ALGORITHM ANALYSIS DESIGN

Group 9

1. Mannoj Sakthivel (Leader)
2. Virosshen A/L Sivasankar
3. Noor Elyana Bt Mohd Hasim
4. Thaqif Iskandar Bin Aminuddin

QUESTION 1: Dataset 1

```
// 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);
```

Dataset Generation Function

- 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.

```
// 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;
```

Dataset Generation For-Loop

- 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.

```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;
}
```

Main Function

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

```
// 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;
}
```

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);
```

Calculating Unique Seeds

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

Storing Datasets in a Map

QUESTION 1: Dataset 2

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

Station Struct

- 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.

```
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));
}
```

Distance Calculation Function

- 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.

```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;
}
```

Random Number Generation Function

- 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.

```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;
}
```

Generating Stations

- 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.

```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));
            }
        }
    }

    // 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;
}
```

Generating 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.

```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;
}
```

Main Function

- 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.

```cpp
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;
}
```

Printing Stations and Printing Routes

- **First Loop (Printing Stations):**
    - Iterates through each **Station** object in the **stations** vector.
- **Second Loop (Printing Routes):**
    - Iterates through each route in the routes vector.

QUESTION 2:

- Heap Sort
- Merge Sort

Heap Sort Algorithm

```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);
    }
}
```

Helper Function to Heapify:

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

Heap Sort Algorithm

```
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  }
```

Heap Sort Function:

- “timings” : This vector stores the execution time of the Heap Sort.
- Line 69 - 70 = Builds the max heap scenario
- Line 73-76 = This loops extracts elements from the heap one by one.

Heap Sort Algorithm

```
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  }
```

Function to check array is sorted:

- “arr” is vector to be checked
- utilizes the std::is_sorted function to check if vector is sorted.

Heap Sort Algorithm

```
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
```

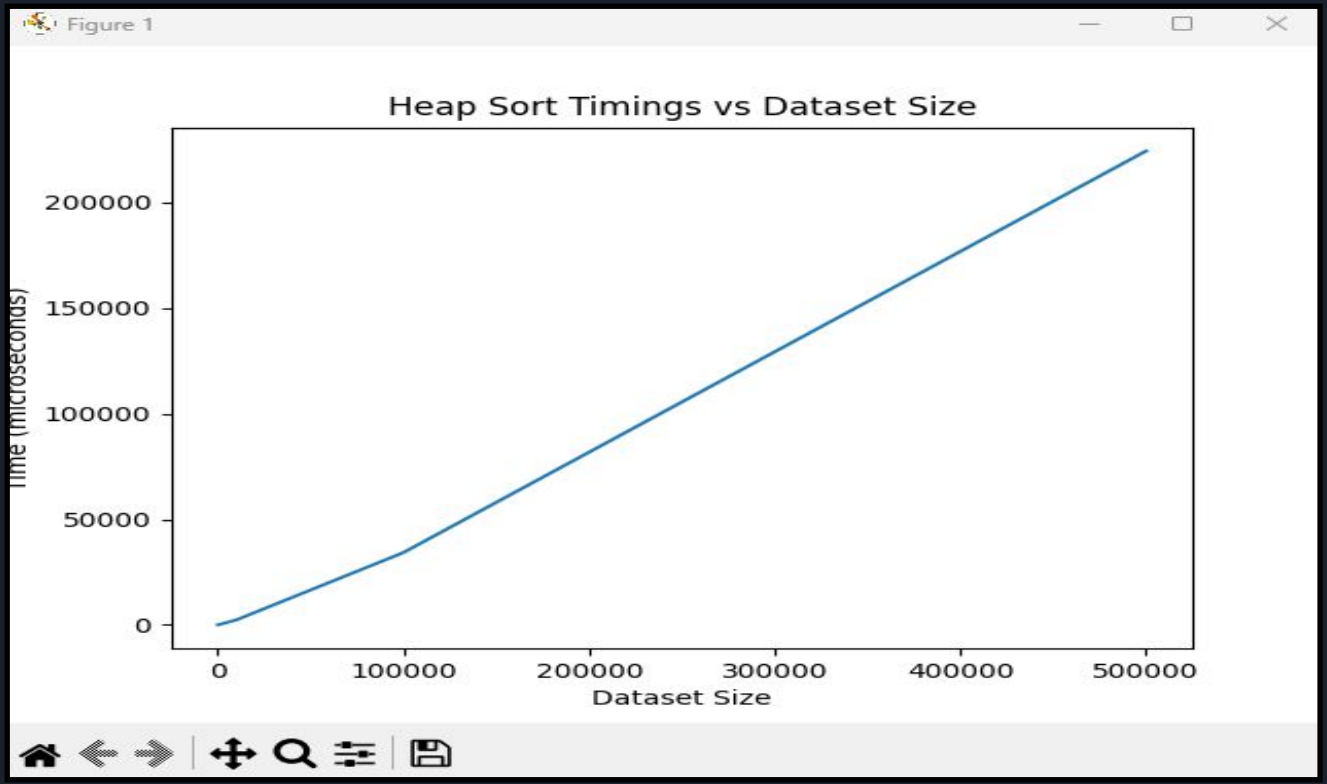
Function to plot the graph:

- Uses the “matplotlibcpp” library to plot the graph to show relationship between datasize and execution time.

Heap Sort Algorithm

