Faridpur Engineering College

Department of Computer Science and Engineering

Design and Analysis of Algorithms – 1 Lab

Course Code: CSE-2212

Submitted To:

Samia Akter

Lecturer

Department of Computer Science and Engineering

Submitted By:

Asif Shariar Tashin

Session: 2021-2022

Reg. no: **908**

Submission Date: 06/02/2025

Serial No	Program List
1	Implement BFS
2	Implement DFS
3	Implement Strongly Connected Component
4	Implement Articulation Point
5	Implement Dijkstra's Algorithm
6	Implement Prim's Algorithm
7	Implement 0/1 Knapsack Problem
8	Implement Kruskal's Algorithm

Experiment No: 1

Experiment Name: Implement BFS

Theory: BFS is a graph traversal algorithm that explores all nodes level by level using a **queue (FIFO)**. It finds the shortest path in an **unweighted graph** and ensures all reachable nodes are visited.

BFS Algorithm

1. **Enqueue** the starting node and mark it as visited.
2. While the queue is **not empty**:
   - Dequeue a node and process it.
   - Enqueue all its **unvisited** neighbors and mark them as visited.

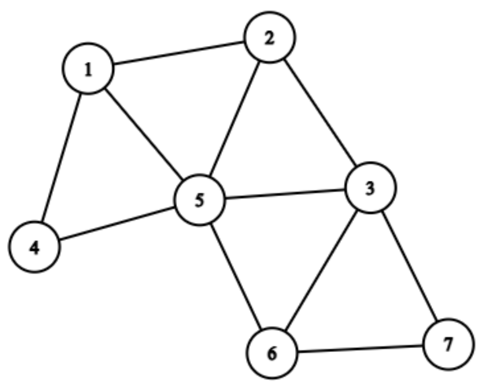
Code:

bfs.cpp > ...

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

int main() {
int nodes, edges;
cout<<"Enter the number of Nodes: ";
cin>>nodes;
cout<<"Enter the number of Edges: ";
cin>>edges;
int graph[100][100] = {0};
cout<<"enter edges: ";
for (int i = 0; i < edges; ++i) {
    int u, v;
    cin>>u>>v;
    graph[u][v] = 1;
    graph[v][u] = 1;
}
int start;
cout<<"Enter starting node: ";
cin>>start;
bool visited[100] = {false};
queue <int> q;
q.push(start);
```

```cpp
visited[start] = true;
cout<<"BFS traversal: ";
while (!q.empty()) {
    int node = q.front();
    q.pop();
    cout<<node<<" ";
    for (int i = 1; i <= nodes; ++i) {
        if (graph[node][i] == 1 && !visited[i]) {
            visited[i] = true;
            q.push(i);
        }
    }

}
cout<<endl;
}
```

Graph:



Input and Output:

```
PS E:\c++> cd "e:\c++\" ; if ($?) { g++ bfs.cpp -o bfs } ; if ($?) { .\bfs }
Enter the number of Nodes: 7
Enter the number of Edges: 11
enter edges: 1 2
2 3
1 4
1 5
2 5
4 5
5 3
5 6
6 3
3 7
6 7
Enter starting node: 1
BFS traversal: 1 2 4 5 3 6 7
```

Experiment No: 02

Experiment Name: Implement DFS

Theory: DFS is a graph traversal algorithm that explores as far as possible along one branch before backtracking. It uses a **stack (LIFO)** (either explicitly or via recursion). DFS is useful for cycle detection, pathfinding, and tree traversals.

DFS Algorithm

1. **Push** the starting node onto a stack (or use recursion).
2. While the stack is **not empty**:
    - Pop a node and process it.
    - Push all its **unvisited** neighbors onto the stack and mark them as visited.

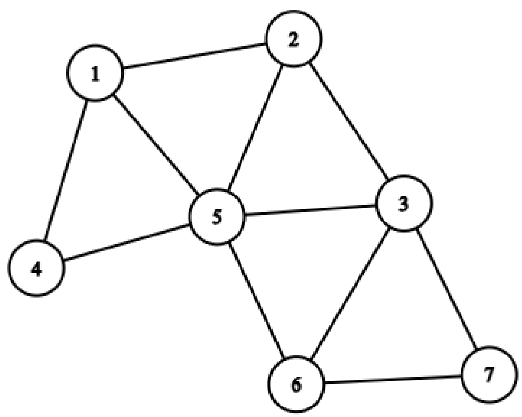
Code:

dfs.cpp > ...

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

int main() {
int nodes, edges;
cout<<"Enter the number of Nodes: ";
cin>>nodes;
cout<<"Enter the number of Edges: ";
cin>>edges;
int graph[100][100] = {0};
cout<<"enter edges: ";
for (int i = 0; i < edges; ++i) {
    int u, v;
    cin>>u>>v;
    graph[u][v] = 1;
    graph[v][u] = 1;
}
int start;
cout<<"Enter starting node: ";
cin>>start;

bool visited[100] = {false};
stack <int> s;
s.push(start);
visited[start] = true;
cout<<"DFS traversal: ";
while (!s.empty()) {
```

```cpp
        int node = s.top();
        s.pop();
        cout<<node<<" ";
        for (int i = 1; i <= nodes; ++i) {
            if (graph[node][i] == 1 && !visited[i]) {
                visited[i] = true;
                s.push(i);
            }
        }

    }
    cout<<endl;
}
```

