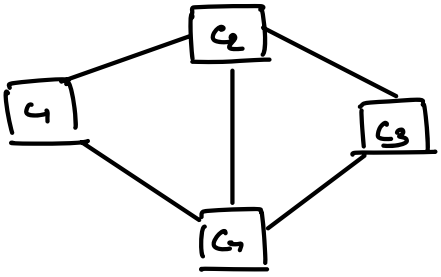
Graphs 1

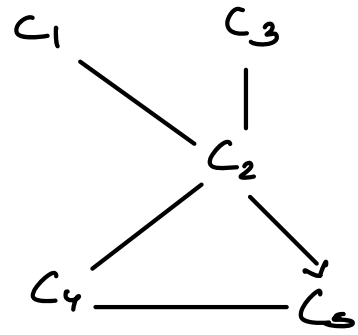
Good Evening

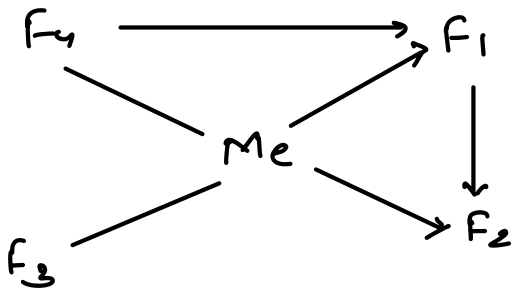
* Agenda for Today

01. Introduction
02. Types of Graph
03. How to store a graph
04. Depth first search (DFS)
05. Breadth first search (BFS) → level order
06. Check if cycle is present

