CS2001 Complexity 18/11/2020 19002143

Project Description

This report documents my work for the CS2001 W10 Practical, in which we explored how a search algorithm's performance is affected by its pathological cases.

Introduction

Quicksort

The quick sort is a divide and conquer algorithm which picks a pivot element, partitions around that element, and recursively sorts the partitions. The pivot theoretically should be close to the median value, however, in an unsorted array the median value could be any one of them. For the purposes of this project, the final element is used as the pivot.

Sortedness

I defined my own metric for sortedness based on the number of sorted pairs in the array. The sortedness, σ, is defined as

$$\sigma = \frac{Number\ of\ Sorted\ Pairs}{Total\ Number\ of\ Pairs}$$

A sorted pair is defined as one where the nth element is less than or equal to the n + 1th element and so is more a measure of how sorted in ascending order an array is. This results in $\sigma \in [0,1]$ with $\sigma = 1$ representing a fully sorted array and $\sigma = 0$ representing a fully unsorted array (sorted in reverse). This means an array of randomly selected elements would have $\sigma \approx 0.5$.

Pathological cases

A pathological case is one which causes sub-optimal performance in the algorithm. For this practical, I divided the sorting cases into five main classes based on their sortedness measure, namely

$$\sigma \approx \{0,\ 0.25, 0.5, 0.75, 1\}.$$

For each σ, a variety of cases were ran to determine which pathological case(s) affected the execution time of the array the most.

Sort Design

The Java code for the quick sort is divided into 2 packages. The sorter is in its own package with a test class. It runs a basic recursive quick sort using the last element as the pivot. Within the cases package, there are three classes. The Results class creates an object containing the name, sortedness and average execution time of a pathological case. The Pathological Cases class is used to run the sort on pathological cases and return Results objects. The Sort Runner calls the methods from the Pathological Cases class and writes the results to a CSV.

Analysis Design

The analysis is done using Jupyter Notebook which uses the results from the java program. The average execution time is then plotted dependant on the sort case for each approximate value of σ.

Methodology

After defining the sortedness, σ, I implemented a quick sorter with a test. In another package, I created a results object to hold the case names, sortedness, and execution times of each case as well as a pathological cases object which has the methods to run the cases and returns results. These are run from a sort runner class, which converts it all into the results CSV.

The table below details the standard case for $\sigma \approx \{0, 0.25, 0.5\}$. The rest of the (non-trivial) cases were implemented by changing the first and last elements.

Case	**Approximate σ**	**Description**
Fully unsorted	0	$n, n-1, n-2, \dots, 2, 1$
Quarter Sort	0.25	$1, n, n-1, n-2, 1, n-3, \dots, 1,3,2,1$
Half Sorted	0.5	$a, a_1 \dots, a_n : a, a_1 \dots, a_n \in [0, 1000]$
Zigzag	0.5	$n, 1, n, 1, \dots, n, 1$
Three Quarters Sorted	0.75	$n, 1, 1, 1, n-1, 1, \dots, 2, 1,1,1$
Sorted	1	$1, 2, 3, \dots, n$

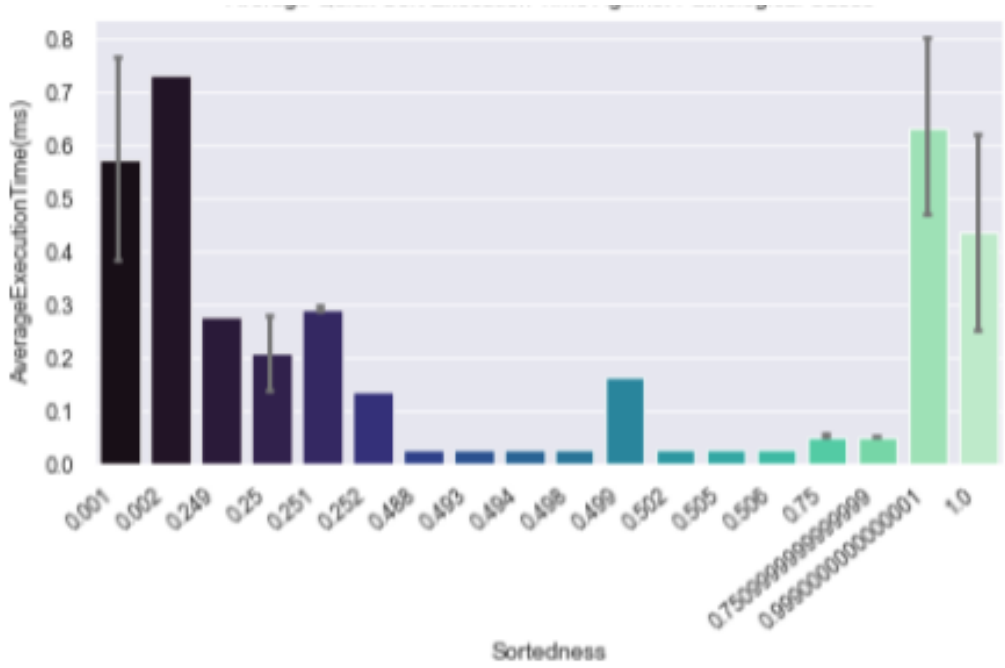
Each case is tested with an array of 1000 elements. The array is sorted and reinitialised 10 500 times after which the 250 highest and lowest run times are removed. The remaining 10 000 execution times are then averaged and returned in the results object along with the case name and initial measure of sortedness.

