DSA Assignment 10

Name: Jashnoor Singh
Roll: 1024030106

Q1:

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

class Graph {
private:
    int V; // number of vertices
    bool directed;
    vector<vector<int>> adjMatrix;
    vector<vector<int>> adjList;
    public:
    Graph(int V, bool directed = false) {
        this->V = V;
        this->directed = directed;
        adjMatrix.assign(V, vector<int>(V, 0));
        adjList.assign(V, vector<int>());
    }
    void addEdge(int u, int v) {
        adjMatrix[u][v] = 1;
        adjList[u].push_back(v);

        // If undirected, add edges both ways
        if (!directed) {
            adjMatrix[v][u] = 1;
            adjList[v].push_back(u);
        }
    }
    // Print adjacency matrix
    void printAdjMatrix() {
        cout << "\nAdjacency Matrix:\n";
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++)
                cout << adjMatrix[i][j] << " ";
            cout << "\n";
        }
    }
    // Print adjacency list
    void printAdjList() {
        cout << "\nAdjacency List:\n";
        for (int i = 0; i < V; i++) {
            cout << i << " -> ";
            for (int v : adjList[i]) cout << v << " ";
            cout << "\n";
```

```
        }
    }
    // Degree of vertex (for undirected graph)
    int degree(int u) {
        return adjList[u].size();
    }
    // Out-degree (for directed)
    int outDegree(int u) {
        return adjList[u].size();
    }
    // In-degree (for directed)
    int inDegree(int u) {
        int count = 0;
        for (int i = 0; i < V; i++)
            for (int v : adjList[i])
                if (v == u) count++;
        return count;
    }
    // Adjacent vertices
    void printAdjacent(int u) {
        cout << "Adjacent vertices of " << u << ": ";
        for (int v : adjList[u])
            cout << v << " ";
        cout << "\n";
    }
    // Count edges
    int numberOfEdges() {
        int count = 0;

        if (directed) {
            for (int i = 0; i < V; i++)
                count += adjList[i].size();
            return count;     // every edge counted once
        } else {
            for (int i = 0; i < V; i++)
                count += adjList[i].size();
            return count / 2; // each edge counted twice
        }
    }
};

int main() {
    Graph g(5, false); // false = undirected graph (set true for
directed)

    g.addEdge(0,1);
    g.addEdge(0,2);
    g.addEdge(1,3);
    g.addEdge(2,3);
    g.addEdge(3,4);

    g.printAdjMatrix();
```

```
    g.printAdjList();

    cout << "\nDegree of vertex 3 = " << g.degree(3) << "\n";
    cout << "Adjacent vertices of 3: ";
    g.printAdjacent(3);

    cout << "Number of edges = " << g.numberOfEdges() << "\n";

}
```

Output:

```
Adjacency Matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 1 0
0 1 1 0 1
0 0 0 1 0

Adjacency List:
0 -> 1 2
1 -> 0 3
2 -> 0 3
3 -> 1 2 4
4 -> 3

Degree of vertex 3 = 3
Adjacent vertices of 3: Adjacent vertices of 3: 1 2 4
Number of edges = 5
```

Q2:

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

class Graph {
    int V;                          // number of vertices
    vector<vector<int>> adjList;    // adjacency list
    public:
    Graph(int V) {
        this->V = V;
        adjList.assign(V, vector<int>());
```

```
    }

    void addEdge(int u, int v) {
        adjList[u].push_back(v);   // directed graph
        adjList[v].push_back(u);   // add this line if you want
UNDIRECTED
    }

    void BFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;
        visited[start] = true;
        q.push(start);
        cout << "BFS Traversal starting from vertex " << start <<
": ";
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cout << node << " ";
            for (int neighbour : adjList[node]) {
                if (!visited[neighbour]) {
                    visited[neighbour] = true;
                    q.push(neighbour);
                }
            }
        }
        cout << endl;
    }
};

int main() {
    Graph g(6);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 4);
    g.addEdge(3, 5);
    g.addEdge(4, 5);

    g.BFS(0);
}
```

Output:

```
BFS Traversal starting from vertex 0: 0 1 2 3 4 5
```

Q3:

```
#include <iostream>
#include <vector>
using namespace std;

class Graph {
    int V;                              // number of vertices
    vector<vector<int>> adjList;    // adjacency list
    public:
    Graph(int V) {
        this->V = V;
        adjList.resize(V);
    }
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u);    // add this for UNDIRECTED 
graph
    }
    void DFSUtil(int node, vector<bool> &visited) {
        visited[node] = true;
        cout << node << " ";
        for (int neighbour : adjList[node]) {
            if (!visited[neighbour])
                DFSUtil(neighbour, visited);
        }
    }
    void DFS(int start) {
        vector<bool> visited(V, false);
        cout << "DFS traversal starting from vertex " << start <<
": ";
        DFSUtil(start, visited);
        cout << endl;
    }
};

int main() {
    Graph g(6);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 4);
    g.addEdge(3, 5);
    g.addEdge(4, 5);
    g.DFS(0);
}
```