* Graph → Network

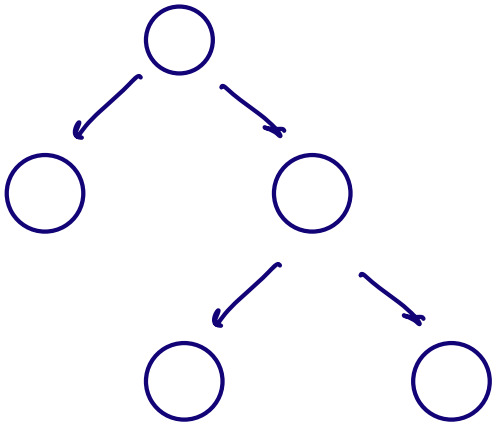






Graph → Collection of nodes & edges

vertices



01 Every tree is a graph ✓

02 Every graph is a tree ✗

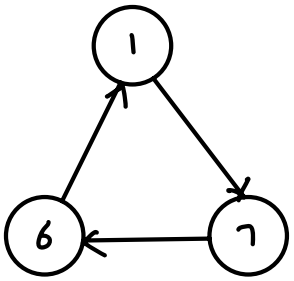
01. N nodes → n-1 edges

02. Trees always have one root node

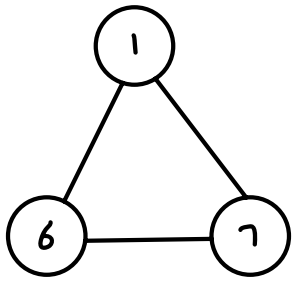
03. Cycle is not possible in trees

* Classification of Graphs

01. Based on type of edges

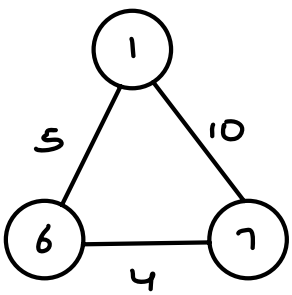


directed graph

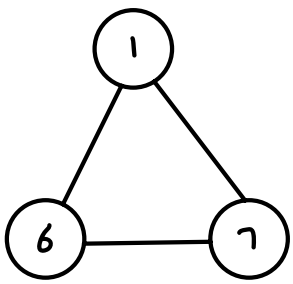


undirected graph

02. Based on edge wt present or not

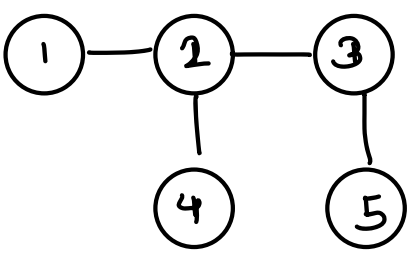


weighted graph

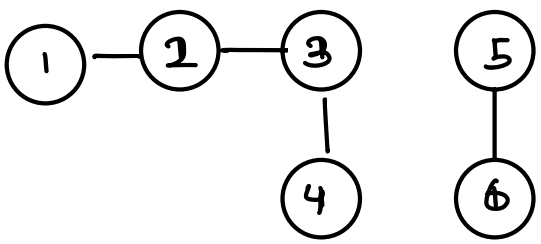


unweighted graph

03. Based on no. of components

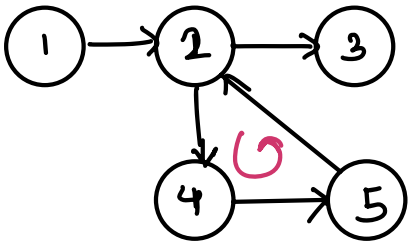


Connected

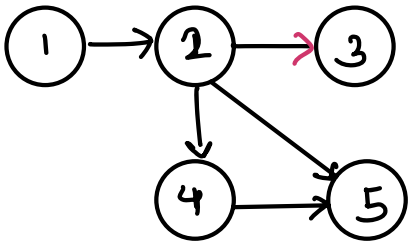


