DAY 7

NAME: -Shivam Yadav

UID: -22BCS15259

Date: -27/12/2024

Section: - 620 A

Question 1: -

WAP to find the degree of given vertex in a graph

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

class Graph {
public:
    int V;
    vector<vector<int>> adjList;
    Graph(int vertices) {
        V = vertices;
        adjList.resize(V);
    }
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }
```

```
    int getDegree(int vertex) {
        return adjList[vertex].size();
    }
};

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

    cout << "Enter number of edges: ";
    cin >> edges;

    cout << "Enter edges (u v): \n";
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    int vertex;
    cout << "Enter vertex to find its degree: ";
    cin >> vertex;
```

```
    cout << "Degree of vertex " << vertex << " is: " <<
g.getDegree(vertex) << endl;


    return 0;
}
```

OUTPUT: -

```
Enter number of vertices: 5
Enter number of edges: 4
Enter edges (u v):
0 1
1 2
2 3
3 4
Enter vertex to find its degree: 2
Degree of vertex 2 is: 2
```

Question 2: -

WAP for DFS

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

class Graph {
public:
    int V;
    vector<vector<int>> adjList;
```

```
Graph(int vertices) {
    V = vertices;
    adjList.resize(V);
}

void addEdge(int u, int v) {
    adjList[u].push_back(v);
    adjList[v].push_back(u);
}
void DFS(int start) {
    vector<bool> visited(V, false);
    stack<int> s;
    s.push(start);
    visited[start] = true;

    while (!s.empty()) {
        int node = s.top();
        s.pop();
        cout << node << " ";
        for (int adj : adjList[node]) {
            if (!visited[adj]) {
                visited[adj] = true;
                s.push(adj);
            }
```

```
            }
        }
        cout << endl;
    }
};

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

    cout << "Enter number of edges: ";
    cin >> edges;

    cout << "Enter edges (u v): \n";
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    int start;
    cout << "Enter starting vertex for DFS: ";
```

```
    cin >> start;

    cout << "DFS traversal starting from vertex " << start << ": ";
    g.DFS(start);

    return 0;
}
```

OUTPUT: -

```
Enter number of vertices: 5
Enter number of edges: 4
Enter edges (u v):
0 1
1 2
2 3
3 4
Enter starting vertex for DFS: 0
DFS traversal starting from vertex 0: 0 1 2 3 4
```

Question 3: -

WAP to detect a cycle in undirected graph

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

class Graph {
public:
    int V;
    vector<vector<int>> adjList;
```

```
Graph(int vertices) {
    V = vertices;
    adjList.resize(V);
}
void addEdge(int u, int v) {
    adjList[u].push_back(v);
    adjList[v].push_back(u);
}
bool DFS(int node, vector<bool>& visited, vector<int>& parent) {
    visited[node] = true;
    for (int adj : adjList[node]) {
        if (!visited[adj]) {
            parent[adj] = node;
            if (DFS(adj, visited, parent))
                return true;
        }
        else if (parent[node] != adj) {
            return true;
        }
    }
    return false;
}
bool detectCycle() {
    vector<bool> visited(V, false);
```

```
        vector<int> parent(V, -1);
        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                if (DFS(i, visited, parent))
                    return true;
            }
        }
        return false;
    }
};

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

    cout << "Enter number of edges: ";
    cin >> edges;

    cout << "Enter edges (u v): \n";
    for (int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
```

```
        g.addEdge(u, v);
    }

    if (g.detectCycle()) {
        cout << "Graph contains a cycle." << endl;
    } else {
        cout << "Graph does not contain any cycle." << endl;
    }

    return 0;
}
```

OUTPUT: -

```
Enter number of vertices: 5
Enter number of edges: 4
Enter edges (u v):
0 1
1 2
2 3
3 4
Graph does not contain any cycle.
```

Question 4: -

Given the root of complete binary tree return the number of nodes in the tree

