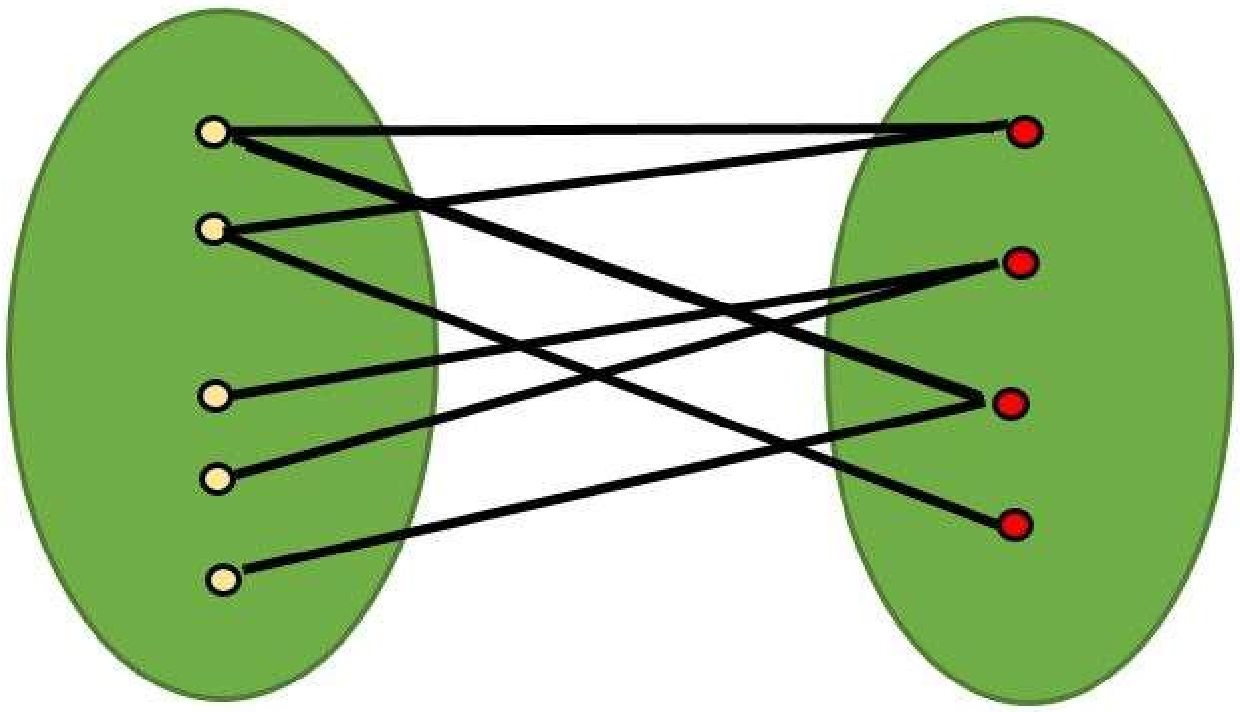
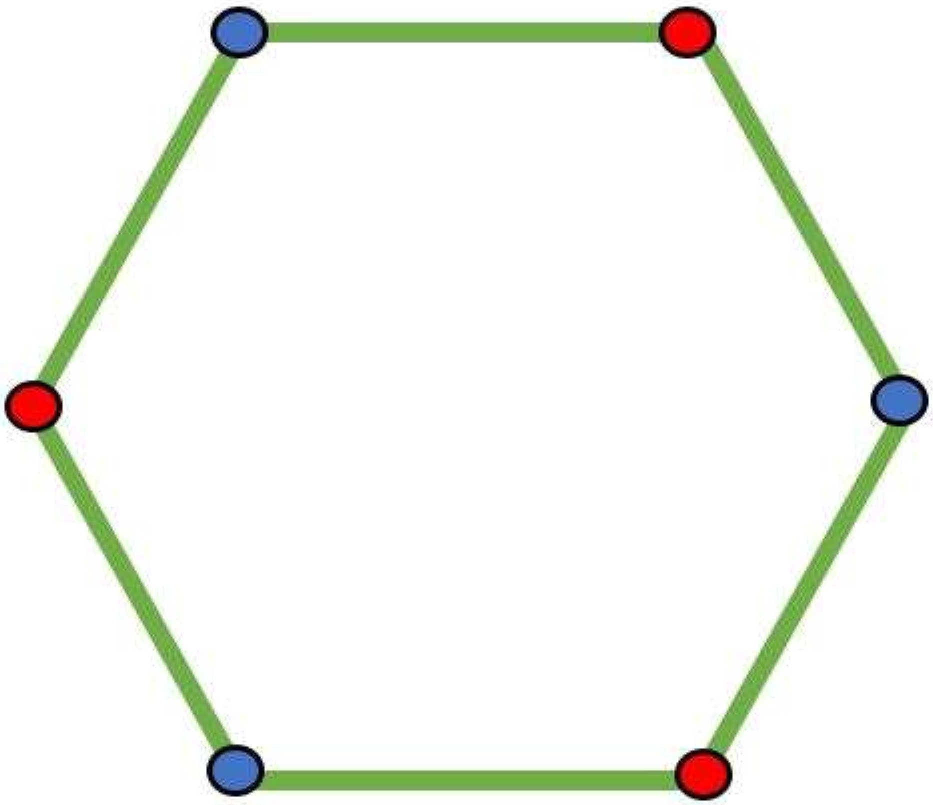
Check whether a given graph is Bipartite or not

