Simple graph

```
#include <iostream>

#include <vector>


using namespace std;


// Graph class

class Graph {

   int V;              // Number of vertices

   vector<vector<int>> adj; // Adjacency list


public:

   // Constructor

   Graph(int vertices) {

      V = vertices;

      adj.resize(V);

   }


   // Function to add an edge between two vertices

   void addEdge(int u, int v) {

      adj[u].push_back(v); // For an undirected graph, you can add adj[v].push_back(u) as well.

   }


   // Function to print the adjacency list representation of the graph

   void printGraph() {

      for (int v = 0; v < V; ++v) {

         cout << "Adjacency list of vertex " << v << ": ";

         for (auto i : adj[v]) {

            cout << i << " ";

         }

         cout << endl;
```

```
        }
    }
};

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    // Create a graph with V vertices
    Graph g(V);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    // Print the adjacency list
    g.printGraph();

    return 0;
}
```

Undirected graph, check contain cycle or no

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

```
using namespace std;

// Graph class
class Graph {
    int V;                    // Number of vertices
    vector<vector<int>> adj;    // Adjacency list

public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adj.resize(V);
    }

    // Function to add an edge between two vertices
    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u); // For an undirected graph, add this line to consider both directions
    }

    // Function to check if the graph contains a cycle
    bool hasCycle() {
        vector<bool> visited(V, false);

        for (int v = 0; v < V; ++v) {
            if (!visited[v]) {
                if (hasCycleUtil(v, visited, -1)) {
                    return true;
                }
            }
        }

        return false;
    }

private:
    // Utility function for cycle detection using DFS
    bool hasCycleUtil(int v, vector<bool>& visited, int parent) {
        visited[v] = true;

        for (auto i : adj[v]) {
            if (!visited[i]) {
                if (hasCycleUtil(i, visited, v)) {
                    return true;
                }
            }
            else if (i != parent) {
                return true;
            }
        }

        return false;
    }
};

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;
    // Create a graph with V vertices
    Graph g(V);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    // Check if the graph contains a cycle
    bool hasCycle = g.hasCycle();

    if (hasCycle) {
        cout << "The graph contains a cycle." << endl;
    } else {
        cout << "The graph does not contain a cycle." << endl;
    }

    return 0;
}
```

Given directed graph, topological sort

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

// Graph class
class Graph {
    int V;                    // Number of vertices
    vector<vector<int>> adj;    // Adjacency list

public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adj.resize(V);
    }

    // Function to add an edge between two vertices
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    // Function to perform topological sort
    vector<int> topologicalSort() {
        vector<bool> visited(V, false);
        stack<int> stk;

        for (int v = 0; v < V; ++v) {
            if (!visited[v]) {
                topologicalSortUtil(v, visited, stk);
            }
        }

        vector<int> result;
        while (!stk.empty()) {
            result.push_back(stk.top());
            stk.pop();
        }
        return result;
    }

private:
    // Utility function for topological sort using DFS
    void topologicalSortUtil(int v, vector<bool>& visited, stack<int>& stk) {
        visited[v] = true;

        for (auto i : adj[v]) {
            if (!visited[i]) {
                topologicalSortUtil(i, visited, stk);
            }
        }

        stk.push(v);
    }
};

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    // Create a graph with V vertices
    Graph g(V);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v;

        cin >> u >> v;

        g.addEdge(u, v);

    }


    // Perform topological sort

    vector<int> sorted = g.topologicalSort();


    cout << "Topological Order: ";

    for (auto v : sorted) {

        cout << v << " ";

    }

    cout << endl;


    return 0;

}
```

Given a weighted directed graph and a source vertex, write a function to find the shortest path from the source vertex to all other vertices using Dijkstra's algorithm.

