Assignment 3: Getting Your Affairs in Order

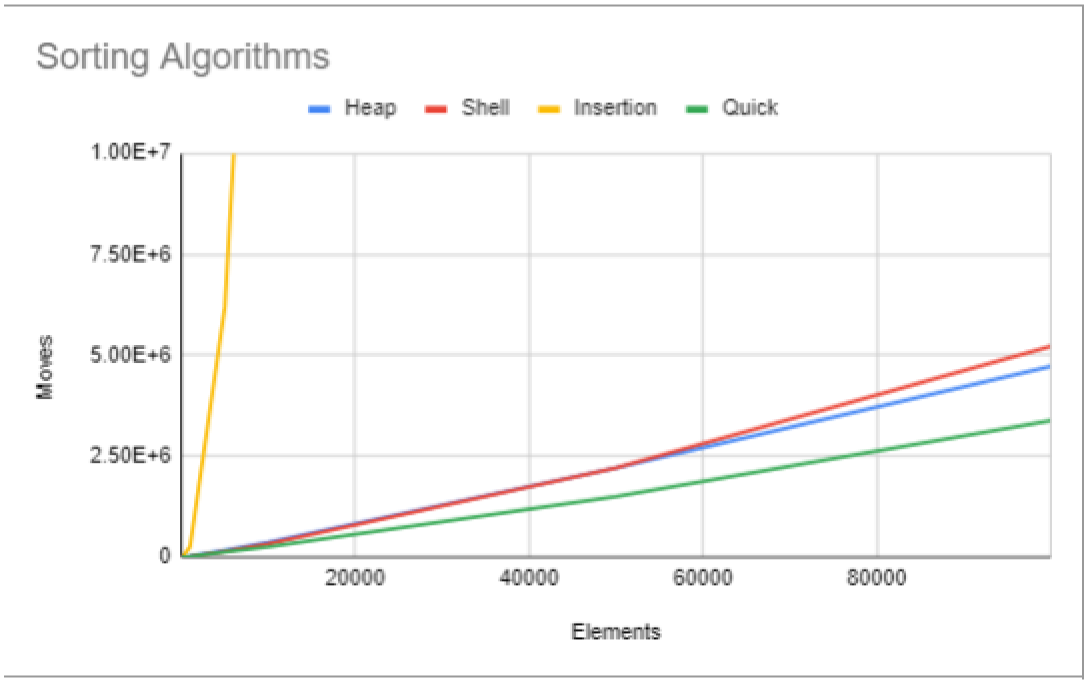
WRITEUP.pdf

Nathan Ong

This program is a library containing 4 different sorting algorithms that put an unordered array in order using different methods. These sorting algorithms are Shell Sort, Insertion Sort, Heap Sort, and Quick Sort.

- Shell Sort
  - Shell Sort uses a gapping sequence to sort through the elements of an array and can be thought of as a much faster Insertion Sort. The gapping sequence defines a gap between the elements using two formulas, one for the maximum gap value, and one to obtain each gap value less than it so the algorithm can keep on proceeding to a much more focused state. Each time the array is gone through, the gap gets smaller and smaller until finally, it reaches a gap of 1. The algorithm's behavior is then similar to Insertion Sort.
- Insertion Sort
  - Insertion Sort considers elements individually one at a time and places them in their correct position by comparing them to the element before them and swapping them if necessary. This specific implementation uses a temporary storage variable to make swapping elements easier and also giving a point of reference for the algorithm's while loop.
- Heap Sort
  - Heap Sort uses a heap data structure, which is usually implemented as a kind of specialized binary tree, to sort through an array. The heap is built with a single parent node at the top, with the rest of the array's elements beneath it, branching off into layers of children. With this implementation, the heap is built as a max heap, meaning the largest element of the array is the parent node from which the heap originates. After the largest element is found it is "locked" in position so to speak at the end of the array (or just before it depending on the number of elements sorted through). The heap is then "fixed" to make a new max heap and the process repeats until the array is fully sorted.
- Quick Sort
  - Quicksort divides the array into two separate subarrays and sorts the subarrays in the same fashion. It can be thought of as a recursive algorithm that divides the array and subarrays until they cannot be divided anymore. It is the most efficient sorting algorithm out of any of the 4.

Listed below are a graph and a table showing the number of moves it takes each sorting algorithm to sort an array of *n* elements. With this data, we can come up with a time complexity for each of these sorting algorithms.



