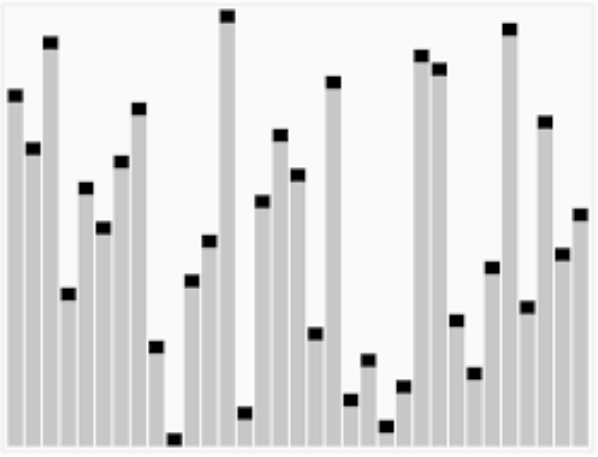
UNIVERSITY OF CALIFORNIA SANTA CRUZ

Analysis - Sorting: Putting Your Affairs in Order

Student name: *Nicholas Reis*

Course: *CSE13S* – Professor: *Darrell Long*
Due date: *October 17th, 2021*



Introduction

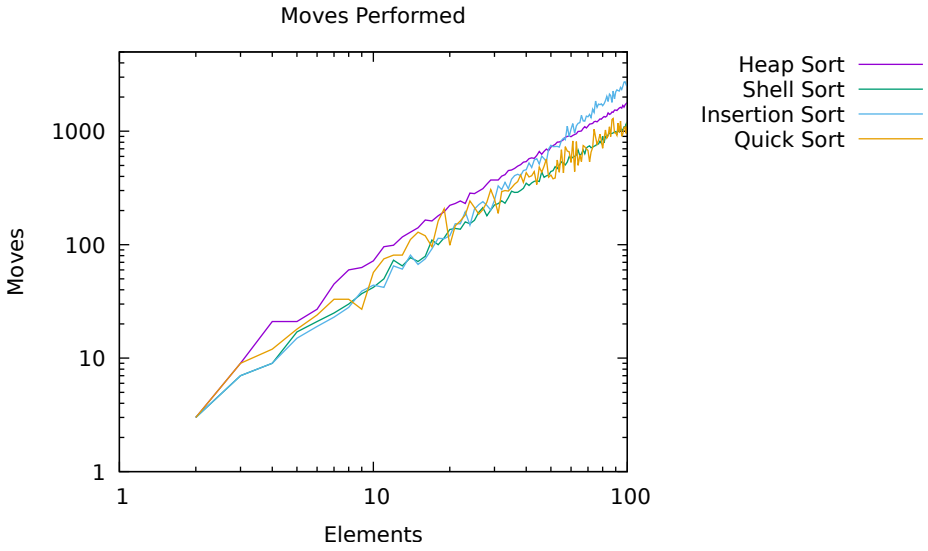
In this write up I will be demonstrating my findings pertaining to four different sorting algorithms. I implemented functions heap_sort(), shell_sort(), insertion_sort(), and quick_sort(), which each sort a given array with differing levels of efficiency, speed, and numbers of moves and comparisons. In the following documentation, I will be analyzing each of the sorting algorithms to determine under which conditions they perform well or poorly, along with other conclusions about their functionality.

Some of the testing performed will include: passing each algorithm small arrays, large arrays, and very large arrays. I also check the time it takes each of them to complete the sorts, along with passing the functions ordered and reverse ordered arrays. This should provide enough information to conclude which situations each algorithm works best, and which situations they should not be used.

