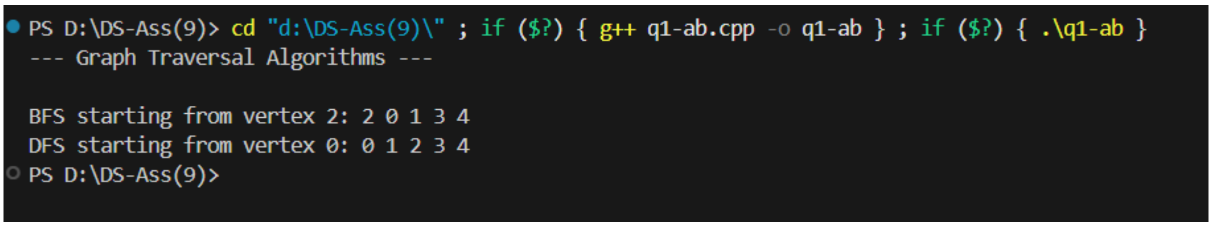
```
Question-1(a,b)
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>

using namespace std;

// Class to represent the graph using an adjacency list
class Graph {
private:
    int V; // Number of vertices
    vector<vector<int> > adj;

public:
    // Constructor (C++98/03 compatible initialization)
    Graph(int v) {
        V = v;
        adj.resize(V);
    }

    // Function to add an edge to the graph (undirected)
    void addEdge(int v, int w) {
        adj[v].push_back(w);
        adj[w].push_back(v);
    }

    // 1. Breadth First Search (BFS)
    void BFS(int startNode) {
        // Mark all vertices as not visited
        vector<bool> visited(V, false);

        // Create a queue for BFS
        queue<int> q;

        // Mark the current node as visited and enqueue it
        visited[startNode] = true;
        q.push(startNode);

        while (!q.empty()) {
            // Dequeue a vertex from queue and print it
            int u = q.front();
            cout << u << " ";
            q.pop();

            // Get all adjacent vertices of the dequeued vertex u.
            // If an adjacent vertex has not been visited, mark it
visited and enqueue it.
            for (size_t i = 0; i < adj[u].size(); ++i) {
                int v = adj[u][i];
                if (!visited[v]) {
                    visited[v] = true;
                    q.push(v);
                }
            }
        }
        cout << "\n";
    }

    // Helper for DFS (Recursive)
    void DFSUtil(int u, vector<bool>& visited) {
        // Mark the current node as visited and print it
        visited[u] = true;
        cout << u << " ";

        // Recur for all the vertices adjacent to this vertex
        for (size_t i = 0; i < adj[u].size(); ++i) {
            int v = adj[u][i];
            if (!visited[v])
                DFSUtil(v, visited);
        }
    }

    // 2. Depth First Search (DFS)
    void DFS(int startNode) {
        // Mark all vertices as not visited
        vector<bool> visited(V, false);

        // Call the recursive helper function
        DFSUtil(startNode, visited);
        cout << "\n";
    }
};
```

```cpp
int main() {
    // Create a graph with 5 vertices (0 to 4)
    Graph g(5);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 3);
    g.addEdge(3, 4);

    cout << "--- Graph Traversal Algorithms ---\n\n";

    cout << "BFS starting from vertex 2: ";
    g.BFS(2);

    cout << "DFS starting from vertex 0: ";
    g.DFS(0);

    return 0;
}
```

Output->



```
PS D:\DS-Ass(9)> cd "d:\DS-Ass(9)\" ; if ($?) { g++ q1-ab.cpp -o q1-ab } ; if ($?) { .\q1-ab }
--- Graph Traversal Algorithms ---

BFS starting from vertex 2: 2 0 1 3 4
DFS starting from vertex 0: 0 1 2 3 4
PS D:\DS-Ass(9)>
```