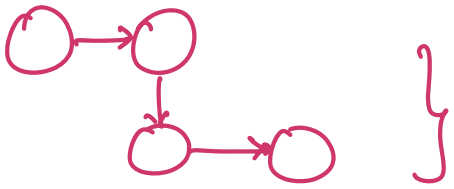
Disconnected

04. Cyclic

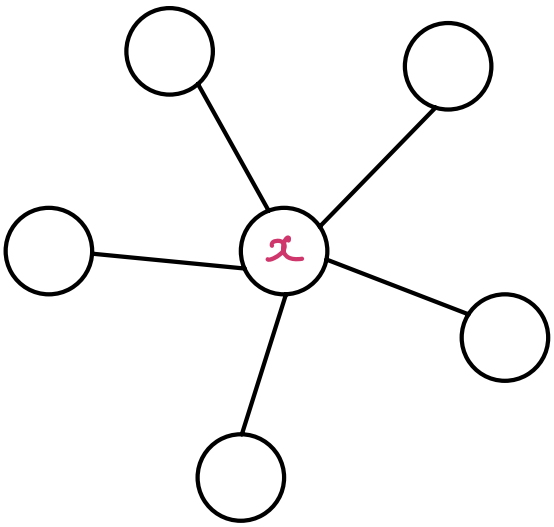


Acyclic





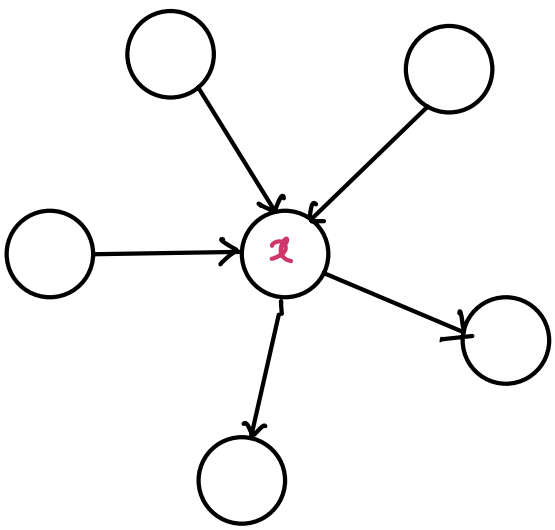
05. Degree



Degree (x) = 5

No. of edges it has

Indegree / Outdegree



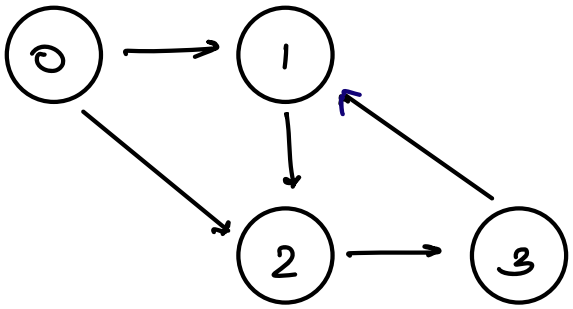
Indegree(x) = 3

Outdegree (x) = 2

* **Simple graph**

Undirected connected graph without self loop & multiedges

* How to store graphs



4 nodes

5 edges

0	1
0	2
1	2
2	3
3	1

01. Adjacency matrix

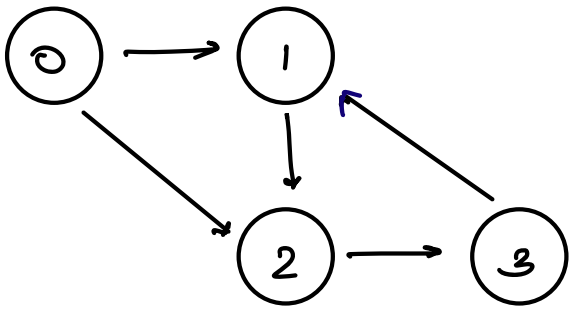
	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

mat[i][j] = 1 edge from i→j

mat[i][j] = 0] no edge b/w i & j

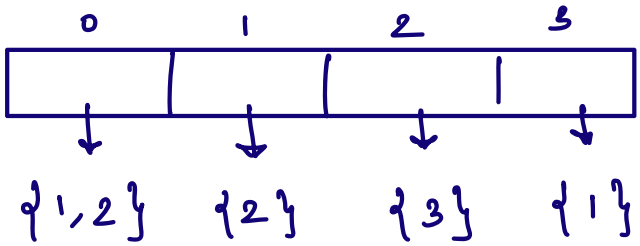
SC: n^2

* Adjacency list (array of lists)



No. of nodes = n

No. of edges = e



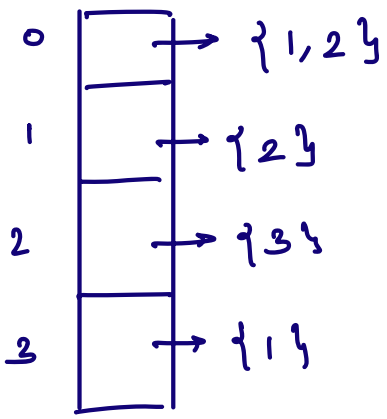
SC: (V + E)