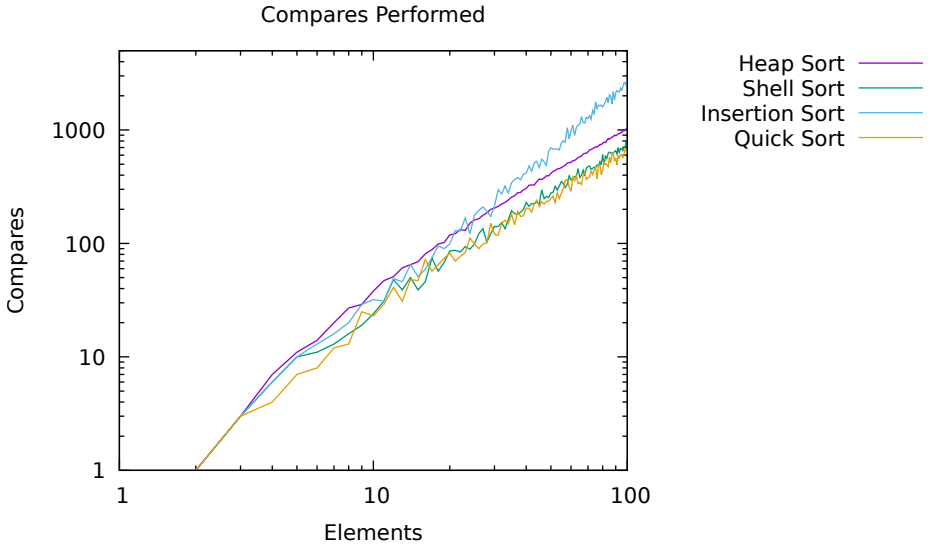
Results

Findings and Graph for Small Number of Elements.

(*a*) In this section, I will evaluate the effectiveness of each sorting algorithm on arrays that have between 1 and 100 elements. This is a relatively small size for arrays and each algorithm is quite quick to sort them, but let's take a closer look into some graphs to see how each of them do.



In terms of how many moves it takes each function, we can see that insertion and shell sort are the best within about 10 element long arrays. However, insertion sort gets worse quickly enough, as it tapers off after about 30 elements and even becomes the worst algorithm after 60. Shell sort maintains its position, with quick sort occasionally spiking down and sorting in less moves, but is not as consistent as the other algorithms. Heap sort is the worst in this range, at least until it becomes better than insertion sort after sorting 60 element long arrays. This graph highlights the disparity between quick sort's best and worst case scenarios, as it can sort in the least amount of moves with one array, yet sort in the most amount of moves with another (at least within this range of sizes). Heap sort and shell sort are the most consistent in this range, not fluctuating at all, yet heap sort does not seem to be very effective in this small range. We can see that insertion sort is useful and effective with a low number of elements, but not so much after arrays reach around 30 elements.

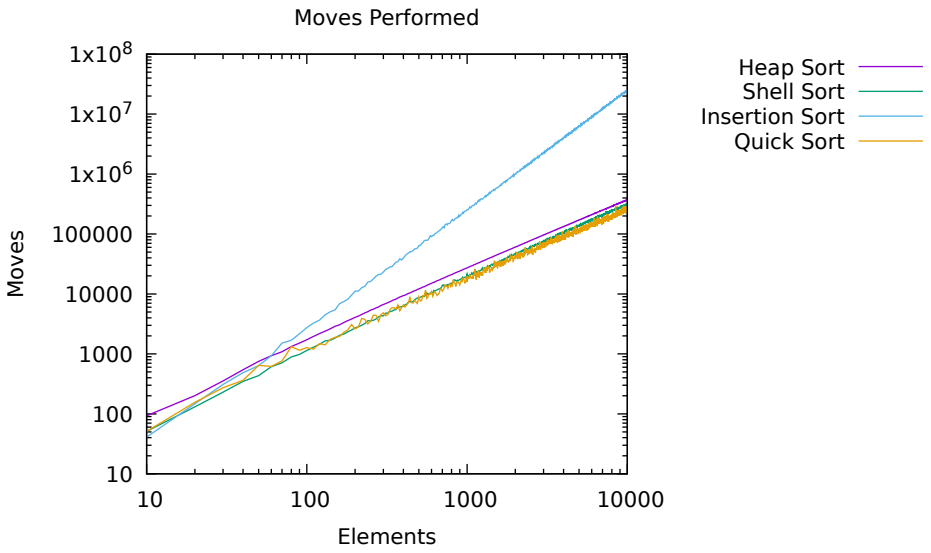


In terms of how many compares each function utilizes, we can see some slight differences from the previous graph. Quick Sort uses the least amount of compares

