**Practical No.16**

**Problem statement:** Given a (directed/undirected) graph, design an algorithm and implement it using a program to find if a path exists between two given vertices or not.

**Algorithm:** To find path exist between two vertices or not.

Step 1:Start

Step 2: Input the number of vertices vertex.

Step 3: Initialize a 2D array mat[vertex][vertex] and fill all its elements with 0.

Step 4: Input the elements of the adjacency matrix mat.

Step 5: Input the source vertex m and destination vertex n.

Step 6: Check the value of mat[m][n]:

a-If mat[m][n] == 1, then Output "Yes path exist".

b-Else, Output "Path not exist".

Step 7: End

**Source code:**

#include<bits/stdc++.h>

using namespace std;

int main(){

    int vertex;

    cin>>vertex;

    int mat[vertex][vertex];

    for(int i=0;i<vertex;i++){

        for(int j=0;j<vertex;j++){

          mat[i][j]=0;

        }

    }

    for(int i=0;i<vertex;i++){

        for(int j=0;j<vertex;j++){

            cin>>mat[i][j];

        }

    }

    int m,n;

    cin>>m>>n;

    if(mat[m][n]==1){

        cout<<"Yes path exist";

    }

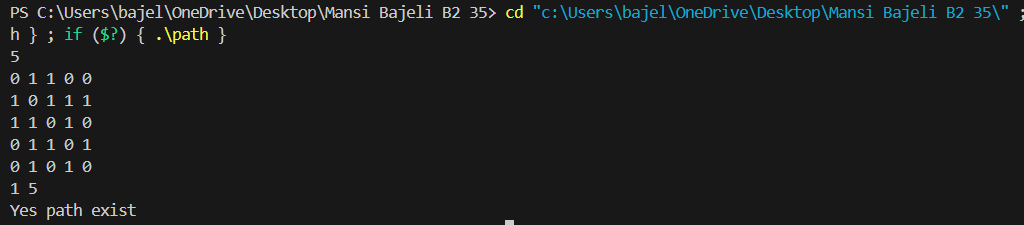
    else{

        cout<<"Path not exist";

    }

}

**OUTPUT:**

****

**Practical No.17**

**Problem statement:** Given a graph, design an algorithm and implement it using a program to find if a graph is bipartite or not.

**Algorithm:** To find if a graph is bipartite or not.

Step 1: Start

Step 2: check(s, v, mat, col)

1- Create a queue q and enqueue the starting node s.

2 -Set col[s] = 0 (assign first color).

3- While the queue is not empty:

a. Dequeue a node node from the queue.

b. For every vertex i from 0 to v - 1:

If mat[node][i] == 1 (i.e., edge exists):

a- If col[i] == -1 (i.e., not colored):

Assign opposite color to i: col[i] = !col[node]

Enqueue i

b-Else if col[i] == col[node]:

Return false

4- return true

Step 3: isBipartite(v, mat)

1-Create a color array col[v] and initialize all values to -1.

2-For each vertex i from 0 to v - 1:

a-If col[i] == -1, meaning not yet visited:

Call check(i, v, mat, col)

If it returns false, return false

3-If all components pass the check, return true

Step 4: main()

1-Read number of test cases T.

2-For each test case:

a. Read number of vertices v.

b. Declare mat[100][100] and initialize all entries to 0.

c. Read adjacency matrix of size v x v into mat.

d. Call isBipartite(v, mat):

If it returns true, print "Yes Bipartite."

Else, print "Not Bipartite."

Step 4: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

bool check(int s, int v, int mat[][100], int col[]) {

    queue<int> q;

    q.push(s);

    col[s] = 0;

    while (!q.empty()) {

        int node = q.front();

        q.pop();

        for (int i = 0; i < v; i++) {

            if (mat[node][i] == 1) {

                if (col[i] == -1) {

                    col[i] = !col[node];

                    q.push(i);

                } else if (col[i] == col[node]) {

                    return false;

                }

            }

        }

    }

    return true;

}

bool isBipartite(int v, int adj[][100]) {

    int col[v];

    for (int i = 0; i < v; i++) {

        col[i] = -1;

    }

    for (int i = 0; i < v; i++) {

        if (col[i] == -1) {

            if (!check(i, v, adj, col)) {

                return false;

            }

        }

    }

    return true;

}

int main() {

    int t;

    cout<<"Test cases:";

    cin>>t;

    while(t>0){

    int vertex;

    cout<<"enter vertices:";

    cin>>vertex;

    int mat[100][100];

    for(int i=0;i<vertex;i++){

        for(int j=0;j<vertex;j++){

          mat[i][j]=0;