```
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     }
```

Heap Sort Algorithm



Merge Sort Algorithm

```
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  }
```

Main Merge Sort Function:

- ‘mergeSort’: The main merge sort function, divides array into half , sorts and then merge them back in together.

Merge Sort Algorithm

```cpp
// Function to perform Merge Sort and measure time
void runMergeSort(std::vector<int>& arr, std::vector<long long>& timings) {
    auto start_time = std::chrono::high_resolution_clock::now();

    int n = arr.size();

    // Perform Merge Sort
    mergeSort(arr, 0, n - 1);

    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start_time);
    timings.push_back(duration.count());
}
```

Run Merge Sort Function:

- Function to perform Merge Sort and measure time, through mergeSort and calculates the duration of the sorting process.

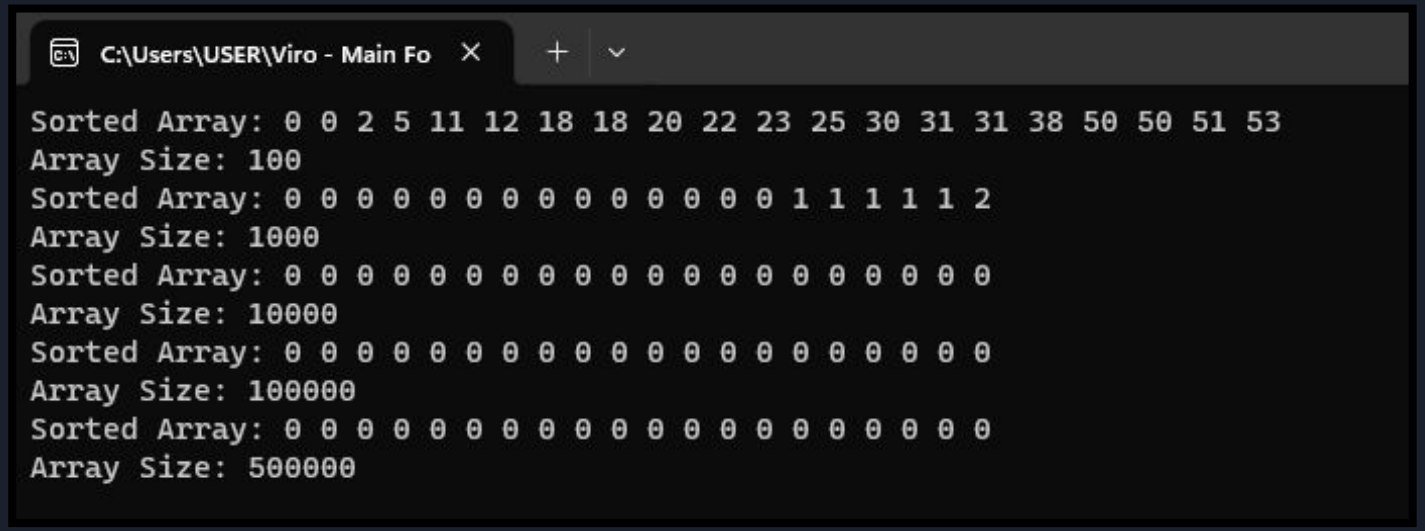
Merge Sort Algorithm

```cpp
int main() {
    // Specify dataset sizes
    std::vector<int> setSizes = {100, 1000, 10000, 100000, 500000};