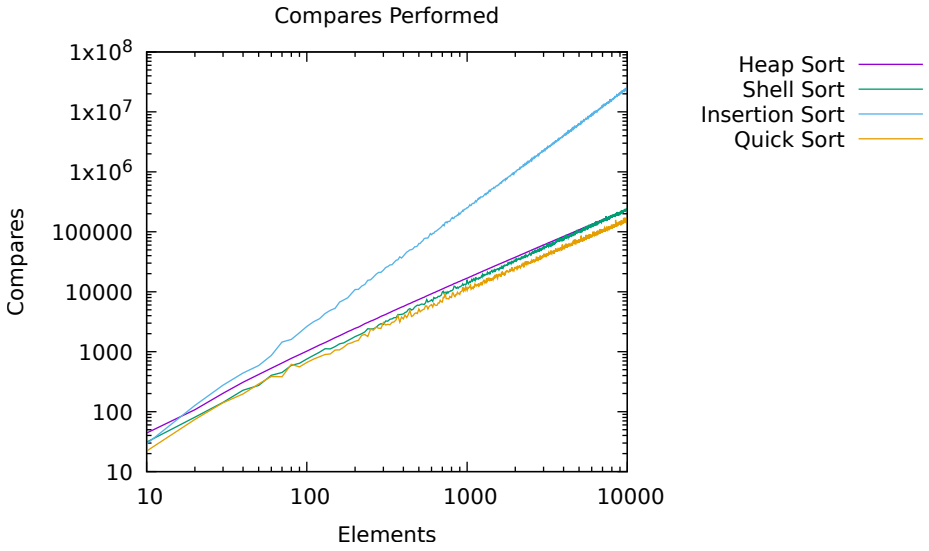
on average compared to the rest of the algorithms. Shell sort uses a very similar amount, only a slight amount more once it nears the 100 element range. Heap sort once again begins the worst, yet is overtaken by insertion sort around the 25 range. This data is interesting, as we can see how the different algorithms function in this range in relation to compares and moves. Quick sort, although using more moves than insertion sort, uses less compares over the first 30 elements. Each of the algorithms seems to end with a similar amount of compares and moves.

Findings and Graph for Large Number of Elements.

(*a*) In this section, I will evaluate the effectiveness of each sorting algorithm on arrays that have between 10 and 10,000 elements. These array become quite large, so some of the algorithms take a little longer to sort them. I recreated the graphs from the previous sections, but this time increased the range of the x and y axis and inserted much larger arrays into the algorithms.



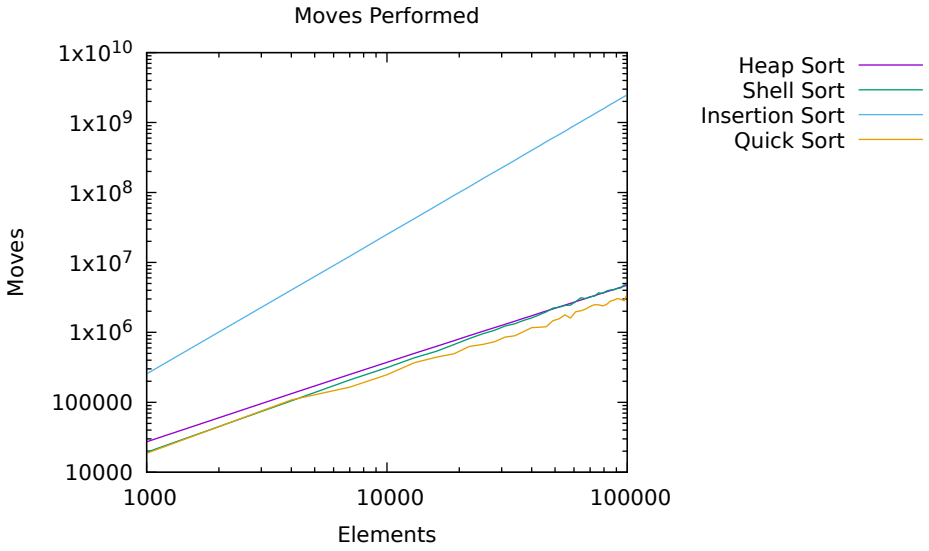
In terms of how many moves it takes each function in this range, we can see that insertion sort becomes exponentially worse than the other functions, and by the 10,000 element long array, it is nearing 100 times as many moves as the other algorithms. Quick sort still spikes quite a bit, however overtakes shell sort as the sort that takes the least amount of moves around the 500 element mark. It maintains this until the end of the graph, and you can even notice a small gap forming between them near the end. Heap sort remains behind shell sort, but seems to get nearer to it as the number of elements increases. From this, we can conclude that quick sort is the best for this range, with shell sort close behind, and heap sort close behind shell. Insertion sort should not be used past around 100 elements without it taking a very long time.