        }

    }

    for(int i=0;i<vertex;i++){

        for(int j=0;j<vertex;j++){

            cin>>mat[i][j];

        }

    }

    if (isBipartite(vertex, mat)) {

        cout << "Yes bipartite." << endl;

    } else {

        cout << "Not bipartite." << endl;

    }

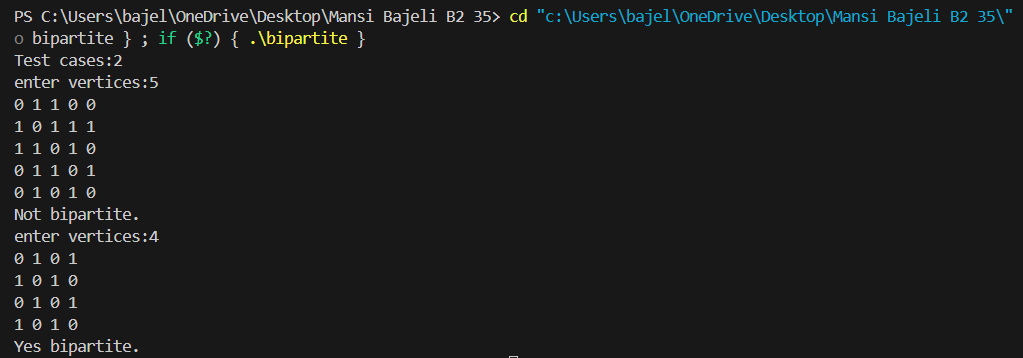
    t--;

    }

    return 0;

}

**Output:**

****

**Practical No.18**

**Problem statement:** Given a directed graph, design an algorithm and implement it using a program to find whether cycle exists in the graph or not

**Algorithm:** To detect cycle in directed graph.

Step 1: Start

Step 2: Function detectcycle(v, s, mat, vis, dfs):

1. Set vis[s] = 1

2. Set dfs[s] = 1

3. For each node i from 0 to v-1:

If mat[s][i] == 1:

a-If vis[i] == 0, call detectcycle(i, mat, vis, dfs)

b- If dfs[i] == 1, return true (cycle detected)

4. Set dfs[s] = 0 (backtrack)

5. Return false

Step 3: iscycle(adj, v):

1. Initialize vis[v] and dfs[v] to 0

2. For each node i from 0 to v-1:

a- If vis[i] == 0, call detectcycle(i, adj, vis, dfs)

b-If detectcycle returns true, return true (cycle detected)

3. Return false (no cycle detected)

Step 4: main():

1. Read the number of test cases t

2. For each test case:

a- Read the number of vertices vertex

b- Initialize mat[100][100] to zeros

c- Read the adjacency matrix mat

d- Call iscycle(mat, vertex)

e- If iscycle returns true, print "Cycle exist", otherwise print "No cycle exist"

Step5: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

bool detectcycle(int v,int s,int mat[][100],vector<int>&vis,vector<int>&dfs){

    vis[s]=1;

    dfs[s]=1;

    for(int i=0;i<v;i++){

    if(mat[s][i]==1){

        if(!vis[i]){

            if(detectcycle(v,i,mat,vis,dfs)){

                return true;

            }

        }

        else if(dfs[i]){

            return true;

        }

    }

    }

    dfs[s]=0;

    return false;

}

bool iscycle(int adj[][100],int v){

    vector<int>vis(v+1,0);

    vector<int>dfs(v+1,0);

    for(int i=0;i<v;i++){

        if(!vis[i]){

            if(detectcycle(v,i,adj,vis,dfs)){

                return true ; } }

    }

    return false;}

int main(){

int t;

cout<<"Test cases:";

cin>>t;

while(t>0){

    int vertex;

    cout<<"enter vertices:";

    cin>>vertex;

    int mat[100][100];

    for(int i=0;i<vertex;i++){

        for(int j=0;j<vertex;j++){

          mat[i][j]=0;

        } }

    for(int i=0;i<vertex;i++){

        for(int j=0;j<vertex;j++){

            cin>>mat[i][j];

        }}

   if(iscycle(mat,vertex)){

    cout<<"Cycle exist"<<endl;

   }

   else{

    cout<<"No cycle exist";

