Faculty of Computing & Informatics (FCI)
Multimedia University
Cyberjaya

TCP2101 - Algorithm Design and Analysis

Trimester 1, 2023/2024

Num	Student ID	Student Name	Task descriptions	Percentage
1	1221303085	Mannoj Sakthivel (Leader)	Question 1: DATASET 1 & 2	25%
2	12011001116	Virosshen A/L Sivasankar	Question 2: Heap Sort and Merge Sort	25%
3	1201201862	Noor Elyana Bt Mohd Hasim	Question 3 : Dijkstra's Algorithm and Kruskal's Algorithm	25%
4	1201201773	Thaqif Iskandar Bin Aminuddin	Question 4: Dynamic Programming	25%

TABLE OF CONTENTS

Question 1: Dataset 1 and Dataset 2

Dataset 1

Section 1: Dataset Generation Function

```
// Function to generate a random dataset using the group leader's ID digits
vector<int> generateDataset(long long seed, int datasetSize, const vector<int>& idDigits) {
    mt19937 generator(seed);  // Mersenne Twister 19937 generator
    vector<int> dataset(datasetSize);
```

- This section defines a function named **generateDataset**. It takes three parameters: **seed** for random number generation, **datasetSize** to determine the size of the dataset, and **idDigits**, which is a vector of individual digits extracted from the group leader's ID. The function returns a vector of integers, which represents the generated dataset.
- Next, we initialize a random number generator using the Mersenne Twister 19937 algorithm. The generator is seeded with the provided **seed**, ensuring that the generated numbers will be reproducible if the same seed is used.
- Finally, the line creates an empty vector called **dataset** with a size specified by **datasetSize**. This vector will hold the generated dataset elements.

Section 2: Dataset Generation For-Loop

```
// Generate numbers by combining digits from the ID
for (int i = 0; i < datasetSize; ++i) {
    int number = 0;
    for (int j = 0; j < 3; ++j) {  // Generate a 3-digit number
        uniform_int_distribution<> dist(0, idDigits.size() - 1);
        number = number * 10 + idDigits[dist(generator)];
    }
    dataset[i] = number;
}

return dataset;
```

- This section begins a **for** loop that will run **datasetSize** times. In other words, it is responsible for generating the specified number of dataset elements.
- Inside the loop, we initialize a variable **number** to zero. This variable will be used to construct each element of the dataset.
- Within the loop, there is another nested **for** loop that runs three times (**j** ranges from 0 to 2). This inner loop is responsible for generating a 3-digit number.
- Next, we create a uniform integer distribution named **dist** that defines a range from 0 to one less than the size of the **idDigits** vector. This distribution will be used to select random digits from the **idDigits** vector.
- Then, we generate a random digit using the **dist** distribution and append it to the **number**. This effectively constructs a 3-digit number by shifting the existing digits left by a factor of 10 and adding the new digit to the units place.
- Finally, we assign the generated 3-digit number to the **i**-th position in the **dataset** vector, effectively populating the dataset element by element.
- This concludes the inner loop responsible for generating a single dataset element.
- At the end of the function, we return the fully generated dataset as a vector of integers.

Section 3: Main Function

```cpp
int main() {
    long long groupLeaderId = 1221303085;
    auto idDigits = getDigitsFromId(groupLeaderId);

    // Specify dataset sizes
    vector<int> setSizes = {100, 1000, 10000, 100000, 500000};

    // Create a map to store datasets
    map<string, vector<int>> datasets;

    // Generate datasets
    for (size_t i = 0; i < setSizes.size(); ++i) {
        long long seed = groupLeaderId * (i + 1);  // Use different seed for each set
        vector<int> dataset = generateDataset(seed, setSizes[i], idDigits);
        // Corrected comment and map key
        datasets["Set " + to_string(i + 1) + " (" + to_string(setSizes[i]) + ")"] = dataset;
    }

    // Print or use the generated datasets as needed
    for (const auto& setPair : datasets) {
        cout << setPair.first << ": ";
        for (int i = 0; i < min(5, static_cast<int>(setPair.second.size())); ++i) {
            cout << setPair.second[i] << " ";
        }
        cout << "...\n";  // Print the first 5 elements as an example
    }

    // Check if the set actually has 500000 elements
    string datasetKey = "Set 5 (500000)";
    if (datasets.find(datasetKey) != datasets.end()) {
        size_t size = datasets[datasetKey].size();
        cout << "Size of " << datasetKey << ": " << size << " elements." << endl;
    } else {
        cerr << "Dataset " << datasetKey << " not found." << endl;
    }

    return 0;
}
```

- This section defines the main function, which serves as the entry point of the program. It's where the program execution starts.

Section 4: Group Leader's ID

```cpp
long long groupLeaderId = 1221303085;
```

- Here, we define a variable **groupLeaderId** and assign it the value **1221303085**. This is the group leader's ID, which is used as the basis for generating random datasets.

Section 5: Extracting Individual Digits

```
auto idDigits = getDigitsFromId(groupLeaderId);
```

- This line calls the **getDigitsFromId** function with **groupLeaderId** as an argument. The function extracts individual digits from the group leader's ID and stores them in a vector called **idDigits**. This vector is used later for generating the datasets.

Section 6: Specifying Dataset Sizes

```
// Specify dataset sizes
vector<int> setSizes = {100, 1000, 10000, 100000, 500000};
```

- In this part, we define a vector called **setSizes** that contains the sizes for five different datasets: 100, 1,000, 10,000, 100,000, and 500,000 elements. These sizes are specified in accordance with the question's requirements.

Section 7: Creating a Map for Datasets

```
// Create a map to store datasets
map<string, vector<int>> datasets;
```

- Here, we create an empty map called **datasets**. This map will be used to store the generated datasets. The keys of the map will indicate both the set number and the size of each dataset, while the values will be vectors containing the actual dataset elements.

Section 8: Generating Datasets

```
// Generate datasets
for (size_t i = 0; i < setSizes.size(); ++i) {
    long long seed = groupLeaderId * (i + 1);  // Use different seed for each set
    vector<int> dataset = generateDataset(seed, setSizes[i], idDigits);
    // Corrected comment and map key
    datasets["Set " + to_string(i + 1) + " (" + to_string(setSizes[i]) + ")"] = dataset;
}
```

- This part initiates a loop that iterates through the **setSizes** vector. It prepares the code to generate datasets of various sizes as specified in the question.

Section 9: Calculating Unique Seeds

```
// Generate datasets
for (size_t i = 0; i < setSizes.size(); ++i) {
    long long seed = groupLeaderId * (i + 1);  // Use different seed for each set
    vector<int> dataset = generateDataset(seed, setSizes[i], idDigits);
```

- Within the loop, this line calculates a unique seed for each dataset. The seed is computed by multiplying the group leader's ID by **(i + 1)**, where **i** is the loop index. This ensures that each dataset is generated with a different seed, guaranteeing randomness.

Section 10: Storing Datasets in the Map

```
// Corrected comment and map key
datasets["Set " + to_string(i + 1) + " (" + to_string(setSizes[i]) + ")"] = dataset;
```

- Finally, in this section, each generated dataset is stored in the **datasets** map. The keys for each dataset are constructed by combining the set number and the specified size. This allows for easy identification and retrieval of datasets.

Dataset 2

Section 1: Generating Dataset

The word game refers to the simulated treasure hunt scenario based on the assignment question.

1.1 Struct Structure

```
struct Station {
    string name;
    int x, y, z, weight, profit;
};
```

- The **Station** struct is a cornerstone of the game's data model. It encapsulates all necessary attributes of a station in a single, tidy package.
- The use of a struct (over a class) emphasizes the plain-data nature of stations - they are primarily data holders without complex behavior or encapsulation needs.
- The **name** field is a string, which offers more flexibility than a char if the naming convention changes.
- The **x, y, z** coordinates define the station's position in a simulated 3D space, paving the way for spatial calculations and possibly for a more immersive 3D representation in a future game interface.
- **weight** and **profit** can be foundational for gameplay dynamics, like managing resources or calculating scores.

1.2 Distance Calculation Function

```
double calculateDistance(const Station& a, const Station& b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2) + pow(a.z - b.z, 2));
}
```

- Central to gameplay mechanics involving movement or range. For instance, it could be used to calculate fuel consumption if moving between stations consumes resources.

- The use of **const** & references in **const Station& a, const Station& b** ensures that the stations are not inadvertently modified within the function, upholding the principle of immutability for function inputs.
- The choice of **sqrt** and **pow** from the **<cmath>** library is for clarity and reliability. The functions are well-tested and optimized, providing accurate and efficient computations.

1.3 Random Number Generation Function

```
// Function to generate random number from seed digits
int generateRandomNumber(mt19937& rng, const vector<int>& digits, int numDigits) {
    uniform_int_distribution<> dist(0, digits.size() - 1);
    int result = 0;
    for (int i = 0; i < numDigits; ++i) {
        int digit = digits[dist(rng)];
        // Ensure the first digit is non-zero when numDigits is more than 1
        if (i == 0 && numDigits > 1) {
            while (digit == 0) {
                digit = digits[dist(rng)];
            }
        }
        result = result * 10 + digit;
    }
    return result;
}
```

- This function is a workhorse behind the variability in the game setup. By generating random numbers for station properties, it ensures each game setup is unique.
- The function signature is carefully designed. It takes an **mt19937& rng**, which is a reference to an external Mersenne Twister random number generator. This design allows for greater control over random number generation, such as seeding for reproducibility or sharing the generator across different parts of the program.
- **const vector<int>& digits** provides the digits to be used in generating the number, allowing for a controlled randomization scope. It's passed by reference for efficiency, avoiding unnecessary copying.
- **int numDigits** explicitly states the desired length of the number, adding clarity and control over the output.