```
#include <iostream>

using namespace std;

struct TreeNode {
```

```
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    int getHeight(TreeNode* root) {
        int height = 0;
        while (root) {
            height++;
            root = root->left;
        }
        return height;
    }
    int countNodes(TreeNode* root) {
        if (!root) return 0;

        int height = getHeight(root);
        if (height == 0) return 0;

        int left = 1, right = (1 << height) - 1;
        while (left < right) {
```

```
            int mid = (left + right + 1) / 2;
            if (exists(root, height, mid)) {
                left = mid;
            } else {
                right = mid - 1;
            }
        }

        return left + (1 << (height - 1)) - 1;
    }

private:
    bool exists(TreeNode* root, int height, int index) {
        int left = 0, right = (1 << height) - 1;
        for (int i = 0; i < height - 1; i++) {
            int mid = (left + right) / 2;
            if (index <= mid) {
                root = root->left;
                right = mid;
            } else {
                root = root->right;
                left = mid + 1;
            }
        }
```

```
        return root != nullptr;
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);

    Solution solution;
    cout << "Number of nodes in the tree: " <<
solution.countNodes(root) << endl;

    return 0;
}
```

OUTPUT: -

```
Number of nodes in the tree: 8
```

Question 5: -

A binary tree find the max depth of binary tree

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

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        return 1 + max(leftDepth, rightDepth);
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
```

```
    root->left->right = new TreeNode(5);


    Solution solution;
    cout << "Maximum depth of the binary tree: " << solution.maxDepth(root) << endl;


    return 0;
}
```

OUTPUT: -

```
Maximum depth of the binary tree: 3
```

Question 6: -

Given the root of binary tree return preorder traverse of its node value

```
#include <iostream>
#include <vector>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
```

```
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        preorderHelper(root, result);
        return result;
    }

private:
    void preorderHelper(TreeNode* node, vector<int>& result) {
        if (node == nullptr) {
            return;
        }
        result.push_back(node->val);
        preorderHelper(node->left, result);
        preorderHelper(node->right, result);
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
```

```
    Solution solution;
    vector<int> result = solution.preorderTraversal(root);
    cout << "Preorder traversal: ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

OUTPUT: -

```
Preorder traversal: 1 2 4 5 3
```

Question 7: -

given a binary tree the task is to count the leaf node A node is a leaf node is both the leaf and right value are null

```
#include <iostream>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
```

```
};

class Solution {
public:
    int countLeafNodes(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        if (root->left == nullptr && root->right == nullptr) {
            return 1;
        }
        return countLeafNodes(root->left) + countLeafNodes(root->right);
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);

    Solution solution;
```

```
    int leafCount = solution.countLeafNodes(root);
    cout << "Number of leaf nodes: " << leafCount << endl;

    return 0;
}
```

OUTPUT: -

```
Number of leaf nodes: 3
```

Question 8: -

implementation off cyclic graph

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

class Graph {
private:
    unordered_map<int, vector<int>> adjList;

public:
    void addEdge(int src, int dest) {
        adjList[src].push_back(dest);
    }
    bool detectCycleUtil(int node, unordered_map<int, int>& visited) {
```

```
        if (visited[node] == 1) {
            return true;
        }
        visited[node] = 1;
        for (int neighbor : adjList[node]) {
            if (visited[neighbor] != 2 && detectCycleUtil(neighbor, visited))
{
                return true;
            }
        }
        visited[node] = 2;
        return false;
    }
    bool detectCycle() {
        unordered_map<int, int> visited;
        for (const auto& pair : adjList) {
            if (visited[pair.first] == 0) {
                if (detectCycleUtil(pair.first, visited)) {
                    return true;
                }
            }
        }
        return false;
    }
```

```
};

int main() {
    Graph g;
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2, 0);

    if (g.detectCycle()) {
        cout << "Cycle detected in the graph!" << endl;
    } else {
        cout << "No cycle detected in the graph." << endl;
    }

    return 0;
}
```

