Graphs 1

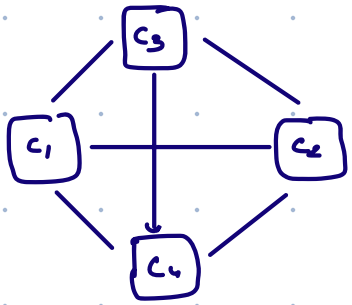
Good Evening

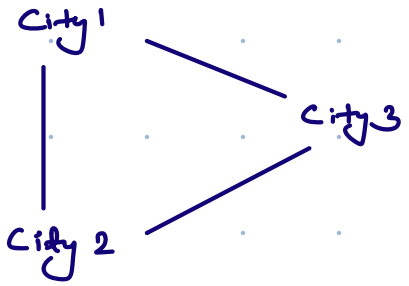
* Agenda for Today

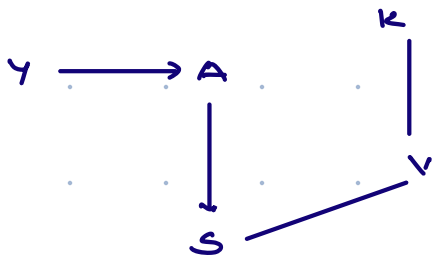
01. Introduction to Graph
02. Types of Graph
03. How to store a graph
04. Depth first search (DFS)
05. Detect cycle in Directed graph

Graph → Collection of nodes & edges

Network



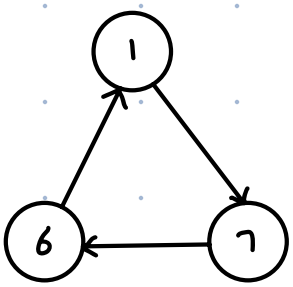




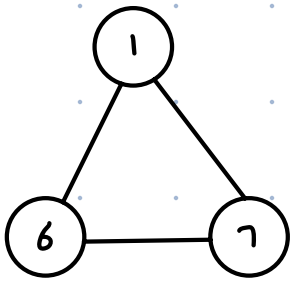
* Tree is a special grap where for N nodes N-1 edges.

* Classification of graphs

01. Type of Edges

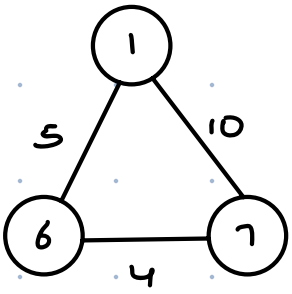


Directed graph
(Unidirectional)

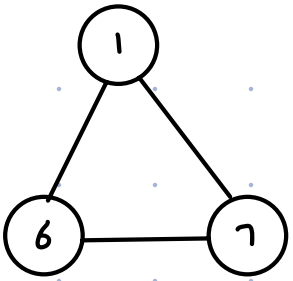


Undirected graph
(Bidirectional)

02 Edge weight present or not

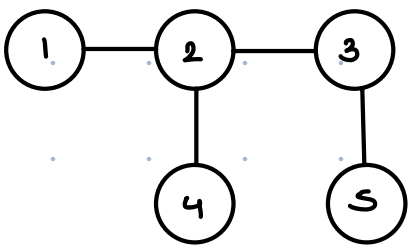


weighted graph



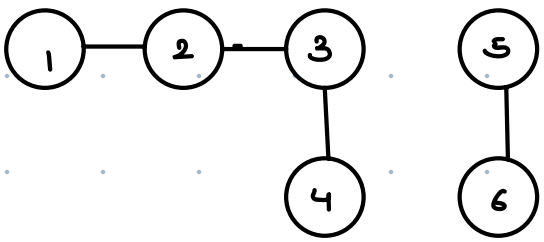
unweighted graph

03. No. of components



1 component

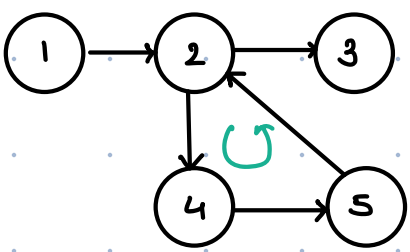
(connected)