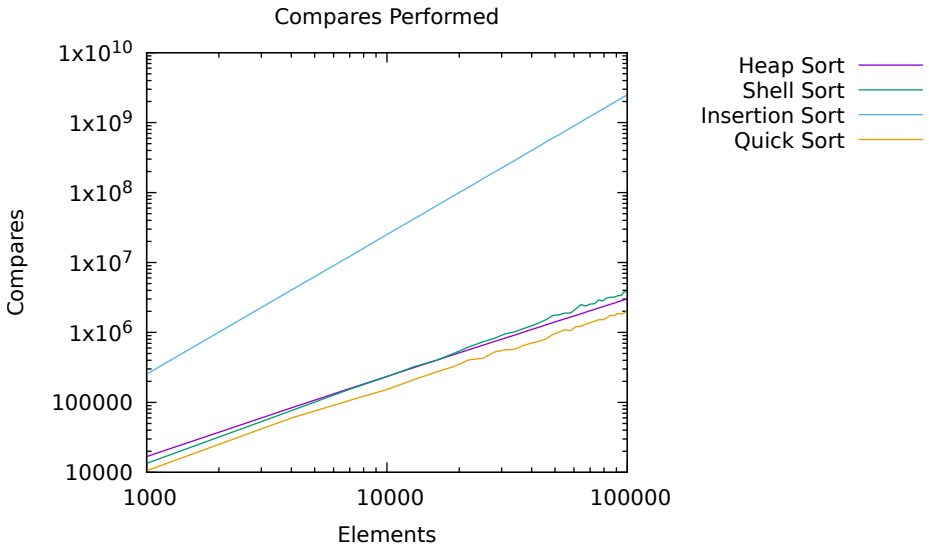
In terms of how many compares each function utilizes, we can see only minor differences from the previous graph. Quick sort still uses the least amount of compares consistently throughout the entire range, apart from a few spikes, and insertion sort still uses the most by a factor of over 100 times by the 10,000th element. However, we can more clearly see heap sort overlapping with shell sort by the end. Our previous conclusion still stands based on this graph, yet it is to be seen whether heap sort becomes a better sort than shell sort for high ranges.

Findings and Graph for Very Large Number of Elements.

(*a*) In this section, I will evaluate the effectiveness of each sorting algorithm on arrays that have between 1,000 and 100,000 elements. These array become very large, making the algorithms take a long time to sort them past around 50,000. I recreated the graphs from the previous sections, but this time increased the range of the x and y axis and inserted much larger arrays into the algorithms.



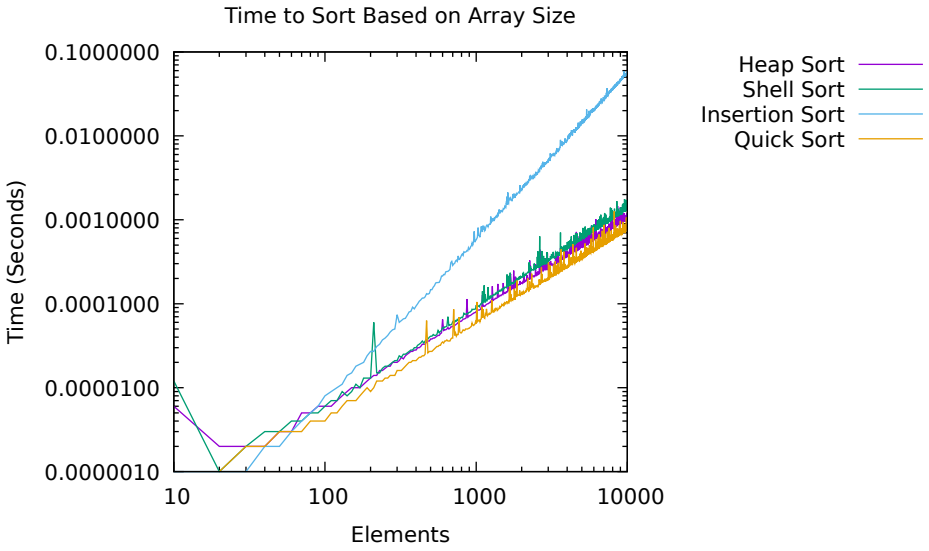
In terms of how many moves it takes each function in this range, we can see that insertion sort uses many more moves than the other functions, approximately 1,000 times as many by the 100,000 element array. Quick sort remains the best sort in this range, using the least amount of moves from around 5,000 elements to the end of the graph. The interesting part of this graph is heap sort and shell sort. Near the end we can see that heap sort overlaps and even uses less moves at some points than shell sort. This leads me to believe that heap sort will become clearly more effective than shell sort for arrays larger than 100,000 elements. To confirm my hypothesis, take a look at the next graph.



