



SUBJECT NAME:	ALGORITHM DESIGN AND ANALYSIS
SUBJECT CODE:	TCP2101
ASSIGNMENT TITLE:	
ASSIGNMENT SETTER NAME:	LOO YIM LING
SUBMISSION DEADLINE:	

No.	Student Name	Student ID	Task Descriptions	Percentage %
1	Mayoof Zayed Yasir Mayoof	1211306856	Q1: Dataset 1 and 2	25%
2	NUR HANNA DANIA BINTI ABDULLAH	1211306199	Q2: Heap and Merge Sort	25%
3	MARESH,MOHANAD AYOUB ALI	1211303662	Q3: Dijkstra's and Kruskal	25%
4	Alassaf, Abdalrahman	1211306080	Q4: 0/1 Knapsack	25%

Table of Contents

1. Dataset Generation (Q1)	4
1.1 Dataset (1)	4
1.1.1 the purpose of Dataset (1).	4
1.1.2 Source Code with Explanation: Data Set (1)	4
1.1.3 Output: Data Generated (Data Set 1)	5
1.2 Dataset (2)	5
1.2.1 the purpose of Dataset (2).	5
1.2.2 Source Code: Data set (2) with Explanation	5
1.2.3 Output: Generated data (Data Set 2)	14
1.2.4 Output: Generated Graph (Data Set 2)	15
1.2.5 Output: Generated Graph visualization (Data set 2)	16
2. Heap Sort and Merge Sort (Q2)	17
2.1 Heap Sort (HS)	17
2.1.1 Heap Sort Overview	17
2.1.2 Source Code: Heap Sort with Explanation	17
2.1.3 Output: Sorted data Using HS (Data Set 1)	19
2.2 Merge Sort (MS)	20
2.2.1 Merge Sort Overview	20
2.2.2 Source Code: Merge Sort with Explanation	20
2.2.3 Output: Sorted data Using MS (Data Set 1)	22
3. Shortest Paths and Minimum Spanning Tree (Q3):	24
3.1 Shortest Paths (SP)	24
3.1.1 Dijkstra's algorithm Overview	24
3.1.2 Source Code (SP using Dijkstra's algorithm with Explanation)	24
3.1.3 Output: SP using Dijkstra's algorithm (Data Set 2)	28
3.1.4 Output: SP Graph Representation using Dijkstra's algorithm (Data Set 2)	28
3.1.5 Output: SP Graph Visualization using Dijkstra's algorithm (Data Set 2)	29
3.2 Minimum Spanning Tree (MST)	30
3.2.1 Kruskal's algorithm Overview	30
3.2.2 Source Code and Explanation: MST using Kruskal's Algorithm	30
3.2.2 Output: MST Using Kruskal's algorithm (Data Set 2)	33

3.2.3 Output: MST Graph Visualization using Kruskal's algorithm (Data Set 2).....	34
4. Dynamic Programming (Q4)	34
4.1 0/1 Knapsack Problem (Dynamic Programming)	34
4.1.1 Knapsack Problem Overview.....	34
4.1.2 Source Code: Knapsack 0\1(Dynamic Programming).....	35
4.1.3 Source Code Explanation (Knapsack 0\1(Dynamic Programming).....	35
4.1.4 Output: available Stations (Knapsack 0\1(Dynamic Programming)	36
4.1.5 Output: Stations to be visited (Knapsack 0\1(Dynamic Programming).....	37
4.1.6 Output: Fastest path to winning stations(Knapsack 0\1(Dynamic Programming)	37
5 Time complexity.....	38
5.1 Data Set 2 (Time complexity)	38
6 Space Complexity	49
6.1 Data Set 2 (Space Complexity)	49
6.2 Kruskal's algorithm (Space Complexity)	49
6.3 Dijkstra's algorithm (Space Complexity).....	50
6.4 0/1 Knapsack algorithm (Space Complexity).....	50
5.4 Dijkstra's Algorithm	Error! Bookmark not defined.
5.5 0/1 knapsack algorithm.....	Error! Bookmark not defined.

1. Dataset Generation (Q1)

1.1 Dataset (1)

1.1.1 The purpose of Dataset (1).

To generate data from it to use it as an input to test the implementation of the Heap and merge sort algorithms.

The generated data sets from this data set implementation we will use only in Q2.

1.1.2 Source Code with Explanation: Data Set (1)

Includes necessary C++ libraries (iostream, list, vector, random, etc.).

RandomDataSetGenerator class has a public member function **generateDataSet** and a private member generator of type **default_random_engine**.

generateDataSet(int[], int): Generates a random dataset of integers and stores it in the provided array. It takes the array and its size as parameters. It uses a **uniform_int_distribution** with a range from 1 to the given size. It fills the array with random integers generated from the distribution.

generator: An instance of **default_random_engine** initialized with the provided seed.

RandomDataSetGenerator(unsigned int seed): Constructor that takes an unsigned integer seed and initializes the generator with it.

The purpose of this class is to provide a simple way to generate a random dataset of integers within a specified range.

In summary, this class allows you to generate random datasets of integers using the **generateDataSet function**, and the seed for randomness can be set during object creation.

```

1  #include <iostream>
2  #include <list>
3  #include <vector>
4  #include <random>
5  using namespace std;
6
7  class RandomDataSetGenerator
8  {
9  public:
10     RandomDataSetGenerator(unsigned int seed) : generator(seed) {}
11
12     void generateDataSet(int dataSet[], int size)
13     {
14         uniform_int_distribution<int> distribution(1, size);
15
16         for (int i = 0; i < size; ++i)
17         {
18             dataSet[i] = (distribution(generator));
19         }
20     }
21
22 private:
23     default_random_engine generator;
24 };
25
26

```

code snippet 1 dataSet1 implementation.

1.1.3 Output: Data Generated (Data Set 1)

Here we have the generated data with input size (100). With the group leader ID as the random seed reference (1211306080). We are just going to show generated data for set 1 as the size of the other sets is big.

Output snippet 1 shows that the data generated for set 1.

```
Set 1 (seed: 1211306080)
13 53 86 40 91 68 66 5 52 79 16 54 2 69 58 77 53 77 93 71 18 7 48 12 17 2 33 86 23 29 73 64 94 4 18 78 73 61 61 11 13 60 24 32 84 95 38 87 82 5 73 56 19 10 77 23 7 66 51 19 4 91 62 35 8 90 76 23 39 20 22 49 35 4
9 85 60 23 99 80 33 93 69 3 15 64 72 43 23 53 92 8 45 12 23 49 33 82 96 32 45
```

Output snippet 1 Data generated (Data Set 1).

Output Area 1 provides a clearer view of the results obtained from Data Set 1. offering additional clarity in case the details in the snippet are not immediately apparent.

Set 1:

```
13 53 86 40 91 68 66 5 52 79 16 54 2 69 58 77 53 77 93 71 18 7 48 12 17 2 33 86 23 29 73
64 94 4 18 78 73 61 61 11 13 60 24 32 84 95 38 87 82 5 73 56 19 10 77 23 7 66 51 19 4 91
62 35 8 90 76 23 39 20 22 49 35 49 85 60 23 99 80 33 93 69 3 15 64 72 43 23 53 92 8 45 12
```

Output Area 1 Data generated (Data Set 1)

1.2 Dataset (2)

1.2.1 the purpose of Dataset (2).

to generate dataset (Graph) for 20 stations (vertices) with 54 routes (edges) that connect between stations. Each station connects to at least 3 stations. Each station has data consisting of name, x-coordinate, y-coordinate, y-coordinate, z-coordinate, weight and profit. The weight and profit refer to the treasure to be collected from the station. The generated data set here will be used as an input to the Q3 and Q4.

1.2.2 Source Code: Data set (2) with Explanation

TreasureHuntDataSet class has local members representing Dataset 2, a graph, a vector of stations, the number of stations, the number of routes, and a string for group IDs sum.

getStations(): Returns the vector of stations.

TreasureHuntDataSet(): Constructor that takes the number of stations, the number of routes, and the group IDs sum. Initializes the graph and calls functions to initialize stations and generate the dataset.

Class: DataSet2.cpp

```

1  #include <iostream>
2  #include <list>
3  #include <vector>
4  #include <random>
5  #include <set>
6  #include <unordered_map>
7  #include "Graph.cpp"
8  using namespace std;
9
10 class TreasureHuntDataSet
11 {
12 private:
13     Graph graph;
14     vector<Station> stations;
15     int numOfStations;
16     int numOfRoutes;
17     string groupIdsSum;
18
19 public:
20     vector<Station> getStations()
21     {
22         return stations;
23     }
24     TreasureHuntDataSet(int numOfStations, int numOfRoutes, string groupIdsSum)
25     {
26         this->numOfStations = numOfStations;
27         this->numOfRoutes = numOfRoutes;
28         this->groupIdsSum = groupIdsSum;
29         graph = Graph(numOfStations, numOfRoutes);
30
31         initializeStations();
32         generateDataSet();
33     }

```

code snippet 2 DataSet2 implementation

addRoute(Station, unordered_map<int, vector<Station>>&): Adds a route between two stations based on provided candidates.

```

1  // Function to add a route between two stations
2  void addRoute(Station x, unordered_map<int, vector<Station>> &candidates)
3  {
4      int stationYIndex = getRandomIndex(candidates[x.id].size());
5      Station y = candidates[x.id][stationYIndex]; // get random station to create a route with Station x
6      int distance = calculateDistance(x, y);
7
8      graph.addRoute(Route(x, y, distance));
9      // Update candidates List to avoid choosing the same station again
10     removeFromCandidates(candidates[x.id], stationYIndex);
11     removeFromCandidates(candidates[y.id], getIndexById(candidates[y.id], x.id));
12 }

```

code snippet 3 DataSet2 implementation.

generateDataSet(): Designs the dataset by ensuring each station has at least 3 connections and then adding the remaining routes.

```

1 void generateDataSet()
2 {
3     // the key represent the station id and the vector<Station> is the List that the key station is allowed to choose from to create a new route
4     unordered_map<int, vector<Station>> candidates;
5     initializeCandidates(candidates);
6     int routesCounter = 0;
7     // Ensure each station has at least 3 connections
8     for (int i = 0; i < numOfStations; i++)
9     {
10         int numOfAdjacents = graph.getNumOfAdjacents(stations[i].id);
11
12         for (int j = numOfAdjacents; j < 3; j++, routesCounter++)
13             addRoute(stations[i], candidates);
14     }
15
16     // Add the remaining routes
17     for (int i = 0; routesCounter < numOfRoutes; routesCounter++, i++)
18         addRoute(stations[i % numOfStations], candidates);
19 }

```

code snippet 4 DataSet2 implementation.

printDataSet(): Prints the generated dataset.

