ADS2_AE1_Report

2888936l

February 2025

Introduction

The code for the first assessed exercise is split among a couple of files. The file *TestSortingAlgs.java* is used to run the bulk of the testing. Within this file each algortim is tested by passing in the list of text files given in the exercise specification. All dataset text files can be found in a folder named "$AE1\_files$". *dutch.txt* can be tested in the *TestAlgsOnDatasets.java* as it is measured in milliseconds as opposed to nanoseconds. To plot the graph shown later in the report, run the file *SortingAlgsPlot.py*. Also included in the submission are the individual/average test times for each algorithm in the Excel file *AlgTimes*. The second part of the assessed exercise is implemented in the file *VideoSortingAlg.java* and contains a way to generate and output the top k most viewed videos.

Part 1

For the first part of the assessed exercise, the algorithms that were specified on Moodle were implemented in Java. After this was done, a Java program was written to test the time taken for each of the implemented algorithms (*TestSortingAlgs.java*). The implementation of this was done by creating a list of the text file names and iterating through it, calling each algorithm with the current file. The "TestSortingAlgorithms" method from Lab 1 was used to ensure that all algorithms returned correctly sorted lists. They were then timed using javas currentTimeMillis() and nanoTime() to return the total time taken to process and sort the file.

After the implementation was finished, each algortihm sorted every file provided in the specification 10 times, and then the averages were calculated. The results were then graphed on the logarithmic scale using the data as seen in the table below but with the exclusion of *bad.txt*. From the graph and table we can deduce a couple of things about the implemented algorithms. Firstly, we can see that as the number of integers to sort increases, both insertion and selection sort take significantly longer than the others. This can be attributed to the fact that these algorithms both have worst-case time complexity of $O(n^2)$ whereas the others have a time complexity of $O(nlogn)$. We can also see from the figures that for the larger files, the basic merge sort and bottom-up merge sort perform better. Furthermore, it can be seen that the three merge sort algorithms consistently sort the files faster than all other tested algorithms for 1,000 integers or more.

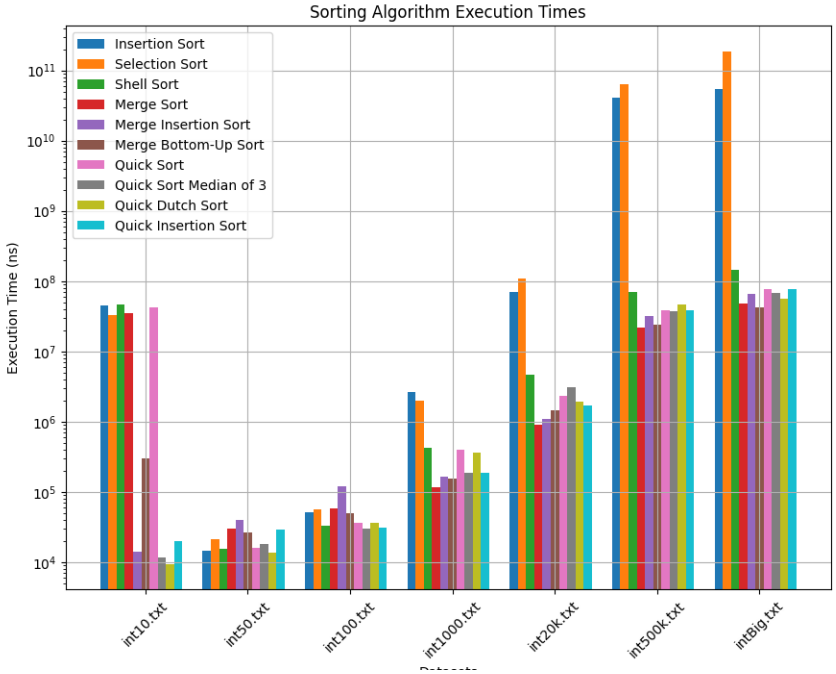

Figure 1: Average Sorting Times on Logarithmic Scale

Part 2

For the second part of the exercise, the task was to research and implement an algorithm to find the k most viewed videos under the assumption $k << n$ where n represents the total number of videos. The implementation of the task uses the data structure known as min-heap. Min-heap is a complete binary tree where the root of the tree is smaller than all descendant nodes and where their descendants are smaller than it, and so on. This data structure was chosen as the base of the algorithm as it is based on the concept of priority queues.

The implemented algorithm works by defining a Video class that contains two parameters, Video ID, and the videos Views. In the method to sort the videos, a new priority queue/ heap with size k is defined along with a way to compare video objects using their views. The algorithm then loops through each video in the list of videos that was passed into the method, if the size of the heap isn't equal to k, it adds the current video. However, if the queue is full it compares the current videos' views with the smallest view count currently in the heap. If the current video has more views, it replaces the video currently in the heap.

Once looped through all videos, a new list is created to store the elements of the heap, which will be the k most viewed videos. This list is then returned in reverse order to give the descending most viewed videos out of the original list.

This algorithm is both concise and efficient as it makes use of a heap. Operations performed on heaps have a time complexity of O(log k), where k is the number of elements in the heap. However, since the algorithm loops over the total number of videos (n), this means the total time complexity of the algorithm is $O(n \log n)$

The algorithm can be tested by running the file *VideoSortingAlg.java*. This file will create a random number of videos with the max being 50, each video will then be assigned a random amount of views under one hundred thousand and call the sorting algorithm with the list of videos and the desired number of videos to be returned (k)

Alg name	int10	int50	int100	int1000	int20k	int500k	intBig	bad
InsertionSort	45697460	14760	50730	2640750	70046570	40440981810	53382993510	114894450
SelectionSort	32972930	21460	55790	2006760	110175390	62467108340	1.85284E+11	185601040
ShellSort	45899570	15670	33300	421850	4614380	69045920	143982710	40161990
MergeSort	34723130	30310	59190	116250	901390	21888850	48596930	34110830
MergeInsertionSort	14040	40370	119650	167700	1078680	31701560	66356540	7361460
MergeBottomUp	302790	26720	49990	156570	1454160	24082980	42977450	3153690
QuickSort	42182860	16220	35960	399210	2299250	38963100	77065850	39851190
QuickSortMedian3	11850	18300	30560	185090	3138690	36916290	68480320	2241400
QuickDutch	9490	13910	36800	368270	1930840	46562430	55716010	4377730
QuickInsertionSort	20280	28880	30920	188510	1715030	38142760	76103710	13235440

Table 1: Average Times for the Algorithms in Nanoseconds

Alg name	dutch
InsertionSort	36206.5
SelectionSort	67464.4
ShellSort	63.9
MergeSort	76.4
MergeInsertionSort	40.1
MergeBottomUp	24
QuickSort	327.5
QuickSortMedian3	127.2
QuickDutch	42.6
QuickInsertionSort	216.2

Table 2: Average Times for the Algorithms against *dutch.txt* in Milliseconds