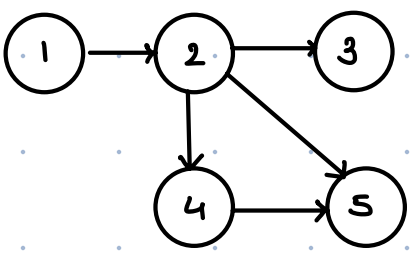
2 component

(Disconnected graph)

04 Cyclic or Acyclic

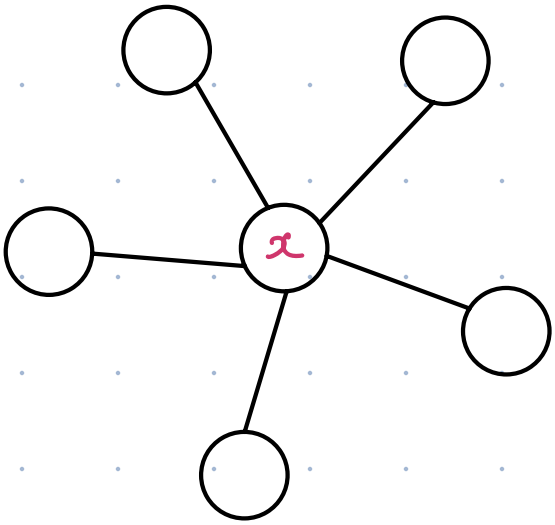


Cyclic



Acyclic

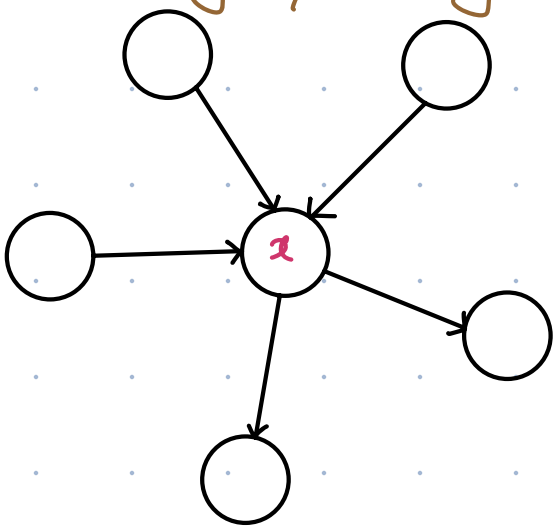
* Degree of graph



Degree (x) = 5

(No. of edges of x)

Indegree / Outdegree



Indegree (x) = 3

(No. of edges pointing towards x)

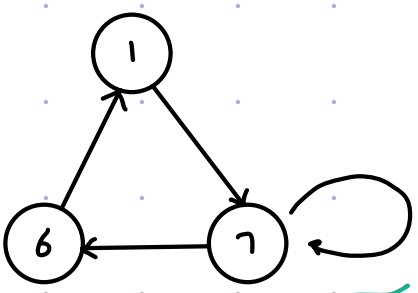
Outdegree (x) = 2

(No. of edges pointing away from x)

