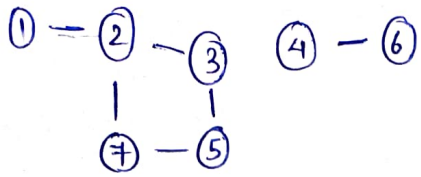
- **Depth first Search**




```
Void dfs (node, ans, adj, vis) {
    vis[node] = 1;
    ans.push_back(node);
    for (auto it : adj[node])
        if (!vis[it])
            dfs (it, ans, adj, vis);
}

vector<int> dfsAns (int v, adj) {
    ans, vis (v, 0);
    for (i → v)
        if (!vis[i])
            dfs (i, ans, adj, vis);
    return ans;
}
```

Adjacency List

1	2
2	3, 7
3	5, 2
5	3, 7
7	2, 5
4	6
6	4

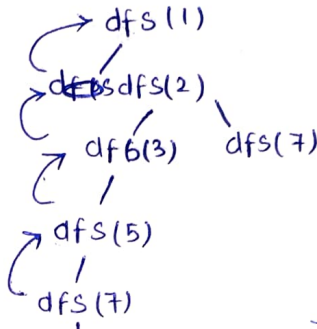
Answer (DFS)

1	2	3	5	7	4	6

Visited array

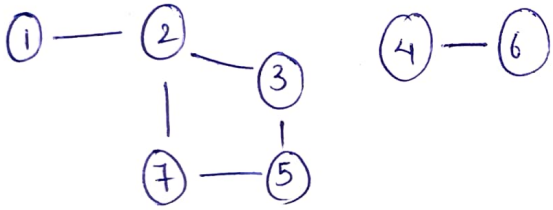
X	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

In DFS, we traverse till the ~~detect~~ depth and till our recursion stack gets empty.
T.C → O(n) + O(n) } Not considering adj. List.
S.C → O(n)



Recursion Tree for DFS Traversal.

- **Breadth first Search**



Lets say,
V = 7, e = 6

```
for (i=1; i<=7; i++) {   ← for all the components of graph.
    if (!vis[i]) {
        queue<int> q;
        q.push(i);
        vis[i] = 1;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            ans ← push_back(i);
            for (auto it : adj[node]) {
                if (!vis[it]) {
                    q.push(it);
                    vis[it] = 1;
                }
            }
        }
    }
```

just return our answer array.

Adjacency List

1	2
2	3, 7
3	2, 5
5	3, 7
7	2, 5
4	6
6	4

Answer (BFS)

1	2	3	7	5	4	6

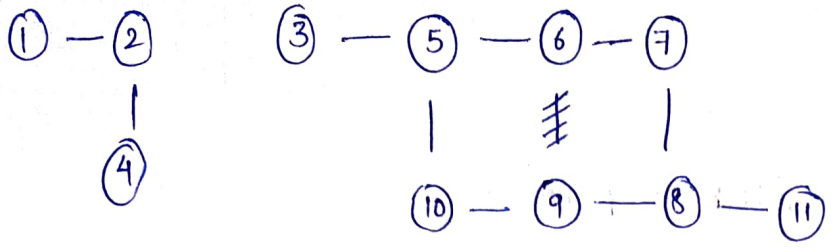
Visited array

X	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7

In BFS, first check all the adjacent nodes of current node and then move further for other nodes.

T.C → O(V) } Not considering Adjacency List.
S.C → O(V) }

• Cycle Detection in Undirected Graph (Using BFS)

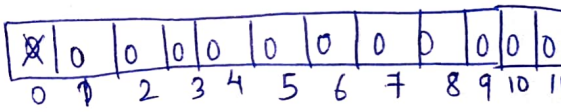


Here, to detect a cycle while traversing through BFS, we have to create a pair of node and prev. node to keep a track, so, if the adjacent node is already visited, then it should ignore prev. node as this is an undirected graph.
And if the adjacent node is visited as well as not prev. node, it means there is a cycle exists otherwise NO.

