Bellman Ford -

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

class Solution {
public:
    vector<int> bellmanFord(int V, vector<vector<int>>& edges, int src) {
        vector<int> dist(V, 1e8);
        dist[src] = 0;

        // Relax all edges V-1 times
        for (int i = 0; i < V - 1; i++) {
            for (auto it : edges) {
                int u = it[0];
                int v = it[1];
                int wt = it[2];

                if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
                    dist[v] = dist[u] + wt;
                }
            }
        }

        // Check for negative weight cycles
        for (auto it : edges) {
            int u = it[0];
            int v = it[1];
            int wt = it[2];
```

```
            if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
                return {-1}; // Negative weight cycle detected
            }
        }

        return dist;
    }
};

int main() {
    int V = 5; // Number of vertices
    int E = 6; // Number of edges

    // Edges: {u, v, weight}
    vector<vector<int>> edges = {
        {0, 1, 2},
        {0, 4, 1},
        {1, 2, 3},
        {2, 3, -6}, // Negative weight edge
        {4, 2, 2},
        {4, 3, 4}
    };

    int src = 0; // Source vertex

    Solution obj;
    vector<int> distances = obj.bellmanFord(V, edges, src);

    if (distances.size() == 1 && distances[0] == -1) {
        cout << "Negative weight cycle detected!" << endl;
    } else {
```

```
        cout << "Shortest distances from node " << src << ":" << endl;
        for (int i = 0; i < V; i++) {
            if (distances[i] == 1e8) {
                cout << "To node " << i << ": INF" << endl;
            } else {
                cout << "To node " << i << ": " << distances[i] << endl;
            }
        }
    }

    return 0;
}
```

Kruskals Algorithms:-

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

// Disjoint Set (Union-Find) with Union by Size and Union by Rank
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);

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

```
        size[i] = 1;
    }
}

int findUltimateParent(int node) {
    if (node == parent[node]) return node;
    return parent[node] = findUltimateParent(parent[node]);
}

void unionByRank(int u, int v) {
    int ulp_u = findUltimateParent(u);
    int ulp_v = findUltimateParent(v);

    if (ulp_u == ulp_v) return;

    if (rank[ulp_u] < rank[ulp_v]) {
        parent[ulp_u] = ulp_v;
    } else if (rank[ulp_v] < rank[ulp_u]) {
        parent[ulp_v] = ulp_u;
    } else {
        parent[ulp_v] = ulp_u;
        rank[ulp_u]++;
    }
}

void unionBySize(int u, int v) {
    int ulp_u = findUltimateParent(u);
    int ulp_v = findUltimateParent(v);

    if (ulp_u == ulp_v) return;
```

```
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        } else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};

// Solution class containing Kruskal's Algorithm (for MST)
class Solution {
public:
    int spanningTree(int V, vector<vector<int>> adj[]) {
        vector<pair<int, pair<int, int>>> edges;

        // Create a list of all edges with their weights
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int adjNode = it[0];
                int wt = it[1];
                int node = i;
                edges.push_back({wt, {node, adjNode}});
            }
        }

        DisjointSet ds(V);
        sort(edges.begin(), edges.end()); // Sort edges by weight

        int mstWeight = 0;
        for (auto it : edges) {
```

```
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            // If u and v are in different components, take the edge
            if (ds.findUltimateParent(u) != ds.findUltimateParent(v)) {
                mstWeight += wt;
                ds.unionBySize(u, v);
            }
        }

        return mstWeight;
    }
};

int main() {
    int V = 5; // Number of vertices
    int E = 6; // Number of edges

    vector<vector<int>> adj[V];

    // Edges: u, v, weight
    vector<vector<int>> edges = {
        {0, 1, 2},
        {0, 3, 6},
        {1, 2, 3},
        {1, 3, 8},
        {1, 4, 5},
        {2, 4, 7}
    };
```

```
    // Building adjacency list
    for (auto it : edges) {
        int u = it[0];
        int v = it[1];
        int wt = it[2];
        adj[u].push_back({v, wt});
        adj[v].push_back({u, wt}); // Because the graph is undirected
    }