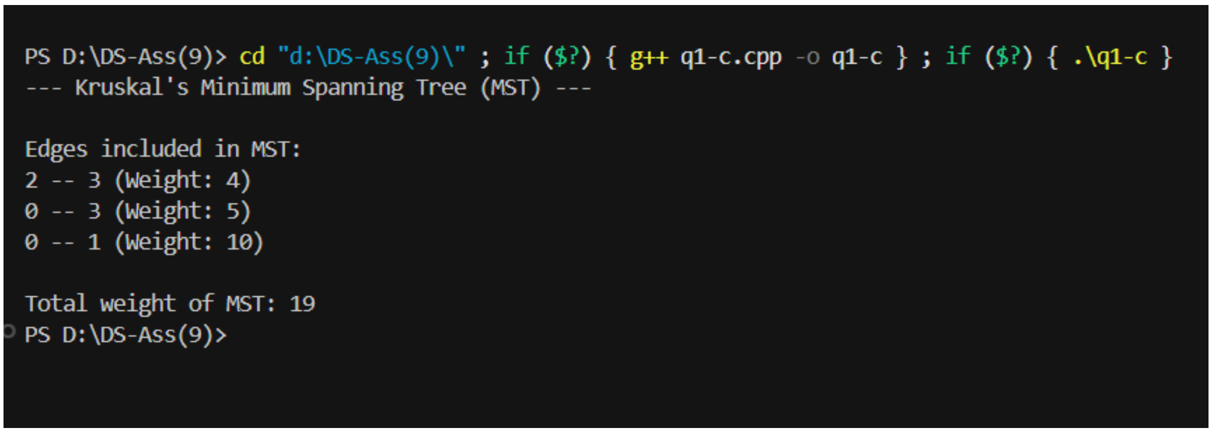
q1-c)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Structure to represent a weighted edge
struct Edge {
    int src, dest, weight;
};

// Comparison function for sorting edges by weight
bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}
```

```cpp
// Disjoint Set Union (DSU) structure
class DSU {
private:
    vector<int> parent;
    vector<int> rank;

public:
    DSU(int n) {
        // C++98 compatible initialization
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }
    }

    // Find the representative (root) of the set containing element i
(with path compression)
    int find(int i) {
        if (parent[i] == i)
            return i;
        return parent[i] = find(parent[i]);
    }

    // Union of two sets (union by rank/height)
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};
```

```cpp
// Kruskal's Algorithm implementation
void kruskalMST(int V, vector<Edge>& edges) {
    // 1. Sort all the edges in non-decreasing order of their weight
    sort(edges.begin(), edges.end(), compareEdges);

    DSU dsu(V);
    vector<Edge> resultMST;
    int mstWeight = 0;
    int edgesCount = 0;

    cout << "Edges included in MST:\n";

    for (size_t i = 0; i < edges.size() && edgesCount < V - 1; ++i) {
        Edge nextEdge = edges[i];

        int rootSrc = dsu.find(nextEdge.src);
        int rootDest = dsu.find(nextEdge.dest);

        // 2. Check if including this edge forms a cycle
        if (rootSrc != rootDest) {
            resultMST.push_back(nextEdge);
            dsu.unite(rootSrc, rootDest);
            mstWeight += nextEdge.weight;
            edgesCount++;

            cout << nextEdge.src << " -- " << nextEdge.dest << "
(Weight: " << nextEdge.weight << ")\n";
        }
    }

    cout << "\nTotal weight of MST: " << mstWeight << "\n";
}

int main() {
    int V = 4; // Number of vertices (0, 1, 2, 3)
    vector<Edge> edges;

    // Graph from common example (V=4, E=5)
    Edge e1 = {0, 1, 10}; edges.push_back(e1);
    Edge e2 = {0, 2, 6}; edges.push_back(e2);
    Edge e3 = {0, 3, 5}; edges.push_back(e3);
    Edge e4 = {1, 3, 15}; edges.push_back(e4);
    Edge e5 = {2, 3, 4}; edges.push_back(e5);
    cout << "--- Kruskal's Minimum Spanning Tree (MST) ---\n\n";
    kruskalMST(V, edges);

