# **CA284 Project Report**

**Ciara Lynch 19353946**

## **Introduction**

My report explores the complexity, running time and execution of three algorithms. I have chosen *selection sort, insertion sort* and *quicksort.* All of these algorithms display different run times and display different behaviour with different datasets.

* Selection Sort
* I chose this algorithm due to prior experience with it in python in previous modules and found it quite simple but inefficient.
* Insertion Sort
* I chose this due to the fact that it is a simple algorithm to code and is efficient with smaller arrays, however deeper into my project I found it was not efficient with larger arrays.
* Quicksort
* I have never worked with this before so I decided to research it. I also learned that it could partition elements into two sub arrays.

## **Datasets**

I tested my algorithms with two data sets *Half Sorted* and *Random*, I did this by building two programs half\_sort\_numbers.c and random\_numbers.c.

1. Half\_sort\_numbers.c takes user input from the command line and generates the dataset according to user input.
2. Random\_numbers.c. also takes input from the command line but the maximum number is defined by the input multiplied by two. Like so int val = rand() % (input \* 2);

The two data sets interact with each algorithm differently depending on the size of numbers generated, I found the execution time of sorting the numbers to be very interesting as the contrast in results was great, see **results.txt**.

## **Algorithm Performance**

* Selection Sort
* Overview:
* As I had dealt with selection sort before I was not expecting great performance results(matched its complexity), however I was interested to see how it would interact with the numbers. At first it would not pass five hundred thousand numbers until I added the dynamic memory allocation *malloc* which helped me allocate memory to my dataset int \*randnum = (int\*) malloc(dataset \* sizeof(int)); \*see code\* this fixed my problem straight away. Malloc allowed me to go up to one million integers and I decided to stop there as selection sort is an extremely slow sorting algorithm.
* Performance with **Half Sorted**
* The half sorted generator worked well with selection sort as mentioned before i wasn’t expecting it to sort things quickly but with the integers already half sorted it sped up the case a little for example when dealing with a million half sorted integers it took a minute quicker which is impressive with selection sort, see **results.txt.**
* Performance with **Random Numbers**
* I found when working with this dataset the numbers took quite some time to sort, during testing I found the sort times to be very long, I started with my largest value one million as I knew this would take so long compared to quicksort selection sort was atrocious displaying a sort time of twenty minutes.
* Insertion Sort
* Overview:
* Insertion sort also displayed a variance of different results, i discovered that it was faster than selection sort as expected but also displayed slower sorting times than expected. I also had problems with larger numbers until I added malloc code above, as seen in results.txt it sorted to one million numbers too.
* Performance with **Half Sorted**
* It worked better here than with random numbers, displaying faster sort times, as always I started with a million integers and worked my way down. Unfortunately I couldn't get it to pass one million integers.
* Performance with **Random Numbers**
* I was happy with the results of random numbers as it displayed good sort times and even better matched its complexity. Even when dealing with a million numbers it managed well.
* Quick Sort
* Overview:
* The results for quicksort interested me as I was expecting fast sort time, however not this fast, what also helped me build this algorithm was my work with it in CA117.
* Performance with **Half Sorted**
* Performance with half sorted was expected less than a minute with a larger amount of integers such as a million.
* Performance with **Random Numbers**
* After adding malloc I decided to sort up to two million as dealing with large numbers with this algorithm is lengthy as insertion sort and selection sort.

## **Negatives**

1. **Not taking numbers from command line in sorting algorithms**

* When writing my code I decided to store all numbers I generated from my datasets into a text file, I soon discovered that this was a problem. When you run my number generation you must ensure that the int dataset corresponds to the number you generated on the command line. I know this is not efficient however due to time constraints this was my only way to sort.

1. **Storing numbers in text files:**

* This was also not very efficient as it overruled the chance of using command line arguments, however it was a learning experience as now I can store information in a text file in C!.

## **Conclusion**

From my analysis of these three sorting algorithms I have discovered that all three have different behaviors when interacting with numbers. This is because they all have different execution times and complexities. You can view the differences in **results.txt.**

**Future work:**

* I now know that text files are not ideal to work with when dealing with so many programs, when i approach a project like this again i will not use them.
* I also learned how to implement a timer to my code which will be very useful if i decide to pursue future projects in C.

Overall I enjoyed this experience as I really expanded my knowledge on C as I felt prior to this it was not as good as my knowledge of other languages like python. See references.txt for any sources used.