Question 1:

You are designing a navigation system for a delivery company that operates in a city with a complex road network. The company needs to optimize its delivery routes to minimize travel time and fuel costs. The city is represented as a graph, where intersections are nodes and roads are edges with associated travel times. The delivery trucks start from a central warehouse and need to visit multiple customer locations throughout the day. Using a Dijkstra's algorithm, you should develop a route optimization system for the delivery trucks

Input Format:

- An integer representing the number of vertices in the graph.
- An integer representing the number of edges in the graph.
- For each edge, provide three integers separated by spaces: source node (u), destination node (v), and the weight of the edge.
- An integer representing the starting node for Dijkstra's algorithm.

Output Format:

- For each node in the graph, print the shortest distance using Dijkstra's algorithm from the starting node.

Title for Question 1: Dijkstra’s Algorithm - Workouts

Solution:

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

const int INF = INT_MAX;

class Graph {
public:
    int V;
    vector<vector<pair<int, int>>> adj;

    Graph(int vertices) : V(vertices), adj(vertices) {}

    void addEdge(int u, int v, int weight) {
        adj[u].emplace_back(v, weight);
        adj[v].emplace_back(u, weight); // For undirected graph
    }

    void dijkstra(int start) {
        vector<int> dist(V, INF);
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pa
```

```
        dist[start] = 0;
        pq.push({0, start});

        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();

            for (const auto& neighbor : adj[u]) {
                int v = neighbor.first;
                int weight = neighbor.second;

                if (dist[v] > dist[u] + weight) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
        }

        cout << "Shortest distances from node " << start << ":\n";
        for (int i = 0; i < V; ++i) {
            cout << "To node " << i << ": " << dist[i] << "\n";
        }
    }
};

int main() {
    int numVertices;
    //cout << "Enter the number of vertices: ";
    cin >> numVertices;

    Graph g(numVertices);

    int numEdges;
   // cout << "Enter the number of edges: ";
    cin >> numEdges;

    //cout << "Enter edges (u v weight):" << endl;
    for (int i = 0; i < numEdges; ++i) {
        int u, v, weight;
        cin >> u >> v >> weight;
        g.addEdge(u, v, weight);
    }

    int startNode;
    //cout << "Enter the starting node: ";
    cin >> startNode;

    g.dijkstra(startNode);

    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	5 7 0 1 2 0 2 4 1 2 1 1 3 7 2 3 3 2 4 5 3 4 1 0	Shortest distances from node 0: To node 0: 0 To node 1: 2 To node 2: 3 To node 3: 6 To node 4: 7

S.No	Inputs	Outputs
2	4 5 0 1 3 0 2 5 1 2 2 1 3 8 2 3 1 0	Shortest distances from node 0: To node 0: 0 To node 1: 3 To node 2: 5 To node 3: 6
3	6 9 0 1 4 0 2 3 1 2 2 1 3 5 2 3 1 2 4 4 3 4 3 3 5 7 4 5 2 0	Shortest distances from node 0: To node 0: 0 To node 1: 4 To node 2: 3 To node 3: 4 To node 4: 7 To node 5: 9
4	3 3 0 1 2 1 2 1 2 0 4 0	Shortest distances from node 0: To node 0: 0 To node 1: 2 To node 2: 3
5	7 10 0 1 3 0 2 5 1 2 2 1 3 1 2 3 4 2 4 6 3 4 2 3 5 7 4 5 3 5 6 4 0	Shortest distances from node 0: To node 0: 0 To node 1: 3 To node 2: 5 To node 3: 4 To node 4: 6 To node 5: 9 To node 6: 13
6	4 6 0 1 2 0 2 3 1 2 1 1 3 5 2 3 2 0 3 8 0	Shortest distances from node 0: To node 0: 0 To node 1: 2 To node 2: 3 To node 3: 5

White List:

Black List:

Question 2:

Imagine you are developing a navigation system for an autonomous drone delivery service in a densely populated urban area. The drone needs to find the shortest path between two given locations while considering both one-way and two-way streets. Implement a bidirectional version of Dijkstra's Algorithm to efficiently calculate the shortest path, taking into account the dynamic nature of traffic and road conditions.