Section 2: Generating the Dataset

2.1 Generating Stations

```
// Function to generate stations with random properties
vector<Station> generateStations(unsigned int seed, int numStations, const vector<int>& seedDigits) {
    vector<Station> stations;
    mt19937 rng(seed);
    for (int i = 0; i < numStations; ++i) {
        Station station;
        // Map station index to a letter (A-Z)
        char stationLetter = 'A' + i;
        station.name = string(1, stationLetter);
        station.x = generateRandomNumber(rng, seedDigits, 3);
        station.y = generateRandomNumber(rng, seedDigits, 3);
        station.z = generateRandomNumber(rng, seedDigits, 3);
        station.weight = generateRandomNumber(rng, seedDigits, 2);
        station.profit = generateRandomNumber(rng, seedDigits, 2);
        stations.push_back(station);
    }
    return stations;
}
```

- This function encapsulates the logic for initializing the game's stations, central to setting the stage for the game.
- It uses the **seed** to initialize the random number generator, which ensures that the same seed will produce the same set of stations, a feature that can be crucial for game testing or recreating specific game scenarios.
- The loop within the function systematically assigns unique names and random properties to each station, ensuring each station has its distinct place and value in the game world.
- The **seedDigits** parameter ties the randomness of the station properties to the initial seed, maintaining consistency and predictability in the randomness.

```
for (int i = 0; i < numStations; ++i) {
    Station station;
    // Map station index to a letter (A-Z)
    char stationLetter = 'A' + i;
    station.name = string(1, stationLetter);
    station.x = generateRandomNumber(rng, seedDigits, 3);
    station.y = generateRandomNumber(rng, seedDigits, 3);
    station.z = generateRandomNumber(rng, seedDigits, 3);
    station.weight = generateRandomNumber(rng, seedDigits, 2);
    station.profit = generateRandomNumber(rng, seedDigits, 2);
    stations.push_back(station);
}
```

- Initializes a new **Station** object.
- Assigns a name to the station, incrementing from 'A' onwards based on the loop counter **i**.
- Generates random numbers for the station's **x, y, z** coordinates, and its **weight** and **profit** attributes. This is done using the **generateRandomNumber** function, ensuring that each station has unique and randomly assigned properties.
- Adds the newly created station to the **stations** vector, building up the complete list of stations in the game.

2.2 Generating Routes

```
// Function to generate a map of connections between stations
vector<pair<int, int>> generateRoutes(const vector<Station>& stations, int numRoutes) {
    vector<pair<int, int>> routes;
    set<pair<int, int>> uniqueRoutes;
    mt19937 rng(random_device{}());
    map<int, set<int>> connections;

    // Ensure all stations have at least 3 connections
    for (int i = 0; i < stations.size(); ++i) {
        while (connections[i].size() < 3) {
            int j = rng() % stations.size();
            if (i != j && connections[i].find(j) == connections[i].end()) {
                connections[i].insert(j);
                connections[j].insert(i);
                uniqueRoutes.insert(minmax(i, j));
            }
        }
    }

    // Add additional unique routes until we reach 54
    while (uniqueRoutes.size() < numRoutes) {
        int a = rng() % stations.size();
        int b = rng() % stations.size();
        if (a != b && uniqueRoutes.find(minmax(a, b)) == uniqueRoutes.end()) {
            uniqueRoutes.insert(minmax(a, b));
            connections[a].insert(b);
            connections[b].insert(a);
        }
    }

    // Transfer unique routes to the vector
    for (const auto& route : uniqueRoutes) {
        routes.emplace_back(route);
    }

    return routes;
}
```

- This function creates the network of routes, effectively shaping how players will navigate the game world.
- The use of a **set** to store **uniqueRoutes** ensures that all routes are unique, preventing any redundant paths.
- The algorithm first ensures a minimum level of connectivity (each station connected to at least three others), promoting a well-connected and traversable game world.
- Additional routes are then added up to **numRoutes**, enhancing the complexity and richness of the navigation possibilities.

- The use of **pair<int, int>** to represent routes is a straightforward choice that clearly denotes a connection between two stations.

```
// Add additional unique routes until we reach 54
while (uniqueRoutes.size() < numRoutes) {
    int a = rng() % stations.size();
    int b = rng() % stations.size();
    if (a != b && uniqueRoutes.find(minmax(a, b)) == uniqueRoutes.end()) {
        uniqueRoutes.insert(minmax(a, b));
        connections[a].insert(b);
        connections[b].insert(a);
    }
}
```

- Randomly selects two different stations (**a** and **b**) as potential endpoints for a new route.
- Checks if a route between these two stations already exists (**uniqueRoutes.find(minmax(a, b)) == uniqueRoutes.end()**). If not, it proceeds.
- Inserts the new route into **uniqueRoutes**, ensuring that each route is recorded only once.
- Records the connection in a bidirectional way in the **connections** map, meaning that you can navigate from station **a** to **b** and vice versa. This part is fundamental for creating an interconnected network of stations.

Section 3: Main Program Flow

3.1 Main Function

```cpp
int main() {
    long long seedSum = 1201100116LL + 1201201773LL + 1201201862LL;
    vector<int> seedDigits;
    long long tempSeedSum = seedSum;
    while (tempSeedSum > 0) {
        seedDigits.push_back(tempSeedSum % 10);
        tempSeedSum /= 10;
    }
    reverse(seedDigits.begin(), seedDigits.end());

    vector<Station> stations = generateStations(static_cast<unsigned int>(seedSum), 20, seedDigits);
    vector<pair<int, int>> routes = generateRoutes(stations, 54);

    for (const Station& station : stations) {
        cout << station.name << " "
                  << station.x << " "
                  << station.y << " "
                  << station.z << " "
                  << station.weight << " "
                  << station.profit << endl;
    }

    for (const auto& route : routes) {
        cout << "Route between " << stations[route.first].name
                  << " (" << route.first << ")"
                  << " and " << stations[route.second].name
                  << " (" << route.second << ")"
                  << " Distance: " << calculateDistance(stations[route.first], stations[route.second])
                  << endl;
    }

    return 0;
}
```

- Serves as the command center of the program, where all the pieces come together.
- The seed and its digits are prepared first, laying the groundwork for all subsequent randomization.
- The calls to **generateStations** and **generateRoutes** are the two key steps that build up the game's data structure.
- The printing loops for stations and routes not only serve as a way to output the data but also demonstrate how the generated structures can be iterated and utilized, hinting at how game logic might interact with these data structures.

```
for (const Station& station : stations) {
    cout << station.name << " "
              << station.x << " "
              << station.y << " "
              << station.z << " "
              << station.weight << " "
              << station.profit << endl;
}

for (const auto& route : routes) {
    cout << "Route between " << stations[route.first].name
              << " (" << route.first << ")"
              << " and " << stations[route.second].name
              << " (" << route.second << ")"
              << " Distance: " << calculateDistance(stations[route.first], stations[route.second])
              << endl;
}
```

- **First Loop (Printing Stations):**
    - Iterates through each **Station** object in the **stations** vector.
    - Prints out the station's properties (name, coordinates, weight, profit) to the console. This is a straightforward way to visualize the station data and can be helpful for debugging or initial game setup verification.
- **Second Loop (Printing Routes):**
    - Iterates through each route in the routes vector.
    - For each route, it identifies the two stations it connects (**route.first** and **route.second**).
    - Prints the names of the connected stations and the distance between them, calculated using the **calculateDistance** function. This output provides a clear representation of how stations are interconnected in the game, useful for understanding the game map's layout and for debugging purposes.

The loops in the **generateStations, generateRoutes,** and **main** functions are critical for building the game's data structures and for providing a basic output to visualize the generated game world. They show a systematic approach to populating the game world with unique stations and connecting them in a meaningful way, while also ensuring that the generated world can be easily reviewed and understood.

Question 2 (Heap Sort & Merge Sort)

Heap Sort

Section 1 : Header Includes

```
1   #include <iostream>
2   #include <vector>
3   #include <random>
4   #include <chrono>
5   #include <algorithm>
6   #include "matplotlibcpp.h"
```

- **<iostream> :** Input and output stream handling.
- **<vector> :** Standard C++ library for dynamic arrays.
- **<random> :** Library for random number generation.
- **<chrono> :** Library for time-related functionality.
- **<algorithm> :** General algorithms.
- **"Matplotlibcpp.h" :** Header file for the matplotlib c++ library which allows C++ code to interface with Python's matplotlib.

Section 2 : Header Guards

```
8   #ifndef MATPLOTLIBCPP_H
9   #define MATPLOTLIBCPP_H
```

- This is a standard technique in C++ to prevent multiple inclusions of the same code in a translation unit.

Section 3 : Namespace Alias

```
11    namespace plt = matplotlibcpp;
```

- Creates an alias "plt" for the "matplotlib c++" namespace, providing a convenient shorthand for accessing elements from that namespace.

Section 4 : Random Dataset Generation Function

```cpp
// Function to generate a random dataset using the group leader's ID digits
std::vector<int> generateDataset(long long seed, int datasetSize, const std::vector<int>& idDigits) {
    std::mt19937 generator(seed);  // Mersenne Twister 19937 generator
    std::vector<int> dataset(datasetSize);

    // Generate numbers by combining digits from the ID
    for (int i = 0; i < datasetSize; ++i) {
        int number = 0;
        for (int j = 0; j < 3; ++j) {  // Generate a 3-digit number
            std::uniform_int_distribution<> dist(0, idDigits.size() - 1);
            number = number * 10 + idDigits[dist(generator)];
        }
        dataset[i] = number;
    }

    return dataset;
}
```

- Parameters **“seed”** for random number generation, **‘datasetSize’** for the size of the dataset **‘idDigits'** vector for constructing random numbers.
- Uses the Mersenne Twister 19937 random number generator.
- Combines digits from **‘idDigits'** to create 3-digit numbers, generating a random dataset.

Section 5 : Helper Function to get digits from ID

```cpp
// Helper function to get the digits from the group leader's ID
std::vector<int> getDigitsFromId(long long id) {
    std::vector<int> digits;
    while (id > 0) {
        digits.insert(digits.begin(), id % 10);  // Insert at the beginning to keep the order
        id /= 10;
    }
    return digits;
}
```

- Parameter: **‘id’** is a long long number.
- Extracts individual digits from a given number, preserving their order

Section 6 : Helper Function to Heapify.

```cpp
// Helper function to perform heapify
void heapify(std::vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

- Parameter: **‘arr’** is a vector to be heapified, **‘n’** is the size of the heap, **‘i’** is the index of the current element.
- Ensures the max heap property is maintained. Swaps elements to ensure the largest element is at the root.

Section 7 : Heap Sort Function.