    Solution obj;
    int mstWeight = obj.spanningTree(V, adj);

    cout << "The total weight of the Minimum Spanning Tree is: " << mstWeight << endl;

    return 0;
}
```

Dijkstras Algorithm:-

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

class Solution {
public:
    vector<int> dijkstra(int V, vector<pair<int, int>> adj[], int src) {
        priority_queue<pair<int, int>,
               vector<pair<int, int>>,
               greater<pair<int, int>>> pq;
```

```
        vector<int> distance(V, 1e9);
        distance[src] = 0;
        pq.push({0, src});

        while (!pq.empty()) {
            int dist = pq.top().first;
            int node = pq.top().second;
            pq.pop();

            for (auto it : adj[node]) {
                int adjnode = it.first;
                int edgeweight = it.second;

                if (dist + edgeweight < distance[adjnode]) {
                    distance[adjnode] = dist + edgeweight;
                    pq.push({distance[adjnode], adjnode});
                }
            }
        }
        return distance;
    }
};

int main() {
    int V = 5; // number of vertices
    int E = 6; // number of edges
    vector<pair<int, int>> adj[V];

    // Edge list: {u, v, w}
    vector<vector<int>> edges = {
```

```
        {0, 1, 2},
        {0, 4, 1},
        {1, 2, 3},
        {2, 3, 6},
        {4, 2, 2},
        {4, 3, 4}
    };

    // Building the undirected graph
    for (auto edge : edges) {
        int u = edge[0];
        int v = edge[1];
        int w = edge[2];
        adj[u].push_back({v, w});
        adj[v].push_back({u, w}); // remove this line if the graph is directed
    }

    int src = 0;

    Solution obj;
    vector<int> distances = obj.dijkstra(V, adj, src);

    cout << "Shortest distances from node " << src << ":\n";
    for (int i = 0; i < V; i++) {
        cout << "To node " << i << ": " << distances[i] << endl;
    }

    return 0;
}
```

Prims Algorithm:-

```
#include <bits/stdc++.h>
```

```
using namespace std;

class Solution {
  public:
    // Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[]) {
        priority_queue<pair<int, int>,
        vector<pair<int,int>>,
        greater<pair<int,int>>> pq;

        vector<int> vis(V,0);
        vector<int> parent(V, -1); // to track MST edges

        pq.push({0,0});   //{wt,node}
        int sum = 0;

        while(!pq.empty()){
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int wt = it.first;

            if(vis[node] == 1) continue;

            //add to the mst
            vis[node] = 1;
            sum += wt;

            for(auto it : adj[node]){
                int adjNode = it[0];
                int edWeight = it[1];
```

```
            if(!vis[adjNode]){
                pq.push({edWeight, adjNode});
                if(parent[adjNode] == -1 || edWeight < adj[adjNode][0][1]) {
                    parent[adjNode] = node;
                }
            }
        }
    }

    // Print MST edges
    cout << "Edges in MST:\n";
    for(int i = 1; i < V; ++i){
        if(parent[i] != -1)
            cout << parent[i] << " - " << i << endl;
    }

    return sum;
  }
};

// Sample main function
int main() {
  int V = 5;
  vector<vector<int>> adj[V];

  // Add edges: node1 -- weight -- node2
  adj[0].push_back({1, 2});
  adj[1].push_back({0, 2});

  adj[0].push_back({3, 6});
  adj[3].push_back({0, 6});
```

```
    adj[1].push_back({2, 3});
    adj[2].push_back({1, 3});

    adj[1].push_back({3, 8});
    adj[3].push_back({1, 8});

    adj[1].push_back({4, 5});
    adj[4].push_back({1, 5});

    adj[2].push_back({4, 7});
    adj[4].push_back({2, 7});

    Solution obj;
    int totalWeight = obj.spanningTree(V, adj);
    cout << "Total weight of MST: " << totalWeight << endl;

    return 0;
}
```

N Queen

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

class Solution {
public:
    bool isSafe(vector<string> &board, int n , int row, int col){
        // Check vertically above
        for(int i = 0; i < row; i++){
            if(board[i][col] == 'Q') return false;
        }
```

```
    // Check left diagonal
    for(int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--){
        if(board[i][j] == 'Q') return false;
    }

