BFS:

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
void bfs(vector<vector<int> >& adjList, int startNode,
        vector<bool>& visited)
{
    queue<int> q;
    visited[startNode] = true;
    q.push(startNode);
    while (!q.empty()) {
        int currentNode = q.front();
        q.pop();
        cout << currentNode << " ";
        for (int neighbor : adjList[currentNode]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
void addEdge(vector<vector<int> >& adjList, int u, int v)
{
    adjList[u].push_back(v);
}
```

```
int main()
{
   int vertices, edges;
   cout << "Enter the number of vertices: ";
   cin >> vertices;
   cout << "Enter the number of edges: ";
   cin >> edges;
   vector<vector<int> > adjList(vertices);
   cout << "Enter edges (format: source destination):" << endl;
   for (int i = 0; i < edges; ++i) {
      int u, v;
      cin >> u >> v;
      addEdge(adjList, u, v);
   }
   vector<bool> visited(vertices, false);
      cout << "Breadth First Traversal starting from vertex 0: ";
   bfs(adjList, 0, visited);
   return 0;
}
```

DFS :

```
#include <iostream>
#include <map>
#include <list>
#include <iterator>
using namespace std;
class Graph {
public:
```

```
    map<int, bool> visited;
    map<int, list<int>> adj;
    void addEdge(int v, int w);
    void DFS(int v);
};
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list
}
void Graph::DFS(int v)
{
    visited[v] = true;
    cout << v << " ";
        list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}
int main()
{
    Graph g;
    int vertices, edges;
    cout << "Enter the number of vertices: ";
    cin >> vertices;
    cout << "Enter the number of edges: ";
    cin >> edges;
    cout << "Enter edges (format: source destination):" << endl;
```

```
    for (int i = 0; i < edges; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }
    int startVertex;
    cout << "Enter the starting vertex for DFS traversal: ";
    cin >> startVertex;
    for (int i = 0; i < vertices; ++i)
        g.visited[i] = false;
    cout << "Depth First Traversal starting from vertex " << startVertex << ": ";
    g.DFS(startVertex);
   return 0;
}
```

Prim's :

```
#include <iostream>
#include <vector>
#include <cstring>
#include <algorithm>
using namespace std;
const int INF = 1e9;
const int MAX = 1e3;
int graph[MAX][MAX];
int n;
int prims(vector<string> cities)
{
    int selected[n];
```

```
memset(selected, false, sizeof(selected));
selected[0] = true;
int no_edge = 0;
int x;
int y;
int min_cost = 0;
cout << "Minimum spanning tree: "<<endl;
while (no_edge < n - 1)
{
   int min = INF;
   x = 0;
   y = 0;
   for (int i = 0; i < n; i++)
   {
      if (selected[i])
      {
         for (int j = 0; j < n; j++)
         {
            if (!selected[j] && graph[i][j])
            {
               if (min > graph[i][j])
               {
                  min = graph[i][j];
                  x = i;
                  y = j;
               }
            }
```

```
            }
          }
        }
        cout << cities[x] << " " << cities[y] << " " << min << endl;
        min_cost += min;
        selected[y] = true;
        no_edge++;
    }
    return min_cost;
}
int main()
{
    cout << "Enter the number of cities: ";
    cin >> n;
    cout<<"Enter the names of the cities: "<<endl;
    vector<string> cities(n);
    for(int i = 0; i < n; i++)
    {
        cin >> cities[i];
    }
    cout<<"Enter the no of the edges: "<<endl;
    int m;
    cin >> m;
    cout<<"Enter the cost of the edges: (city1 city2 cost) "<<endl;
    for(int i = 0; i < m; i++)
    {
        string city1, city2;
```

```
        int cost;
        cin >> city1 >> city2 >> cost;
        int index1 = find(cities.begin(), cities.end(), city1) - cities.begin();
        int index2 = find(cities.begin(), cities.end(), city2) - cities.begin();
        graph[index1][index2] = cost;
        graph[index2][index1] = cost;
    }
    cout << prims(cities) << endl;
    return 0; }
```

Krushkal :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;
const int MAX = 1e3;
vector<pair<int, pair<int, int>>> edges;
int parent[MAX];
int n;
int find(int x)
{
    if(parent[x] == x)
    {
        return x;
    }
    return parent[x] = find(parent[x]);
}
```

```
void union_set(int x, int y)
{
   parent[find(x)] = find(y);
}
int kruskal(vector<string> cities)
{
   sort(edges.begin(), edges.end());
   for(int i = 0; i < n; i++)
   {
      parent[i] = i;
   }
   int min_cost = 0;
   cout << "Minimum spanning tree : "<<endl;
   for(auto e : edges)
   {
      int w = e.first;
      int x = e.second.first;
      int y = e.second.second;
      if(find(x) != find(y))
      {
         cout << cities[x] << " " << cities[y] << " " << w <<
             endl;
         min_cost += w;
         union_set(x, y);
      }
   }
   return min_cost;
```

```
}
int main()
{
   cout << "Enter the total number of cities: ";
   cin >> n;
   cout << "Enter the names of the cities: " << endl;
   vector<string> cities(n);
   for(int i = 0; i < n; i++)
   {
      cin >> cities[i];
   }
   cout << "Enter the number of the edges: " << endl;
   int m;
   cin >> m;
   cout<<"Enter the cost of the edges: (city1 city2 cost)"<<endl;
      for(int i = 0; i < m; i++)
   {
      string city1, city2;
      int cost;
      cin >> city1 >> city2 >> cost;
      int index1 = find(cities.begin(), cities.end(), city1) -
                  cities.begin();
      int index2 = find(cities.begin(), cities.end(), city2) -
                  cities.begin();
      edges.push_back({cost, {index1, index2}});
   }
   cout << kruskal(cities) << endl;
```

```
    return 0;
}
```