We can more obviously see in the comparisons graph that shell sort uses more compares than heap sort around the 20,000 element mark, and I assume the difference would be more noticeable as larger arrays are sorted. The previous conclusions about quick sort and insertion sort still stand with this graph.

Findings and Graph for Time to Sort Arrays.

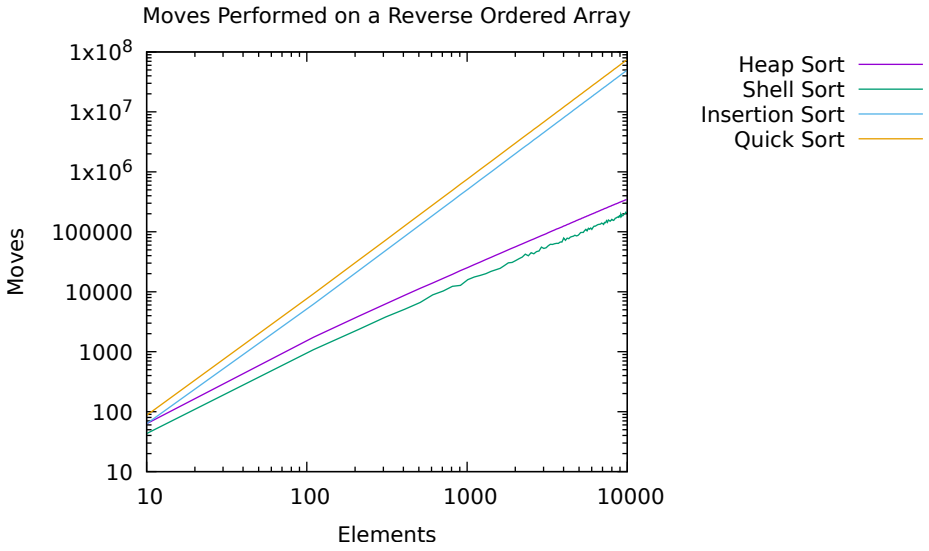
(*a*) In this section, I will evaluate the amount of time each algorithm takes to sort a given array of varying numbers of elements. To perform these calculations, I implemented the time.h header and used the clock() function to keep track of how long each algorithm took to run.



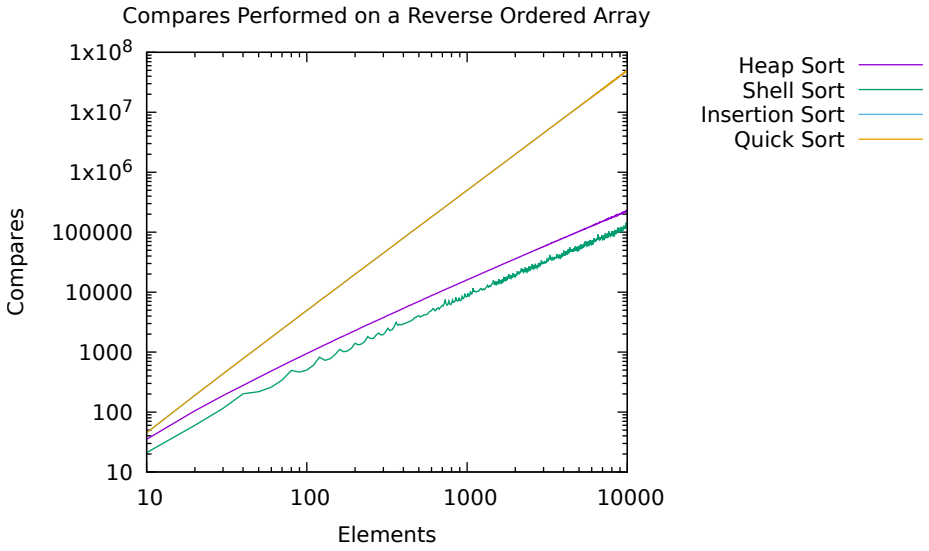
This time graph shows us similar results that we have seen in the previous graphs. Although insertion sort is quite fast for the first 60 or 70 elements, past that it becomes much slower than the other algorithms. Quick sort, fitting to its name, sorts the quickest for the most part over the course of 10 to 10,000 element long arrays. You may notice that the lines for each algorithm had a lot of spikes, and this is because certain arrays just end up taking longer to sort depending on their random positioning of the values. This causes some algorithms to do better and some to do worse at random points, like the very noticeable spike in the shell sort line at 200 elements. A fascinating observation is that heap sort, although taking more moves and compares over this range (as seen in previous graphs), still is actually a faster sort than shell sort. This contradicts what you might expect, and caught me by surprise when I created this graph.

