1. Breadth-First Search (BFS)

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

using namespace std;

void BFS(int start, vector<vector<int>>& adj, int n) {
   vector<bool> visited(n, false);
   queue<int> q;
   visited[start] = true;
   q.push(start);

   while (!q.empty()) {
      int node = q.front();
      q.pop();
      cout << node << " ";  // Print the current node

      for (int neighbor : adj[node]) {
         if (!visited[neighbor]) {
            visited[neighbor] = true;
            q.push(neighbor);
         }
      }
   }
}

int main() {
   int n = 6; // Number of nodes
   vector<vector<int>> adj(n);

   adj[0].push_back(1);
   adj[0].push_back(2);
   adj[1].push_back(3);
   adj[1].push_back(4);
   adj[2].push_back(5);

   cout << "BFS starting from node 0: ";
   BFS(0, adj, n);
   cout << endl;

   return 0;
}
```

2. Depth-First Search (DFS)

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

```cpp
using namespace std;

void DFS(int node, vector<vector<int>>& adj, vector<bool>& visited) {
  visited[node] = true;
  cout << node << " ";  // Print the current node

  for (int neighbor : adj[node]) {
    if (!visited[neighbor]) {
      DFS(neighbor, adj, visited);
    }
  }
}

int main() {
  int n = 6; // Number of nodes
  vector<vector<int>> adj(n);

  adj[0].push_back(1);
  adj[0].push_back(2);
  adj[1].push_back(3);
  adj[1].push_back(4);
  adj[2].push_back(5);

  vector<bool> visited(n, false);

  cout << "DFS starting from node 0: ";
  DFS(0, adj, visited);
  cout << endl;

  return 0;
}
```

3. Prim's Algorithm (Minimum Spanning Tree)

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

using namespace std;

// Structure to represent the edge with its weight
struct Edge {
  int destination;
  int weight;

  // Comparator for the priority queue (min-heap)
  bool operator>(const Edge& other) const {
    return weight > other.weight;
  }
```

```
};

// Function to implement Prim's algorithm
void primMST(int n, vector<vector<Edge>>& adj) {
    vector<int> key(n, INT_MAX);  // Initialize all keys as infinity
    vector<bool> inMST(n, false); // To track which nodes are included in MST
    vector<int> parent(n, -1);    // To store the MST

    key[0] = 0;  // Start from the first node
    priority_queue<Edge, vector<Edge>, greater<Edge>> pq;  // Min-heap for the edges
    pq.push({0, 0});  // Start with node 0 and weight 0

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

        // If the node is already in MST, skip it
        if (inMST[u]) continue;

        // Include this node in MST
        inMST[u] = true;

        // Update the adjacent vertices of the node
        for (const Edge& edge : adj[u]) {
            int v = edge.destination;
            int weight = edge.weight;

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

    // Print the MST
    cout << "Edge \tWeight\n";
    for (int i = 1; i < n; ++i) {
        cout << parent[i] << " - " << i << "\t" << key[i] << endl;
    }
}

int main() {
    int n = 5;  // Number of vertices
    vector<vector<Edge>> adj(n);

    // Add edges to the graph
```

```
    adj[0].push_back({1, 2});
    adj[0].push_back({3, 6});
    adj[1].push_back({0, 2});
    adj[1].push_back({2, 3});
    adj[1].push_back({3, 8});
    adj[2].push_back({1, 3});
    adj[2].push_back({3, 7});
    adj[3].push_back({0, 6});
    adj[3].push_back({1, 8});
    adj[3].push_back({2, 7});

    // Run Prim's algorithm to find MST
    primMST(n, adj);

    return 0;
}
```

4. Kruskal's Algorithm (Minimum Spanning Tree)

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

using namespace std;

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

// Compare function to sort edges by their weight
bool compare(Edge a, Edge b) {
    return a.weight < b.weight;
}

// Disjoint Set Union (DSU) or Union-Find structure
class DSU {
public:
    vector<int> parent, rank;

    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
            parent[i] = i;  // Each node is its own parent initially
        }
    }

    // Find the representative (or root) of the set containing 'x'
    int find(int x) {
```

```
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // Path compression
        }
        return parent[x];
    }

    // Union by rank: merge two sets
    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};

// Function to implement Kruskal's algorithm
void kruskalMST(int n, vector<Edge>& edges) {
    // Sort all edges in non-decreasing order of weight
    sort(edges.begin(), edges.end(), compare);

    DSU dsu(n);  // Initialize DSU for n vertices
    vector<Edge> mst;  // To store the MST edges
    int mstWeight = 0;  // Total weight of the MST

    // Iterate over the sorted edges
    for (Edge edge : edges) {
        int u = edge.u;
        int v = edge.v;
        int weight = edge.weight;

        // If u and v are in different sets, include this edge in the MST
        if (dsu.find(u) != dsu.find(v)) {
            dsu.unionSets(u, v);
            mst.push_back(edge);
            mstWeight += weight;
        }
    }