    // Create vectors to store timings
    std::vector<long long> mergeSortTimings;

    // Generate datasets and perform Merge Sort
    for (int size : setSizes) {
        long long groupLeaderId = 1221303085;
        auto idDigits = getDigitsFromId(groupLeaderId);
        long long seed = groupLeaderId * 5;  // Use seed for Set 5
        std::vector<int> dataset = generateDataset(seed, size, idDigits);

        std::vector<int> mergeSortData = dataset;
        runMergeSort(mergeSortData, mergeSortTimings);

        // Check if the array is sorted
        if (!isSorted(mergeSortData)) {
            std::cerr << "Error: The array is not sorted after Merge Sort.\n";
            return 1;
        }

        // Print the first few elements of the sorted array
        printArray(mergeSortData);
    }

    // Plot the graph
    plotGraph(setSizes, mergeSortTimings, "Merge Sort Timings vs Dataset Size");

    return 0;
}
```

Main Function:

- Specify the dataset sizes
- Create vectors to store timing
- Generate Datasets and perform merge sort
- Plot the graph using matplotlibcpp

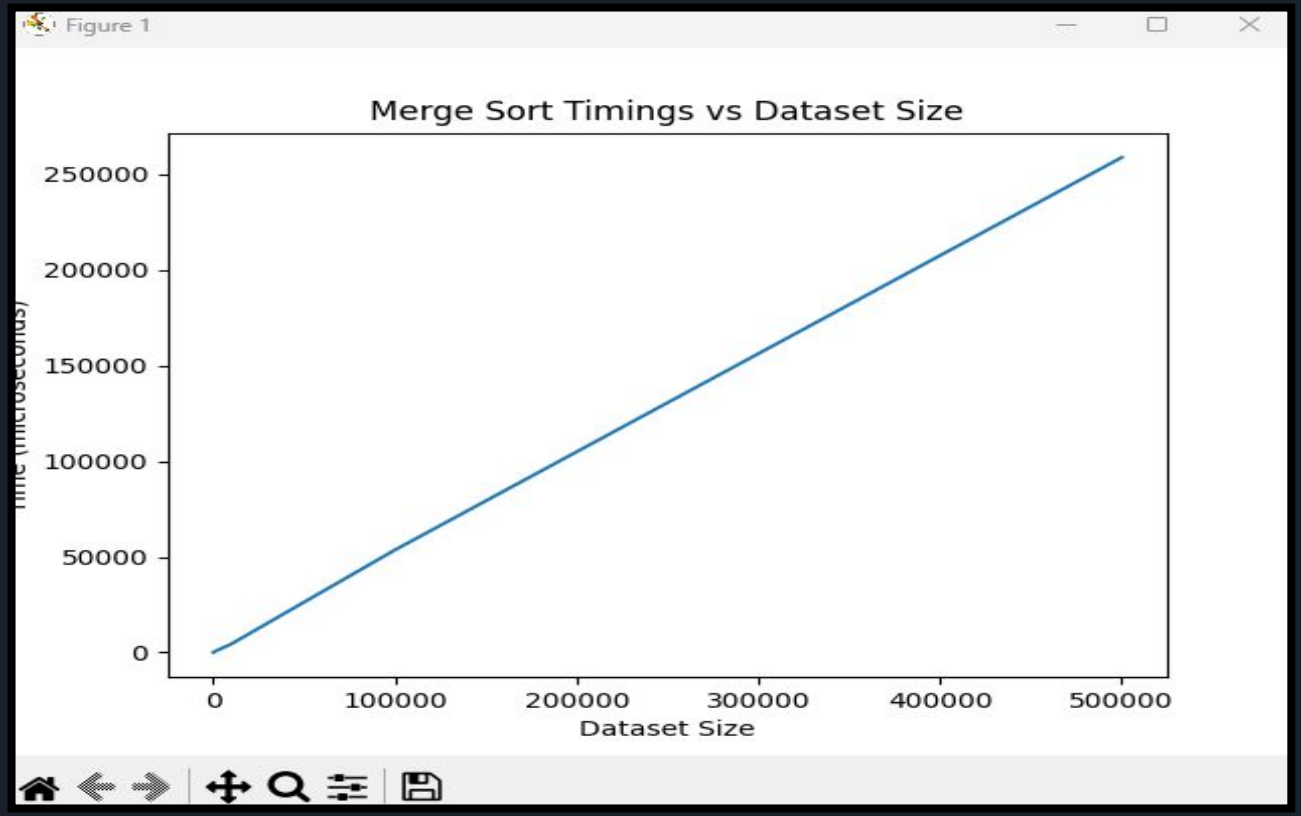
Merge Sort Algorithm



The Result:

- 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.

Merge Sort Algorithm



QUESTION 3:

- Dijkstra’s Algorithm
- Kruskal Algorithm

DIJKSTRA'S ALGORITHM

- Dataset 2