OUTPUT: -

```
Cycle detected in the graph!
```

Question 9: -

find the centre of the star graph

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

```
int findCenter(vector<vector<int>>& adjList) {

    int n = adjList.size();
    for (int i = 0; i < n; i++) {
        if (adjList[i].size() == n - 1) {
            return i;
        }
    }

    return -1
}

int main() {
    int n = 5;
    vector<vector<int>> adjList(n);
    adjList[0] = {1, 2, 3, 4};
    adjList[1] = {0};
    adjList[2] = {0};
    adjList[3] = {0};
    adjList[4] = {0};

    int center = findCenter(adjList);
    cout << "The center of the star graph is node: " << center << endl;
```

```
    return 0;
}
```

OUTPUT: -

```
The center of the star graph is node: 0
```

Question 10: -

Write a program to detect a cycle in a directed graph by using DFS

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

class Graph {
public:
    int V;
    vector<vector<int>> adj;

    Graph(int V);
    void addEdge(int u, int v);
    bool dfs(int node, vector<bool>& visited, vector<bool>&
recursionStack);
    bool hasCycle();
};
Graph::Graph(int V) {
    this->V = V;
```

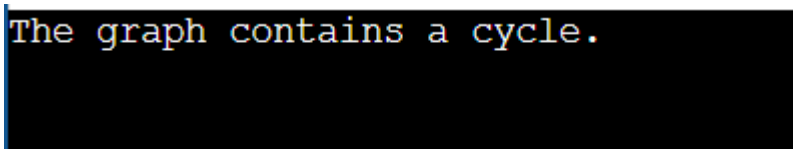
```
    adj.resize(V);
}
void Graph::addEdge(int u, int v) {
    adj[u].push_back(v);
}
bool Graph::dfs(int node, vector<bool>& visited, vector<bool>&
recursionStack) {
    visited[node] = true;
    recursionStack[node] = true;
    for (int neighbor : adj[node]) {
        if (recursionStack[neighbor]) {
            return true;
        }
        if (!visited[neighbor] && dfs(neighbor, visited, recursionStack)) {
            return true;
        }
    }
    recursionStack[node] = false;
    return false;
}
bool Graph::hasCycle() {
    vector<bool> visited(V, false);
    vector<bool> recursionStack(V, false);
    for (int i = 0; i < V; i++) {
```

```
        if (!visited[i]) {
            if (dfs(i, visited, recursionStack)) {
                return true;
            }
        }
    }
    return false;
}

int main() {
    int V = 4;
    Graph g(V);
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2, 3);
    g.addEdge(3, 1);
    if (g.hasCycle()) {
        cout << "The graph contains a cycle." << endl;
    } else {
        cout << "The graph does not contain a cycle." << endl;
    }

    return 0;
}
```

OUTPUT: -



Question 11: -

WAP to find the minimum spanning tree

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

class Graph {
public:
    int V;
    vector<vector<pair<int, int>>> adj;

    Graph(int V);
    void addEdge(int u, int v, int w);
    int primMST();
};
Graph::Graph(int V) {
    this->V = V;
    adj.resize(V);
}
```

```
void Graph::addEdge(int u, int v, int w) {
    adj[u].push_back({v, w});
    adj[v].push_back({u, w});
}
int Graph::primMST() {
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;

    vector<bool> inMST(V, false);

    vector<int> key(V, INT_MAX);

    pq.push({0, 0});
    key[0] = 0;

    int totalWeight = 0;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        if (inMST[u]) continue;
        inMST[u] = true;
        totalWeight += key[u];
        for (auto& neighbor : adj[u]) {
```

```
            int v = neighbor.first;
            int weight = neighbor.second;
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                pq.push({key[v], v});
            }
        }
    }

    return totalWeight;
}

int main() {
    int V = 5;
    Graph g(V);
    g.addEdge(0, 1, 2);
    g.addEdge(0, 3, 6);
    g.addEdge(1, 2, 3);
    g.addEdge(1, 3, 8);
    g.addEdge(1, 4, 5);
    g.addEdge(2, 4, 7);
    int mstWeight = g.primMST();
    cout << "The weight of the Minimum Spanning Tree is: " <<
mstWeight << endl;
```

```
    return 0;
}
```