Graph:



Input and Output:

```
PS E:\c++> cd "e:\c++\" ; if ($?) { g++ dfs.cpp -o dfs } ; if ($?) { .\dfs }
Enter the number of Nodes: 7
Enter the number of Edges: 11
enter edges: 1 2
2 3
1 4
1 5
2 5
4 5
5 3
5 6
6 3
3 7
6 7
Enter starting node: 1
DFS traversal: 1 5 6 7 3 4 2
```

Experiment No: 3

Experiment Name: Implement Strongly Connected Component

Theory: A **Strongly Connected Component (SCC)** of a **directed graph** is a maximal subgraph where every node is **reachable from every other node** in that subgraph. **Kosaraju's Algorithm** is a popular method to find SCCs using **two DFS traversals**.

Kosaraju's Algorithm for SCC

1. **Run DFS** on the graph and store nodes in a stack based on **finishing time**.
2. **Reverse** the graph (transpose).
3. **Process nodes** from the stack using DFS on the transposed graph to find SCCs.

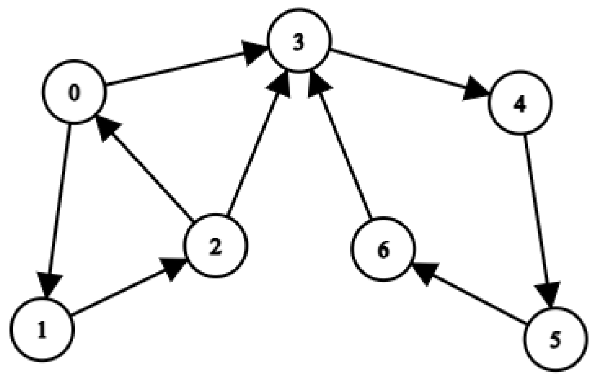
Code:

new2.cpp > ...

```cpp
#include <iostream>
#include <stack>
using namespace std;
const int MAX = 100;
int adj[MAX][MAX];
int adjT[MAX][MAX];
int visited[MAX];
stack<int> st;
int n, e;
void dfs(int u) {
    visited[u] = 1;
    for (int v = 0; v < n; v++) {
        if (adj[u][v] && !visited[v]) {
            dfs(v);
        }
    }
    st.push(u);
}
void dfsTranspose(int u) {
    visited[u] = 1;
    cout << u << " ";
    for (int v = 0; v < n; v++) {
        if (adjT[u][v] && !visited[v]) {
            dfsTranspose(v);
        }
    }
}
```

```cpp
int main() {
    cout << "Enter number of nodes and edges: ";
    cin >> n >> e;
    cout << "Enter edges (u v):" << endl;
    for (int i = 0; i < e; i++) {
        int u, v;
        cin >> u >> v;
        adj[u][v] = 1;
        adjT[v][u] = 1;
    }
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }
    int scc = 0;
    cout << "Strongly Connected Components:" << endl;
    while (!st.empty()) {
        int u = st.top();
        st.pop();
        if (!visited[u]) {
            dfsTranspose(u);
            scc++;
            cout << endl;
        }
    }
    cout << "Number of SCCs: " << scc << endl;
    return 0;
}
```

