Practical No 2:

1. **Using any suitable programming language write a program to find Minimum Cost Spanning**

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

struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

vector<int> parent, rank;

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

void union_sets(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
```

```
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}

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

    cout << "Enter edges (u v weight):\n";
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        edges.push_back({u, v, w});
    }

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

```
    sort(edges.begin(), edges.end());

    int cost = 0;
    cout << "Edges in MST:\n";
    for (Edge e : edges) {
        if (find(e.u) != find(e.v)) {
            cost += e.weight;
            cout << e.u << " - " << e.v << " = " << e.weight << endl;
            union_sets(e.u, e.v);
        }
    }

    cout << "Minimum Cost = " << cost << endl;
    return 0;
}
```

g++ kruskal.cpp -o kruskal

./Kruskal

2. Using any suitable programming language write a program to find Minimum Cost Spanning

Tree of a given undirected graph using Prim‟s algorithm

```
// prim.cpp
#include <iostream>
#include <vector>
#include <limits.h>
using namespace std;
```

```
int minKey(vector<int>& key, vector<bool>& mstSet, int V) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (!mstSet[v] && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

void primMST(vector<vector<int>>& graph, int V) {
    vector<int> parent(V);
    vector<int> key(V, INT_MAX);
    vector<bool> mstSet(V, false);

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet, V);
        mstSet[u] = true;

        for (int v = 0; v < V; v++)
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
```

```
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]] << endl;
}

int main() {
    int V;
    cout << "Enter number of vertices: ";
    cin >> V;
    vector<vector<int>> graph(V, vector<int>(V));

    cout << "Enter adjacency matrix:\n";
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            cin >> graph[i][j];

    primMST(graph, V);
    return 0;
}
```

g++ prim.cpp -o prim

./prim

3. Using any suitable programming language write a program to from a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijikstra‟s algorithm

```
// dijkstra.cpp
#include <iostream>
```

```
#include <vector>
#include <limits.h>
using namespace std;

int minDistance(vector<int>& dist, vector<bool>& sptSet, int V) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
      if (!sptSet[v] && dist[v] <= min)
         min = dist[v], min_index = v;

   return min_index;
}

void dijkstra(vector<vector<int>>& graph, int src, int V) {
   vector<int> dist(V, INT_MAX);
   vector<bool> sptSet(V, false);

   dist[src] = 0;

   for (int count = 0; count < V - 1; count++) {
      int u = minDistance(dist, sptSet, V);
      sptSet[u] = true;

      for (int v = 0; v < V; v++)
         if (!sptSet[v] && graph[u][v] &&
            dist[u] != INT_MAX &&
            dist[u] + graph[u][v] < dist[v])
```

```
                dist[v] = dist[u] + graph[u][v];
    }

    cout << "Vertex \tDistance from Source\n";
    for (int i = 0; i < V; i++)
        cout << i << "\t" << dist[i] << endl;
}

int main() {
    int V, src;
    cout << "Enter number of vertices: ";
    cin >> V;
    vector<vector<int>> graph(V, vector<int>(V));

    cout << "Enter adjacency matrix:\n";
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            cin >> graph[i][j];

    cout << "Enter source vertex: ";
    cin >> src;

    dijkstra(graph, src, V);
    return 0;
}
```

g++ dijkstra.cpp -o dijkstra

./Dijkstra

4. Using any suitable programming language write a program to implement Knapsack problems using Greedy method

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

struct Item {
    int weight, value;
    double ratio;
};

bool compare(Item a, Item b) {
    return a.ratio > b.ratio;
}

void fractionalKnapsack(int W, vector<Item>& items) {
    sort(items.begin(), items.end(), compare);

    double totalValue = 0.0;
    for (auto item : items) {
        if (W == 0) break;

        if (item.weight <= W) {
            W -= item.weight;
```

```
            totalValue += item.value;
        } else {
            totalValue += item.ratio * W;
            W = 0;
        }
    }

    cout << "Maximum value in Knapsack = " << totalValue << endl;
}

int main() {
    int n, W;
    cout << "Enter number of items and Knapsack capacity: ";
    cin >> n >> W;

    vector<Item> items(n);
    cout << "Enter value and weight of each item:\n";
    for (int i = 0; i < n; i++) {
        cin >> items[i].value >> items[i].weight;
        items[i].ratio = (double)items[i].value / items[i].weight;
    }

    fractionalKnapsack(W, items);
    return 0;
}
```

g++ knapsack.cpp -o knapsack

./knapsack