Introduction:

This weeks Practical asked us to search how an algorithm's speed changes near pathological cases. The main point of this practical was to show that when arrays of the same length are less sorted, the quick sort can sort them faster than if they were more sorted. So this would take the time complexity from O(n) to something closer to O(n log n).

Design/Implementation:

To make the quick sort work, I split up the array that was read in by getting the pivot point, find all values greater than the pivot, less than the pivot and equal to the pivot. These three sets of values were stored in temporary arrays, which were then recurssed by using the new array as the parameter. The same was split again, and then once all the values are set, the three temporary arrays are combined together and returned.

The generate method would take in the size of the array, initialize it, then loop through it's length filling it with random values. I also implemented a separate method which would again take in the size of the array as a parameter, then move through the length of the array filling it with values which are in order.

The messer method was used to keep track of the number of switches which occurred. I would call the method, and use the number of switches as the parameter which would tell the method the number of times I wanted it to change random places in the array for random values between 0 and 5000.

Hypothesis:

If the number of swaps increases, the time taken for the sort to be completed would decrease. This is because the sort will have to recurse through each value of the array until it can recreate the list which was already in place. Also in my case I am using the last element in the array, which in a sorted list means that every value is less than or equal to my pivot. The program would then split the values which were smaller than the original pivot creating a new temporary pivot in turn. This would continue on and on until the smallest value was left. This value would then be added to another array and returned recursevly until the original array had been recreated. So in this example the time Complexity would tend towards O(n*n) and not the optimum solution of O(n log n).

Where as the unsorted list would tend more towards O(n log n) depending on how many swaps were needed. This is due to the recursion which would allow the program to check for values which are both above, equal to and less than the pivot all in one operation, while the sorted list would need three attempts of the method to achieve the same result.

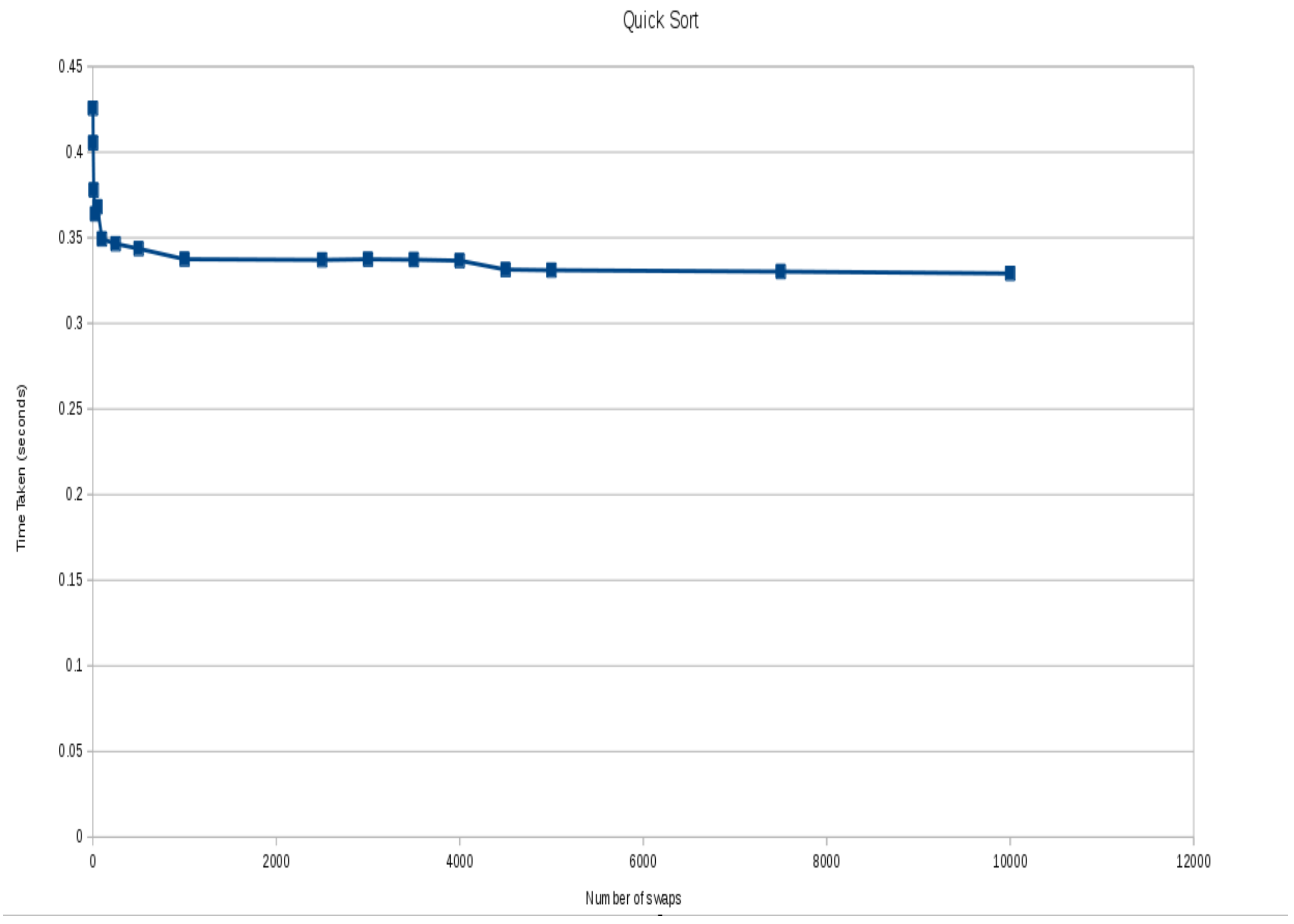
Quick Sort:

The quick sort initially was started with my generate method, which I used to create an array which was unsorted and had a random assortment of numbers that could be anything between 0 and 5000. For the test I used an array of size 10,000 as I felt this was suitably large to provide a good set of data.

So now that I had a sorted list of elements, I proceeded to change random values within the array using the messer method. This allowed me to keep track of the number of swaps which would occur, which then meant I could tell how 'unsorted' the array was, and as such map out the number of swaps corresponding to the time taken to process the array.

In the end I made a graph correlating the different times the sort took, against the number of swaps that were completed. To make it a fair test, I conducted ten of the same test to get an average result. Also each array had 10,000 characters, and the time taken for each of the ten tests was an average of five individual tests which were completed to get an average sort time for that array in that order.

Due to lack of optimisation, my program was rather slow. To the degree that there was not the largest gap between each of the test times. However, the correlation of the graph does indeed show that the Hypothesis is correct.



As can be seen in the graph. The fewer the number of swaps, the longer the time taken to sort the array. The larger the number of swaps decreases the length of time needed to sort the list. It then begins to tail off as the time complexity tends towards O(n log n) more and more. This is what was expected to be seen, and although the scale of the graph is not the ideal scenario to demonstrate this with, it does show the fact that the hypothesis was indeed correct.

Conclusion:

This practical went relatively well. The hypothesis that was put forward was proved, even with a very poorly optimised sort. I had initially intended to implement the more complex 'in-place' version of the quick sort, but after numerous attempts I could not get it to work and so went for the ordinary version of the quick sort. While it was useful as it worked, I feel the final outcome (the graph) could have been improved in it's quality if I had used an in-place version of the program. Regardless I feel that this practical has helped a lot. Although the code was not the main part of this aspect, I feel that this sort could be very useful in the future and I will most definitely be using it should I find a place where the use of a quick sort could be justified.