* Simple graph

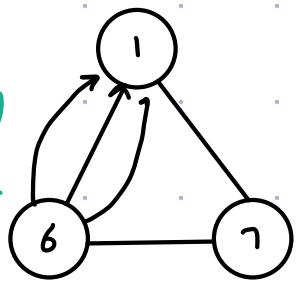
→ No self loops

→ NO multiple edges b/w two vertices

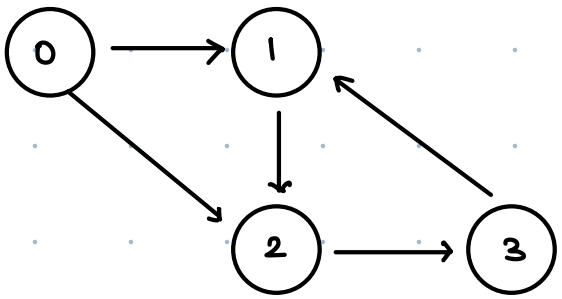


Self loop

multiple edges



* How to store a graph



Edges

u	v
0	1
0	2
1	2
2	3
3	1

4 nodes
5 edges

* 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

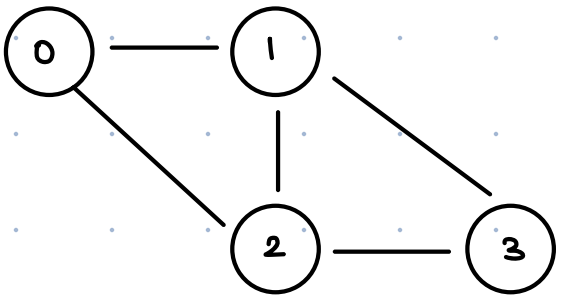
edge $i \rightarrow j$

$\downarrow$

mat[i][j] = 1

```
int [][] mat = new int[n][n]      // n → no. of nodes

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



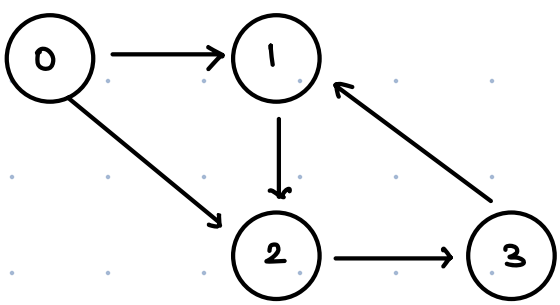
	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0

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

Adv → Easy to update new edges

Disadvantage → space wastage

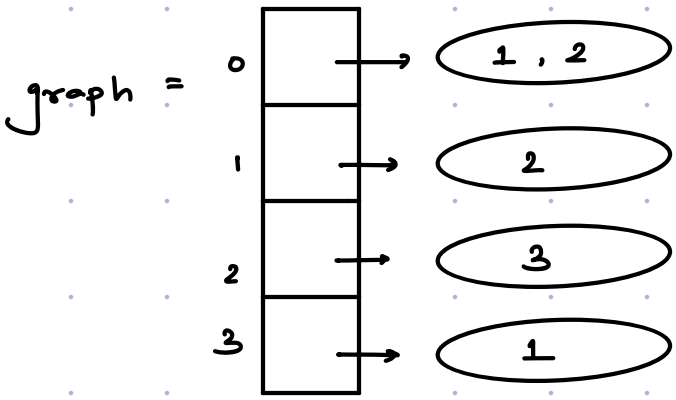
* Adjacency list (Array of Arraylist)



u	v
0	1
0	2
1	2
2	3
3	1

4 nodes

5 edges



graph[i] = list of all the nbrs of 'i' node

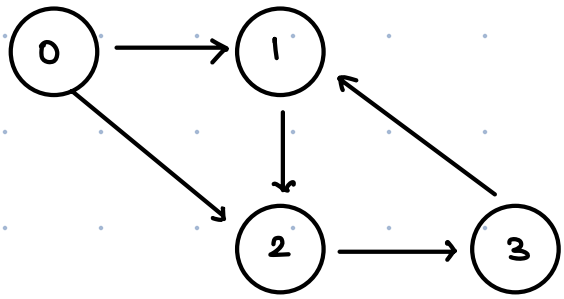
```
ArrayList<I>[] graph = new AL<>[n]        // n → no of nodes

for (i=0; i<n; i++){
    graph[i] = new ArrayList<>();
}

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

* Note ↓

In adjacency matrix, instead of 1, store the weight of edge.