initializeCandidates(unordered_map<int, vector<Station>>&): Initializes the candidates map.

```

1 // Function to print the dataset
2
3 void printDataSet()
4 {
5     graph.printGraph();
6 }
7
8 Graph getDataSet()
9 {
10     return graph;
11 }
12
13 private:
14     // Function to initialize the candidates map
15     void initializeCandidates(unordered_map<int, vector<Station>> &candidates)
16     {
17         for (int i = 0; i < stations.size(); ++i)
18         {
19             candidates[stations[i].id] = stations;
20             removeFromCandidates(candidates[stations[i].id], i); // Remove the station itself from the candidates
21         }
22     }

```

code snippet 5 DataSet2 implementation.

removeFromCandidates(vector<Station>&, int): Removes an element from a vector.

```

1 // Function to remove an element from a vector
2 void removeFromCandidates(vector<Station> &candidates, int indexToRemove)
3 {
4     swap(candidates[indexToRemove], candidates.back());
5     candidates.pop_back();
6 }

```

code snippet 6 DataSet2 implementation.

getIndexById(vector<Station>&, int): Gets the index of a station in a vector by its ID.

```

1 // Function to get the index of a station in a vector by its ID
2 int getIndexById(vector<Station> &stations, int id)
3 {
4     for (int i = 0; i < stations.size(); ++i)
5     {
6         if (stations[i].id == id)
7             return i;
8     }
9     return -1;
10 }

```

code snippet 7 DataSet2 implementation.

calculateDistance(Station, Station): Calculates the distance between two stations.

```

1 // Function to calculate the distance between two stations
2 int calculateDistance(Station stationX, Station stationY)
3 {
4     double power = 2.0;
5     double arg1 = stationY.x - stationX.x;
6     double arg2 = stationY.y - stationX.y;
7     double arg3 = stationY.z - stationX.z;
8
9     int distance = sqrt(pow(arg1, power) + pow(arg2, power) + pow(arg3, power));
10    return distance;
11 }

```

code snippet 8 DataSet2 implementation.

getRandomIndex(int): Gets a random index within a given range. Used to connect a station to another random station.

```

1 // Function to get a random index within a given range
2 int getRandomIndex(int upperBound)
3 {
4     random_device rd;
5     mt19937 gen(rd());
6     uniform_int_distribution<int> distribution(0, upperBound - 1);
7     return distribution(gen);
8 }

```

code snippet 9 DataSet2 implementation.

getRandomValueForData(int): Takes the length of the required value and then returns a random value from the group IDs sum.

```
1 // Function to get a random value based on group IDs sum
2 int getRandomValueForData(int Length)
3 {
4
5     string str = "";
6     for (int i = 0; i < Length; i++)
7         str += groupIdsSum[getRandomIndex(groupIdsSum.length())];
8
9     return stoi(str) + 10;
10 }
```

code snippet 10 DataSet2 implementation.

initializeStations(): Initializes stations with random values.

```
1 // Function to initialize stations with random values
2 void initializeStations()
3 {
4
5     for (int i = 0; i < numOfStations; i++)
6     {
7         char name = 'A' + i;
8         int id = i;
9         int x = getRandomValueForData(3);
10        int y = getRandomValueForData(3);
11        int z = getRandomValueForData(3);
12        int weight = getRandomValueForData(2);
13        int profit = getRandomValueForData(2);
14        stations.push_back(Station(name, id, x, y, z, weight, profit));
15    }
16 }
17 ;;
```

code snippet 11 DataSet2 implementation

In summary, this class provides a convenient way to generate a dataset for a treasure hunt scenario, considering station attributes, distances between stations, and group IDs sum. The dataset is represented as a graph with routes between stations.

Station Class:

Represents a station in a treasure hunt scenario.

Attributes include the station's name (character), ID, coordinates (x, y, z), weight, and profit.

Provides multiple constructors for different ways of initializing a station.

Class : Graph.cpp

```

1  #include <iostream>
2  #include <list>
3  #include <vector>
4  #include <random>
5  #include <set>
6  #include <unordered_map>
7  using namespace std;
8
9  class Station
10 {
11 public:
12     char name;
13     int id;
14     int x;
15     int y;
16     int z;
17     int weight;
18     int profit;
19     Station(char name, int id, int x, int y, int z, int weight, int profit)
20     {
21         this->name = name;
22         this->id = id;
23         this->x = x;
24         this->y = y;
25         this->z = z;
26         this->weight = weight;
27         this->profit = profit;
28     }
29     Station(int weight, int profit)
30     {
31         this->weight = weight;
32         this->profit = profit;
33     }
34     Station(int id)
35     {
36         this->id = id;
37     }
38     Station(char name, int id)
39     {
40         this->name = name;
41         this->id = id;
42     }
43     Station() : id(0), x(0), y(0), z(0), weight(0), profit(0) {}
44     ~Station() = default;
45 };

```

code snippet 12 Graph implementation.

Route Class:

Represents a route between two stations.

Contains source (srcStation), destination (dstStation) stations, and the distance between them. Constructor initializes the source, destination, and distance.

```

1  class Route
2  {
3  public:
4      Station srcStation;
5      Station dstStation;
6      int distance;
7      Route(Station srcStation, Station dstStation, int distance)
8      {
9          this->srcStation = srcStation;
10         this->dstStation = dstStation;
11         this->distance = distance;
12     }
13 };

```

code snippet 13 Graph implementation.

Graph Class:

Represents a graph where stations are nodes, and routes between stations are edges.

Attributes include the number of stations, the number of routes, and an array of linked lists (routes) to store the routes for each station. Provides methods to interact with the graph, such as adding routes, checking adjacency, getting adjacent stations, and printing the graph.

Uses adjacency lists to represent the connections between stations. The Graph class supports adding undirected routes, updating adjacency lists accordingly.

```

1  class Graph
2  {
3      int numOfStations;
4      int numOfRoutes;
5      list<Route> *routes;
6
7  public:
8      Graph(int numOfStations, int numOfRoutes)
9      {
10         this->numOfStations = numOfStations;
11         this->numOfRoutes = numOfRoutes;
12         routes = new list<Route>[numOfStations];
13     }
14     Graph() {}

```

code snippet 14 Graph implementation.

```
1  int getNumOfStations()
2  {
3      return numOfStations;
4  }
5
6  int getNumOfRoutes()
7  {
8      return numOfRoutes;
9  }
10 list<Route> *getRoutes()
11 {
12     return routes;
13 }
```

code snippet 15 Graph implementation.

```
1  bool isAdjacent(int xId, int yId)
2  {
3      list<Route> adjacents = routes[xId];
4      for (Route adjacent : adjacents)
5      {
6          if (adjacent.dstStation.id == yId)
7              return true;
8      }
9      return false;
10 }
```

code snippet 16 Graph implementation.

```
1  void addRoute(Route route)
2  {
3      routes[route.srcStation.id].push_back(route);
4      routes[route.dstStation.id].push_back(Route(route.dstStation, route.srcStation, route.distance));
5  }
```

code snippet 17 Graph implementation.



```
1  int getNumOfAdjacents(int vertexId)
2  {
3      return routes[vertexId].size();
4  }
```

code snippet 18 Graph implementation.



```
1  int getDistance(int srcID, int dstID)
2  {
3      list<Route> adjacents = routes[srcID];
4      for (Route route : adjacents)
5      {
6          if (route.dstStation.id == dstID)
7              return route.distance;
8      }
9      return numeric_limits<int>::max();
10 }
```

code snippet 19 Graph implementation.

getAdjacents(int sourceStationID) returns a list of routes adjacent to a given station.



```
1  list<Route> getAdjacents(int srcID)
2  {
3      return routes[srcID];
4  }
```

code snippet 20 Graph implementation.

The printGraph function outputs the connections between stations in a human-readable format.

```
1  void printGraph()
2  {
3
4      for (int i = 0; i < numOfStations; i++)
5      {
6          char name = 'A' + i;
7          cout << "Station " << name << "---->";
8          int j = 0;
9          for (Route x : routes[i])
10         {
11             cout << "[" << x.dstStation.name << " - " << x.distance << "]";
12             j++;
13             if (j < routes[i].size())
14                 cout << "---->";
15         }
16
17         cout << endl;
18         cout << endl;
19     }
20 }
21 ;
```

code snippet 21 Graph implementation

1.2.3 Output: Generated data (Data Set 2)

Output snippet 2 shows information about different stations in a logistics scenario. Each station is identified with coordinates (x, y, z), a weight representing cargo, and a profit value.

===== STATIONS =====												
	Station		x		y		z		Weight		Profit	
	Station A		629		296		366		85		9	
	Station B		980		832		366		39		86	
	Station C		336		563		633		63		33	
	Station D		358		339		633		62		93	
	Station E		365		920		83		90		93	
	Station F		336		633		320		36		86	
	Station G		553		823		662		32		66	
	Station H		66		663		383		89		66	
	Station I		383		360		333		29		35	
	Station J		580		336		950		69		63	
	Station K		208		3		336		33		86	
	Station L		625		325		689		23		56	
	Station M		332		363		998		93		92	
	Station N		866		682		622		30		96	
	Station O		623		666		528		63		3	
	Station P		636		602		386		30		30	
	Station Q		933		806		966		33		66	
	Station R		223		660		392		33		56	
	Station S		868		933		338		85		33	
	Station T		958		96		363		50		83	

Output snippet 2 Generated Data (Data Set2).

1.2.4 Output: Generated Graph (Data Set 2)

Output snippet 3 displays the graph generated from the implementation of Data Set 2. It presents station names along with their connected stations and the corresponding route weights.

