PA5 Report

Introduction:

In this Programming Assignment, 4 different sorting algorithms were implemented. The first sorting algorithm that was implemented is Bubble Sort. Bubble sort repeatedly steps through the array, while repeatedly comparing adjacent elements, if these elements are in the wrong order, the algorithm will swap them. This process repeats itself until no more swaps are required which would indicate that no more sorting is required. The second sorting algorithm that was implemented is heap sort. Heap sort inserts all the elements in the given array into a min-heap after which it would repeatedly extract the minimum value from the min-heap. The third sorting algorithm that was implemented is merge sort. Merge sort recursively divides the arrays into 2 subarrays, sorts these subarrays, and merges all these subarrays to get one sorted array. The final sorting algorithm that was implemented is Quick sort. Quick sort recursively creates 3 partitions using a pivotal index (chosen randomly), the leftmost partition has all the values less than the value at the pivotal index, the rightmost partition has all the values greater than the value at the pivotal index, and finally the middle partition has all the values equal to the value at the pivotal index. After this the same process is repeated for the right and left partitions.

Theoretical Analysis:

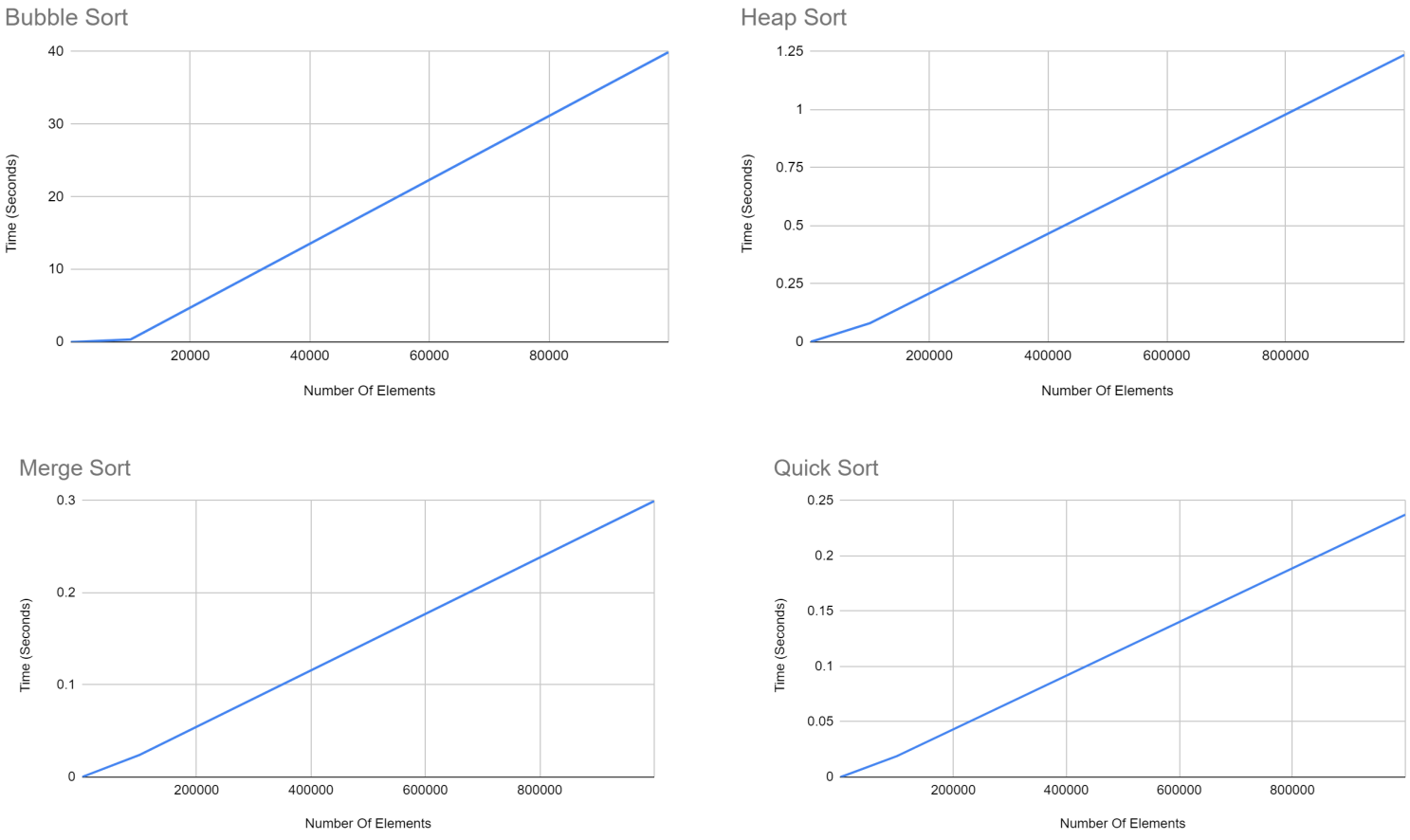
All the 4 different algorithms have different runtimes. Bubble sort's runtimes are as follows: best case runtime is O(n^2), the average runtime is O(n^2), and the worst case runtime is O(n^2). Heap sort's runtimes are as follows: best case runtime is O(n log n), the average runtime is O(n log n), and the worst case runtime is O(n log n). Merge sort's runtimes are as follows: best case runtime is O(n), the average runtime is O(n log n), and the worst case runtime is O(n log n). Quick sort's runtimes are as follows: best case runtime is O(n log n), the average runtime is O(n log n), and the worst case runtime is O(n^2). Bubble sort has the worst runtime complexity as it is O(n^2) for all cases. This is because bubble sort's basic function iteratively goes through the entire list, compares elements that are next to each other, and switches them if they are not in order. Heap sort's runtime is consistent among all scenarios as the elements would just be needed to min heap and be extracted from the heap in order to be sorted. The insertion and deletion in a min-heap can be done in O(log n) therefore resulting in O(n log n) for the entire array to be sorted. Merge sort's divide and conquer strategy effectively handles the sorting process by recursively splitting the array into smaller parts, sorting each half separately, and then merging them back together. This division and merging operation is responsible for the consistent O(n log n) runtime in both average and worst scenarios. Quick sort's best case and average runtime complexity is O(n log n) which is enabled by its divide and conquer strategy. But its worst case runtime complexity is O(n^2) and this can especially happen when the pivot index chosen is the smallest or largest element which would result in unbalanced partitions. Therefore, it can be said

