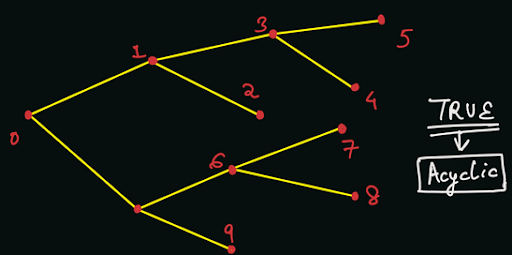
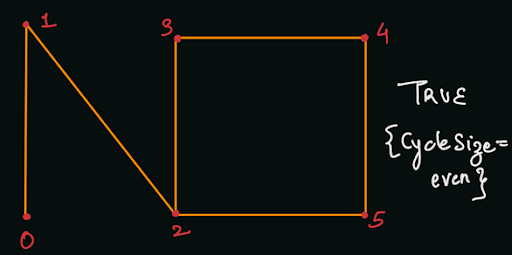
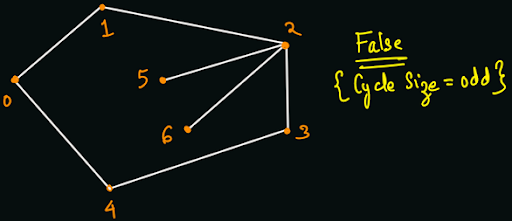
**1. Problem Discussion :**

1• Given a graph (undirected in this case), check whether the graph is bipartite or not. 2• A graph is called bipartite if it is possible to split it's vertices in two mutually exclusive and exhaustive disjoint sets of vertices such that all edges are across sets only, and there is no edge present between the vertices of the same set. 3• Print true if the graph is bipartite otherwise false. Note: Input is given in the form of adjacency list.

Example :

****

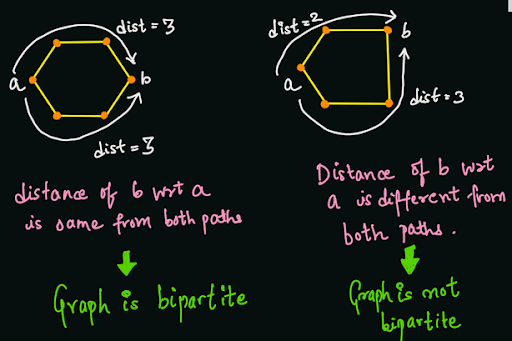
****

****

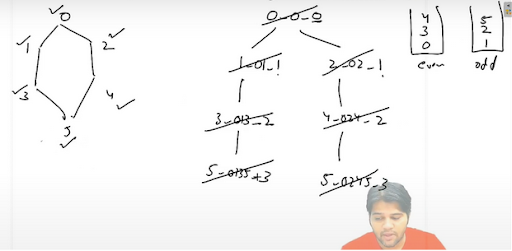
**2. Approach:**

Solution:

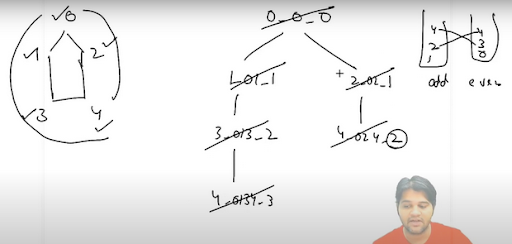
According to the definition, we need to distribute vertices into two subsets which are mutually exclusive: No vertex should appear in both the sets, or the intersection of sets should be empty. Also, they need to be exhaustive: Every vertex should appear in either one of the two sets, or the union of sets should be the entire set {0, 1, 2, ... n-1}. Let us first analyze WHAT we will have to do in the algorithm. If the graph is acyclic, i.e. there is no cycle present in the graph, then the graph is always bipartite. If there exists any one cycle of odd length, i.e. the number of vertices in the cycle are odd in counting, then the graph is not bipartite. If all the cycles (if present) are of even length, i.e. the number of vertices in each cycle are even in counting, then the graph is bipartite. The algorithm, which we are going to use to check bipartiteness of a graph is known as graph-coloring algorithm, which uses BFS traversals. A graph is bipartite if and only if it is two-colorable. Now, we will discuss HOW to check the presence of odd or even length cycles in the graph.