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.

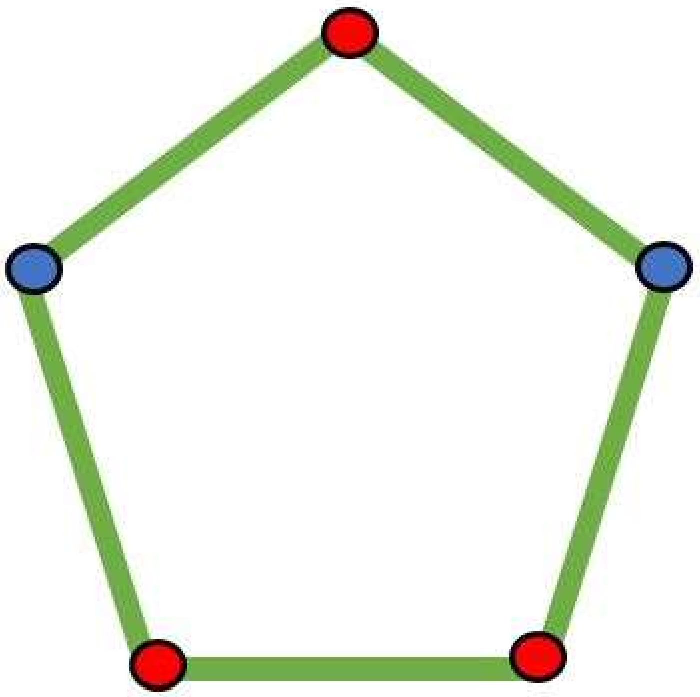


A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



Cycle graph of length 6

It is not possible to color a cycle graph with odd cycle using two colors.



Cycle graph of length 5

Algorithm to check if a graph is Bipartite:

One approach is to check whether the graph is 2-colorable or not using backtracking algorithm m coloring problem.

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Recommended Problem

```cpp
// C++ program to find out whether a
// given graph is Bipartite or not
#include <iostream>
#include <queue>
#define V 4

using namespace std;

// This function returns true if graph
// G[V][V] is Bipartite, else false
bool isBipartite(int G[][V], int src)
{
    // Create a color array to store colors
    // assigned to all vertices. Vertex
    // number is used as index in this array.
    // The value '-1' of colorArr[i]
    // is used to indicate that no color
    // is assigned to vertex 'i'. The value 1
    // is used to indicate first color
    // is assigned and value 0 indicates
    // second color is assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex
    // numbers and enqueue source vertex
    // for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices
    // in queue (Similar to BFS)
    while (!q.empty())
    {
        // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
        int u = q.front();
        q.pop();
        // Return false if there is a self-loop
        if (G[u][u] == 1)
        return false;

        // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
        {
            // An edge from u to v exists and
            // destination v is not colored
            if (G[u][v] && colorArr[v] == -1)
            {
                // Assign alternate color to this adjacent v of u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }

            // An edge from u to v exists and destination
            // v is colored with same color as u
            else if (G[u][v] && colorArr[v] == colorArr[u])
                return false;
        }
    }

    // If we reach here, then all adjacent
    // vertices can be colored with alternate color
    return true;
}

// Driver program to test above function
int main()
{
    int G[][V] = {{0, 1, 0, 1},
        {1, 0, 1, 0},
        {0, 1, 0, 1},
        {1, 0, 1, 0}
    };

    isBipartite(G, 0) ? cout << "Yes" : cout << "No";
    return 0;
}
```

Output

```
Yes
```

Time Complexity : O(V*V) as adjacency matrix is used for graph but can be made O(V+E) by using adjacency list
Auxiliary Space: O(V) due to queue and color vector.

