Sort, and Quick Sort all had their own caveats. In the next graphing scenario we will examine an extreme case of array lengths.

3.1.3 Elements Against Moves Taken: 100,000 elements



The third graph displaying the number of elements against moves taken will examine how the 4 sorting algorithms perform with 100,000 elements. The graph starts at an array length of 1,000 and increments by 4,000.

Similar to the previous graph, Insertion Sort has left the competition and is taking approximately 1,000,000,000 (1 billion) moves to sort an array of 100,000 elements in increasing order. This is astonishingly larger than the other 3 sorting algorithms which are only in the area of 3,000,000 moves.

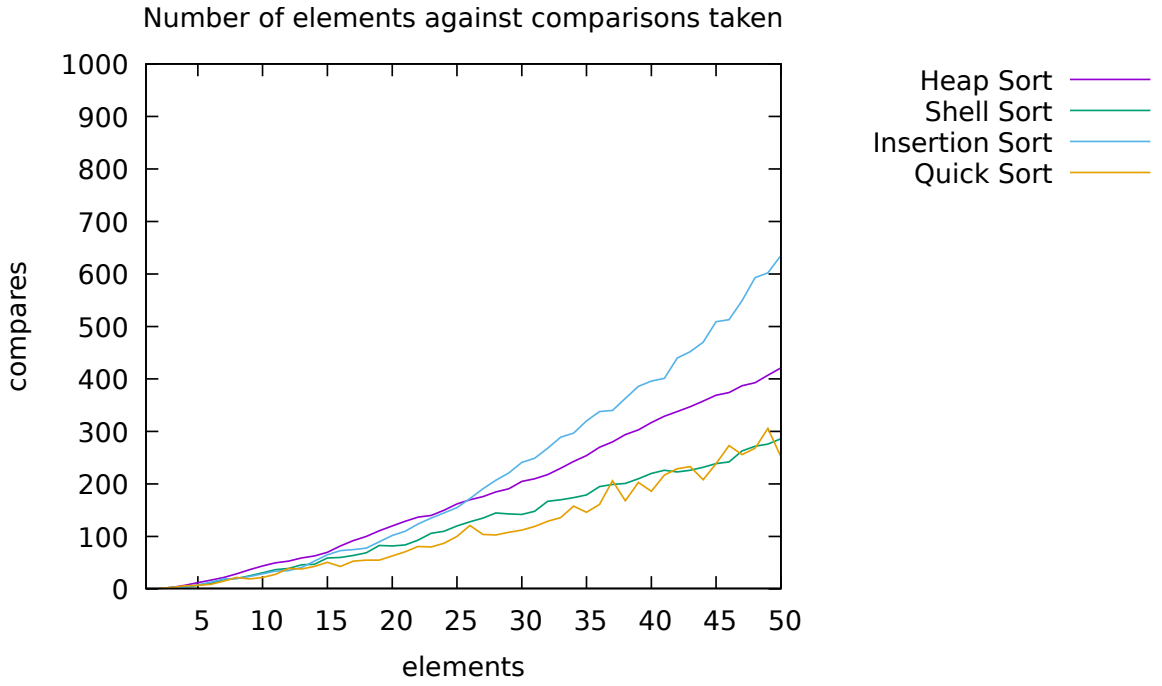
As expected, Quick Sort continues to perform the best out of the sorting algorithms. But, an interesting thing to take away from this graph is that at the very end of the graph (at the 100,000 length array mark) Heap Sort begins to beat Shell Sort in taking the least amount of moves possible to sort an array of extreme lengths.

3.2 Array Length Against Compares: Randomly Generated Arrays

This subsection will continue to use graphs that examine how the 4 sorting algorithms deal with pseudo-randomly generated arrays, but instead will look at a statistic called

compares. The statistic compares is used inside each of the sorting functions to keep track of instances in which the elements of the array are compared against each other (one on one). What does this tell us? Every time the array elements are compared usually occurs inside or is part of loop, so what the number of comparisons tells us is how many times a certain function (algorithm) had to iterate/loop to sort all the elements in increasing order.