graph[i] → list of all the nbrs from i^th node

* N = no. of nodes = 4
  E = no. of edges = 5

edges

0	1
0	2
1	2
2	3
3	1



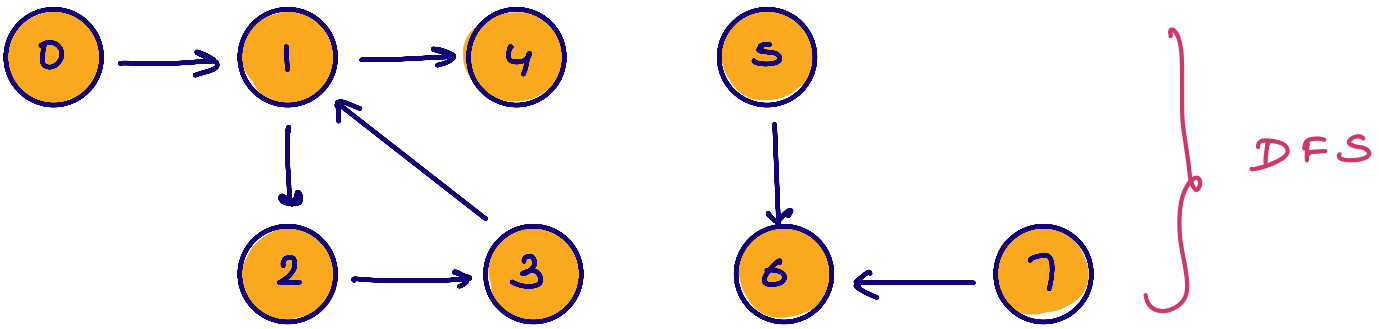
size of arr

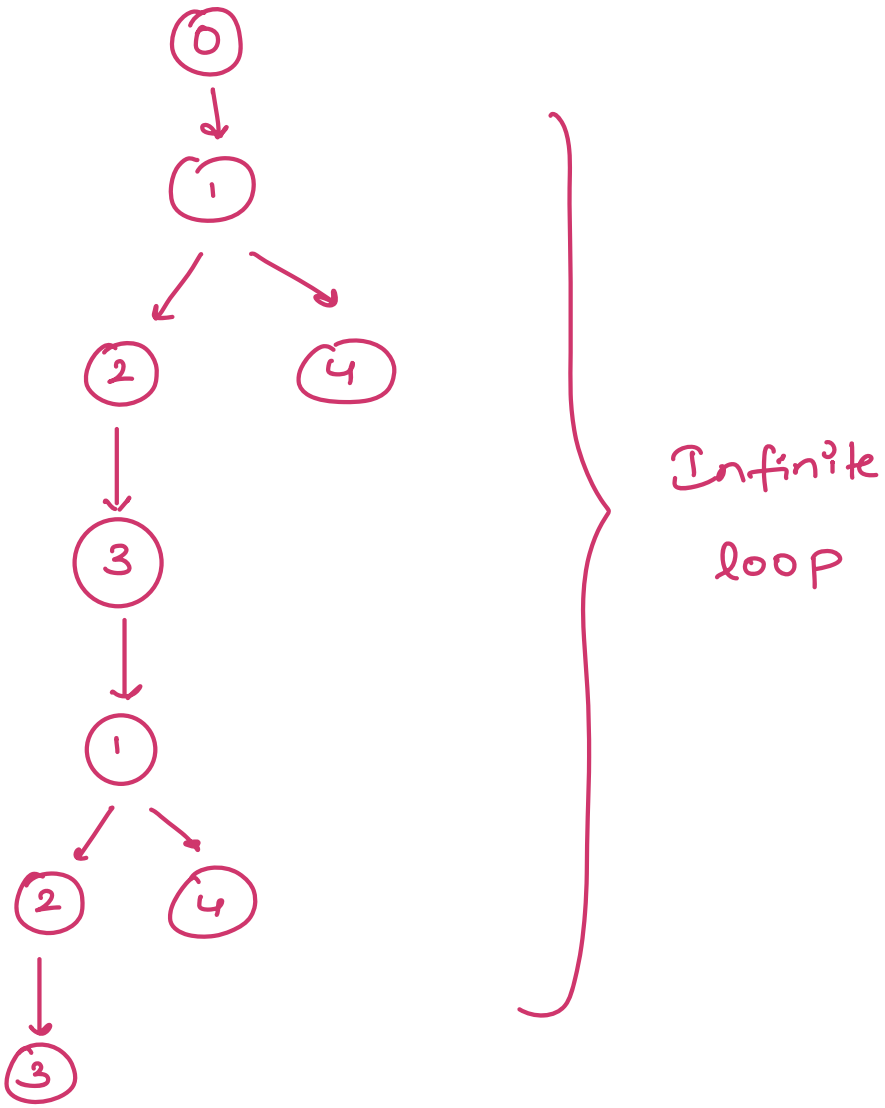
```
list<Integer> graph[N];
```

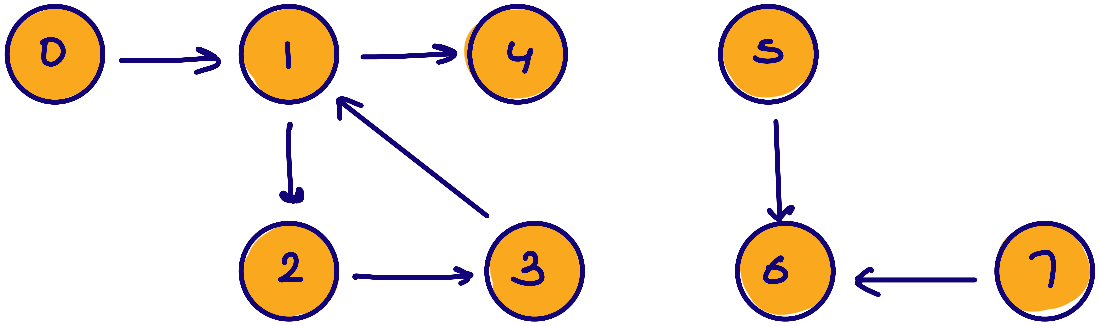
Type of data — name of arr

```
for(i=0; i<E; i++){
    int u = edges[i][0];
    int v = edges[i][1];
    graph[u].add(v);
}
```