```
62  // Function to perform Heap Sort and measure time
63  void heapSort(std::vector<int>& arr, std::vector<long long>& timings) {
64      auto start_time = std::chrono::high_resolution_clock::now();
65
66      int n = arr.size();
67
68      // Build max heap
69      for (int i = n / 2 - 1; i >= 0; i--)
70          heapify(arr, n, i);
71
72      // Extract elements one by one from the heap
73      for (int i = n - 1; i > 0; i--) {
74          std::swap(arr[0], arr[i]);
75          heapify(arr, i, 0);
76      }
77
78      auto end_time = std::chrono::high_resolution_clock::now();
79      auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start_time);
80      timings.push_back(duration.count());
81  }
```

- **Parameters:**
  - **'arr':** This is the vector (array) that needs to be sorted using the Heap Sort algorithm.
  - **'timings':** This is a vector that will store the execution times of the Heap Sort for different dataset sizes.
- **Body Function:**
  - Line 64 = Records the starting time before the Heap Sort process begins.
  - Line 66 = **'n'** represents the size of the array **'arr'**
  - Line 68 - 70 (Build Max Heap) = This loop builds a max heap from the input array. It starts from the last non-leaf node and calls the **heapify function** for each node to ensure the max heap property is satisfied.
  - Line 72 - 76 (Extracting Elements) = This loop extracts elements from the heap one by one. It swaps the root of the heap (which contains the maximum element) with the last element in the heap. After each swap, it calls **heapify** to restore the max heap property.
  - Line 78 - 81 (Timings): Records the ending time after the Heap Sort process is complete. Calculates the duration of the Heap Sort in microseconds. Lastly, appends the duration of the 'timings' vector.

Section 8 : Function to check if Array is sorted.

```
83    // Function to check if the array is sorted
84    bool isSorted(const std::vector<int>& arr) {
85        return std::is_sorted(arr.begin(), arr.end());
86    }
```

- Parameter : **'arr'** is the vector to be checked.
- Utilizes the **std::is_sorted** function to check if the vector is sorted.

Section 9 : Function to Plot the Graph.

```
88  // Function to plot the graph
89  void plotGraph(const std::vector<int>& sizes, const std::vector<long long>& timings, const std::string& title) {
90      plt::plot(sizes, timings);
91      plt::title(title);
92      plt::xlabel("Dataset Size");
93      plt::ylabel("Time (microseconds)");
94      plt::show();
95  }
96
```

- Parameters: **'sizes'** is the vector of dataset sizes, **'timings'** is the vector of execution times, **'title'** is the title of the graph.
- Uses the **'matplotlibcpp'** library to plot a graph showing the relationship between dataset size and execution time.

Section 10 : Main Function.

```
99  int main() {
100     // Specify dataset sizes
101     std::vector<int> setSizes = {100, 1000, 10000, 100000, 500000};
102
103     // Create vectors to store timings
104     std::vector<long long> heapSortTimings;
105
106     // Generate datasets and perform Heap Sort
107     for (int size : setSizes) {
108         long long groupLeaderId = 1221303085;
109         auto idDigits = getDigitsFromId(groupLeaderId);
110         long long seed = groupLeaderId * 5;  // Use seed for Set 5
111         std::vector<int> dataset = generateDataset(seed, size, idDigits);
112
113         std::vector<int> heapSortData = dataset;
114         heapSort(heapSortData, heapSortTimings);
115
116         // Check if the array is sorted (for debugging)
117         if (!isSorted(heapSortData)) {
118             std::cerr << "Error: The array is not sorted.\n";
119             return 1;
120         }
121     }
```

- **Body Function:**
  - **'setSizes'** is a vector that contains different sizes for the datasets. In this case, it includes 100, 1000, 10000, 100000, and 500000 elements.
  - Line 103 - 104 = **'heapSortTimings'** is a vector that will be used to store the execution times of Heap Sort for each dataset size.
  - Line 106 - 121:
    - This loop iterates through each dataset size specified in **'setSizes'**
    - It generates a dataset using the **'generateDataset'** function based on a seed (calculated from the group leader's ID)
    - A copy of the dataset is created 'heapSortData' and Heap Sort is performed on it. The timings are recorded in the **'heapSortTimings'** vector.
    - Lastly, It checks whether the array is indeed sorted using the 'isSorted' function; if not it will display an error message. This is solely for debugging purposes.

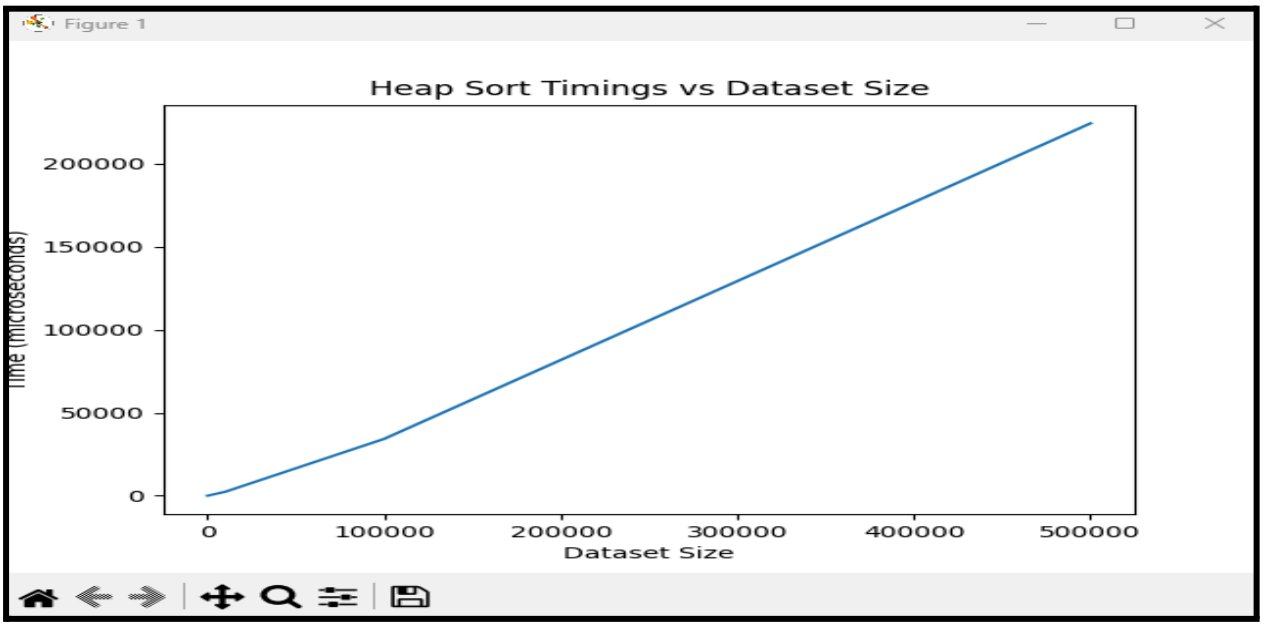
Section 11 : Plot The Graph.

```
123     // Plot the graph
124     plotGraph(setSizes, heapSortTimings, "Heap Sort Timings vs Dataset Size");
```

- After sorting all datasets and recording timings, this line plots a graph using the **'plotGraph'** function. It visualizes the execution times of Heap Sort for different dataset sizes.

Section 12 : The Result

- To run the HeapSort.cpp code using matplotlib.cpp. We first have to save the file the matplotlib cpp in the same directory as the HeapSort.cpp.
- Next we enter the command prompt and write a code to execute the cpp files.
- **Run in C:\Users\USER\Viro - Main Folder\Algorithm and Analysis Design> g++ -o heapsort HeapSort.cpp -std=c++11 -I "C:\Program Files\Python312\include" -I "C:\Users\USER\AppData\Roaming\Python\Python312\site-packages\numpy\core\include" -L "C:\Program Files\Python312\libs" -lpython312 -Wno-deprecated-declarations**
- After executing the code, we will see a heapsort.exe file produced in the directory. This executable file is actually the plotted graph.



- **X - Axis:**
    - Dataset Size: The x-axis represents the size of the datasets used for testing the Heap Sort algorithm. In this case, the sizes are 100, 1000, 10000, 100000, and 500000 elements.
- **Y - Axis:**
    - Time (Microseconds): The y-axis represents the time taken by the Heap Sort algorithm to sort the datasets, measured in microseconds.
- **Graph Line:**
    - The graph is a line plot, where each point on the line corresponds to a dataset size, and its corresponding y-coordinate is the time taken to sort the dataset.
- **Interpretation:**
    - **Increasing Trend:** Generally, as the dataset size increases, the execution time of the Heap Sort algorithm tends to increase. This is expected because larger datasets require more comparisons and swaps during the sorting process.
    - **Time Complexity Analysis:** If the graph shows a roughly linear or logarithmic trend, it indicates that Heap Sort has a time complexity of O(n log n). If the trend is quadratic, it could suggest a time complexity of O(n^2).

Merge Sort

Section 1 : Libraries and Namespace

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include "matplotlibcpp.h"

namespace plt = matplotlibcpp;
```

- **<iostream> :** Input and output stream handling.
- **<vector> :** Standard C++ library for dynamic arrays.
- **<random> :** Library for random number generation.
- **<chrono> :** Library for time-related functionality.
- **<algorithm> :** General algorithms.
- **“Matplotlibcpp.h” :** Header file for the matplotlib c++ library which allows C++ code to interface with Python’s matplotlib.
- Creates an alias “plt” for the “matplotlib c++” namespace, providing a convenient shorthand for accessing elements from that namespace.

Section 2 : Random Dataset Generation Function

```cpp
// Function to generate a random dataset using the group leader's ID digits
std::vector<int> generateDataset(long long seed, int datasetSize, const std::vector<int>& idDigits) {
    std::mt19937 generator(seed);
    std::vector<int> dataset(datasetSize);

    for (int i = 0; i < datasetSize; ++i) {
        int number = 0;
        for (int j = 0; j < 3; ++j) {
            std::uniform_int_distribution<> dist(0, idDigits.size() - 1);
            number = number * 10 + idDigits[dist(generator)];
        }
        dataset[i] = number;
    }

    return dataset;
}
```

- **‘generateDataset’:** This function creates a dataset of random numbers based on a seed, size, and given digits. It utilises the Mersenne Twister 19937 algorithm for randomness.
- **‘getDigitsFromId’:** It extracts individual digits from a given ID in reverse order.

Section 3 : Main Merge Sort Function

```
77  // Main Merge Sort function
78  void mergeSort(std::vector<int>& arr, int left, int right) {
79      if (left < right) {
80          // Same as (left + right) / 2, but avoids overflow for large left and right
81          int middle = left + (right - left) / 2;
82
83          // Sort first and second halves
84          mergeSort(arr, left, middle);
85          mergeSort(arr, middle + 1, right);
86
87          // Merge the sorted halves
88          merge(arr, left, middle, right);
89      }
90  }
```

- **'merge':** This function takes two sorted halves of an array and merges them into a single sorted array.
- **'mergeSort':** The main recursive function for the merge sort algorithm. It divides the array into halves, sorts them, and then merges them back together.
- **'runMergeSort':** It measures the time taken to perform merge sort on an array and records this timing.

