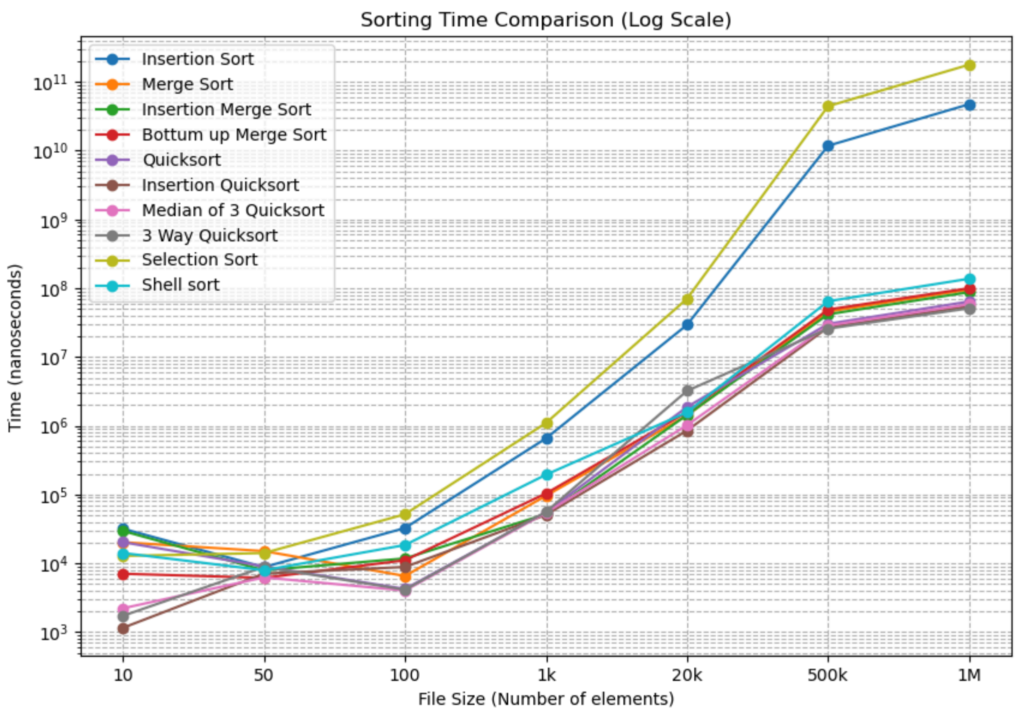
Jack Little - 2874766L

Part 1

This task asked us to implement a selection of sorting algorithms and test them to not only make sure they work correctly and efficiently to their individual standards, but also analyse a comparison between the running times of each separate algorithm. After ensuring that all algorithms worked as intended for small test datasets using JUnit, i proceeded to write code that would iterate through all implemented algorithms and files. Each algorithm was ran 10 times per files and the average time taken per algorithm to completely sort the file was noted in nanoseconds.
After execution these results were written to respective csv files. I then transferred the results of 'dutch.txt' and 'bad.txt' to tabular form and processed the data for the initial specified datasets using MatPlotLib to graphically represent a comparison of time taken per file between the algorithms. This graphical data had to be processed in logarithmic form to show meaningful connections between such large disparities in data.

Figure 1.1 - Graphical comparison of Algorithms

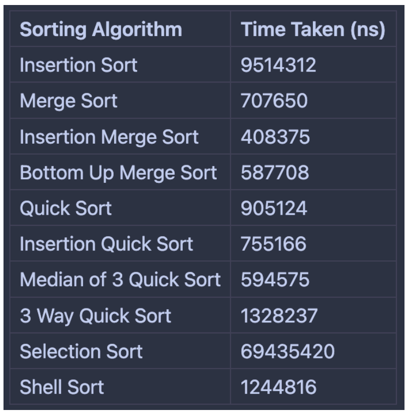


From the above image we can see a few things. Firstly that where $n \approx 100$ or less, the difference in time taken for all the algorithms to completely sort the file is minimal to negligible. However after this point we begin to see a stark contrast, where two clearly

