**C project report**

**Student number: 19378783**

**Introduction**

I chose selection sort, radix sort and quick-sort as my sorting algorithms. I picked selection sort because although it is a slow algorithm to use with large datasets, it’s an easy algorithm to understand and implement. (selection\_sort.c)

I picked radix sort as I have never coded this algorithm, so this was a challenge for me. I had to do a lot of research online, but I like the way radix sort sorts the dataset. It is also a very fast sorting algorithm.  (radix\_sort.c)

I chose quick-sort as my third and final one because it uses the divide and conquer method to sort datasets. I thought it would be interesting to see how these three different sorting algorithms compare. (quick\_sort.c)

**The types of datasets I generated**

**Random (number\_generator.c)**

I used the number generator program that I built to create a random sorted data set with numbers between 0-100. These numbers are in no particular order.

**Half ascending and half descending (half\_and\_half.c)**

I made a program that sorts one half of a data set in ascending order (line 23), and the other half of the data set in descending order (line 33). I did this so that I could test my sorting algorithms on a data set that is semi sorted already, and I thought it would be interesting to see how it deals with the half ascending, half descending aspects.

**Reversed array (reverse\_sort.c)**

I created a reversed version of the radix sort algorithm. I used this to reverse the random numbers generated by the number generator program. I chose to reverse radix sort as it is one of the fastest algorithms.

**Sorted array (radix\_sort.c)**

I used my fastest algorithm (radix sort) to sort the randomly generated numbers, and then tested my sorting algorithms on this already sorted array.

**Half sorted array (radix\_half\_sorted.c)**

I used radix sort to first half an array (line 64) and leave the second half unsorted. I created this to see how the sorting algorithms would deal with it.

**Algorithm performance**

Selection sort is the least efficient of the three. This is because of the way it sorts the data, by iterating through the entire array and finding the smallest element and swapping it to the front. (line 26, selection\_sort.c) Selection sort is fast when it comes to small arrays but is very inefficient for large arrays in particular. For each of my datasets when I was trying to sort 1 million integers, I had to force the selection sort program to stop. I didn’t do any larger than 1,000,000 as its clear that selection sort is not an efficient way to sort large arrays. This is due to selection sort being O(n^2).  Selection sort was most efficient sorting an already sorted array, and least efficient sorting a randomly sorted array.

Quicksort was inefficient in sorting the reverse array, the half ascending half descending and already sorted datasets. This is because the pivot is chosen as the last element. (line 30, quick\_sort.c) This makes the quick-sort algorithm behave quite similar to selection sort. If the pivot ends up close to the middle of the array, the algorithm is most efficient. Quick sort took longer than I expected sorting the random dataset, as it is known for being nearly 2 or 3 times faster than merge sort and heap sort. Quicksort was most efficient sorting a randomly sorted array.

After testing the three algorithms and gathering the test data, its clear that radix sort is the most efficient and fastest algorithm of the three. For each dataset I tested, radix sort had virtually the same run time for each set, close to 0 seconds. However, I had a lot of difficulty building this algorithm at the beginning. Sometimes the code would run seamlessly. Sometimes it would fail and if I tried to run it again a minute later it would work. I found out that this is due to my memory allocation request getting denied.

When I tried to use radix sort for 3 million integers, I encountered the segmentation fault error. This was due to the count sort algorithm used in radix sort. I had to use dynamic memory allocation (line 37, radix\_sort.c) for the output array as the program was trying to access a memory location not allocated to it.  It took a lot of research to figure out how to implement the correct dynamic memory allocation. However, once I figured it out, it started to make more sense to me.

I found that I could optimise my programs performance by doing the following:

For swapping elements in an array,

Firstly I had:

**swap = \*(a + i);**

**\*(a + i) = \*(a + position);**

**\*(a + position) = swap;**

Then changed it to (selection\_sort.c, line 29):

\*(a + i) += \*(a + position);

\*(a + position) = \*(a + i) - \*(a + position);

\*(a + i) -= \*(a + position);

By doing this, I avoided creating a new variable and allocating memory for that value. This method doesn’t work if the two values point to the same memory location.

**Negatives**

In this project, I think I could have improved by implementing a number generator that creates the data sets I wanted to test on. For example, I made my number generator that generates random numbers between 0-100. Then if I wanted to test on a reverse sorted array, I had to then reverse that random generated data set, and then use my algorithms to sort that data.

This was a great obstacle in testing my algorithms, especially with the half ascending half descending sorting program because it took a long time to sort the data sets into the half ascending half descending state. I tried to do this for 1 million integers and had to force it to stop as it took too long and is very inefficient. I still included this dataset even though I couldn’t test 1 million or any larger as I thought it was interesting to see how the sorting algorithms dealt with it.

**Conclusion & future work**

In conclusion, I realised that when sorting an array anywhere from 0 to 1000 integers, you can use any of the three algorithms that I tested, and they all take a similar amount of time. However, selection sort doesn’t handle large arrays very well due to its O(n^2) time complexity. Quick sort is O(nlogn) which makes it quite efficient but compared to radix sort which is O(d\*(n+b)), its performance isn’t as good.

If I had more time, I would test another sorting algorithm that is similar in time complexity to radix sort. I would also code my sorting algorithms in another language (such as python) and time them to see which language it takes longer to sort in.

It is clear from my testing of the sorting algorithms that any dataset from 10,000 integers or above, should be sorted with radix sort. This is by far the fastest and most efficient algorithm in sorting any type of dataset. It takes nearly the same amount of time to sort 0 and 1 million integers with radix sort.