Once run, the results CSV is transformed into a pandas data frame. The seaborn library is then used to plot the execution time dependant on σ.

Results

The results for all the cases were as follows:

Figure 1: Average Quick Sort Execution Time Dependant on Sortedness



Besides for a spike near the middle, the graph shows that the run time is quickest where $\sigma \approx 0.5$ and slows as σ approaches 0 or 1. The rest of the cases were as follows:

Figure 2: Average Quick Sort Execution Time For Cases Where $\sigma \approx 0$

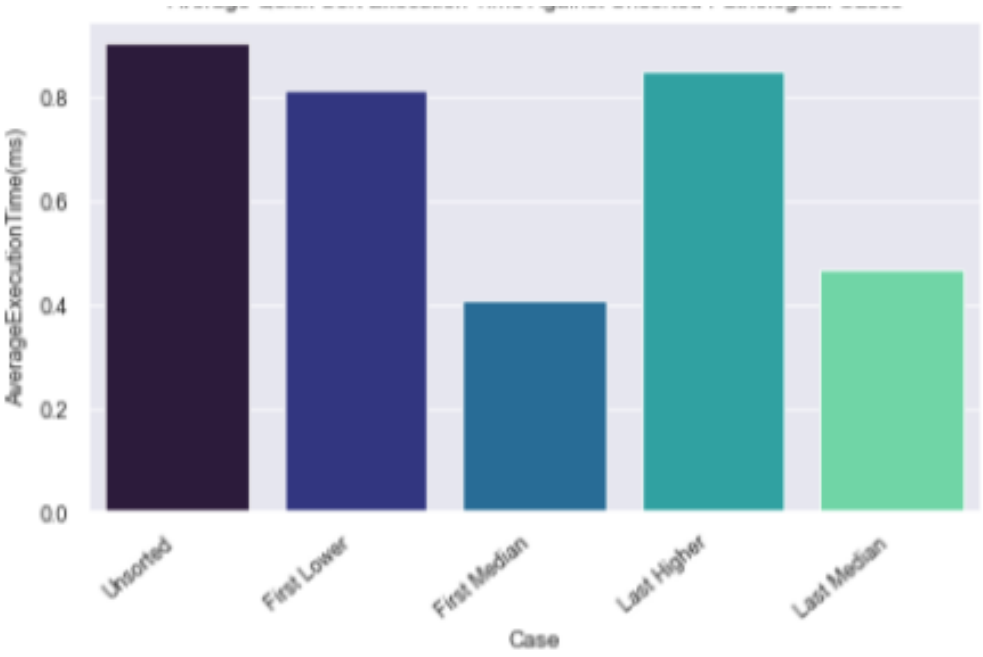


Figure 3: Average Quick Sort Execution Time For Cases Where $\sigma \approx 0.25$

