Bipartite Graph | DFS Implementation

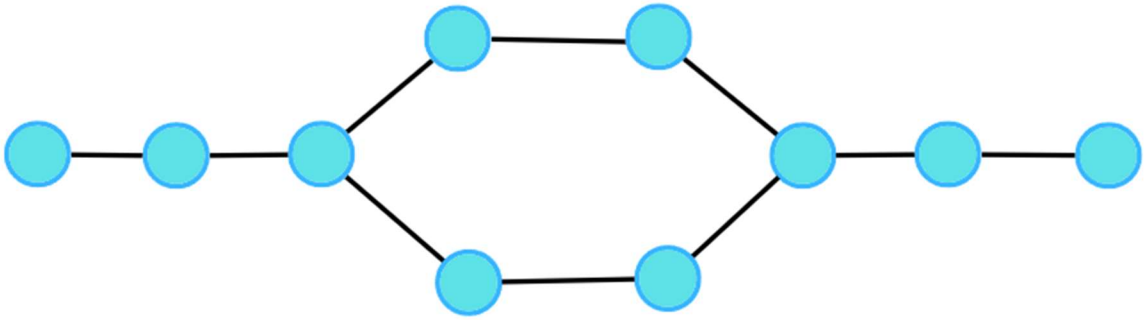
Problem Statement: Given an adjacency list of a graph adj of V no. of vertices having 0 based index. Check whether the graph is bipartite or not.

If we are able to colour a graph with two colours such that no adjacent nodes have the same colour, it is called a bipartite graph.

Examples:

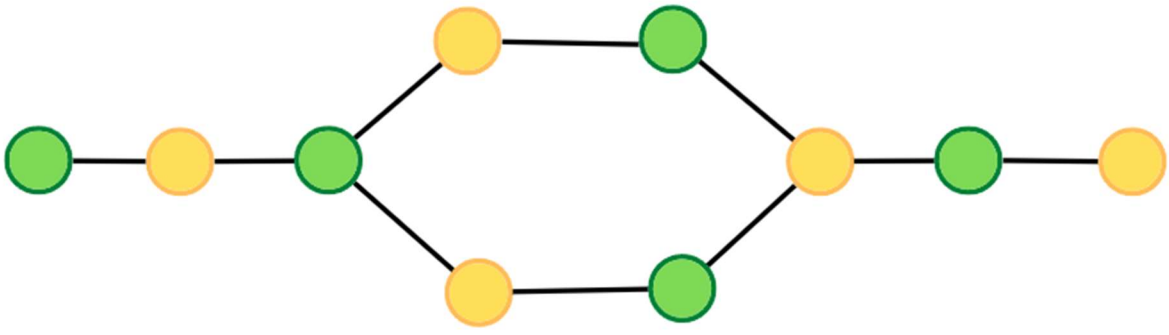
Example 1:

Input:



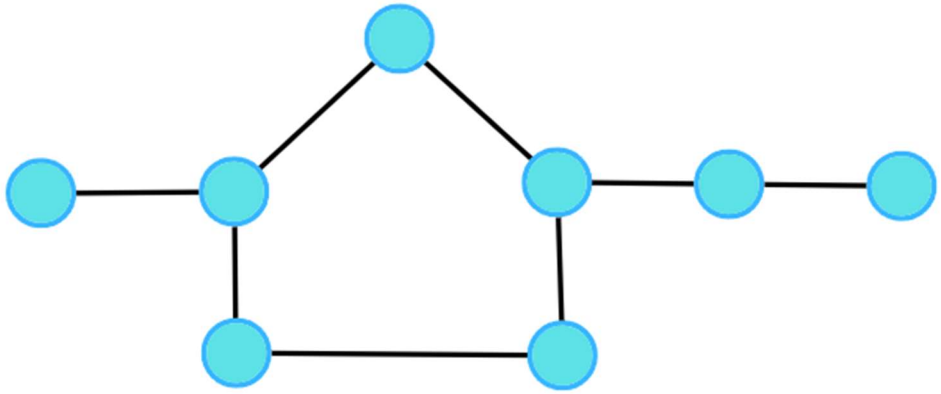
Output: 1

Explanation:



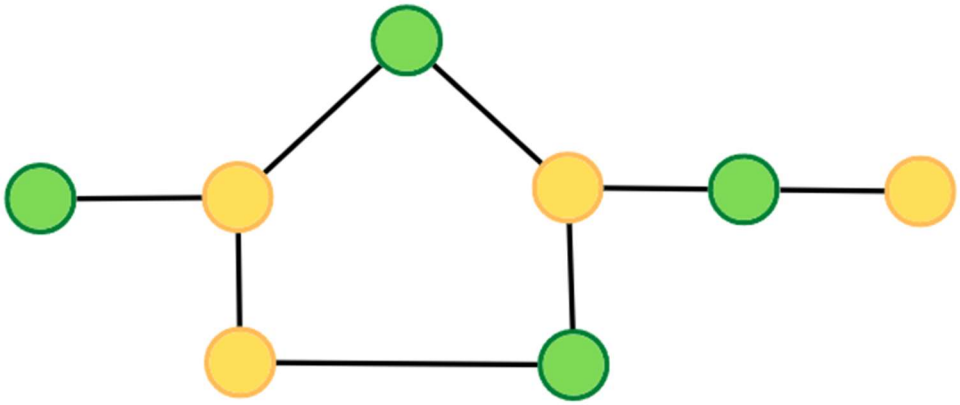
Example 2:

Input:



Output: 0

Explanation:



Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Intuition:.

A bipartite graph is a graph which can be coloured using 2 colours such that no adjacent nodes have the same colour. Any linear graph with no cycle is always a bipartite graph. With a cycle, any graph with an even cycle length can also be a bipartite graph. So, any graph with an odd cycle length can never be a bipartite graph.

The intuition is the brute force of filling colours using any traversal technique, just make sure no two adjacent nodes have the same colour. If at any moment of traversal, we find the adjacent nodes to have the same colour, it means that there is an odd cycle, or it cannot be a bipartite graph.

Approach:

We can follow either of the traversal techniques. In this article, we will be solving it using DFS traversal.

DFS is a traversal technique which involves the idea of recursion and backtracking. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

We will be defining the DFS traversal below, but this check has to be done for every component, for that we can use the simple for loop concept that we have learnt, to call the traversals for unvisited nodes.

```
// check for connected components in a graph
for ( i = 1; i<= n; i++ )
{
    if(!vis[i])
    {
        if( dfs(i) == true)
            return true;
    }
}
return false;
```

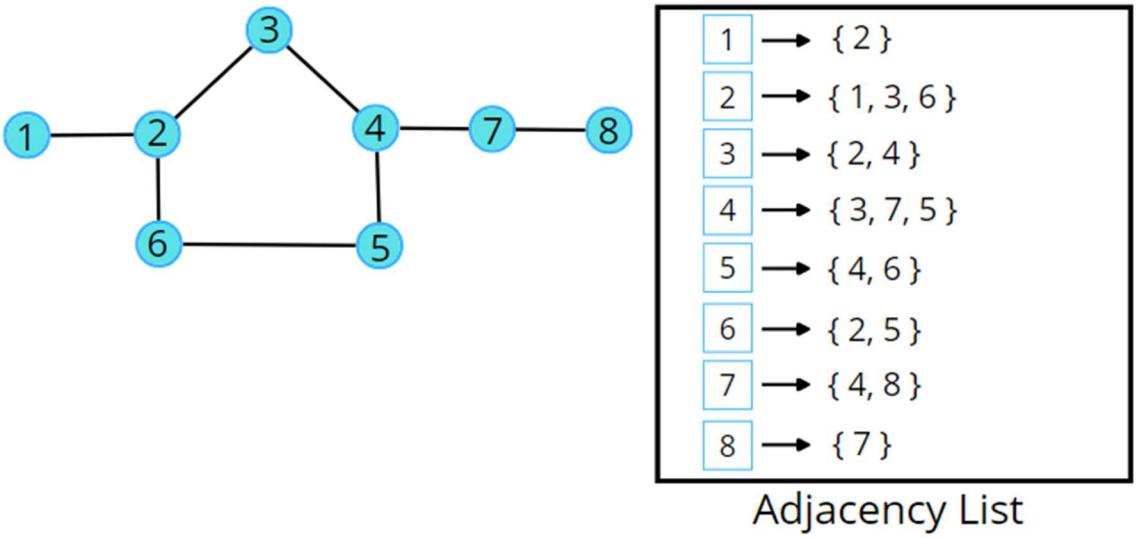
The algorithm steps are as follows:

