Dear reader, welcome to the article on the problem named **‘**[**Is Graph Cyclic**](https://www.pepcoding.com/resources/online-java-foundation/graphs/is-cyclic-official/ojquestion)**’.**

The cycle detection algorithm, which we are going to learn now is **VERY IMPORTANT**, and you can expect various problems on it, in your coding rounds or the technical interviews.

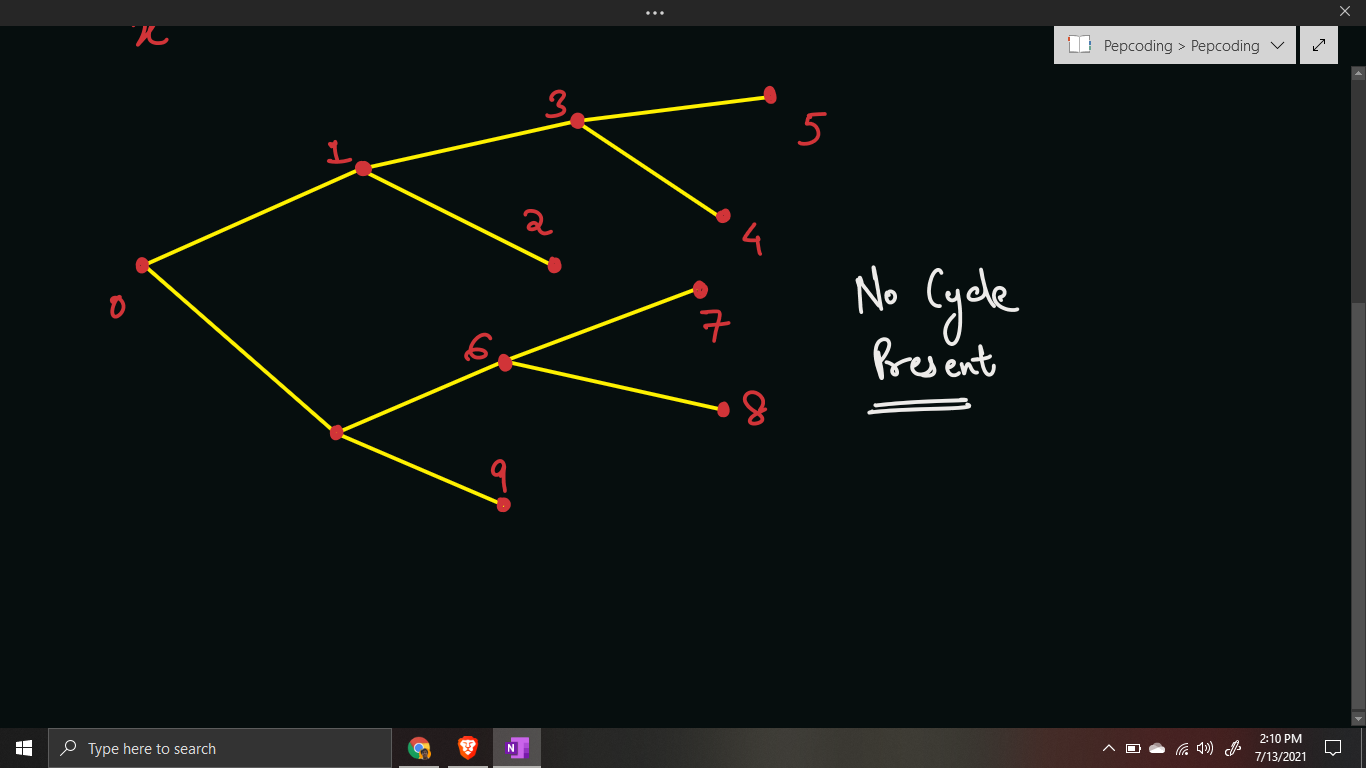
***Problem Statement:***

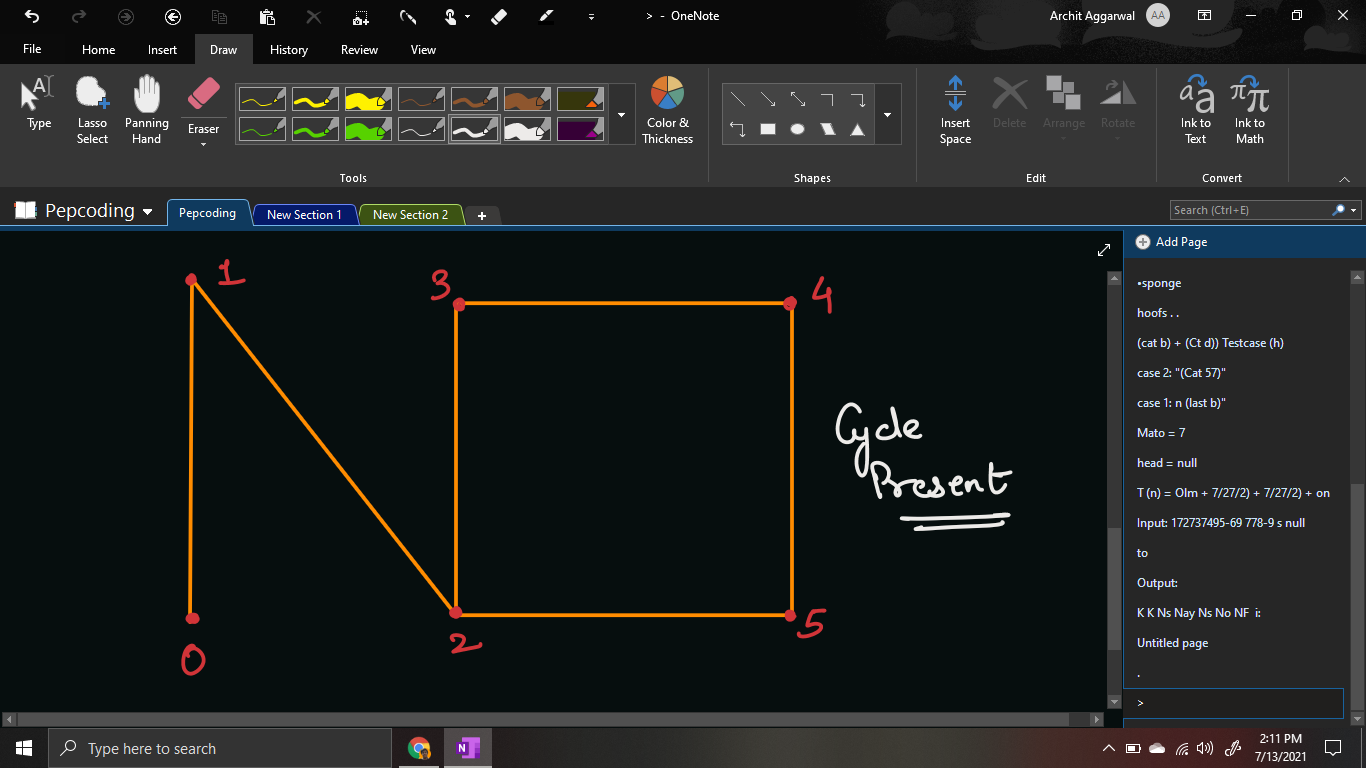
* Given an ***undirected graph***, check whether the graph is cyclic or not, i.e. find if there exists a cycle in the graph or not.
* Print *true* if there exists a cycle, else *false*.

*Note*: Input is given in the form of adjacency list.

*Note*: Graph may or may not be *connected*.