```
#include <iostream>

#include <vector>

#include <queue>

#include <climits>


using namespace std;


// Graph class

class Graph {

    int V;                    // Number of vertices

    vector<vector<pair<int, int>>> adj;   // Adjacency list of pairs (destination, weight)


public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adj.resize(V);
    }

    // Function to add a weighted edge between two vertices
    void addEdge(int u, int v, int weight) {
        adj[u].push_back(make_pair(v, weight));
    }

    // Function to find the shortest path from a source vertex to all other vertices using Dijkstra's
algorithm
    vector<int> dijkstra(int source) {
        vector<int> dist(V, INT_MAX);   // Initialize distances to infinity
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

        dist[source] = 0;
        pq.push(make_pair(0, source));

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

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

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

        return dist;
    }
};

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    // Create a graph with V vertices
    Graph g(V);

    cout << "Enter the edges and weights (format: source destination weight):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v, weight;
        cin >> u >> v >> weight;
        g.addEdge(u, v, weight);
    }

    int source;
    cout << "Enter the source vertex: ";
    cin >> source;

    // Find the shortest path from the source vertex to all other vertices
    vector<int> shortestPath = g.dijkstra(source);
    cout << "Shortest Path from vertex " << source << ":" << endl;
    for (int i = 0; i < V; ++i) {
        cout << "To vertex " << i << ": ";
        if (shortestPath[i] != INT_MAX) {
            cout << shortestPath[i] << endl;
        } else {
            cout << "No path" << endl;
        }
    }

    return 0;
}
```

BFS template

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

using namespace std;

// Graph class
class Graph {
    int V;                  // Number of vertices
    vector<vector<int>> adjList;    // Adjacency list of vertices

public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adjList.resize(V);
    }

    // Function to add an edge between two vertices
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u);   // For an undirected graph
    }

    // Breadth-First Search traversal of the graph
    void BFS(int source) {
        queue<int> q;               // Queue to store vertices to be visited
        unordered_set<int> visited;  // Set to keep track of visited vertices

        q.push(source);
        visited.insert(source);

        while (!q.empty()) {
            int currVertex = q.front();
            q.pop();

            cout << currVertex << " ";

            for (int neighbor : adjList[currVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    q.push(neighbor);
                    visited.insert(neighbor);
                }
            }
        }
    }
};
```

```
int main() {
   int V, E;
   cout << "Enter the number of vertices: ";
   cin >> V;
   cout << "Enter the number of edges: ";
   cin >> E;

   // Create a graph with V vertices
   Graph g(V);

   cout << "Enter the edges (format: source destination):" << endl;
   for (int i = 0; i < E; ++i) {
      int u, v;
      cin >> u >> v;
      g.addEdge(u, v);
   }

   int source;
   cout << "Enter the source vertex: ";
   cin >> source;

   cout << "BFS Traversal: ";
   g.BFS(source);

   return 0;
}
```

DFS template

```
#include <iostream>
#include <vector>
#include <stack>
```

```
#include <unordered_set>

using namespace std;

// Graph class
class Graph {
   int V;                          // Number of vertices
   vector<vector<int>> adjList;    // Adjacency list of vertices

public:
   // Constructor
   Graph(int vertices) {
      V = vertices;
      adjList.resize(V);
   }

   // Function to add an edge between two vertices
   void addEdge(int u, int v) {
      adjList[u].push_back(v);
      adjList[v].push_back(u);   // For an undirected graph
   }

   // Depth-First Search traversal of the graph
   void DFS(int source) {
      stack<int> s;               // Stack to store vertices to be visited
      unordered_set<int> visited;  // Set to keep track of visited vertices

      s.push(source);
      visited.insert(source);

      while (!s.empty()) {
            int currVertex = s.top();
            s.pop();

            cout << currVertex << " ";

            for (int neighbor : adjList[currVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    s.push(neighbor);
                    visited.insert(neighbor);
                }
            }
        }
    }
};

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    // Create a graph with V vertices
    Graph g(V);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }
    int source;
    cout << "Enter the source vertex: ";
    cin >> source;

    cout << "DFS Traversal: ";
    g.DFS(source);

    return 0;
}
```

You are given an undirected graph with 'n' vertices and 'm' edges. Implement a BFS traversal algorithm to visit all the vertices in the graph and print them in the order they are visited.