3.2.1 Elements Against Comparisons: 50 elements



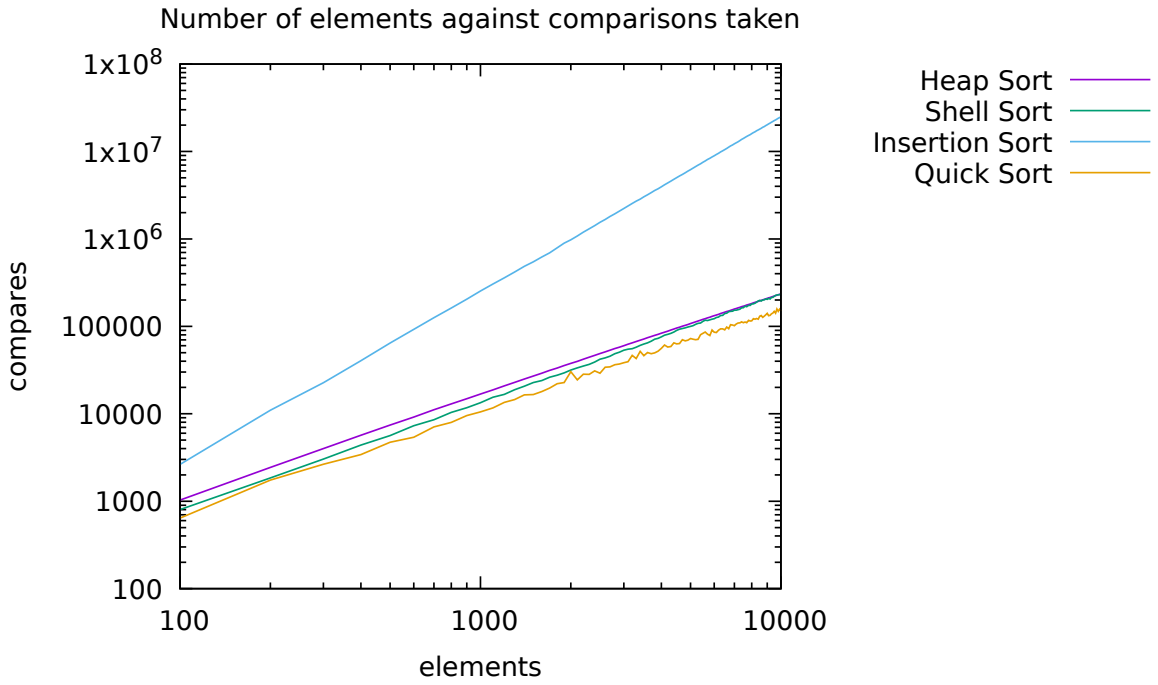
The first graph of elements against compares displays up to a range of 50 element arrays, where the elements are incremented by 1.

As seen in the graph, all the sorting algorithms appear to be within the same vicinity of comparisons taken when dealing with arrays of length 0 to 15, but afterwards they clearly separate. The most obvious algorithm rising in the number of comparisons is Insertion Sort. After Insertion Sort's almost linear behavior between 0 to 15 elements, it rapidly increases to the top and is taking approximately 625 compares to sort an array of length 50.

Next let us observe the best performing algorithm, Quick Sort, which despite continuing to have quite noticeable oscillations, it is for the most part taking the least amount of compares aside from some unlucky cases where Shell Sort wins.

Heap Sort, though it does not perform as badly as Insertion Sort does, it certainly has a large gap between the amount of compares it takes when compared to the best-performing algorithms Shell and Quick Sort when it comes to the least amount of comparisons. Though this is the case now, let us observe if this behavior stays the same or changes in the next graphs.

