ADS2: Assessed Exercise 1

Caleb Henshaw – 2810458H

Instructions to run the code:

The main method for the project is inside the TimeSortingAlgorithm.java. The testing for Part 3 and code are inside TaskThreeA.java and TaskThreeB.java

Inside the TimeSortingAlgorithm.java file, please execute the main method inside and the output in the terminal should have everything requested in this assessment. All the files are located in the folder. If the path does not work, please look at line 12 in TimeSortingAlgorithm.java

ADSAE1 ---> SRC ---> ae1

Part 3 is divided into two files. TaskThreeA.java and TaskThreeB.java. Both have their main method inside each file that can be run to show the example output.

Part 1

3-Way-Quicksort

This method involves using swap, used to swap elements during partition. This is just a utility method.

The Partition method partitions the array into three parts: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. It takes four parameters, the array, low and high which mark the range of the subarray to partition and an array pivotIndices to store the pivot elements. The method initializes three pointers: i, mid, and j. Pointer mid scans through the array from left to right, while pointers i and j move inward from the left and right ends. The pivot element is then chosen as the last elements of the subarray (A[high]). The method then iterates through the array, moving elements to the left (less than pivot), to the right (greater than pivot), or leaving them in place (equal to pivot). Finally the pivotIndices is updated with indices less than and equal to partitions end.

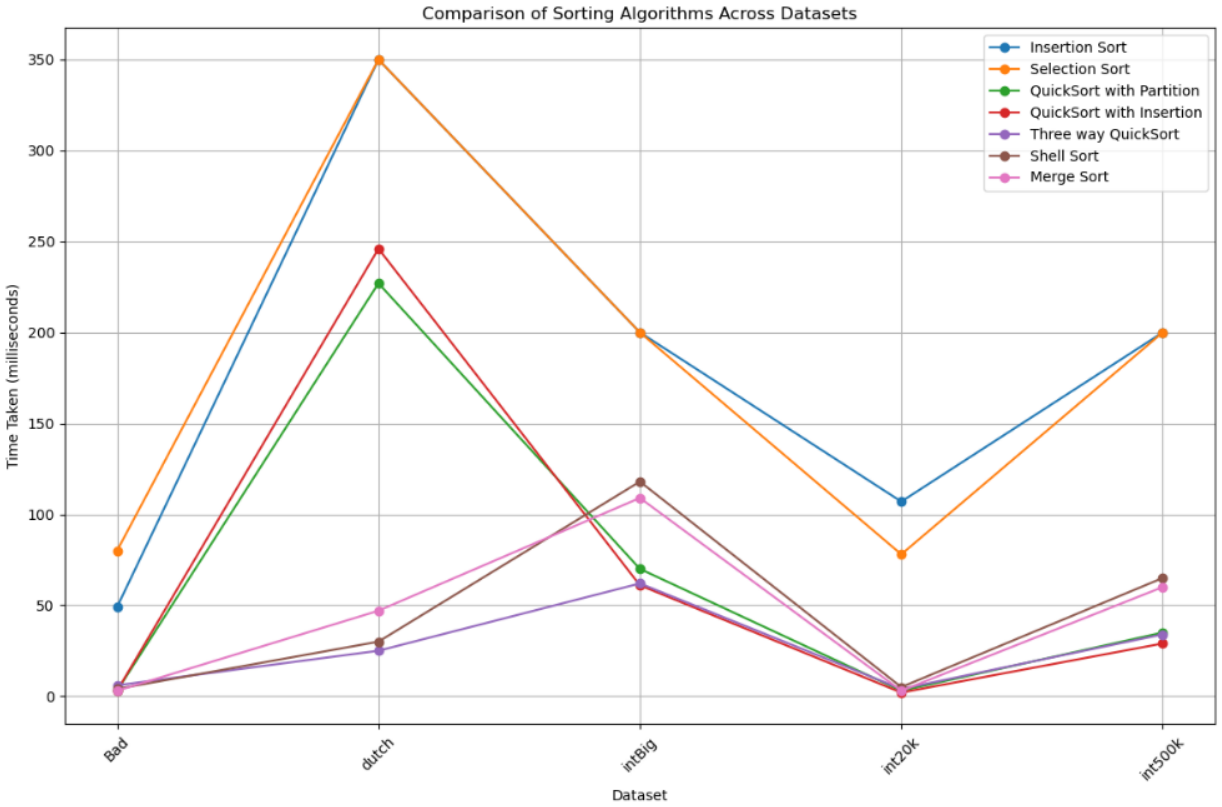
The QuickSort Method recursively sorts the array using the three-way partition. If the low to high range only contains one element or is empty it returns as the array is already sorted, otherwise is calls the partition method to place into three parts and obtain the indices of the partitions. It then recursively sorts the subarrays from left to right of the equal to partition.