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

using namespace std;

// Graph class
class Graph {
    int V;                     // Number of vertices
    vector<vector<int>> adjList;    // Adjacency list of vertices

public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adjList.resize(V);
    }
    // Function to add an edge between two vertices
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u);   // For an undirected graph
    }

    // Breadth-First Search traversal of the graph
    void BFS(int source) {
        queue<int> q;               // Queue to store vertices to be visited
        unordered_set<int> visited;  // Set to keep track of visited vertices

        q.push(source);
        visited.insert(source);

        while (!q.empty()) {
            int currVertex = q.front();
            q.pop();

            cout << currVertex << " ";

            for (int neighbor : adjList[currVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    q.push(neighbor);
                    visited.insert(neighbor);
                }
            }
        }
    }
};

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

    // Create a graph with n vertices
    Graph g(n);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    cout << "BFS Traversal: ";
    g.BFS(0); // Starting BFS from vertex 0

    return 0;
}
```

You are given an undirected graph with 'n' vertices and 'm' edges. Implement a BFS traversal algorithm to visit all the vertices in the graph and print them in the order they are visited. Additionally, count the number of connected components in the graph.

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

using namespace std;

// Graph class
```

```
class Graph {
    int V;                          // Number of vertices
    vector<vector<int>> adjList;    // Adjacency list of vertices

public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adjList.resize(V);
    }

    // Function to add an edge between two vertices
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u);   // For an undirected graph
    }

    // Breadth-First Search traversal of the graph
    void BFS(int source, int& components) {
        queue<int> q;               // Queue to store vertices to be visited
        unordered_set<int> visited;  // Set to keep track of visited vertices

        q.push(source);
        visited.insert(source);

        while (!q.empty()) {
            int currVertex = q.front();
            q.pop();

            cout << currVertex << " ";
            for (int neighbor : adjList[currVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    q.push(neighbor);
                    visited.insert(neighbor);
                }
            }
        }

        components++;
    }

    // Function to perform BFS traversal and count connected components
    int BFSandCount() {
        int components = 0;
        unordered_set<int> visited;

        for (int v = 0; v < V; v++) {
            if (visited.find(v) == visited.end()) {
                BFS(v, components);
                visited.insert(v);
            }
        }

        return components;
    }
};

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

    // Create a graph with n vertices
    Graph g(n);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    cout << "BFS Traversal: ";
    int components = g.BFSandCount();
    cout << endl;
    cout << "Number of Connected Components: " << components << endl;

    return 0;
}
```

You are given an undirected graph with 'n' vertices and 'm' edges. Implement a BFS traversal algorithm to visit all the vertices in the graph and print them in the order they are visited. Additionally, find and print the shortest path distance from a given source vertex to every other vertex in the graph.

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

using namespace std;
```

```
// Graph class
class Graph {
  int V;                    // Number of vertices
  vector<vector<int>> adjList;   // Adjacency list of vertices

public:
  // Constructor
  Graph(int vertices) {
    V = vertices;
    adjList.resize(V);
  }

  // Function to add an edge between two vertices
  void addEdge(int u, int v) {
    adjList[u].push_back(v);
    adjList[v].push_back(u);   // For an undirected graph
  }

  // Breadth-First Search traversal of the graph
  void BFS(int source, vector<int>& distances) {
    queue<int> q;             // Queue to store vertices to be visited
    unordered_set<int> visited;  // Set to keep track of visited vertices

    q.push(source);
    visited.insert(source);
    distances[source] = 0;

    while (!q.empty()) {
      int currVertex = q.front();
      q.pop();
```

```
            for (int neighbor : adjList[currVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    q.push(neighbor);
                    visited.insert(neighbor);
                    distances[neighbor] = distances[currVertex] + 1;
                }
            }
        }
    }

    // Function to find shortest path distances from a given source vertex
    void findShortestPaths(int source) {
        vector<int> distances(V, INT_MAX);
        BFS(source, distances);

        cout << "Shortest Path Distances from Source Vertex " << source << ":" << endl;
        for (int i = 0; i < V; ++i) {
            cout << source << " -> " << i << " : ";
            if (distances[i] != INT_MAX)
                cout << distances[i];
            else
                cout << "Not Reachable";
            cout << endl;
        }
    }
};

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

    // Create a graph with n vertices
    Graph g(n);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    int source;
    cout << "Enter the source vertex: ";
    cin >> source;

    cout << "BFS Traversal: ";
    vector<int> dummy(n);
    g.BFS(source, dummy);
    for (int i = 0; i < n; ++i)
        cout << i << " ";
    cout << endl;

    g.findShortestPaths(source);

    return 0;
}
```

You are given an undirected graph with 'n' vertices and 'm' edges. Implement a DFS traversal algorithm to visit all the vertices in the graph and print them in the order they are visited. Additionally, count the number of connected components in the graph.