3.2.2 Elements Against Comparisons: 10,000 elements



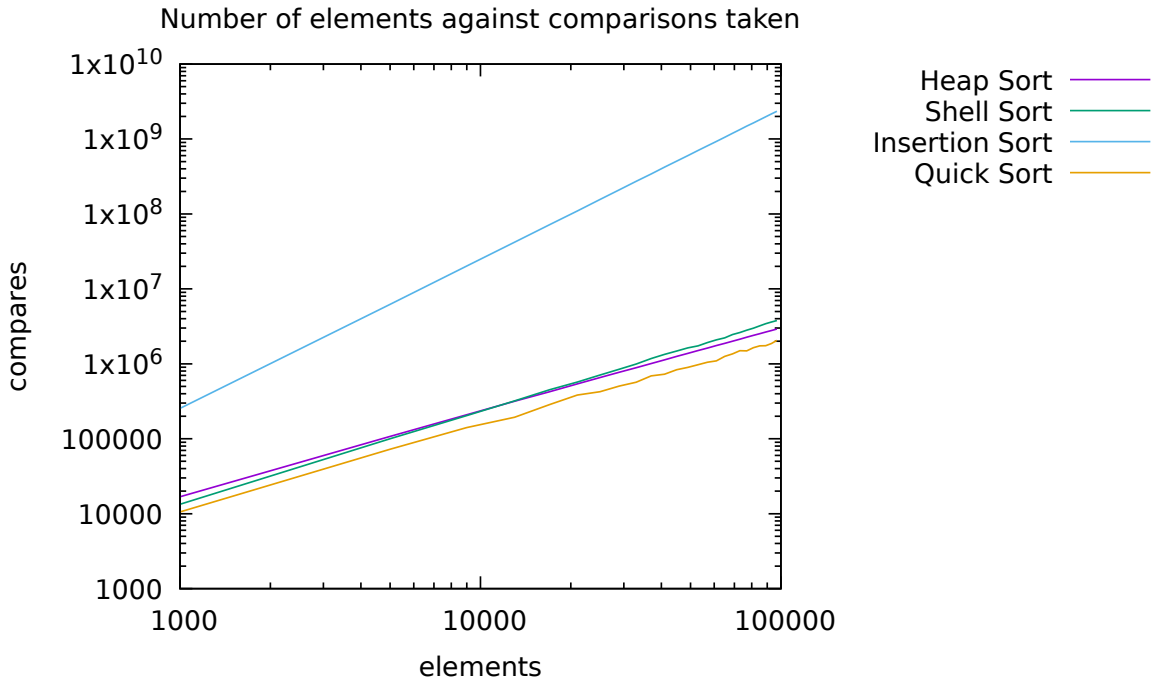
The second graph of elements against compares displays up to a range of 10,000 element arrays, using increments of 100.

As expected, Insertion Sort retains the same end-behavior it had in the previous graph displaying up to 50 element arrays, and is undoubtedly taking much more comparisons to sort arrays of far larger length.

Quick Sort remains in the lead for being the sorting algorithm taking the least amount of compares. But an interesting change is how near the end of the graph, Heap and Shell Sort appear to intersect.

In the next graph, the behavior of Heap and Shell Sort will be examined when sorting arrays that are much larger than 10,000 elements.

3.2.3 Elements Against Comparisons: 100,000 elements



The third and last graph of elements against compares displays up to a range of 100,000 element arrays, using increments of 4,000.