- For DFS traversal, we need a start node and a visited array but in this case, instead of a visited array, we will take a colour array where all the nodes are initialised to -1 indicating they are not coloured yet.
- In the DFS function call, make sure to pass the value of the assigned colour, and assign the same in the colour array. We will try to colour with 0 and 1, but you can choose other colours as well. We will start with the colour 0, you can start with 1 as

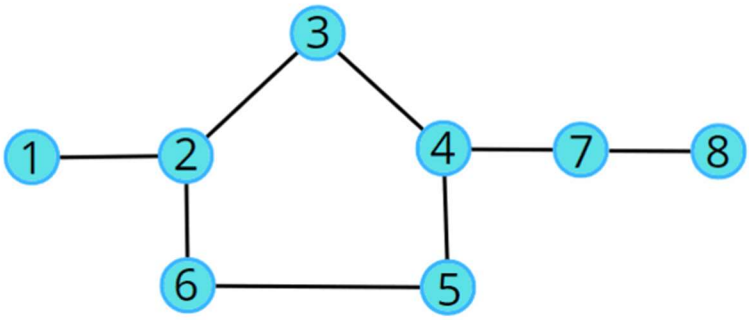
well, just make sure for the adjacent node, it should be opposite of what the current node has.

- In DFS traversal, we travel in-depth to all its uncoloured neighbours using the adjacency list. For every uncoloured node, initialise it with the opposite colour to that of the current node.
- If at any moment, we get an adjacent node from the adjacency list which is already coloured and has the same colour as the current node, we can say it is not possible to colour it, hence it cannot be bipartite. Thereby return a false indicating the given graph is not bipartite; otherwise, keep on returning true.

Consider the following graph and its adjacency list.



Consider the following illustration to understand the colouring of the nodes using DFS traversal.



1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1

Colour Array

Code:

- C++ Code
- Java Code

```cpp
#include<bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool dfs(int node, int col, int color[], vector<int> adj[]) {
        color[node] = col;

        // traverse adjacent nodes
        for(auto it : adj[node]) {
            // if uncoloured
            if(color[it] == -1) {
                if(dfs(it, !col, color, adj) == false) return false;
            }
            // if previously coloured and have the same colour
            else if(color[it] == col) {
                return false;
```

```
            }
        }

        return true;
    }
public:
        bool isBipartite(int V, vector<int>adj[]){
            int color[V];
            for(int i = 0;i<V;i++) color[i] = -1;

            // for connected components
            for(int i = 0;i<V;i++) {
                if(color[i] == -1) {
                    if(dfs(i, 0, color, adj) == false)
                        return false;
                }
            }
            return true;
        }
};

void addEdge(vector <int> adj[], int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main(){

        // V = 4, E = 4
        vector<int>adj[4];

        addEdge(adj, 0, 2);
        addEdge(adj, 0, 3);
        addEdge(adj, 2, 3);
        addEdge(adj, 3, 1);

        Solution obj;
        bool ans = obj.isBipartite(4, adj);
        if(ans)cout << "1\n";
        else cout << "0\n";

        return 0;
}
```

Output: 0

Time Complexity: O(V + 2E), Where V = Vertices, 2E is for total degrees as we traverse all adjacent nodes.

Space Complexity: O(3V) ~ O(V), Space for DFS stack space, colour array and an adjacency list.