Elements	Heap	Shell	Insertion	Quick
10	93	71	43	51
100	1755	1383	2743	1053
1000	27225	21323	254771	18642
5000	171552	147334	6216116	126234
10000	372558	333890	24901708	256734
50000	2212407	2214172	625490377	1497948
100000	4723917	5218105	2496734843	3382473

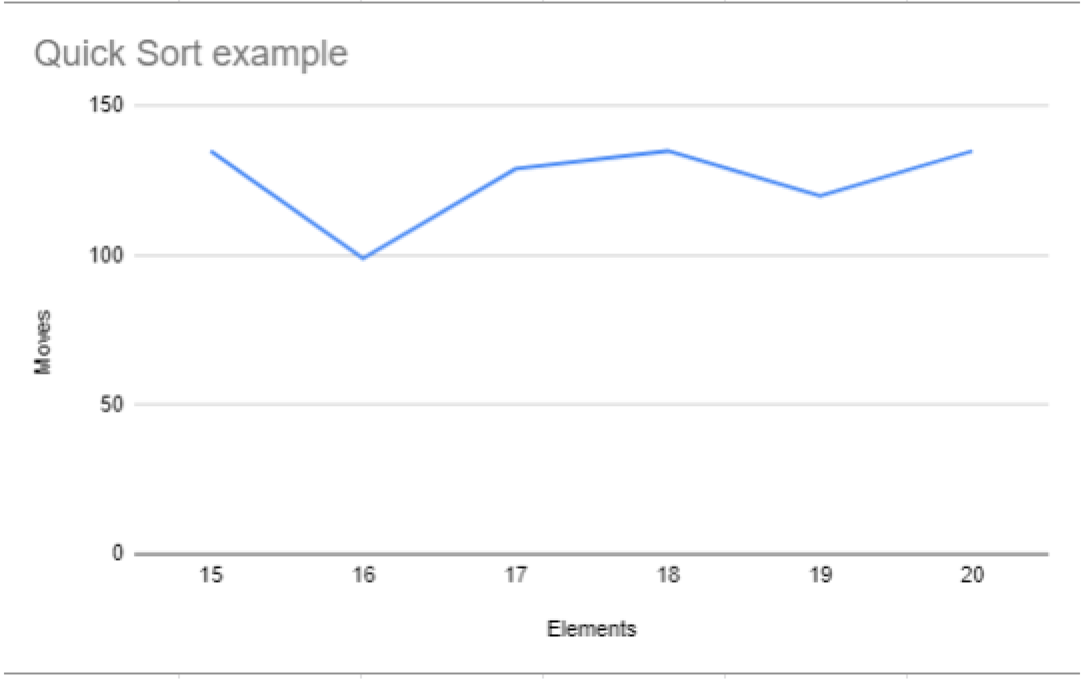
The time complexity of these sorting algorithms is listed in big O notation and this notation describes the rate at which the run time of the algorithm changes with the number of elements in the array they are assigned to sort. The average time complexities of the 4 sorting algorithms that are present in this assignment are listed below.

- Insertion Sort: $O(n^2)$
- Shell Sort: $O(n^{5/3})$
- Quick Sort: O(log(n))
- Heap Sort: O(n log(n))

The graph and chart above show that insertion sort gets more and more inefficient the large the number of elements are. At the early stages, it's on par, or even faster than some or the rest of the sorting algorithms, but later it gets more and more inefficient, dwarfing the others in terms of the number of moves required to sort the given array.

For most of the data, the Heap and Shell Sorts are shown to be pretty close to each other at the beginning, but around an array size of 60000 elements is when they start to diverge, with the Heap Sort becoming faster and more efficient than the Shell Sort. Because of this, we can assume as well as observe that as the number of elements increases, the distance between the two algorithm's moves increases.

Lastly, Quick Sort is shown to be evidently faster than any of the other methods. The number of moves is substantially less than the other three functions and as a result, the algorithm is shown to be much more efficient than any of the others. Along with being more efficient than the other methods, the graph up close also has a much more different behavior than the other data points.



As shown above, in the graph of the Quick Sort algorithm, the graph shows that the algorithm's behavior and speed depend on the number of elements being sorted. It seems like it varies as to how fast it goes and at what difference compared to sorting a smaller array, but if we observe the difference between sorting a 15, 16, and 17-element array, it seems like the algorithm favors even numbers in terms of the number of moves that it takes to sort through the array.

What I learned from this assignment and as a result of experimenting with the sorting algorithms is that with smaller arrays, the choice of algorithm would not really affect sorting the array except for fractional losses of run time. However, with larger databases and data structures, the choice of algorithm can greatly save time or waste time depending on what is being used. In addition, conceptually I learned about the big O notation and the time complexities of the different algorithms.