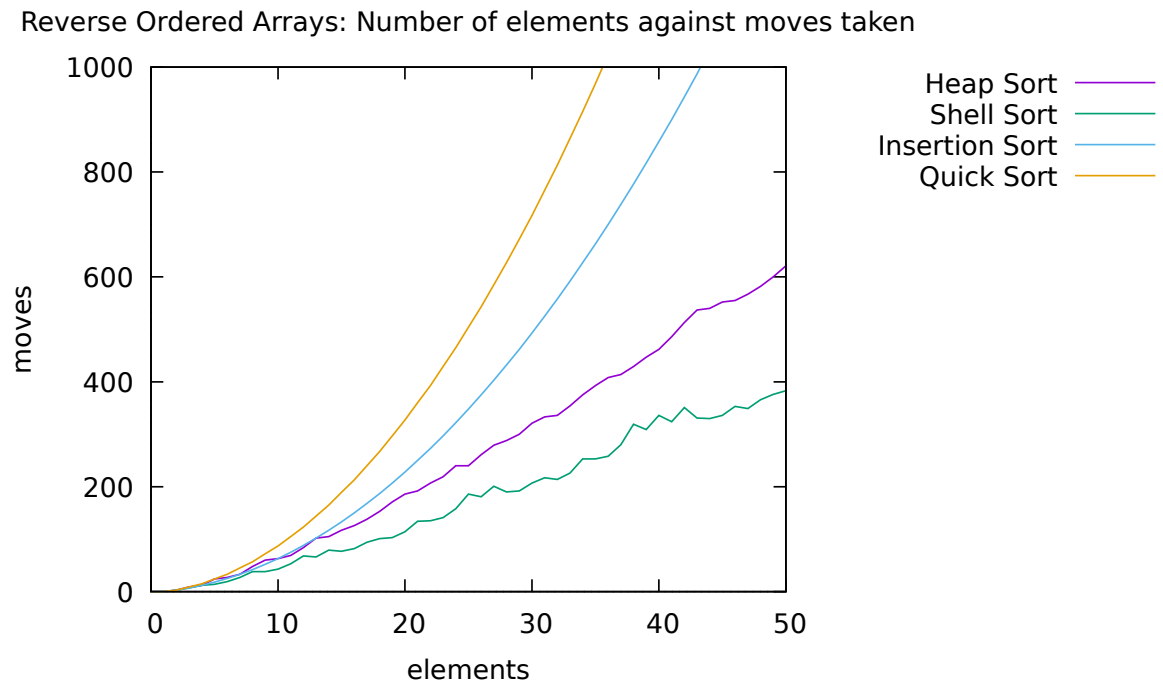
As hinted in the previous graph, the gap between Heap and Shell Sort does shorten up to the point that Heap Sort overtakes Shell Sort in terms of taking the least amount of compares to sort an array of lengths in the ten-thousands range and above. This is visually more clear when you see the gap widen once again between the Heap and Shell Sort algorithms near the 100,000 elements mark, but this time Heap Sort is in the lead.

So what do these graphs exactly tell us about the sorting algorithms? As mentioned previously, the amount of compares can give us a fair idea of how many iterations an algorithm takes to sort an array. That being said, these graphs (especially ones graphing larger ranges of array lengths) give us a good idea of the time differences it takes for each algorithm to successfully sort an unsorted array in increasing order.

3.3 Array Length Against Moves: Reverse Ordered Arrays

This subsection will take on an interesting scenario, what if the arrays being sorted were already sorted, but in reverse order? An example of this would be an array such as [9, 8, 7, 5, 1], so that the array has been arranged in such a way that may or may not impact the performance of the 4 sorting algorithms.