Graph:



Input and Output:

```
PS E:\c++> cd "e:\c++\" ; if ($?) { g++ new2.cpp -o new2 } ; if ($?) { .\new2 }
Enter number of nodes and edges: 7 9
Enter edges (u v):
0 1
1 2
2 0
0 3
2 3
6 3
3 4
4 5
5 6
Strongly Connected Components:
0 2 1
3 6 5 4
Number of SCCs: 2
```

Experiment No: 04

Experiment Name: Implement Articulation Point

Theory: An **Articulation Point** (or **Cut Vertex**) in a graph is a vertex that, when removed, increases the number of connected components. It is crucial for network reliability.

Algorithm (Using DFS):

1. Use **DFS** to assign **discovery time (`disc[]`)** and **lowest reachable node (`low[]`)**.
2. For each vertex `u`, explore its neighbors `v`:
   - If `v` is unvisited, recursively DFS and update `low[u]`.
   - If `low[v] ≥ disc[u]`, `u` is an articulation point.
   - If `u` is the root and has **two or more children**, it is an articulation point.
3. Repeat for all nodes.

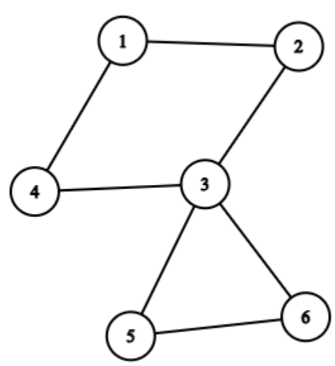
Code:

articulationPoint.cpp > ...

```cpp
#include<iostream>
#include<cstring>
#define MAX 100
using namespace std;

int numNodes, adj[MAX][MAX], parent[MAX], low[MAX], dis[MAX], timecount;
bool isArticulation[MAX], visited[MAX];

void DFS (int node) {
    visited[node] = true;
    dis[node] = low[node] = ++timecount;
    int child = 0;

    for (int i = 0; i < numNodes; i++) {
        if (adj[node][i]) {
            if (!visited[i]) {
                parent[i] = node;
                child++;
                DFS(i);
                low[node] = min(low[node], low[i]);

        if ((parent[node] == -1 && child > 1) || (parent[node] != -1 && low[i] >= dis[node])) {
                    isArticulation[node] = true;
                }
            }
            else if (i != parent[node]) {
                low[node] = min (low[node], dis[i]);
            }
```

```
        }
    }
}
int main() {
    int numEdges, node1, node2;
    cout<<"Enter the number of Nodes: ";
    cin >> numNodes;
    cout<<"Enter the number of Edges: ";
    cin>> numEdges;
    memset(adj, 0, sizeof(adj));
    memset(parent, -1, sizeof(parent));

    while (numEdges--) {
        cout<<"Enter edges: ";
        cin >> node1 >> node2;
        adj[node1][node2] = adj[node2][node1] = 1;
    }

    for (int i = 0; i < numNodes; i++) if (!visited[i]) DFS(i);
    for (int i = 0; i < numNodes; i++) if (isArticulation[i]) cout <<"Articulation point: "<< i << " ";
}
```

Graph:



Input and Output:

```
PS E:\c++> cd "e:\c++\" ; if ($?) { g++ articulationPoint.cpp -o articulationPoint } ; if ($?) { .\articulationPoint }
Enter the number of Nodes: 6
Enter the number of Edges: 7
Enter edges: 1 4
Enter edges: 1 2
Enter edges: 4 3
Enter edges: 2 3
Enter edges: 3 5
Enter edges: 3 6
Enter edges: 5 6
Articulation point: 3
```

Experiment No: 05

Experiment Name: Implement Dijkstra's Algorithm

Theory: Dijkstra's Algorithm finds the **shortest path** from a **single source** to all other vertices in a **graph with non-negative weights**. It uses a **priority queue** to always expand the nearest unvisited node first.

