QUESTION 1;

```
#include <iostream>
#include <chrono>
#include <cstdlib> // For rand() and srand()
#include <ctime>   // For time()
// Number of vertices in the graph
#define V 6
// Placeholder for Dijkstra's algorithm
void dijkstraAlgorithm(const std::vector<std::vector<int>>& graph, int source, int destination) {
   std::vector<int> dist(V, INT_MAX);
   std::vector<bool> sptSet(V, false);
   std::vector<int> parent(V, -1);
   dist[source] = 0; // Distance from source vertex to itself is always 0
   for (int count = 0; count < V - 1; ++count) {
      int u = minDistance(dist, sptSet);
      sptSet[u] = true;
      for (int v = 0; v < V; ++v) {
         if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
            dist[v] = dist[u] + graph[u][v];
            parent[v] = u;
         }
      }
   }
   // Print the shortest path from source to destination
   std::cout << "Shortest Path from " << source << " to " << destination << ": ";
   printShortestPath(parent, destination);
   std::cout << "Shortest Path Length: " << dist[destination] << '\n'; // Print the shortest path length
```

```
}
// Function to print the shortest path from source to destination
void printShortestPath(const std::vector<int>& parent, int destination) {
    if (parent[destination] == -1) {
        std::cout << destination << ' ';
        return;
    }
    printShortestPath(parent, parent[destination]);
    std::cout << destination << ' ';
}
// Function to find the vertex with the minimum distance value
int minDistance(const std::vector<int>& dist, const std::vector<bool>& sptSet) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; ++v) {
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}
int main() {
    // Set random seed
    std::srand(std::time(0));
    // Example adjacency matrix representing the graph
    std::vector<std::vector<int>> graph = {
        {0, 1, 4, 0, 0, 0},
```

```
        {1, 0, 4, 2, 7, 0},
        {4, 4, 0, 3, 5, 0},
        {0, 2, 3, 0, 4, 6},
        {0, 7, 5, 4, 0, 7},
        {0, 0, 0, 6, 7, 0}
    };
    // Loop for 200 random source-destination pairs
    for (int i = 0; i < 200; ++i) {
        int source = std::rand() % 511;  // Assuming 512 nodes in Florida graph (0-511)
        int destination = std::rand() % 511;
        // Measure the start time
        auto start_time = std::chrono::high_resolution_clock::now();
        // Call Dijkstra's algorithm for each pair
        dijkstraAlgorithm(graph, source, destination);
        // Measure the end time
        auto end_time = std::chrono::high_resolution_clock::now();
        // Calculate and print the runtime in seconds
        auto duration = std::chrono::duration_cast<std::chrono::seconds>(end_time - start_time);
        std::cout << "Total runtime in seconds for Dijkstra's algorithm: " << duration.count() << "s\n";
        // Add a newline for better output separation
        std::cout << std::endl;
    }
    return 0;
}
```

(QUESTION 2)

```
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <unordered_map>
#include <algorithm> // For std::find
struct Route {
    std::string id;
    std::vector<std::string> stops;
};
std::unordered_map<std::string, Route> routes;  // Assuming this is a global variable
// Function to read GTFS data and populate data structures
void readGTFSData() {
    std::ifstream file("your_gtfs_data_file.txt");  // Change to your actual file name
    // Check if the file is open
    if (!file.is_open()) {
        std::cerr << "Error opening file." << std::endl;
        exit(1);
    }
    std::string line;
    while (std::getline(file, line)) {
        std::istringstream iss(line);
        std::string token;
        iss >> token;
        if (token == "route") {
            // Parse route information
```

```
            Route route;
            iss >> route.id;
            while (iss >> token) {
                route.stops.push_back(token);
            }
            routes[route.id] = route;
        }
    }
}
// Function to find direct journeys
void findDirectJourneys(const std::string& source_stop_id, const std::string& destination_stop_id) {
    for (const auto& entry : routes) {
        const Route& route = entry.second;
        if (std::find(route.stops.begin(), route.stops.end(), source_stop_id) != route.stops.end() &&
            std::find(route.stops.begin(), route.stops.end(), destination_stop_id) != route.stops.end()) {
            std::cout << "Direct journey: " << route.id << "(" << source_stop_id << " > " <<
destination_stop_id << ")\n";
        }
    }
}
// Function to find journeys with one transfer
void findJourneysWithOneTransfer(const std::string& source_stop_id, const std::string&
destination_stop_id) {
    for (const auto& entry1 : routes) {
        const Route& route1 = entry1.second;
        if (std::find(route1.stops.begin(), route1.stops.end(), source_stop_id) != route1.stops.end()) {
            for (const auto& entry2 : routes) {
                const Route& route2 = entry2.second;
```

```
            if (route1.stops.back() == route2.stops.front()) {

                std::cout << "Journey with one transfer: " << route1.id << "(" << source_stop_id << " > "
<< route1.stops.back() << ") - " << route2.id << "(" << route2.stops.back() << " > " <<
destination_stop_id << ")\n";

            }

        }

    }

}

// Function to find journeys with two transfers

void findJourneysWithTwoTransfers(const std::string& source_stop_id, const std::string&
destination_stop_id) {

    for (const auto& entry1 : routes) {

        const Route& route1 = entry1.second;

        if (std::find(route1.stops.begin(), route1.stops.end(), source_stop_id) != route1.stops.end()) {

            for (const auto& entry2 : routes) {

                const Route& route2 = entry2.second;

                if (route1.stops.back() == route2.stops.front()) {

                    for (const auto& entry3 : routes) {

                        const Route& route3 = entry3.second;



                        if (std::find(route3.stops.begin(), route3.stops.end(), destination_stop_id) !=
route3.stops.end() &&

                            route2.stops.back() == route3.stops.front()) {

                            std::cout << "Journey with two transfers: " << route1.id << "(" << source_stop_id << "
> " << route1.stops.back() << ") - " << route2.id << "(" << route2.stops.back() << " > " <<
route3.stops.front() << ") - " << route3.id << "(" << route3.stops.back() << " > " <<
destination_stop_id << ")\n";

                        }

                    }
```

```
            }
        }
    }
}
int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <source_stop_id> <destination_stop_id>" << std::endl;
        return 1;
    }
    // Read GTFS data and populate data structures
    readGTFSData();
    std::string source_stop_id = argv[1];
    std::string destination_stop_id = argv
```