Output:

```
DFS traversal starting from vertex 0: 0 1 3 5 4 2
```

Q4:

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

// Structure to store graph edges
struct Edge {
    int u, v, wt;
};

// Disjoint Set Union (Union-Find) with path compression
class DSU {
    vector<int> parent, rank;

    public:
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }
    int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]);   // path compression
        return parent[x];
    }
    void unite(int x, int y) {
        int px = find(x);
        int py = find(y);
        if (px != py) {
            if (rank[px] < rank[py])
                parent[px] = py;
            else if (rank[px] > rank[py])
                parent[py] = px;
            else {
                parent[py] = px;
                rank[px]++;
            }
        }
    }
};

// Kruskal's algorithm
```

```cpp
void kruskalMST(vector<Edge> &edges, int V) {
    // Step 1: Sort edges by weight
    sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
        return a.wt < b.wt;
    });

    DSU dsu(V);

    int mstCost = 0;
    vector<Edge> mstEdges;

    // Step 2: Pick edges in increasing weight order
    for (auto &e : edges) {
        if (dsu.find(e.u) != dsu.find(e.v)) {
            dsu.unite(e.u, e.v);
            mstCost += e.wt;
            mstEdges.push_back(e);
        }
    }

    // Output the MST
    cout << "Minimum Spanning Tree Edges:\n";
    for (auto &e : mstEdges) {
        cout << e.u << " - " << e.v << " (weight: " << e.wt << ")
\n";
    }
    cout << "Total cost of MST = " << mstCost << endl;
}

int main() {
    int V = 6; // number of vertices
    // Graph edges (u, v, weight)
    vector<Edge> edges = {
        {0, 1, 4},
        {0, 2, 4},
        {1, 2, 2},
        {1, 3, 5},
        {2, 3, 5},
        {2, 4, 11},
        {3, 4, 2},
        {3, 5, 1},
        {4, 5, 7}
    };
    kruskalMST(edges, V);
}
```

Output:

```
Minimum Spanning Tree Edges:
3 - 5 (weight: 1)
1 - 2 (weight: 2)
3 - 4 (weight: 2)
0 - 1 (weight: 4)
1 - 3 (weight: 5)
Total cost of MST = 14
```

Q5:

```
#include <iostream>
using namespace std;

#define INF 999999

int main() {
    int n;
    cout << "Enter number of vertices: ";
    cin >> n;
    int graph[20][20];
    cout << "Enter the adjacency matrix (0 for no edge):\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> graph[i][j];
            if (graph[i][j] == 0)
                graph[i][j] = INF;     // no edge
        }
    }
    int selected[20] = {0};
    selected[0] = 1;   // start from vertex 0
    int no_of_edges = 0;
    int total_cost = 0;
    cout << "\nEdges in Minimum Spanning Tree:\n";
    while (no_of_edges < n - 1) {
        int min = INF;
        int x = 0, y = 0;
        // Pick the smallest edge connecting selected → non-
selected
        for (int i = 0; i < n; i++) {
            if (selected[i]) {
                for (int j = 0; j < n; j++) {
                    if (!selected[j] && graph[i][j] < min) {
                        min = graph[i][j];
                        x = i;
                        y = j;
                    }
                }
```

```
            }
        }
        cout << x << " -- " << y << "  cost: " << graph[x][y] <<
endl;
        total_cost += graph[x][y];
        selected[y] = 1;
        no_of_edges++;
    }
    cout << "\nTotal cost of MST = " << total_cost << endl;
}
```

Q6:

```
#include <iostream>
using namespace std;

#define INF 999999

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

    int graph[20][20];

    cout << "Enter the adjacency matrix (0 for no edge):\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> graph[i][j];
            if (graph[i][j] == 0 && i != j)
                graph[i][j] = INF; // Treat 0 as no edge
        }
    }

    int src;
    cout << "Enter the source vertex (0 to " << n - 1 << "): ";
    cin >> src;

    int dist[20];          // shortest distances
    int visited[20] = {0};

    // Initialize distances
    for (int i = 0; i < n; i++) {
        dist[i] = graph[src][i];
    }
    dist[src] = 0;
    visited[src] = 1;

    // Dijkstra's main loop
    for (int count = 1; count < n; count++) {
```

```
        int minDist = INF, u = -1;

        // Select unvisited vertex with minimum dist[]
        for (int i = 0; i < n; i++) {
            if (!visited[i] && dist[i] < minDist) {
                minDist = dist[i];
                u = i;
            }
        }

        visited[u] = 1;

        // Relax edges
        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] != INF &&
                dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    cout << "\nShortest distance from vertex " << src << ":\n";
    for (int i = 0; i < n; i++) {
        cout << "To " << i << " = " << dist[i] << endl;
    }

    return 0;
}
```