Section 3 : Run Merge Function

```
92  // Function to perform Merge Sort and measure time
93  void runMergeSort(std::vector<int>& arr, std::vector<long long>& timings) {
94      auto start_time = std::chrono::high_resolution_clock::now();
95
96      int n = arr.size();
97
98      // Perform Merge Sort
99      mergeSort(arr, 0, n - 1);
100
101     auto end_time = std::chrono::high_resolution_clock::now();
102     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start_time);
103     timings.push_back(duration.count());
104 }
```

- Line 94 = This line records the current time as the starting point for measuring the execution time. It uses the high-resolution clock, which provides more precision than a regular clock.
- Line 96 = This line retrieves the size (number of elements) of the input array
- Line 99 = The function calls the mergeSort function passing the array, and the indices representing the entire range of the array ('0' to 'n-1'). This initiates the merge sort process.
- Line 101 = After the merge sort operation is complete, the current time is recorded as the end point for measuring the execution time.
- Line 102 = This line calculates the duration of the sorting process by subtracting the start time from the end time. The result is cast to microseconds for clarity.

- Line 103 = The duration, represented in microseconds, is added to the timings vector. This vector is intended to store the execution times of various runs for later analysis or plotting.

Section 4 : Print Merge Function

```
// Function to print the first few elements of the array
void printArray(const std::vector<int>& arr, int count = 10) {
    std::cout << "Sorted Array: ";
    for (int i = 0; i < std::min(count, static_cast<int>(arr.size())); ++i) {
        std::cout << arr[i] << " ";
    }
    std::cout << "...\n";
}
```

- Line 112 = The function takes a constant reference to a vector of integers as its parameter. The vector is assumed to be sorted.
- Line 112 = An optional parameter with a default value of 10. This parameter determines the number of elements to print from the array. If not provided, it defaults to 10.
- Line 114 = The function uses a for loop to iterate through the array elements. It iterates up to the minimum of 'count' and the size of the array to avoid going beyond the array bounds.
- Line 115 = For each element in the loop, the current element is printed, followed by a space.

Section 5 : Main Function

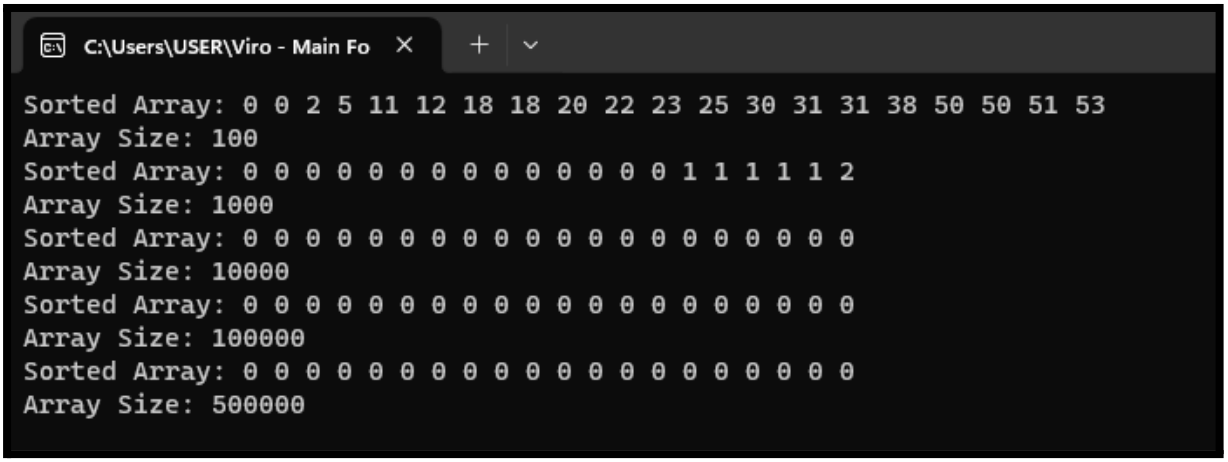
```
129  int main() {
130      // Specify dataset sizes
131      std::vector<int> setSizes = {100, 1000, 10000, 100000, 500000};
132
133      // Create vectors to store timings
134      std::vector<long long> mergeSortTimings;
135
136      // Generate datasets and perform Merge Sort
137      for (int size : setSizes) {
138          long long groupLeaderId = 1221303085;
139          auto idDigits = getDigitsFromId(groupLeaderId);
140          long long seed = groupLeaderId * 5;  // Use seed for Set 5
141          std::vector<int> dataset = generateDataset(seed, size, idDigits);
142
143          std::vector<int> mergeSortData = dataset;
144          runMergeSort(mergeSortData, mergeSortTimings);
145
146          // Check if the array is sorted
147          if (!isSorted(mergeSortData)) {
148              std::cerr << "Error: The array is not sorted after Merge Sort.\n";
149              return 1;
150          }
151
152          // Print the first few elements of the sorted array
153          printArray(mergeSortData);
154      }
155
156      // Plot the graph
157      plotGraph(setSizes, mergeSortTimings, "Merge Sort Timings vs Dataset Size");
158
159      return 0;
160  }
```

- Line 131 = Specifies the sizes of datasets to be used in the experiment. In this case, datasets of sizes 100, 1000, 10000, 100000, and 500000 are used.
- Line 134 = A vector used to store the execution times of Merge Sort for different dataset sizes.
- Line 137 - 154:
    - A loop iterates over each dataset size specified in **‘setSizes’**
    - The group leader's ID is used to generate a set of digits **‘idDigits’**
    - A seed is calculated based on the group leader's ID for randomness in dataset generation.
    - A dataset is generated using the **‘generateDataset’** function.
    - A copy of the dataset is created for Merge Sort **‘mergeSortData’**
    - Merge Sort is performed on the dataset, and the execution time is recorded.
    - After processing all dataset sizes, a graph is plotted using **‘plotGraph’** to show Merge Sort timings vs Dataset Size.

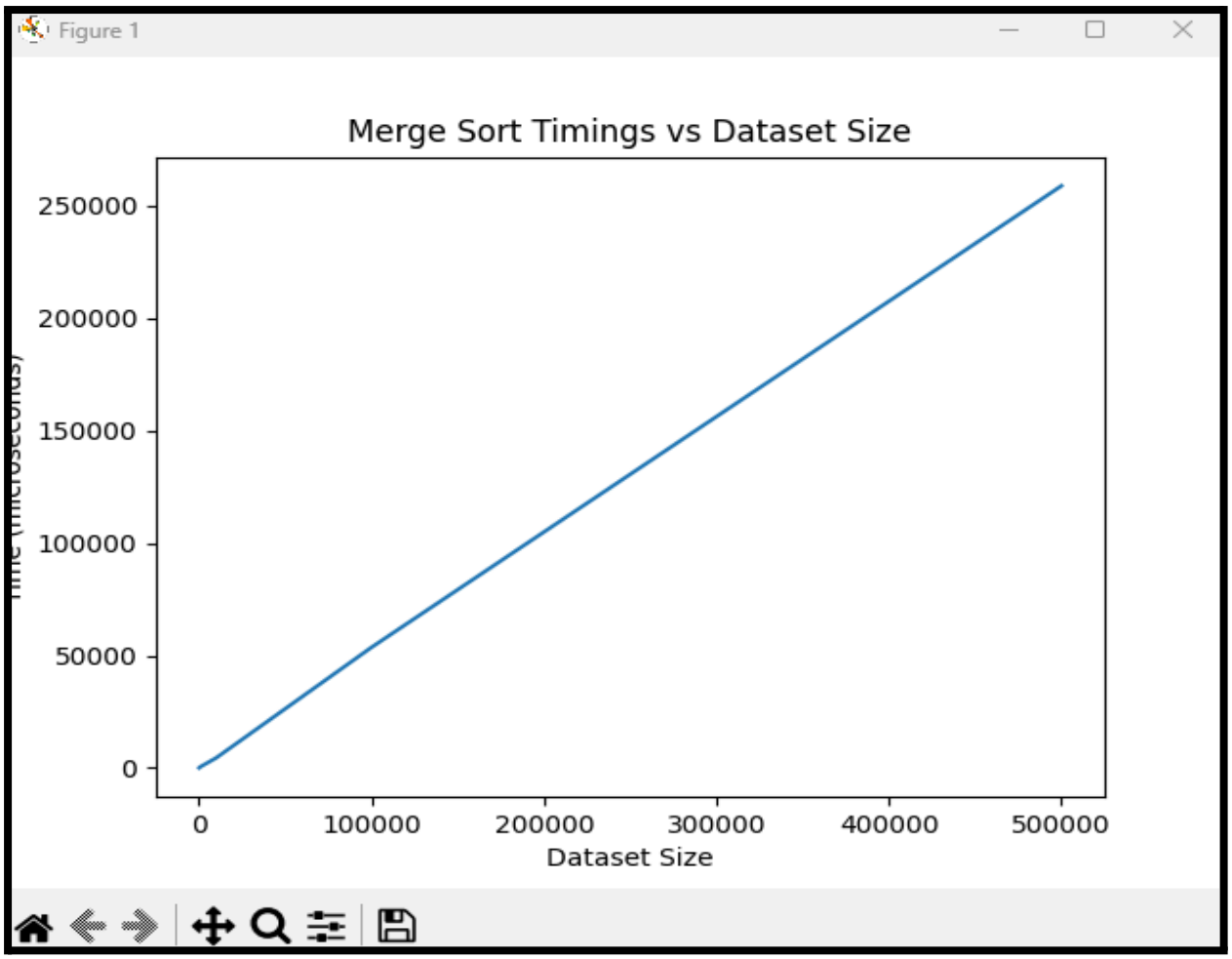
Section 6 : The Result.

- To run the MergeSort.cpp code using matplotlib.cpp. We first have to save the file the matplotlib cpp in the same directory as the MergeSort.cpp.
- Next we enter the command prompt and write a code to execute the cpp files.
  Run in C:\Users\USER\Viro - Main Folder\Algorithm and Analysis Design>
  g++ -o mergesort MergeSort.cpp -std=c++11 -I "C:\Program Files\Python312\include" -I "C:\Users\USER\AppData\Roaming\Python\Python312\site-packages\numpy\core\include" -L "C:\Program Files\Python312\libs" -lpython312 -Wno-deprecated-declarations

- After executing the code, we will see a mergesort.exe file produced in the directory. This executable file is actually the plotted graph.