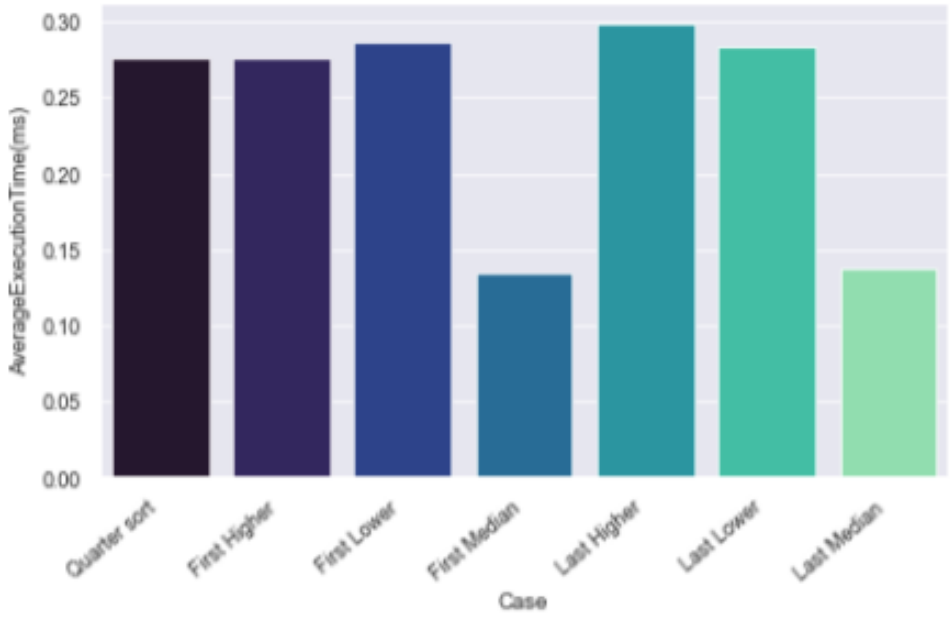


Figure 4: Average Quick Sort Execution Time For Cases Where $\sigma \approx 0.5$

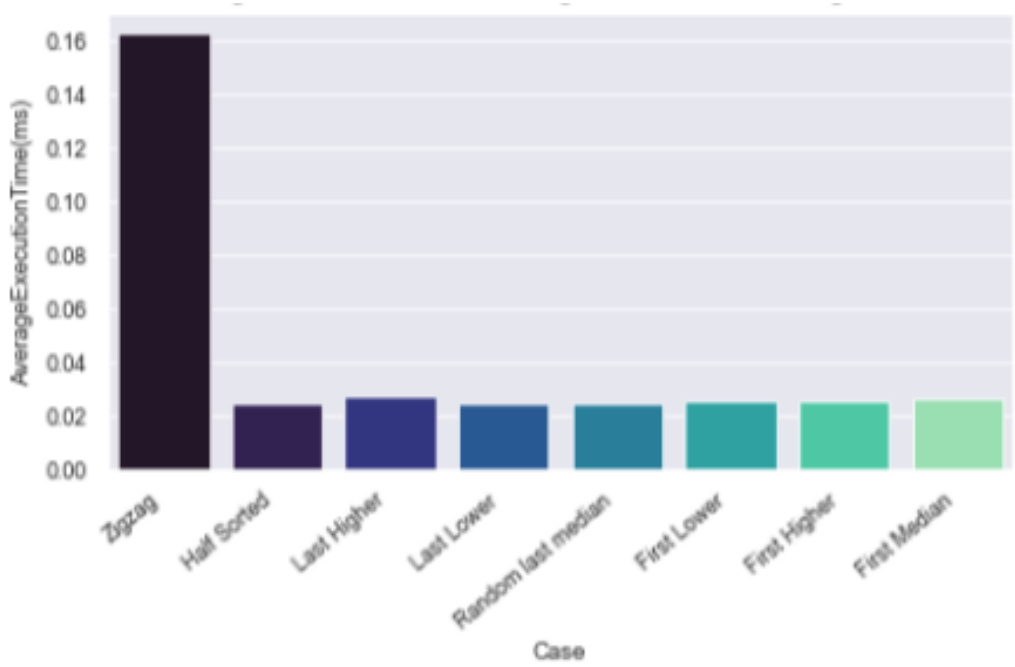


Figure 5: Average Quick Sort Execution Time For Cases Where $\sigma \approx 0.75$

