```cpp
#include <bits/stdc++.h>
using namespace std;

string color[10];
int dis[10];
int pred[10];

void BFS(vector<int> graph[], int source, int n)
{
    for (int i = 0; i < n; i++) {
        color[i] = "w";
        dis[i] = INT_MAX;
        pred[i] = -1;
    }
    queue<int> q;
    color[source] = "g";
    dis[source] = 0;
    q.push(source);
    while(!q.empty()) {
        int u = q.front();
        q.pop();
        for (int i = 0; i < graph[u].size(); i++) {
            int v = graph[u][i];
            if (color[v] == "w") {
                color[v] = "g";
                dis[v] = dis[u] + 1;
                pred[v] = u;
                q.push(v);
            }
        }
        color[u] = "b";
    }
}

void path(int source, int des)
{
    if (des == -1) return;
    int parent = pred[des];
    path(source, parent);
    cout << des << " ";
}

int main()
```

```
{
    vector<int> graph[10];

    graph[0].push_back(1);
    graph[1].push_back(0);
    graph[1].push_back(2);
    graph[2].push_back(1);
    graph[2].push_back(3);
    graph[2].push_back(4);
    graph[3].push_back(2);
    graph[4].push_back(2);

    BFS(graph, 0, 4);

    cout << dis[3] << endl;

    path(0, 3);

    return 0;
}
```

Bellman Ford

```cpp
#include<bits/stdc++.h>
using namespace std;

void bellman_ford(vector<int>graph[], vector<int>cost[], int n, int source)
{

    int dis[n+1];
    int parent[n+1];
    //memset(parent, -1, sizeof(parent));
    for (int i = 0; i <= n; i++) {
        parent[i] = -1;
        dis[i] = 10000;
    }
    dis[source] = 0;

    for (int k = 0; k < n-1; k++) {
        for (int i = 1; i <= n; i++) {  // handle each node
            int u = i;
            for (int j = 0; j < graph[u].size(); j++) {
                int v = graph[u][j];
                int w = cost[u][j];
                // Relaxation
                if (dis[u] + w < dis[v]) {
                    dis[v] = dis[u] + w;
                    parent[v] = u;
                }
            }
        }
    }

    cout << "dis" << endl;
    for (int i = 1; i <= n; i++) {
        cout << dis[i] << " ";
    }
    return;
}

int main()
{
    int n, m;
    cout << "Enter the no of node : ";
    cin >> n;
    vector<int> graph[n+1], cost[n+1];
```

```
    cout << "Enter the no of edge : ";
    cin >> m;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u].push_back(v);
        cost[u].push_back(w);
    }
    bellman_ford(graph, cost, n, 1);
    return 0;
}
```

Dijkstra

```cpp
#include<bits/stdc++.h>
using namespace std;

vector<int> graph[20], cost[20];
int dis[20], parent[20];

void dijkstra(int source, int n)
{
    priority_queue<pair<int, int> > pq;
    for (int i = 0; i < n; i++) {
        dis[i] = INT_MAX;
        parent[i] = -1;
    }
    dis[source] = 0;
    pq.push({-dis[source], source});
    while (!pq.empty()) {
        pair<int, int> node;
        node = pq.top();
        pq.pop();
        int u = node.second;
        for (int i = 0; i < graph[u].size(); i++) {
            int v = graph[u][i];
            int w = cost[u][i];

            // Relaxation
            if (dis[u] + w < dis[v]) {
                dis[v] = dis[u] + w;
                parent[v] = u;
                pq.push({-dis[v], v});
            }
        }
    }
}

int main()
{
    int n, m, u, v, w;
    cout << "Enter the no. of vertex and edges : ";
    cin >> n >> m;          // n = no of node, m = no of edge
    cout << "Enter u v w for each edge : " << endl;
    for (int i = 0; i < m; i++) {
        cin >> u >> v >> w;
        graph[u].push_back(v);
```

```
        cost[u].push_back(w);
    }
    dijkstra(0, n);
    for (int i = 0; i < n; i++) {
        cout << dis[i] << " ";
    }
    cout << endl;
    return 0;
}
```