    return 0;
}
```

Output->



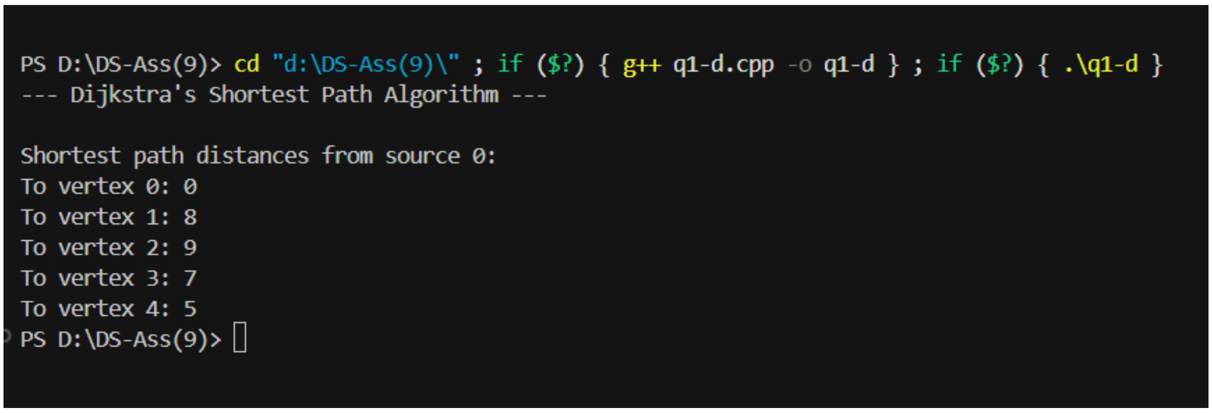
```
PS D:\DS-Ass(9)> cd "d:\DS-Ass(9)\" ; if ($?) { g++ q1-c.cpp -o q1-c } ; if ($?) { .\q1-c }
--- Kruskal's Minimum Spanning Tree (MST) ---

Edges included in MST:
2 -- 3 (Weight: 4)
0 -- 3 (Weight: 5)
0 -- 1 (Weight: 10)

Total weight of MST: 19
PS D:\DS-Ass(9)>
```

q1-d)

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <limits>

using namespace std;

// Infinity value (C++98 compatible)
const int INF = 999999;

// Dijkstra's Shortest Path implementation
void dijkstra(int V, const vector<vector<pair<int, int> > >& adj, int
startNode) {
    // Stores {distance, vertex} pairs. 'greater' makes it a min-heap.
    priority_queue<pair<int, int>, vector<pair<int, int> >,
greater<pair<int, int> > > pq;

    // Distance array, initialized to infinity
    vector<int> dist(V, INF);

    // Distance of source node to itself is 0
    dist[startNode] = 0;
    pq.push(make_pair(0, startNode)); // {distance, vertex}

    while (!pq.empty()) {
        // Extract the vertex with the minimum distance
        int d = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        // If the extracted distance is greater than the current
recorded distance, skip (stale entry)
        if (d > dist[u])
            continue;

        // Traverse all adjacent vertices of u
        for (size_t i = 0; i < adj[u].size(); ++i) {
            int v = adj[u][i].first;
            int weight = adj[u][i].second;

            // Relaxation step: If a shorter path is found to v through
u
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    // Print the shortest distances
    cout << "Shortest path distances from source " << startNode <<
":\n";
    for (int i = 0; i < V; ++i) {
        cout << "To vertex " << i << ": ";
        // C++98 compatible way to print INF or the distance
        if (dist[i] == INF) {
            cout << "INF";
        } else {
            cout << dist[i];
        }
        cout << "\n";
    }
}

int main() {
```

```cpp
    int V = 5; // Number of vertices (0 to 4)
    // Adjacency List: stores {neighbor_vertex, weight}
    vector<vector<pair<int, int> > > adj(V);

    // Graph from a common example:
    // Edge: (u, v, weight)
    adj[0].push_back(make_pair(1, 10));
    adj[0].push_back(make_pair(4, 5));

    adj[1].push_back(make_pair(2, 1));
    adj[1].push_back(make_pair(4, 2));

    adj[2].push_back(make_pair(3, 4));

    adj[3].push_back(make_pair(2, 6));
    adj[3].push_back(make_pair(0, 7)); // Added edge to show complex
path

    adj[4].push_back(make_pair(1, 3));
    adj[4].push_back(make_pair(2, 9));
    adj[4].push_back(make_pair(3, 2));

    cout << "--- Dijkstra's Shortest Path Algorithm ---\n\n";

    // Find shortest paths starting from vertex 0
    dijkstra(V, adj, 0);

    return 0;
}
```

Output->



```
PS D:\DS-Ass(9)> cd "d:\DS-Ass(9)\" ; if ($?) { g++ q1-d.cpp -o q1-d } ; if ($?) { .\q1-d }
--- Dijkstra's Shortest Path Algorithm ---

Shortest path distances from source 0:
To vertex 0: 0
To vertex 1: 8
To vertex 2: 9
To vertex 3: 7
To vertex 4: 5
PS D:\DS-Ass(9)>
```