Algorithms and Data Structures

I. Background

In computer science, it is often imperative to explore the efficiencies of sorting algorithms, since any computing-oriented project, at its core, rely on sorting as a key elementary operation. This is akin to how, quite often, one relies on addition, or multiplication – since subtraction and division are 'derivatives' of these operations – as a key operation in a variety of Mathematical problems. Thus far in 'Algorithms and Data Structures 2', there was much exploration into some of these sorting methods. In addition, there were analyses performed to gain insight into related characteristics of these algorithms, i.e.: Worst-Case times via 'Big-O' notation.

This paper will explore some variations of the 'Quick-Sort' algorithm, compare them (alongside 'Merge-Sort' and 'Insertion-Sort'), and construct an algorithm to generate pathological sequences for the 'Median-Of-Three' variation of the 'Quick-Sort' algorithm. In laymen terms, 'pathological sequences' in the context of this task refers to an array (or list) of numbers, which will generate the worst-time for the considered algorithm.

For the purposes of this task, the algorithms were written in Java with the Eclipse IDE.

II. How to run the program

In the attached Eclipse project, there are two types of class files within the 'src' folder. The first, which consists of algorithms for the three variants of Quick-Sort and the second, which contains the Java program to generate pathological sequences for the Median-Of-Three Quick-Sort variant.

The former is straightforward to operate. Simply ensure that the text file of numbers is placed within the 'src' folder of the Eclipse project, which should be in the Eclipse workspace directory. Then navigate to the 'main' method and rename the text-file appropriately in the absolute path stored in the string variable 'fname'.

The latter contains the algorithm in the 'pathologicalInput' method, which takes two inputs: the desired length of the pathological array and the maximum possible integer for the array. Simply edit the parameters to the desired quantities in the 'main' method.

III. Part 1

For parts (a) – (c) see the uploaded Eclipse project for the commented class files. This section will be dedicated to exploring and justifying the implementation of (d). First and foremost, the differentiating attribute of the '3-way Quick-Sort' from the ordinary 'Quick-Sort' algorithm is the initial re-ordering of the elements about the pivot, which for this task was right-most selected. The 'threeWaySort' method starts indexing at the left-most point of the array and compares the value at that position with the pivot (which is the value at the end of the array). Depending on whether it is greater or less than the pivot, a shift is made. In the case of equality, the indexer simply increases by one. Eventually, we reach a point where the array is sorted into the following three consecutive sections: {less than pivot, equal to pivot, greater than pivot}.

At this point however, it is not necessarily true that the elements to the left and right of the pivot are themselves ordered. The algorithm does not consider comparisons between

elements themselves, only with the pivot. Therefore, at this point, a call is made to 'quickSortA', which proceeds to sort the algorithm as per the conventional 'Quick-Sort' algorithm – or as seen in the implementation of (a). This is recursively performed until the array is sorted in an ascending fashion.

IV. Part 2

After constructing the algorithms as per the requirements of the task, one must now analyse the efficiencies of them. The manner in which the analysis was performed was by measuring the time (by utilizing the 'System.nanoTime' module in Java) required to sort varying lengths of integer arrays. The integers were stored line-by-line in a text file. In the 'main' method, the text-file was read by a 'BufferedReader' and each line, which contained a single integer, was appended to an 'ArrayList'. Then an 'int[]' array was created of the same size and each element within the 'ArrayList' would then be appended to the newly created array, which underwent the various sorting algorithms.

Before analysing the efficiencies of the algorithms, a test was done across each text-file to ensure it was operating correctly; the 'TestSortingAlgorithms' method did just that. It traversed each element of the array, checking whether the given element was greater than the one succeeding it. If this occurred for *any* element, the method returned false, as it would imply the whole array was not sorted correctly. Otherwise, it returned true. The tests came out successful for each algorithm.

The text-files used for testing contained 10, 50, 100, 1000, 20 000, 500 000, and 1 000 000 integers, and the file would be named 'intX.txt', where X corresponded to the number of integers within the file. Additionally, 'dutch.txt' contained 500 000 numbers, some of which would repeat. For the purposes of this task, it would be futile to compare the results for every single file, i.e.: a trend is already observed when we compare 'int100.txt', 'int20k.txt', 'int500k.txt', and 'intBig.txt', and would be unnecessary to include 'int10.txt' or 'int50.txt'. Each text-file underwent 5 runs for each algorithm and an average was recorded. The following will graphically describe the results of these tests for the 6 algorithms: 'Quick-Sort', 'Quick-Sort-Insertion-Sort hybrid', 'Quick-Sort via median-of-three partitioning', and 'Quick-Sort via 3-way partitioning'.

