**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**

**https://link.springer.com/chapter/10.1007/BFb0038186**

There are a variety of ways in which one could define how sorted a list is. The general consensus is that Vladmir Estivill-Castro and Derick Wood give the best axiomatic definitions of measures of disorder.

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

This results in with σ = 1 representing a fully sorted array and σ = 0 representing a fully unsorted array (sorted in reverse).

**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 last higher | 0 |  |
| Unsorted last median | 0 |  |
|  |  |  |
| Random array | 0.5 |  |
| Random 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 |  |
|  |  |  |

# Results

The results were divided into 3 classes with respective subcases.

## Evaluation and Conclusion