This three-way quicksort efficiently handles arrays with many duplicate elements. It achieves an average time complexity of O(n log n). It has a best-case scenario of O(n log n) and has a Worst-case scenario of O(n^2). This is not a stable algorithm. The three-way technique does not guarantee the relative order of equal elements because elements with equal keys may change their positions after

sorting. This algorithm also sorts in place. It rearranges elements within the array without requiring additional storage in regard to the size of the array.

Part 2

Shown in the figure below, is the plotting of all three versions of the quicksort algorithm alongside InsertionSort, ShellSort, SelectionSort and MergeSort.



The findings of the timings of these algorithms show their time complexities first hand. The **InsertionSort and SelectionSort** have both been given cutoff times for the Dutch, intBig, and int500k data sets. Due to their average time complexities of O(n^2). Even with a sorted array, these algorithms have a best case at O(n) and O(n^2), making them very slow. In the Bad data set, these two algorithms worked much harder to solve a very complex data set, even though the size of the input was not nearly as large as others.

The QuickSort with partition algorithm performed considerably well with smaller data sets that are completely unsorted. Because of its average case of O(n log n) it's relatively fast but struggles more with the Dutch data set. This could be because of the pivot point changing and the algorithm struggles more with a larger data set. The **QuickSort with insertion sort** had relative times to its partition counterpart, but was slightly faster in almost every data set. This is because of its time complexity of O(n log n), but has a best case of O(n), where it sometimes has the advantage on smaller subarrays.

The ThreeWay Quicksort also performs relatively close to the other quickSort algorithms except for the Dutch data set, this is because the three way pivot reduces the number of swaps where there are many duplicate values, as seen in the Dutch.txt data set. For the other data sets, the three way performed close to the other quicksort algorithms.

ShellSort is an improvement over insertion sort allowing the exchange of tiems that are far apart. The time complexity is faster than the O(n^2) as it runs at the O(n log n) time complexity. This algorithm ran particularly well when the dataset is partially sorted or nearly sorted. It has low overhead and performs better for larger datasets compared to SelectionSort and InsertionSort.

MergeSort has the time complexity of O(n log n) in all cases, making this a very versatile sorting algorithm. It's very suitable for large data sets (as seen in most of the datasets given). Nothing that even in the Bad data set, MergeSort experiences no troubles as it has only one case for complexity.

Part 3

(A)

For this part, the algorithm can be found in the DescendingMergeSort.java. In terms of the running time of n and i. MergeSort has a time complexity of O(n log n) in all cases. The mergeSortDescending method performs O(n) at each level of recursion. Then the depth of the recursion is is O(log n). For the complexity of i, after sorting the array, the first i elements of the sorted array are copied. This copying takes O(i) time.

```
mergeSortDescending(arr, low:0, arr.length - 1);
return Arrays.copyOfRange(arr, from:0, i);
```

(B)

This solution can be found in the TaskThreeB.java file.

The algorithm implemented is a min heap. It builds a min heap for the first i elements of the array. It iterates over the remaining elements of the array and determine if the current element is greater than the root of the min heap, replaces the root with the current element and maintains the min heap property.

I've used a priority queue in java, as a min heap to efficiently maintain the smallest i elements. The time complexity of this algorithm is O(n log i), n being the size of the input array and i is the number of largest elements to find. Since given in the task that i is always smaller than n, the time complexity can be effectively seen as O(n).

Building the min heap takes O(n) time. Iterating over remaining elements takes O(n-i) time. Each insertion or removal takes O(log i) time. Since i is smaller thah n, log i is effectively a constant factor. Combining the complexities, we have O(n) + O(n-i)= O(n).