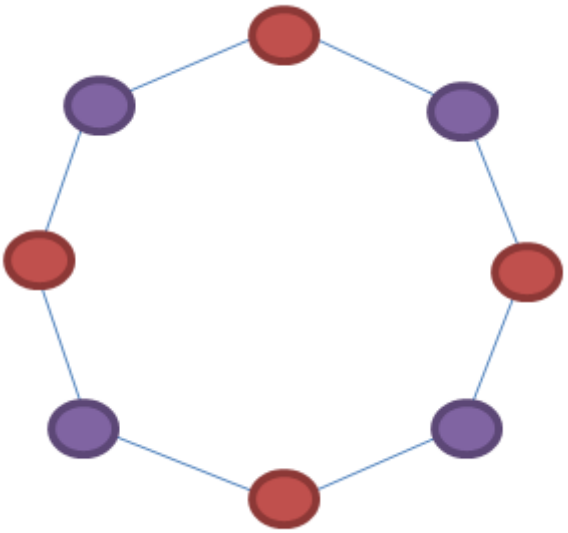
Checking if a graph is bipartite

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 an even cycle using two colors. For example, see the following graph.



It can be easily noticed that it is not possible to color a graph with an odd cycle with two colors.
One approach to check if a graph is bipartite or not is by checking if it is 2 colorable or not.
But here we will discuss another simple solution.

Algorithm

- Initialize a set array to -1 of size equal to total vertices.
- Set set[src]=1.
- Initialize a queue q and add source vertex to the q.
- Do the following till the q is not empty
    - Dequeue the vertex u
    - Return false if there is an edge from u to u, that is self-loop
    - Iterate over the adjacent vertices of u, for each vertex v adjacent to u, do the following
        - If the set[v] equals -1, which means if not colored then color the vertex and add it to the queue.
        - Else check if set[v] equals set[u], then return false.
- Return true

```
function isBipartite(Graph G, src)
{
                    //   declare a set array of size V and initialize all the elements of it as -1.
                    set[V];
                    i=0;
```

```
                while(i < V)
                        set[i]=-1
                        i++;


                //   Assign color to src
                set[src] = 1

                //   Create a queue and add src vertex to it for BFS traversal
                q.enqueue(src)

                 //   Loop while q is not empty
                while(q is not empty)  {

                        //   Dequeue vertex from the queue
                         u = q.dequeue();

                        //   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]==1 && set[v]==-1)
                                {
                                         //   Assign alternate color to this adjacent v of u
                                        set[v] = 1-set[u];
                                         q.enqueue(v);
                                 }

                                  /*
                                         An edge from u to v exists and destination
                                         v is colored with the same color as u
                                 */
                                  else if (G[u][v]==1 && set[v]==set[u])
                                        return false;
                        }
                }
                /*
                        If we reach here, then all adjacent vertices can
                        be colored with alternate color
                */
                 return true;
}
```

Time Complexity: Time Complexity of the above approach is the same as that of Breadth-First Search. For the above implementation, it is O(V^2) where V is a number of vertices. If the graph is represented using an adjacency list, then the complexity becomes O(V+E).