- The generator is producing a considerable number of zeros, dominating the initial part of the sorted arrays. This could be due to the distribution of random numbers or a specific characteristic of the generator.
- The method of generating random numbers using the group leader's ID and its digits might contribute to patterns or biases in the generated values. It's possible that the ID or its digits lead to certain values being favoured during the generation process.



- **X-Axis (Dataset Size):**
  - The x-axis represents the size of the datasets that were sorted. In your case, the sizes are {100, 1000, 10000, 100000, 500000}. Each point on the x-axis corresponds to a specific dataset size.
- **Y-Axis (Time in Microseconds):**
  - The y-axis represents the time taken by the Merge Sort algorithm to sort the datasets. Each point on the y-axis corresponds to the execution time (in microseconds) for sorting a dataset of a specific size.
- **Graph Shape:**
  - As the dataset size increases, the execution time may also increase. The graph might exhibit a trend that shows how the execution time scales with the dataset size.
- **Performance Characteristics:**
  - A steep increase in execution time with a larger dataset size might indicate that the algorithm has a higher time complexity for larger inputs.
  - A relatively flat line or a slower increase might suggest that the algorithm performs well even with larger datasets.

Question 3: Dijkstra's Algorithm and Kruskal Algorithm

Dijkstra's Algorithm

From Dataset 2, we can identify the shortest path from Station A to other stations using Dijkstra's Algorithm.

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <random>
#include <algorithm>
#include <set>
#include <map>
#include <queue>
#include <iterator>
#include <fstream>

using namespace std;

struct Station
{
    string name;
    int x, y, z, weight, profit;
};

double calculateDistance(const Station& a, const Station& b)
{
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2) + pow(a.z - b.z, 2));
}
```

```cpp
// Function to generate random number from seed digits
int generateRandomNumber(mt19937& rng, const vector<int>& digits, int numDigits)
{
    uniform_int_distribution<> dist(0, digits.size() - 1);
    int result = 0;
    for (int i = 0; i < numDigits; ++i)
    {
        int digit = digits[dist(rng)];
        // Ensure the first digit is non-zero when numDigits is more than 1
        if (i == 0 && numDigits > 1)
        {
            while (digit == 0)
            {
                digit = digits[dist(rng)];
            }
        }
        result = result * 10 + digit;
    }

    return result;
}

```

```
47
48  // Function to generate stations with random properties
49  vector<Station> generateStations(unsigned int seed, int numStations, const vector<int>& seedDigits)
50  {
51      vector<Station> stations;
52      mt19937 rng(seed);
53
54      for (int i = 0; i < numStations; ++i)
55      {
56          Station station;
57          // Map station index to a letter (A-Z)
58          char stationLetter = 'A' + i;
59          station.name = string(1, stationLetter);
60          station.x = generateRandomNumber(rng, seedDigits, 3);
61          station.y = generateRandomNumber(rng, seedDigits, 3);
62          station.z = generateRandomNumber(rng, seedDigits, 3);
63          station.weight = generateRandomNumber(rng, seedDigits, 2);
64          station.profit = generateRandomNumber(rng, seedDigits, 2);
65          stations.push_back(station);
66      }
67
68      return stations;
69  }
70
```

```
71
72  // Function to generate a map of connections between stations
73  vector<pair<int, int>> generateRoutes(const vector<Station>& stations, int numRoutes)
74  {
75      vector<pair<int, int>> routes;
76      set<pair<int, int>> uniqueRoutes;
77      mt19937 rng(random_device{}());
78      map<int, set<int>> connections;
79
80      // Ensure all stations have at least 3 connections
81      for (int i = 0; i < stations.size(); ++i)
82      {
83          while (connections[i].size() < 3)
84          {
85              int j = rng() % stations.size();
86              if (i != j && connections[i].find(j) == connections[i].end())
87              {
88                  connections[i].insert(j);
89                  connections[j].insert(i);
90                  uniqueRoutes.insert(minmax(i, j));
91              }
92          }
93      }
94
```

```
95      // Add additional unique routes until we reach 54
96      while (uniqueRoutes.size() < numRoutes)
97      {
98          int a = rng() % stations.size();
99          int b = rng() % stations.size();
100         if (a != b && uniqueRoutes.find(minmax(a, b)) == uniqueRoutes.end())
101         {
102             uniqueRoutes.insert(minmax(a, b));
103             connections[a].insert(b);
104             connections[b].insert(a);
105         }
106     }
107
108     // Transfer unique routes to the vector
109     for (const auto& route : uniqueRoutes)
110     {
111         routes.emplace_back(route);
112     }
113
114     return routes;
115 }
```

- From line 1 to line 115, it is a code segment from Dataset 2 where it generates random stations and routes for the graph as explained in Dataset 2 section.

Section 1 : Build Graph Function

```
116
117  // Function to build the graph representation
118  vector<vector<pair<int, int>>> buildGraph(const vector<Station>& stations, const vector<pair<int, int>>& routes)
119  {
120      vector<vector<pair<int, int>>> graph(stations.size());
121
122      for (const auto& route : routes)
123      {
124          int a = route.first;
125          int b = route.second;
126          int distance = calculateDistance(stations[a], stations[b]);
127
128          graph[a].push_back({b, distance});
129          graph[b].push_back({a, distance});
130      }
131
132      return graph;
133  }
134
```

- This function is used to build a 2D vector of pairs called **'graphs'** that represent a graph where each station is a node and the connection is edge.
- It takes two parameters which are a vector of **'Station'** objects **'stations'** and a vector of pairs **'routes'** to represent connections between stations.
- Line 122 is the loop that iterates through each route in the 'routes' vector and each route represented by a pair of station indices as **'a'** and **'b'**.
- It will calculate distance between the two connected stations using the **'calculateDistance'** function and the distance will be used as the weight of the edge that connects the two stations.
- Line 128 and 129 will add the connection between the two stations to **'graph'** vector where line 128 will add connection to inner vector at index **'a'** and line 129 add connection to inner vector at index **'b'**.
- This pair then added to the adjacency list of station **'a'** in the **'graph'** vector, same as for **'b'**.
- Lastly, it will return a **'graph'** vector which now contains graph representation with all stations and their connections and distances.

Section 2 : Dijkstra's Algorithm Function

```
135 //Dijkstra's Algorithm
136 void DijkstraAlgorithm(const vector<vector<pair<int, int>>>& graph, int start, vector<int>& distance, vector<int>& previous)
137 {
138     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;     //declares priority queue 'pq' of pairs
139
140     distance[start] = 0;    //initializes distance starting node as 0
141     pq.push({0, start});    //initializes priority queue start at distance of 0
142
143     while(!pq.empty())
144     {
145         int current = pq.top().second ;
146         int currentDistance = pq.top().first ;
147         pq.pop();
148
149         if(currentDistance > distance[current])
150         {
151             continue ;
152         }
153
154         for(const auto& neighbour : graph[current])     //iterates through neighbors of current node in graph
155         {
156             int next = neighbour.first;
157             int weight = neighbour.second;
158
159             if(distance[current] + weight < distance[next])
160             {
161                 distance[next] = distance[current] + weight;    //if new distance is shorter, it will update the distance of the neighbour
162                 previous[next] = current;
163                 pq.push({distance[next], next});                //add neighbour to priority queue with updated distance
164             }
165         }
166     }
167 }
```

- This function implements Dijkstra's Algorithm to find the shortest path from a starting node in a weighted graph.
- The parameters are the **'graph'**, starting node **'start'** and two vectors which are **'distance'** and **'previous'**.
- The **'distance'** vector will store the shortest distance from starting node to each node and **'previous'** vector will store the previous node in shortest path.
- It uses priority queue as **'pq'** of pair to select the node with smallest first element as the highest priority and it initialises priority queue with starting node and a distance of 0.
- In line 143, the function will enter a loop that continues as long as the priority queue is not empty.
- Line 145 and line 146 will remove the node with highest priority from priority queue and store its distance and index in **'current'** and **'currentDistance'** variables.
- Line 149 will check if the currentDistance greater than the stored distance and if it is, the function will continue with next iteration of the loop
- Then, it will iterate through neighbours of the current node in the graph.
- Line 156 and 157 declare two variables **'next'** and **'weight'** and initialise them with the index of neighbour and the weight of edge that connect the current node and the neighbour.
- Then, line 159 will check if the distance to the neighbour through the current node is shorter than the stored distance.
- If it is, the line then will update the distance and previous vectors and add the neighbour to the priority queue with an updated distance.

Section 3 : Shortest Path Function

```
169  //Print shortest path
170  void shortestPath(const vector<int>& distance, const vector<int>& previous, int start)
171  {
172      for(int i=0; i<distance.size(); ++i)    //iterates through all stations in graph
173      {
174          cout << "Shortest path from Station " << start << " to Station " << i << " is " << distance[i] << endl ;
175          cout << "Shortest path : " ;
176
177          int current = i ;
178
179          while(current != -1)    //loop continue as long as current node is not -1
180          {
181              cout << current << " " ;
182              current = previous[current];    //update current note index to be its previous
183          }
184
185          cout << endl << endl ;
186      }
187  }
```

- This function used to print the shortest path from starting station to all other stations in the graph
- It takes three parameters, **'distance'** that contains the shortest distance from starting stations to each station, **'previous'** that contains the previous node in shortest path and **'start'** which is the starting station.
- It uses a loop to iterate over all stations in the graph and in the loop, it will start by printing the shortest distance from starting stations to current station **'i'** and it will print the beginning of the shortest path.
- Line 179 where it uses a **'while'** loop to print the sequence of stations in shortest path as long as the current station is not -1 to indicate that's the end of the shortest path.
- Then, it will print the index of current station and update current station to be its previous station in shortest path until it reaches the starting point that indicates by -1 in **'previous'** vector where the loop will stop to iterate.

Section 4 : Draw Shortest Path Graph Function