****

We will start a breadth-first traversal, starting from each vertex which hasn't been visited yet. But, instead of just pushing the node's value into the queue, we will also push a visiting time of the node. We will try to add the nodes with even visiting time in the first set, and the nodes with odd visiting time in the second set. We know already that during the bfs traversal, if we find any node which is already visited, then there exists a cycle. If the length of cycle is even, then the visiting time of the last node will be the same from both the paths (if even from the first path, then it will be even from the second path as well, and vice-versa).

****

Else, since the length of cycle is odd, the visiting time of the last node in the cycle will be different from both the paths (if even from the first path, then it will be odd from the second path, and vice-versa). Hence, we cannot put the last node in either the first set or the second set. Thus, the graph will not be bipartite in an odd-length cycle case.

****

You can clearly see in the example, that the last node of the odd-length cycle in the above case is 4. If we place node = 4 in the odd set, then we will have an edge 2-4 within the set. If we place node = 4 in the even set, then we will have an edge 3-4 within the set. Hence, we cannot place the node = 4 in either of the two sets. Once we've visited all vertices, and did not find any odd-length cycle, and successfully assigned them to one of the two disjoint sets, we know that the graph is bipartite and we have constructed its partitioning. Note: If you get this problem in a face-to-face interview, then you must ask about the corner cases like: Q) Is the graph connected, i.e. there is only one single-connected component? 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. But, the graph can have more than 1 component. Thus, we must call for the BFS traversal from each unvisited vertex in {0, 1, 2, 3, ... n-1}. 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.

**3. Code :**

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));

}

HashMap< Integer, Integer> visited = new HashMap<>();

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

if (!visited.containsKey(v)) {

boolean bip = IsBipartite(graph, v, visited);

if (!bip) {

System.out.println(false);

return;

}

}

}

System.out.println(true);

}

static class Pair {

int vtx;

int level;

Pair(int vtx, int level) {

this.vtx = vtx;

this.level = level;

}

}

public static boolean IsBipartite(ArrayList< Edge>[] graph,

int src, HashMap< Integer, Integer> visited) {

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

queue.add(new Pair(src, 0));

while (queue.size() > 0) {

Pair rem = queue.remove();

if (visited.containsKey(rem.vtx)) {

if (visited.get(rem.vtx) % 2 != rem.level % 2) {

return false;

}

} else {

visited.put(rem.vtx, rem.level);

}

for (Edge e : graph[rem.vtx]) {

if (!visited.containsKey(e.nbr)) {

queue.add(new Pair(e.nbr, rem.level + 1));

}

}

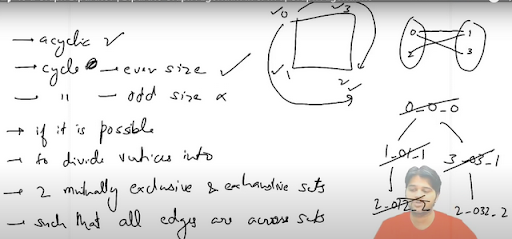
}

return true;

}

}

This code is written and explained by our team in the solution video . Do check it out to understand the concept completely.

****

**4. Analysis:**

Time Complexity:

We are simply doing a BFS traversal of the entire graph, which will take O(N + E) time, where N = number of vertices, and E = number of edges.

Space Complexity:

We are using a queue data structure for the BFS traversal, which will store at max N vertices. Hence, the space complexity is O(N). 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.

**5. Follow Up: O(h)**

We are not asked to print the two disjoint sets. But can you do the same?

Please remember that there can be more than 1 bipartite matching possible, if the graph is disconnected. It is because 2 nodes which belong to different components can belong to either the same disjoint set or different as well. Hope that you liked the article on "Is Graph Bipartite". 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.