COMP333 Assignment 1 Report

Name: George Hayward
Student Number: 43692486

Stage 1:

routeMinDistance:

Algorithm Description: The algorithm used for routeMinDistance employs an exhaustive approach. A depth first search is used to traverse the stations until the destination is found. Once it is found that path and its distance from the origin is stored within an additional arrayList variable added to the station class. The DFS then continues to search for a shorter route, however if a path exceeds the current shortest routes distance, the traversal from that station is terminated (does not look at that stations adjacent stations).

Algorithm Explanation: This algorithm is a mixture between an exhaustive DFS and Dijkstra's algorithm (due to its halting when exceeding the current shortest path). I believe this combination of algorithm implementations was effective because without knowing what adjacent station is on what line, it is difficult to follow lines which would lead to a destination faster. Therefore, a DFS approach seemed to be the best way to find the destination, instead of BFS which may exhaust all possible stations before reaching the destination.

The implementation of the DFS is quite typical, using a stack to hold all the adjacent stations that need visiting.

```
while(!stationStack.empty()) {
```

The main loop of this algorithm keeps going until the stack is empty. Stations to visit are only added to the stack if the distance required to go to these stations is less than the current shortest route and if this current route is shorter than the one already held within that station object.

```
private ArrayList<String> bestRoute;
```

Each time a new best route is found, it is recorded within the bestRoute of that destination. Once there are no more stations to visit in the stack, the while loop is exited and the bestRoute of the destination is returned.

routeMinStop:

Algorithm Description: The routeMinStop algorithm operates nearly exactly the same as the routeMinDistance algorithm. The only differences being that instead of a DFS approach it uses are BFS approach. Also the distance travelled is not used, the main point of attention is instead focused on the. size() of the bestRoute arrayList path's within the station objects.
E.g.

```
stationList.get(destination).getBestRoute().size() > adjacentStationPath.size()
```

We are comparing the sizes instead of the distance in routeMinDistance:

```
findTotalDistance(stationList.get(adjStation).getBestRoute()) >
findTotalDistance(adjacentStationPath)
```

findTotalDistance will be explained later in this report.

routeMinDistance + Failures and routeMinStop + Failures:

These two methods work nearly exactly the same as the previously mentioned methods. The only difference being this one additional conditional statement within the adjacent station iteration loop.

```
if(!failures.contains(adjStation)) {
```

This ensures that no station within the input failures treeSet is visited.

findTotalDistance:

This method is quite simple. If the input path is less than 2 then this means there isn't a distance to calculate. If it is greater than 2 then there is a path. Iterate through every item of the path adding the neighbours distances to the distance variable and returning it at the end.

Stage 2:

optimalScanCost: PLEASE NOTE! STAGE 1 WORKS CORRECTLY, THIS METHOD IS NOT COMPLETE AND WILL RUN FOREVER (unless you have a quantum computer ☺)

This website was used to help my implementation.
https://www.geeksforgeeks.org/cutting-a-rod-dp-13/

Algorithm Description: This algorithm works similarly to the rod cutting algorithm discussed in lectures. However, I believe it is not passing all the tests due to the lack of memoization. It is taking too long to divide large routes without memory of optimum sub problems.

Algorithm Explanation: The algorithm loops through every possible place the route can be cut. Starting at the second station to the second last station. Two sub-routes are created from the split in the original route

```
headRoute.addAll(route.subList(0, i + 1));
tailRoute.addAll(route.subList(i, route.size()));
```

and these routes distances are calculated.

```
int headDistance = findTotalDistance(headRoute);
int tailDistance = findTotalDistance(tailRoute);
```

The returned value is then these routes distances + a recursive call of these sub routes. This step finds the optimum sub route cutting combination which will yield the lowest scan cost.

```
bestCombination = Math.min(bestCombination, headDistance + tailDistance +
optimalScanCost(headRoute) + optimalScanCost(tailRoute));
```

In order to have an effective algorithm that works faster, there would need to be an exterior data structure holding the optimum solution to subproblems .

optimalScanSolution:

While not completing this method. The way that it would be completed is by running a version of optimalScanCost. Every time there is a split which yields an optimum sub route the index of that split (station) will be saved within the arrayList.