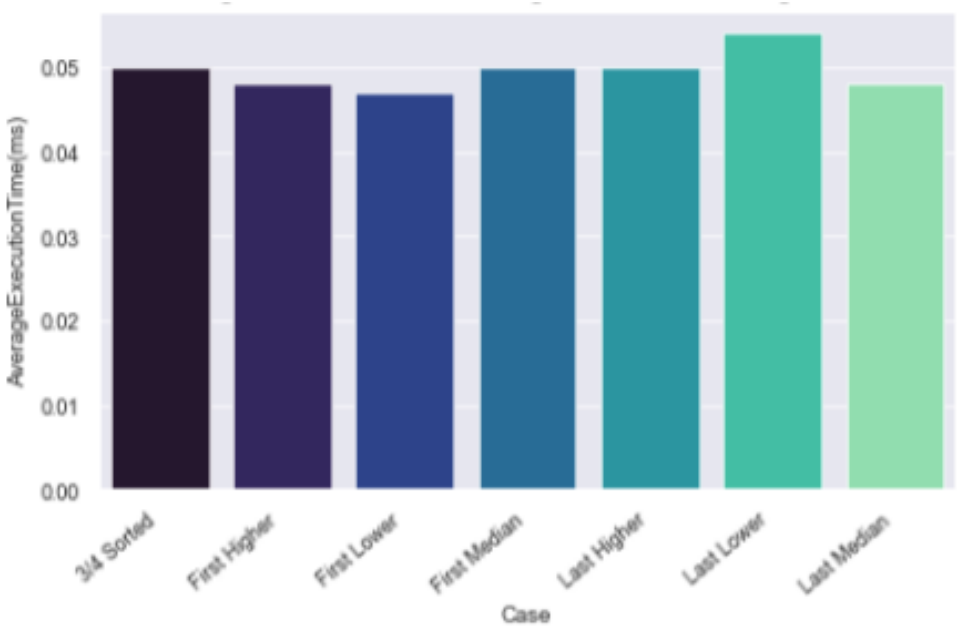
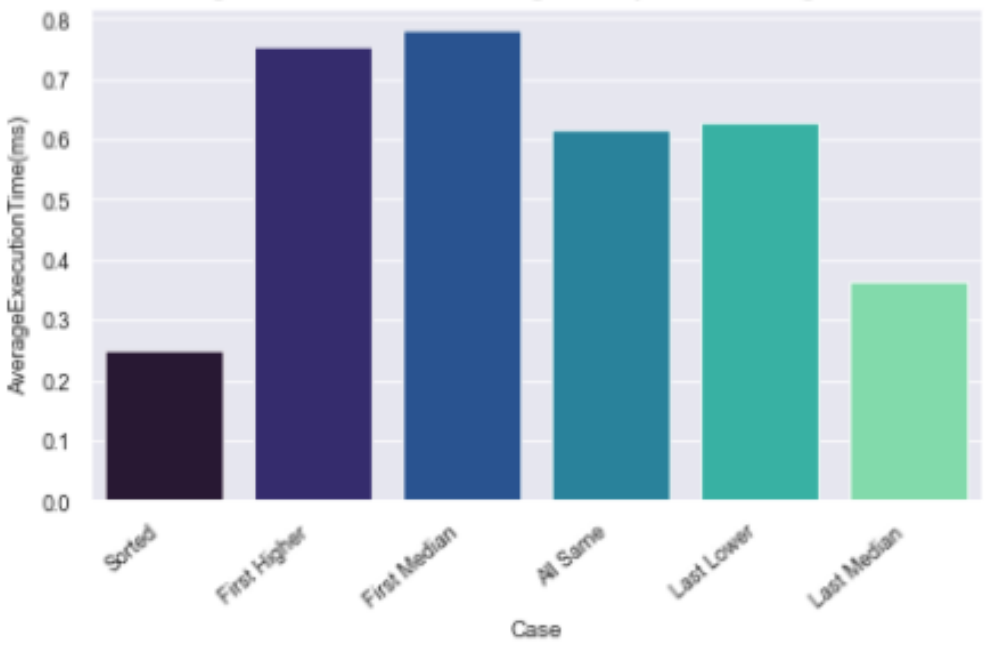


Figure 6: Average Quick Sort Execution Time For Cases Where $\sigma \approx 1$



Evaluation and Conclusion

Result Analysis

It is evident that the more absolutely sorted (where σ is closer to 0 or 1) the array is, the worse the quicksort performs. This is largely a result of the pivot being chosen as the final element. This causes the sort to perform the maximum number of divides as possible since the list of elements either greater than or less than the pivot will be empty. This is confirmed as the case in which the last value is the median generally performs best, no matter the sortedness.

Furthermore, it is apparent that certain cases which I thought would alter the performance of the sort did not have as much of an effect as I hypothesised. Actually, the cases that increased the run time were cases with sortedness closer to 0 or 1 depending on the case.

An interesting observation is that a zigzagged list (depicted in *figure 4)* is sorted significantly slower than other arrays of $\sigma \approx 0.5$. This could explain why the quarter sorted cases perform significantly worse than the three quarter sorted cases: they were created using a zigzag pattern. Another peculiarity is when $\sigma < 0.25$, setting the first value to the median improves execution time whereas with $\sigma \approx 1$ the execution time worsens by doing the same.

Conclusion

An ordinary quick sort performs worse on arrays that more absolutely sorted i.e. sorted in ascending or descending order. As a result of this, it is evident that a more representative measure of sortedness should have been used.

I would have liked to create arrays which span the possible sortedness range to further depict how the sortedness value affects the runtime. Despite this, I am confident in the results that the analysis produced.