* Traversals on Graphs



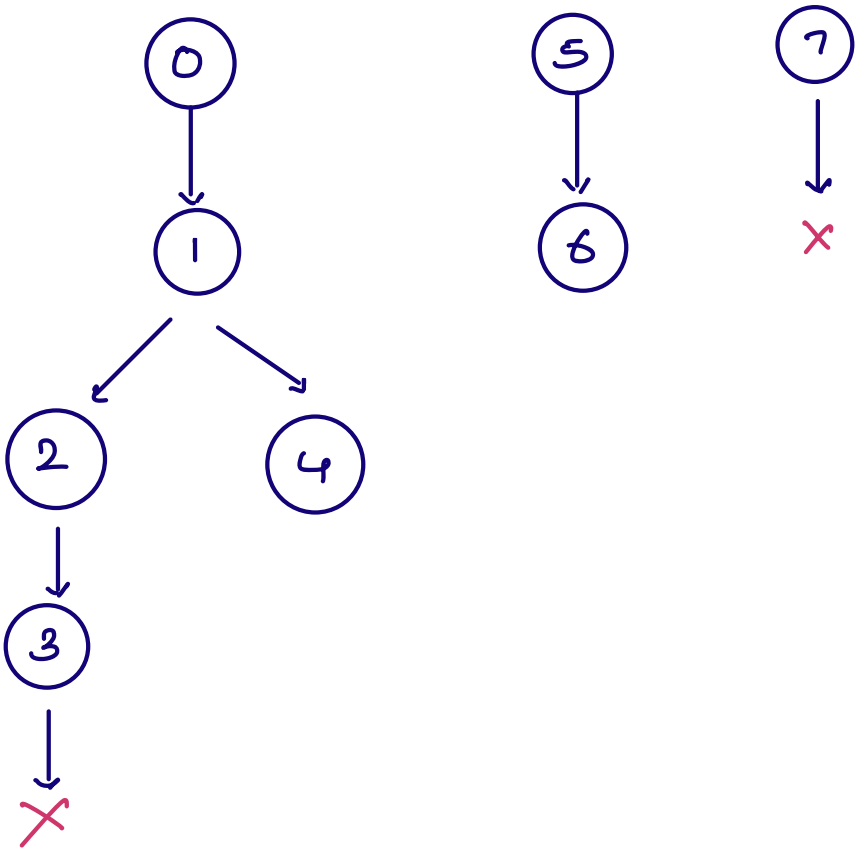




{ vertices
Nodes

* vis arr =

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



code

graph → given

boolean [ ] vis = new boolean [N];

```
for ( i = 0 ; i < N ; i++ ){
    if ( ! vis[i] ){
        dfs( graph , i , vis )
    }
}
```

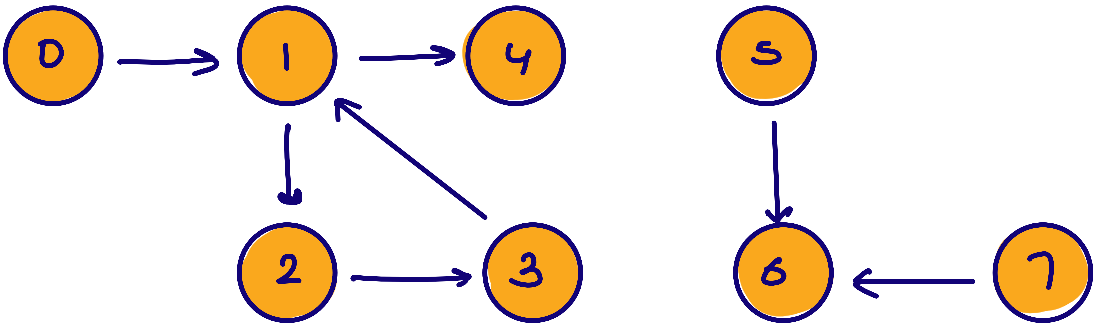
TC : O(V + E)

SC : O(V)

```
dfs ( graph , src , vis )
    print ( src ) ;
    vis[src] = true ;
    list<Integer> al : graph[src]
    for( int nbr : al )
        if ( ! vis[nbr] ){
            dfs ( graph , nbr , vis )
        }
}
```

10 : 12 pm → 10 : 22 pm

* BFS → level order Traversal



0 → { 1 }

1 → { 2, 4 }

2 → { 3 }

3 → { 1 }

4 → { }

5 → { 6 }

6 → { x }

7 → { 6 }

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

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

0 1 2 4 3 5 6 7

code

graph → given → no. of vertices

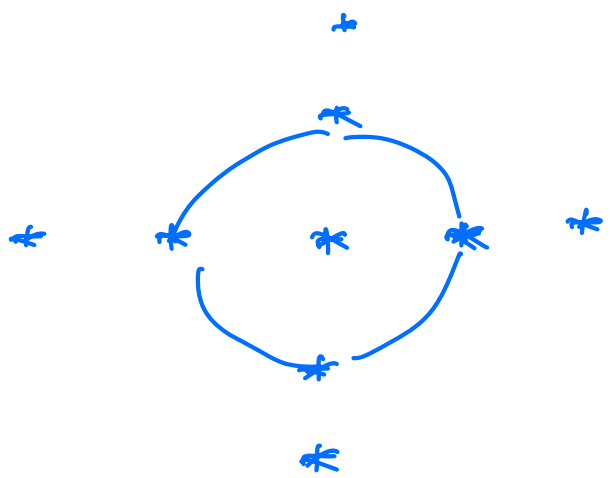
```
boolean vis[V];
for ( i=0; i<V ; i++) {
    if ( ! vis[i]) {
        bfs(graph, i, vis);
    }
}
```

```
bfs(graph, src, vis){
    Queue<Integer> q;
    q.enque(src);
    vis[src] = True;
    while(q.size()>0){
        x = q.remove();
        print(x);
        for(int nbr: graph[x]){
            if(!vis[nbr]){
                q.enque(nbr);
                vis[nbr]=True;
            }
        }
    }
}
```

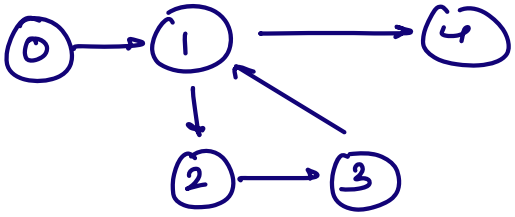
TC: O(V+E)

SC: O(V)

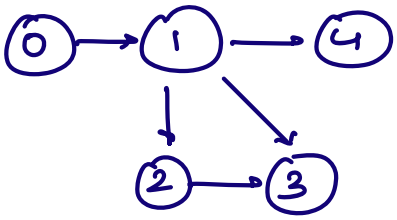
BFS → move radially



Q Check if given directed graph has cycle or not.



True



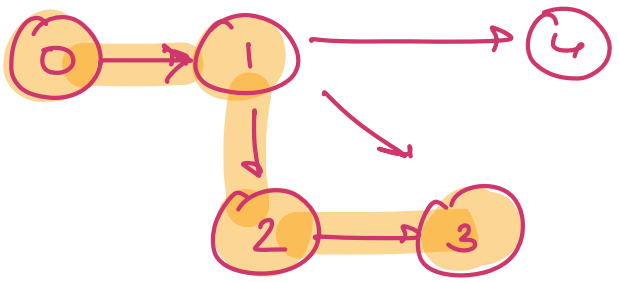
If a visited node is encountered again → cycle ✗

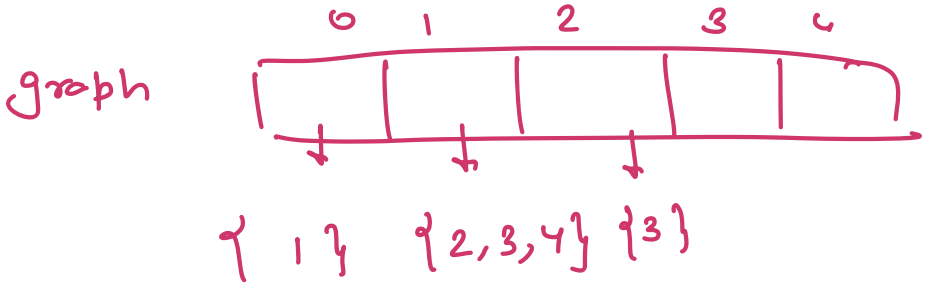
If a visited node in current path is → cycle encountered again

```
boolean [] vis = new boolean[v];

for (i=0; i<v; i++){
    if (!vis[i]){
        boolean ans = dfs(graph, i, vis, path)
        if (ans == true) return true;
    }
}
return false;
```

```
boolean dfs(graph, src, vis){
    vis[src] = True;
    path[src] = True;
    for(int nbr : graph[src])
    {
        if(path[nbr] == true){
            return true;
        }
        if(!vis[nbr] && dfs(graph, nbr, vis)==T)
            return true;
    }
    path[src] = false;
    return false;
}
```





vis = T T T T _
0 1 2 3 4

path = T T T T _
0 1 2 3 4