```
#include <iostream>
#include <vector>
#include <stack>
#include <unordered_set>

using namespace std;

// Graph class
class Graph {
    int V;                       // Number of vertices
    vector<vector<int>> adjList;    // Adjacency list of vertices

public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adjList.resize(V);
    }

    // Function to add an edge between two vertices
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u);   // For an undirected graph
    }

    // Depth-First Search traversal of the graph
    void DFS(int source, int& components) {
        stack<int> stk;             // Stack to store vertices to be visited
        unordered_set<int> visited;  // Set to keep track of visited vertices

        stk.push(source);
        visited.insert(source);

        while (!stk.empty()) {
            int currVertex = stk.top();
            stk.pop();

            cout << currVertex << " ";

            for (int neighbor : adjList[currVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    stk.push(neighbor);
                    visited.insert(neighbor);
                }
            }
        }

        components++;
    }

    // Function to perform DFS traversal and count connected components
    int DFSandCount() {
        int components = 0;
        unordered_set<int> visited;

        for (int v = 0; v < V; v++) {
            if (visited.find(v) == visited.end()) {
                DFS(v, components);
                visited.insert(v);
            }
        }
```

```
        return components;
    }
};

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

    // Create a graph with n vertices
    Graph g(n);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    cout << "DFS Traversal: ";
    int components = g.DFSandCount();
    cout << endl;
    cout << "Number of Connected Components: " << components << endl;

    return 0;
}
```

You are given an undirected graph with 'n' vertices and 'm' edges. Implement a DFS traversal algorithm to visit all the vertices in the graph and print them in the order they are visited. Additionally, find and print the number of connected components in the graph, along with the vertices in each connected component.

```
#include <iostream>

#include <vector>

#include <stack>

#include <unordered_set>

#include <algorithm>


using namespace std;


// Graph class

class Graph {

    int V;                          // Number of vertices

    vector<vector<int>> adjList;    // Adjacency list of vertices


public:

    // Constructor

    Graph(int vertices) {

        V = vertices;

        adjList.resize(V);

    }


    // Function to add an edge between two vertices

    void addEdge(int u, int v) {

        adjList[u].push_back(v);

        adjList[v].push_back(u);   // For an undirected graph

    }


    // Depth-First Search traversal of the graph

    void DFS(int source, vector<int>& component, unordered_set<int>& visited) {

        stack<int> stk;             // Stack to store vertices to be visited


        stk.push(source);
        visited.insert(source);

        while (!stk.empty()) {
            int currVertex = stk.top();
            stk.pop();

            component.push_back(currVertex);

            for (int neighbor : adjList[currVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    stk.push(neighbor);
                    visited.insert(neighbor);
                }
            }
        }
    }

    // Function to perform DFS traversal and count connected components
    void DFSandCount() {
        int components = 0;
        unordered_set<int> visited;
        vector<vector<int>> connectedComponents;

        for (int v = 0; v < V; v++) {
            if (visited.find(v) == visited.end()) {
                vector<int> component;
                DFS(v, component, visited);
                connectedComponents.push_back(component);
                components++;
            }
        }
```

```
        cout << "DFS Traversal: ";
        for (const auto& component : connectedComponents) {
            for (int vertex : component)
                cout << vertex << " ";
        }
        cout << endl;

        cout << "Number of Connected Components: " << components << endl;
        cout << "Connected Components:" << endl;
        for (int i = 0; i < components; ++i) {
            cout << "Component " << i + 1 << ": ";
            sort(connectedComponents[i].begin(), connectedComponents[i].end());
            for (int vertex : connectedComponents[i])
                cout << vertex << " ";
            cout << endl;
        }
    }
};