Input Format:

- The first line contains two integers: numNodes and numEdges, representing the number of nodes and edges in the graph, respectively.
- The next numEdges lines each contain three integers: from, to, and weight, representing an undirected edge between nodes from and to with the given weight.
- The last line contains two integers: source and target, representing the source and target nodes for which the shortest path needs to be found.
-

Output Format:

- If a valid path exists between the source and target nodes:
- Output a line in the format: "Shortest Path from source to target: shortestPath", where shortestPath is the length of the shortest path.
- If no valid path exists between the source and target nodes:
- Output a line: "No path found from source to target".

Title for Question 2: Bidirectional Dijkstra's Algorithm

Solution:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
using namespace std;

const int INF = numeric_limits<int>::max();

struct Edge {
    int to, weight;
    Edge(int t, int w) : to(t), weight(w) {}
};

class Graph {
public:
    int numNodes;
    vector<vector<Edge>> adjList;

    Graph(int n) : numNodes(n), adjList(n) {}

    void addEdge(int from, int to, int weight) {
        adjList[from].emplace_back(to, weight);
        adjList[to].emplace_back(from, weight);
    }
};

vector<int> dijkstra(const Graph& graph, int start) {
    vector<int> dist(graph.numNodes, INF);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<i

    dist[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        int u = pq.top().second;
        int currentDist = pq.top().first;
        pq.pop();

        if (currentDist > dist[u]) {
            continue;
        }

        for (const Edge& edge : graph.adjList[u]) {
            int v = edge.to;
            int weight = edge.weight;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}

int bidirectionalDijkstra(const Graph& graph, int source, int target) {
    vector<int> distForward = dijkstra(graph, source);
    vector<int> distBackward = dijkstra(graph, target);
```

```
    int shortestPath = INF;
    for (int i = 0; i < graph.numNodes; ++i) {
        if (distForward[i] != INF && distBackward[i] != INF) {
            shortestPath = min(shortestPath, distForward[i] + distBackwar
        }
    }

    return shortestPath;
}

int main() {
    int numNodes, numEdges;
    cin >> numNodes >> numEdges;
    Graph graph(numNodes);
    for (int i = 0; i < numEdges; ++i) {
        int from, to, weight;
        cin >> from >> to >> weight;
        graph.addEdge(from, to, weight);
    }
    int source, target;
    cin >> source >> target;
    int shortestPath = bidirectionalDijkstra(graph, source, target);
    if (shortestPath != INF) {
        cout << "Shortest Path from " << source << " to " << target << ":
    } else {
        cout << "No path found from " << source << " to " << target << en
    }
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	5 7 0 1 1 1 2 2 2 3 3 3 4 4 4 0 5 0 3 2 1 4 3 0 2	Shortest Path from 0 to 2: 3
2	5 6 0 1 1 0 2 4 1 2 2 1 3 5 2 3 1 3 4 3 0 4	Shortest Path from 0 to 4: 7
3	4 4 0 1 1 1 2 2 2 3 3 3 0 4 0 3	Shortest Path from 0 to 3: 4
4	4 5 0 1 1 0 2 2 1 2 3 2 3 1 1 3 4 0 3	Shortest Path from 0 to 3: 3
5	3 2 0 1 2 1 2 3 0 2	Shortest Path from 0 to 2: 5
6	4 4 0 1 1 0 2 2 2 3 3 1 3 1 0 3	Shortest Path from 0 to 3: 2

White List:

Black List:

Question 3:

Imagine you are designing a real-time navigation system for a smart city, where roads and intersections can be dynamically added or removed due to construction, accidents, or changes in traffic patterns. How would you implement Dijkstra's Algorithm to continuously update and optimize the route for vehicles in such a dynamic graph? Consider scenarios where new roads are constructed, existing roads are closed for maintenance, or intersections are temporarily blocked. Discuss the data structures and algorithms you would use to efficiently handle these dynamic changes while ensuring the navigation system provides accurate and up-to-date routes for

vehicles.

Input Format:

- Number of Vertices (vertices): An integer representing the number of vertices in the graph.
- Number of Edges (edges): An integer representing the initial number of edges in the graph.
- Edges (u v weight): For each edge, three integers separated by space: source vertex u, destination vertex v, and edge weight.
- Source Vertex (source): An integer representing the source vertex for Dijkstra's Algorithm.
- Edge Removal (u v): Two integers representing the vertices of the edge to be removed during runtime.
- Edge Addition (u v weight): Three integers representing the source vertex u, destination vertex v, and edge weight for the edge to be added during runtime.