slower algorithms begin to appear. While the main body of algorithms maintain the small difference in time for complete execution, Selection Sort and Insertion sort take significantly larger amounts of time, with the former being the slowest by up to 3 orders of magnitude slower than the fastest. Whilst a clear winner is hard to decipher from this graph it is important to note that we can see that '3 way quicksort' was the fastest in execution for our two largest files, however was slower than the main body for '20k'. This graph clearly highlights the efficiency and time complexities of the algorithms used when it comes to larger and larger datasets.
It is also important to note that the set '1M' is in reference to the 'intBig.txt' file. We can also again highlight the need to use a logarithmic scale as the time taken for the aforementioned slower algorithms is exponentially so that we cannot make accurate comparisons with the other algorithms.

Figure 1.2 - 'bad.txt' algorithm sort times

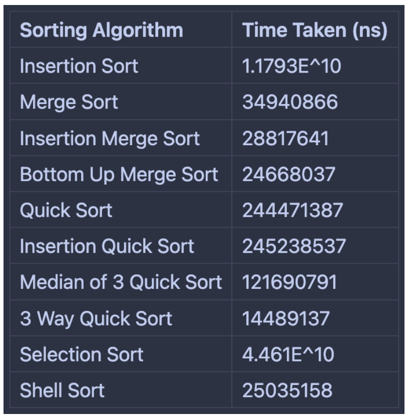
Below is a table showing the times (in nanoseconds) for each algorithm to completely sort the file 'bad.txt'. This file contained around twenty thousand elements. It mirrors the results of figure 1.1, but allows us to more clearly compare, numerically, the difference in execution times. Again with selection sort having the highest time, followed by insertion sort. Here again however our data shows that 3 way quick sort was the slowest of the main grouping of algorithms, while the fastest was merge sort with insertion at n = 20.



Sorting Algorithm	Time Taken (ns)
Insertion Sort	9514312
Merge Sort	707650
Insertion Merge Sort	408375
Bottom Up Merge Sort	587708
Quick Sort	905124
Insertion Quick Sort	755166
Median of 3 Quick Sort	594575
3 Way Quick Sort	1328237
Selection Sort	69435420
Shell Sort	1244816

Dutch.txt algorithm sort times

Similar to above this table shows the time taken (in nanoseconds) for the file 'dutch.txt' to be completely sorted by each algorithm. This file contains around five hundred thousand elements. Again we see our two mirrored outliers for selection sort and insertion sort, however we see that again 3 way quick sort is the fastest in comparison to figure 1.2.



Sorting Algorithm	Time Taken (ns)
Insertion Sort	1.1793E^10
Merge Sort	34940866
Insertion Merge Sort	28817641
Bottom Up Merge Sort	24668037
Quick Sort	244471387
Insertion Quick Sort	245238537
Median of 3 Quick Sort	121690791
3 Way Quick Sort	14489137
Selection Sort	4.461E^10
Shell Sort	25035158

Part 2

We are being tasked with finding the K most viewed videos out of N videos. One of the best probable solutions to this, as N is far greater than K, is to use a form of min-heap sort. Our heap will have K elements and we will iterate through our list of N videos. If our heap is not full then we will add our video N, otherwise if our heap is full, we will then compare the like count of video N with the lowest like count in our heap K, if it is higher, replace K with this element of N and continue, if the video count of N is less then we continue on without changing anything.
By doing this we only ever process our video list once, that is O(n), each insertion and deletion takes O(log k) which worst case scenario we do N times so we can assume O(n log k). So we can conclude that our best, worst and average cases are all O(n log k).

Note

If running the main timeAll.java file, please ensure it is ran in the same directory as the rest of the algorithms as they have some shared methods etc, additionally file oaths etc may need changed to suit. Execution of this took around 40 mins on my laptop. All algorithms were tested on small edge case sets using JUnit and maven.