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

    // Create a graph with n vertices
    Graph g(n);

    cout << "Enter the edges (format: source destination):" << endl;
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    g.DFSandCount();

    return 0;
}
```

MST template

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

using namespace std;

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

public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adjList.resize(V);
    }

    // Function to add an edge between two vertices with a given weight
```

```
void addEdge(int u, int v, int weight) {
    adjList[u].push_back({v, weight});
    adjList[v].push_back({u, weight});   // For an undirected graph
}

// Function to find the minimum spanning tree using Prim's algorithm
void findMinimumSpanningTree() {
    vector<int> parent(V, -1);        // Array to store the parent of each vertex in MST
    vector<int> key(V, INT_MAX);      // Array to store the minimum weight for each vertex
    vector<bool> inMST(V, false);     // Array to track whether a vertex is included in the MST

    // Priority queue to get the vertex with the minimum key
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    // Start with vertex 0 as the root
    int root = 0;
    pq.push({0, root});
    key[root] = 0;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        inMST[u] = true;

        // Traverse through all adjacent vertices of u
        for (auto& neighbor : adjList[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            // If v is not in MST and weight of (u,v) is smaller than current key of v
            if (!inMST[v] && weight < key[v]) {
                    // Update the key and parent of v
                    key[v] = weight;
                    parent[v] = u;
                    pq.push({key[v], v});
                }
            }
        }

        // Print the edges of the minimum spanning tree
        cout << "Edges of the Minimum Spanning Tree:" << endl;
        for (int i = 1; i < V; ++i) {
            cout << parent[i] << " - " << i << endl;
        }
    }
};

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

    // Create a graph with n vertices
    Graph g(n);

    cout << "Enter the edges (format: source destination weight):" << endl;
    for (int i = 0; i < m; ++i) {
        int u, v, weight;
        cin >> u >> v >> weight;
        g.addEdge(u, v, weight);
    }

    g.findMinimumSpanningTree();

    return 0;
}
```

You are given a weighted, undirected graph with 'n' vertices and 'm' edges. Implement Prim's algorithm to find the minimum spanning tree of the graph and calculate the total weight of the MST.

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

using namespace std;

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

public:
    // Constructor
    Graph(int vertices) {
        V = vertices;
        adjList.resize(V);
    }

    // Function to add an edge between two vertices with a given weight
    void addEdge(int u, int v, int weight) {
        adjList[u].push_back({v, weight});
        adjList[v].push_back({u, weight});   // For an undirected graph
```

```cpp
}

// Function to find the minimum spanning tree using Prim's algorithm
void findMinimumSpanningTree() {
    vector<int> parent(V, -1);        // Array to store the parent of each vertex in MST
    vector<int> key(V, INT_MAX);      // Array to store the minimum weight for each vertex
    vector<bool> inMST(V, false);     // Array to track whether a vertex is included in the MST

    // Priority queue to get the vertex with the minimum key
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    // Start with vertex 0 as the root
    int root = 0;
    pq.push({0, root});
    key[root] = 0;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        inMST[u] = true;

        // Traverse through all adjacent vertices of u
        for (auto& neighbor : adjList[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            // If v is not in MST and weight of (u,v) is smaller than current key of v
            if (!inMST[v] && weight < key[v]) {
                // Update the key and parent of v
                key[v] = weight;
                parent[v] = u;
                pq.push({key[v], v});
            }
        }
    }

    // Print the edges of the minimum spanning tree
    cout << "Edges of the Minimum Spanning Tree:" << endl;
    int totalWeight = 0;
    for (int i = 1; i < V; ++i) {
        cout << parent[i] << " - " << i << endl;
        totalWeight += key[i];
    }

    cout << "Total Weight of the Minimum Spanning Tree: " << totalWeight << endl;
  }
};

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

  // Create a graph with n vertices
  Graph g(n);

  cout << "Enter the edges (format: source destination weight):" << endl;
  for (int i = 0; i < m; ++i) {
      int u, v, weight;
      cin >> u >> v >> weight;
```

```
        g.addEdge(u, v, weight);
    }

    g.findMinimumSpanningTree();

    return 0;
}
```