**CA284 - Final Project Report**

**Introduction:**

This report is to analyse the performance of each sort used within the CA284 sorting project. I chose Selection sort, Quicksort, Bubble sort and Heap sort. I chose selection sort due to the fact that it could be used as a base to analyse my other more advanced sorting algorithms. I chose Quick sort due to its ability to sort arrays of large integers in an efficient manner. I chose Bubble sort because it is a case-specific algorithm and I wanted to see its performance on arrays other than partially sorted arrays. I chose Heap sort as I wanted to compare its performance to the less optimised Selection Sort.

**Datasets:**

I used two different datasets to analyse the performance of my sorting algorithms. I used a regular random generated array and a reversed array to do this analysis. I picked these two algorithms as a randomly generated array is a base level to analyse the performance of sorting algorithms and can be used as a comparison to study the performance of said algorithms on other datasets. I used a reverse array as I was interested to see if an already sorted but reversed array would cause any changes to the performance of these algorithms.

**Algorithm Performance:**

I will talk about Selection Sort first this performance was optimal in an array of size 10 this makes sense in relation to Selection sort as it is generally regarded as an effective sorting algorithm when using a sample size of 10-20. It’s performance as array size increased suffered as can be seen by checking the individual test data in my testdata.txt file. As the array came closer to the upper limit of my tests the performance of my selection sort truly suffered as can be seen by the time it took to run an array of size 1000 and 2500. This sort didn’t change much when the dataset changed from a randomly generated array to a reversed sorted array. This can be seen as generally the reversed compared to a normal array only increased the time taken to run the sort never increased the time taken by over 3 seconds. This is due to the fact that Selection sort is a comparison based algorithm so it still carries out the same amount of comparisons whether the array is reversed or not.

Quick Sort performance as the sample size of my array grew was not affected all that much; this is due to the fact that Quicksort is generally regarded as a sorting algorithm that works well on large sample sizes. The performance of this algorithm didn’t change much when using a reverse array instead of a normal generated randomly. This can be seen in my testdata.txt file as the time taken to run the sort on reversed arrays never takes a second longer than the normal array in the tests I conducted. The sorting algorithm performed best on the smallest sample size given of a normal randomly generated array but in comparison to all my other sorting algorithms, it worked the best out of all on sorting large sample sizes.

Bubble sort performance as the sample size of my array size grew was greatly affected. This is due to the fact that bubble sort is a sorting algorithm that is very case-specific and most useful when sorting partially sorted arrays. Bubble sort was my, by far, worst affected sorting algorithm as sample size increased taking 100 seconds to sort a normally generated array of size 2500 integers. Bubble sort worked faster on reversed arrays in comparison to the normally generated random arrays, this is due to the fact that bubble sort works by repetitively swapping adjacent elements if they are in the wrong order. Therefore reverse arrays cause bubble sort to be more efficient using reversed arrays as they are already sorted then reversed meaning it requires fewer swaps. Bubble sorts optimal conditions were of small sample sizes of an already sorted then reversed array.

Heap sort performance as the sample size of the array size grew was hardly affected. This is due to the fact that Heap sort is another sorting algorithm that has a good level of performance on large sample sizes just like Quicksort. Using my two types of datasets didn’t affect my sorting algorithms performance much but generally, the time taken to sort the array if reversed took longer than a regular normally generated array. Heap sorts performance when comparing it to selection sort, another comparison based algorithm is interesting as its performance on large sample size is noticeably more efficient. The optimal conditions for Heap sort were a normal randomly generated array of small size in my tests this is a sample size of 10.

**Negatives:**

There are several aspects of my program which could’ve been greatly improved upon. The fact that I’m carrying out all my sorts in a single program greatly limits its performance as it needs to generate four arrays within the individual program. Another aspect which could be improved upon is my timer as it does not actually work correctly the timer starts and stops correctly but does not reset between the algorithms causing test results when testing all programs at once to be a combined time rather than an individual time for each algorithm. Also using arrays without defining them as global variables means their memory is on the stack creating a limit of generating 100,000 integers in a single array.

**Conclusion and Future Work:**

In conclusion, I would summarise that my implementation of algorithms is actually quite effective but several errors within my program including the timer and a single file for all algorithms are a disappointing result. Overall, this is my first large project in programming so overall I am satisfied with my results but if I was to continue this work I would definitely fix my timer and make it reset between programs and separate the file into four separate programs rather than using a switch.