```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;
}

```

```
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 Dataset 2

- It will generate random stations and routes.
- Can identify shortest path from Station A to other stations using Dijkstra's Algorithm

DIJKSTRA'S ALGORITHM
- 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 builds a 2D vector of vector of pairs representing a graph with stations as nodes and connections as edges.
- It iterates through each route, calculates the distance between connected stations, and adds the connection with its weight to the graph vector

DIJKSTRA'S ALGORITHM
- Dijkstra's Algorithm Function

```cpp
//Dijkstra's Algorithm
void DijkstraAlgorithm(const vector<vector<pair<int, int>>>& graph, int start, vector<int>& distance, vector<int>& previous)
{
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;     //declares priority queue 'pq' of pairs

    distance[start] = 0;    //initializes distance starting node as 0
    pq.push({0, start});    //initializes priority queue start at distance of 0

    while(!pq.empty())
    {
        int current = pq.top().second ;
        int currentDistance = pq.top().first ;
        pq.pop();

        if(currentDistance > distance[current])
        {
            continue ;
        }

        for(const auto& neighbour : graph[current])     //iterates through neighbors of current node in graph
        {
            int next = neighbour.first;
            int weight = neighbour.second;

            if(distance[current] + weight < distance[next])
            {
                distance[next] = distance[current] + weight;    //if new distance is shorter, it will update the distance of the neighbour
                previous[next] = current;
                pq.push({distance[next], next});                //add neighbour to priority queue with updated distance
            }
        }
    }
}
```

- The function applies Dijkstra's Algorithm to identify the shortest path
- It utilizes two vectors, distance and previous, to store the shortest distance and the previous node in the optimal path, respectively.
- It will use a priority queue to explore paths and to select nodes with the smallest distance as the highest priority.

DIJKSTRA'S ALGORITHM
- 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 print the shortest path from from starting stations to all other stations in the graph

- It iterates through all stations, printing the shortest distance and the sequence of stations in the shortest path using a 'while' loop until it reaches the starting point indicated by -1 in the 'previous' vector.

DIJKSTRA'S ALGORITHM
- Draw Shortest Path Function

```cpp
//Function to generate Grapgviz for shortest path Dijkstra's Algorithm
void graphShortestPath(const vector<Station>& stations, const vector<pair<int, int>>& routes, const vector<int>& previous)
{
    ofstream dotFile("shortestPathGraph.dot");

    dotFile << "graph G { " << endl ;

    //Nodes
    for(const Station& station : stations)
    {
        dotFile << "    " << station.name << " [label=\"" <<
                   station.name << "\", shape=circle];" << endl;
    }

    //Edges
    for(const auto& route : routes)
    {
        dotFile << "    " << stations[route.first].name << " -- "
                << stations[route.second].name;
        dotFile << " [label=\"" << calculateDistance(stations[route.first], stations[route.second]) << "\"];" << endl;
    }

    //Shortest path
    for(size_t i = 0 ; i < stations.size() ; ++i)
    {
        int current = i ;

        while(current != -1 && previous[current] != -1)
        {
            dotFile << "    " << stations[current].name << " -- " << stations[previous[current]].name << " [color=red];" << endl;
            current = previous[current] ;
        }
    }

    dotFile << "}" << endl ;
    dotFile.close();
}
```

- This function generates a Graphviz DOT file to visualize the shortest path in a graph using Dijkstra's algorithm.
- It takes 'stations' (vector of Station objects), 'routes' (vector of pairs representing connections), and 'previous' (vector storing the previous node in the shortest path)