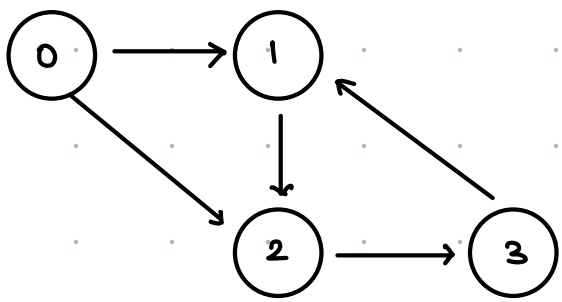


u	v	wt
0	1	10
0	2	20
1	2	30
2	3	40
3	1	50

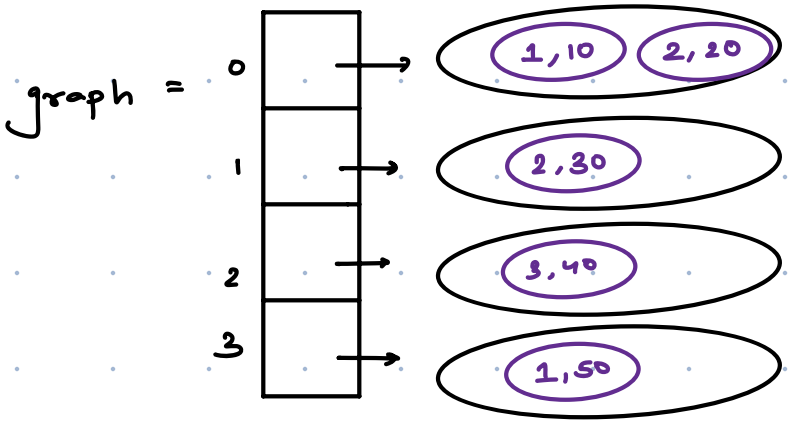
Adjacency matrix

	0	1	2	3
0	0	10	20	0
1	0	0	30	0
2	0	0	0	40
3	0	50	0	0

* Weighted graph adjacency list



u	v	wt
0	1	10
0	2	20
1	2	30
2	3	40
3	1	50




```
for (i=0; i< edges.length; i++){
    u = edges[i][0]
    v = edges[i][1]
    w = edges[i][2]
    Pair p = new pair(v,w);
    graph[u].add(p);
}
```

TC: O(E)

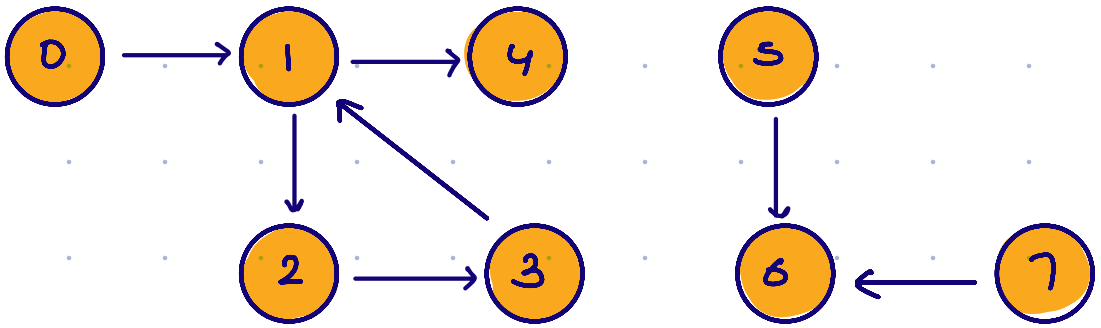
SC: O(V+E)