    // Check right diagonal
    for(int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++){
        if(board[i][j] == 'Q') return false;
    }

    return true;
}

void nQueens(vector<string> &board, int row, int n, vector<vector<string>> &ans){
    if(row == n) {
        ans.push_back(board);
        return;
    }

    for(int col = 0; col < n; col++){
        if(isSafe(board, n, row, col)){
            board[row][col] = 'Q';
            nQueens(board, row + 1, n, ans);
            board[row][col] = '.'; // backtrack
        }
    }
}

vector<vector<string>> solveNQueens(int n) {
    vector<string> board(n, string(n, '.'));
```

```
        vector<vector<string>> ans;
        nQueens(board, 0, n, ans);
        return ans;
    }
};

int main() {
    Solution sol;
    int n = 4; // You can change this to any N

    vector<vector<string>> solutions = sol.solveNQueens(n);

    cout << "Total solutions for " << n << "-Queens: " << solutions.size() << endl << endl;

    for(int i = 0; i < solutions.size(); i++){
        cout << "Solution " << i + 1 << ":\n";
        for(const string &row : solutions[i]){
            cout << row << endl;
        }
        cout << endl;
    }

    return 0;
}
```

Job Scheduling SJF:-

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

```
class Solution {
public:
    long long solve(vector<int>& bt) {
        int n = bt.size();
        sort(bt.begin(), bt.end());
        int totalTime = 0;
        int waitingTime = 0;
        for (int i = 0; i < n; i++) {
            waitingTime += totalTime;
            totalTime += bt[i];
        }
        return (long long)waitingTime / n;
    }
};

int main() {
    Solution sol;
    vector<int> burstTimes = {3, 1, 6, 2, 8}; // Sample inputN

    long long avgWaitingTime = sol.solve(burstTimes);

    cout << "Average Waiting Time: " << avgWaitingTime << endl;

    return 0;
}
```

DFS

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

```
class Solution {
public:
   // DFS Helper Function
   void dfsHelper(int node, vector<vector<int>>& adj, vector<int>& visited, vector<int>& result) {
      visited[node] = 1;
      result.push_back(node);

      for (int neighbor : adj[node]) {
         if (!visited[neighbor]) {
            dfsHelper(neighbor, adj, visited, result);
         }
      }
   }
   // Main DFS Function
   vector<int> dfs(int n, vector<vector<int>>& adj) {
      vector<int> visited(n, 0);
      vector<int> result;
      int startNode = 0;

      dfsHelper(startNode, adj, visited, result);
      return result;
   }
};

// Main function to test DFS
int main() {
   Solution sol;

   int n = 5; // Number of nodes
   vector<vector<int>> edges = {
      {0, 1}, {0, 2}, {1, 3}, {1, 4}
```

```
    };

    // Build the adjacency list
    vector<vector<int>> adj(n);
    for (auto edge : edges) {
        adj[edge[0]].push_back(edge[1]);
        adj[edge[1]].push_back(edge[0]); // Remove for directed graph
    }

    // Call DFS
    vector<int> dfsResult = sol.dfs(n, adj);

    // Print DFS Traversal
    cout << "DFS Traversal starting from node 0: ";
    for (int node : dfsResult) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

BFS

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

// Function to build adjacency list from edge list
vector<vector<int>> buildAdjacencyList(int n, const vector<vector<int>>& edges) {
    vector<vector<int>> adj(n);  // using 0-based indexing
```

```
    for (const auto& edge : edges) {
        int u = edge[0];
        int v = edge[1];
        adj[u].push_back(v);
        adj[v].push_back(u); // For undirected graph. Remove for directed.
    }

    return adj;
}

// Function to perform BFS traversal from a given starting node
vector<int> bfsTraversal(int startNode, const vector<vector<int>>& adj, int n) {
    vector<int> visited(n, 0);
    vector<int> bfsOrder;
    queue<int> q;

    q.push(startNode);
    visited[startNode] = 1;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        bfsOrder.push_back(node);

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = 1;
                q.push(neighbor);
            }
        }
    }
```

```
    return bfsOrder;
}

// Main function
int main() {
    int n = 6;  // Number of nodes
    vector<vector<int>> edges = {
        {0, 1}, {0, 2}, {1, 3}, {2, 4}, {4, 5}
    };

    // Build the adjacency list
    vector<vector<int>> adj = buildAdjacencyList(n, edges);

    // Perform BFS starting from node 0
    vector<int> bfs = bfsTraversal(0, adj, n);

    // Print BFS result
    cout << "BFS Traversal starting from node 0: ";
    for (int node : bfs) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```