3.3.1 Reverse Ordered Arrays – Elements Against Moves Taken: 50 elements

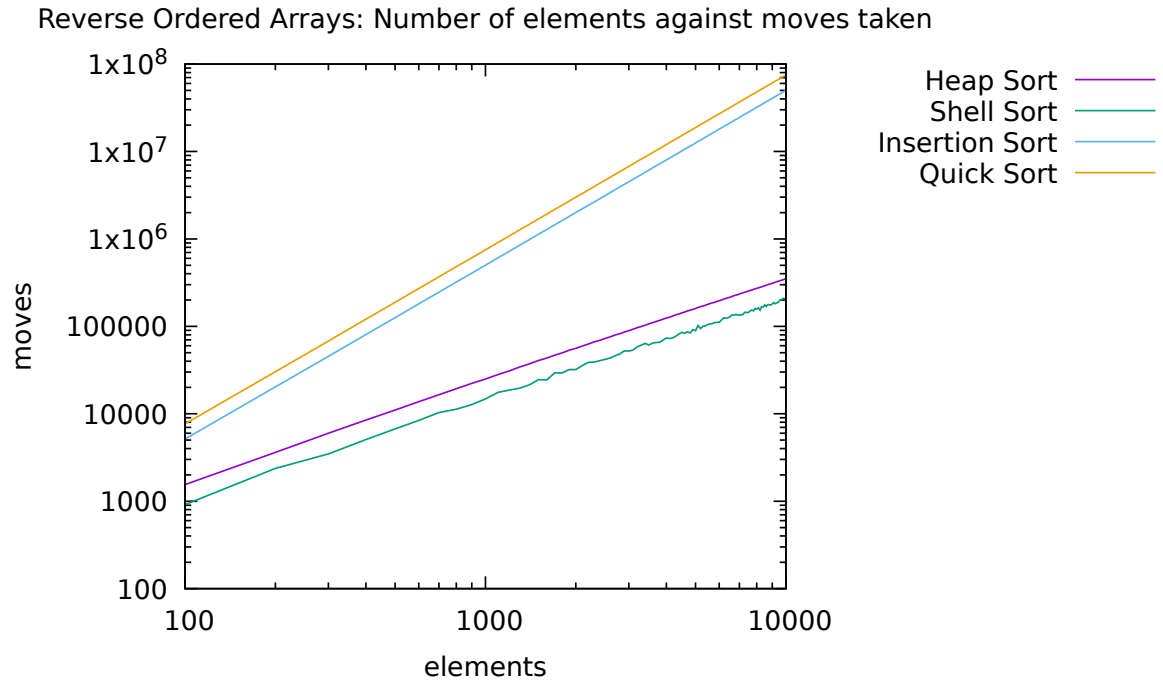


The graph above displays how the 4 sorting algorithms perform when sorting an array intentionally made to be reverse sorted. This graph in particular displays up to an array length of 50, with increments of 1.

As seen, Shell Sort remains to be in the lead of being the most versatile and efficient graph by taking the least amount of moves out of the 4 sorting algorithms. Heap Sort behaves similarly to Shell Sort, but as seen still takes more moves than Shell Sort.

The most drastic change in this scenario of reverse ordered arrays though appears to be the new worst-performing algorithm, that being algorithm being Quick Sort. As seen in **3.1** which dealt with sorting pseudo-randomly generated arrays (the most realistic and practical scenario), Quick Sort inevitably won in the end for being the fastest algorithm to sort arrays of large lengths. Despite that, as seen in the graph above Quick Sort at approximately 35 elements goes out of bounds, meaning that it requires more than 1,000 moves to sort a reverse sorted array. This is likely caused by the nature of partitions (the chosen pivot), which we will go into more detail in the next subsection when dealing with ordered arrays.

3.3.2 Reverse Ordered Arrays – Elements Against Moves Taken: 10,000 elements



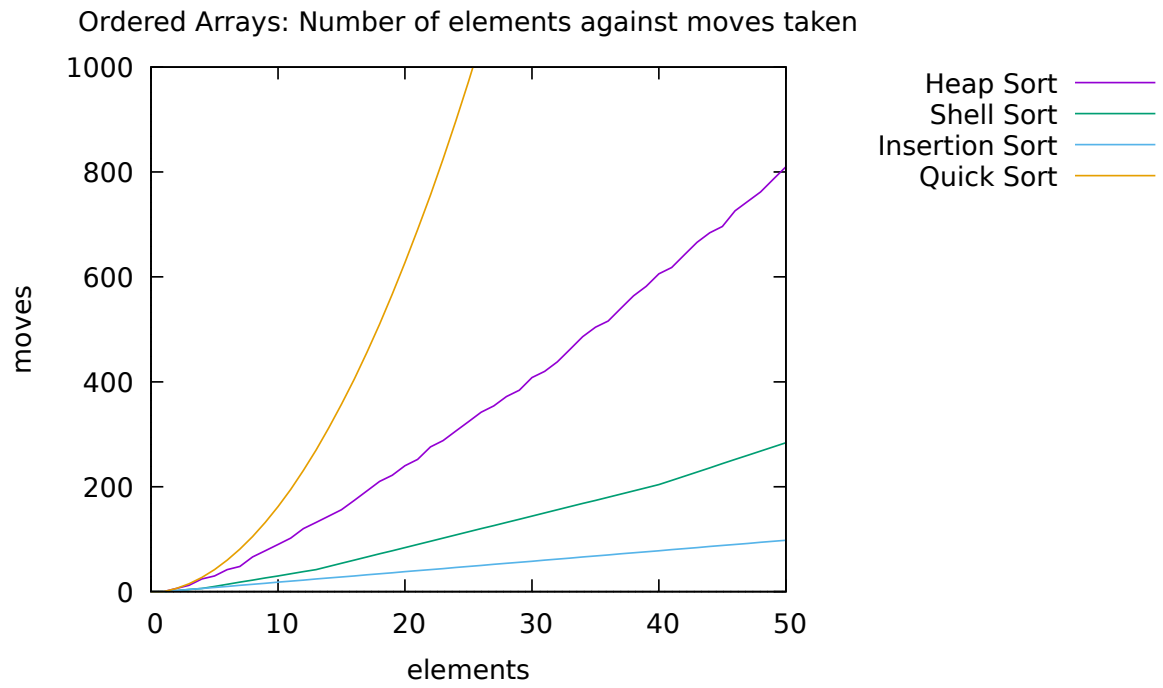
The next graph above still uses a reverse sorted array to sort, but now displays up to 10,000 elements, with increments of 100.

