Question 1:

Imagine where a telecommunication company needs to connect several cities with fiber optic cables to provide high-speed internet services. Each city represents a node in the network, and the distance between cities represents the weight of the edges.

Input Format:

- The first line contains an integer V representing the number of vertices in the graph.
- The second line contains an integer E representing the number of edges in the graph.
- The following E lines contain three space-separated integers each: src, dest, and weight, representing an edge from vertex src to vertex dest with weight weight.

Output Format:

- The output consists of the edges forming the Minimum Spanning Tree (MST) using Kruskal's Algorithm.
- For each edge in the MST, output a line containing three space-separated integers: src, dest, and weight, representing the source vertex, destination vertex, and weight of the edge respectively.

Title for Question 1: Kruskal's Algorithm

Solution:

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

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

// Structure to represent a subset for union-find
struct Subset {
    int parent, rank;
};

class Graph {
private:
    int vertices;
    vector<Edge> edges;

public:
    Graph(int v) : vertices(v) {}

    void addEdge(int src, int dest, int weight) {
        Edge edge = {src, dest, weight};
        edges.push_back(edge);
    }
    // Helper function for finding the parent of a vertex in a subset
    int find(Subset subsets[], int i) {
        if (subsets[i].parent != i)
            subsets[i].parent = find(subsets, subsets[i].parent);
        return subsets[i].parent;
    }

    // Helper function for union of two subsets
    void Union(Subset subsets[], int x, int y) {
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        if (subsets[xroot].rank < subsets[yroot].rank)
            subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
            subsets[yroot].parent = xroot;
        else {
            subsets[yroot].parent = xroot;
            subsets[xroot].rank++;
        }
    }

    // Function to perform Kruskal's algorithm to find MST
    void kruskalMST() {
        vector<Edge> result; // Store the resultant MST
        int e = 0; // Index variable for sorted edges
        int i = 0; // Index variable for result[]

        // Sort all the edges in non-decreasing order of their weights
        sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
            return a.weight < b.weight;
        });

        // Allocate memory for creating subsets
        Subset* subsets = new Subset[vertices];

        // Create subsets with single elements
        for (int v = 0; v < vertices; v++) {
            subsets[v].parent = v;
            subsets[v].rank = 0;
        }

        // Number of edges to be taken is equal to V-1
        while (i < vertices - 1 && e < edges.size()) {
            Edge next_edge = edges[e++];

            int x = find(subsets, next_edge.src);
            int y = find(subsets, next_edge.dest);

            // If including this edge doesn't cause a cycle, include it i
            if (x != y) {
                result.push_back(next_edge);
                Union(subsets, x, y);
                i++;
            }
        }

        // Print the constructed MST
        cout << "Minimum Spanning Tree using Kruskal's Algorithm:\n";
        for (i = 0; i < result.size(); i++)
            cout << result[i].src << " -- " << result[i].dest << " == " <

        delete[] subsets;
    }
};

int main() {
    int vertices, edges, src, dest, weight;;
    cin >> vertices;
    Graph g(vertices);
    cin >> edges;
    for (int i = 0; i < edges; i++) {
        cin >> src >> dest >> weight;
        g.addEdge(src, dest, weight);
    }
    g.kruskalMST();
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	4 5 0 1 10 0 2 6 0 3 5 1 3 15 2 3 4	Minimum Spanning Tree using Kruskal's Algorithm: 2 -- 3 == 4 0 -- 3 == 5 0 -- 1 == 10
2	5 7 0 1 2 0 2 1 1 2 3 1 3 4 2 4 5 3 4 6 1 4 7	Minimum Spanning Tree using Kruskal's Algorithm: 0 -- 2 == 1 0 -- 1 == 2 1 -- 3 == 4 2 -- 4 == 5
3	6 9 0 1 1 0 2 3 1 2 4 1 3 2 2 3 5 3 4 6 2 4 7 3 5 8 4 5 9	Minimum Spanning Tree using Kruskal's Algorithm: 0 -- 1 == 1 1 -- 3 == 2 0 -- 2 == 3 3 -- 4 == 6 3 -- 5 == 8
4	3 3 0 1 4 0 2 5 1 2 6	Minimum Spanning Tree using Kruskal's Algorithm: 0 -- 1 == 4 0 -- 2 == 5
5	5 7 0 1 3 0 2 5 1 2 1 1 3 2 2 3 6 3 4 4 2 4 7	Minimum Spanning Tree using Kruskal's Algorithm: 1 -- 2 == 1 1 -- 3 == 2 0 -- 1 == 3 3 -- 4 == 4
6	5 5 0 1 1 0 2 2 1 2 3 1 3 4 2 3 5	Minimum Spanning Tree using Kruskal's Algorithm: 0 -- 1 == 1 0 -- 2 == 2 1 -- 3 == 4