```
189  //Function to generate Grapgviz for shortest path Dijkstra's Algorithm
190  void graphShortestPath(const vector<Station>& stations, const vector<pair<int, int>>& routes, const vector<int>& previous)
191  {
192      ofstream dotFile("shortestPathGraph.dot");
193
194      dotFile << "graph G { " << endl ;
195
196      //Nodes
197      for(const Station& station : stations)
198      {
199          dotFile << "    " << station.name << " [label=\"" <<
200                  station.name << "\", shape=circle];" << endl;
201      }
202
203      //Edges
204      for(const auto& route : routes)
205      {
206          dotFile << "    " << stations[route.first].name << " -- "
207                  << stations[route.second].name;
208          dotFile << " [label=\"" << calculateDistance(stations[route.first], stations[route.second]) << "\"];" << endl;
209      }
210
211      //Shortest path
212      for(size_t i = 0 ; i < stations.size() ; ++i)
213      {
214          int current = i ;
215
216          while(current != -1 && previous[current] != -1)
217          {
218              dotFile << "    " << stations[current].name << " -- " << stations[previous[current]].name << " [color=red];" << endl;
219              current = previous[current] ;
220          }
221      }
222
223      dotFile << "}" << endl ;
224      dotFile.close();
225  }
```

- This function generates a Graphviz DOT file to visualise shortest path in a graph that uses Dijkstra's algorithm.
- The function uses three parameters which are:
    - A vector of **'Station'** objects **'stations'**
    - A vector of pairs **'routes'** representing the connections between stations
    - A vector **'previous'** storing the previous node in the shortest path.
- It opens an output file stream **'dotFile'** names as **'shortestPathGraph.dot**' for writing
- Next line is the beginning of Graphviz DOT file
- Starting from line 197, it will loop through all stations in the graph and write lines for each station in the DOT file. Each station represents a node with a label as its name and sets the shape to a circle.
- Then, in line 204, it iterates through the routes as edges and writes lines for each edge in the DOT file. It will write the weight of the edge which is the distance between the two stations in the Graphviz DOT file.
- Finally, it iterates over all stations in the **'stations'** vector where it will write the edge between the current station and its previous station in the shortest path in Graphviz DOT file with a red colour.
- The loop will keep on updating the current stations to be its previous stations in shortest path till it becomes 01 to indicate the end of shortest path.
- Function then completes the DOT file and close the output file stream.

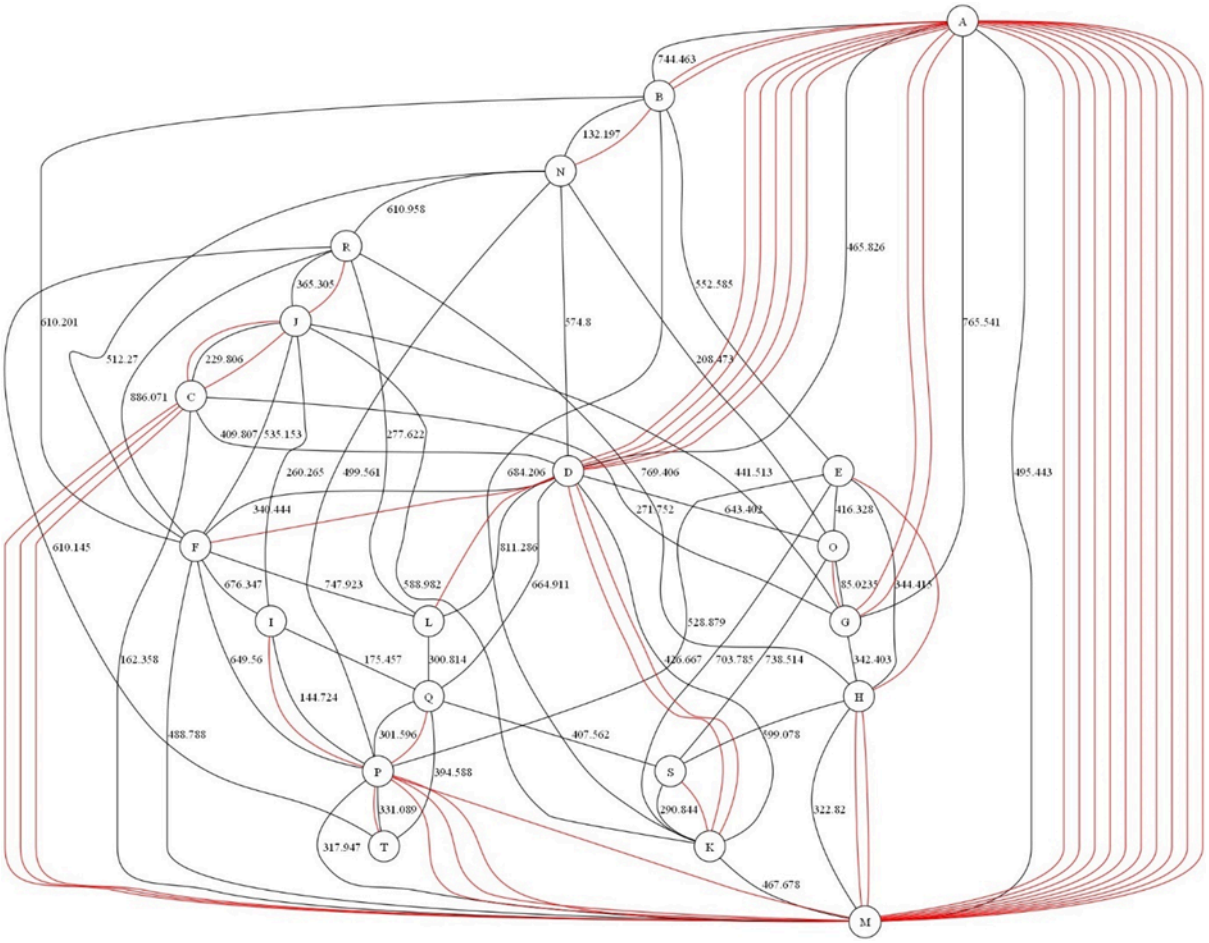
Section 5 : Main Function:

```cpp
int main()
{
    long long seedSum = 1201100116LL + 1201201773LL + 1201201862LL;
    vector<int> seedDigits;
    long long tempSeedSum = seedSum;

    while (tempSeedSum > 0)
    {
        seedDigits.push_back(tempSeedSum % 10);
        tempSeedSum /= 10;
    }
    reverse(seedDigits.begin(), seedDigits.end());

    vector<Station> stations = generateStations(static_cast<unsigned int>(seedSum), 20, seedDigits);
    vector<pair<int, int>> routes = generateRoutes(stations, 54);

    for (const Station& station : stations)
    {
        cout << station.name << " "
             << station.x << " "
             << station.y << " "
             << station.z << " "
             << station.weight << " "
             << station.profit << endl;
    }

    cout << endl ;
```

```cpp
    for (const auto& route : routes)
    {
        cout << "Route between " << stations[route.first].name
             << " (" << route.first << ")"
             << " and " << stations[route.second].name
             << " (" << route.second << ")"
             << " Distance: " << calculateDistance(stations[route.first], stations[route.second])
             << endl;
    }

    vector<vector<pair<int, int>>> graph = buildGraph(stations, routes);

    int startStation = 0 ;

    vector<int> distance(stations.size(), numeric_limits<int> :: max());
    vector<int> previous(stations.size(), -1);

    DijkstraAlgorithm(graph, startStation, distance, previous);

    cout << endl ;

    //print Shortest Path
    shortestPath(distance, previous, startStation);

    // Generate Graphviz DOT file
    graphShortestPath(stations, routes, previous);

    // Inform the user
    cout << "Graphviz DOT file generated: shortestPathGraph.dot" << endl;

    return 0;
}
```

- In the main function, we will declare a vector **‘graph’** of a vector of pairs of integers and initialise it by calling the **‘buildGraph’** function with **‘stations’** and **‘routes’** as arguments.
- Then, declare variable **‘startStation’** and initialise it with 0, vector **‘distance’** of integers and initialise it with the maximum value of an integer for each element and vector **‘previous’** with initialization of -1 of each element.
- Call the ‘**Dijkstra Algorithm’** function to find the shortest paths from a starting station to all other stations.
- Call the **‘shortestPath’** function to print information about the shortest path.
- Call the **‘graphShortestPath’** function with **‘stations'**, **‘routes’** and **‘previous’** as arguments to visualise the graph with highlighted shortest paths.
- Lastly, print a message to indicate that the Graphviz DOT file has been generated.

Section 6 : Graph for Shortest Path using Dijsktra's Algorithm:



To get the .png for the graph, use the command prompt and navigate it to the same directory as the code file and run **'dot -Tpng shortestPathGraph.dot -o shortestPathGraph.png'.**

Section 7 : Time Complexity for Dijkstra's Algorithm:

1. Generating Station
    - The code will iterate over the number stations, resulting in time complexity of O(V) where V is the number of vertices(stations).
2. Generating Routes
    - Process of generating routes involves selecting random routes until the desired number is reached, resulting in time complexity of **O(E)** where E is the number of edges(routes).
3. Building Graph
    - Building the graph involves iterating over the routes and adding edges to the graph, leading to time complexity of **O(E)**.
4. Dijsktra's Algorithm
    - The core of the algorithm, Dijkstra's Algorithm runs in **O((V+E)*log(V))** time

- Combining all the steps, the overall time complexity of the algorithm is dominated by Dijkstra's algorithm and is expressed as **O((V + E) * log(V))**, where V corresponds to the number of vertices (stations) and E corresponds to the number of edges (routes).

Section 8 : Space Complexity for Dijkstra's Algorithm:

1. Stations and Routes
    - Space complexity to store the stations and routes is **O(V + E)** where V is the number of vertices (stations) and E is the number of edges (routes).
2. Building Graph
    - Space complexity to story graph also **O(V + E)** as it involves creating an adjacency list representation of the graph.
3. Dijkstra's Algorithm
    - Space complexity for Dijkstra's algorithm is **O(V)** for storing the distances and precious nodes.

- Combining all steps, the overall space complexity is **O(V + E)** where V corresponds to the number of vertices (stations) and E corresponds to the number of edges (routes).

Kruskal Algorithm

From Dataset 2, we can identify the Minimum Spanning Tree using Kruskal's Algorithm.

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <random>
#include <algorithm>
#include <set>
#include <map>
#include <queue>
#include <iterator>
#include <fstream>

using namespace std;

struct Station
{
    string name;
    int x, y, z, weight, profit;
};

double calculateDistance(const Station& a, const Station& b)
{
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2) + pow(a.z - b.z, 2));
}
```

```cpp
// Function to generate random number from seed digits
int generateRandomNumber(mt19937& rng, const vector<int>& digits, int numDigits)
{
    uniform_int_distribution<> dist(0, digits.size() - 1);
    int result = 0;
    for (int i = 0; i < numDigits; ++i)
    {
        int digit = digits[dist(rng)];
        // Ensure the first digit is non-zero when numDigits is more than 1
        if (i == 0 && numDigits > 1)
        {
            while (digit == 0)
            {
                digit = digits[dist(rng)];
            }
        }
        result = result * 10 + digit;
    }

    return result;
}
```

```cpp

// Function to generate stations with random properties
vector<Station> generateStations(unsigned int seed, int numStations, const vector<int>& seedDigits)
{
    vector<Station> stations;
    mt19937 rng(seed);

    for (int i = 0; i < numStations; ++i)
    {
        Station station;
        // Map station index to a letter (A-Z)
        char stationLetter = 'A' + i;
        station.name = string(1, stationLetter);
        station.x = generateRandomNumber(rng, seedDigits, 3);
        station.y = generateRandomNumber(rng, seedDigits, 3);
        station.z = generateRandomNumber(rng, seedDigits, 3);
        station.weight = generateRandomNumber(rng, seedDigits, 2);
        station.profit = generateRandomNumber(rng, seedDigits, 2);
        stations.push_back(station);
    }

    return stations;
}