adj list

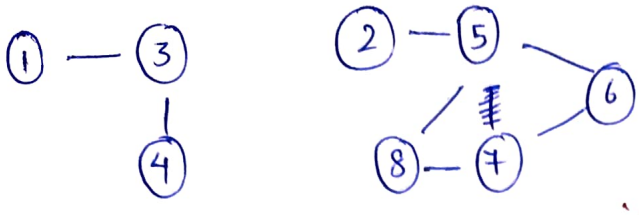
1	2
2	1, 4
4	2
3	5
5	6, 10
6	7, 5
10	5, 9
7	6, 8
9	10, 8
8	7, 9, 11
11	8

visited array




```
bool check(int V, vector<int> adj[]) {
    vis(V,0);
  for(i → v) {
    if(!(vis[i])){
        queue<pair<int,int>> q;
        q.push({i,-1});
        vis[i] = 1;
        while (!q.empty()) {
           int node = q.front().first; int par = q.front().second;
           q.pop();
           for (auto it : adj) {
              if (!vis[it]) {
                 q.push ({it, node});
                 vis[it] = 1;
              }
              else if (it != par) return true;
           }
        }
     }
   }
   return false;
}
```

- Cycle Detection in Undirected graph (using DFS)



adj. list

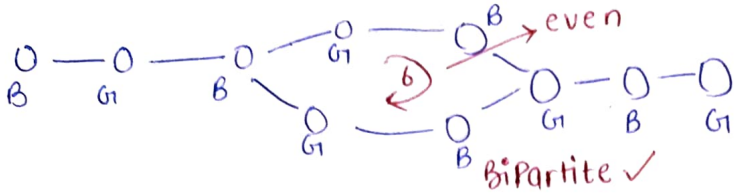
1	3
3	4
4	3
2	5
5	8, 6
8	7, 5
7	6, 8
6	5, 7

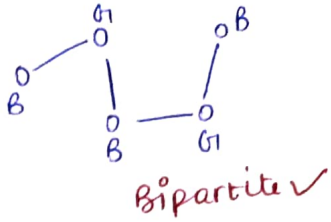
The same idea or approach will be implemented here as BFS technique.
We will use parent node and will check if the adjacent node is visited as well as its not a parent node, then there exists a cycle otherwise NO.

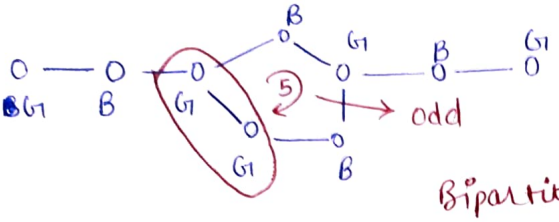
```
bool check(int node, parent, adj, vis){
    vis[node] = 1;
    for(auto it : adj[node]){
        if(vis[it] == 0)
            if(check(it, node, vis, adj)) return true;
        else if(it != par) return true;
    }
    return false;
}
bool isCycle(V, adj[]){
    vis(V,0);
  for(i → V)
      if(!vis[i])
          if(check) return true;

  return false;
}
```

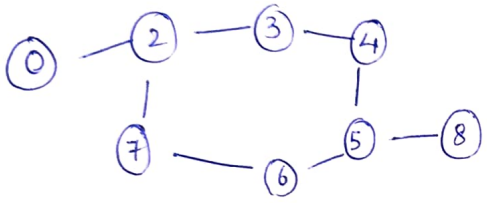
- **Bipartite Graph (BFS)**

Graph that can be coloured using 2 colors such that no two adjacent nodes have same color.









visited array

X	-1	-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6	7	8

Approach :

- Take 2 diff. colors - 0 and 1
- Marks node's colors as 0 or 1 after checking its adjacent nodes.
- If we get 2 adjacent nodes of same color, just return True. else False.

```
bool isBipartite(V, adj) {
    color(v, -1);
  for(i → v)
    if (color[i] == -1) {
      color[i] = 0;
      q.push(i);
      while(!q.empty) {
        int node = q.front();
        q.pop();
        for(auto it : adj[node]) {
          if(color[it] == -1) {
            color[it] = 1 - color
                        [node];
            q.push(it);
          }
          elseif(color[it]
                 == color[node])
            return false
        }
      }
    }
    return true;
}
```

- **Bipartite Graph (DFS)**

```
bool check(node, adj, color) {
    if(color[node] == -1)
        color[node] = 1;
  for(auto it : adj[node]) {
      if(color[it] == -1) {
        color[it] = 1 - color[node];
        if(check(it, adj, color)) return true;
      }
      else if(color[it] == color[node])
          return false;
  } return true;
}
bool isBipartite(V, adj) {
  color(v, -1);
  for(i → v)
    if(color[i] == -1)
      if(isBipartite(i, adj, color)) return true;
  return false; }.
```