Output Format:

- Shortest Distances Before Graph Changes:
- For each vertex, print "To vertex i: distance" where i is the vertex number, and distance is the shortest distance from the source vertex.
- Graph Changes:
- The program removes an edge specified by the user (u v).
- The program adds a new edge specified by the user (u v weight).
- Shortest Distances After Graph Changes:
- For each vertex, print "To vertex i: distance" where i is the vertex number, and distance is the shortest distance from the source vertex after the graph changes.

Title for Question 3: Dynamic Graphs

Solution:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <algorithm>
using namespace std;

const int INF = numeric_limits<int>::max();

class Graph {
private:
    int V; // Number of vertices
    vector<vector<pair<int, int>>> adjList; // Adjacency list representat

public:
    Graph(int vertices) : V(vertices), adjList(vertices) {}

    // Add an edge to the graph
    void addEdge(int u, int v, int weight) {
```

```
        adjList[u].emplace_back(v, weight);
        adjList[v].emplace_back(u, weight);
    }

    // Remove an edge from the graph
    void removeEdge(int u, int v) {
        adjList[u].erase(remove_if(adjList[u].begin(), adjList[u].end(),
            [v](const pair<int, int>& edge) { return edge.first == v; }),

        adjList[v].erase(remove_if(adjList[v].begin(), adjList[v].end(),
            [u](const pair<int, int>& edge) { return edge.first == u; }),
    }

    // Dijkstra's Algorithm
    vector<int> dijkstra(int source) {
        vector<int> dist(V, INF);
        dist[source] = 0;

        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pa
        pq.push({0, source});

        while (!pq.empty()) {
            int u = pq.top().second;
            int uDist = pq.top().first;
            pq.pop();

            if (uDist > dist[u]) continue;

            for (const auto& neighbor : adjList[u]) {
                int v = neighbor.first;
                int weight = neighbor.second;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
        }

        return dist;
    }
};

int main() {
    int vertices, edges;
    cin >> vertices;
    Graph graph(vertices);
    cin >> edges;
    for (int i = 0; i < edges; ++i) {
        int u, v, weight;
        cin >> u >> v >> weight;
        graph.addEdge(u, v, weight);
    }
    int source;
    cin >> source;
    vector<int> distances = graph.dijkstra(source);
    cout << "Shortest distances from vertex " << source << ":" << endl;
    for (int i = 0; i < vertices; ++i) {
        cout << "To vertex " << i << ": " << distances[i] << endl;
    }
```

```
    int u, v;
    cin >> u >> v;
    graph.removeEdge(u, v);
    cin >> u >> v >> edges;
    graph.addEdge(u, v, edges);
    distances = graph.dijkstra(source);
    cout << "Shortest distances from vertex " << source << " after graph
    for (int i = 0; i < vertices; ++i) {
        cout << "To vertex " << i << ": " << distances[i] << endl;
    }
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	5 8 0 1 2 0 2 4 1 2 1 1 3 7 2 3 3 2 4 5 3 4 6 0 2 3 4 2 2	Shortest distances from vertex 4: To vertex 0: 8 To vertex 1: 6 To vertex 2: 5 To vertex 3: 6 To vertex 4: 0 Shortest distances from vertex 4 after graph changes: To vertex 0: 8 To vertex 1: 6 To vertex 2: 5 To vertex 3: 6 To vertex 4: 0
2	4 5 0 1 1 1 2 2 2 3 3 3 0 4 0 2 5 0 1 3 3 2 1	Shortest distances from vertex 0: To vertex 0: 0 To vertex 1: 1 To vertex 2: 3 To vertex 3: 4 Shortest distances from vertex 0 after graph changes: To vertex 0: 0 To vertex 1: 1 To vertex 2: 3 To vertex 3: 4
3	3 3 0 1 5 1 2 2 2 0 3 1 0 2 2 1 1	Shortest distances from vertex 1: To vertex 0: 5 To vertex 1: 0 To vertex 2: 2 Shortest distances from vertex 1 after graph changes: To vertex 0: 5 To vertex 1: 0 To vertex 2: 1
4	6 7 0 1 2 1 2 3 2 3 1 3 4 5 4 5 4 5 0 3 0 4 2 5 2 4 3 1 6	Shortest distances from vertex 5: To vertex 0: 3 To vertex 1: 5 To vertex 2: 8 To vertex 3: 9 To vertex 4: 4 To vertex 5: 0 Shortest distances from vertex 5 after graph changes: To vertex 0: 3 To vertex 1: 5 To vertex 2: 8 To vertex 3: 9 To vertex 4: 4 To vertex 5: 0
5	6 7 0 1 2 1 2 3 2 3 1 3 4 5 4 5 4 5 0 3 0 4 2 5 2 4 3 1 6	Shortest distances from vertex 5: To vertex 0: 3 To vertex 1: 5 To vertex 2: 8 To vertex 3: 9 To vertex 4: 4 To vertex 5: 0 Shortest distances from vertex 5 after graph changes: To vertex 0: 3 To vertex 1: 5 To vertex 2: 8 To vertex 3: 9 To vertex 4: 4 To vertex 5: 0
6	3 3 0 1 5 1 2 2 2 0 3 1 0 2 2 1 1	Shortest distances from vertex 1: To vertex 0: 5 To vertex 1: 0 To vertex 2: 2 Shortest distances from vertex 1 after graph changes: To vertex 0: 5 To vertex 1: 0 To vertex 2: 1