Algorithm:

1. **Initialize** distances as ∞ for all nodes except the source (0).
2. **Use a priority queue** to pick the node with the smallest distance.
3. **Update** distances of its neighbors if a shorter path is found.
4. **Repeat** until all nodes are processed.

Time Complexity: O((V + E) log V) using a priority queue.

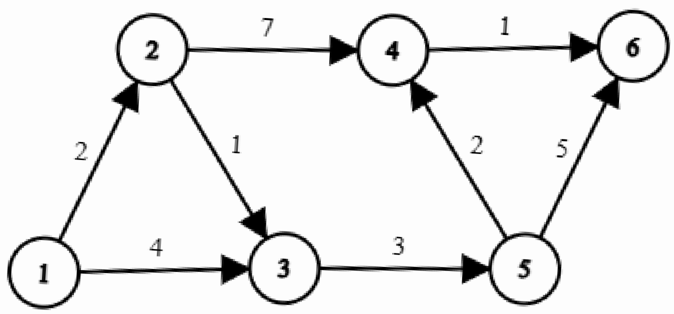
Code:

dijkastra.cpp > ...

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

int main() {
    int numNodes, numEdges, startNode;
    cout << "Enter number of nodes: ";
    cin >> numNodes;
    cout << "Enter number of edges: ";
    cin >> numEdges;
    cout << "Enter starting node: ";
    cin >> startNode;

    startNode--;

    int adjacencyMatrix[100][100] = {0};

    for (int i = 0; i < numEdges; ++i) {
        int sourceNode, destinationNode, edgeWeight;
        cout << "Enter source, destination, weight: ";
        cin >> sourceNode >> destinationNode >> edgeWeight;

        adjacencyMatrix[sourceNode - 1][destinationNode - 1] = edgeWeight;
    }

    int shortestDistance[100];
    for (int i = 0; i < numNodes; ++i)
```

```
        shortestDistance[i] = INT_MAX;
    shortestDistance[startNode] = 0;

    bool visited[100] = {false};

    for (int processedNodes = 0; processedNodes < numNodes; ++processedNodes) {
        int currentNode = -1, minDistance = INT_MAX;
        for (int i = 0; i < numNodes; ++i) {
            if (!visited[i] && shortestDistance[i] < minDistance) {
                currentNode = i;
                minDistance = shortestDistance[i];
            }
        }

        if (currentNode == -1) break;
        visited[currentNode] = true;

        for (int neighborNode = 0; neighborNode < numNodes; ++neighborNode) {
             if (adjacencyMatrix[currentNode][neighborNode] != 0 &&
             shortestDistance[currentNode] + adjacencyMatrix[currentNode][neighborNode] <
             shortestDistance[neighborNode]) {
            shortestDistance[neighborNode] = shortestDistance[currentNode] + adjacencyMatrix[currentNode][neighborNode];
            }
        }

    }

    int totalWeight = 0;
    cout << "Shortest distance of nodes: ";
    for (int i = 0; i < numNodes; ++i) {
        if (shortestDistance[i] == INT_MAX) {
            cout << "-1 ";
        } else {
            cout << shortestDistance[i] << " ";
            totalWeight += shortestDistance[i];
        }
    }
    cout << endl;

    cout << "Total weight of shortest paths: " << totalWeight << endl;

    return 0;
}
```

Graph:



Input and Output:

```
PS E:\c++> cd "e:\c++\" ; if ($?) { g++ dijkastra.cpp -o dijkastra } ; if ($?) { .\dijkastra }
Enter number of nodes: 6
Enter number of edges: 8
Enter starting node: 1
Enter source, destination, weight: 1 2 2
Enter source, destination, weight: 2 4 7
Enter source, destination, weight: 1 3 4
Enter source, destination, weight: 3 5 3
Enter source, destination, weight: 5 6 5
Enter source, destination, weight: 2 3 1
Enter source, destination, weight: 5 4 2
Enter source, destination, weight: 4 6 1
Shortest distance of nodes: 0 2 3 8 6 9
Total weight of shortest paths: 28
```

Experiment No: 06

Experiment Name: Implement Prim's Algorithm

Theory: Prim's Algorithm finds the **Minimum Spanning Tree (MST)** of a **connected, weighted, undirected graph**. It grows the MST **one edge at a time**, always adding the **smallest edge** that connects a new vertex to the tree.

Algorithm:

1. **Initialize** an empty MST and start from any node.
2. **Use a priority queue** to select the **smallest edge** connecting the MST to an unvisited node.
3. **Add the selected edge** to the MST and mark the node as visited.
4. **Repeat** until all nodes are included.

Time Complexity: O((V + E) log V) using a priority queue.

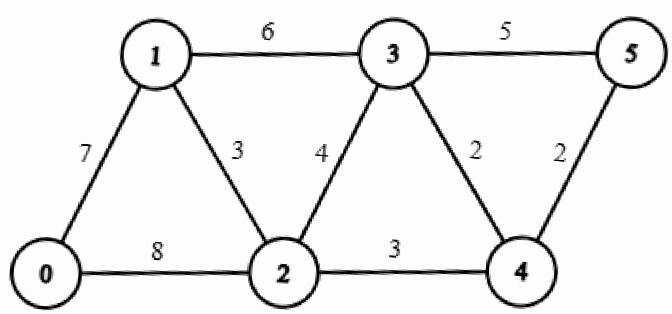
Code:

prims.cpp > ...

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

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

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

    int key[vertices], parent[vertices], inMST[vertices];
    int totalWeight = 0;

    for (int i = 0; i < vertices; i++) {
        key[i] = INT_MAX;
        parent[i] = -1;
        inMST[i] = 0;
    }

    key[0] = 0;

    for (int count = 0; count < vertices - 1; count++) {
        int minKey = INT_MAX, u;
```

```
        for (int v = 0; v < vertices; v++) {
            if (!inMST[v] && key[v] < minKey) {
                minKey = key[v];
                u = v;
            }
        }

        inMST[u] = 1;

        for (int v = 0; v < vertices; v++) {
            if (graph[u][v] && !inMST[v] && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
                parent[v] = u;
            }
        }
    }
    cout << "Edge   Weight\n";
    for (int i = 1; i < vertices; i++) {
        cout << parent[i] << " - " << i << "    " << graph[i][parent[i]] << endl;
        totalWeight += graph[i][parent[i]];
    }
    cout << "Total weight of MST: " << totalWeight << endl;
    return 0;
}
```

Graph:



Input and Output:

```
PS E:\c++> cd "e:\c++\" ; if ($?) { g++ prims.cpp -o prims } ; if ($?) { .\prims }
Enter number of vertices: 6
Enter adjacency matrix:
0 7 8 0 0 0
7 0 3 6 0 0
8 3 0 4 3 0
0 6 4 0 2 5
0 0 3 2 0 2
0 0 0 5 2 0
Edge   Weight
0 - 1   7
1 - 2   3
4 - 3   2
2 - 4   3
4 - 5   2
Total weight of MST: 17
```

Experiment No: 07

Experiment Name: Implement 0/1 Knapsack Problem

Theory: The **0/1 Knapsack Problem** involves selecting items with given weights and values to maximize the total value without exceeding a weight limit. Each item can be included (1) or excluded (0). The goal is to find the optimal combination of items within the weight capacity.

It's a classic dynamic programming problem used to illustrate optimization techniques.

Code:

```cpp
#include <iostream>
using namespace std;
int max(int a, int b) { return (a > b) ? a : b; }
int knapsack(int maxCapacity, int weights[], int values[], int itemCount) {
    int dp[100][100];

    for (int i = 0; i <= itemCount; i++) {
        for (int w = 0; w <= maxCapacity; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (weights[i - 1] <= w)
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    return dp[itemCount][maxCapacity];
}
int main() {
    int itemCount, maxCapacity;
    cout << "Enter the number of items: ";
    cin >> itemCount;
    int weights[100], values[100];

    cout << "Enter the weights of the items: ";
    for (int i = 0; i < itemCount; i++) {
        cin >> weights[i];
```

```cpp
    }

    cout << "Enter the values of the items: ";
    for (int i = 0; i < itemCount; i++) {
        cin >> values[i];
    }

    cout << "Enter the maximum weight capacity of the knapsack: ";
    cin >> maxCapacity;

    cout << "Maximum value in Knapsack = " << knapsack(maxCapacity, weights, values, itemCount) << endl;
    return 0;
}
```