    // Print the MST and its total weight
```

```
    cout << "Minimum Spanning Tree (MST) Edges: \n";
    for (Edge edge : mst) {
        cout << edge.u << " - " << edge.v << " : " << edge.weight << endl;
    }
    cout << "Total weight of MST: " << mstWeight << endl;
}

int main() {
    int n = 5;  // Number of vertices
    vector<Edge> edges;

    // Add edges to the graph (u, v, weight)
    edges.push_back({0, 1, 2});
    edges.push_back({0, 3, 6});
    edges.push_back({1, 2, 3});
    edges.push_back({1, 3, 8});
    edges.push_back({2, 3, 7});

    // Run Kruskal's algorithm to find the MST
    kruskalMST(n, edges);

    return 0;
}
```

5. Dijkstra's Algorithm (Shortest Path)

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

using namespace std;

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

// Comparator for the priority queue (min-heap)
struct compare {
    bool operator()(const pair<int, int>& a, const pair<int, int>& b) {
        return a.second > b.second;  // Min-heap based on distance
    }
};

// Function to implement Dijkstra's algorithm
void dijkstra(int n, int src, vector<vector<Edge>>& adj) {
    vector<int> dist(n, INT_MAX);  // Store the shortest distance to each node
    dist[src] = 0;  // Distance to the source is 0
```

```
    priority_queue<pair<int, int>, vector<pair<int, int>>, compare> pq;
    pq.push({src, 0});  // Push the source node with distance 0

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

        // If the current distance is already larger than the recorded distance, skip it
        if (current_dist > dist[u]) continue;

        // Iterate over the adjacent nodes
        for (const Edge& edge : adj[u]) {
            int v = edge.v;
            int weight = edge.weight;

            // If a shorter path to v is found, update the distance and push to the priority queue
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({v, dist[v]});
            }
        }
    }

    // Print the shortest distances from the source
    cout << "Vertex \tDistance from Source (" << src << ")\n";
    for (int i = 0; i < n; ++i) {
        cout << i << " \t" << (dist[i] == INT_MAX ? "INF" : to_string(dist[i])) << endl;
    }
}

int main() {
    int n = 5;  // Number of vertices
    vector<vector<Edge>> adj(n);

    // Add edges to the graph (u, v, weight)
    adj[0].push_back({1, 2});
    adj[0].push_back({3, 6});
    adj[1].push_back({0, 2});
    adj[1].push_back({2, 3});
    adj[2].push_back({1, 3});
    adj[2].push_back({3, 7});
    adj[3].push_back({0, 6});
    adj[3].push_back({2, 7});

    int source = 0;  // Starting node for Dijkstra's algorithm
    dijkstra(n, source, adj);
```

```
  return 0;
}
```

6. **Minimum Spanning Tree (MST)**

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

using namespace std;

// Structure to represent an edge with its weight
struct Edge {
  int destination;
  int weight;

  // Comparator for the priority queue (min-heap)
  bool operator>(const Edge& other) const {
    return weight > other.weight;
  }
};

// Function to implement Prim's algorithm
void primMST(int n, vector<vector<Edge>>& adj) {
  vector<int> key(n, INT_MAX);  // Initialize all keys as infinity
  vector<bool> inMST(n, false); // To track which nodes are included in MST
  vector<int> parent(n, -1);    // To store the MST

  key[0] = 0;  // Start from the first node
  priority_queue<Edge, vector<Edge>, greater<Edge>> pq;  // Min-heap for the edges
  pq.push({0, 0});  // Start with node 0 and weight 0

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

    // If the node is already in MST, skip it
    if (inMST[u]) continue;

    // Include this node in MST
    inMST[u] = true;

    // Update the adjacent vertices of the node
    for (const Edge& edge : adj[u]) {
      int v = edge.destination;
      int weight = edge.weight;

      // If vertex v is not in MST and weight is smaller than the current key value
      if (!inMST[v] && weight < key[v]) {
```

```
            key[v] = weight;
            parent[v] = u;
            pq.push({v, key[v]});
          }
        }
    }

    // Print the MST
    cout << "Edge \tWeight\n";
    for (int i = 1; i < n; ++i) {
      cout << parent[i] << " - " << i << "\t" << key[i] << endl;
    }
}

int main() {
    int n = 5;  // Number of vertices
    vector<vector<Edge>> adj(n);

    // Add edges to the graph
    adj[0].push_back({1, 2});
    adj[0].push_back({3, 6});
    adj[1].push_back({0, 2});
    adj[1].push_back({2, 3});
    adj[1].push_back({3, 8});
    adj[2].push_back({1, 3});
    adj[2].push_back({3, 7});
    adj[3].push_back({0, 6});
    adj[3].push_back({1, 8});
    adj[3].push_back({2, 7});

    // Run Prim's algorithm to find MST
    primMST(n, adj);

    return 0;
}
```