***Example:***



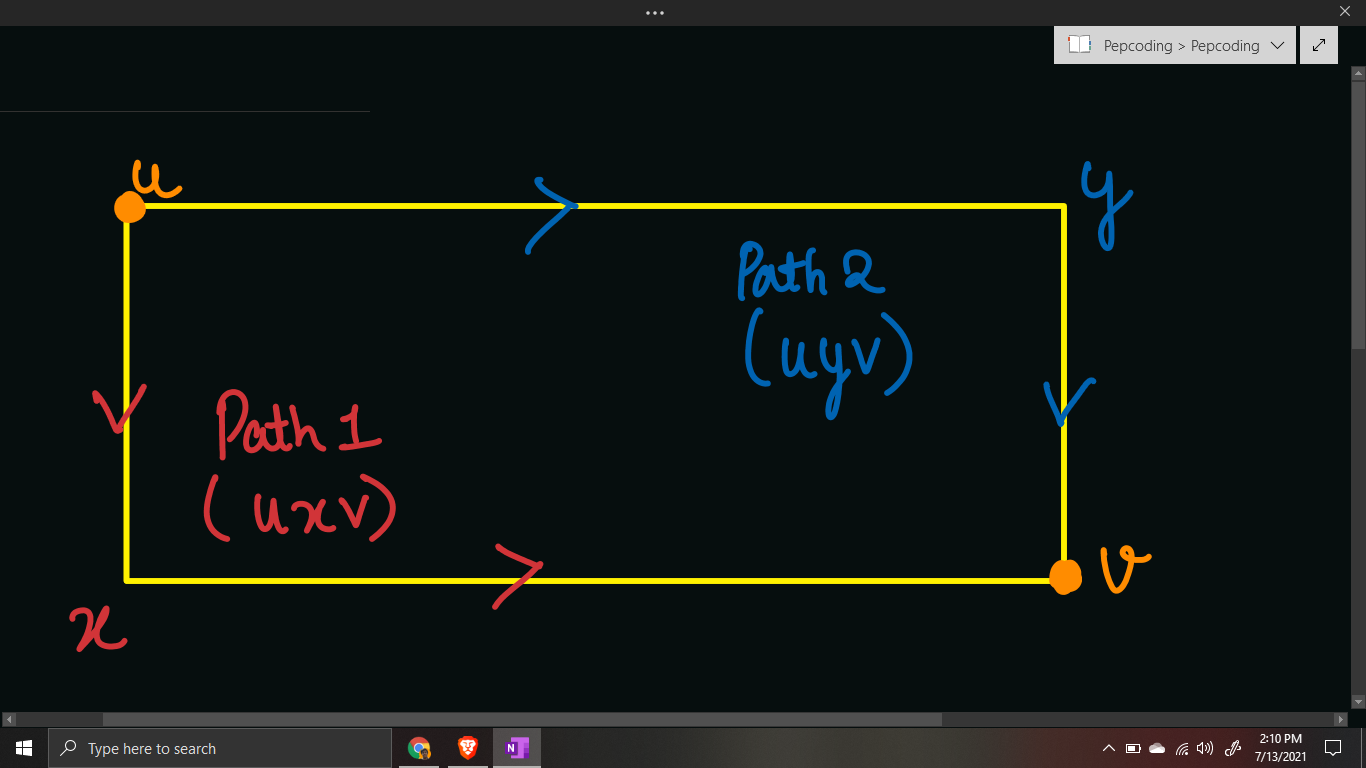


***Solution:***

The solution we are going to discuss in this article uses ***Breadth First Search (BFS)*** Algorithm.

What does it mean, when we say that there is a cycle present in the graph?

It means that there exists a pair of vertices (u, v) such that there is not an unique path from node u to node v. In simple words, there exists at least two different paths of reaching node v from the node u.



To find if there exists such two paths, we can do a BFS traversal of the given graph. For every visited vertex ‘v’, we will push all the neighbouring vertices ‘u’ into the queue.

Now, here is the ***crux*** of the algorithm. If the neighboring vertex is already visited, then there exists at least two different paths from node v to node u. It is because we were able to visit the vertex u, as it must have been popped out from the queue earlier.

Now, since there exists two different paths from node v to node u, there is a cycle present in the graph, (and nodes u and v must be part of that cycle).

**Q)** Can you figure out a problem here? (HINT: Graph is undirected, i.e. edges are bidirectional) ?

* Since the graph is undirected, it means there will be edge u to v, as well as edge v to u. Hence, node u will be present in the adjacency list of node v and similarly vice-versa.
* Once we will find the neighbours of node v, then we will get node u as a neighbour which is already visited. We will report that there is a cycle present, but the bidirectional edge does not mean a cycle.
* Thus, we should also maintain the information of the parent node, and not consider the parent node as the neighbour of the child node.
* We use a parent array to keep track of the parent vertex for a vertex so that we do not consider the visited parent as a cycle.

**Q)** Will the algorithm work if we start the BFS from any one node, let us suppose node 0?

* We are given in the problem statement that the graph may or may not be connected. Hence, we will have to check whether there exists a cycle in any component of the graph.
* To check cycles in any component, we will start BFS from each unvisited node in [0, n-1], and report true if there exists a cycle in any component.

***Note***: If you get this problem in a face-to-face interview, then you must ask about the corner cases like:

**Q)** Can there be edges of type (u, u), i.e. does self loops exist in the graph ?

**Q)** Can there be multiple parallel edges, i.e. does there exist more than 1 edge (u, v) in the graph ?

For simplicity, we can assume that there are no self loops and parallel edges in the current problem.

***Implementation***

*Note*: Before reading the Code, we recommend that you must try to come up with the solution on your own. Now, hoping that you have tried by yourself, here is the Java code.

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));

}

boolean[] visited = new boolean[vtces];

int[] parent = new int[vtces];

Arrays.fill(parent, -1); // Initially, all nodes will have parent -1.

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

if(visited[v] == false){

boolean cyclic = IsCyclic(graph, v, visited, parent);

if(cyclic){

System.out.println(true);

return;

}

}

}

System.out.println(false);

}

public static boolean IsCyclic(ArrayList<Edge>[] graph, int src,

boolean[] visited, int[] parent) {

ArrayDeque<Integer> queue = new ArrayDeque<>();

queue.add(src);

visited[src] = true;

while(queue.size() > 0){

int rem = queue.remove();

for (Edge e : graph[rem]) {

if(visited[e.nbr] && parent[rem] != e.nbr){

// e.nbr is visited already & it is not the parent of rem.

// Hence, there exists a cycle

return true;

}

if (!visited[e.nbr]) {

queue.add(e.nbr);

visited[e.nbr] = true;

parent[e.nbr] = rem;

}

}

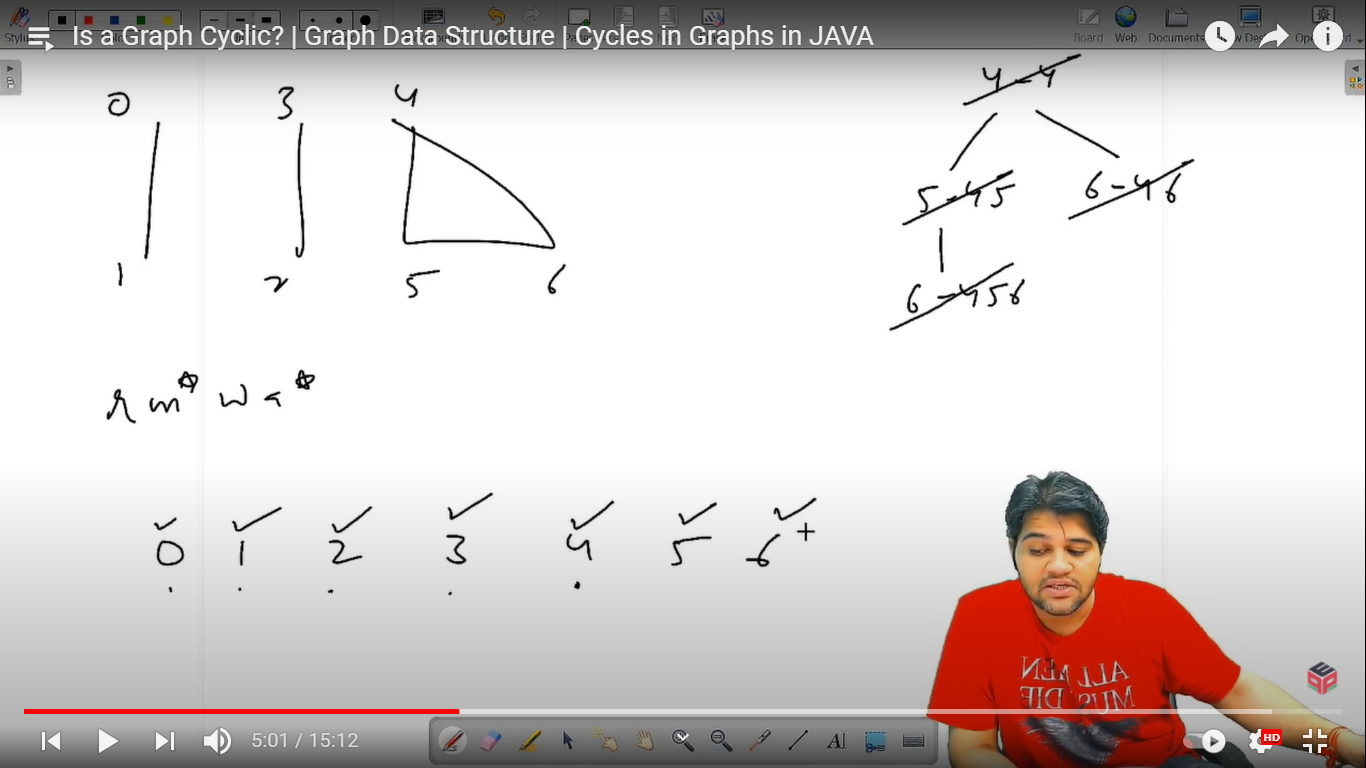
}

return false;

}

}

This code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=qbQq-k75bcY&list=TLGGc_WC0z8RiFoxMzA3MjAyMQ). Do check it out to understand the concept completely.



* What is the ***time complexity*** of the above code?

We are doing a simple BFS traversal of the graph. Hence the time complexity will be **O(N + E)** where N = number of vertices and E = number of edges.

* What is the ***space complexity*** of the above code ?

We are building a visited array and a parent array of size equal to the number of vertices. Also, we are taking the queue data structure to perform BFS traversal, which will store at max n nodes.

Hence, the total space complexity will be O(n + n + n) = **O(n)** only.

Please note that we are not taking into account the space taken to build the adjacency list, as it was given to us as an *input*.

**Follow Up:**

There are many methods of cycle detection in graphs, such as:

1. **Cycle Detection in Undirected Graph Using DFS**
2. **Cycle Detection in Undirected Graph Using BFS**
3. **Cycle Detection in Undirected Graph Using DSU (Disjoint Set Union)**
4. **Cycle Detection in Directed Graph using DFS**
5. **Cycle Detection in Directed Graph using Graph Coloring**
6. **Cycle Detection in Directed Graph using Topological Sort (Kahn’s Algorithm/BFS)**

All these algorithms are different from each other, and can be used interchangeably depending upon the type of graph (directed/undirected) and the type of problem.

Hope that you liked the article on *‘Is Graph Cyclic’*.

Subscribe to Pepcoding’s youtube channel for more such amazing video content on Data Structures & Algorithms.

You can suggest any improvements to the article on our telegram channel, or on the youtube channel’s comment section.

Article Contributed by: [Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)