White List:

Black List:

Question 2:

Imagine you are tasked with designing a network infrastructure for a large organization that spans multiple locations. The network topology includes various buildings and departments, and each department operates independently with its own set of connections. However, there is a requirement to establish a reliable and cost-effective communication network that connects all departments while minimizing the overall cost of cables and connections. In this context, modify Kruskal's Algorithm to handle disconnected graphs and find the minimum spanning forest for the entire network. Consider factors such as optimizing the total cable length, ensuring connectivity between all departments, and minimizing the cost of establishing and maintaining the network.

Input Format:

- The first line contains two integers, V and E, representing the number of vertices and edges in the graph.
- The next E lines each contain three integers, src, dest, and weight, representing an edge from vertex src to vertex dest with a given weight.

Output Format:

- The Minimum Spanning Forest is printed to the console, with each line representing an edge in the forest.
- Each line contains three integers, src, dest, and weight, separated by spaces, indicating an edge from vertex src to vertex dest with the given weight in the minimum spanning forest.

Title for Question 2: Handling Disconnected Graphs

Solution:

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

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    vector<Edge> edges;
};

// Disjoint Set Data Structure
struct DisjointSet {
    vector<int> parent, rank;

    DisjointSet(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }
    }

    int find(int v) {
        if (v != parent[v]) {
            parent[v] = find(parent[v]);
        }
        return parent[v];
    }

    void unionSets(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]++;
            }
        }
    }
};

bool compareEdges(const Edge &a, const Edge &b) {
    return a.weight < b.weight;
}

void kruskal(Graph &graph) {
    sort(graph.edges.begin(), graph.edges.end(), compareEdges);

    DisjointSet ds(graph.V);

    vector<Edge> result;

    for (const Edge &edge : graph.edges) {
        int rootSrc = ds.find(edge.src);
        int rootDest = ds.find(edge.dest);

        if (rootSrc != rootDest) {
            result.push_back(edge);
            ds.unionSets(rootSrc, rootDest);
        }
    }

    cout << "Minimum Spanning Forest:" << endl;
    for (const Edge &edge : result) {
        cout << edge.src << " -- " << edge.dest << " : " << edge.weight <
    }
}

int main() {
    Graph graph;
    cin >> graph.V >> graph.E;
    for (int i = 0; i < graph.E; ++i) {
        Edge edge;
        cin >> edge.src >> edge.dest >> edge.weight;
        graph.edges.push_back(edge);
    }
    kruskal(graph);
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	4 5 0 1 2 0 2 3 0 3 1 1 2 4 2 3 5	Minimum Spanning Forest: 0 -- 3 : 1 0 -- 1 : 2 0 -- 2 : 3