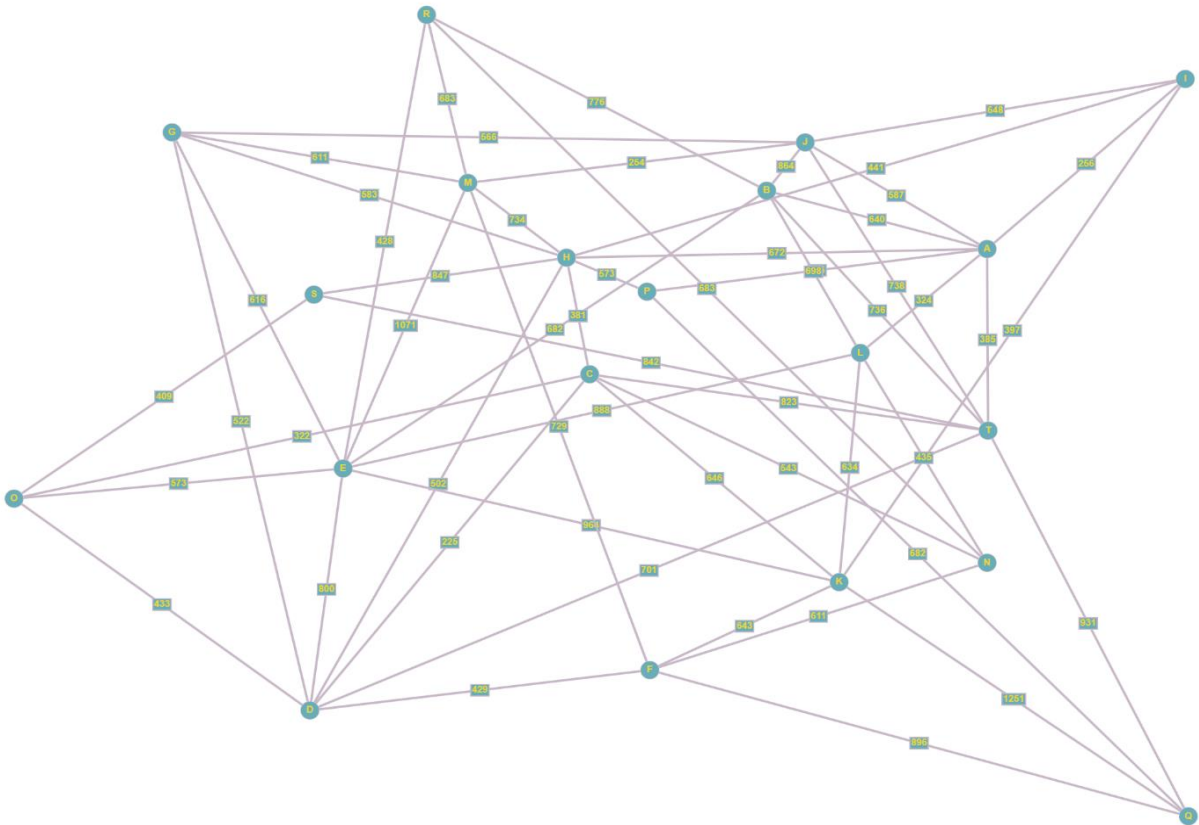
```

===== GENERATED GRAPH =====
Station A--->[ J - 587]--->[ I - 256]--->[ L - 324]--->[ B - 640]--->[ H - 672]--->[ P - 306]--->[ T - 385]
Station B--->[ A - 640]--->[ T - 736]--->[ R - 776]--->[ E - 682]--->[ L - 698]--->[ J - 864]
Station C--->[ T - 823]--->[ D - 225]--->[ K - 646]--->[ N - 543]--->[ O - 322]--->[ H - 381]
Station D--->[ C - 225]--->[ F - 429]--->[ T - 701]--->[ E - 800]--->[ G - 522]--->[ O - 433]--->[ H - 502]
Station E--->[ O - 573]--->[ D - 800]--->[ B - 682]--->[ G - 616]--->[ K - 964]--->[ R - 428]--->[ L - 888]--->[ M - 1071]
Station F--->[ D - 429]--->[ N - 611]--->[ Q - 896]--->[ K - 643]--->[ M - 729]
Station G--->[ M - 611]--->[ D - 522]--->[ E - 616]--->[ H - 583]--->[ J - 566]
Station H--->[ A - 672]--->[ S - 847]--->[ G - 583]--->[ I - 441]--->[ M - 734]--->[ P - 573]--->[ C - 381]--->[ D - 502]
Station I--->[ A - 256]--->[ J - 648]--->[ H - 441]--->[ K - 397]
Station J--->[ A - 587]--->[ I - 648]--->[ M - 254]--->[ B - 864]--->[ G - 566]--->[ T - 738]
Station K--->[ C - 646]--->[ F - 643]--->[ E - 964]--->[ L - 634]--->[ I - 397]--->[ Q - 1251]
Station L--->[ A - 324]--->[ K - 634]--->[ B - 698]--->[ E - 888]--->[ N - 435]
Station M--->[ G - 611]--->[ J - 254]--->[ H - 734]--->[ R - 683]--->[ F - 729]--->[ E - 1071]
Station N--->[ F - 611]--->[ C - 543]--->[ R - 683]--->[ L - 435]
Station O--->[ E - 573]--->[ S - 409]--->[ C - 322]--->[ D - 433]
Station P--->[ H - 573]--->[ A - 306]--->[ Q - 682]
Station Q--->[ F - 896]--->[ P - 682]--->[ T - 931]--->[ K - 1251]
Station R--->[ B - 776]--->[ N - 683]--->[ M - 683]--->[ E - 428]
Station S--->[ H - 847]--->[ O - 409]--->[ T - 842]
Station T--->[ B - 736]--->[ C - 823]--->[ D - 701]--->[ Q - 931]--->[ S - 842]--->[ A - 385]--->[ J - 738]

```

Output snippet 3 Generated Graph (Data Set 2).

1.2.5 Output: Generated Graph visualization (Data set 2)



2. Heap Sort and Merge Sort (Q2)

2.1 Heap Sort (HS)


2.1.1 Heap Sort Overview.

HS is a comparison-based sorting algorithm that uses a binary heap data structure to build a max-heap or min-heap. It then repeatedly extracts the root (maximum or minimum) element and rebuilds the heap until the entire array is sorted.

2.1.2 Source Code: Heap Sort with Explanation.

Includes necessary C++ libraries (iostream) and the custom dataset-related classes from the previous code (DataSet, etc.).

printArray function prints the elements of an array.



```
1  #include <iostream>
2  #include "../question1/dataSet1/dataSet1.cpp"
3  #include <chrono>
4  using namespace std;
5
6  void printArray(int A[], int n)
7  {
8      for (int i = 0; i < n; i++)
9          cout << A[i] << " ";
10     cout << endl;
11 }
```

code snippet 22 HS implementation

heapify function takes an array, its size, and an index (j).

Compares the current element with its left and right children.

Swaps the current element with the maximum of its children, ensuring the max-heap property.

```
1 void heapify(int A[], int arraySize, int j)
2 {
3     int max;
4     int left = 2 * j + 1;
5     int right = 2 * j + 2;
6     if (left < arraySize && A[left] > A[j])
7         max = left;
8     else
9         max = j;
10    if (right < arraySize && A[right] > A[max])
11        max = right;
12    if (max != j)
13    {
14        swap(A[j], A[max]);
15        heapify(A, arraySize, max);
16    }
17 }
```

code snippet 23 HS implementation

heapSort function builds a max-heap from the given array.

Repeatedly extracts the maximum element from the heap, places it at the end, and maintains the max-heap property.

Continues this process until the entire array is sorted.

```
1 void heapSort(int A[], int arraySize)
2 {
3     for (int j = (arraySize - 1) / 2; j >= 0; j--)
4         heapify(A, arraySize, j);
5     for (int i = arraySize - 1; i >= 1; i--)
6     {
7         swap(A[0], A[i]);
8         heapify(A, --arraySize, 0);
9     }
10 }
```

code snippet 24 HS implementation

In summary, the code provides an implementation of the Heap Sort algorithm, known for its time efficiency and space efficiency.

2.1.3 Output: Sorted data Using HS (Data Set 1)

Output snippet 4 shows that the data set 1 before and after using HS sort algorithm, as we can see the data set 1 is sorted in ascending order and the time taken for the sorting the set using this algorithm is 9 microseconds, we got this time by using the build in function for C++ to calculate the time until this algorithm executed.

```
C:\Users\12113\Desktop\code\code\question2\heapSort>heapSort
Set 1 (Before sorting) :
13 53 86 40 91 68 66 5 52 79 16 54 2 69 58 77 53 77 93 71 18 7 48 12 17 2 33 86 23 29 73 64 94 4 18 78 73 61 61 11 13 60 24 32 84 95 38 87 82 5 73 56 19 10 77 23 7 66 51 19 4 91 62 35 8 90 76 23 39 20 22 49 35 4
9 85 60 23 99 80 33 93 69 3 15 64 72 43 23 53 92 8 45 12 23 49 33 82 96 32 45

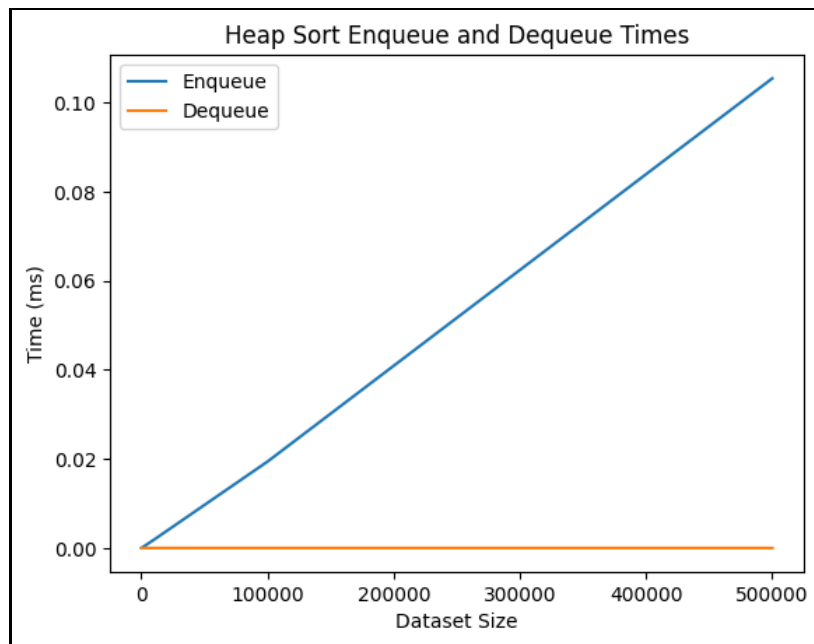
Time taken for sorting: 9 microseconds

set 1 (After sorting) :
2 2 3 4 4 5 5 7 7 8 8 10 11 12 12 13 13 15 16 17 18 18 19 20 22 23 23 23 23 23 24 29 32 32 33 33 33 35 35 38 39 40 43 45 45 48 49 49 49 51 52 53 53 53 54 56 58 60 60 61 61 62 64 64 66 66 68 69 69 71 72 73
73 73 76 77 77 77 78 79 80 82 82 84 85 86 86 87 90 91 91 92 93 93 94 95 96 99
```

Output snippet 4 Sorted data Using HS Algorithm.

2.1.4 Timing Graph of vs dataset size (In Milliseconds).

Here we have the graph of timing vs dataset size to have some experimental validation of the algorithm.



Output Area 2 provides a clearer view of the results obtained from Data Set 1. It showcases the data before and after applying the HS algorithm, offering additional clarity in case the details in the snippet are not immediately apparent.

