MEDIUM

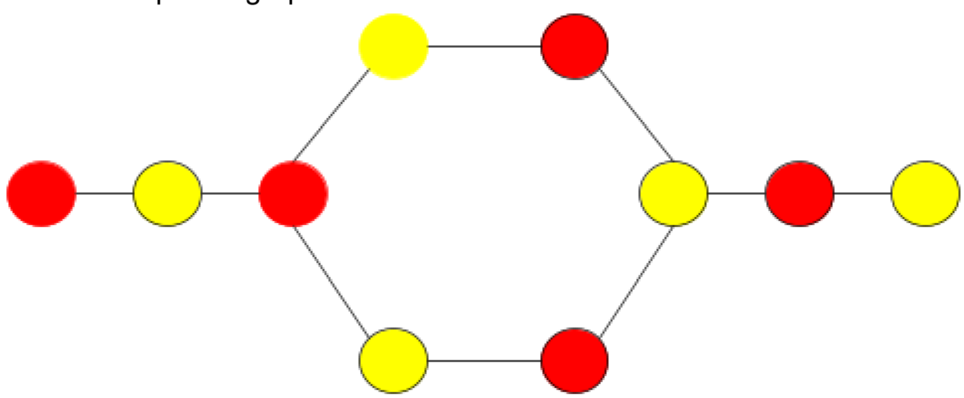
Bi-partite Graph | DFS

Intuition

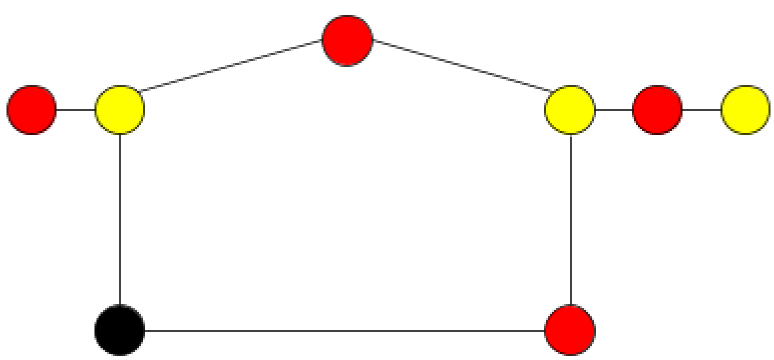
Definition : If you can color the graph with two colors such that no adjacent nodes have the same color then the graph is Bipartite graph.

Eg.

This is a Bi-partite graph :



This is not a Bi-partite Graph :



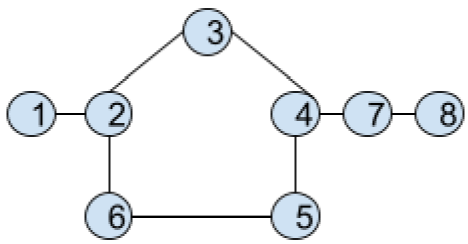
When we come to the red node it leads us to a dilemma that the color of the adjacent node should be colored yellow but the adjacent node has a neighbor of yellow color which make is not a bi-partite graph

Inference

- The linear graphs with no cycles are always bi-partite graphs
- If the graph has a cycle then :
    - If cycle length is even than it can also be bi-partite

- Else it's not a bi-partite Graph

eg.



Adjacency List :

1 : [2]
2 : [1,3]
3 : [2,4]
4 : [3,5,7]
5 : [4,6]
6 : [2,5]
7 : [4,8]
8 : [7]

Color : [0, 1, 0, 1 ,0, 1, 0, 1]

DFS(1,color = 0)->DFS(2.color = 1)->DFS(3,color = 0)->DFS(4,color=1)->DFS(5,color=0)

DFS(5,0) [returns false] | DFS(7,0)
DFS(6,1) [returns false] DFS(8,1) [returns]
2->adjacent node with same color hence its not a bi-partite

The series moves upwards as false, hence this is not a bi-partite

Approach :

DFS() :

- Color the source node
- Traverse through the adjacent nodes of the supplied source :
    - Check if the node is uncolored :
        - Check by making dfs call as dfs(adj_node, !col, adj, color) if the graph cannot be colored
            - Return false if the graph cannot be colored
    - Check if the node is colored and the color is same as the source :

- Return false because in this case the node cannot be colored in bi-partite fashion
- Return true outside the loop

IsBipartite()
- Declare :
    - Color vector having size v initially all elements assigned value -1
- Traverse for all graph components :
    - Check if the node is not colored :
        - Check by making dfs call as dfs(node, col, adj, color) if the graph cannot be colored
            - Return false if the graph cannot be colored
- Return true

Function Code :

```
bool dfs(int node, int col, vector<int> adj[], vector<int> &color) {
        // Coloring the supplied source node
        color[node] = col;
        // traversing through the adjacent elements of the given node
        for (int i : adj[node]) {
            // check if the adjacent node is uncolored
            if (color[i] == -1) {
                // checking if the node can not be colored
                if (dfs(i, !col, adj, color)==false) {
                    // return false if the node can not be colored
                    return false;
                }
            }
            // checking if the node adjacent is already colored with the
color of the current element
            else if (color[i] == col) {
                // return false in this case because it now cannot be
colored in a bi-partite fashion
                return false;
            }
        }
        // return true after the loop
        return true;
    }

    bool isBipartite(int V, vector<int> adj[]) {
        // Declare
        // Color vector having size v and initially all elements are
```

```
assigned no color ie. -1
        vector<int> color(V, -1);
        // traverse through all of the elements ie. nodes for getting all
graph components
        for (int i = 0; i < V; i++) {
            // check if the component node is already colored or not
            if (color[i] == -1) {
                // check if the component node can not be colored
                if (!dfs(i, 0, adj, color)) {
                    // return false if it can not be colored
                    return false;
                }
            }
        }
        // return true outside the component loop means the graph is
bi-partite
        return true;
    }
```

Time Complexity

O(V*E)