S.No	Inputs	Outputs
2	5 7 0 1 2 0 2 3 0 3 1 1 2 4 2 3 5 3 4 6 2 4 7	Minimum Spanning Forest: 0 -- 3 : 1 0 -- 1 : 2 0 -- 2 : 3 3 -- 4 : 6
3	3 3 0 1 1 1 2 2 2 0 3	Minimum Spanning Forest: 0 -- 1 : 1 1 -- 2 : 2
4	6 9 0 1 2 0 2 3 0 3 1 1 2 4 2 3 5 3 4 6 2 4 7 4 5 8 1 5 9	Minimum Spanning Forest: 0 -- 3 : 1 0 -- 1 : 2 0 -- 2 : 3 3 -- 4 : 6 4 -- 5 : 8
5	4 4 0 1 1 1 2 2 2 3 3 3 0 4	Minimum Spanning Forest: 0 -- 1 : 1 1 -- 2 : 2 2 -- 3 : 3
6	5 5 0 1 1 1 2 2 2 3 3 3 4 4 4 0 5	Minimum Spanning Forest: 0 -- 1 : 1 1 -- 2 : 2 2 -- 3 : 3 3 -- 4 : 4

White List:

Black List:

Question 3:

Imagine you are a network engineer tasked with optimizing the layout of communication cables for a city's newly established smart grid. The goal is to minimize the total cost of laying cables while ensuring that every building in the city is connected to the smart grid. Apply Kruskal's algorithm to determine the most cost-effective and efficient network design for the smart grid in this scenario. Consider factors such as the distance between buildings, existing infrastructure, and cost constraints associated with laying cables.

Input Format:

- The first line contains an integer representing the number of vertices in the graph, denoted as vertices.
- The second line contains an integer representing the number of edges in the graph, denoted as edges.
- The next edges lines each contain three space-separated integers: src, dest, and weight. These represent an undirected edge between vertex src and vertex dest with a weight of weight.

?Output Format:

- The program outputs the Minimum Spanning Tree (MST) using Kruskal's algorithm. The output format consists of several lines.
- Each line contains three space-separated integers: src, dest, and weight.
- The lines collectively represent the edges in the Minimum Spanning Tree.
- The edges are sorted in ascending order based on their weights.

Title for Question 3: Real-world Graph Application

Solution:

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

using namespace std;

// Structure to represent an edge in the graph
struct Edge {
    int src, dest, weight;
};

// Structure to represent a subset for union-find
struct Subset {
    int parent, rank;
};

class Graph {
    int V, E;
    vector<Edge> edges;

public:
    Graph(int vertices, int edgesCount) : V(vertices), E(edgesCount) {}

    // Add an edge to the graph
    void addEdge(int src, int dest, int weight) {
        edges.push_back({src, dest, weight});
    }

    // Function to find set of an element i
    int find(Subset subsets[], int i) {
        if (subsets[i].parent != i)
            subsets[i].parent = find(subsets, subsets[i].parent);

        return subsets[i].parent;
    }

    // Function that does union of two sets
    void Union(Subset subsets[], int x, int y) {
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        if (subsets[xroot].rank < subsets[yroot].rank)
            subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
            subsets[yroot].parent = xroot;
        else {
            subsets[yroot].parent = xroot;
            subsets[xroot].rank++;
        }
    }

    // Function to run Kruskal's algorithm
    void Kruskal() {
        vector<Edge> result;
        int i = 0, e = 0;

        // Sort edges in ascending order based on their weights
        sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
            return a.weight < b.weight;
        });
        // Allocate memory for creating V subsets
        Subset* subsets = new Subset[V];

        // Create V subsets with single elements
        for (int v = 0; v < V; v++) {
            subsets[v].parent = v;
            subsets[v].rank = 0;
        }

        // Keep adding edges until we reach V-1 edges in the MST
        while (e < V - 1 && i < E) {
            Edge next_edge = edges[i++];

            int x = find(subsets, next_edge.src);
            int y = find(subsets, next_edge.dest);

            // If including this edge doesn't cause a cycle, add it to th
            if (x != y) {
                result.push_back(next_edge);
                Union(subsets, x, y);
                e++;
            }
        }

