210017213

# CS2001 – W10 Complexity

## Overview

In this practical we were asked to run experiments to test the time the standard quicksort algorithm can sort lists of varying degrees of sortedness. We were to find different ways to measure sortedness and how changing the measure of sortedness of a list alters the amount of time taken to completely sort the list.

## Design

To run experiments on the quicksort algorithm I would generate lists with an increasing amount of sortedness each time, for value of sortedness I would run 100 repetitions to gain more accurate results as in the environment that these tests are run there are many external factors which can affect the time taken to complete the quicksort. So, using this high number of repetitions negates this factor and allows for more accurate results to be computed.

To generate the lists, I would take in value of sortedness and then would half this number so that the greatest number of swaps would be 500 and the lowest is 0 (completely sorted list). I would then swap two random indexes with each other however many times relevant to the sortedness to be generated. I found this to be enough to show the decreasing time of the algorithm's runtime. Any more than 500 and I found that it didn't make much difference to the results.

I decided not to use the number of swaps as one of my metrics for measuring sortedness because there is a chance by swapping two random elements you are making the list more sorted instead of less sorted. This is the reason why the two metrics that I have described below measure their value after the list has been produced.

The first metric I used to measure the sortedness of a list is to measure how many adjacent elements were out of order. I would traverse through the list of integers and if the number directly to the right of the element in the list was smaller, I would increment a total by one each time and that would be a measure of how sorted the list was. The more out of order pairs the more unsorted the list would be. A completely sorted list would have a value of zero.

The second metric I used was to sum the difference between where a value is the list and where it should be if the list was completely sorted. For example, if there was a list [2,3,1] then 2 would be out of place by 1, 3 would be out of place by 1 and 1 would be out of place by 2. This would mean this list would have a total value of 4. The higher the value calculated the lower the sortedness is. A list that is completely sorted would have a value of 0.

