Final Project Report

Background

A delivery company which processes orders in the order of the soonest deadline using a max-heap, then finds customer information using Red-Black Tree, and finally finds the shortest path to the delivery destination using Dijkstra's algorithm.

This program will use Java as the programming language and Windows OS.

Functionalities:

(I/O: Read in the provided input and save the information in data structure.)

1. Identify the order with earliest delivery date among all the orders.
2. Find the order information including name, address, delivery deadline, and order ID with a given order ID.
3. Find the closest distance from the source point to the desired destination (Address ID)

Assumptions:

There are 4 inputs to be given by the user in this program, brief descriptions are shown below:

1. `listOfOrders`: A list of Orders that will be sorted using heap sort and inserted in the Red-Black Tree. (Functionality 1, 2)

   limitations: The orders are expected to be in the form of the Order class, all ID numbers should be unique.
2. `map`: a 2D array providing the connections and their distances between address IDs. A full map with all possible connections between all IDs needs to be provided. (Functionality 3)

   limitations: map[x][y]==0 should be satisfied when x == y. map is a square 2D

array.

3. `destinationID`: A desired address ID (int) that will help returning the shortest path to this point from where the delivery company is located.
   (Functionality 3)
   limitations: The address ID should be valid from the map above.
4. `findCustomerByID`: This ID will return its corresponding order information.
   (Functionality 2)
   limitations: The order ID should be valid from the input orders above.

All input information will be certain and will not change over the runtime except it may be marked off when its calculation is done. Some examples and class specifications can be found in the README.md file in the assignment package.

Algorithms analysis:

1. Heap sort (Functionality 1)

This algorithm is implemented using the Heap sort. Given a list of customer orders, heap sort will be implemented to sort these orders in an ascending order of the soonest deadline.

In this functionality, heap sort algorithm, more specifically Max-heap is implemented to sort a list of order to make further access to the list more convenient, this algorithm is chosen because it provides one of the least theoretic time complexity to sort an array among the common sorting algorithms. Furthermore, heap structure can be implemented in-place within an array to avoid the over-complicated tree construction while still maintaining its properties with space complexity of O(1). To assist with the functionality, Max-heap method sorts the list in an ascending order of

delivery date and enables the access to the order with soonest delivery date by accessing the first element of the sorted list.

- **Theoretical time complexity analysis**

A complete heap sort is split into two sections, heap formation and heapify, the detailed time complexity is as follow:

1. Heapify

Worst case:

The total time of heapify function is a recurrence: $T(n) \leq T\left(\frac{2n}{3}\right) + O(1)$

Worst case occurs when the leaves of the tree is exactly half full, where the largest possible subtree (left most) will have at most the size of 2/3 of the input size since it has the height of h while the right subtree having height of $h-1$. Therefore, the size of the sub-problem is 2/3 and the recursive call will take T(2/3) time. Using the master theorem, a = 1, b = 3/2, f (n) = 1, by satisfying case 2:

$$T(n) = \theta(n^{log_{3/2}1} \log n) = \theta(\log n)$$

Average case:

$$T(n) \geq T\left(\frac{n}{2}\right) + O(1)$$

In average case, it is assumed that the height of the tree is balanced, so the height of the left subtree equals the height of the right subtree, accordingly, the size of the sub-problem is halved in each recursive call, Using the master theorem, a = 1, b = 2, f (n) = 1, by satisfying case 2:

$$T(n) = \theta(n^{log_2 1} \log n) = \theta(\log n)$$

2. **Heap-buiding:**

This function is called sort in the source code, it contains the heap formation by calling to heapify for each element in the array, as well as rearrange the heap structure to ascending list order, the time complexity is $\theta\left(\frac{n}{2}+n\right)=\theta(n)$ in both worst and average cases as it loops through the array list in which ever cases, of which $\frac{n}{2}$ denotes the calling to heapify and n denotes the rearrangement of the array, from array with "heap tags" to normal ascending array, each element will only be looped once.

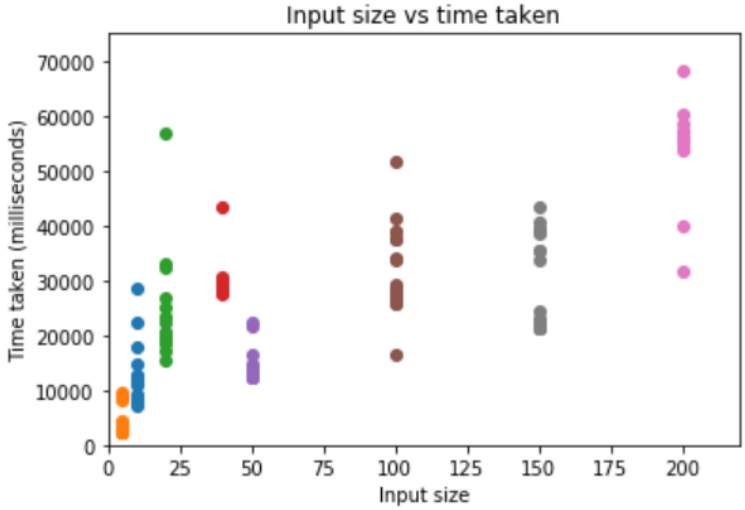
Total complexity:

By combining the above, a time complexity can be obtained:

$$T(n) = \theta(n) \times \theta(\log n) = \theta(n \log n)$$

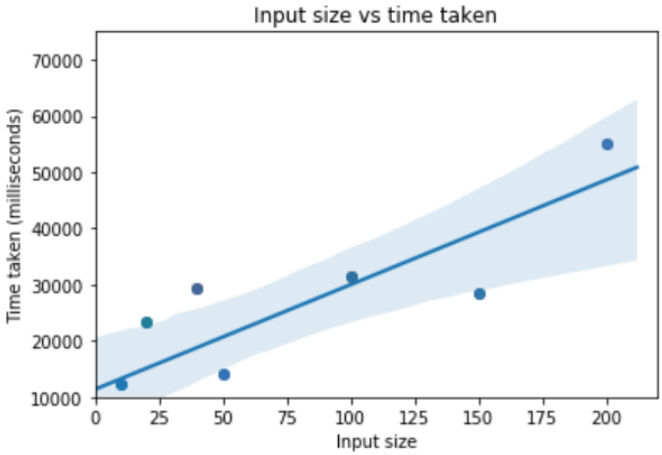
- **Empirical time complexity analysis and comparison**

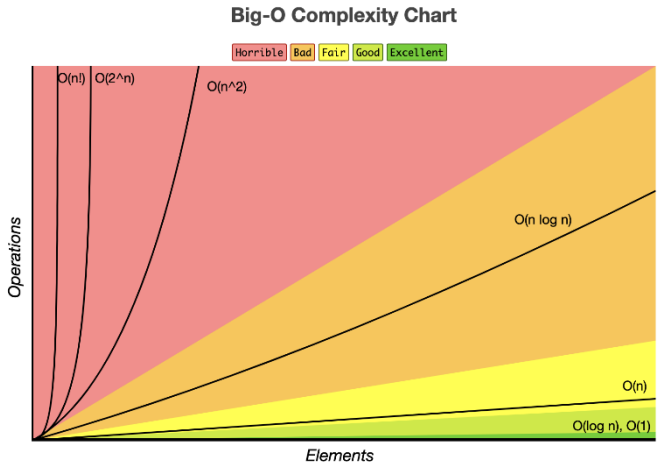
Test cases with input size between 5 and 200 is implemented with 20 trials in each test case. The output of ascending list according to the delivery date was produced as expected, and in terms of performance, the time taken for them to go through sorting function were measured in millisecond as outputs. Below is the scatter chart of all test cases with different input sizes.



After calculating the median of all trials, a regression plot is also obtained using python's built-in function, with intercept of 186.400 and coefficient of 11369.356, from these information and the plot below, it shows an approximate matching between theoretical complexity of θ(n log n).

```
intercept: 186.40026189436918
coefficient: 11369.356176342208
```





Big-O Complexity Chart: http://bigocheatsheet.com/

2. Red-Black Tree (Functionality 2)

Functionality 2 is implemented with a Red-Black tree data structure. With the given list of orders as above, the data will be inserted into a Red-Black Tree with assist of relevant methods within the same class, the tree formulation is based on the order IDs, hence the functionality 2 can access the order information from the Red-Black Tree according to these IDs.

The properties of Red-Black Tree make it a balanced tree comparing to normal binary search tree, and the number of rotations required for an insertion in average is lower than AVL tree. Accessing an element in a balanced tree would guarantee the stability of time taken, moreover, since the functionality is storing orders' information, which