   }

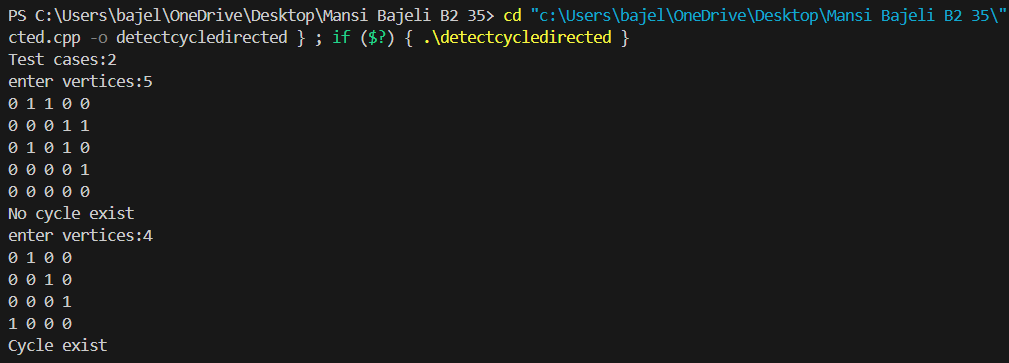
   cout<<endl;

   t--;

   }

}

**OUTPUT:**

****

**Practical No.19**

**Problem statement:** After end term examination, Akshay wants to party with his friends. All his friends are living as paying guest and it has been decided to first gather at Akshay’s house and then move towards party location. The problem is that no one knows the exact address of his house in the city. Akshay as a computer science wizard knows how to apply his theory subjects in his real life and came up with an amazing idea to help his friends. He draws a graph by looking in to location of his house and his friends’ location (as a node in the graph) on a map. He wishes to find out shortest distance and path covering that distance from each of his friend’s location to his house and then whatsapp them this path so that they can reach his house in minimum time. Akshay has developed the program that implements Dijkstra’s algorithm but not sure about correctness of results. Can you also implement the same algorithm and verify the correctness of Akshay’s results?

**Algorithm:** To find out shortest distance and path.

Step 1: Start

Step 2: printPath(parent, node)

1.If parent[node] == -1:  
 a- Print node + 1 (to convert from 0-based to 1-based indexing)  
 b-Return

2.Recursively call printPath(parent, parent[node])

3.Print node + 1

Step 3: Define dijkstra(graph[100][100], V, src)

1.Initialize vector dist[V] with INT\_MAX  
 a-To store shortest distance from src to all other nodes

2.Initialize vector parent[V] with -1  
 a-To help track the path

3.Use a min-priority queue pq of pairs {distance, node}

4.Set dist[src] = 0 and push {0, src} to pq

Step 4: While pq is not empty

1.Pop top element: u = pq.top().second, d = pq.top().first

2.For every node v from 0 to V - 1:

If graph[u][v] != 0 (i.e., there is an edge from u to v) anddist[v] > d + graph[u][v]:

a-Update dist[v] = d + graph[u][v]

b-Set parent[v] = u

c-Push {dist[v], v} into pq

Step 5: After the loop

1.For each node i from 0 to V - 1:

a-Call printPath(parent, i) to print the shortest path from source to i

b-Print : dist[i] to show the total distance

Step 6: Define main()

1.Read number of vertices V

2.Create and read adjacency matrix graph[V][V]

3.Read source node src (1-based), subtract 1 to convert to 0-based

4.Call dijkstra(graph, V, src)

Step 7: Stop

**Source code:**

#include<bits/stdc++.h>

#include <climits>

using namespace std;

void printPath(vector<int> &parent, int node) {

    if (parent[node] == -1) {

        cout << node + 1 << " ";

        return;

    }

    printPath(parent, parent[node]);

    cout << node + 1 << " ";

}

void dijkstra(int graph[100][100], int V, int src) {

    vector<int> dist(V, INT\_MAX);

    vector<int> parent(V, -1);

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    dist[src] = 0;

    pq.push({0, src});

    while (!pq.empty()) {

        int u = pq.top().second;

        int d = pq.top().first;

        pq.pop();

        for (int v = 0; v < V; v++) {

            if (graph[u][v] && dist[v] > d + graph[u][v]) {

                dist[v] = d + graph[u][v];

                parent[v] = u;

                pq.push({dist[v], v});

            }

        }

    }

    for (int i = 0; i < V; i++) {

        printPath(parent, i);

        cout << ": " << dist[i] << endl;

    }

}

int main() {

    int V;

    cin >> V;

    int graph[100][100];

    for (int i = 0; i < V; i++) {

        for (int j = 0; j < V; j++) {

            cin >> graph[i][j];

        }

    }

    int src;

    cin >> src;

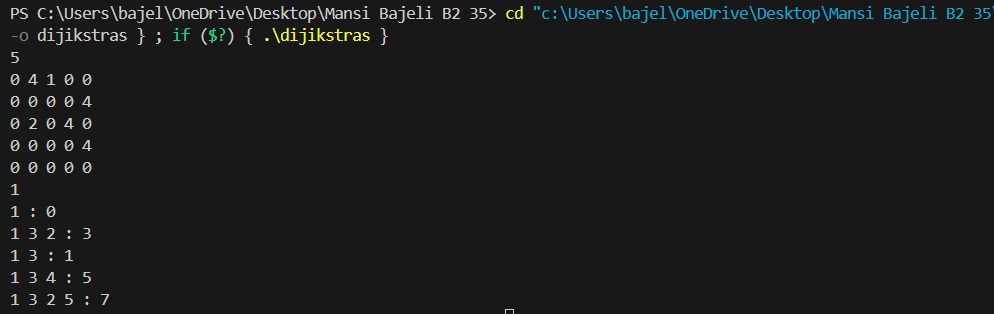
    src--;

    dijkstra(graph, V, src);

    return 0;

}

**OUTPUT:**

****

**Practical No.20**

**Problem statement:** Design an algorithm and implement it using a program to solve previous question's problem using Bellman- Ford's shortest path algorithm.

**Algorithm:** To find shortest path using Bellman- ford’s .

Step 1: Start

Step 2: printPath(parent, node)

1.Initialize an empty path list.

2.While node is not -1:

a-Append node + 1 to the path.

b-Set node = parent[node].

3.Print elements of path from beginning to end.

Step 3: bellmanford(V, start, graph, dist, parent)

1.Initialize:

a-dist[] to INT\_MAX for all vertices.

b-parent[] to -1 for all vertices.

c-Set dist[start] = 0

2.For i = 1 to V-1:

a-For each vertex u = 0 to V-1:

If dist[u] == INT\_MAX, skip to next u.

For each vertex v = 0 to V-1:

If graph[u][v] != 0 and dist[u] + graph[u][v] < dist[v], then:

a-Update dist[v] = dist[u] + graph[u][v]

b-Set parent[v] = u

3.After V-1 iterations, check for negative weight cycles:

a-For each vertex u = 0 to V-1:

If dist[u] == INT\_MAX, skip.

For each vertex v = 0 to V-1:

a-If graph[u][v] != 0 and dist[u] + graph[u][v] < dist[v], return false

4.If no negative cycle is found, return true.

Step 4: main()

1.Read number of vertices v

2.Read v x v adjacency matrix graph[100][100]

3.Read source vertex start and convert to 0-based by doing start--

4.Declare vectors dist[v] and parent[v]

5.Call bellmanford(v, start, graph, dist, parent)

a-If it returns true:

For each vertex i = 0 to v-1:

a-Call printPath(parent, i)

b-Print : dist[i] if reachable; otherwise, print INF

b-Else:

Print "Negative weight cycle detected"

Step 5: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

void printPath(vector<int>& parent, int node) {

    vector<int> path;

    while (node != -1) {

        path.push\_back(node + 1);

        node = parent[node];

    }

    for (int i = 0; i < path.size(); i++) {

        cout << path[i] << " ";

    }

}

bool bellmanford(int V, int start, int graph[100][100], vector<int>& dist, vector<int>& parent) {

    dist.assign(V, INT\_MAX);

    parent.assign(V, -1);

    dist[start] = 0;

    for (int i = 1; i <= V - 1; i++) {

        for (int u = 0; u < V; u++) {

            if (dist[u] == INT\_MAX) continue;

            for (int v = 0; v < V; v++) {

                if (graph[u][v] != 0 && dist[u] + graph[u][v] < dist[v]) {

                    dist[v] = dist[u] + graph[u][v];

                    parent[v] = u;

                }

            }

        }

    }

    for (int u = 0; u < V; u++) {

        if (dist[u] == INT\_MAX) continue;

        for (int v = 0; v < V; v++) {

            if (graph[u][v] != 0 && dist[u] + graph[u][v] < dist[v]) {

                return false;

            }

        }

    }

    return true;

}

int main() {

    int v;

    cin >> v;

    int graph[100][100];

    for (int i = 0; i < v; i++) {

        for (int j = 0; j < v; j++) {

            cin >> graph[i][j];

        }

    }

    int start;

    cin >> start;

    start--;

    vector<int> dist(v), parent(v);

    if (bellmanford(v, start, graph, dist, parent)) {

        for (int i = 0; i < v; i++) {

            printPath(parent, i);

            cout << ": ";

            if (dist[i] == INT\_MAX)

                cout << "INF";

            else

                cout << dist[i];

            cout << endl;

