**CA284 Project**

**Student Name:** David Weir

**Student ID:** 19433086

**Date:** 09/01/2021

**DCU Academic Integrity and Plagiarism Policy**

By signing this form or by submitting material for assessment online I confirm that I have read

and understood the DCU Academic Integrity and Plagiarism Policy (available at

<http://www.dcu.ie/registry/examinations/index.shtml>).

Signature: David Weir

Date: 10/01/2021

**Introduction**

In this report I intend to analyse the efficiency of 3 different sorting algorithms on a variety of data sets of varying sizes, to highlight how these algorithms perform under different circumstances, determine when they perform at their best and worst and compare their performances to each other. To complete this task, I chose three algorithms:

1. Merge Sort 🡪 An O(nlogn) algorithm that splits the array, sorts and then merges them.
2. Quick Sort 🡪 An O(nlogn) algorithm based on partitioning an array into smaller ones and sorting based on a pivot.
3. Selection Sort 🡪 A simple O(n^2) sorting algorithm that finds the minimum element and puts it at the beginning until the array is sorted

These 3 algorithms should provide a detailed comparison of sorting efficiencies. For the purpose of comparison my tests will also include qsort(), a C library function using a different implementation of quick sort, I did this to compare to my own version of the quick sort algorithm.

**Datasets**

The data sets used in this project come in the form of 4 different order types (see lines 7-12 and 65-91 in generator.c):

1. Random 🡪 an array of random integers e.g. 14342 28543 29514 28085 8126 19896 16315 23882 13194 18164
2. Sorted 🡪 a sorted array of random integers e.g. 7967 8437 9463 17450 17575 19865 20324 26425 28430 28706
3. Reverse Sorted 🡪 a sorted array that is reversed e.g. 29001 25909 25804 22602 19611 18119 17666 14074 12009 6294
4. Partially Sorted 🡪 an array that is half sorted e.g. 10475 10978 13893 17741 30462 11622 12345 24012 31328 7786

These datasets allow for rigorous testing of our algorithms allowing us to see the differences in performances depending on the type of array we are working with.

**Algorithm Performance**

Selection Sort:

On smaller arrays from 10–100,000 integers, selection sort performs well staying below 1 second to sort random arrays up to 10,000 integers and only 22 seconds to sort 100,000 integers. However, we begin to see its limitations as these arrays grow as it quickly falls behind the efficiency of the other 2 algorithms. From my tests it too 40 minutes to sort 1,000,000 integers, 2.845 hours to sort 2,000,000 and 15 hours to sort 5,000,000. Once selection sort hits 2,000,000 it becomes quite inefficient and by 5,000,000 it becomes completely impractical to use.

In general, there is minimal change in runtime when we use different types of datasets. Almost every time reversed data took the longest followed by partial, random and sorted. These differences of course become more obvious as our list gets bigger with a reversed array taking 20 minutes longer than on a random array of 2,000,000 numbers but our ultimately unimportant on smaller arrays where we would expect to use Selection Sort.

Selection Sort appears to perform best on numbers below 100,000 and performs worst on numbers over 2,000,000 becoming redundant at such large sizes. This is because Selection Sort has an O(n^2) time complexity making it inefficient on large lists.

Merge Sort:

Much more versatile than Selection Sort, Merge Sort is significantly faster, especially on large arrays. During my tests, Merge Sort’s runtime remains below 1 second until it reaches 2,000,000 integers. On random arrays Merge Sort takes 0.003000 seconds on 1000 numbers and 0.00.5000 seconds on 10,000, only really increasing when we hit 100,000 and 1,000,000 integers where it takes 0.0630000 and 0.661000 respectively. As our arrays increase in size so too does runtime, however, we only reach a 1.183 second runtime at 2,000,000 and 4.469 seconds at 5,000,000. As we can see these times remain relatively low even at large numbers making Merge Sort very efficient.

Merge Sort tends to vary in runtime more as we use different types of datasets with no obvious trend as there is for Selection Sort. These differences of course become more apparent as we sort more numbers, however, these times remain low, in some case being faster than a random array. For each dataset, runtime only hits 1 second at 2,000,000 integers but the other types are about 50% faster at 5,000,000 integers than on a random list.

Merge Sort performs well even on 5,000,000 integers long arrays however, we begin to see its limits at 100,000,000 integers taking over 2 minutes, still very fast but we can see it will start to take longer from here.

Quick Sort / qsort:

Quick Sort is by far the most efficient of the 3 algorithms, with my own implementation surpassing even the efficiency of C’s version, qsort(). To display just how fast Quick Sort is qsort() doesn’t reach a runtime of 1 second on random lists until 5,000,00 integers, while my implementation remains below 1 second until sorting arrays with 100,000,000 integers. Using my code my implementation doesn’t reach a recordable level of time until 100,000 integers. With qsort() needing 1.697 seconds for 5,000,000 and my own running for 0.093000 seconds.

These running times remain notably low even on our other 3 datasets. For all 3 reverse, sorted and partial Quick Sort is even more efficient, as seen below for 5,000,000 integers:

* Sorted = 0.054000 seconds
* Reverse = 0.020000 seconds
* Partial = 0.074000 seconds

This trend is seen on all arrays smaller than above too.

Quick Sort is more than capable of sorting even the largest of arrays in a very short space of time regardless of their order before sorting. Unfortunately, due to this and my own limitations on my machine it is hard to determine this algorithm’s best/worst cases as I cannot manage much larger arrays.

**Negatives**

Despite the rigorous testing undergone in this project several improvements could/should be improved. I believe that my selection sort code could be improved to increase its efficiency on larger arrays. My analysis of the sorting algorithms results could also be improved and made more accurate by using more sizes and testing each order type for each size multiple times on different arrays.

However, I believe most downfalls were discovered during testing. Merge Sort was changed to use dynamic memory allocation to handle larger arrays (see lines 10 and 11 in mergesort.c) and Quick Sort now uses both iteration and recursion on partitions as originally it could not sort over 100,000 integers before hitting maximum recursion depth (see lines 32-47 in quicksort.c).

**Conclusion / Future Work**

**Conclusion**

Ultimately, we find exactly what was expected regarding how these algorithms perform. As the arrays grow so does the algorithms runtime with no algorithms having issues on smaller arrays. However, we see selection sort becomes impractical on large arrays while the more complex Quick and Merge Sort algorithms remain quite efficient even when sorting arrays with 5,000,000 (5 million) integers. We also find that Quick Sort is the most efficient algorithm even on massive arrays, beating even C’s qsort, at below 0.307000 seconds for 5,000,000.

**Future Work**

If the time constraints of this project had been lifted, I would aim to improve 2 main areas:

1. I would first write more algorithms with different time complexities. This would allow for a more in-depth analysis on how these algorithms operate and perform. It would also improve my ability to compare their performances to provide a more accurate best/worst case scenario for each one
2. I would also run each algorithm more times on each dataset to provide more accurate results on performance and efficiency.