        // Display the minimum spanning tree
        cout << "Minimum Spanning Tree:\n";
        for (auto edge : result) {
            cout << edge.src << " - " << edge.dest << " : " << edge.weigh
        }

        delete[] subsets;
    }
};

int main() {
    int vertices, edges;
    cin >> vertices;
    cin >> edges;
    Graph graph(vertices, edges);
    for (int i = 0; i < edges; i++) {
        int src, dest, weight;
        cin >> src >> dest >> weight;
        graph.addEdge(src, dest, weight);
    }
    graph.Kruskal();
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	4 5 0 1 1 0 2 2 1 2 3 1 3 4 2 3 5	Minimum Spanning Tree: 0 - 1 : 1 0 - 2 : 2 1 - 3 : 4
2	5 7 0 1 2 0 2 1 1 2 3 1 3 4 2 3 5 2 4 7 3 4 6	Minimum Spanning Tree: 0 - 2 : 1 0 - 1 : 2 1 - 3 : 4 3 - 4 : 6
3	4 4 0 1 1 0 2 2 0 3 3 1 2 4	Minimum Spanning Tree: 0 - 1 : 1 0 - 2 : 2 0 - 3 : 3

S.No	Inputs	Outputs
4	5 7 0 1 2 0 2 1 0 3 3 1 3 4 1 4 5 2 3 7 3 4 6	Minimum Spanning Tree: 0 - 2 : 1 0 - 1 : 2 0 - 3 : 3 1 - 4 : 5
5	6 9 0 1 3 0 2 5 1 2 2 1 3 4 2 3 6 2 4 7 3 4 8 3 5 9 4 5 1	Minimum Spanning Tree: 4 - 5 : 1 1 - 2 : 2 0 - 1 : 3 1 - 3 : 4 2 - 4 : 7
6	3 3 0 1 1 0 2 1 1 2 1	Minimum Spanning Tree: 0 - 1 : 1 0 - 2 : 1

White List:

Black List:

Question 4:

You are working on a project that involves optimizing the construction of a new network infrastructure in a city. The city is represented as a graph, where each vertex represents a location, and the edges represent the possible connections between these locations. The construction cost varies for each connection, and you want to find the minimum-cost spanning tree to efficiently connect all the locations. You may assume that the graph is connected and that there is at least one valid solution. Your solution should provide a clear and organized way for users to input the graph, view the input graph, and then obtain the minimum-cost spanning tree using Kruskal's algorithm.

Input Format:

- The input consists of two integers, V and E, separated by a newline, where V represents the number of vertices in the graph and E represents the number of edges.
- Following that are E lines, each containing three space-separated integers: src, dest, and weight, representing an edge from vertex src to vertex dest with weight weight.

Output Format:

- The output begins with a printout of the input graph.
- After that, the Minimum Spanning Tree (MST) obtained using Kruskal's Algorithm is printed.
- The output consists of multiple lines, each containing either an edge of the original graph or an edge of the MST.
- For each edge printed, it is represented by its source vertex, weight, and destination vertex, separated by appropriate delimiters.

Title for Question 4: Graph Input/Output

Solution:

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

```
struct Edge {
    int src, dest, weight;
};

class Graph {
public:
    int V, E;
    vector<Edge> edges;

    Graph(int vertices, int edges) : V(vertices), E(edges) {}

    void addEdge(int src, int dest, int weight) {
        Edge edge = {src, dest, weight};
        edges.push_back(edge);
    }

    void inputGraph() {
        cin >> V >> E;
        for (int i = 0; i < E; ++i) {
            int src, dest, weight;
            cin >> src >> dest >> weight;
            addEdge(src, dest, weight);
        }
    }

    void printGraph() {
        cout << "Graph:\n";
        for (const Edge& edge : edges) {
            cout << edge.src << " --(" << edge.weight << ")-- " << edge.d
        }
    }

    vector<Edge> kruskalMST();
};

bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}

int findParent(int parent[], int i) {
    if (parent[i] == -1)
        return i;
    return findParent(parent, parent[i]);
}

void unionSets(int parent[], int x, int y) {
    int xSet = findParent(parent, x);
    int ySet = findParent(parent, y);
    parent[xSet] = ySet;
}

vector<Edge> Graph::kruskalMST() {
    vector<Edge> result;

    // Sort edges in ascending order
    sort(edges.begin(), edges.end(), compareEdges);

    int parent[V];
    fill(parent, parent + V, -1);
    for (const Edge& edge : edges) {
        int x = findParent(parent, edge.src);
        int y = findParent(parent, edge.dest);

        // If including this edge does not form a cycle, include it in re
        if (x != y) {
            result.push_back(edge);
            unionSets(parent, x, y);
        }
    }
    return result;
}

int main() {
    Graph graph(0, 0);
    graph.inputGraph();
    graph.printGraph();
    vector<Edge> mst = graph.kruskalMST();
    cout << "\nMinimum Spanning Tree (Kruskal's Algorithm):\n";
    for (const Edge& edge : mst) {
        cout << edge.src << " --(" << edge.weight << ")-- " << edge.dest
    }
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	6 9 0 1 2 0 2 4 1 2 1 1 3 3 2 3 5 2 4 6 3 4 8 3 5 7 4 5 9	Graph: 0 --(2)-- 1 0 --(4)-- 2 1 --(1)-- 2 1 --(3)-- 3 2 --(5)-- 3 2 --(6)-- 4 3 --(8)-- 4 3 --(7)-- 5 4 --(9)-- 5 Minimum Spanning Tree (Kruskal's Algorithm): 1 --(1)-- 2 0 --(2)-- 1 1 --(3)-- 3 2 --(6)-- 4 3 --(7)-- 5
2	5 7 0 1 4 0 2 2 1 2 3 1 3 1 2 3 5 2 4 6 3 4 3	Graph: 0 --(4)-- 1 0 --(2)-- 2 1 --(3)-- 2 1 --(1)-- 3 2 --(5)-- 3 2 --(6)-- 4 3 --(3)-- 4 Minimum Spanning Tree (Kruskal's Algorithm): 1 --(1)-- 3 0 --(2)-- 2 1 --(3)-- 2 3 --(3)-- 4
3	4 5 0 1 1 0 2 3 1 2 2 1 3 5 2 3 4	Graph: 0 --(1)-- 1 0 --(3)-- 2 1 --(2)-- 2 1 --(5)-- 3 2 --(4)-- 3 Minimum Spanning Tree (Kruskal's Algorithm): 0 --(1)-- 1 1 --(2)-- 2 2 --(4)-- 3
4	5 7 0 1 2 0 2 4 1 2 1 1 3 3 2 3 5 2 4 6 3 4 8	Graph: 0 --(2)-- 1 0 --(4)-- 2 1 --(1)-- 2 1 --(3)-- 3 2 --(5)-- 3 2 --(6)-- 4 3 --(8)-- 4 Minimum Spanning Tree (Kruskal's Algorithm): 1 --(1)-- 2 0 --(2)-- 1 1 --(3)-- 3 2 --(6)-- 4
5	6 9 0 1 3 0 2 1 1 2 4 1 3 7 2 3 2 2 4 5 3 4 6 3 5 8 4 5 9	Graph: 0 --(3)-- 1 0 --(1)-- 2 1 --(4)-- 2 1 --(7)-- 3 2 --(2)-- 3 2 --(5)-- 4 3 --(6)-- 4 3 --(8)-- 5 4 --(9)-- 5 Minimum Spanning Tree (Kruskal's Algorithm): 0 --(1)-- 2 2 --(2)-- 3 0 --(3)-- 1 2 --(5)-- 4 3 --(8)-- 5
6	4 5 0 1 2 0 2 3 1 2 1 1 3 4 2 3 5	Graph: 0 --(2)-- 1 0 --(3)-- 2 1 --(1)-- 2 1 --(4)-- 3 2 --(5)-- 3 Minimum Spanning Tree (Kruskal's Algorithm): 1 --(1)-- 2 0 --(2)-- 1 1 --(4)-- 3

White List:

Black List:

Question 5:

Imagine you are a network administrator responsible for designing the network infrastructure of a small town. The town has several buildings, and you need to connect them using a reliable network of cables. Each building represents a node in the graph, and the cables between the buildings represent the edges with associated costs (the cost can be the distance, latency, or any relevant metric). Consider a scenario where the buildings are schools, hospitals, and government offices, and the edges represent the cost of laying cables between them. Apply Prim's algorithm to find the minimum spanning tree for this network.

Input Format:

- The input begins with an integer V, representing the number of vertices in the graph.
- The next line contains an integer E, representing the number of edges in the graph.
- Subsequently, there are E lines, each containing three space-separated integers u, v, and w, where u and v are the vertices connected by the edge, and w is the weight of that edge.

Output Format:

- The output consists of lines representing the edges in the minimum spanning tree (MST) found by Prim's algorithm.
- Each line contains two vertices and the weight of the corresponding edge in the MST.

Title for Question 5: Prim's Algorithm

Solution:

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

using namespace std;

class Prim {
public:
    Prim(int vertices);
    void addEdge(int u, int v, int weight);
    void primMST();

private:
    int vertices;
    vector<vector<int>> graph;
    int minKey(const vector<int>& key, const vector<bool>& mstSet);
    void printMST(const vector<int>& parent);
};

Prim::Prim(int vertices) {
    this->vertices = vertices;
    graph = vector<vector<int>>(vertices, vector<int>(vertices, 0));
}
```

```
void Prim::addEdge(int u, int v, int weight) {
    // Adding edges to the adjacency matrix
    graph[u][v] = weight;
    graph[v][u] = weight;
}

void Prim::primMST() {
    vector<int> parent(vertices, -1); // Array to store constructed MST
    vector<int> key(vertices, INT_MAX); // Key values used to pick the mi
    vector<bool> mstSet(vertices, false); // To represent the set of vert

    // Initialize the source vertex with a key value of 0
    key[0] = 0;

    // MST will have V vertices
    for (int count = 0; count < vertices - 1; ++count) {
        // Pick the minimum key vertex from the set of vertices not yet i
        int u = minKey(key, mstSet);

        // Include the picked vertex in MST
        mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices
        for (int v = 0; v < vertices; ++v) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printMST(parent);
}

int Prim::minKey(const vector<int>& key, const vector<bool>& mstSet) {
    // Find the vertex with the minimum key value from the set of vertice
    int minKey = INT_MAX, minIndex;

    for (int v = 0; v < vertices; ++v) {
        if (!mstSet[v] && key[v] < minKey) {
            minKey = key[v];
            minIndex = v;
        }
    }

    return minIndex;
}

void Prim::printMST(const vector<int>& parent) {
    cout << "Edge \tWeight\n";
    for (int i = 1; i < vertices; ++i) {
        cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]] <<
    }
}

int main() {
    int vertices, edges;
    cin >> vertices;
    Prim g(vertices);
    cin >> edges;
    for (int i = 0; i < edges; ++i) {
        int u, v, weight;
        cin >> u >> v >> weight;
        g.addEdge(u, v, weight);
    }
    g.primMST();
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	5 7 0 1 2 0 3 6 1 2 3 1 3 8 1 4 5 2 4 7 3 4 9	Edge Weight 0 - 1 2 1 - 2 3 0 - 3 6 1 - 4 5
2	4 4 0 1 1 1 2 2 2 3 3 3 0 4	Edge Weight 0 - 1 1 1 - 2 2 2 - 3 3
