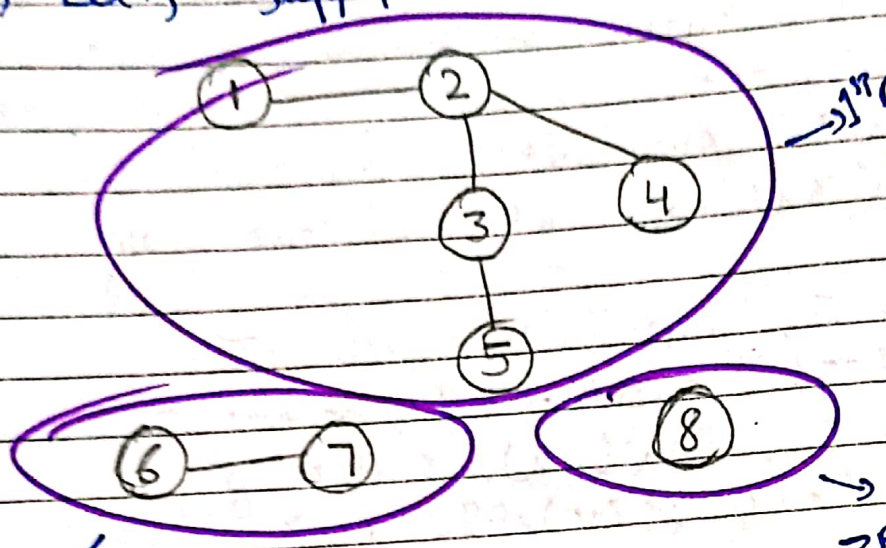
HOW TO FIND CONNECTED COMPONENTS AND CYCLE IN A GRAPH USING DFS

1) Connected Components:

→ Let's suppose we have this graph:



This kind of graphs sometimes usually called ~~Trees~~ Forest.

∴ It has 3 connected components as we can see.

APPROACH: We will run DFS on every node (only if that node isn't in visited array), ~~As soon~~ And the number of times our DFS runs ~~on given~~ will be our count of connected components.

e.g. Let's suppose we start with a loop of 1 - 8 (our vertices). First our iteration will run for 1 and as it is not in visited array DFS will run on 1 and mark all

its connected nodes as visited.
e.g: here VISITED = 1, 2, 3, 5, 4

Now, same process will be for 6 and 8.

And ∴ our count of connected components will be 3 here.

CODE

→ Maximum capacity of vertices
```
CONST INT N = 1e5+10;
```
→ Adjacency list
```
VECTOR<INT> G[N];
```
→ Visited array
```
BOOL VIS[N];
```

→ Vector to store connected components
```
VECTOR<VECTOR<INT>> CC;
```
→ To put in CC
```
VECTOR<INT> CURRENT_CC;

VOID DFS(INT VERTEX)
{
    VIS[VERTEX] = TRUE;
    CURRENT_CC.PB(VERTEX);
    FOR (INT CHILD : G[VERTEX])
    {
        IF (VIS[CHILD]) CONTINUE;
        DFS(CHILD);
    }
```

```
}

INT MAIN()
{
    INT n, e;
    CIN >> n >> e;

    -> creating Adjacency list representation
       of graph
    FOR (INT i=0; i<e; i++)
    {
        INT V1, V2;
        CIN >> V1 >> V2;
        g[V1].PB(V2);
        g[V2].PB(V1);

    }
    -> To store count of connected components
    INT C=0;
    FOR (INT i=1; i<=n; i++)
    { -> To run DFS only for unvisited nodes
        IF (VIS[i]) CONTINUE;
        -> clearing CURRENT_CC before running DFS
        CURRENT_CC.CLEAR();
        DFS(i);
        -> storing each connected components
        CC.PB(CURRENT_CC);
        C++;
    }

    -> Printing no. of connected components
```

```
cout << cc.size() << "\n";

-> Printing each connected components
for (auto c_cc : cc)
{
    for (auto vertex : c_cc)
    {
        cout << vertex << " ";
    }
    cout << "\n";
}

return 0;
}
```

Input:

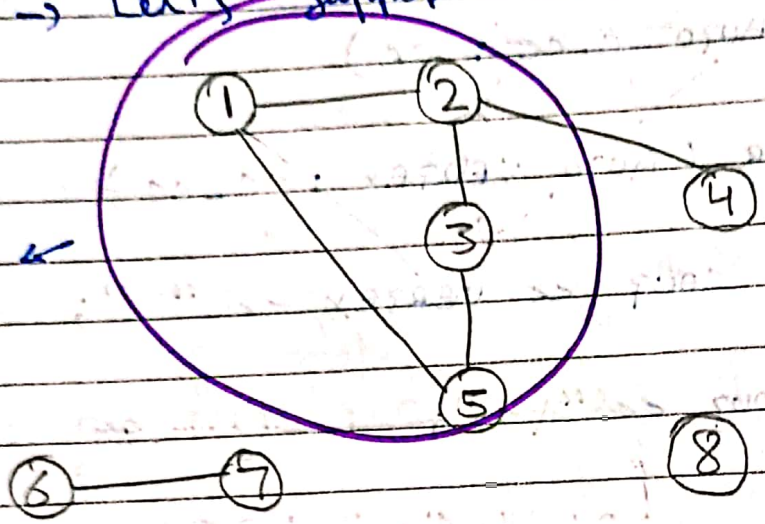
```
8 5
1 2
2 3
2 4
3 5
6 7
```

Output:

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

2) Cycle :

→ Let's suppose we have this graph:

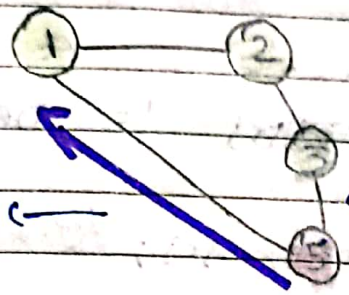


It has a cycle 1, 2, 3, 5, 1

APPROACH: Let's suppose we are running DFS, from ① and we have reached at ⑤. Now, ⑤ will try to access its child node, and if it try to visit ① (which is already visited) then we can say that a loop is formed here.
But the issue is ① isn't the only child node of ⑤. ③ is also its child node. So, ⑤ will try to visit ③ as well and it is also already visited but it doesn't ensure that a loop is formed here.
So, to prevent this, we will keep track of parent node as well.
∴ The final condition will be:
If child node is already visited and

it is not the parent of current node then it ensures a cycle.

e.g



cycle as ① isn't parent of ⑤

→ No cycle as ③ is parent of ⑤

NOTE: Here, parent == Last visited node (PREVIOUS)

CODE

```
-> Maximum capacity of vertices
CONST INT N = 1e5 + 10;
-> Adjacency list
VECTOR<INT> G[N];
-> Visited array
BOOL VIS[N];

BOOL DFS(INT VERTEX, INT PAR)
{
    VIS[VERTEX] = true;
    BOOL ISLOOPEXISTS = false;
    FOR (INT CHILD : G[VERTEX])
    {
        IF (VIS[CHILD] && CHILD == PAR)
            CONTINUE; // Skipping visited node
                      which is parents of current
                      child.
```

```
        IF (VIS[CHILD]) RETURN true; // if visited node isn't parent of
                                     current child, it means cycle
                                     exists.
        ISLOOPEXISTS |= DFS(CHILD, VERTEX);
    }
    RETURN ISLOOPEXISTS;
}

INT MAIN()
{
    INT n, e;
    CIN >> n >> e;

    -> Creating Adjacency list representation
       of graph.
    FOR (INT i=0; i<e; i++)
    {
        INT V1, V2;
        CIN >> V1 >> V2;
        G[V1].PB(V2);
        G[V2].PB(V1);
    }

    BOOL ISLOOPEXISTS = false;
    FOR (INT i=1; i<=n; i++)
    {
        IF (VIS[i]) CONTINUE; // To run
            DFS for only unvisited nodes.
        IF (DFS(i, 0))
        {
```

```
            isLoopExist = true;
            BREAK;
        }
    }
    cout << isLoopExist << "\n";

    RETURN 0;
}
```

INPUT:

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

OUTPUT:

1