White List:

Black List:

Question 4:

Imagine you are working on a navigation system for an autonomous vehicle, and you've implemented Dijkstra's Algorithm to find the shortest path between two points on a map. Now, the stakeholders want to visualize the actual path on the map, not just the distance. You should modify your existing implementation to include path reconstruction, ensuring that the autonomous vehicle can follow the computed route accurately.

Input Format:

- The first line contains the number of vertices (vertices) in the graph.
- The second line contains the number of edges (edges) in the graph.
- The next edges lines contain three space-separated integers each, representing an edge in the graph. The three integers are from, to, and weight, where from is the starting vertex of the edge, to is the ending vertex, and weight is the weight of the edge.
- The last two lines contain two integers, start and end, representing the starting and ending vertices for which you want to find the shortest path.

Output Format:

- If there is no path from the starting vertex to the ending vertex, print: "There is no path from <start> to <end>."
- If there is a path, print:
- "Shortest Distance from <start> to <end>: <distance>"
- "Shortest Path: <vertex1> <vertex2> ... <end>"

Title for Question 4: Path Reconstruction

Solution:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <algorithm>
using namespace std;

const int INF = numeric_limits<int>::max();

struct Edge {
    int to, weight;
};

typedef pair<int, int> pii; // pair<distance, vertex>

class Dijkstra {
private:
    int n; // number of vertices
    vector<vector<Edge>> graph;

public:
    Dijkstra(int vertices) : n(vertices), graph(vertices) {}

    void addEdge(int from, int to, int weight) {
        graph[from].push_back({to, weight});
    }

    pair<vector<int>, vector<int>> dijkstra(int start, int end) {
        vector<int> distance(n, INF);
```

```
        vector<int> parent(n, -1);

        priority_queue<pii, vector<pii>, greater<pii>> pq;
        pq.push({0, start});
        distance[start] = 0;

        while (!pq.empty()) {
            int u = pq.top().second;
            int dist = pq.top().first;
            pq.pop();

            if (dist > distance[u]) continue;

            for (const Edge& edge : graph[u]) {
                int v = edge.to;
                int w = edge.weight;

                if (distance[u] + w < distance[v]) {
                    distance[v] = distance[u] + w;
                    parent[v] = u;
                    pq.push({distance[v], v});
                }
            }
        }

        return {distance, parent};
    }

    vector<int> reconstructPath(int start, int end, const vector<int>& pa
        vector<int> path;
        for (int v = end; v != -1; v = parent[v]) {
            path.push_back(v);
        }
        reverse(path.begin(), path.end());
        return path;
    }
};

int main() {
    int vertices, edges;
    cin >> vertices;
    cin >> edges;
    Dijkstra dijkstra(vertices);
    for (int i = 0; i < edges; ++i) {
        int from, to, weight;
        cin >> from >> to >> weight;
        dijkstra.addEdge(from, to, weight);
    }
    int start, end;
    cin >> start;
    cin >> end;
    auto result = dijkstra.dijkstra(start, end);
    vector<int> distance = result.first;
    vector<int> parent = result.second;
    if (distance[end] == INF) {
        cout << "There is no path from " << start << " to " << end << "."
    } else {
        cout << "Shortest Distance from " << start << " to " << end << ":
        cout << "Shortest Path: ";
        vector<int> path = dijkstra.reconstructPath(start, end, parent);
```