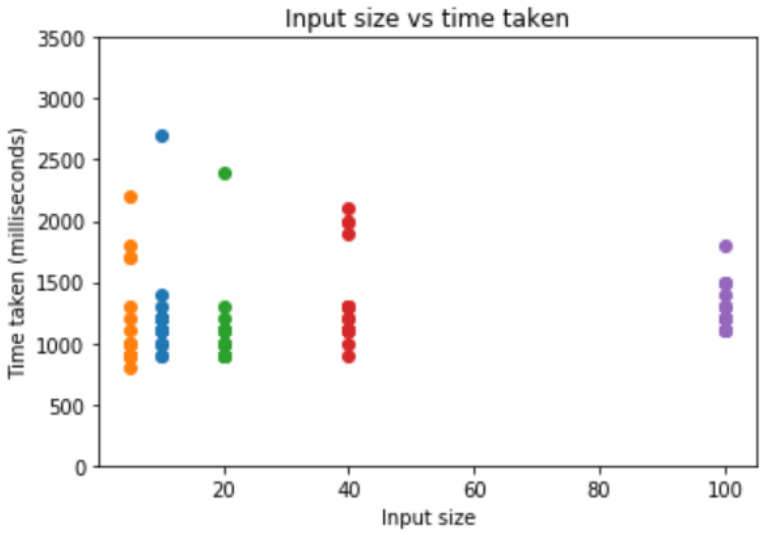
is likely to be changing all the time, I have chosen Red-Black Tree to be the data structure of storing order information to reduce the cost of rotation operations and gives more flexibility on balancing tree.

- **Theoretical time complexity analysis**

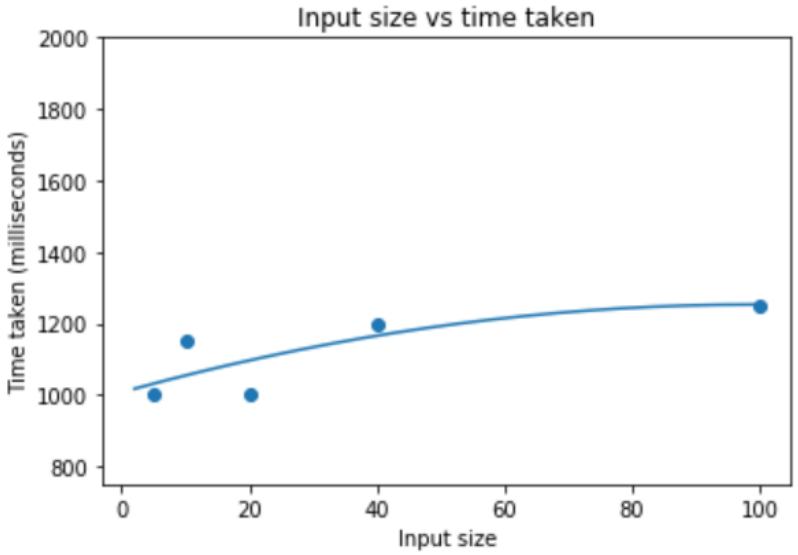
  In red-black tree data structure, operations including insert, delete, and search (or find) share similar concepts, hence they have the same time complexity, they are all finding the correct node then do the following constant time operations total time of $O(1)$, such as at most 3 rotations of subtrees and assign or access to the value of the node. For finding the node, since red-black tree guarantees the balance of the tree by having the same number of black nodes in each path. The time complexity of accessing a node is O(height of the tree), in this case, the height of the tree is $\log n$ in both average case and worst case, therefore, for all red-black tree operations: insertion, deletion and searching, the time complexity in both average and worst cases is $T(n) = \theta(\log n) + O(1) = \theta(\log n)$.

- **Empirical time complexity analysis and comparison**

In the Red-Black Tree section, I have implemented the common methods such as insert, delete, and search. All the methods have returned the correct output to have the correctness of the program proven. After testing with several test cases and practical time measurements, I have observed a high similarity in time taken between all these methods, hence I choose to do the empirical time complexity analysis and comparison based on the search method, which is the most relevant one to the second functionality. To test the time complexity of the search method, 5 test cases is applied with 20 trials in each test case, the following plot shows the time taken (milliseconds) of corresponding input size in all trials.



After extracting the medians of all test cases, the regression plot is demonstrated below.



The result has shown a slow growth in time taken with increase in input size, greater than $\theta(1)$ and smaller than $\theta(n)$ as it tends to increase less with greater input. Hence the time complexity of Red-Black tree is proven to be θ(log n), identical to the hypothesis.

• Dijkstra Algorithm (Functionality 3)

Dijkstra algorithm is implemented to find the shortest distances between a fixed location to all other vertices, to make the algorithm more intuitive, I decided to denote every location on the map by an address ID, throughout the program, the address of each customer is also represented by an address ID. In this algorithm, the shortest path to an address from the delivery company with consideration of connections and distances between different locations.

This algorithm will suit the above situation as it is a path finding approach from a single source point, which would be the delivery company in this case. Each step is decided according to the current status (sum of the distances).