Set 1 (Before sorting):

```
13 53 86 40 91 68 66 5 52 79 16 54 2 69 58 77 53 77 93 71 18 7 48 12 17 2 33 86 23 29 73
64 94 4 18 78 73 61 61 11 13 60 24 32 84 95 38 87 82 5 73 56 19 10 77 23 7 66 51 19 4 91
62 35 8 90 76 23 39 20 22 49 35 49 85 60 23 99 80 33 93 69 3 15 64 72 43 23 53 92 8 45 12
23 49 33 82 96 32 45
```

Time taken for sorting: 9 microseconds.

set 1 (After sorting):

```
2 2 3 4 4 5 5 7 7 8 8 10 11 12 12 13 13 15 16 17 18 18 19 19 20 22 23 23 23 23 23 23 24 29
32 32 33 33 33 35 35 38 39 40 43 45 45 48 49 49 49 51 52 53 53 53 54 56 58 60 60 61 61 62
64 64 66 66 68 69 69 71 72 73 73 73 76 77 77 77 78 79 80 82 82 84 85 86 86 87 90 91 91 92
93 93 94 95 96 99
```

Output Area 2 Sorted data Using HS Algorithm.

2.2 Merge Sort (MS)

2.2.1 Merge Sort Overview.

MS is a divide-and-conquer sorting algorithm that divides the array into two halves, sorts each half recursively, and then merges the sorted halves. It leverages the merging step to produce a sorted array.

2.2.2 Source Code: Merge Sort with Explanation

merge function takes an array and its left, middle, and right indices.

Creates two temporary arrays (leftArray and rightArray) to store the elements of the subarrays to be merged. Merges the two sorted subarrays back into the original array.

```
1
2 #include <iostream>
3 #include <vector>
4 #include "../question1/dataSet1/dataSet1.cpp"
5 #include <chrono>
6 #include <bits/stdc++.h>
7 using namespace std;
8
9 void merge(int array[], int const left, int const mid,
10           int const right)
11 {
12     int const subArrayOne = mid - left + 1;
13     int const subArrayTwo = right - mid;
14
15     auto *leftArray = new int[subArrayOne],
16         *rightArray = new int[subArrayTwo];
17
18     for (auto i = 0; i < subArrayOne; i++)
19         leftArray[i] = array[left + i];
20     for (auto j = 0; j < subArrayTwo; j++)
21         rightArray[j] = array[mid + 1 + j];
22
23     auto indexOfSubArrayOne = 0, indexOfSubArrayTwo = 0;
24     int indexOfMergedArray = left;
25
26     while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo)
27     {
28         if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo])
29         {
30             array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
31             indexOfSubArrayOne++;
32         }
33         else
34         {
35             array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
36             indexOfSubArrayTwo++;
37         }
38         indexOfMergedArray++;
39     }
40
41     while (indexOfSubArrayOne < subArrayOne)
42     {
43         array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
44         indexOfSubArrayOne++;
45         indexOfMergedArray++;
46     }
47
48     while (indexOfSubArrayTwo < subArrayTwo)
49     {
50         array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
51         indexOfSubArrayTwo++;
52         indexOfMergedArray++;
53     }
54     delete[] leftArray;
55     delete[] rightArray;
56 }
```

code snippet 25 MS implementation

mergeSort function recursively divides the array into two halves until each subarray has one element. Calls the merge function to merge the sorted subarrays.

```
1 void mergeSort(int array[], int const begin, int const end)
2 {
3     if (begin >= end)
4         return;
5
6     int mid = begin + (end - begin) / 2;
7     mergeSort(array, begin, mid);
8     mergeSort(array, mid + 1, end);
9     merge(array, begin, mid, end);
10 }
```

code snippet 26 MS implementation

printArray function prints the elements of an array.

```
1 void printArray(int A[], int size)
2 {
3     for (int i = 0; i < size; i++)
4         cout << A[i] << " ";
5     cout << endl;
6 }
```

code snippet 27 MS implementation

In summary, the code provides an implementation of the merge sort algorithm, which is known for its stability and guarantees $O(n \log n)$ time complexity for sorting.

2.2.3 Output: Sorted data Using MS (Data Set 1)

Output snippet 5 shows that the data set 1 before and after using MS sort algorithm, as we can see the data set 1 is sorted in ascending order and the time taken for the sorting the set using this algorithm is 42 microseconds, we got this time by using the build in function for C++ to calculate the time until this algorithm executed.

```
C:\Users\12113\Desktop\code\code\question2\mergeSort>mergeSort
Set 1 (Before sorting) :
13 53 86 40 91 68 66 5 52 79 16 54 2 69 58 77 53 77 93 71 18 7 48 12 17 2 33 86 23 29 73 64 94 4 18 78 73 61 61 11 13 60 24 32 84 95 38 87 82 5 73 56 19 10 77 23 7 66 51 19 4 91 62 35 8 90 76 23 39 20 22 49 35 4
9 85 60 23 99 80 33 93 69 3 15 64 72 43 23 53 92 8 45 12 23 49 33 82 96 32 45

Time taken for sorting: 42 microseconds

set 1 (After sorting) :
2 2 3 4 4 5 5 7 7 8 8 10 11 12 12 13 13 15 16 17 18 18 19 19 20 22 23 23 23 23 23 23 23 24 29 32
32 33 33 33 35 35 38 39 40 43 45 45 48 49 49 49 51 52 53 53 53 54 56 58 60 60 61 61 62 64 64
64 66 66 68 69 69 71 72 73 73 76 77 77 77 78 79 80 82 82 84 85 86 86 87 90 91 91 92 93
93 94 95 96 99
```

Output snippet 5 Sorted data Using Ms algorithm.

Output Area 3 provides a clearer view of the results obtained from Data Set 1. It showcases the data before and after applying the MS algorithm, offering additional clarity in case the details in the snippet are not immediately apparent.

Set 1 (Before sorting):

13 53 86 40 91 68 66 5 52 79 16 54 2 69 58 77 53 77 93 71 18 7 48 12 17 2 33 86 23 29 73 64
94 4 18 78 73 61 61 11 13 60 24 32 84 95 38 87 82 5 73 56 19 10 77 23 7 66 51 19 4 91 62 35
8 90 76 23 39 20 22 49 35 49 85 60 23 99 80 33 93 69 3 15 64 72 43 23 53 92 8 45 12 23 49
33 82 96 32 45

Time taken for sorting: 42 microseconds.

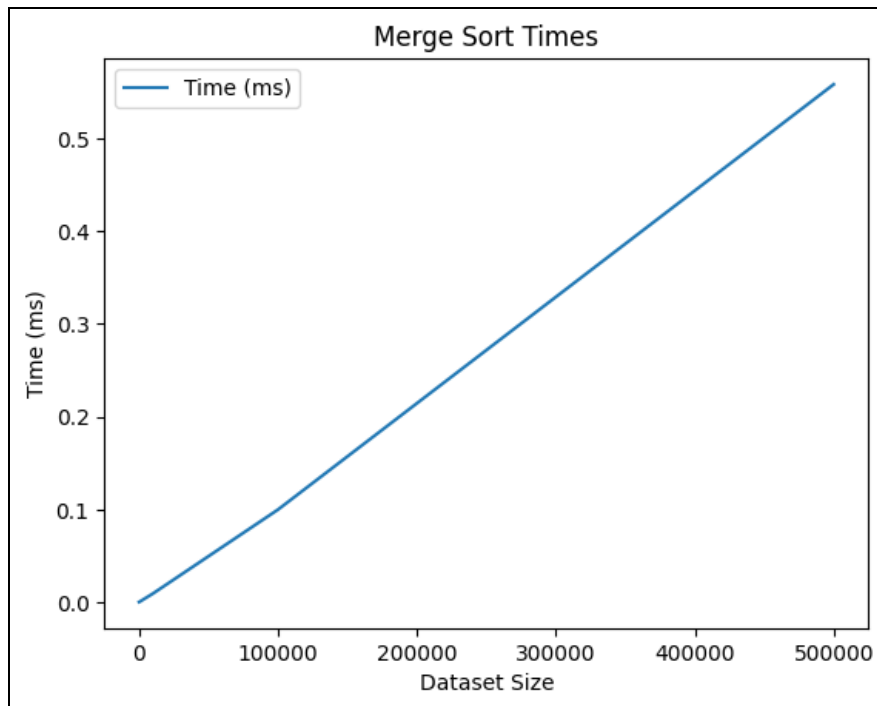
set 1 (After sorting):

2 2 3 4 4 5 5 7 7 8 8 10 11 12 12 13 13 15 16 17 18 18 19 19 20 22 23 23 23 23 23 23 24 29 32
32 33 33 33 35 35 38 39 40 43 45 45 48 49 49 49 51 52 53 53 53 54 56 58 60 60 61 61 62 64
64 66 66 68 69 69 71 72 73 73 76 77 77 77 78 79 80 82 82 84 85 86 86 87 90 91 91 92 93
93 94 95 96 99

Output Area 3 Sorted data Using MS Algorithm.

2.2.4 graph of timing vs dataset size (In Milliseconds):

Here we have the graph of timing vs dataset size to have some experimental validation of the algorithm.



3. Shortest Paths and Minimum Spanning Tree (Q3):

3.1 Shortest Paths (SP)

3.1.1 Dijkstra's algorithm Overview.

Dijkstra's algorithm, named after Dutch computer scientist Edsger Dijkstra, is a widely used graph search algorithm designed to find the shortest path between two vertices in a weighted graph. This algorithm is particularly effective for solving the single-source shortest path (SP) problem, where the goal is to determine the shortest path from a designated source node to all other nodes in the graph.

3.1.2 Source Code (SP using Dijkstra's algorithm with Explanation)

Defines a custom comparator **class (RouteComparator)** to compare routes based on their distances. It's used for the priority queue in Dijkstra's algorithm.


```

1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <map>
5  #include <set>
6  #include <unordered_set>
7  #include "../question1/DataSet2/DataSet2.cpp"
8  // #include "../question1/DataSet2/Graph.cpp"
9  #define INF 0x3f3f3f3f
10 using namespace std;
11
12 class RouteComparator
13 {
14 public:
15     bool operator()(const Route &route1, const Route &route2)
16     {
17         return route1.distance > route2.distance;
18     }
19 };

```

code snippet 28 Dijkstra's algorithm implementation

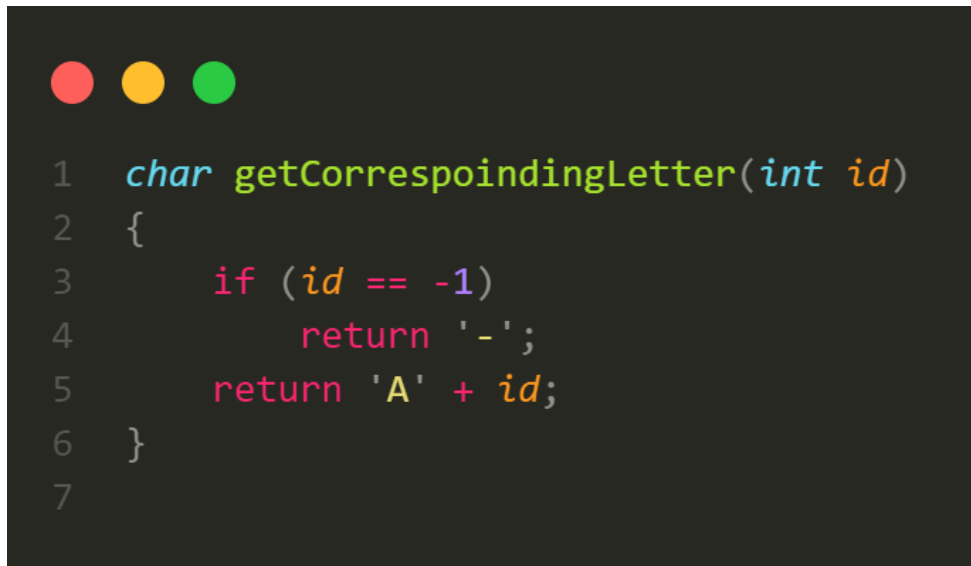
findShortestPaths Function:

Takes a source station, a graph, and an array to store parent stations. Utilizes Dijkstra's algorithm to find the shortest paths from the source station to all other stations. Uses a priority queue (pq) to efficiently select the station with the minimum distance. Updates distances and parent stations based on the calculated shortest paths. Returns vector containing distances from the source station to all other stations.

```
1 // Function to find the shortest paths from a source station to all other stations
2 // using Dijkstra's algorithm
3
4 vector<int> findShortestPaths(int srcStation, Graph &graph, int (&parent)[])
5 {
6     // Get the number of stations in the graph
7     int numOfStations = graph.getNumOfStations();
8
9     // Priority queue to store routes based on their distances
10    // RouteComparator is a custom comparator to prioritize routes with shorter distances
11    priority_queue<Route, vector<Route>, RouteComparator> pq;
12
13    // Vector to store distances from the source station to all other stations
14    vector<int> distances(numOfStations, INF);
15
16    // Set to keep track of visited stations
17    unordered_set<int> visited;
18
19    // Initialize the parent array with -1 for all stations
20    parent[srcStation] = -1;
21
22    // Push the source station into the priority queue with distance 0
23    pq.push(Route(Station(srcStation), Station(srcStation), 0));
24
25    // Set distance of source station to itself as 0
26    distances[srcStation] = 0;
27
28    // Loop until all stations are visited
29    while (visited.size() < numOfStations)
30    {
31        // Extract the station with the minimum distance from the priority queue
32        int minDistance = pq.top().dstStation.id;
33        pq.pop();
34
35        // If the station is already visited, skip it
36        if (visited.count(minDistance) > 0)
37            continue;
38
39        // Get adjacent stations of the current station
40        auto adjacents = graph.getAdjacents(minDistance);
41
42        // Iterate through adjacent stations
43        for (const auto &adjacent : adjacents)
44        {
45            int adjID = adjacent.dstStation.id;
46
47            // Calculate new distance from source to the adjacent station through the current station
48            int newDistance = distances[minDistance] + adjacent.distance;
49
50            // If the adjacent station is not visited and the new distance is shorter
51            // than the previously known distance, update the distance and add it to the priority queue
52            if (visited.count(adjID) == 0 && newDistance < distances[adjID])
53            {
54                distances[adjID] = newDistance;
55                pq.push(Route(adjacent.srcStation, adjacent.dstStation, newDistance));
56
57                // Update parent of the adjacent stations
58                parent[adjID] = minDistance;
59            }
60        }
61        // Mark the current station as visited
62        visited.insert(minDistance);
63    }
64
65    // Return the vector of distances from the source station to all other stations
66    return distances;
67 }
```

code snippet 29 Dijkstra's algorithm implementation

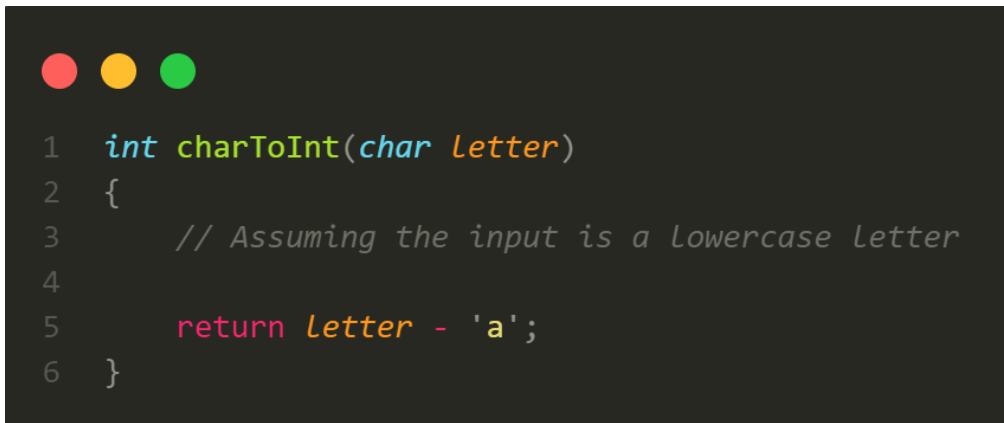
getCorrespondingLetter: Maps station IDs to corresponding letters ('A' for 0, 'B' for 1, etc.).



```
1 char getCorrespondingLetter(int id)
2 {
3     if (id == -1)
4         return '-';
5     return 'A' + id;
6 }
7
```

code snippet 30 Dijkstra's algorithm implementation

charToInt: Converts a lowercase letter to its corresponding integer value (assuming the input is a lowercase letter).



```
1 int charToInt(char letter)
2 {
3     // Assuming the input is a lowercase letter
4
5     return letter - 'a';
6 }
```

code snippet 31 Dijkstra's algorithm implementation

In summary, the code provides a function to find the shortest paths from a source station to all other stations in a graph using Dijkstra's algorithm. It also includes helper functions for mapping station IDs to letters and converting characters to integers.

3.1.3 Output: SP using Dijkstra's algorithm (Data Set 2)

Output snippet 6 unveils the outcomes of applying the Shortest Path (SP) algorithm to the dataset generated from Data Set 2. Within this output, you'll find the identification of the source station and the corresponding shortest paths leading to every other station. This provides a comprehensive overview of the efficient routes originating from the source station to each destination within the dataset.

```
Enter source station : A

===== shortest paths from station a=====
Name: A, Shortest distance is 0, Shortest path : A
Name: B, Shortest distance is 640, Shortest path : A=>B
Name: C, Shortest distance is 1053, Shortest path : A=>H=>C
Name: D, Shortest distance is 1086, Shortest path : A=>T=>D
Name: E, Shortest distance is 1212, Shortest path : A=>L=>E
Name: F, Shortest distance is 1296, Shortest path : A=>I=>K=>F
Name: G, Shortest distance is 1153, Shortest path : A=>J=>G
Name: H, Shortest distance is 672, Shortest path : A=>H
Name: I, Shortest distance is 256, Shortest path : A=>I
Name: J, Shortest distance is 587, Shortest path : A=>J
Name: K, Shortest distance is 653, Shortest path : A=>I=>K
Name: L, Shortest distance is 324, Shortest path : A=>L
Name: M, Shortest distance is 841, Shortest path : A=>J=>M
Name: N, Shortest distance is 759, Shortest path : A=>L=>N
Name: O, Shortest distance is 1375, Shortest path : A=>H=>C=>O
Name: P, Shortest distance is 306, Shortest path : A=>P
Name: Q, Shortest distance is 988, Shortest path : A=>P=>Q
Name: R, Shortest distance is 1416, Shortest path : A=>B=>R
Name: S, Shortest distance is 1227, Shortest path : A=>T=>S
Name: T, Shortest distance is 385, Shortest path : A=>T
```

Output snippet 6 SP using Dijkstra's algorithm (Data Set 2).

3.1.4 Output: SP Graph Representation using Dijkstra's algorithm (Data Set 2)

Output snippet 7 unveils the graph generated by leveraging the outcomes of the Shortest Path (SP) algorithm presented in output snippet 6. This graph encapsulates the intricate connections established through the algorithm, offering a kind of visual representation of the shortest paths from the source station to every other station in the dataset.

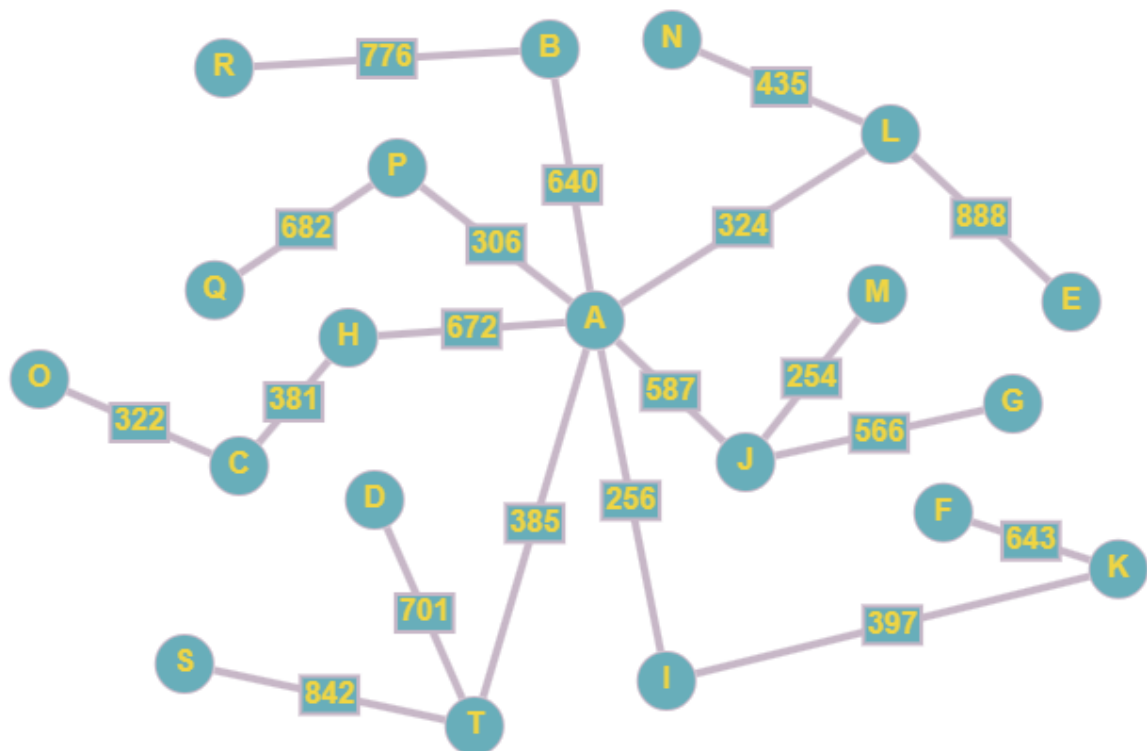
```

===== Graph representing the shortest paths =====
A, parent => -
B, parent => A
C, parent => H
D, parent => T
E, parent => L
F, parent => K
G, parent => J
H, parent => A
I, parent => A
J, parent => A
K, parent => I
L, parent => A
M, parent => J
N, parent => L
O, parent => C
P, parent => A
Q, parent => P
R, parent => B
S, parent => T
T, parent => A

```

Output snippet 7 Graph represents SP using Dijkstra's algorithm (Data Set 2)

3.1.5 Output: SP Graph Visualization using Dijkstra's algorithm (Data Set 2)



3.2 Minimum Spanning Tree (MST)

3.2.1 Kruskal's algorithm Overview.

Kruskal's algorithm is a greedy algorithm that finds the minimum spanning tree (MST) for a connected, undirected graph. The minimum spanning tree is a subgraph that includes all the vertices of the original graph and satisfies two properties:

1. It is a tree, which means it must be acyclic and connected.
2. The sum of the weights of its edges is minimized.

3.2.2 Source Code and Explanation: MST using Kruskal's Algorithm.

Defines a structure node representing a subset in a disjoint set data structure, used for Union-Find operations in Kruskal's algorithm.