Just like the previous graph, the sorting algorithms remain in the same order in terms of most efficient to least when dealing with reverse sorted arrays. And as seen in a graph with larger array lengths the gap between the worst-performing (Quick and Insertion) and the best performing (Heap and Shell) widens.

3.4 Array Length Against Moves: Ordered Arrays

In this next subsection we will examine the opposite of what was graphed in **3.2**. This time we will be graphing how the 4 sorting algorithms perform when sorting arrays that are already sorted. Of course, this scenario is only for the sake of knowing which of the algorithms can notice that the array they are dealing with is already sorted, not how practical it would be.

3.4.1 Ordered Arrays – Elements Against Moves Taken: 50 elements

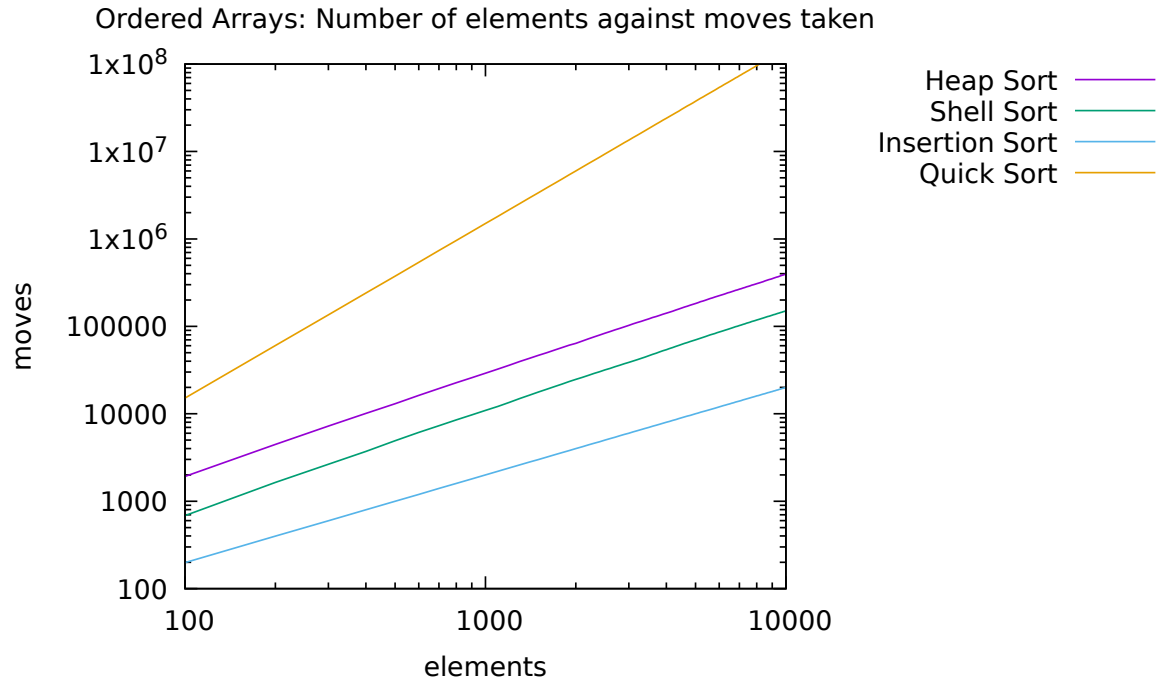


The graph above displays how the 4 sorting algorithms perform when sorting an array that has already been sorted. This graph in particular displays up to 50 elements, with increments of 1.

