Assignment 3:

Write Up Analysis of Different Sorting Algorithms

By Derfel Terciano

Introduction

Ever since the beginning of man-kind, collections of unordered items have been sorted by different means. For instance, the most ancient way of sorting is with *Insertion Sort*. It has been observed with how humans sorted items and eventually someone formally wrote down their observations and called it insertion sort.

We've always been striving to find the best and the most efficient sorting algorithm so I will be covering four different sorting algorithms that vary with how efficient they sort a given array. In this assignment, we are analyzing *Insertion Sort, Shell Sort, Heap Sort, and Quick Sort*.

Experimenting and Testing

In order to test each algorithm, I have developed a system that tells the user how many moves and comparisons an algorithm took in order to sort an array. From that system, I am able to generate graphs to see how well the algorithms did for sorting arrays. Now let's take a look at the system I have built to test each algorithm.

Before I build a test harness that tells us all the information about moves and comparisons, we need to implement our sorting algorithms. First, I implemented *insertion sort* which was very simple to implement. Next, I implemented *shell sort* which basically has the same implementation as *insertion sort* however, I needed a way to calculate the gaps which was difficult at first but was doable in the end. Then I implemented heap sort which was very overwhelming to write since it involved so many moving parts. For heap sort, I implemented the max heap sort which became overwhelming since the algorithm as a whole had so many moving parts to it.

For max heap sort, I had to find the biggest child in the array. Then once I found the biggest child, I had to build the heap and fix the heap. This was difficult due to the fact that I have to move so many elements around the array just to make sure the array meets the heap requirements.

Lastly, I implemented *quick sort* which was pretty simple to implement once you figure out how to find the pivot element in the partition.

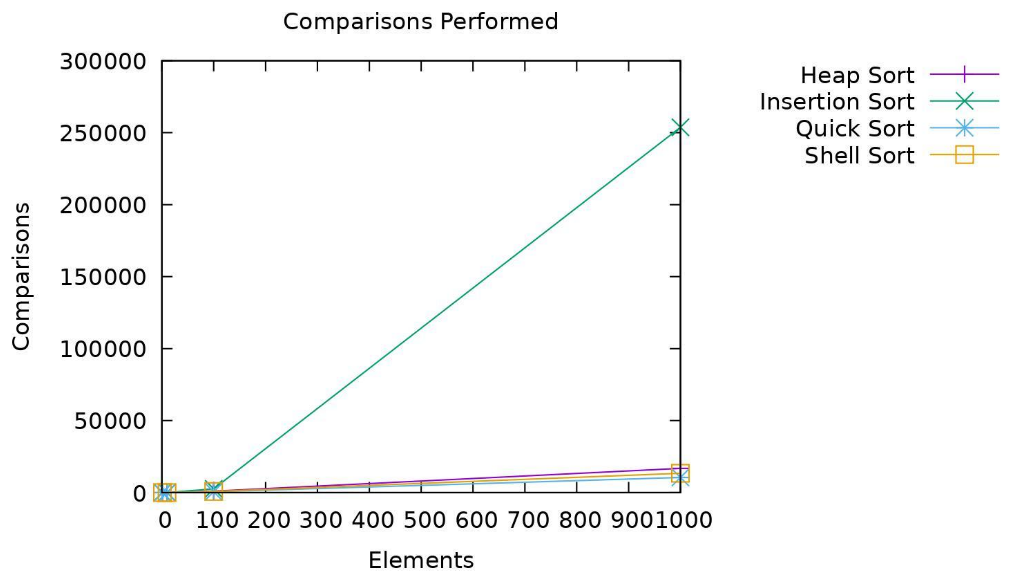
Once I implemented all the algorithms, I then built a command-line based test harness in C. The test harness took in command options that specified which algorithm to use, how many elements are to be sorted in the array, what random seed was to be used, and how many elements you would want to be displayed in the final output. By default, the program would want the user to specify which or all of the sorting algorithms to be used, display the numbers of moves and compare the algorithm(s) used, and display the sorted array in the end.