Declares a global vector dsuf of type node to represent the disjoint set data structure.

Implements a comparator function (comparator) for sorting routes based on distance.

```

1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <map>
5  #include <set>
6  #include "../question1/DataSet2/DataSet2.cpp"
7
8  #include <bits/stdc++.h>
9  using namespace std;
10
11 struct node
12 {
13     int parent;
14     int rank;
15 };
16
17 vector<node> dsuf;
18
19 bool comparator(Route a, Route b)
20 {
21     return a.distance < b.distance;
22 }
```

code snippet 32 Kruskal's algorithm implementation

setRoutesList function populates a vector **routesList** with all routes from the given graph.

```
1 // setRoutesList
2 void setRoutesList(Graph graph, vector<Route> &routesList, int numOfStations)
3 {
4     list<Route> *routes = graph.getRoutes();
5     for (int i = 0; i < numOfStations; i++)
6     {
7         for (Route route : routes[i])
8         {
9             routesList.push_back(route);
10        }
11    }
12 }
```

code snippet 33 Kruskal's algorithm implementation

setDsuf initializes the disjoint set data structure with each station as a separate subset.

```
1 void setDsuf(int numOfStations)
2 {
3     dsuf.resize(numOfStations); // Mark all vertices as separate subsets with only 1 element
4     for (int i = 0; i < numOfStations; ++i) // Mark all nodes as independent set
5     {
6         dsuf[i].parent = -1;
7         dsuf[i].rank = 0;
8     }
9 }
```

code snippet 34 Kruskal's algorithm implementation

find function implements the FIND operation in Union-Find with path compression to find the absolute parent of a subset.

```
1 // FIND operation
2 int find(int v)
3 {
4     if (dsuf[v].parent == -1)
5         return v;
6     return dsuf[v].parent = find(dsuf[v].parent); // Path Compression
7 }
8
```

code snippet 35 Kruskal's algorithm implementation

union_op function performs the UNION operation in Union-Find based on rank to merge two subsets.

```
1 void union_op(int fromP, int toP)
2 {
3
4     // UNION by RANK
5     if (dsuf[fromP].rank > dsuf[toP].rank) // fromP has higher rank
6         dsuf[toP].parent = fromP;
7     else if (dsuf[fromP].rank < dsuf[toP].rank) // toP has higher rank
8         dsuf[fromP].parent = toP;
9     else
10    {
11        // Both have same rank and so anyone can be made as parent
12        dsuf[fromP].parent = toP;
13        dsuf[toP].rank += 1; // Increase rank of parent
14    }
15 }
```

code snippet 36 Kruskal's algorithm implementation

findMst function implements Kruskal's algorithm to find the Minimum Spanning Tree (MST) of the given graph. It sorts all routes by distance, initializes the disjoint set, and iterates through sorted routes to construct the MST. It uses Union-Find operations to avoid cycles and keeps track of the total MST cost. The function returns the total cost of the Minimum Spanning Tree.

```
1 int findMst(Graph &graph, Graph &mst)
2 {
3     int numOfStations = graph.getNumOfStations();
4     int numOfRoutes = graph.getNumOfRoutes();
5
6     int mstCost = 0; // keep track of minimum spanning tree cost
7     vector<Route> routesList; // vector to store all the routes
8     setRoutesList(graph, routesList, numOfStations);
9
10    setDsuf(numOfStations);
11    sort(routesList.begin(), routesList.end(), comparator); // sort all routes by distance
12
13    int i = 0, j = 0;
14    while (i < numOfStations - 1 && j < numOfRoutes * 2)
15    {
16        int fromP = find(routesList[j].srcStation.id); // FIND absolute parent of subset
17        int toP = find(routesList[j].dstStation.id); // FIND absolute parent of subset
18
19        if (fromP == toP) // if both are equal then we would ignore this route because adding it will create a cycle
20        {
21            ++j;
22            continue;
23        }
24
25        // UNION operation
26        union_op(fromP, toP); // UNION of 2 sets
27        mst.addRoute(routesList[j]); // add this route to the minimum spanning tree
28        mstCost += routesList[j].distance; // increase the minimum spanning tree cost
29        ++i;
30        ++j;
31    }
32
33    return mstCost;
34 }
```

code snippet 37 Kruskal's algorithm implementation

In summary, this code efficiently finds the Minimum Spanning Tree of a graph using Kruskal's algorithm, considering the weights (distances) of the routes. The Union-Find data structure is employed to manage subsets and avoid cycles during the MST construction.

3.2.3 Output: MST Using Kruskal's algorithm (Data Set 2)

Output snippet 8 shows the results of applying the MST algorithm to the data that generated in set 2, the output snippet 8 shows the generated graph that could be made with the minimum costs and without forming a cycle.

```
===== MINIMUM SPANNING TREE =====
Station A--->[ I - 256]--->[ P - 306]--->[ L - 324]--->[ T - 385]--->[ B - 640]
Station B--->[ A - 640]
Station C--->[ D - 225]--->[ O - 322]--->[ H - 381]
Station D--->[ C - 225]--->[ F - 429]--->[ G - 522]
Station E--->[ R - 428]--->[ O - 573]
Station F--->[ D - 429]
Station G--->[ D - 522]--->[ J - 566]
Station H--->[ C - 381]--->[ I - 441]
Station I--->[ A - 256]--->[ K - 397]--->[ H - 441]
Station J--->[ M - 254]--->[ G - 566]
Station K--->[ I - 397]
Station L--->[ A - 324]--->[ N - 435]
Station M--->[ J - 254]
Station N--->[ L - 435]
Station O--->[ C - 322]--->[ S - 409]--->[ E - 573]
Station P--->[ A - 306]--->[ Q - 682]
Station Q--->[ P - 682]
Station R--->[ E - 428]
Station S--->[ O - 409]
Station T--->[ A - 385]

MINIMUM SPANNING TREE COST : 7975
```

Output snippet 8 MST using Kruskal's Algorithm (Data Set 2).

4.1.2 Source Code: Knapsack 0/1(Dynamic Programming).

```

1 int Knapsack(vector<Station> &stations, vector<vector<int>> &matrix, int ansr[], int capacity)
2 {
3     int rows = stations.size() + 1; // Number of rows in the matrix
4     int columns = capacity + 1; // Number of columns in the matrix
5
6     for (int i = 0; i < columns; i++)
7         matrix[0][i] = 0;
8
9     // Iterate through each station and each possible capacity
10    for (int i = 1; i < rows; i++)
11    {
12        for (int w = 0; w < columns; w++)
13        {
14            if (i == 0 || w == 0)
15                matrix[i][w] = 0; // Base case: no items or no capacity left
16
17            else if (stations[i - 1].weight > w)
18                matrix[i][w] = matrix[i - 1][w]; // Cannot include the current station
19
20            else
21                matrix[i][w] = max(matrix[i - 1][w], matrix[i - 1][w - stations[i - 1].weight] + stations[i - 1].profit); // Maximum value including or excluding the current station
22        }
23    }
24
25    // Determine the winning stations based on the calculated matrix
26    setWinningStations(rows, columns, matrix, stations, ansr);
27
28    // Determine the winning stations based on the calculated matrix
29    int maxProfit = matrix[rows - 1][columns - 1];
30    return maxProfit;
31 }

```

code snippet 38 Knapsack implementation

```

1 bool compareByWeight(const Station &a, const Station &b)
2 {
3     return a.weight < b.weight;
4 }
5
6 void setWinningStations(int rows, int columns, vector<vector<int>> &matrix, vector<Station> &stations, int ansr[])
7 {
8     int n = rows - 1; // Total number of stations
9     int i = rows - 1, w = columns - 1; // Initializing pointers for traversing the matrix
10    int arri = stations.size() - 1; // Initializing index to traverse stations vector from the end
11    int arri = stations.size() - 1;
12    while (i >= 1)
13    {
14        // Check if the value in the current cell is equal to the value in the cell above it
15        if (matrix[i][w] == matrix[i - 1][w]) // Station is not included in the optimal solution
16            ansr[stations[arri].id] = 0;
17        else
18        {
19            ansr[stations[arri].id] = 1; // Station is included in the optimal solution
20            w = w - stations[arri].weight; // Update the remaining capacity
21        }
22
23        i--; // Move to the row above
24        arri--; // Move to the previous station
25    }
26 }

```

code snippet 39 Knapsack implementation

4.1.3 Source Code Explanation (Knapsack 0/1(Dynamic Programming)).

Knapsack function solves the 0/1 Knapsack Problem using dynamic programming. Given stations with weights and profits, it determines the optimal subset of stations to maximize profit without exceeding a given capacity.

setWinningStations sets an array to represent which stations are included in the optimal knapsack solution.

In summary, the code solves the 0/1 Knapsack Problem to maximize profit while considering weight constraints.

4.1.4 Output: available Stations (Knapsack 0\1(Dynamic Programming))

```

===== Treasure hunt project =====

You have these stations that you are allowd to visit :

=====
|| Station || Weight || Profit ||
=====
|| Station a || 85 || 9 ||
=====
|| Station b || 39 || 86 ||
=====
|| Station c || 63 || 33 ||
=====
|| Station d || 62 || 93 ||
=====
|| Station e || 90 || 93 ||
=====
|| Station f || 36 || 86 ||
=====
|| Station g || 32 || 66 ||
=====
|| Station h || 89 || 66 ||
=====
|| Station i || 29 || 35 ||
=====
|| Station j || 69 || 63 ||
=====
|| Station k || 33 || 86 ||
=====
|| Station l || 23 || 56 ||
=====
|| Station m || 93 || 92 ||
=====
|| Station n || 30 || 96 ||
=====
|| Station o || 63 || 3 ||
=====
|| Station p || 30 || 30 ||
=====
|| Station q || 33 || 66 ||
=====
|| Station r || 33 || 56 ||
=====
|| Station s || 85 || 33 ||
=====
|| Station t || 50 || 83 ||
=====

```

Output snippet 9 Available Stations Using Knapsack Algorithm (Data Set 2).