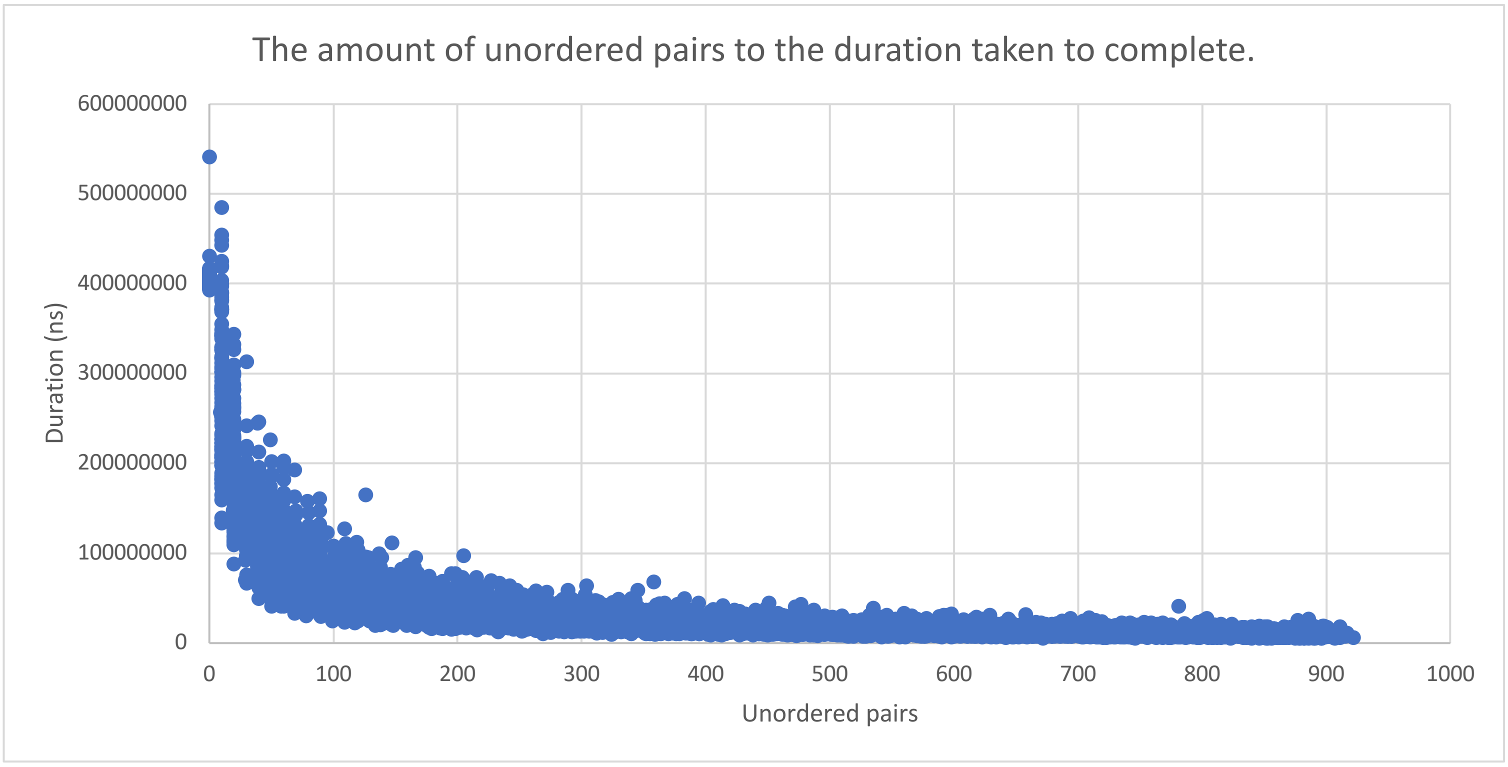

Figure 1 – Metric 1 – the number of pairs out of order to the duration.