The first noticeable part of this graph is how Insertion Sort behaves, particularly in how linear and close to the x-axis it is. This likely has to deal with how the Insertion Sort algorithm works, which in brief loops through the length of the array and compares the current element it is on to the preceding elements, and if that value is less than any of the preceding elements, it will be moved before the element. Because of this, the number of loops being done to check the array is simply cut down to the original length of the array 'n', or in other words, it would take $O(n)$ time.

On the other hand, Quick Sort undoubtedly performs the worst. This is caused by how partitions work, since the pivot element chosen is typically the first element, in a normal scenario there would be a small chance that the value of that element is the smallest or largest value of that array. But since the array is already sorted, the partition will cause one side to include all the values and the other to include none. Of course, this will change as new pivot values are used, but that being said this is an example of the Quick Sort algorithms worst scenario which takes $O(n^2)$ time.

3.4.2 Ordered Arrays – Elements Against Moves Taken: 10,000 elements



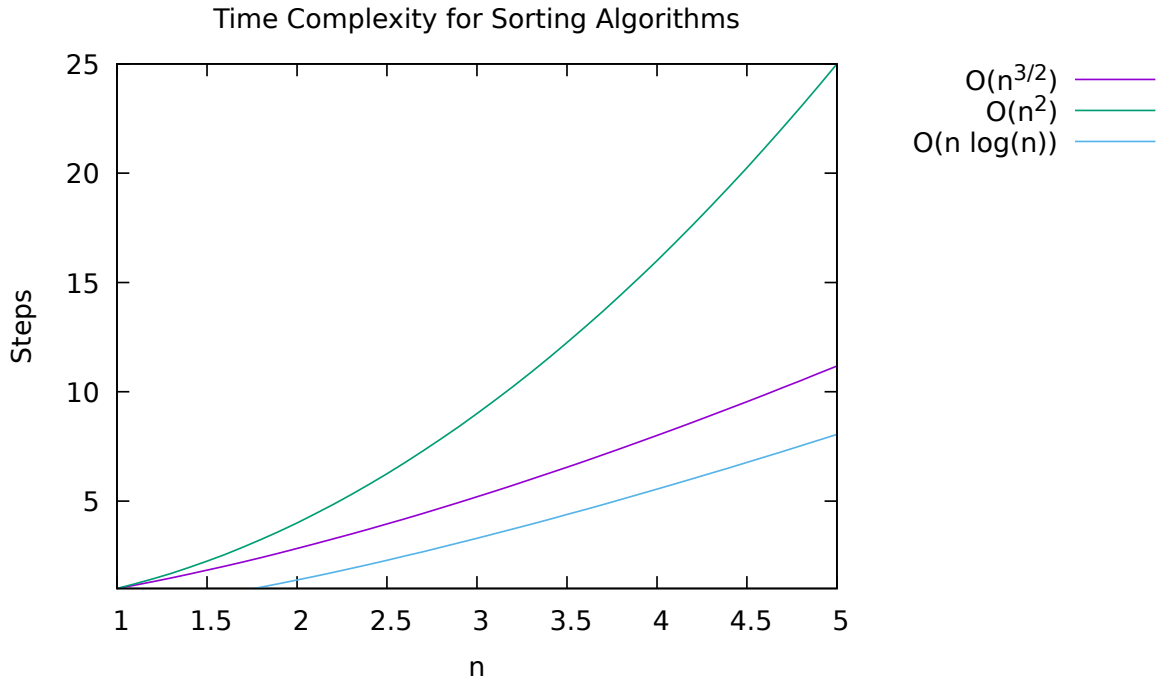
The next graph above continues to use a sorted array to sort, but this time displays up to 10,000 elements with increments of 100.