3	6 9 0 1 5 0 2 2 1 2 4 1 3 9 2 3 8 2 4 7 3 4 1 3 5 3 4 5 6	Edge Weight 2 - 1 4 0 - 2 2 4 - 3 1 2 - 4 7 3 - 5 3
4	5 7 0 1 2 0 3 6 1 2 3 1 3 8 1 4 5 2 4 7 3 4 1	Edge Weight 0 - 1 2 1 - 2 3 4 - 3 1 1 - 4 5
5	3 3 0 1 4 0 2 6 1 2 8	Edge Weight 0 - 1 4 0 - 2 6
6	4 5 0 1 1 0 2 3 0 3 4 1 2 2 2 3 5	Edge Weight 0 - 1 1 1 - 2 2 0 - 3 4

White List:

Black List:

Question 6:

Imagine you are working on a logistics optimization project for a delivery company that needs to efficiently connect various warehouses in a city. The warehouses are represented as nodes in a graph, and the edges between them indicate the possible routes. Your task is to write a function to calculate the weight of the minimum spanning tree (MST) using Prim's algorithm and also return the edges of the spanning tree.

Input Format:

- The first line contains an integer vertices, representing the number of vertices in the graph.
- The second line contains an integer edges, representing the number of edges in the graph.
- The next edges lines each contain three integers u, v, and weight, representing an undirected edge between vertices u and v with the given weight.

Output Format:

- The program outputs a single line containing an integer representing the weight of the minimum spanning tree.

Title for Question 6: Minimum Spanning Tree Weight Calculation

Solution:

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

typedef pair<int, int> iPair;

class Graph {
private:
    int V; // Number of vertices
    vector<vector<iPair>> adjList; // Adjacency list to represent the gra

public:
    Graph(int vertices);
    void addEdge(int u, int v, int weight);
    int primMST();
};

Graph::Graph(int vertices) {
    this->V = vertices;
    adjList.resize(V);
}

void Graph::addEdge(int u, int v, int weight) {
    adjList[u].push_back({v, weight});
    adjList[v].push_back({u, weight});
}

int Graph::primMST() {
    priority_queue<iPair, vector<iPair>, greater<iPair>> pq;

    int src = 0; // Start from vertex 0
    vector<int> key(V, INT_MAX); // Key values to pick minimum weight edg
    vector<bool> inMST(V, false); // To represent vertices included in MS

    pq.push({0, src});
    key[src] = 0;

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

        inMST[u] = true;

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

            if (!inMST[v] && key[v] > weight) {
                key[v] = weight;
                pq.push({key[v], v});
            }
        }
    }

    int minWeight = 0;
    for (int i = 0; i < V; ++i) {
        minWeight += key[i];
    }

    return minWeight;
}

int main() {
    int vertices, edges;
    cin >> vertices;
    Graph g(vertices);
    cin >> edges;
    for (int i = 0; i < edges; ++i) {
        int u, v, weight;
        cin >> u >> v >> weight;
        g.addEdge(u, v, weight);
    }
    int minWeight = g.primMST();
    cout << "The weight of the minimum spanning tree is: " << minWeight <
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	4 5 0 1 2 0 2 3 1 2 1 1 3 4 2 3 5	The weight of the minimum spanning tree is: 7
2	5 7 0 1 2 0 2 1 0 3 4 1 3 2 1 4 3 2 4 5 3 4 1	The weight of the minimum spanning tree is: 6
3	6 9 0 1 2 0 2 3 0 3 1 1 3 5 1 4 4 2 3 2 2 4 1 3 4 7 3 5 6	The weight of the minimum spanning tree is: 12
4	3 3 0 1 2 0 2 1 1 2 3	The weight of the minimum spanning tree is: 3
5	5 8 0 1 1 0 2 4 0 3 6 1 2 4 1 3 3 1 4 2 2 3 2 3 4 5	The weight of the minimum spanning tree is: 8
6	4 6 0 1 5 0 2 6 0 3 2 1 2 7 1 3 1 2 3 4	The weight of the minimum spanning tree is: 7

White List:

Black List: