**1. Introduction-**

Recursion. If you have already gone through that problem then this problem will be very easy for you to understand.Before you move any further, it is advised that you give this problem a fair try. Let's now jump to the problem!

Understanding the problem: In this problem you are given a graph, a source vertex and a destination vertex. All you have to do is to find if a path exists between src and dest. If it does, print true.> Input is managed for you If you are finding it difficult to understand then we recommend you to watch the question video of this topic. I am sure that question will get much clearer once you watch it.

**2. Approach :**

The problem here is to check whether there is any path present between src and dest. To solve this problem we will use Recursion. We know that if there is a path between src vertex and dest vertex then there must also exist a path between src's neighbor vertex and dest vertex too.So we can develop some expectation and faith for this problem.

The expectation from the hasPath(src, dest) function is that it will return us a Boolean value, indicating whether the path exists between src and dest.Keeping the faith> on the recursive call, hasPath(nbr, dest), that it works perfectly and will tell us whether there exists a path between src's neighbor and dest vertex. And meet this expectation with faith by trusting the faith's result and returning the same result.

If hasPath(nbr, dest) returns false then no path exists between nbr and dest. So, it implies that no path exists between src and dest too. And if it returns true then it means path exists between nbr and dest which means path also exists between src and dest.

And if src has n number of neighbors then we call hasPath() for each of the neighboring vertex unless a true is returned, and if true is received through any call then the further iterations are halted and true is returned.And the base case for this function will be when src becomes equal to dest. It means we have reached our destination and therefore we return true.But if we notice, as this is a recursive function so we can get to see some redundant calls.

As src will make recursive calls to each of its neighbors, suppose that it made the first call to its neighbor 1, nbr1. So now this nbr1 will further make calls to each of its neighbors. And not to forget, src is also one of nbr1's neighbors. So src will again make a recursive call to its neighbor and first of all nbr1 will be called.And it will give us run time error of stack overflow.So, to handle this situation, we make use of the Boolean array, visited, which is already present in the signature of the hasPath() function in the code editor.

Every time we enter the hasPath() function corresponding to a certain src, we update the visited array's src index value to true.And before making a call for any neighbor, we first check for the corresponding value of this neighbor's visited array And make a call only if it's false. As it implies that this vertex is being visited for the first time. Let's code this

ConsoleJava

public static boolean hasPath(ArrayList[] graph, int src, int dest, boolean[] visited) {

//base case

if (src == dest)

return true;

//setting the value corresponding to src in visited as true.

visited[src] = true;

//applying for loop to access each neighbor of src

for (Edge e : graph[src]) {

//checking whether this neighbor has been visited or not.

if (!visited[e.nbr]) {

//if not than a recursive call is made to this neighbor and result is stored

boolean nbrHasPath = hasPath(graph, e.nbr, dest, visited);

//if result is true than true is returned.

if (nbrHasPath)

return true;

}

}

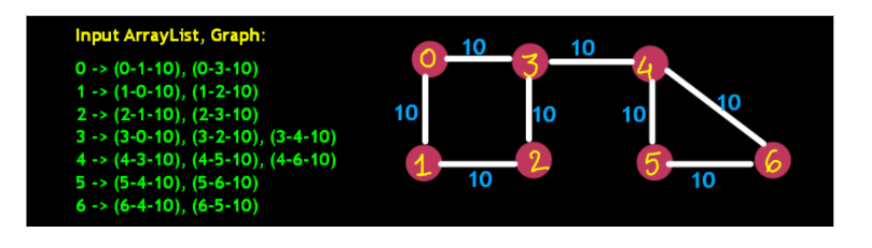
//if control reaches this point than it means that none of the neighbor has a path

//to reach the destination, so we false is returned.

return false;

}

Yes, that looks perfect!! Let's take an example and dry run this code.

****

ConsoleJava

public static boolean hasPath(ArrayList[] graph, int src,

int dest, boolean[] visited) {

if (src == dest) //1

return true;

visited[src] = true; //2

for (Edge e : graph[src]) { //3

if (!visited[e.nbr]) { //4

boolean nbrHasPath = hasPath(graph, e.nbr, dest, visited); //5

if (nbrHasPath) //6

return true;