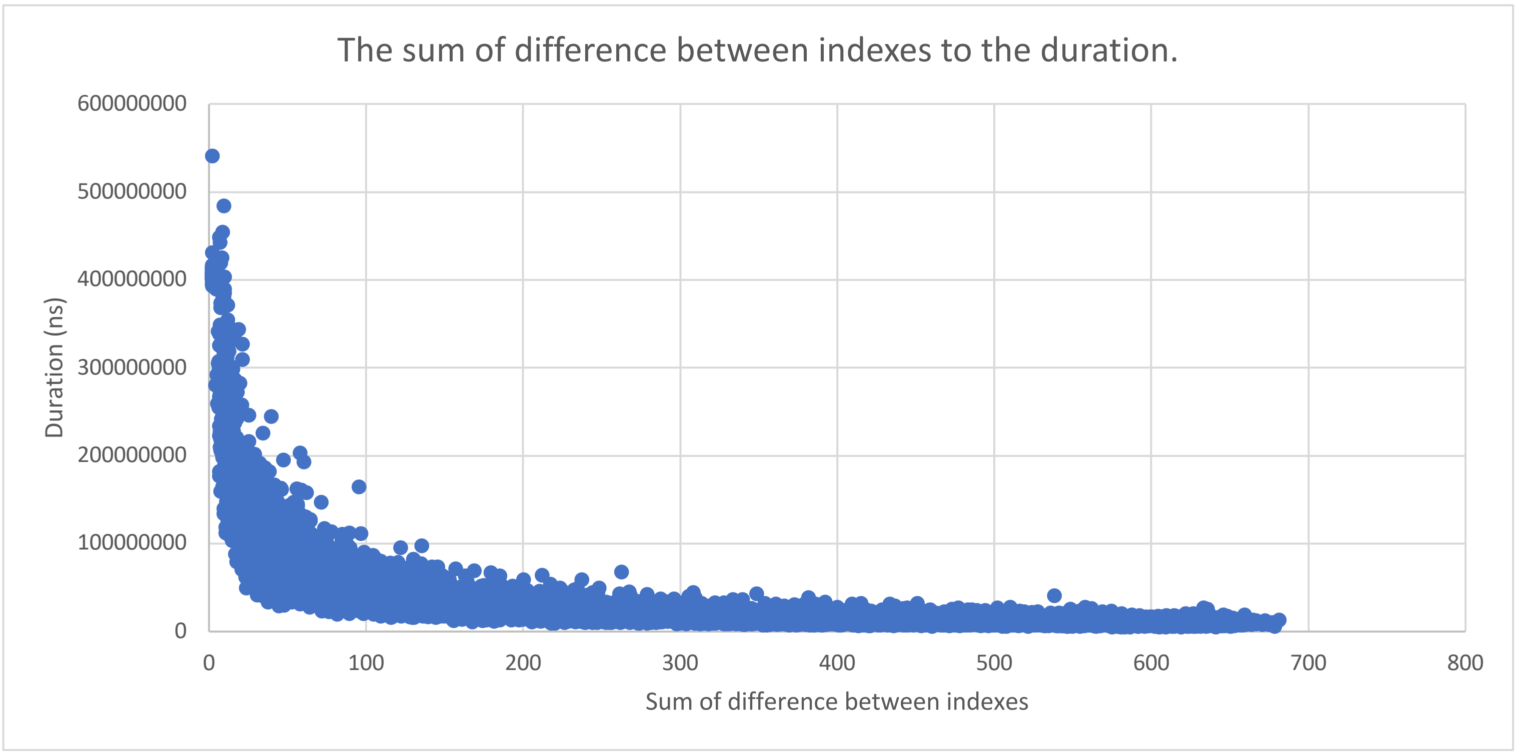

Figure 2 – Metric 2 – the sum of incorrect position of elements to duration.

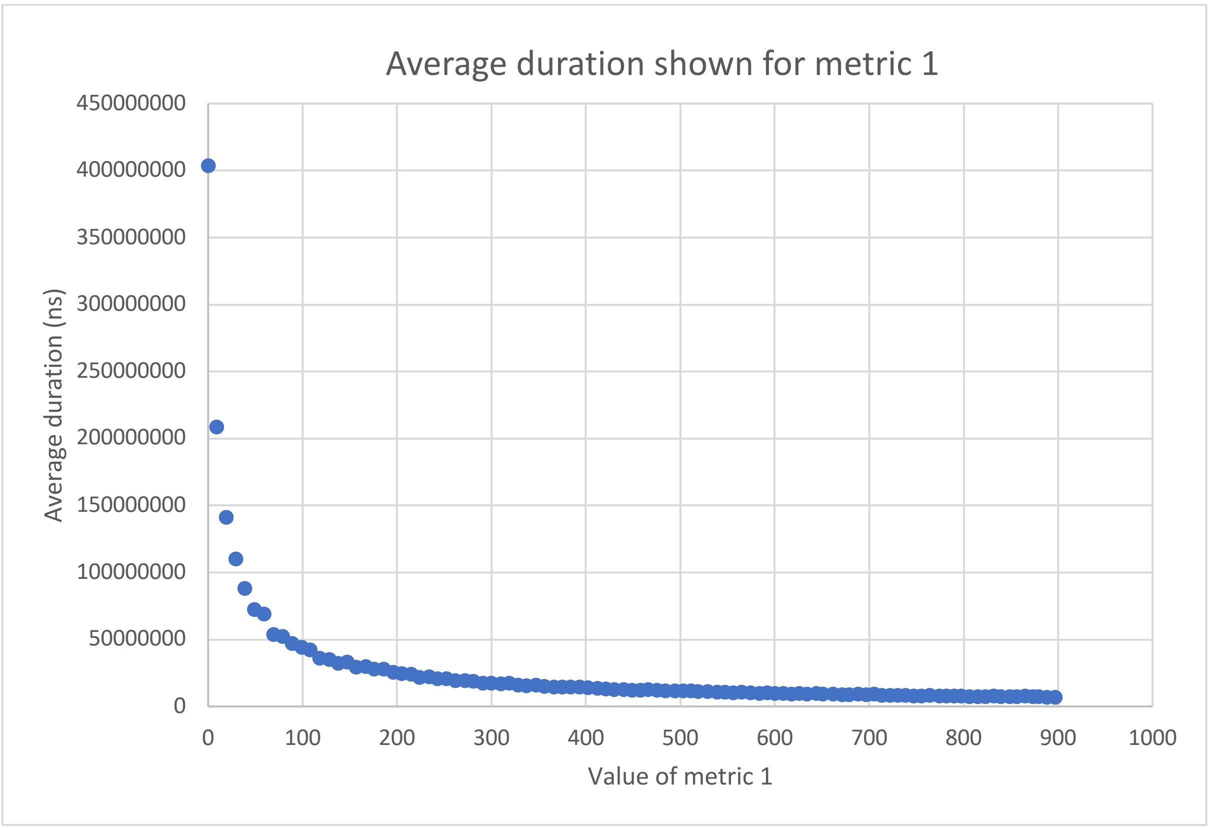

Figure 3.1 – The average values of Metric 1.