4.1.5 Output: Stations to be visited (Knapsack 0\1(Dynamic Programming))

Since you can only carry 800kg of profit, the maximum profit you can get is 1153
In order to get this maximum profit you need to visit these stations :

Station	Weight	Profit
Station b	39	86
Station d	62	93
Station e	90	93
Station f	36	86
Station g	32	66
Station h	89	66
Station i	29	35
Station j	69	63
Station k	33	86
Station l	23	56
Station m	93	92
Station n	30	96
Station p	30	30
Station q	33	66
Station r	33	56
Station t	50	83

Output snippet 10 Stations to be visited Using Knapsack Algorithm (Data Set 2).

4.1.6 Output: Fastest path to winning stations(Knapsack 0\1(Dynamic Programming))

Input your home station to view the fastest path from your home station to the winning stations : A

===== Fastest path to winning stations =====

A=>I=>K=>L=>N=>F=>D=>H=>P=>Q=>T=>B=>E=>R=>M=>J=>G

===== Good luck with the treasure hunt =====

Output snippet 11 Fastest path to winning stations Using Knapsack Algorithm (Data Set 2).

5 Time complexity

5.1 Data Set 2 (Time complexity)

```
// Function to design the dataset
void generateDataSet()
{
    unordered_map<int, vector<Station>> candidates; // the key represent the
    initializeCandidates(candidates);
    int routesCounter = 0;
    // Ensure each station has at least 3 connections
    for (int i = 0; i < numOfStations; i++)
    {
        int numOfAdjacents = graph.getNumOfAdjacents(stations[i].id);

        for (int j = numOfAdjacents; j < 3; j++, routesCounter++)
            addRoute(stations[i], candidates);
    }

    // Add the remaining routes
    for (int i = 0; routesCounter < numOfRoutes; routesCounter++, i++)
        addRoute(stations[i % numOfStations], candidates);
}
```

code snippet 40 Dataset 2 data generation

To analyze the time complexity of this algorithm, we'll begin by examining the initializeCandidates and addRoute functions, as they are integral components of the algorithm.

Data Set 2 (initializeCandidates)

```
// Function to initialize the candidates map
void initializeCandidates(unordered_map<int, vector<Station>> &candidates)
{
    for (int i = 0; i < numOfStations; ++i)
    {
        candidates[stations[i].id] = stations;
        removeFromCandidates(candidates[stations[i].id], i); // Remove the station itself from the candidates
    }
}
```

code snippet 41 Dataset 2 Candidates Initializing

In this function we iterate over all the stations and, for each station, assign the entire vector of stations to the candidates, and then remove the station itself from its own list of candidates.

The time complexity of this function can be expressed as follows:

Loop Over Stations:

```
for (int i = 0; i < numOfStations; ++i)
```

In this loop, you iterate through all stations once. The time complexity of this part is $O(\text{numOfStations})$.

Assign and Remove Operations:

```
candidates[stations[i].id] = stations;
```

```
removeFromCandidates(candidates[stations[i].id], i);
```

Assigning the entire vector of stations to `candidates[stations[i].id]` takes $O(\text{numOfStations})$ time, and removing the station itself is performed in constant time.

Considering both parts, the overall time complexity of the `initializeCandidates` function is $O(\text{numOfStations} * \text{numOfStations})$ which is $O(\text{numOfStations}^2)$.

initializeCandidates (Time Complexity Table)

Algorithm	Number of primitive operations
for (int i = 0; i < numOfStations; ++i)	$O(\text{numOfStations})$
candidates[stations[i].id] = stations;	$O(\text{numOfStations}) * O(\text{numOfStations})$
removeFromCandidates(candidates[stations[i].id], i);	$O(1) * O(\text{numOfStations})$
T(N)	$O(\text{numOfStations})^2$

Data Set 2(AddRoute Function)

```
// Function to add a route between two stations
void addRoute(Station x, unordered_map<int, vector<Station>> &candidates)
{
    int stationYIndex = getRandomIndex(candidates[x.id].size());
    Station y = candidates[x.id][stationYIndex]; // get random station to create a route with Station x
    int distance = calculateDistance(x, y);

    graph.addRoute(Route(x, y, distance));
    // Update candidates list to avoid choosing the same station again
    removeFromCandidates(candidates[x.id], stationYIndex);
    removeFromCandidates(candidates[y.id], getIndexById(candidates[y.id], x.id));
}
```

code snippet 42 Dataset 2 Route Adding

The first five lines of code execute in constant time $O(1)$. However, the last line's time complexity is $O(\text{numOfStations})$ because it involves finding the index of a station based on its ID. This operation necessitates iterating through the entire list of candidates until the index of the target station is found.

AddRoute Function (Time Complexity Table)

Algorithm	Number of primitive operations
int stationYIndex = getRandomIndex(candidates[x.id].size());	O(1)
Station y = candidates[x.id][stationYIndex];	O(1)
int distance = calculateDistance(x, y);	O(1)
graph.addRoute(Route(x, y, distance));	O(1)
removeFromCandidates(candidates[x.id], stationYIndex);	O(1)
removeFromCandidates(candidates[y.id], getIndexById(candidates[y.id], x.id));	O(numOfStations)
T(n)	O(numOfStations)

Data Set 2 (main algorithm)

```
// Function to design the dataset
void generateDataSet()
{
    unordered_map<int, vector<Station>> candidates; // the key represent the
    initializeCandidates(candidates);
    int routesCounter = 0;
    // Ensure each station has at least 3 connections
    for (int i = 0; i < numOfStations; i++)
    {
        int numOfAdjacents = graph.getNumOfAdjacents(stations[i].id);

        for (int j = numOfAdjacents; j < 3; j++, routesCounter++)
            addRoute(stations[i], candidates);
    }

    // Add the remaining routes
    for (int i = 0; routesCounter < numOfRoutes; routesCounter++, i++)
        addRoute(stations[i % numOfStations], candidates);
}
```

code snippet 43 Dataset 2 Main Algorithm

Loop Over Stations:

```
for (int i = 0; i < numOfStations; ++i)
```

In this loop, you iterate through all stations once. The time complexity of this part is $O(\text{numOfStations})$.

Getting number of Adjacent Stations:

```
int numOfAdjacents = graph.getNumOfAdjacents(stations[i].id);
```

this line would take a constant time which is $O(1)$

making sure that each station is connected to at least 3 routes:

```
for (int j = numOfAdjacents; j < 3; j++, routesCounter++)
```

```
    addRoute(stations[i], candidates);
```

In the worst case, each station needs to establish connections until it has at least 3 adjacents. The worst-case scenario for the inner loop is when a station initially has 0 adjacents, so it needs to add 3 routes. The addRoute function takes $O(\text{numOfStations})$ combining this with and would give us $O(\text{numOfStations} * 1 * 1 * \text{numOfStations})$, which simplifies to $O(\text{numOfStations}^2)$.

Loop for Adding the Remaining Routes:

```
for (int i = 0; routesCounter < numOfRoutes; routesCounter++, i++)
```

```
    addRoute(stations[i % numOfStations], candidates);
```

this loop continues until 'routesCounter' reaches 'numOfRoutes'. Within each iteration, it adds a route. Executing the loop until 'routesCounter' reaches 'numOfRoutes' takes $O(\text{numOfRoutes})$. Additionally, since the 'addRoute' function operates in $O(\text{numOfStations})$, the worst-case time complexity of this loop is $O(\text{numOfRoutes} * \text{numOfStations})$

Conclusion:

So the whole time complexity for the generateDataSet function would be :

$O(\text{numOfStations}^2) + O(\text{numOfStations}^2) + O(\text{numOfRoutes} * \text{numOfStations})$ which simplifies to

$O((\text{numOfStations}^2) + (\text{numOfRoutes} * \text{numOfStations}))$

Data Set 2 (Time complexity Table)

Algorithm	Number of primitive operations
<code>unordered_map<int, vector<Station>> candidates;</code>	$O(1)$
<code>initializeCandidates(candidates);</code>	$O(\text{numOfStations}^2)$
<code>int routesCounter = 0;</code>	$O(1)$
<code>for (int i = 0; i < numOfStations; i++)</code>	$O(\text{numOfStations})$
<code>int numOfAdjacents = graph.getNumOfAdjacents(stations[i].id);</code>	$O(\text{numOfStations})$
<code>for (int j = numOfAdjacents; j < 3; j++, routesCounter++)</code>	$O(1)$
<code>addRoute(stations[i], candidates);</code>	$O(\text{numOfStations})$
<code>for (int i = 0; routesCounter < numOfRoutes; routesCounter++, i++)</code>	$O(\text{numOfRoutes})$
<code>addRoute(stations[i % numOfStations], candidates);</code>	$O(\text{numOfRoutes} * \text{numOfStations})$
$T(N)$	$O(\text{numOfStations}^2 + (\text{numOfRoutes} * \text{numOfStations}))$

5.2 Heap sort

Considering these trends, analyzing it helps us understand how complex Heap Sort is in terms of time. The consistent timing in dequeue operation, even as the dataset grows, indicates an efficient and smooth performance with a time complexity of $O(n \log n)$. This matches the expected efficiency of Heap Sort, making it a strong algorithm for sorting large datasets effectively.

5.3 Merge Sort

The observed upward trend in timing suggests that Merge Sort operates with a time complexity that might be higher than linear. Merge Sort's time complexity is typically $O(n \log n)$, which is efficient for large datasets. However, the increasing trend implies that the algorithm's performance becomes more apparent as the dataset size grows.

5.3 Kruskals algorithm

```
int findMst(Graph &graph, Graph &mst)
{
    int numOfStations = graph.getNumOfStations();
    int numOfRoutes = graph.getNumOfRoutes();

    int mstCost = 0;           // keep track of minimum spanning tree cost
    vector<Route> routesList; // vector to store all the routes
    setRoutesList(graph, routesList, numOfStations);

    setDsuf(numOfStations);
    sort(routesList.begin(), routesList.end(), comparator); // sort all routes by distance

    int i = 0, j = 0;
    while (i < numOfStations - 1 && j < numOfRoutes * 2)
    {
        int fromP = find(routesList[j].srcStation.id); // FIND absolute parent of subset
        int toP = find(routesList[j].dstStation.id);  // FIND absolute parent of subset

        if (fromP == toP) // if both are equal then we would ignore this route because adding it will create a cycle
        {
            ++j;
            continue;
        }

        // UNION operation
        union_op(fromP, toP);           // UNION of 2 sets
        mst.addRoute(routesList[j]);    // add this route to the minimum spanning tree
        mstCost += routesList[j].distance; // increase the minimum spanning tree cost
        ++i;
        ++j;
    }

    return mstCost;
}
```

code snippet 44 Kruskals algorithm

sorting the routes: we are sorting the edges/routes based on their weights/distance. This sorting operation typically takes $O(E \log E)$ time, where E is the number of edges/routes in the graph.