Like the previous graph, the sorting algorithms remain in the same order of efficiency (greatest to worst). That being said, it can be assumed that the linear behavior displayed above will continue for the 4 sorting algorithms when dealing with already sorted arrays.

3.5 Time Complexity

In the final subsection of the Analysis portion of this paper, of the 4 sorting algorithms: Heap, Insertion, Shell, and Quick Sort(s) we will discuss and graph each of the algorithms average time complexity in the form of "Big O notation" which will describe the limiting behavior of the 4 sorting algorithms.

In addition, we will only be discussing the time complexity of pseudo-randomly generated arrays to keep things relevant and practical.



As seen above, there are 3 time complexities graphed, all of which correspond to the 4 sorting algorithms, and one of the 4 sorting algorithms can be represented to have 2 cases, which will be discussed later on.

Going from the highest growth rate to the lowest, the first Order (Big O notation) we will examine is $O(n^2)$ and this growth is an example of the Insertion Sort, which is indeed supported by the previous graphs, particularly in **3.1** which examined how the Insertion Sort algorithm typically rose to be the fastest-growing algorithm when it came to how many moves it took, which also means it was the algorithm which took the most loops (inside its coded function) to sort the elements of an array.

Next, $O(n^{\frac{3}{2}})$ is an example of the Shell Sort algorithm. And as noted by previous graphs, Shell Sort was seen to be quite reliable in all forms of array length, whether it be small such as 50, larger in the thousands range, or in extreme cases of the hundred thousand range. It should also be noted that the Order of the Shell Sort algorithm is in most cases $O(n^{\frac{5}{3}})$, but this case has adopted Donald Knuth's gap sequence which does indeed cut down the number of steps as graphed above, this is also another reason Shell Sort was able to compete with other algorithms such as Heap Sort for quite a length of time.

Lastly, $O(nlog(n))$ is an Order that is used by both the Heap and Quick Sort algorithms. And as visible by the graph above, it is by far the most efficient Order when it comes to the least amount of steps. What does this tell us? Similar to what was discussed with $O(n^2)$, an Order tells us as computer scientists the approximate amount of loops or recursion an algorithm takes to produce its output.

Though as hinted at earlier, there is a sorting algorithm that has 2 cases, specifically an average case and a worst case, that sorting algorithm would be Quick Sort. Quick Sorts cases would be $O(nlog(n))$ as its average case (as mentioned recently), and $O(n^2)$ as its

worst case. And as examined in the graphs of **3.3** where sorting algorithms dealt with already sorted arrays, this would be an example of the Quick Sort algorithm taking $O(n^2)$ to sort all the elements due to how the pivot works by splitting up the elements.

4 Conclusion

After analyzing the results and graphs, it has been made evident that certain sorting algorithms perform better than others, and that each sorting algorithm had its caveat (whether that caveat be minimal or important). Though of course, despite the caveats there are simply better algorithms to use over another, especially when it came to comparing Quick, Shell, and Heap Sorts with the Insertion Sort algorithm.

So what did we gain from analyzing these 4 sorting algorithms in various scenarios? It has been made clear that there is no one definitive sorting algorithm that can be used for all cases, which is not necessarily a bad thing, but what that means for us is that we need to be more considerate about the sorting algorithms we use to deal with certain amounts of data (which in this case, the data we deal with is arrays).

Being considerate of figuring out what the problem is (how much data we are dealing with), and the most optimal algorithm to solve that problem is imperative to us as computer scientists. As analyzed, if one were to use Insertion Sort on large element arrays, it would simply take far too many moves/steps to sort randomized arrays in ascending order. But perhaps we are dealing with very small arrays such as an array of length 10, which it would then be appropriate to choose a simple algorithm such as Insertion Sort.

That being said, it is very important to choose an algorithm appropriate to the complexity of a problem. An efficient algorithm will always win in speed (time taken to solve a problem), regardless of what machine the problem is solved on.