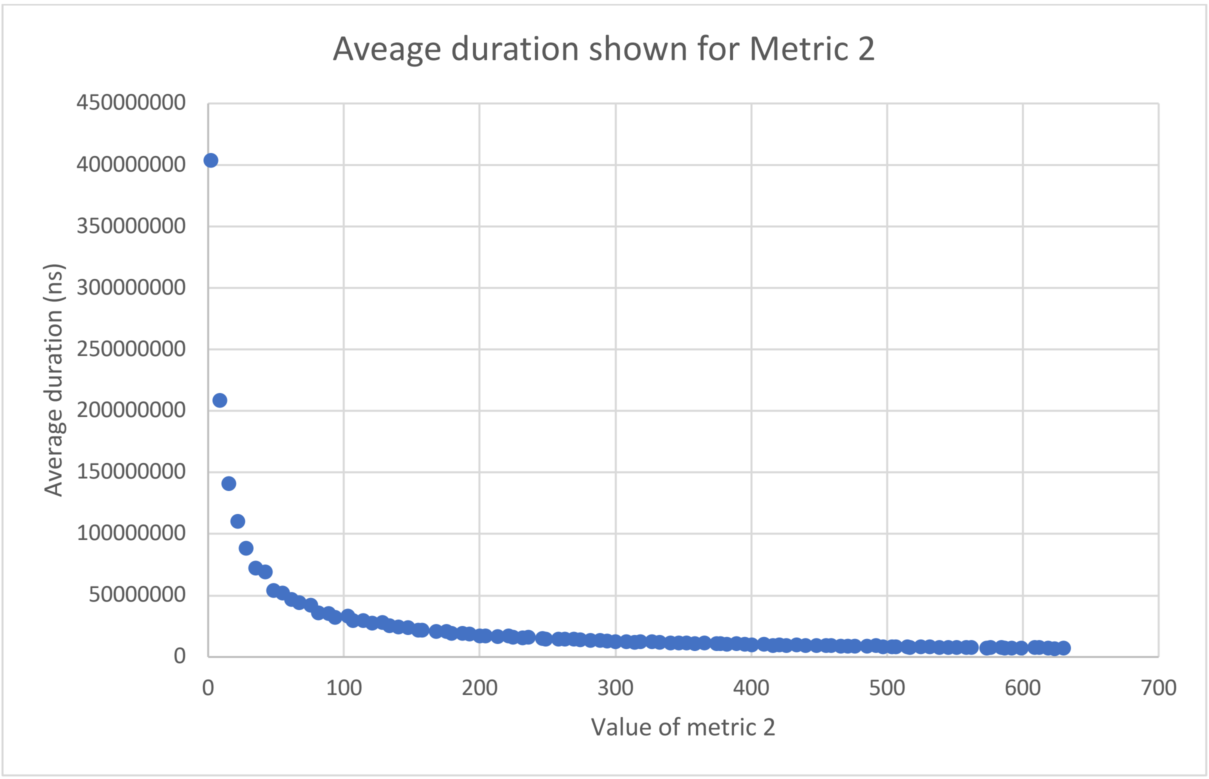

Figure 3.2 – The average values of Metric 2.

As shown in the graphs I have produced above (All the graphs have been produced using data that can be found in src/data directory) for both of my metrics I have used as the list became more sorted the faster the quicksort algorithm would run.

For both of my metrics it can be seen that a completely sorted list has by far the slowest run time even compared to a list that is only slightly more unsorted.

For both metrics it can be seen that at roughly 100-200 on the graph the time starts to tail off and there isn't much difference between 200 and 600 in value for example.

## Evaluation

I believe I have created experiments and used suitable to measure the amount degree of sortedness of a list and how varying this alters how long it takes for a list to be sorted.

## Conclusion

In conclusion I enjoyed this practical as it was a different style to previous that have been issued. I enjoyed analysing and coming up with experiments to come up with ideas about how different factors affect processes.