Initializing DSUF (Disjoint Set Union-Find) data structure: we're initializing the data structure to manage disjoint sets. This operation takes $O(V)$ time, where V is the number of vertices/stations in the graph.

While loop : this while loop will run at most for $O(E * 2)$ which simplifies to $O(E)$ where E is the number of edges / routes in the graph.

Finding the absolute parent of a subset (find operation): we are using path compression which makes the time complexity of the find operation almost constant. The worst-case time complexity for find operation with path compression is $O(\log V)$, where V is the number of vertices. Since the find operation is inside the loop and the loop runs for $O(E)$ then the overall time complexity for this line is $O(E \log V)$.

Union operation: since we are just retrieving one element from an array and updating its rank / parent The time complexity of the union operation is $O(1)$. Since the union operation is inside the loop and the loop runs for $O(E)$ then the time complexity would be $O(E * 1)$ which simplifies to $O(E)$

conclusion

Combining all of the above we have $O(E \log E) + O(V) + O(E \log V) + O(E)$.

since E at most is V^2 for a dense graph, the term $O(E \log E)$ dominates. Therefore, the overall time complexity for the implemented Kruskals algorithm is $O(E \log E)$.

Kruskals algorithm (Time Complexity Table)

Algorithm	Number of primitive operations
<code>int numOfStations = graph.getNumOfStations();</code>	1
<code>int numOfRoutes = graph.getNumOfRoutes();</code>	1
<code>int mstCost = 0;</code>	1
<code>vector<Route> routesList;</code>	$O(E)$ where E is the number of routes in the graph
<code>setDsuf(numOfStations);</code>	$O(V)$ where V is the number of stations
<code>sort(routesList.begin(), routesList.end(), comparator);</code>	$O(E \log E)$
<code>int i = 0, j = 0;</code>	$O(1)$
<code>while (i < numOfStations - 1 && j < numOfRoutes * 2)</code>	$O(E)$
<code>int fromP = find(routesList[j].srcStation.id);</code>	$O(E \log V)$
<code>if (fromP == toP)</code>	$O(E)$
<code>++j;</code>	$O(E)$
<code>continue;</code>	$O(E)$
<code>union_op(fromP, toP);</code>	$O(E)$
<code>mst.addRoute(routesList[j]);</code>	$O(E)$
<code>mstCost += routesList[j].distance;</code>	$O(E)$
<code>++i; ++j;</code>	$O(E)$
<code>return mstCost;</code>	$O(1)$
T(N)	$O(E \log E)$

5.4 Dijkstra's Algorithm

Initialization:

- . Initializing the priority queue: $O(1)$
- . Initializing the distances vector: $O(V)$, where V is the number of stations
- . Initializing the visited set: $O(1)$

Main Loop:

- . The main loop runs until all vertices have been visited, so it iterates at most V times where.
- . Extracting the minimum distance vertex from the priority queue: $O(\log V)$ per iteration, thus $O(V \log V)$
- . Checking if the vertex has been visited: $O(1)$ per iteration, $O(V)$ overall.
- . Getting adjacent vertices and iterating over them: $O(A)$ where A is the adjacent vertices of the current vertex,. This occurs withing the loop over all vertices, thus $O(V * A)$ which is equal to $O(E)$ where E is the number of routes in the graph.
- . Updating distances and pushing into the priority queue: Since we potentially push every edge into the queue, $O(E \log V)$.

Conclusion:

The total time complexity is $O(V \log V) + O(V) + O(E) + O(E \log V)$, which simplifies to $O((V + E) \log V)$.

Kruskals algorithm (Time Complexity Table)

Algorithm	Number of primitive operations
int numOfStations = graph.getNumOfStations();	$O(1)$
priority_queue<Route, vector<Route>, RouteComparator> pq;	$O(1)$
vector<int> distances(numOfStations, INF);	$O(V)$ where V is the number of Stations
while (visited.size() < numOfStations)	$O(V)$
int minDistance = pq.top().dstStation.id; pq.pop();	$O(V \log V)$
auto adjacents = graph.getAdjacents(minDistance);	$O(E)$
for (const auto &adjacent : adjacents)	$O(E)$
int adjID = adjacent.dstStation.id;	$O(E)$
int newDistance = distances[minDistance] + adjacent.distance;	$O(E)$
if (visited.count(adjID) == 0 && newDistance < distances[adjID])	$O(E)$
distances[adjID] = newDistance;	$O(E)$
pq.push(Route(adjacent.srcStation, adjacent.dstStation, newDistance));	$O(E \log V)$
parent[adjID] = minDistance;	$O(E)$
visited.insert(minDistance);	$O(V)$
Return distances;	$O(1)$
T(N)	$O((V + E) \log V)$

5.5 0/1 knapsack algorithm

```
int Knapsack(vector<Station> &stations, vector<vector<int>> &matrix, int ansr[], int capacity)
{
    int rows = stations.size() + 1; // Number of rows in the matrix
    int columns = capacity + 1; // Number of columns in the matrix

    for (int i = 0; i < columns; i++)
        matrix[0][i] = 0;

    // Iterate through each station and each possible capacity
    for (int i = 1; i < rows; i++)
    {
        for (int w = 0; w < columns; w++)
        {
            if (i == 0 || w == 0)
                matrix[i][w] = 0; // Base case: no items or no capacity left

            else if (stations[i - 1].weight > w)
                matrix[i][w] = matrix[i - 1][w]; // Cannot include the current station

            else
                matrix[i][w] = max(matrix[i - 1][w], matrix[i - 1][w - stations[i - 1].weight] + stations[i - 1].profit); // Maximum value
        }
    }

    // Determine the winning stations based on the calculated matrix
    setWinningStations(rows, columns, matrix, stations, ansr);

    // Determine the winning stations based on the calculated matrix
    int maxProfit = matrix[rows - 1][columns - 1];
    return maxProfit;
}
```

code snippet 45 knapsack algorithm

Initialization: Initializing the first row of the matrix with zeros takes $O(\text{columns})$ time, where 'columns' is the capacity plus one.

Nested Loops: The nested loops iterate over each item and each possible weight capacity. The outer loop iterates over the items, and the inner loop iterates over the weight capacities. Therefore, the nested loops contribute $O(\text{rows} * \text{columns})$ time complexity where rows are the items and columns are the capacity.

Inner Conditional Statements: Inside the nested loops, there are conditional statements that execute in constant time. These statements compare weights and calculate maximum profits based on previous computations.

setWinningStations Function: this function sets the winning stations to be included to form the maximum profit which takes $O(\text{rows})$.

Overall, the dominant factor in the time complexity is the nested loops, which iterate over all items and weight capacities. Hence, the time complexity of your algorithm is $O(\text{rows} * \text{columns})$, where 'rows' is the number of items ($\text{stations.size()} + 1$) and 'columns' is the capacity +1.

knapsack algorithm (time complexity Table)

Algorithm	Number of primitive operations
int rows = stations.size() + 1;	O(1)
int columns = capacity + 1;	O(1)
for (int i = 0; i < columns; i++)	O(columns)
for (int i = 1; i < rows; i++)	O(rows)
for (int w = 0; w < columns; w++)	O(rows * columns)
if (i == 0 w == 0)	O(rows * columns)
matrix[i][w] = 0;	O(rows * columns)
else if (stations[i - 1].weight > w)	O(rows * columns)
matrix[i][w] = matrix[i - 1][w];	O(rows * columns)
else	O(rows * columns)
matrix[i][w] = max(matrix[i - 1][w], matrix[i - 1][w - stations[i - 1].weight] + stations[i - 1].profit);	O(rows * columns)
setWinningStations(rows, columns, matrix, stations, ansr);	O(rows)
int maxProfit = matrix[rows - 1][columns - 1];	O(1)
return maxProfit	O(1)
T(N)	O(rows * columns)

6 Space Complexity

6.1 Data Set 2 (Space Complexity)

The space complexity **TreasureHuntDataSet class** primarily depends on the space required by its member variables and any auxiliary data structures used within its methods.

Graph: $O(V + E)$ where V is the number of stations and E is the number of routes.

Vector of Stations: $O(V)$ where V is the number of stations.

Auxiliary Data Structures: candidates map which takes $O(V^2)$

Overall Space Complexity: $O(V^2)$.

6.2 heap sort (Space Complexity)

While the time complexity offers insights into the efficiency of Heap Sort, it's also crucial to consider the space complexity. Heap Sort's space complexity is generally considered to be $O(1)$ due to its in-place sorting nature, making efficient use of memory without requiring additional space proportional to the dataset size.

6.3 merge sort (Space Complexity)

Merge Sort is known for its stable space complexity of $O(n)$ due to its divide-and-conquer approach, which efficiently uses additional memory during the merging phase. This ensures that the algorithm remains efficient in terms of space utilization, even for larger datasets.

6.4 Kruskal's algorithm (Space Complexity)

The space complexity Kruskal's algorithm primarily depends on the data structures used, Minimum spanning tree and Auxiliary variables.

Data Structures:

vector<node> dsuf: This vector stores information about the disjoint-set data structure. It contains information about the parent and rank of each node. The space complexity of this vector is $O(V)$, where V is the number of stations.

vector<Route> routesList: This vector stores all the routes in the graph. The space complexity of this vector is $O(E)$ where E is the number of routes in the input graph.

Graph mst: This represents the minimum spanning tree. The space complexity depends on the number of routes in the MST, hence the space complexity would be $O(e)$ where e is the number of routes in the MST.

Auxiliary Variables: int mstCost: This variable stores the total cost of the minimum spanning tree. It takes constant space.

Overall space complexity: $O(V) + O(E)$.

6.5 Dijkstra's algorithm (Space Complexity)

For Dijkstra's algorithm we used multiple data structures, let's view the space complexity for each data structure and then get the overall space complexity.

Priority Queue (pq): $O(E)$ where E is the number of routes.

Vector distances: $O(V)$ where V is the number of stations.

Set visited: $O(V)$ where V is the number of stations.

Array parent[]: $O(V)$ where V is the number of stations.

Overall Space Complexity: $O(E) + O(V) + O(V) + O(V)$, which simplifies to $O(E + V)$.

6.6 0/1 Knapsack algorithm (Space Complexity)

For 0/1 Knapsack algorithm we used multiple data structures, let's view the space complexity for each data structure and then get the overall space complexity.

Dynamic Programming Matrix (matrix): $O(N * C)$, where N is the number of stations and C is the capacity of the knapsack.

Winning Stations Array (ansr[]): $O(N)$, where N is the number of stations.

Overall Space Complexity: $O(N * C) + O(N)$.