Input and Output:

```
PS E:\c++> cd "e:\c++\" ; if ($?) { g++ new2.cpp -o new2 } ; if ($?) { .\new2 }
Enter the number of items: 4
Enter the weights of the items: 3 2 5 4
Enter the values of the items: 4 3 6 5
Enter the maximum weight capacity of the knapsack: 5
Maximum value in Knapsack = 7
```

Experiment No: 08

Experiment Name: Implement Kruskal's Algorithm

Theory: Kruskal's Algorithm is used to find the Minimum Spanning Tree (MST) of a graph. It sorts all edges by weight and adds the shortest edge to the MST if it doesn't form a cycle. This process continues until the MST spans all vertices, ensuring the total edge weight is minimized.

It works by:

1. Sorting all edges by weight.
2. Adding the shortest edge to the MST if it doesn't form a cycle.
3. Repeating step 2 until all vertices are included in the MST.

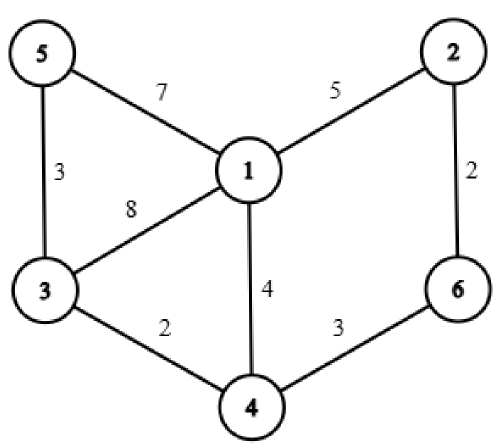
Code:

kruskal.cpp > ...

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

int find (int parent[], int vertex) {
    if (parent[vertex] != vertex) {
        parent[vertex] = find(parent, parent[vertex]);
    }
    return parent[vertex];
}

int main () {
int vertices, edges, start, end, weight, totalweight = 0;
cout<<"Enter the number of vertices: ";
cin>>vertices;
cout<<"Enter the number of edges: ";
cin>>edges;

int startNode[edges], endNode[edges], w[edges];
cout<<"Enter start,end,weight: "<<endl;
for (int i = 0; i < edges; i++) {
    cin>>start>>end>>weight;
    startNode[i] = start-1;
    endNode[i] = end-1;
    w[i] = weight;
}
for (int i = 0; i < edges-1; i++) {
    for (int j = 0; j < edges-1-i; j++) {
```

```
            if (w[j] > w[j+1]) {
                swap(w[j], w[j+1]);
                swap(startNode[j], startNode[j+1]);
                swap(endNode[j], endNode[j+1]);
            }
        }
}
int parent[vertices];
for (int i = 0; i < vertices; i++) {
    parent[i] = i;
}
cout<<"The MST: "<<endl;
for (int i = 0; i < edges; i++) {
    int rootStart = find (parent, startNode[i]);
    int rootEnd  = find (parent, endNode[i]);

    if (rootStart != rootEnd) {
        totalweight += w[i];
        parent[rootStart] = rootEnd;
        cout<<startNode[i]+1<<"->"<<endNode[i]+1<<" "<<w[i]<<" "<<endl;
    }

    }
    cout<<"Total Weight: "<<totalweight<<endl;
}
```

Graph:



Input and Output:

```
PS E:\c++> cd "e:\c++\" ; if ($?) { g++ kruskal.cpp -o kruskal } ; if ($?) { .\kruskal }
Enter the number of vertices: 6
Enter the number of edges: 8
Enter start,end,weight:
1 5 7
1 3 8
3 5 3
1 2 5
2 6 2
4 6 3
3 4 2
1 4 4
The MST:
2->6 2
3->4 2
3->5 3
4->6 3
1->4 4
Total Weight: 14
```