**Assignment 1**

**Code-**

1] BFS –

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// Data structure to store a graph edge

struct Edge {

    int src, dest;

};

// A class to represent a graph object

class Graph

{

public:

    // a vector of vectors to represent an adjacency list

    vector<vector<int>> adjList;

    // Graph Constructor

    Graph(vector<Edge> const &edges, int n)

    {

        // resize the vector to hold `n` elements of type `vector<int>`

        adjList.resize(n);

        // add edges to the undirected graph

        for (auto &edge: edges)

        {

            adjList[edge.src].push\_back(edge.dest);

            adjList[edge.dest].push\_back(edge.src);

        }

    }

};

// Perform BFS recursively on the graph

void recursiveBFS(Graph const &graph, queue<int> &q, vector<bool> &discovered)

{

    if (q.empty()) {

        return;

    }

    // dequeue front node and print it

    int v = q.front();

    q.pop();

    cout << v << " ";

    // do for every edge (v, u)

    for (int u: graph.adjList[v])

    {

        if (!discovered[u])

        {

            // mark it as discovered and enqueue it

            discovered[u] = true;

            q.push(u);

        }

    }

    recursiveBFS(graph, q, discovered);

}

int main()

{

    // vector of graph edges as per the above diagram

    vector<Edge> edges = {

        {1, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}, {5, 9},

        {5, 10}, {4, 7}, {4, 8}, {7, 11}, {7, 12}

        // vertex 0, 13, and 14 are single nodes

    };

    // total number of nodes in the graph (labelled from 0 to 14)

    int n = 15;

    // build a graph from the given edges

    Graph graph(edges, n);

    // to keep track of whether a vertex is discovered or not

    vector<bool> discovered(n, false);

    // create a queue for doing BFS

    queue<int> q;

    // Perform BFS traversal from all undiscovered nodes to

    // cover all connected components of a graph

    for (int i = 0; i < n; i++)

    {

        if (discovered[i] == false)

        {

            // mark the source vertex as discovered

            discovered[i] = true;

            // enqueue source vertex

            q.push(i);

            // start BFS traversal from vertex `i`

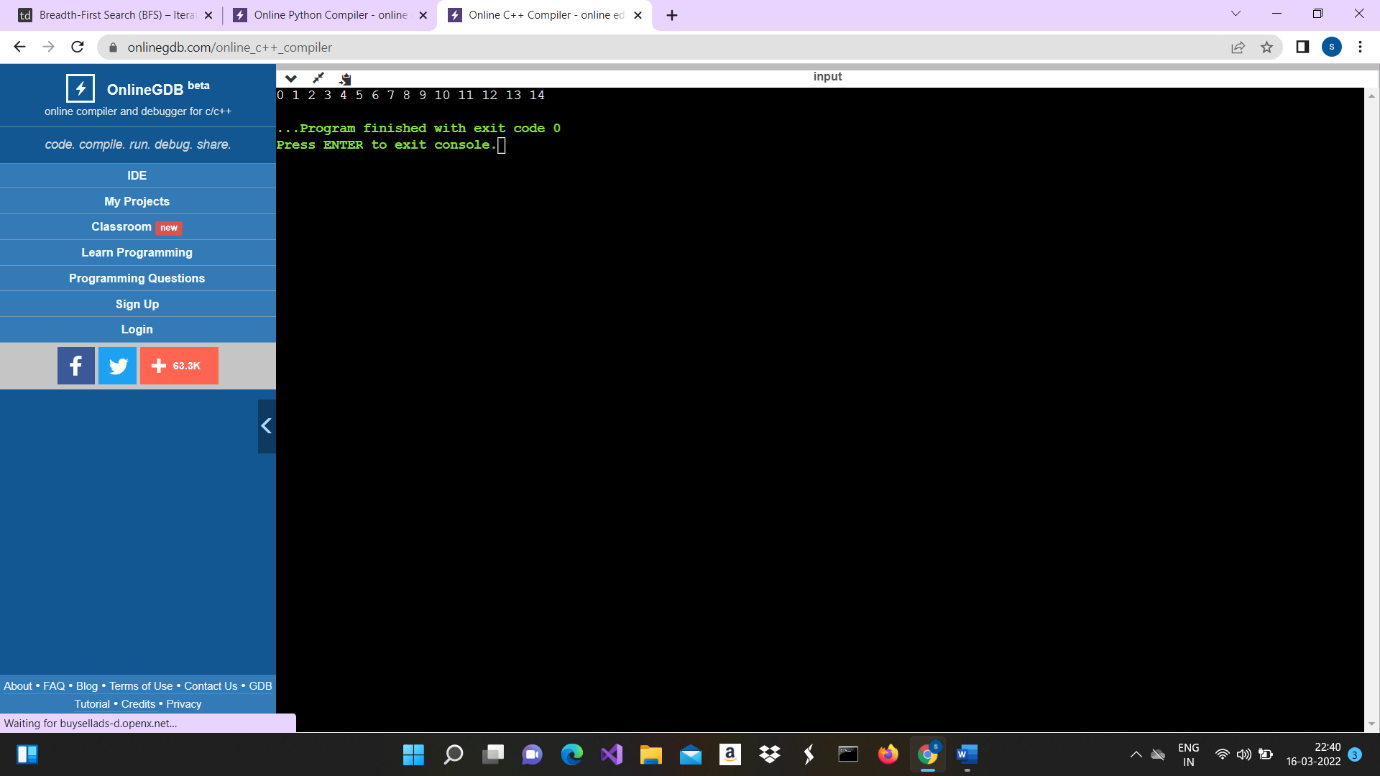
            recursiveBFS(graph, q, discovered);

        }

    }

    return 0;

}



DFS –

#include <iostream>

#include <vector>

using namespace std;

// Data structure to store a graph edge

struct Edge {

    int src, dest;

};

// A class to represent a graph object

class Graph

{

public:

    // a vector of vectors to represent an adjacency list

    vector<vector<int>> adjList;

    // Graph Constructor

    Graph(vector<Edge> const &edges, int n)

    {

        // resize the vector to hold `n` elements of type `vector<int>`

        adjList.resize(n);

        // add edges to the undirected graph

        for (auto &edge: edges)

        {

            adjList[edge.src].push\_back(edge.dest);

            adjList[edge.dest].push\_back(edge.src);

        }

    }

};

// Function to perform DFS traversal on the graph on a graph

void DFS(Graph const &graph, int v, vector<bool> &discovered)

{

    // mark the current node as discovered

    discovered[v] = true;

    // print the current node

    cout << v << " ";

    // do for every edge (v, u)

    for (int u: graph.adjList[v])

    {

        // if `u` is not yet discovered

        if (!discovered[u]) {

            DFS(graph, u, discovered);

        }

    }

}

int main()

{

    // vector of graph edges as per the above diagram

    vector<Edge> edges = {

        // Notice that node 0 is unconnected

        {1, 2}, {1, 7}, {1, 8}, {2, 3}, {2, 6}, {3, 4},

        {3, 5}, {8, 9}, {8, 12}, {9, 10}, {9, 11}

    };

    // total number of nodes in the graph (labelled from 0 to 12)

    int n = 13;

    // build a graph from the given edges

    Graph graph(edges, n);

    // to keep track of whether a vertex is discovered or not

    vector<bool> discovered(n);

    // Perform DFS traversal from all undiscovered nodes to

    // cover all connected components of a graph

    for (int i = 0; i < n; i++)

    {

        if (discovered[i] == false) {

            DFS(graph, i, discovered);

        }

    }

    return 0;

}