```

```cpp

// Function to generate a map of connections between stations
vector<pair<int, int>> generateRoutes(const vector<Station>& stations, int numRoutes)
{
    vector<pair<int, int>> routes;
    set<pair<int, int>> uniqueRoutes;
    mt19937 rng(random_device{}());
    map<int, set<int>> connections;

    // Ensure all stations have at least 3 connections
    for (int i = 0; i < stations.size(); ++i)
    {
        while (connections[i].size() < 3)
        {
            int j = rng() % stations.size();
            if (i != j && connections[i].find(j) == connections[i].end())
            {
                connections[i].insert(j);
                connections[j].insert(i);
                uniqueRoutes.insert(minmax(i, j));
            }
        }
    }

```

```
95      // Add additional unique routes until we reach 54
96      while (uniqueRoutes.size() < numRoutes)
97      {
98          int a = rng() % stations.size();
99          int b = rng() % stations.size();
100         if (a != b && uniqueRoutes.find(minmax(a, b)) == uniqueRoutes.end())
101         {
102             uniqueRoutes.insert(minmax(a, b));
103             connections[a].insert(b);
104             connections[b].insert(a);
105         }
106     }
107
108     // Transfer unique routes to the vector
109     for (const auto& route : uniqueRoutes)
110     {
111         routes.emplace_back(route);
112     }
113
114     return routes;
115 }
```

- From line 1 to line 115, it is a code segment from Dataset 2 where it generates random stations and routes for the graph as explained in Dataset 2 section.

Section 1 : Structure for Edges

```
122  //Structure for edge between two stations
123  struct Edge
124  {
125      int u, v;
126      double distance;
127
128      Edge(int x, int y, double dist)
129      {
130          u = x ;
131          v = y ;
132          distance = dist ;
133      }
134  };
```

- This structure is used for edges in the graph, and instances of this structure will store information about information between two stations, including the indices of the connected stations and the distance between them.
- It declares integer members **'u'** and **'v'** to represent indices of two stations connected by edge and double member **'distance'** to represent distance that is associated with the edge.
- The constructor on line 128 initialises the **'u'**, **'v'**, and **'distance'** values with the provided parameters **'x'**, **'y'**, and **'dist'**, respectively.

```
136  bool comp(const Edge &a, const Edge &b)
137  {
138      return a.distance < b.distance ;
139  }
```

- Line 136 declares a function named **'comp'** that returns a boolean value. It takes two **'Edge'** objects which are **'a'** and **'b'**, as the constant references.
- This purpose of the function is to serve as a comparison function for sorting and it will compare two **'Edge'** objects based on their 'distance' member.
- The function will return **'true'** if the **'distance'** of **'a'** is less than **'distance'** of **'b'**, and return **'false'** otherwise.

Section 2 : Union-find data structure

```
141  //find function - to determine root of set
142  int find(int u, std::vector<int> &parent)
143  {
144      if(u == parent[u])      //check if u is its own parent
145          return u;
146
147      return parent[u] = find(parent[u], parent) ;
148  }
```

- This is a function **'find'** that takes element **'u'** as representative of node or element and a reference to the vector of integers 'parent'. Purpose of this function is to find the root of the set which element **'u'** belongs.
- Line 144 is a base case check to check if element **'u'** is its own parent, and if it is, then the function will immediately return **'u'**.
- If **'u'** is not its own parent, the function will recursively call itself with the parent of **'u'** as an argument.
- The recursive call will continue until the root of the set is found and the path compression techniques also applied here by updating the parent of u directly to the root.

```
150  //union function - perform union of two sets
151  void unionn(int u, int v, std::vector<int> &parent, vector<int> &rank)
152  {
153      //find root to which u and v belong to
154      u = find (u, parent);
155      v = find (v, parent);
156
157      if(rank[u] < rank[v])
158      {
159          parent[u] = v ;
160      }
161      else if(rank[v] < rank[u])
162      {
163          parent[v] = u ;
164      }
165      else
166      {
167          parent[v] = u ;
168          rank[u] ++ ;
169      }
170  }
```

- This function performs as the union of two sets in which elements **'u'** and **'v'** belong. It also takes the vector **'parent'** and the vector **'rank'**.
- Line 154 and 155 are the roots of sets which **'u'** and **'v'** belong to are found using the previous **'find'** function.
- Following the **'if-else'** condition, it handle union operation based on the ranks of the sets where:
    - If the rank of the set with root **'u'** is less than the rank of the set with root **'v'**, the set will be parent of **'u'** and **'v'**.
    - If the rank of the set with root **'u'** is more than the rank of the set with root **'v'**, the set will be the parent of **'v'** and **'u'**.
    - If the ranks are equal, set will be parent of **'v'** and **'u'** and increment rank of **'u'**

Section 3 : Kruskal Algorithm Function

```
172  vector<Edge> KruskalsAlgorithm (const vector<Edge> &edges, int numOfVertices)
173  {
174      vector<Edge> minimumSpanningTree;
175
176      //initializes parent array
177      vector<int> parent(numOfVertices);
178
179      for(int i = 0 ; i < numOfVertices ; i++)
180          parent[i] = i;
181
182      vector<int> rank(numOfVertices, 0);
183
184      //sort edges in non-decreasing order of distance
185      vector<Edge> sortedEdges = edges ;
186      sort(sortedEdges.begin(), sortedEdges.end(), comp);
187
188      for(const Edge &edge : sortedEdges)
189      {
190          if(find(edge.u, parent) != find(edge.v, parent))
191          {
192              minimumSpanningTree.push_back(edge);
193              unionn(edge.u, edge.v, parent, rank);
194          }
195      }
196
197      return minimumSpanningTree ;
198  }
```

- The function is to implement Kruskal's algorithm to find the minimum spanning tree (MST) in the graph. It takes two arguments:
    - Vector of **'Edge'** as **'edges'** to represent edges of the graph
    - Integer variable **'numOfVertices'** to represent number of vertices in the graph
- Line 174 is where the vector will store the edges of the minimum spanning tree.
- The function initialises a vector of integers **'parent'** to represent the parent of each vertex in a disjoint set data structure and initially, each vertex is its own parent.
- Function also initialises the vector of integers **'rank'** to represent the depth of each tree in forest and it is set to 0 for all vertices.
- Line 185 creates a copy of the input **'edges'** vector and sorts it in nondecreasing order of distance using the **'comp'** function.
- Then it starts to iterate over the sorted edges and it will check if adding the edge to the minimum spanning tree would create a cycle. It's done by comparing the root of sets to which the vertices **'edge.u**' and '**edge.v**' belong using the **'find'** function.
- Line 192, if the edge does not create a cycle, edge will be added to **'minimumSpanningTree'** vector.
- Function will call 'union' function to perform union operation where it merge the sets which vertices **'edge.u'** and **'edge.v'** belong
- Lastly, it will return 'minimumSpanningTree' which now contain edges of minimum spanning tree.

Section 4 : Draw Minimum Spanning Tree Function

```
200  //Function to generate Graphviz for Minimum Spaning Tree
201  void graphMST(const vector<Station>& stations, const vector<Edge>& minimumSpanningTree)
202  {
203      ofstream dotFile("MSTGraph.dot");
204
205      dotFile << "graph MST { " << endl ;
206
207      //Vertices
208      for (int i = 0; i < stations.size(); ++i)
209      {
210          dotFile << "  " << i << " [label=\"" <<
211              stations[i].name << "\"];" << endl;
212      }
213
214      //Edges
215      for (const auto& edge : minimumSpanningTree)
216      {
217          dotFile << "  " << edge.u << " -- " <<
218              edge.v << " [label=\"" << edge.distance << "\"];" << endl;
219      }
220
221      dotFile << "}" << endl ;
222      dotFile.close();
223  }
```

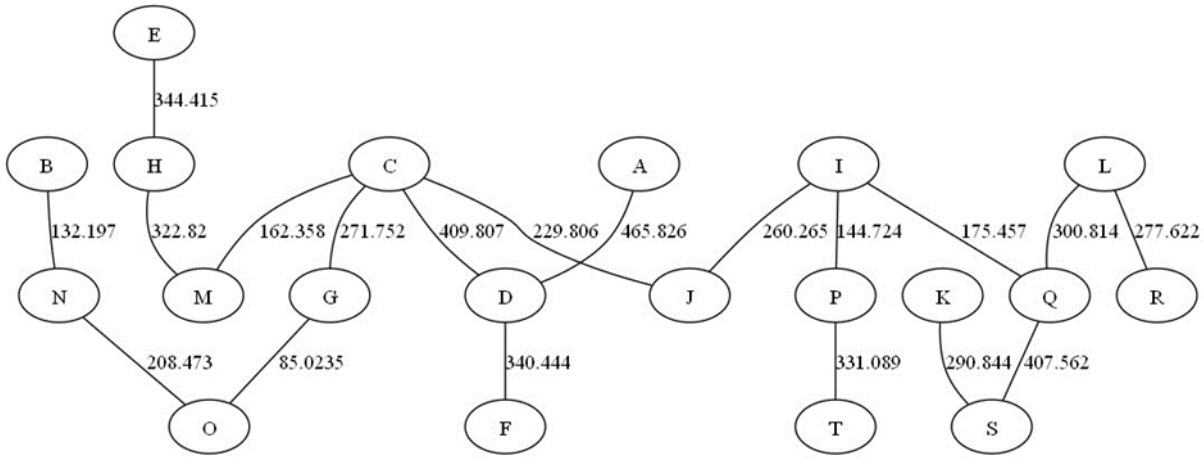
- This function generates a Graphviz DOT file to visualise minimum spanning tree and its corresponding graph.
- The function uses two parameters:
  - A vector of **'Station'** objects **'stations'** that represent vertices of graph
  - A vector of **'Edge'** as **'minimumSpanningTree'** to represent the minimum spanning tree of the graph.
- It opens an output file stream **'dotFile'** names as **'MSTGraph.dot**' for writing
- Next line is the beginning of Graphviz DOT file where it specifies that the graph names "**MST**"
- Line 208 to line 212 will iterate through each vertex in **'stations'** vector and it writes in DOT statement for each vertex of minimum spanning tree.
- Line 215 to line 219 then iterates over each edge in **'minimumSpanningTree'** and writes a DOT statement for each edge. It will specify a connection between vertices '**edge.u**' and '**edge.v**', with a label indicating the distance of the edge.
- Function then completes the DOT file and close the output file stream.