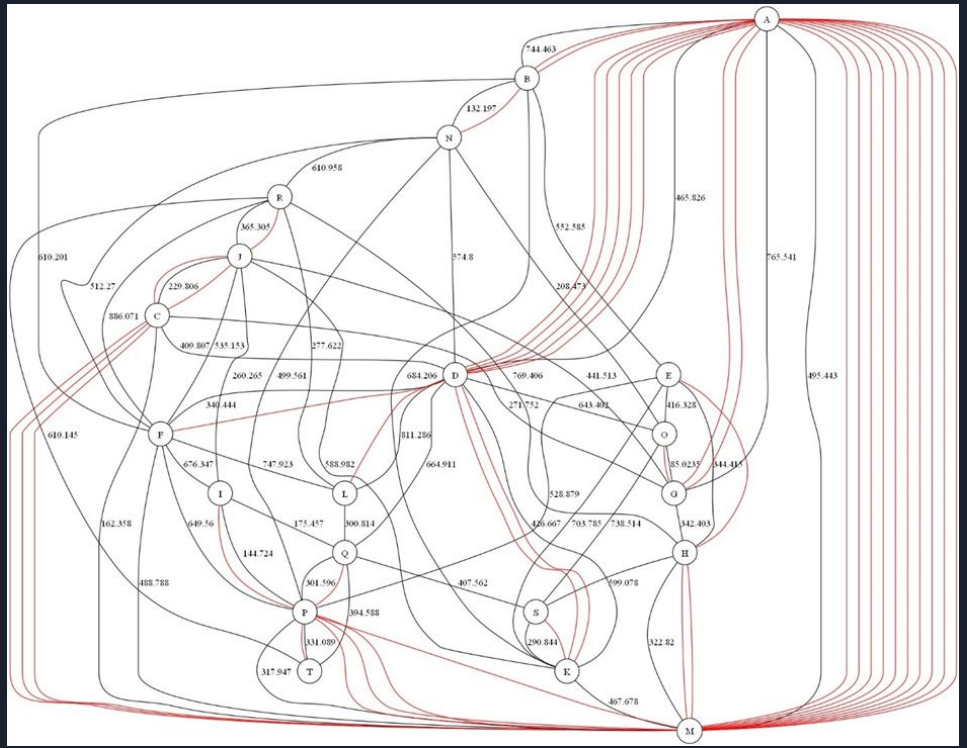
DIJKSTRA'S ALGORITHM
- Main Function

```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;
}
```

DIJKSTRA'S ALGORITHM

- Graph



DIJKSTRA'S ALGORITHM

- Time and Space Complexity

- Time complexity for 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)

- Space complexity for Dijkstra's algorithm is **O(V + E)** where V corresponds to the number of vertices (stations) and E corresponds to the number of edges (routes).

KRUSKAL ALGORITHM

- Dataset 2

```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;
}

```

```
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 Dataset 2

- It will generate random stations and routes.
- Can identify the Minimum Spanning Tree using Kruskal's Algorithm

KRUSKAL ALGORITHM
- 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  };
```

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

- This structure represents edges in a graph, storing information about connections between two stations, including the indices of the connected stations ('u' and 'v') and the distance between them ('distance')

- ‘comp’ function act as comparison function for sorting, comparing the two 'Edge' objects based on their 'distance' member

KRUSKAL ALGORITHM
- 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  }
```

```
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 combines the 'find' and 'union' operations for disjoint-set data structure.
- Given elements 'u' and 'v', it uses the 'find' function to determine their respective set roots and then performs the 'union' operation based on the ranks of the sets

KRUSKAL ALGORITHM
- 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 implements Kruskal's algorithm to find the Minimum Spanning Tree (MST) in a graph
- It initializes a disjoint set structure and sorts the edges by distance.
- It iterates through the sorted edges, adding non-cycle edges to the 'minimumSpanningTree' vector and performing union operations

KRUSKAL ALGORITHM
- Draw MST 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 visualize the Minimum Spanning Tree (MST) and its corresponding graph
- It takes a vector of 'Station' objects ('stations') representing vertices and a vector of 'Edge' objects ('minimumSpanningTree') representing the MST