Findings and Graph for Reverse Ordered Arrays.

(*a*) In this section, I will evaluate the effectiveness of each sorting algorithm on arrays that have between 1,000 and 100,000 elements. The difference from the previous analysis is that these arrays are ordered in reverse before being placed into the sorting algorithms.



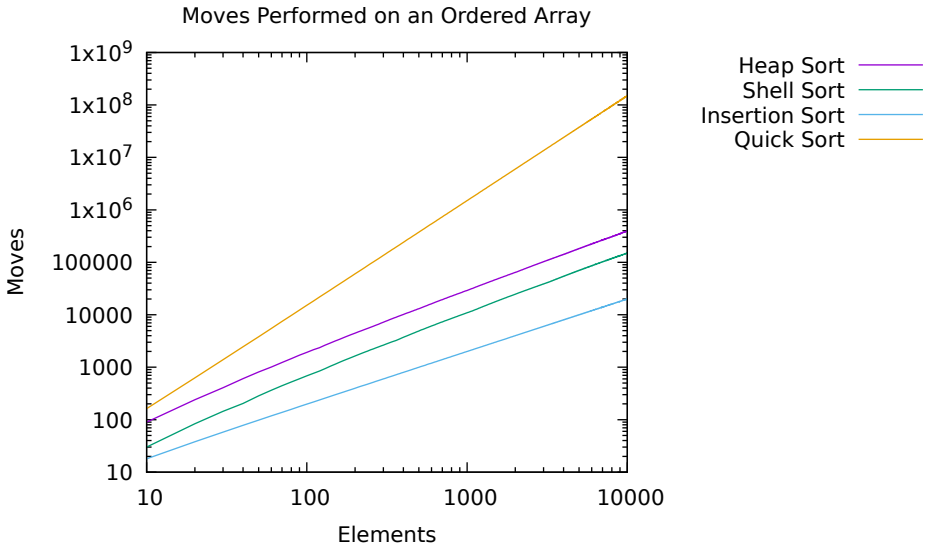
This is a very different graph than what we are used to with these algorithms. Here, we can see that quick sort takes the most moves to sort these arrays! Although we have already determined that in most cases with randomly organized arrays, quick sort is the best, when a reverse ordered array is passed to the algorithms, it is clearly the worst (in terms of moves). Insertion sort still takes almost as many moves, and is not much better here either. Shell sort uses the least amount of moves, and heap sort is second best. An observation that can be made here is that there is little to no overlap - the algorithms are very consistent throughout this range of elements. Thus, we can assume that these trends will likely continue for a very large number of array sizes (I tested up to 100,000 and the results remained the same).