        }

    } else {

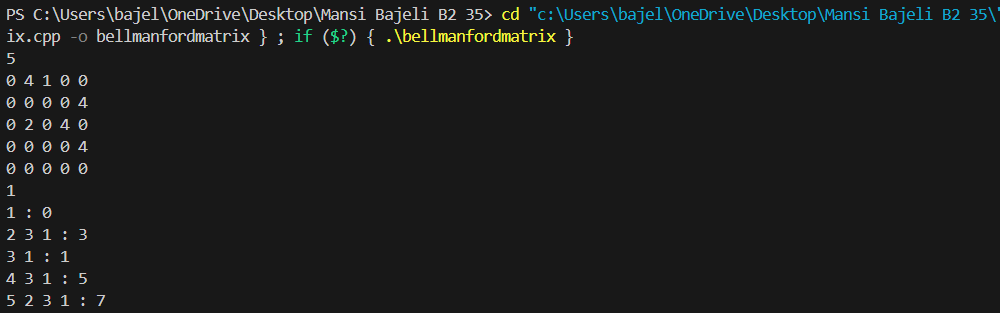
        cout << "Negative weight cycle detected" << endl;

    }

    return 0;

}

**OUTPUT:**

****

**Practical No.21**

**Problem statement:** Given a directed graph with two vertices ( source and destination). Design an algorithm and implement it using a program to find the weight of the shortest path from source to destination with exactly k edges on the path.

**Algorithm:** To find the weight of the shortest path from source to destination.

Step 1: Start

Step 2: shortestPath(graph, u, v, k, vertex)

1.If k == 0 and u == v, return 0 (base case: same node, 0 edges).

2.If k == 0 and u != v, return INT\_MAX (no path possible).

3.Initialize res = INT\_MAX.

4.Loop over all vertices i from 0 to vertex - 1:

a-If there is an edge from u to i (i.e., graph[u][i] != 0): a. Recursively call shortestPath(graph, i, v, k-1, vertex) and store the result in subPath. b. If subPath != INT\_MAX:

Update res = min(res, graph[u][i] + subPath).

5.Return res.

Step 3: main()

1.Read integer v (number of vertices).

2.Initialize a 2D array graph[100][100].

3.Read v x v adjacency matrix into graph.

4.Read integers src, dest, and k.

5.Decrement src and dest by 1 to convert to 0-based indexing.

6.Call shortestPath(graph, src, dest, k, v) and store result in result.

7.If result == INT\_MAX:

a-Print “no path of length k is available”.

8.Else:

b-Print “Weight of shortest path from (src+1, dest+1) with k edges: result”.

Step 4: Stop

**Source code:**

#include <bits/stdc++.h>

#include<climits>

using namespace std;

int shortestPath(int graph[100][100], int u, int v, int k, int vertex) {

    if (k == 0 && u == v) return 0;

    if (k == 0) return INT\_MAX;

    int res = INT\_MAX;

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

        if (graph[u][i] != 0) {

            int subPath = shortestPath(graph, i, v, k - 1, vertex);

            if (subPath != INT\_MAX) {

                res = min(res, graph[u][i] + subPath);

            }

        }

    }

    return res;

}

int main() {

    int v;

    cin >> v;

    int graph[100][100];

    for (int i = 0; i < v; i++) {

        for (int j = 0; j < v; j++) {

            cin >> graph[i][j];

        }

    }

  int src, dest, k;

    cin >> src >> dest >> k;

    src--; dest--;

    int result = shortestPath(graph, src, dest, k, v);

    if (result == INT\_MAX)

        cout << "no path of length " << k << " is available" << endl;

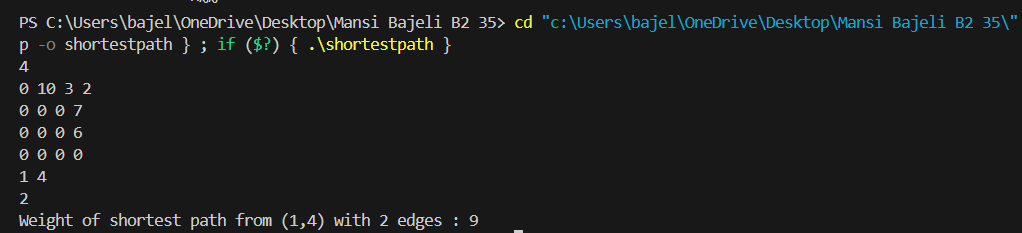
    else

        cout << "Weight of shortest path from (" << src + 1 << "," << dest + 1 << ") with " << k << " edges : " << result << endl;

    return 0;

}

**OUTPUT:**

****