**Name: Ashutosh Raj Gupta**

**Roll no: TECO2223A045**

**Batch: A3**

**ASSIGNMENT 1:** Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

Code: BFS

#include<iostream>

#include<queue>

#include<vector>

using namespace std;

struct Edge{

    int src,dest;

};

class Graph{

    public:

    vector<vector<int>> adjlist;   //adjacenecy list is created

    //class constructor

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

        //for holding n vectors of type int

        adjlist.resize(n);

        for(auto &edge:edges){

            //connecting the edges of undirected graph

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

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

        }

    }

};

void recursive\_bfs(Graph const &graph, queue<int> &q, vector<bool> &visited){

    if(q.empty()){

        return;

    }

    //getting first source node

    int v= q.front();

    //pop it

    q.pop();

    cout<<v<<" ";

    // do for every edge (v, u)

    for (int u: graph.adjlist[v])

    {

        if (!visited[u])

        {

            // mark it as discovered and enqueue it

            visited[u] = true;

            q.push(u);

        }

    }

    recursive\_bfs(graph, q, visited);

}

int main()

{

    cout<<"Enter the number of Edges\n";

    int numberOfEdges;

    cin>>numberOfEdges;

    vector<Edge> edges;

    for(int i =0;i<numberOfEdges;i++){

        Edge edge;

        int src,dest;

        cout<<"Enter "<<i+1<<"th edge\n";

        cout<<"source: ";

        cin>>src;

        cout<<"destination: ";

        cin>>dest;

        edge.src = src;

        edge.dest = dest;

        edges.push\_back(edge);

    }

    int n;

    cout<<"Enter number of nodes";

    cin>>n;

    Graph graph(edges, n);

    // to check whether a vertex is visited or not

    //in starting

    vector<bool> visited(n, false);

    queue<int> q;

    // Perform BFS traversal

    for (int i = 0; i < n; i++)

    {

        if (visited[i] == false)

        {

            visited[i] = true;

            q.push(i);

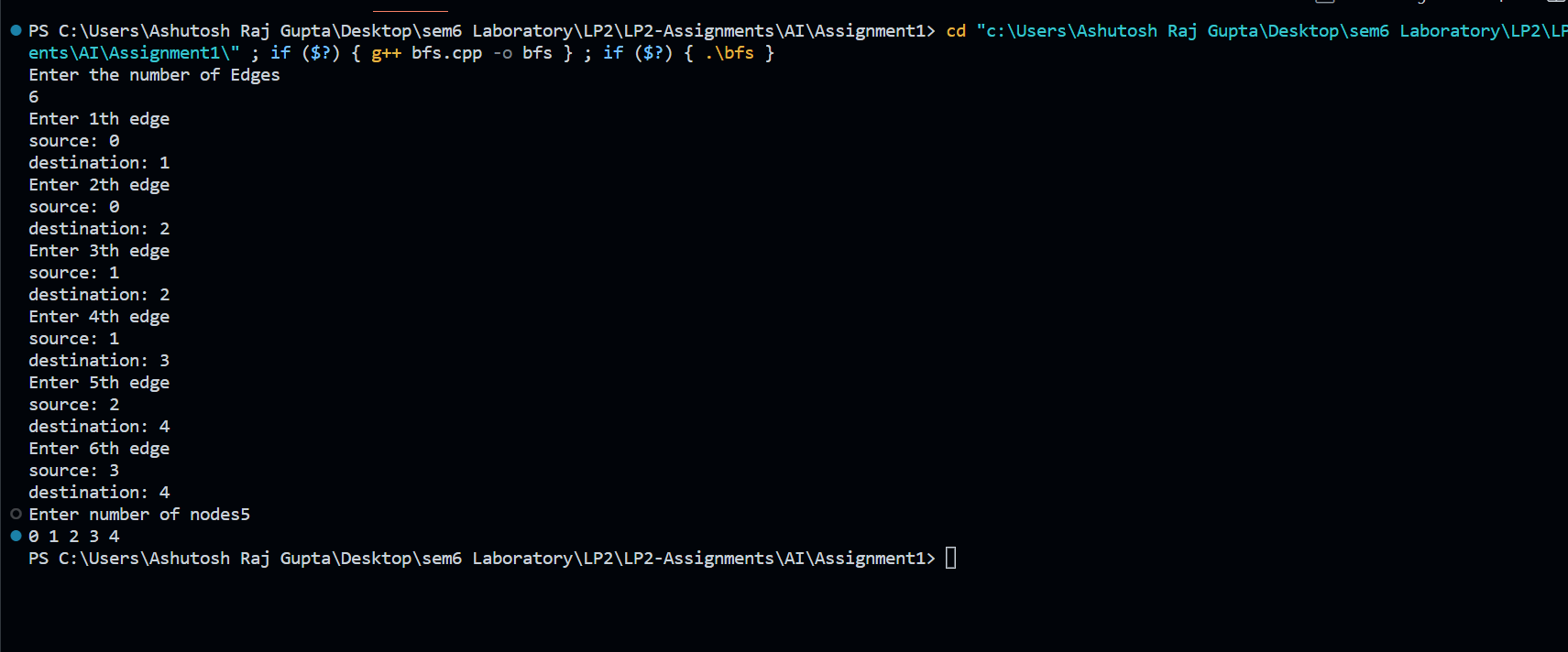
            recursive\_bfs(graph, q, visited);

        }

    }

    return 0;

}

****

DFS:

Code:

#include <iostream>

#include <vector>

using namespace std;

struct Edge

{

    int src, dest;

};

class Graph

{

public:

    // a vector of vectors to represent an adjacency list

    vector<vector<int>> adjList;

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

    {

        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

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

{

    // mark the current node as discovered

    visited[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 (!visited[u])

        {

            DFS(graph, u, visited);

        }

    }

}

int main()

{

    cout<<"Enter the number of Edges\n";

    int numberOfEdges;

    cin>>numberOfEdges;

    vector<Edge> edges;

    for(int i =0;i<numberOfEdges;i++){

        Edge edge;

        int src,dest;

        cout<<"Enter "<<i+1<<"th edge\n";

        cout<<"source: ";

        cin>>src;

        cout<<"destination: ";

        cin>>dest;

        edge.src = src;

        edge.dest = dest;

        edges.push\_back(edge);

    }

    int n;

    cout<<"Enter number of nodes";

    cin>>n;

    // build a graph from the given edges

    Graph graph(edges, n);

    vector<bool> visited(n);

    // Perform DFS traversal from all undiscovered nodes to

    for (int i = 0; i < n; i++)

    {

        if (visited[i] == false)

        {

            DFS(graph, i, visited);

        }

    }

    return 0;

}

