**CS2001**

**Complexity**

**18/11/2020 190022143**

# Abstract

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. The program was required to run a tested quicksort on pathological sorting cases and XXX.

# 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

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 with σ = 1 representing a fully sorted array and σ = 0 representing a fully unsorted array (sorted in reverse). This means an array of randomly selected elements would have σ 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

.

For each , 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. The results CSV is transformed into a pandas data frame. The average execution time is then plotted dependant on the sort case for each value of .

# Methodology

After defining the sortedness, , 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 pathological cases were designed as follows:

|  |  |  |
| --- | --- | --- |
| **Case** | **Approximate** | **Description** |
| Fully unsorted | 0 |  |
| Unsorted First Lower | 0 |  |
| Unsorted First Higher | 0 |  |
| Unsorted First Median | 0 |  |
| Unsorted Last Higher | 0 |  |
| Unsorted Last Lower | 0 |  |
| Unsorted Last Median | 0 |  |
|  |  |  |
| Half Sorted | 0.5 |  |
| Half sorted first higher | 0.5 |  |
| Half sorted first lower | 0.5 |  |
| Half sorted first median | 0.5 |  |
| Half sorted last higher | 0.5 |  |
| Half sorted last lower | 0.5 |  |
| Half sorted last median | 0.5 |  |
| Zigzag | 0.5 |  |
|  |  |  |
| First Half Same | 0.75 |  |
| Last Half Same | 0.75 |  |
| Middle Half Same | 0.75 |  |
|  |  |  |
| Fully Sorted | 1 |  |
| Sorted first higher | 1 |  |
| Sorted last lower | 1 |  |
| All values same | 1 |  |
| Sorted last median | 1 |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

The medians are approximate and generally the value 499 is used.

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.

# Results

The results for all the cases were as follows:

Figure : Average Quick Sort Execution Time Dependant on Sortedness

Chart

Description automatically generated

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

Figure : Average Quick Sort Execution Time For Cases Where σ ≈ 0

Chart, bar chart

Description automatically generated

Figure : Average Quick Sort Execution Time For Cases Where σ ≈ 0.25

Chart, bar chart

Description automatically generated

Figure : Average Quick Sort Execution Time For Cases Where σ ≈ 0.5

Chart, bar chart

Description automatically generated

Figure : Average Quick Sort Execution Time For Cases Where σ ≈ 0.75

Chart

Description automatically generated

Figure : Average Quick Sort Execution Time For Cases Where σ ≈ 1

Chart, bar chart

Description automatically generated

## Evaluation and Conclusion

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