Section 5 : Main Function

```
225  int main()
226  {
227      long long seedSum = 1201100116LL + 1201201773LL + 1201201862LL;
228      vector<int> seedDigits;
229      long long tempSeedSum = seedSum;
230
231      while (tempSeedSum > 0)
232      {
233          seedDigits.push_back(tempSeedSum % 10);
234          tempSeedSum /= 10;
235      }
236
237      reverse(seedDigits.begin(), seedDigits.end());
238
239      vector<Station> stations = generateStations(static_cast<unsigned int>(seedSum), 20, seedDigits);
240      vector<pair<int, int>> routes = generateRoutes(stations, 54);
241
242      for (const Station& station : stations)
243      {
244          cout << station.name << " "
245                    << station.x << " "
246                    << station.y << " "
247                    << station.z << " "
248                    << station.weight << " "
249                    << station.profit << endl;
250      }
251
252      cout << endl ;
253
254      vector<Edge> edges;  // Convert routes to edges
255
256      for (const auto& route : routes)
257      {
258          double distance = calculateDistance(stations[route.first], stations[route.second]);
259          edges.emplace_back(route.first, route.second, distance);
260      }
261
262      vector<Edge> minimumSpanningTree = KruskalsAlgorithm(edges, stations.size());
263
264      cout << "\nMinimum Spanning Tree Edges : " << endl ;
265
266      for(const auto&edge : minimumSpanningTree)
267      {
268          cout << "Edge between " << stations[edge.u].name
269               << " (" << edge.u << ")"
270               << " and " << stations[edge.v].name
271               << " (" << edge.v << ")"
272               << " Distance: " << edge.distance
273               << endl;
274      }
275
276      cout << endl ;
277
278      // Generate Graphviz DOT file
279      graphMST(stations, minimumSpanningTree);
280
281      // Inform the user
282      cout << "Graphviz DOT file generated: MSTGraph.dot" << endl;
283
284      return 0;
285  }
```

- In the main function, we will convert routes to edges and from line 256 to line 260, the distance between the connected stations will be calculated.
- Call the **'KruskalAlgorithm'** function to find the minimum spanning tree using the generated edges.
- Line 266 to line 274 prints the edges of the minimum tree to the console.
- Call **'graphMST'** function to generate a Graphviz DOT file to visualise the minimum spanning tree.
- Lastly, print a message to indicate that the Graphviz DOT file has been generated.

Section 6 : Graph for Minimum Spanning Tree



Section 7 : Time Complexity for Minimum Spanning Tree

1. Generate Edges from Routes
    - The code iterates over routes and calculates the distance between stations to create the edges. It will take O(E) time where E is the number of edges (routes) in the graph.
2. Sorting Edges
    - The edges will be sorted based on their distances. Sorting has a time complexity of O(E log E).
3. Union-Find Data Structure
    - Kruskal Algorithms use union-find data structure to detect cycles and build the minimum spanning tree. The time complexity of Kruskal's Algorithm is O (E log E), where E is the number of edges in the graph. This complexity is because the algorithm uses a priority queue, and each union-find operation takes O(log E) time.
    -

- Combining all the steps, overall time complexity will be **O(E log E)**.

Section 8 : Space Complexity for Minimum Spanning Tree

1. Edges
    - The edges are stored in a vector that will contribute to space complexity of O(E), where E is the number of edges.
2. Union-Find Data Structure
    - The space complexity for the union-find data structure is O(V), where V is the number of vertices. This includes the parent and rank arrays, which are used to keep track of subsets and perform union-find operations.

- Combining these, the overall space complexity is **O(E + V)**.

Question 4: Dynamic Programming

0/1 Knapsack Algorithm

Section 1: Header Files

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Include this header for 'reverse'

using namespace std;
```

This part contains essential header files: <iostream> for input/output operations, <vector> for using vectors, and <algorithm> for the reverse function. The "using namespace std;" statement is employed to circumvent the need to explicitly write "std::" before standard library identifiers.

Section 2: Define a Structure called “Station”

```cpp
struct Station {
    string name;
    int weight;
    int profit;
};
```

A data structure called Station is defined to represent a station. The station consists of three members: name (a string), weight (an integer indicating the station's weight), and profit (an integer indicating the station's profit).

Section 3: Define Function to Load Dataset

```
13  vector<Station> loadDataset() {
14      // Define the dataset as an array of Station structs.
15      vector<Station> dataset = {
16          {"Station A", 250, 100},
17          {"Station B", 125, 80},
18          {"Station C", 315, 120},
19          {"Station D", 670, 230},
20          {"Station E", 420, 140},
21          {"Station F", 220, 90},
22          {"Station G", 315, 110},
23          {"Station H", 670, 220},
24          {"Station I", 420, 130},
25          {"Station J", 220, 85}
26          // Add more stations as needed
27      };
28
29      return dataset;
30  }
```

The loadDataset function initialises and returns a vector of Station structs, which represents the dataset of stations. There are ten stations specified in this scenario, each with its respective names, weights, and profits.

Section 4: Define Function to Solve Knapsack

```
32  vector<Station> solveKnapsack(const vector<Station>& stations, int maxCapacity) {
33      int numStations = stations.size();
34      // Create a 2D table to store the maximum profit for each station and capacity combination.
35      vector<vector<int>> dp(numStations + 1, vector<int>(maxCapacity + 1, 0));
36
37      // Fill the dp table using dynamic programming.
38      for (int i = 1; i <= numStations; ++i) {
39          for (int w = 1; w <= maxCapacity; ++w) {
40              if (stations[i - 1].weight <= w) {
41                  dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - stations[i - 1].weight] + stations[i - 1].profit);
42              } else {
43                  dp[i][w] = dp[i - 1][w];
44              }
45          }
46      }
47
48      // Trace back to find the selected stations.
49      int i = numStations;
50      int w = maxCapacity;
51      vector<Station> selectedStations;
52
53      while (i > 0 && w > 0) {
54          if (dp[i][w] != dp[i - 1][w]) {
55              selectedStations.push_back(stations[i - 1]);
56              w -= stations[i - 1].weight;
57          }
58          --i;
59      }
60
61      reverse(selectedStations.begin(), selectedStations.end());
62      return selectedStations;
63  }
```

- The solveKnapsack function accepts two parameters: the dataset of stations (stations) and the maximum capacity of the knapsack (maxCapacity). The algorithm use dynamic programming to populate a two-dimensional table (dp) that represents the highest possible profit for each combination of station and capacity.
- The nested loops cycle over each station and every potential knapsack capacity, modifying the dynamic programming table according to the most advantageous selections. The solution use the conventional dynamic programming methodology to solve the 0/1 Knapsack problem.
- Upon completing the dynamic programming table, the function retraces its steps to identify the chosen stations that yield the highest profit. The chosen stations are kept in the selectedStations vector, and their sequence is inverted using the reverse function.

Section 5: Main Function

```
65  int main() {
66      vector<Station> stations = loadDataset();
67      int maxCapacity = 800;
68
69      vector<Station> selectedStations = solveKnapsack(stations, maxCapacity);
70
71      // Output the selected stations and their properties.
72      cout << "Selected Stations:" << endl;
73      for (const Station& station : selectedStations) {
74          cout << "Station Name: " << station.name << endl;
75          cout << "Weight: " << station.weight << " Profit: " << station.profit << endl;
76      }
77
78      return 0;
79  }
```

The main function initialises the dataset of stations, establishes the maximum knapsack capacity, and invokes the solveKnapsack function. Subsequently, it generates a display of the chosen stations and their attributes, encompassing the station's name, weight, and profit. The ultimate outcome is displayed on the console.

Section 6: Result

```
Selected Stations:
Station Name: Station A
Weight: 250 Profit: 100
Station Name: Station B
Weight: 125 Profit: 80
Station Name: Station E
Weight: 420 Profit: 140
```

- The algorithm employs dynamic programming to determine the most advantageous arrangement of stations that maximises the overall profit, while adhering to the limitation of the knapsack's maximum capacity.
- The selection process involves analysing the highest attainable profit for each combination of station and capacity. The algorithm takes into account the balance between the weight of the stations and their corresponding revenues in order to choose the most lucrative combination that adheres to the specified capacity limitation.
- To summarise, the output is the solution to the 0/1 Knapsack problem. The algorithm has effectively chosen the stations to maximise profit while adhering to the capacity constraints of the knapsack.

Section 7: Time And Space Complexity

Time Complexity

The algorithm's time complexity is O(numStations * maxCapacity), where numStations represents the total number of stations in the dataset and maxCapacity denotes the highest possible capacity of the knapsack.

- The outer loop iterates across each station in the dataset for a total of ‘numStations’ iterations.
- The inner loop iterates over each conceivable knapsack capacity from 1 to maxCapacity, running for a total of maxCapacity iterations.
- The algorithm executes constant-time operations within the loops.

Space Complexity

The algorithm's space complexity is represented by the Big O notation O(numStations * maxCapacity).

- The algorithm uses a two-dimensional array, dp, with dimensions (numStations + 1) x (maxCapacity + 1), to record the highest profit for each combination of station and capacity.
- The size of this array is closely correlated with both the number of stations and the maximum knapsack capacity.
- Hence, the space complexity can be expressed as O(numStations * maxCapacity).