Fig. 1.: Time measured for 1000 integer array in 10^5 *nanoseconds.*

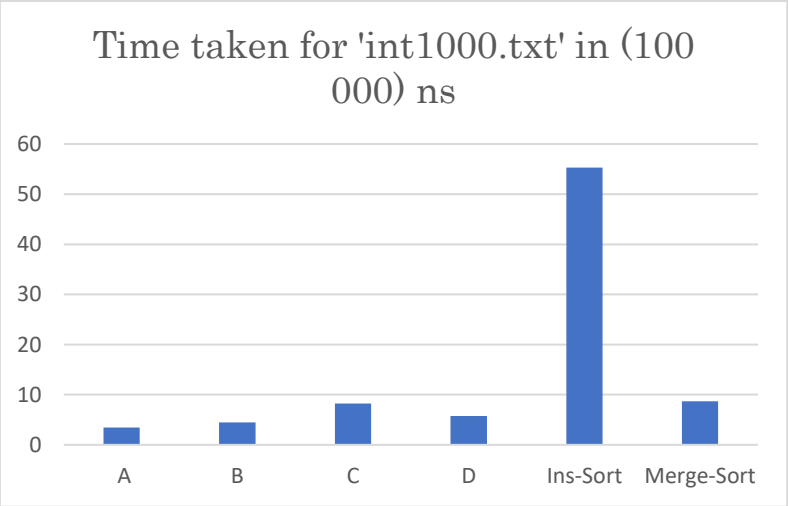


Fig. 2.: Time measured for 20 000 integer array in 10^6 *nanoseconds.*

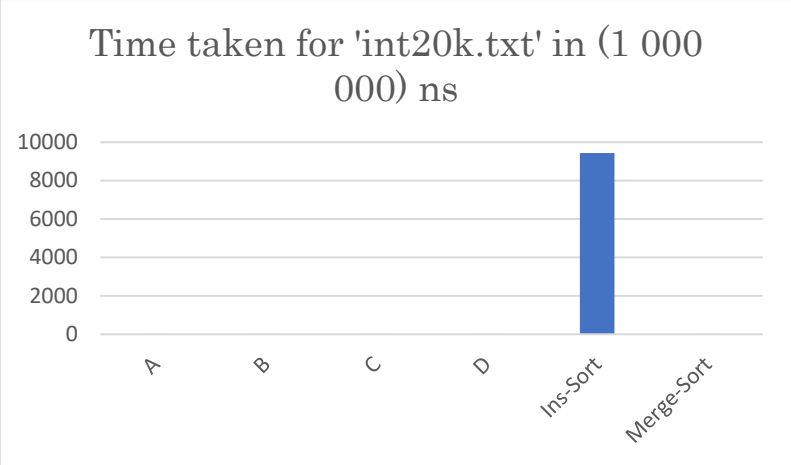
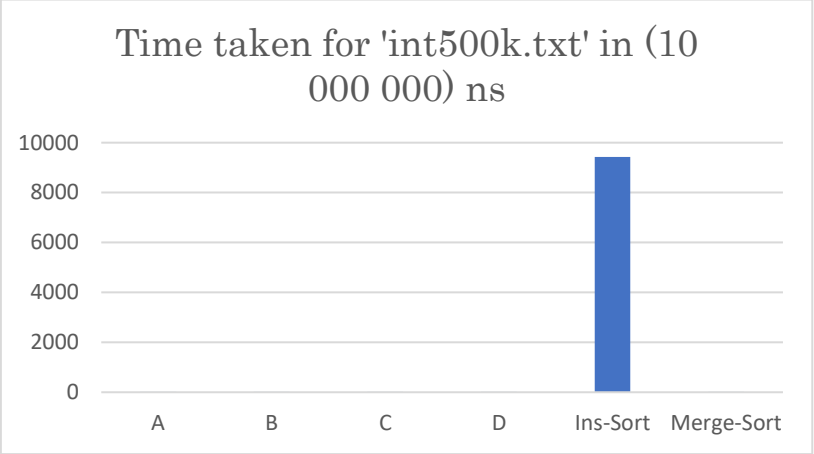


Fig. 3.: Time measured for 500 000 integer array in 10^7 *nanoseconds.*



Note we can see, quite evidently, that implementation A formed better than implementations B – D, although marginally. Although we can see that implementation C perform the worst. This is likely owed to the fact that in relatively large arrays, performing a median of three re-arrangement provides little added performance and thus does not compensate for the actual time required for the total number of calls to the method. Additionally, it should be noted that though A performs marginally better than B, in the case of small sized areas, such as in the case of 'int10.txt', B did outperform it.

Limitation: It's important to know that a clear limitation of using any technique to record the time required to execute a block of code is that a computer, at any given time, hosts a multitude of processes, all competing for a time slice in the memory. Additionally, the number of processes and priority of any given one will vary over time, especially if different background applications are running, and as a result the recorded time is not always precise.

V. Part 3

The final part is tasked with constructing an algorithm, which will generate pathological input sequences for the 'Median-Of-Three Quick-Sort' variant. The method which employs this algorithm takes two integer inputs, *length* and *max*, which respectively correspond to the length of the desired pathological array and the maximum possible integer within the sequence. In order to obtain the worst-case time, the selected pivot must reduce the problem size as minimally as possible. In other words, we would like to form a 'lopsided

partition' after successive partitions. We can achieve this by generated rotated sequences, i.e.: [9, 4, 5, 6, 7, 8]. Each successive partition results in another rotated sequence, thus forcing a worst-case time for sorting the given array.