```
        for (int vertex : path) {
            cout << vertex << " ";
        }
        cout << endl;
    }
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	6 9 0 1 2 0 2 4 1 2 1 1 3 7 2 4 3 3 4 1 1 4 5 3 5 6 4 5 2 0 5	Shortest Distance from 0 to 5: 8 Shortest Path: 0 1 2 4 5
2	4 5 0 1 2 0 2 4 1 2 1 1 3 7 2 3 2 0 3	Shortest Distance from 0 to 3: 5 Shortest Path: 0 1 2 3
3	4 5 0 1 2 0 2 4 1 2 1 1 3 7 2 3 2 2 0	There is no path from 2 to 0.
4	3 3 0 1 2 1 2 3 2 0 1 0 2	Shortest Distance from 0 to 2: 5 Shortest Path: 0 1 2
5	3 3 0 1 2 1 2 3 2 0 1 1 0	Shortest Distance from 1 to 0: 4 Shortest Path: 1 2 0
6	3 2 0 1 2 1 2 1 0 2	Shortest Distance from 0 to 2: 3 Shortest Path: 0 1 2

White List:

Black List:

Question 5:

You are working as a software engineer for a transportation management company that oversees a network of cities connected by roads. The company wants to optimize its logistics by finding the shortest paths between all pairs of cities. The cities and roads are represented as a weighted graph, where the weight of an edge represents the distance between two cities. Your task is to implement the Floyd-Warshall algorithm to find the shortest paths and distances between all pairs of cities.

Input Format:

- The user is prompted to enter the number of vertices.
- The user is then prompted to enter the adjacency matrix representing the graph.
- For each pair of vertices (i, j), the program expects the user to input the distance from vertex i to vertex j.
- If there is no direct edge between vertices i and j, the user should input -1 to represent infinity (no direct edge).

Output Format:

- The program outputs the shortest distances between all pairs of vertices after applying the Floyd-Warshall algorithm.
- The output is presented in the form of a matrix, where each entry represents the shortest distance between the corresponding pair of vertices.
- If the distance is infinity or Not Connected (-1), it is displayed as "INF."
- Otherwise, the actual shortest distance is displayed.

Title for Question 5: Floyd-Warshall Algorithm

Solution:

```
#include <iostream>
#include <vector>
#include <limits.h>
const int INF = INT_MAX; // Represents infinity

void floydWarshall(std::vector<std::vector<int>>& graph, int numVertices)
    for (int k = 0; k < numVertices; ++k) {
        for (int i = 0; i < numVertices; ++i) {
            for (int j = 0; j < numVertices; ++j) {
                // Check if vertex k is an intermediate vertex that impro
                if (graph[i][k] != -1 && graph[k][j] != -1 && graph[i][k]
                    graph[i][j] = graph[i][k] + graph[k][j];
                }
            }
        }
    }
}

int main() {
    int numVertices;

    // Get the number of vertices from the user
    //std::cout << "Enter the number of vertices: ";
    std::cin >> numVertices;

    // Initialize the adjacency matrix with INF representing no direct ed
    std::vector<std::vector<int>> graph(numVertices, std::vector<int>(num

    // Get the adjacency matrix from the user
    //std::cout << "Enter the adjacency matrix (INF for no direct edge):\
    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {
           // std::cout << "Enter distance from vertex " << i + 1 << " to
            std::cin >> graph[i][j];
        }
    }

    // Apply Floyd-Warshall algorithm
    floydWarshall(graph, numVertices);

    // Print the shortest paths and distances
    std::cout << "\nShortest distances between all pairs of vertices:\n";
    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {
            if (graph[i][j] == -1) {
```

```
                std::cout << "INF\t";
            } else {
                std::cout << graph[i][j] << "\t";
            }
        }
        std::cout << "\n";
    }

    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	2 0 3 -1 0	Shortest distances between all pairs of vertices: 0 3 INF 0
2	4 0 2 -1 1 -1 0 3 -1 4 -1 0 2 -1 -1 -1 0	Shortest distances between all pairs of vertices: 0 2 INF 1 INF 0 3 INF 4 INF 0 2 INF INF INF 0
3	5 0 3 -1 7 -1 -1 0 1 4 -1 2 -1 0 -1 -1 -1 -1 -1 0 1 6 -1 -1 -1 0	Shortest distances between all pairs of vertices: 0 3 INF 7 INF INF 0 1 4 INF 2 INF 0 INF INF INF INF INF 0 1 6 INF INF INF 0
4	3 0 -1 5 2 0 -1 -1 1 0	Shortest distances between all pairs of vertices: 0 INF 5 2 0 INF INF 1 0
5	4 0 1 -1 5 -1 0 2 -1 -1 -1 0 1 -1 -1 -1 0	Shortest distances between all pairs of vertices: 0 1 INF 5 INF 0 2 INF INF INF 0 1 INF INF INF 0
6	3 0 2 -1 -1 0 1 3 -1 0	Shortest distances between all pairs of vertices: 0 2 INF INF 0 1 3 INF 0

White List:

Black List:

Question 6:

Imagine you are a software engineer working on a project that involves optimizing the calculation of shortest paths in a graph. The current implementation utilizes the Floyd-Warshall algorithm for this purpose. However, you've identified a potential optimization by introducing a bidirectional version of the algorithm. The project involves a weighted, directed graph representing a transportation network. Nodes represent locations, and edges represent roads with travel times as weights. Your goal is to explore and implement a bidirectional version of the Floyd-Warshall algorithm to improve the efficiency of the shortest path calculations.

Input Format:

- The user is prompted to enter an integer numVertices, representing the number of vertices in the graph.
- The user is then expected to input the adjacency matrix of the graph. The matrix should be of size numVertices x numVertices.
- Each element in the matrix represents the weight of the edge between the corresponding vertices.

- The user should enter either the weight of the edge or -1 if there is no direct edge between the vertices.

Output Format:

- The program calculates and displays the shortest paths between all pairs of vertices using the bidirectional Floyd-Warshall algorithm.
- The output is presented in the form of a matrix, where each element dist[i][j] represents the shortest path from vertex i to vertex j.
- If there is no direct path between vertices i and j, the program prints "INF" for that entry.
- The output is printed in a matrix format, where each row corresponds to a source vertex, and each column corresponds to a destination vertex.
- The result is displayed with a label: "Shortest paths between all pairs of vertices:"

Title for Question 6: Bidirectional Search

Solution:

```
#include <iostream>
#include <vector>
#include <climits>

void floydWarshallBidirectional(std::vector<std::vector<int>>& graph, int
    std::vector<std::vector<int>> dist(numVertices, std::vector<int>(numV

    // Initializing distance matrix with given graph
    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {
            dist[i][j] = graph[i][j];
        }
    }

    // Bidirectional Floyd-Warshall algorithm
    for (int k = 0; k < numVertices; ++k) {
        for (int i = 0; i < numVertices; ++i) {
            for (int j = 0; j < numVertices; ++j) {
                // Check if the path through vertex k is shorter
                if (dist[i][k] != -1 && dist[k][j] != -1 && dist[i][k] +
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Display the shortest paths
    std::cout << "Shortest paths between all pairs of vertices:\n";
    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {
            if (dist[i][j] == -1) {
                std::cout << "INF ";
            } else {
                std::cout << dist[i][j] << " ";
            }
```

```
        }
        std::cout << "\n";
    }
}

int main() {
    int numVertices;
    std::cin >> numVertices;
    std::vector<std::vector<int>> graph(numVertices, std::vector<int>(num
    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {

            std::cin >> graph[i][j];
        }
    }
    floydWarshallBidirectional(graph, numVertices);
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	3 0 2 -1 -1 0 3 -1 -1 0	Shortest paths between all pairs of vertices: 0 2 INF INF 0 3 INF INF 0
2	4 0 3 -1 -1 2 0 -1 1 -1 -1 0 5 -1 -1 -1 0	Shortest paths between all pairs of vertices: 0 3 INF INF 2 0 INF 1 INF INF 0 5 INF INF INF 0
3	5 0 1 -1 -1 4 -1 0 5 1 -1 -1 -1 0 -1 2 -1 -1 3 0 -1 -1 -1 -1 0	Shortest paths between all pairs of vertices: 0 1 INF INF 4 INF 0 4 1 INF INF INF 0 INF 2 INF INF 3 0 INF INF INF INF 0 0
4	3 0 2 1 -1 0 -1 3 -1 0	Shortest paths between all pairs of vertices: 0 2 1 INF 0 INF 3 INF 0
5	4 0 -1 -1 -1 2 0 3 -1 -1 -1 0 -1 -1 -1 -1 0	Shortest paths between all pairs of vertices: 0 INF INF INF 2 0 3 INF INF INF 0 INF INF INF INF 0
6	2 0 5 -1 0	Shortest paths between all pairs of vertices: 0 5 INF 0

White List:

Black List:
