MEIDUM

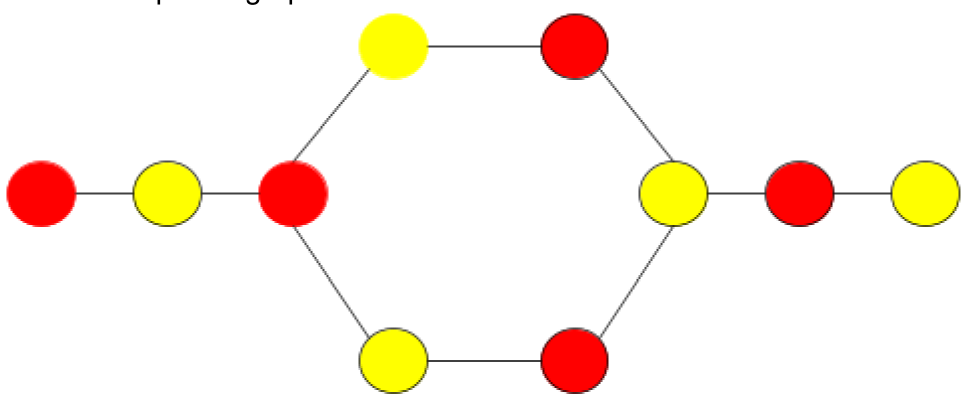
BIPARTITE GRAPH

Intuition

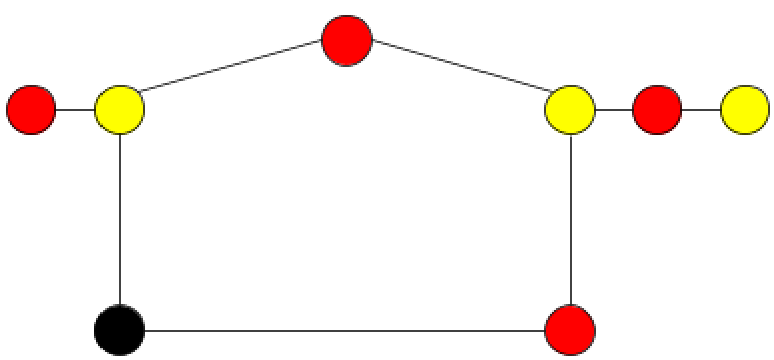
Definition : If you can color the graph with two colors such that no adjacent nodes have the same color then the graph is Bipartite graph.

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

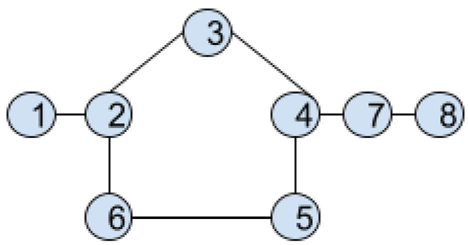
- The linear graphs with no cycles are always bi-partite graphs
- If the graph has a cycle then :
    - If cycle length is even than it can also be bi-partite

- Else it's not a bi-partite Graph

This can be solved using a BFS algorithm

We will require a queue data structure to solve and also a colored array where its initialized with -1 so that represents all are uncolored in the start

eg.



// Initial Configuration
Queue = [0]

	0	1	2	3	4	5
Colored = [	-1	-1	-1	-1	-1	-1

6	7	8
-1	-1	-1]

Adjacency List :

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

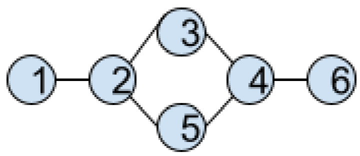
Traversing

Queue = [1]
[2]
[3,6]
[6,4]
[4,5]

Color = **[0,1,0,1,1,0,-1,-1]**

As soon as we traverse the node 4 we have its neighbor in form of 3,5,7 ie. 5 has the same color as the node 4 therefore it will return the control immediately with returning a bool variable as false because its not a bi-partite graph.

Eg.



Q = [1]
[2]
[3,5]
[5,4]
[4]
[6]

Colored = **[0,1,0,1,0,1**]

Therefore it is a bi-partite graph

Approach

IsBipartite()
- Declare :
    - Color array of v size and initialize it with -1
- Traverse through all the nodes : (in order to have all components)
    - Check if the node is not colored :
        - Call check function as (source,v,adj,color) and then check in condition if its false :
            - Return false because in this case the two adjacent nodes will end up with same color
- Outside the complete function return true

Check()
- Declare :
    - Empty queue
- Add source node to the queue
- Color the source node with color 0
- Traverse until the queue becomes empty :

        - Extract the first node
        - Pop the first node
        - Traverse through the adjacent elements of the node :
            - Check if the node is uncolored :
                - Color it with opposite color of the node
                - Push it to the queue
            - Check if the node is colored and is of the same color as the node :
                - Return false because in this case the bi-partite graph is not possible
    - Return true outside the loops

Function Code

```
bool check(int start,int v,vector<int>adj[],int color[])
    {
        // Declare
        // Empty queue
        queue<int> q;
        // pushing the source to the queue
        q.push(start);
        // coloring the source to be 0
        color[start] = 0;

        // traversing until the queue becomes empty
        while(!q.empty())
        {
            // extracting the first element of the queue
            int node = q.front();
            // popping the first element from the queue
            q.pop();
            // traversing the adjacent element of the node
            for(auto it:adj[node])
            {
                // checking if the adjacent node is uncolored
                if(color[it]==-1)
                {
                    // coloring it with the opposite color of the original
node
                    color[it] = !color[node];
                    // pushing the adjacent node to the queue
                    q.push(it);
                }
                // checking if the node is colored and is same color as the
original node
```

```
                else if(color[it]==color[node])
                {
                    // it's not possible to color it now therefore
returning false
                    return false;
                }
            }
        }

        // returning true because the component can be colored in a
bi-partite method
        return true;
    }
      bool isBipartite(int v, vector<int>adj[]){
          // Declare
          // Array of size v named color initialized with -1
          int color[v];
          // initializing color
          for(int i=0;i<v;i++)color[i] = -1;
          // traversing for all graph components
          for(int i=0;i<v;i++)
          {
              // checking if a node is uncolored
              if(color[i]==-1)
              {
                  // checking if the node can not be colored in a
bi-partite manner
                  if(check(i,v,adj,color)==false)
                  {
                      return false;
                  }
              }
          }
          // graph can be colored in bi-partite format
          return true;
      }
```

Time Complexity

O(N*E)