Ailbhe Byrne – 19424402

**Analysing the Performance of Sorting Algorithms in C**

1. **Introduction**

In this report, I will discuss the performance of several sorting algorithms in the programming language C. The first algorithm I chose was selection sort. This works by finding the smallest element in the unsorted part of the array and swapping it to the start. The next algorithm I chose was heap sort. Heap sort works by making a max heap of the array and then swapping the largest element to the end. The last sorting algorithm I chose was radix sort. This works by sorting the numbers on each digit in the number, from least significant. I chose these three because they all have different time complexities in big O notation and they all work very differently.

1. **Datasets**

While testing these algorithms, I needed different types of datasets. I used my random number generator, see random\_ints.c in the src folder, to generate a random array of numbers and then kept a copy of them in random order, partially sorted order, sorted order and reverse sorted order. I picked these different sort types to better test and understand my algorithms, to see if they performed differently on any of them. I had 4 types of datasets, which were in different order, and had 10 sizes in each. When timing my algorithms, I ran them on 40 different datasets (except for selection sort which I ran on 28). The sizes of the datasets increased by different amounts, to better understand the sorting algorithm’s performance.

1. **Algorithm Performance**

By using these datasets on all the algorithms, I was able to analyse each one’s performance. Selection sort was very quick at sorting small array sizes but as the size grew it became much slower, see sel\_sort\_time.txt in the test folder for examples of this. It took roughly 40 seconds to sort 200,000 numbers which is extremely slow for a sorting algorithm and when I tried to sort 1,000,000 numbers, I had to stop it as it was taking too long. On small array sizes (10-200) it was quickest on sorted datasets. I optimised selection sort by adding a check swap to only swap if the number is not already in the right place, see line 61 in src/selection\_sort.c. This helped to make it quicker, especially on sorted data. Random data took the longest to sort on these small datasets. As the datasets increased in size (100,000 and 200,000), random data was sorted the quickest, although sorted and partially sorted were close behind, and reverse sorted data was sorted the slowest. This is because for each new element the algorithm comes across it must assign it to the smallest. Selection sort is the slowest of the three algorithms.

For 10 numbers, selection sort and heap sort took about the same amount of time, refer to heap\_sort\_time.txt in the test folder. As the datasets grew, heap sort took more time but the time increased much slower that it did for selection sort. It took about half the time for 100 and 200 numbers. By the time it reached 200,000 numbers it had only slowed down to about 0.05 seconds which is 800 times faster than selection sort on the same amount. I was easily able to sort 5,000,000 numbers using heap sort. Heap sort took about the same time, on numbers up to 2000, to sort random data as partially sorted data. After this it was clear that it took the longest time to sort random data. It was also close between reverse sorted and sorted data, but it was quickest on sorted data, although reverse sorted was still sorted much quicker than random data. For large numbers (2,000,000 and 5,000,000), it was about twice as fast on sorted data than it was for random data, see heap\_sort\_time.txt. This is because there are less swaps required on sorted data.

Radix sort performed much worse on arrays of size 10, it took about 10 times longer than selection sort and heap sort. This can be seen in radix\_sort\_time.txt in the test folder. At 200 numbers it surpasses selection sort but is still slower than heap sort. Radix sort is best for sorting large arrays of numbers and we can see that it is much quicker at sorting these as it reaches 100,000 numbers. At 200,000 numbers, it is sorting at about 0.03 seconds. This is about 1300 times faster than selection sort and about twice as fast as heap sort on the same amount of numbers. It sorts 5,000,000 numbers with ease about 3 times faster than heap sort. Between the different sorted array types, the only real difference in time that can be seen is among the small array sizes. Radix sort takes the longest to sort small arrays in random order. It is much quicker at sorting reverse sorted and sorted arrays at this size. As the arrays get larger, the time difference lessens and doesn’t appear to follow any pattern. This is all present in radix\_sort\_time.txt.

1. **Negatives**

While completing this project some things did go wrong. I tried to optimise heap sort by removing the math header. Instead of using the floor function, see line 77 in src/heap\_sort.c, I worked it out manually and then tested it. Unfortunately, it didn’t make any difference, so I kept the math header. Another thing that isn’t quite right is that my implementation of radix sort doesn’t sort negative numbers. I added negative numbers to my datasets, which stopped radix sort from working. I didn’t have time to try to fix it so I removed the negative numbers from the datasets.

1. **Conclusion and Future Work**

In conclusion, selection sort is the fastest sorting algorithm for very small array sizes i.e. of size 10. It is quicker than both heap sort and radix sort. Heap sort is only best for sorting small arrays of size 100/200. As the arrays get larger than this, radix sort is much quicker than both. If I had more time, I would run the different sorting algorithms on many datasets and then get the average for each array size and data type. I think this would make analysing them more accurate. I would also spend more time optimising the algorithms, for example I would try to change selection sort to find and swap the smallest and largest element at the same time and I would try to get radix sort to sort negative numbers.