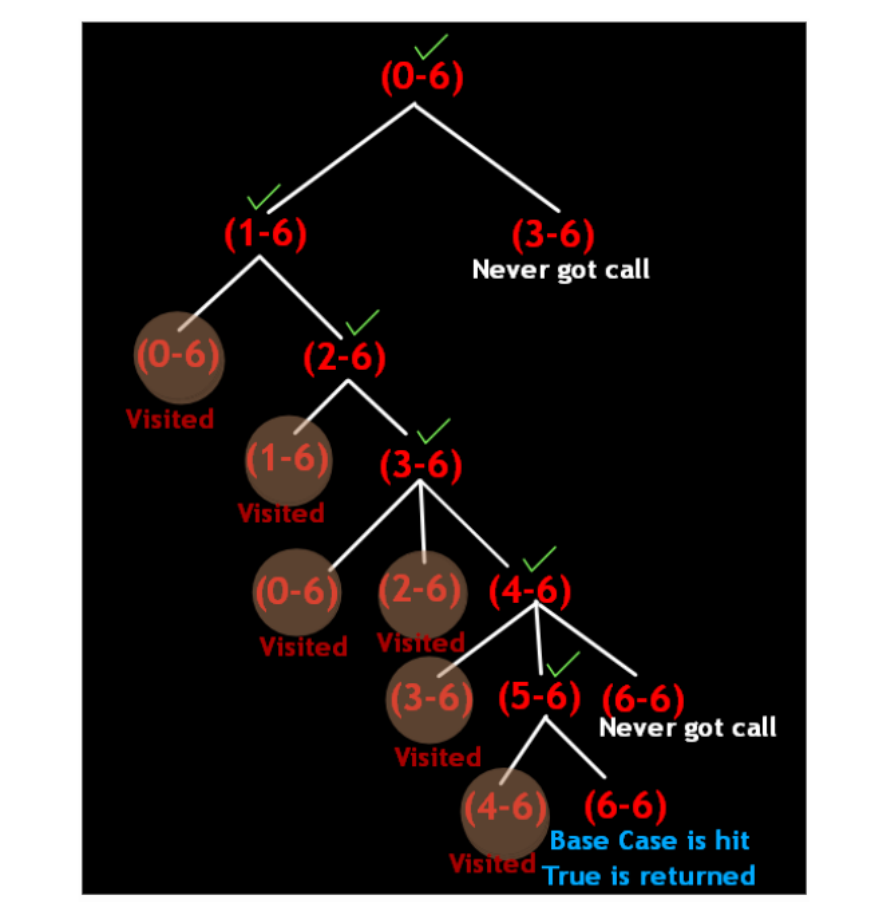
}

}

return false; //7

}

Statement-1 is false for source 0. So control jumps to statement-2 and the value at index 0 is set true in the visited array.Then using a for loop, a recursive call to 0's neighbor is made. First neighbor is 1.Then we check whether 1 has been visited earlier or not. 1 has not been visited. Therefore, a recursive call is made to hasPath(1, 6)

****

Statement-1 is false for source 1. So control jumps to statement-2 and the value at index 1 is set true in the visited array.Then using a for loop, a recursive call to 1's neighbor is made. First neighbor is 0.Then we check whether 0 has been visited earlier or not. 0 has been visited therefore we move to the next neighbor of 1 i.e. 2 in this case.Then we check whether 2 has been visited earlier or not. 2 hasn't been visited, therefore hasPath(2, 6) will be called.

Statement-1 is false for source 2. So control jumps to statement-2 and the value at index 2 is set true in the visited array.Then using a for loop, a recursive call to 2's neighbor is made. First neighbor is 1.Then we check whether 1 has been visited earlier or not. 1 has been visited, therefore we move to the next neighbor of 2 i.e. 3 in this case.Then we check whether 3 has been visited earlier or not. 3 has not been visited, so hasPath(3, 6) will be called.

Statement-1 is false for source 3. So control jumps to statement-2 and the value at index 3 is set true in the visited array.Then using a for loop, a recursive call to 3's neighbor is made. First neighbor is 0.Then we check whether 2 has been visited earlier or not. 2 has been visited, therefore we move to the next neighbor of 3 i.e. 4 in this case.

Then we check whether 4 has been visited earlier or not. 4 hasn't been visited, therefore hasPath(4, 6) will be called.Statement-1 is false for source 4. So control jumps to statement-2 and the value at index 4 is set true in the visited array.

Then using a for loop, a recursive call to 4's neighbor is made. First neighbor is 3.Then we check whether 3 has been visited earlier or not. 3 has been visited, therefore we move to the next neighbor of 4 i.e. 5 in this case.We check whether 5 has been visited earlier or not. 5 hasn't been visited, so hasPath(5, 6) will be called.

Statement-1 is false for source 5. So control jumps to statement-2 and the value at index 5 is set true in the visited array.Then using a for loop, a recursive call to 5's neighbor is made. First neighbor is 4.Then we check whether 4 has been visited earlier or not. 4 has been visited, therefore we move to the next neighbor of 5 i.e. 6 in this case.Then we check whether 6 has been visited earlier or not. 6 hasn't been visited, therefore hasPath(6, 6) will be called.

Statement-1 is true for source 6. Therefore "true" is returned.Then using a for loop, a recursive call to 5's neighbor is made. First neighbor is 4.Then we check whether 4 has been visited earlier or not. 4 has been visited, therefore we move to the next neighbor of 5 i.e. 6 in this case.Then we check whether 6 has been visited earlier or not. 6 hasn't been visited, therefore hasPath(6, 6) will be called.

Statement-1 is true for source 6. Therefore "true" is returned.

**3. Complexities:**

Time Complexity: O(V+E)

Where V is the number of vertices and E is the number of edges. In the worst case, all the vertices and all the edges will be travelled. The time complexity of the while loop (k log(k)).Which sum up to O(n log(k)).

Space Complexity: O(V)

It will be the height of the recursion stack, which can be O(V) at max

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

static class Edge {

int src;

int nbr;

int wt;

Edge(int src, int nbr, int wt) {

this.src = src;

this.nbr = nbr;

this.wt = wt;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int vtces = Integer.parseInt(br.readLine());

ArrayList< Edge>[] graph = new ArrayList[vtces];

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

graph[i] = new ArrayList<>();

}

int edges = Integer.parseInt(br.readLine());

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

String[] parts = br.readLine().split(" ");

int v1 = Integer.parseInt(parts[0]);

int v2 = Integer.parseInt(parts[1]);

int wt = Integer.parseInt(parts[2]);

graph[v1].add(new Edge(v1, v2, wt));

graph[v2].add(new Edge(v2, v1, wt));

}

int src = Integer.parseInt(br.readLine());

int dest = Integer.parseInt(br.readLine());

boolean[] visited = new boolean[vtces];

boolean flag = hasPath(graph, src, dest, visited);

System.out.println(flag);

}

public static boolean hasPath(ArrayList< Edge>[] graph, int src, int dest, boolean[] visited) {

if (src == dest)

return true;

visited[src] = true;

for (Edge e : graph[src]) {

if (!visited[e.nbr]) {

boolean nbrHasPath = hasPath(graph, e.nbr, dest, visited);

if (nbrHasPath)

return true;

}

}

return false;

}

}