Dijkstra's :

```
#include <iostream>
#include <limits.h>
using namespace std;
#define V 9
int minDistance(int dist[], bool sptSet[])
{    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}
void printSolution(int dist[])
{
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t\t\t" << dist[i] << endl;
}
void dijkstra(int graph[V][V], int src)
{
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[src] = 0;
```

```
    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {.
        int u = minDistance(dist, sptSet);
        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];
    }
    printSolution(dist);
}
// driver's code
int main()
{
    int graph[V][V];
    cout << "Enter the adjacency matrix (9x9):" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cin >> graph[i][j];
        }
    }
     int sourceVertex;
    cout << "Enter the source vertex (0 to 8): ";
    cin >> sourceVertex;
    dijkstra(graph, sourceVertex);
   return 0;
}
```

Bellman-Ford :

```
#include <iostream>
#include <limits.h>
using namespace std;
struct Edge {
   int src, dest, weight;
};
struct Graph {
   int V, E; // V-> Number of vertices, E-> Number of edges
   struct Edge* edge; // Graph represented as an array of edges
};
struct Graph* createGraph(int V, int E)
{
   struct Graph* graph = new Graph;
   graph->V = V;
   graph->E = E;
   graph->edge = new Edge[E];
   return graph;
}
void printArr(int dist[], int n)
{
   cout << "Vertex   Distance from Source" << endl;
   for (int i = 0; i < n; ++i)
      cout << i << " \t\t " << dist[i] << endl;
}
void BellmanFord(struct Graph* graph, int src)
{
```

```
    int V = graph->V;
    int E = graph->E;
    int dist[V]; // Array to store the shortest distances
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX; // Set all distances to infinity
    dist[src] = 0; // Distance from source to itself is 0
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
                    if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
     for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            cout << "Graph contains negative weight cycle" << endl;
            return;
        }
    }
        printArr(dist, V);
    return;
}
```

```
// Driver's code
int main()
{
  int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;
    struct Graph* graph = createGraph(V, E);
    cout << "Enter the edges (format: source destination weight):" << endl;
    for (int i = 0; i < E; ++i) {
        cin >> graph->edge[i].src >> graph->edge[i].dest >> graph->edge[i].weight;
    }
  int sourceVertex;
    cout << "Enter the source vertex: ";
    cin >> sourceVertex;
    BellmanFord(graph, sourceVertex);
    return 0;
}
```

Floyd-Warshall :

```
#include <iostream>
#include <limits>
using namespace std;
#define MAX_VERTICES 10 // Maximum number of vertices in the graph
#define INF numeric_limits<int>::max() // Represents infinity
void printMatrix(int matrix[][MAX_VERTICES]); // Function to print the adjacency matrix
void floydWarshall(int graph[][MAX_VERTICES], int nV) {
```

```
  int matrix[MAX_VERTICES][MAX_VERTICES], i, j, k;
  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] != INF && matrix[k][j] != INF &&
            matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
  // Print the shortest paths matrix
  printMatrix(matrix);
}
void printMatrix(int matrix[][MAX_VERTICES]) {
  for (int i = 0; i < MAX_VERTICES; i++) {
    for (int j = 0; j < MAX_VERTICES; j++) {
      if (matrix[i][j] == INF)
        cout << "INF" << "\t"; // Print "INF" for infinity
      else
        cout << matrix[i][j] << "\t"; // Print the distance
    }
    cout << endl;
  }
}
```

```
int main() {
  int nV; // Number of vertices
  cout << "Enter the number of vertices: ";
  cin >> nV;
  int graph[MAX_VERTICES][MAX_VERTICES];
  cout << "Enter the adjacency matrix (" << nV << "x" << nV << "):" << endl;
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
      cin >> graph[i][j];
      if (i != j && graph[i][j] == 0)
        graph[i][j] = INF;
    }
  }
 floydWarshall(graph, nV);
  return 0;
}
```