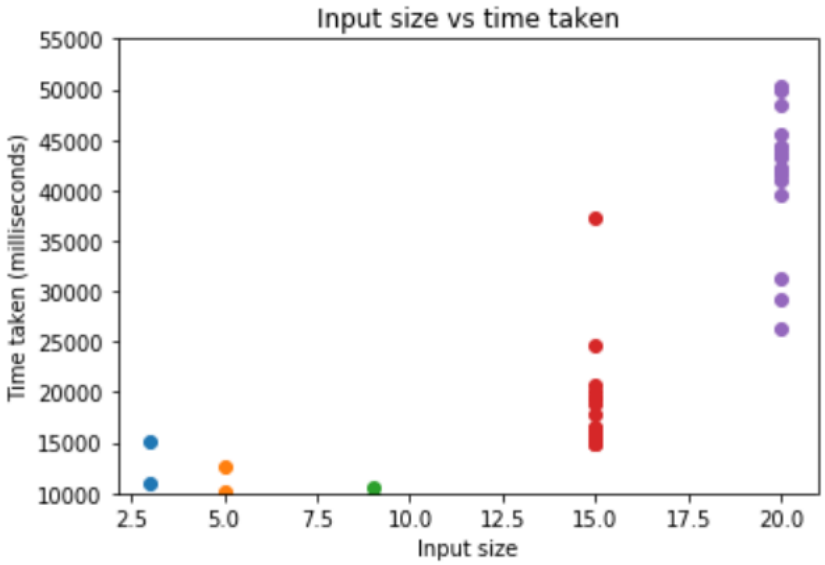
- **Theoretical time complexity analysis**

  In Dijkstra algorithm there are mainly 2 loops affecting on the time complexity of the algorithm, one is finding the vertex with minimum distance value while the other loop sums up all the paths and decides whether or not to update the path based on comparing old one with the new one. In the finding minimum distance loop, I did not choose to implement it with max-heap, so the time complexity will require $T(n) = O(n)$ to loop through all vertices and find the minimum value, where n represents the number of input(vertices), this loop is included in the second loop. And for each adjacent vertex, a decision of whether or not to update the latest option according to the distance has a time complexity of $T(n) = O(n)$, this loop will be iterated until the target vertex is found. Hence the total time complexity of this algorithm is

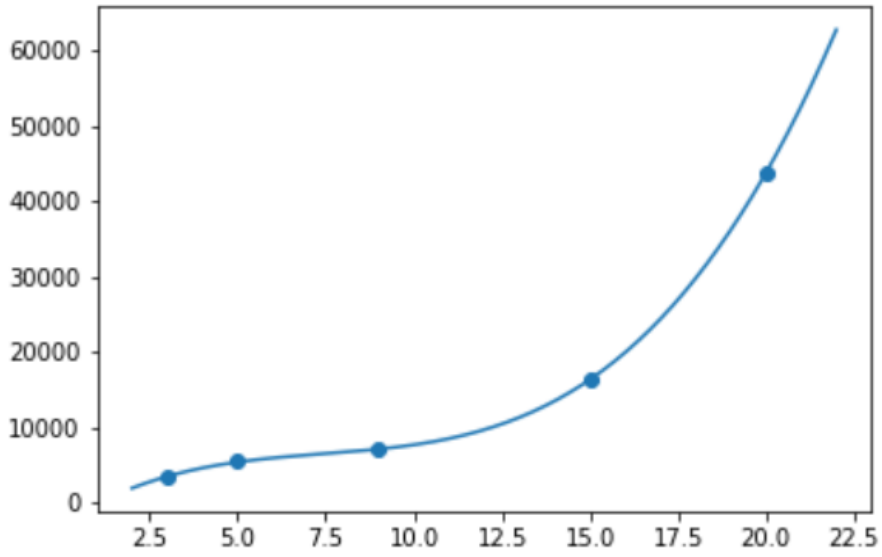
  $$T(n) = O(n) * O(n) = O(n^2)$$

- **Empirical time complexity analysis and comparison**

  Test cases with input size of 3,5,9,15,20 is implemented to obtain the empirical time complexity, the chart below shows the relationship between input cases and the time taken, each test with 20 trials.



The chart below shows the regression of median time taken vs input size.



The output has come out as expected in form of the shortest distance to the desired destination to prove the correctness of the program. In the aspect of time complexity, this test result has shown a relatively big increase in time taken proportional to the input size, with a regression patten similar to n^2 curve,

according to the data obtained, the Dijkstra algorithm has an empirical time complexity similar but not up to n^2, which matches with the theory that the time complexity is $O(n^2)$. In all the time complexity test cases above, due to the hardware reasons such as CPU time, some of the data might be off the expected values and inconsistent, especially through my observe, the first trial often takes more time, however, in most cases, the practical testing has proven the correctness of the theoretical time complexity against the real time complexity.

References:

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

http://algebra.sci.csueastbay.edu/~grewe/CS3240/Mat/Heap/index.html

https://www.geeksforgeeks.org/heap-sort/

https://www.programiz.com/dsa/red-black-tree