that each sorting algorithm has its own advantages and disadvantages which are dependent on the type of input and input size.

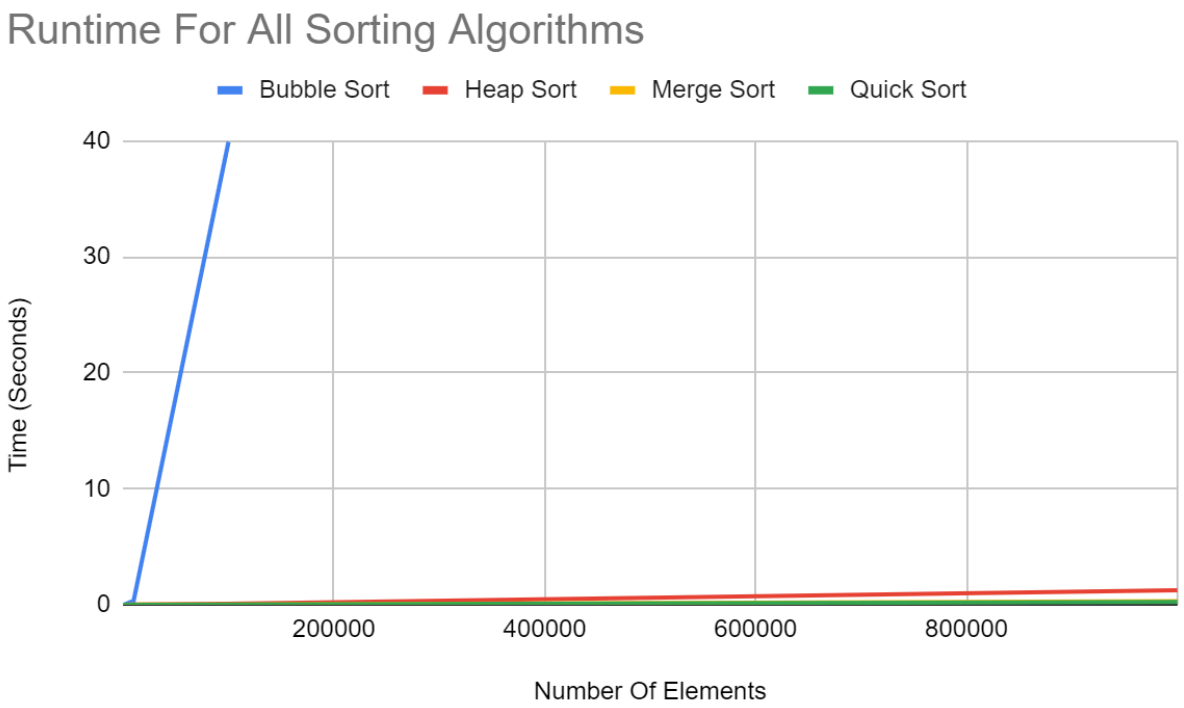
Experimental Setup:

This experiment was conducted in order to find the runtimes for different sorting algorithms when the number of elements in the list is changed. The size of the array was decided based on the suggestion on the PA5 manual and these sizes were - 10, 100, 1000, 10000, 100000, 10000000. The numbers in these arrays were randomly generated using the rand() function and these numbers ranged between 1 and 1000000. The same array with the randomly generated numbers was then passed to the sorting algorithms. The times for sorting the arrays with each of the sizes mentioned above were recorded for each sorting algorithm and saved into a csv file in order to be plotted.

Experimental Results:



Individual graphs for the runtimes of each sorting algorithm



Graph for all sorting algorithms

Table with the data				
Input Size	Bubble Sort	Heap Sort	Merge Sort	Quick Sort
10	9.84E-07	6.94E-06	1.93E-06	8.60E-07
100	3.49E-05	5.14E-05	1.63E-05	1.09E-05
1000	0.0038646	0.0010789	0.000289661	0.000191899
10000	0.330469	0.00837727	0.00251469	0.00173954
100000	39.915	0.0810496	0.0239027	0.0188811
1000000	-	1.23547	0.299533	0.2373

Analysis:

The best sorting algorithm based on the results would be quick sort. It was the best sorting algorithm regardless of the size of the array that was passed. Though its worst case runtime complexity is O(n^2), given that the proper pivot index is selected, it tends to outperform all the other sorting algorithms in all input sizes and relatively performs close to its average and best case runtime complexity of O(n log n). The second best sorting algorithm was merge sort which

performs very well with all sizes. It is on a very close level with quick sort and performs substantially better than Bubble sort and Heap Sort. This is as expected due to the fact that it has a consistent O(n log n) performance in both average and worst scenarios. It also has a low runtime when dealing with a very large number of elements making it very scalable. The next best sorting algorithm was heap sort. Although bubble sort performed better than heap sort initially, starting from 10000 elements heap sort performed far better. The runtime complexity of heap sort of O(n log n) can be seen with the steady increase in execution time as the array size increases. Though it is slower than Merge and Quick Sort for smaller array sizes, it works rather well across all input sizes. The worst sorting algorithm was bubble sort which especially when the input size is large. When dealing with small input sizes, the duration is relatively small. However, the execution time gets worse when the input size gets to 10,000 and larger, taking over 39 seconds for an array with a size of 100,000. This is consistent with Bubble Sort's O(n^2) complexity. This experiment shows that choosing the right algorithm based on the requirements is very important as each algorithm is beneficial for different sizes of inputs and other cases such as stability. In conclusion, quick sort performed the best for all input sizes (likely due to good selection of pivotal indices) and hence might be the best option for sorting for any given requirements.