OUTPUT: -

```
The weight of the Minimum Spanning Tree is: 16
```

Question 12

Write a program to count the number of connected component in undirected graph

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

class Graph {
public:
    int V;
    vector<vector<int>> adj;

    Graph(int V);
    void addEdge(int u, int v);
    void dfs(int node, vector<bool>& visited);
    int countConnectedComponents();
};
```

```
Graph::Graph(int V) {
    this->V = V;
    adj.resize(V);
}
void Graph::addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}
void Graph::dfs(int node, vector<bool>& visited) {

    stack<int> s;
    s.push(node);
    visited[node] = true;
    while (!s.empty()) {
        int current = s.top();
        s.pop();
        for (int neighbor : adj[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                s.push(neighbor);
            }
        }
    }
}
```

```
int Graph::countConnectedComponents() {
    vector<bool> visited(V, false);
    int connectedComponents = 0;
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            dfs(i, visited);
            connectedComponents++;
        }
    }

    return connectedComponents;
}

int main() {
    int V = 5;
    Graph g(V);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(3, 4);
    int numComponents = g.countConnectedComponents();
    cout << "The number of connected components in the graph is: "
<< numComponents << endl;

    return 0;
```

}

OUTPUT: -

```
The number of connected components in the graph is: 2
```

Question 13: -

Write a program to solve traveling sales man problem

```
#include <iostream>
#include <vector>
#include <climits>
#include <cstring>
using namespace std;
#define MAX 16
int dp[1 << MAX][MAX];
int dist[MAX][MAX];
int tsp(int mask, int pos, int n) {
    if (mask == (1 << n) - 1) {
        return dist[pos][0];
    }
    if (dp[mask][pos] != -1) {
        return dp[mask][pos];
    }
    int ans = INT_MAX;
    for (int city = 0; city < n; city++) {
        if ((mask & (1 << city)) == 0) {
```

```
            int newAns = dist[pos][city] + tsp(mask | (1 << city), city, n);
            ans = min(ans, newAns);
        }
    }
    return dp[mask][pos] = ans;
}
int main() {
    int n;
    cout << "Enter the number of cities: ";
    cin >> n;
    cout << "Enter the distance matrix (n x n):" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> dist[i][j];
        }
    }
    memset(dp, -1, sizeof(dp));
    int result = tsp(1, 0, n);
    cout << "The minimum cost of visiting all cities and returning to the
starting point is: " << result << endl;
    return 0;
}
```

OUTPUT: -

```
Enter the number of cities: 4
Enter the distance matrix (n x n):
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
The minimum cost of visiting all cities and returning to the starting point is: 80
```

QUESTION 14: -

Find the diameter of an undirected graph use DFS

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

class Graph {
public:
    int V;
    vector<vector<int>> adj;
    Graph(int V);
    void addEdge(int u, int v);
    void dfs(int node, vector<bool>& visited, int dist, int&
farthestNode, int& maxDist);
    int findDiameter();
};
Graph::Graph(int V) {
    this->V = V;
```

```
    adj.resize(V);
}
void Graph::addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}
void Graph::dfs(int node, vector<bool>& visited, int dist, int&
farthestNode, int& maxDist) {
    visited[node] = true;
    if (dist > maxDist) {
        maxDist = dist;
        farthestNode = node;
    }
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, visited, dist + 1, farthestNode, maxDist);
        }
    }
}
int Graph::findDiameter() {
    vector<bool> visited(V, false);
    int farthestNode = 0, maxDist = 0;
    dfs(0, visited, 0, farthestNode, maxDist)
    fill(visited.begin(), visited.end(), false);
```

```
    int newFarthestNode = farthestNode;
    maxDist = 0;
    dfs(newFarthestNode, visited, 0, farthestNode, maxDist);

    return maxDist;
}

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    Graph g(V);

    cout << "Enter the edges (u v) format for undirected graph:" <<
endl;
    for (int i = 0; i < E; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);

    int diameter = g.findDiameter();
```

```
    cout << "The diameter of the graph is: " << diameter << endl;


    return 0;
}
```

OUTPUT: -