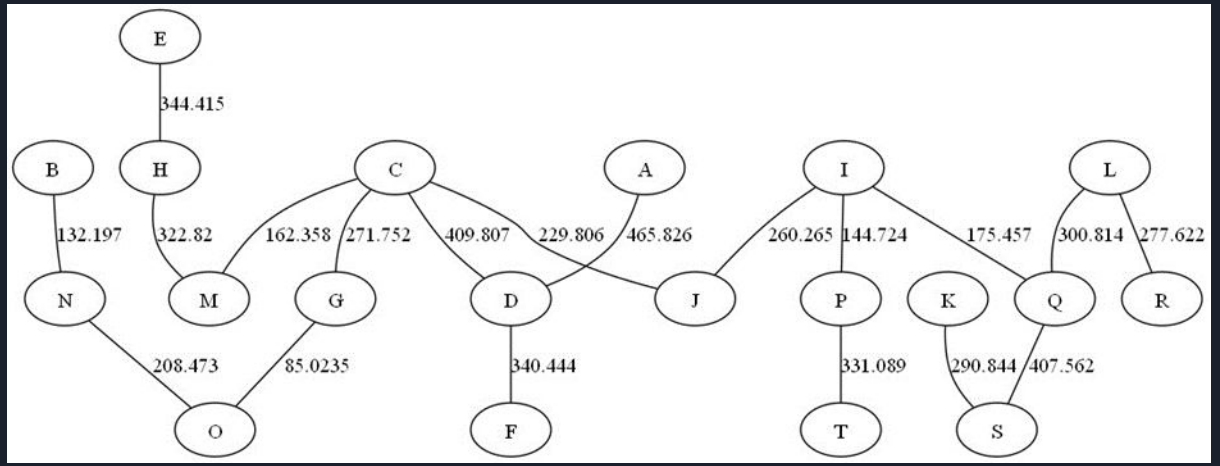
KRUSKAL ALGORITHM
- Main Function

```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 ;

    vector<Edge> edges;  // Convert routes to edges
```

```cpp
    for (const auto& route : routes)
    {
        double distance = calculateDistance(stations[route.first], stations[route.second]);
        edges.emplace_back(route.first, route.second, distance);
    }

    vector<Edge> minimumSpanningTree = KruskalsAlgorithm(edges, stations.size());

    cout << "\nMinimum Spanning Tree Edges : " << endl ;

    for(const auto&edge : minimumSpanningTree)
    {
        cout << "Edge between " << stations[edge.u].name
             << " (" << edge.u << ")"
             << " and " << stations[edge.v].name
             << " (" << edge.v << ")"
             << " Distance: " << edge.distance
             << endl;
    }

    cout << endl ;

    // Generate Graphviz DOT file
    graphMST(stations, minimumSpanningTree);

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

    return 0;
}
```

KRUSKAL ALGORITHM

- Graph MST



KRUSKAL ALGORITHM

- Time and Space Complexity

- Time complexity for Kruskal Algorithm will be **O(E log E)** where E is the number of edges (routes) in the graph.

- Space complexity for Kruskal Algorithm is **O(E + V)** where V is the number of vertices and E is the number of edges (routes) in the graph.

QUESTION 4:
- Dynamic Programming

0/1 Knapsack Algorithm

```
7    struct Station {
8        string name;
9        int weight;
10       int profit;
11   };
```

Structure called Station

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  }
```

```cpp
vector<Station> solveKnapsack(const vector<Station>& stations, int maxCapacity) {
    int numStations = stations.size();
    // Create a 2D table to store the maximum profit for each station and capacity combination.
    vector<vector<int>> dp(numStations + 1, vector<int>(maxCapacity + 1, 0));

    // Fill the dp table using dynamic programming.
    for (int i = 1; i <= numStations; ++i) {
        for (int w = 1; w <= maxCapacity; ++w) {
            if (stations[i - 1].weight <= w) {
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - stations[i - 1].weight] + stations[i - 1].profit);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    // Trace back to find the selected stations.
    int i = numStations;
    int w = maxCapacity;
    vector<Station> selectedStations;

    while (i > 0 && w > 0) {
        if (dp[i][w] != dp[i - 1][w]) {
            selectedStations.push_back(stations[i - 1]);
            w -= stations[i - 1].weight;
        }
        --i;
    }

    reverse(selectedStations.begin(), selectedStations.end());
    return selectedStations;
}
```

Function to Solve Knapsack

```cpp
int main() {
    vector<Station> stations = loadDataset();
    int maxCapacity = 800;

    vector<Station> selectedStations = solveKnapsack(stations, maxCapacity);

    // Output the selected stations and their properties.
    cout << "Selected Stations:" << endl;
    for (const Station& station : selectedStations) {
        cout << "Station Name: " << station.name << endl;
        cout << "Weight: " << station.weight << " Profit: " << station.profit << endl;
    }

    return 0;
}
```

Main Function