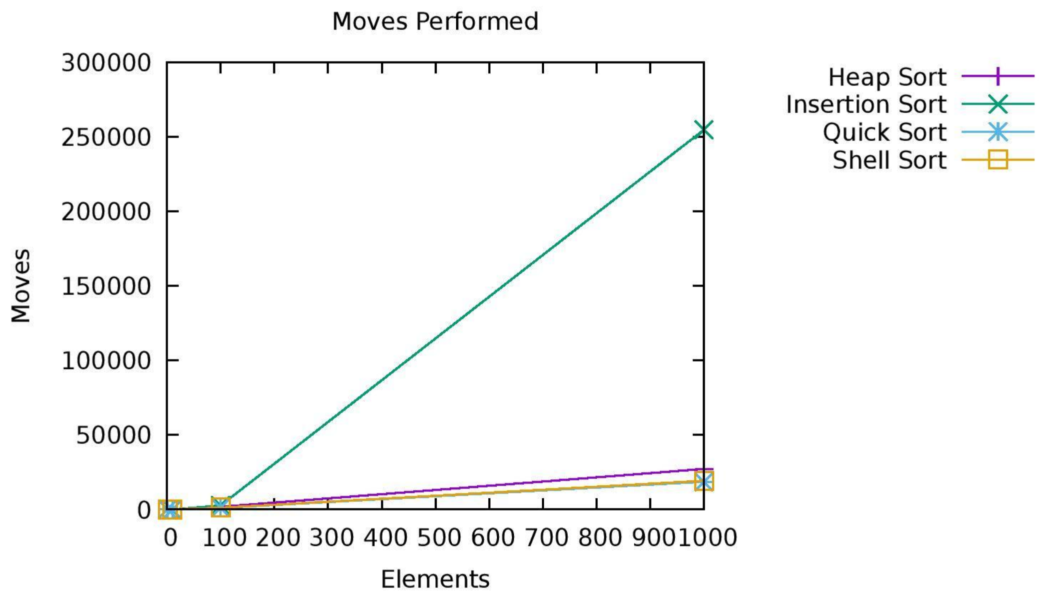
From that test harness, I then built a bash file that would use GNUplot in order to parse and graph all the outputs from running the sorting binary file.

Results

Graphs

Below are the graphs of moves and comparisons each algorithm took for n = 1 to n = 1000 elements in the array.





Based on the graphs, we can see that shell sort and quicksort seem to be the most efficient algorithms to use overall. However, it seems like that when n < 100, then all algorithms are equally as effective.

Insertion Sort

Insertion sort is a sorting algorithm that has the best case scenario of $O(n)$while having $O(n^2)$ as its average or worst case scenario. When looking at both the comparison and moves graphs, we can definitely see why this is true. We can see that there is a strong direct relationship between *n* and the amount of comparisons and moves it uses. As *n* gets bigger, then the number of comparisons and moves becomes significantly bigger. However, if the number of elements is less than 100, then it seems to be as effective as all the other algorithms.

Another reason as to why the algorithm is not as efficient as the other algorithms is due to the fact that the algorithm has to iterate through each element in the given array. If an algorithm has to iterate through each element in the array then it will take a long time to finish. If we have an array of 100 million elements, then insertion sort will take a long time to be completed.

Shell Sort

Shell sort is a sorting algorithm that has an average complexity of $O(n \cdot log(n))$ and a worst complexity of $O(n^2)$. However, since we are using Knuth's gap sequence, our Big-O has been reduced to $O(n^{\frac{3}{2}})$. In short, shell sort has the same algorithm as insertion sort; however, it uses gap sequences rather than comparing the neighboring elements. By using gap sequence, the algorithm is able to sort an unordered array in a much faster time.

If we look at the moves graph, we can see that it roughly used the same amount of moves as quick sort which is considered the fastest sorting algorithm in this assignment. However, if we take a look at the comparison graph, we can see that it calculates more comparisons than quick sort. The main reason this happens is due to gap sequence. In this assignment we use Knuth's gap sequence which helped reduce the complexity time but still we need to compare a lot of elements.

Heap Sort

Heap sort is a sorting algorithm that has an overall (best, worst, and average) complexity of $O(n \cdot log(n))$. If we look at the graphs, we can see that the algorithm is pretty fast however, it is the slowest compared to quick sort and insertion sort. The main reason as to why this algorithm uses more moves and compares is due to the fact that it needs to build a heap and fix a heap.

The process for both building a heap and fixing a heap involves a lot of element swapping. In this assignment, a single swap adds 3 moves to the overall counter. In addition, in order to swap elements, we most likely have to compare elements as well which is why heap sort becomes slower than shell or quick sort and which is why we have more compares and moves than either algorithm.

Quick Sort

Quick Sort is a sorting algorithm that has an average and best time complexity of $O(n \cdot log(n))$. However, it has a worst time complexity is $O(n^2)$. If we look at both graphs, we can see that it has the least amount of compares and moves compared to the rest of the sorting algorithms.

If we look closely at the graphs, we can see that it is not completely a straight line. The reason for this is due to the nature of how pivot elements are chosen. If the "wrong" pivot element is chosen, then more comparisons and moves are needed. If the optimal pivot point is chosen then the algorithm will sort pretty quickly.

So what's the deal with pivot elements? In the quick sort algorithm, a pivot element is chosen and then smaller elements are put to the left of the pivot while bigger elements are put to the right of the pivot. So if we choose a pivot element with all bigger elements on the left side of the pivot then the algorithm has to keep swapping elements until all the bigger elements are on the right side of the pivot.

Conclusion

In conclusion, we find that quicksort is the most effective and efficient sorting algorithm compared to insertion, heap, and shell sort. However, quick sort suffers a fatal flaw of having a $O(n^2)$ complexity if the pivot element is poorly chosen.