BFS_in_Grid_direction_array

```cpp
#include<bits/stdc++.h>
using namespace std;

int fx[] = {1, 0, -1, 0};
int fy[] = {0, 1, 0, -1};

// BFS in GRID (using direction array)
void bfs_grid(int n, int x, int y)
{
     int dis[n][n], color[n][n];
     queue<pair<int, int> > q;
     for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dis[i][j] = INT_MAX;
            color[i][j] = -1;
        }
     }
     dis[x][y] = 0;
     color[x][y] = 1;   // -1->white, 1->gray, 2->black
     q.push({x, y});
     while (!q.empty()) {
        pair<int, int> u = q.front();
        q.pop();
        for (int i = 0; i < 4; i++) {
            int new_x = u.first + fx[i];
            int new_y = u.second + fy[i];
            if (new_x >= 0 && new_x < n && new_y >= 0 && new_y < n &&
color[new_x][new_y] == -1) {
                color[new_x][new_y] = 1;
                dis[new_x][new_y] = dis[u.first][u.second] + 1;
                q.push({new_x, new_y});
            }
        }
        color[u.first][u.second] = 2;
     }
     cout << "Distance:" << endl;
     for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << dis[i][j] << " ";
        }
        cout << endl;
```

```
    }
    return;
}

int main()
{
    int n, x, y;
    cout << "Enter the grid size : ";
    cin >> n;
    cout << "Enter the row and column of source: ";
    cin >> x >> y;
    bfs_grid(n, x, y);
    return 0;
}
```

5_Prims

```
#include<bits/stdc++.h>
using namespace std;

int prims(vector<int>graph[], vector<int>cost[], int start_node)
{
    priority_queue<pair<int,int> > pq;
    bool color[10];
    for (int i = 0; i < 10; i++) color[i] = false;
    pq.push({0, start_node});
    int c = 0;
    while (!pq.empty()) {
        pair<int, int> node = pq.top();
        pq.pop();
        if (color[node.second] == false) {
            c = c + (-node.first);
            int u = node.second;
            for (int i = 0; i < graph[u].size(); i++) {
                int v = graph[u][i];
                int w = cost[u][i];

                if (color[v] == false) {
                    pq.push({-w, v});
                }
            }
            color[u] = true;
        }
    }
    return c;
}

int main()
{
    vector<int> graph[10], cost[10];
    int n, m;
    cout << "Enter no of Nodes: ";
    cin >> n;
    cout << "Enter no of Edges: ";
    cin >> m;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
```

```
        graph[u].push_back(v);
        graph[v].push_back(u);
        cost[u].push_back(w);
        cost[v].push_back(w);
    }
    int mst_cost = prims(graph, cost, 1);
    cout << mst_cost << endl;
    return 0;
}
```

6_ Kruskal Algo

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

// Structure to represent an edge
struct Edge {
   int u, v, weight;
};

// For sorting edges by weight
bool compare(Edge a, Edge b) {
   return a.weight < b.weight;
}

// Disjoint Set Union (Union-Find) structure
class DSU {
   vector<int> parent, rank;
public:
   DSU(int n) {
      parent.resize(n+1);
      rank.resize(n+1, 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) return;
      if(rank[px] < rank[py])
         parent[px] = py;
      else if(rank[px] > rank[py])
         parent[py] = px;
      else {
         parent[py] = px;
         rank[px]++;
      }
   }
};
```

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

    vector<Edge> edges(m);
    cout << "Enter edges (u v weight):\n";
    for(int i=0; i<m; i++)
        cin >> edges[i].u >> edges[i].v >> edges[i].weight;

    // Sort edges by weight
    sort(edges.begin(), edges.end(), compare);

    DSU dsu(n);
    int mst_weight = 0;
    vector<Edge> mst_edges;

    for(auto edge : edges) {
        if(dsu.find(edge.u) != dsu.find(edge.v)) {
            dsu.unite(edge.u, edge.v);
            mst_weight += edge.weight;
            mst_edges.push_back(edge);
        }
    }

    cout << "Minimum Spanning Tree weight: " << mst_weight << "\n";
    cout << "Edges in MST:\n";
    for(auto edge : mst_edges)
        cout << edge.u << " - " << edge.v << " = " << edge.weight << "\n";

    return 0;
}
```