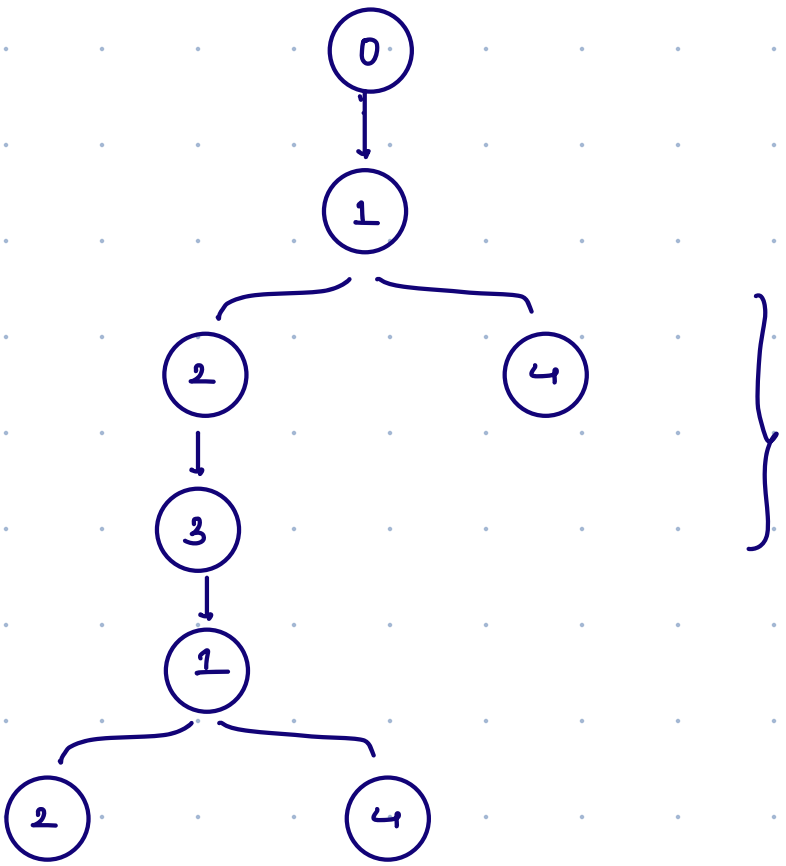
10:04 pm → 10:14 pm

* Traversal on Graph

- DFS — Depth first search
- BFS — Breadth first search

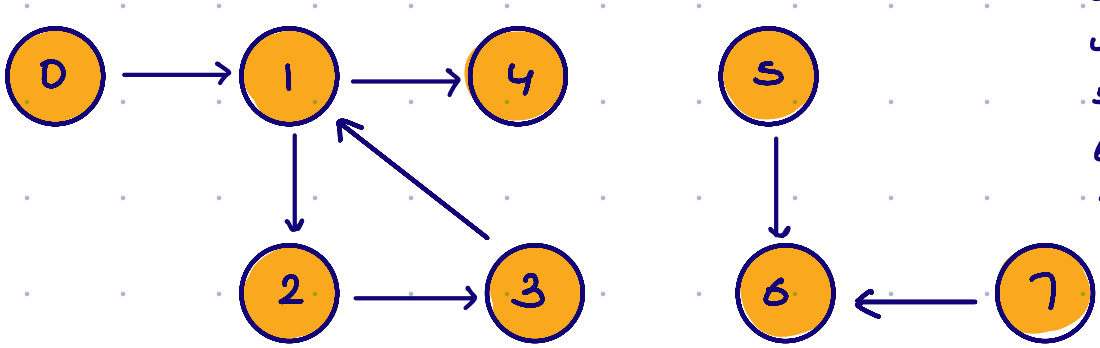
* Depth first search





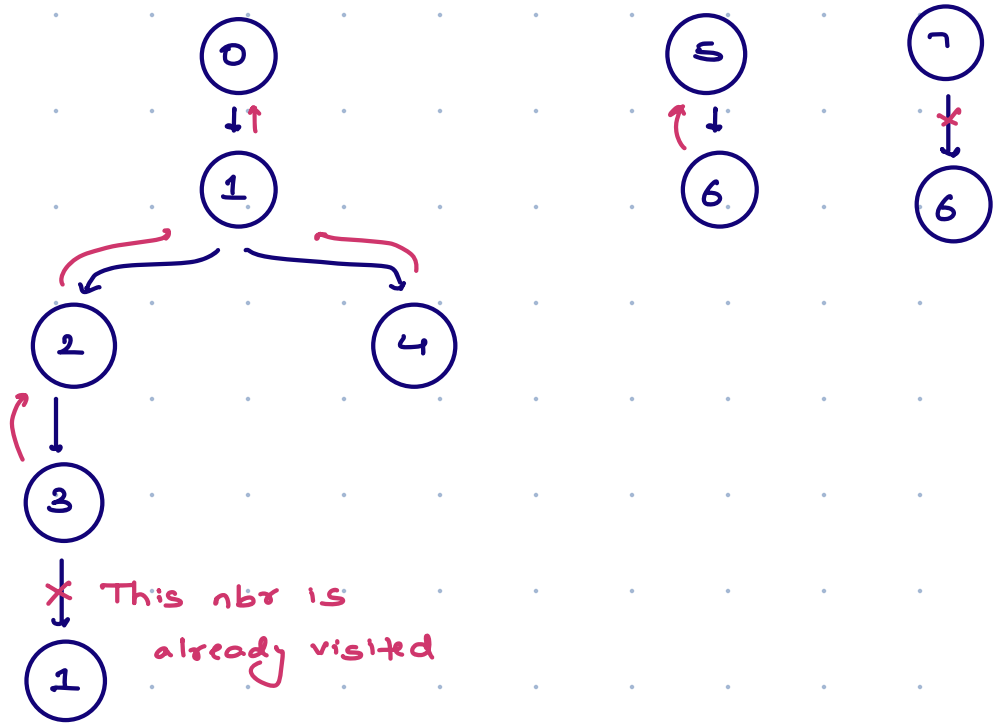
Infinite loop

0 → { 1 }
1 → { 2, 4 }
2 → { 3 }
3 → { 1 }
4 → { }
5 → { 6 }
6 → { }
7 → { 6 }



vis[ ] =

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



Process of DFS:

1. **Start at a Vertex:** Choose a starting vertex (Any).
2. **Visit and Mark:** Visit the starting vertex and mark it as visited.
3. **Explore Unvisited Neighbors:** From the current vertex, choose an unvisited adjacent vertex, visit, and mark it.
4. **Recursion:** Repeat step 3 recursively for each adjacent vertex.
5. **Backtrack:** If no unvisited adjacent vertices are found, backtrack to the previous vertex and repeat.
6. **Complete When All Visited:** The process ends when all vertices reachable from the starting vertex have been visited.

Code → graph is given

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

→ no. of vertices/nodes

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

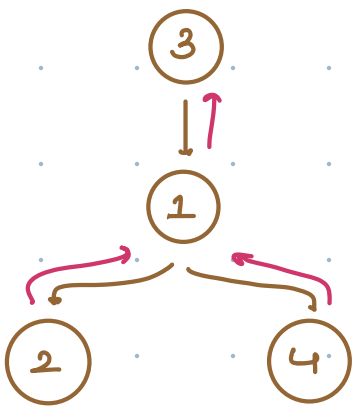
TC: O(V+E)

SC: O(V)

```
void dfs(graph, src, vis){
    vis[src] = true
    AL<I> al = graph[src]
    for(int nbr: al){
        if(vis[nbr] == false){
            dfs(graph, nbr, vis);
        }
    }
}
```