The above algorithm works only if the graph is connected. In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite. Note that the Bipartite condition says all edges should be from one set to another.
We can extend the above code to handle cases when a graph is not connected. The idea is repeatedly called above method for all not yet visited vertices.

```
// C++ program to find out whether
// a given graph is Bipartite or not.
// It works for disconnected graph also.
#include <bits/stdc++.h>

using namespace std;

const int V = 4;

// This function returns true if
// graph G[V][V] is Bipartite, else false
bool isBipartiteUtil(int G[][V], int src, int colorArr[])
{
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers a
    // nd enqueue source vertex for BFS traversal
    queue<int> q;
    q.push(src);

    // Run while there are vertices in queue (Similar to
    // BFS)
    while (!q.empty()) {
        // Dequeue a vertex from queue ( Refer
        // http://goo.gl/35oz8 )
        int u = q.front();
        q.pop();
        // Return false if there is a self-loop
        if (G[u][u] == 1)
            return false;

        // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v) {
            // An edge from u to v exists and
            // destination v is not colored
            if (G[u][v] && colorArr[v] == -1) {
                // Assign alternate color to this
                // adjacent v of u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }

            // An edge from u to v exists and destination
            // v is colored with same color as u
            else if (G[u][v] && colorArr[v] == colorArr[u])
                return false;
        }
    }

    // If we reach here, then all adjacent vertices can
    // be colored with alternate color
    return true;
}

// Returns true if G[][] is Bipartite, else false
bool isBipartite(int G[][V])
{
    // Create a color array to store colors assigned to all
    // vertices. Vertex/ number is used as index in this
    // array. The value '-1' of colorArr[i] is used to
    // indicate that no color is assigned to vertex 'i'.
    // The value 1 is used to indicate first color is
    // assigned and value 0 indicates second color is
    // assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // This code is to handle disconnected graph
    for (int i = 0; i < V; i++)
```

```
        if (colorArr[i] == -1)
            if (isBipartiteUtil(G, i, colorArr) == false)
                return false;

    return true;
}

// Driver code
int main()
{
    int G[][V] = { { 0, 1, 0, 1 },
                   { 1, 0, 1, 0 },
                   { 0, 1, 0, 1 },
                   { 1, 0, 1, 0 } };

    isBipartite(G) ? cout << "Yes" : cout << "No";
    return 0;
}
```

Output

```
Yes
```

Time complexity: **O(V*V)**.
Auxiliary Space: O(V), because we have a V-size array.
If Graph is represented using Adjacency List .Time Complexity will be O(V+E).
Works for connected as well as disconnected graph.

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

bool isBipartite(int V, vector<int> adj[])
{
    // vector to store colour of vertex
    // assigning all to -1 i.e. uncoloured
    // colours are either 0 or 1
      // for understanding take 0 as red and 1 as blue
    vector<int> col(V, -1);

    // queue for BFS storing {vertex , colour}
    queue<pair<int, int> > q;
```

```
      //loop incase graph is not connected
    for (int i = 0; i < V; i++) {

      //if not coloured
        if (col[i] == -1) {

          //colouring with 0 i.e. red
            q.push({ i, 0 });
            col[i] = 0;

            while (!q.empty()) {
                pair<int, int> p = q.front();
                q.pop();

                  //current vertex
                int v = p.first;
                  //colour of current vertex
                int c = p.second;

                  //traversing vertexes connected to current vertex
                for (int j : adj[v]) {

                      //if already coloured with parent vertex color
                      //then bipartite graph is not possible
                    if (col[j] == c)
                        return 0;

                      //if uncoloured
                    if (col[j] == -1) {
                      //colouring with opposite color to that of parent
                        col[j] = (c) ? 0 : 1;
                        q.push({ j, col[j] });
                    }
                }
            }
        }
    }
    //if all vertexes are coloured such that
      //no two connected vertex have same colours
    return 1;
}
```

```
// { Driver Code Starts.
int main()
{

    int V, E;
    V = 4 , E = 8;
      //adjacency list for storing graph
    vector<int> adj[V];
      adj[0] = {1,3};
      adj[1] = {0,2};
      adj[2] = {1,3};
      adj[3] = {0,2};


    bool ans = isBipartite(V, adj);
    //returns 1 if bipartite graph is possible
      if (ans)
        cout << "Yes\n";
    //returns 0 if bipartite graph is not possible
      else
        cout << "No\n";

    return 0;
}
 // code Contributed By Devendra Kolhe
```

Output

```
Yes
```

Time Complexity: O(V+E)
Auxiliary Space: O(V)