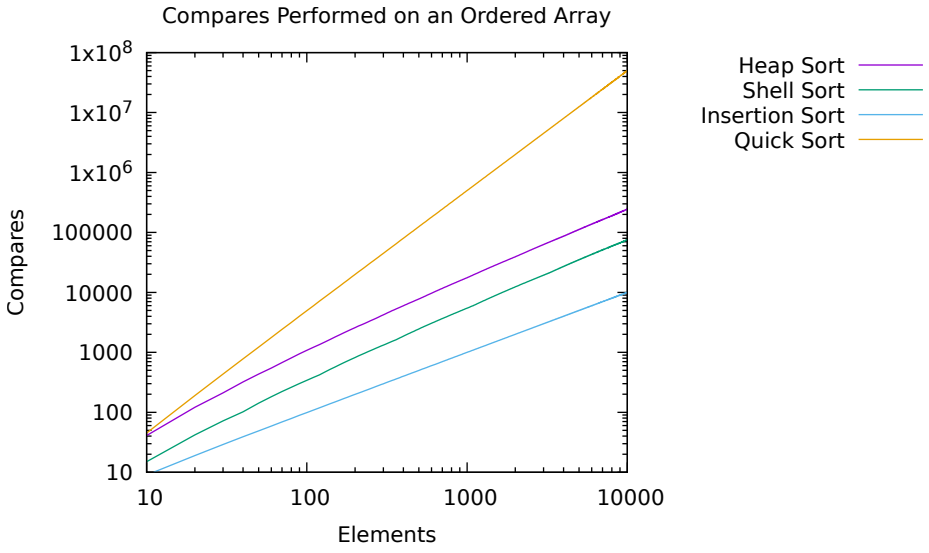
This graph may seem as though something is wrong, as there is no line for insertion sort! However, upon further testing, I noticed that quick sort and insertion sort actually use the exact same amount of compares throughout the entire graph. Other than that, most of the conclusions made with the moves graph still apply here.

Findings and Graph for Ordered Arrays.

(*a*) In this section, I will evaluate the effectiveness of each sorting algorithm on arrays that have between 1,000 and 100,000 elements. The difference from the previous analysis is that these arrays are ordered before being placed into the sorting algorithms.



Here we can see a flaw of these algorithms. They should not move anything around in an array that is already ordered, yet they still do. To fix this, I would need to implement a helper function that checks if an array is ordered, and if it is, I simply tell the function to return without editing the array whatsoever. However, I think these graphs are still interesting to look into, as they show us how these algorithms would react should the function not check if the array is ordered beforehand. We can see that quick sort performs very poorly, as it takes many more moves than any of the other functions to figure out how to deal with a sorted array. Surprisingly insertion sort performs better than all of the other algorithms, with shell sort being next best and then heap sort following that.



This graph does not tell us any new information from the previous graph, but it does confirm the fact that quick sort is the worst at dealing with ordered arrays, and that insertion sort is the best.

Conclusions

There are many conclusions that I've been able to make from the analysis of using my sorting algorithms with a variety of different inputs. Let's begin with how effective each algorithm is when they are passed arrays that have a small number of elements in them. I was able to determine that insertion sort is an effective algorithm for arrays that have 30 or less elements in them, however it performs more poorly than the other functions after that point. Quick sort and Shell sort are both quite effective in this range, taking a similar number of moves and comparisons to complete the sort. Heap sort is less effective than the other three for the first 30 elements, but becomes better than insertion after that.

For larger arrays between 100 and 10,000 elements, quick sort takes the least amount of moves and compares, followed by shell sort, and then heap sort. Insertion sort is considerably worse than the other algorithms in this range. Towards the end of this range, we can notice heap sort starting to use less compares than shell sort. For very large arrays between 10,000 and 100,000 elements, quick sort remains the best option, using the least amount of moves and compares. Heap sort uses less compares over most of this range, and begins to use less moves as well towards the end. Insertion sort, unsurprisingly, still does the worst with arrays of this size.

When I graphed the time each algorithm took to sort arrays of length 10 to 10,000, I noticed that it was relatively consistent with my other findings. The difference was that heap sort, although taking more moves and comparisons than shell sort in this range, actually runs faster. Quick sort runs the fastest and insertion sort runs the slowest.

When I passed the algorithms a reverse ordered array, I got very different results than my previous tests. Quick sort used the most moves and compares out of all the algorithms. Insertion used the second-most moves and tied quick sort for most compares. Shell sort performed the best, with heap sort being the second best.

When I passed the algorithms an ordered array, I got another different result. Quick sort still was the worst in this case, however insertion sort became the best. It dealt with ordered arrays in the least amount of moves and compares, followed by shell sort and then heap sort.

Overall, I would venture to say that quick sort is the most effective algorithm if they are passed randomized arrays, however it can perform very poorly if it is given sorted arrays or reverse sorted arrays. Shell sort and heap sort both are quite consistent across all of my tests, with shell sort being better for smaller arrays and heap sort being better for very large arrays. Insertion sort performs best with arrays smaller than about 30 elements, or if it is passed an ordered array, it will use the least amount of moves and compares.