0 → {1}
1 → {2, 4}
2 → {3}
3 → {1}
4 → {}
5 → {6}
6 → {}
7 → {6}

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

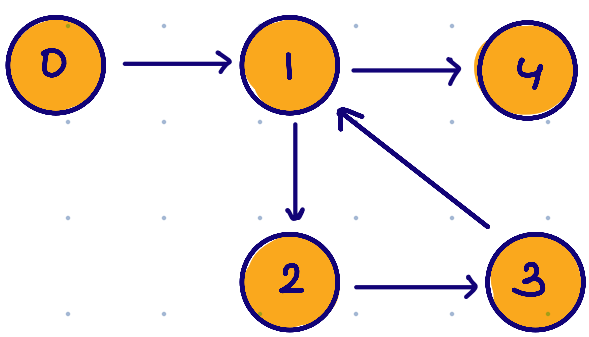


al = {2,4}

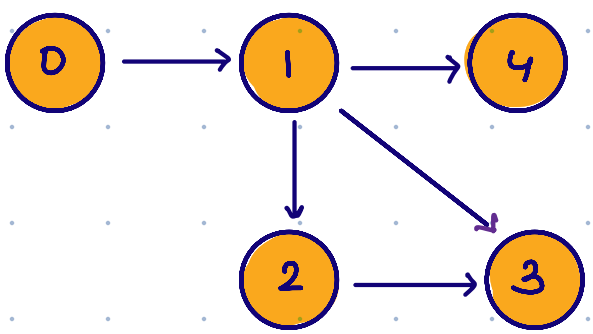
nbr = 2

al = {3}

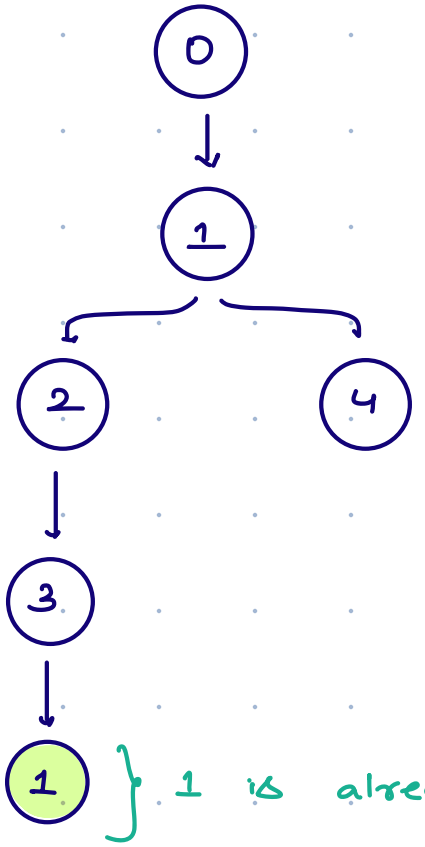
* Check if graph has a cycle



T	T	T	T	
0	1	2	3	4

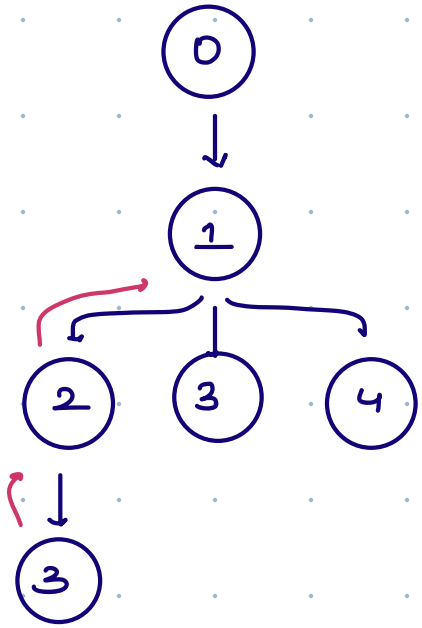


T	T	T	T	
0	1	2	3	4



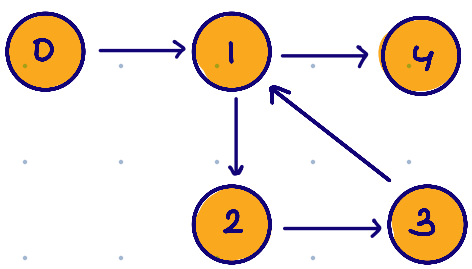
1 is already visited, so there is cycle

Ans = True



If vis node is encountered again → cyle ✗

If vis node is encountered again → cycle ✓ in same path

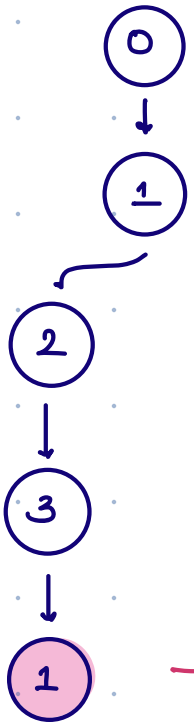


vis ()

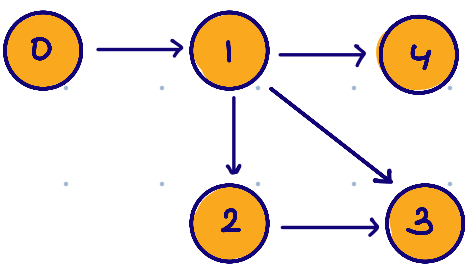
T	T	T	T	
0	1	2	3	4

path

T	T	T	T	
0	1	2	3	4



→ This node is falling in same path again, so its a cycle

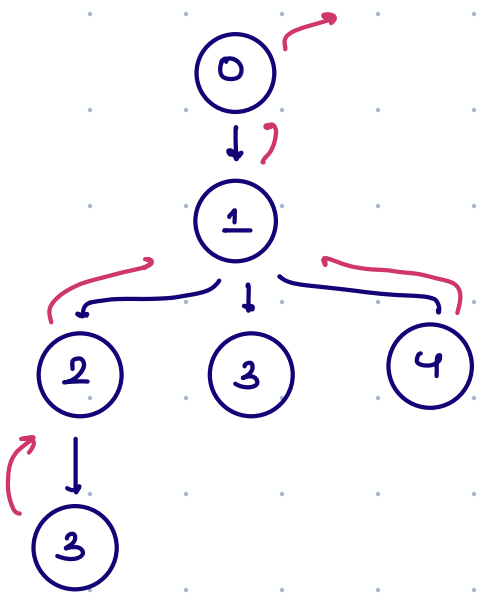


vis[]

T	T	T	T	T
0	1	2	3	4

path

F	F	F	F	F
0	1	2	3	4



```
boolean [] vis = new boolean [n];        // n → no. of vertices/nodes
boolean [] path = new boolean [n];
for (i=0; i<N; i++){
    if (vis[i] == false){
        if (dfs (graph, i, vis, path) == true) return true;
    }
}
return false;
```

```
boolean dfs (graph, src, vis){
    vis[src] = true
    path[src] = true;
    AL<I> al = graph[src]
    for (int nbr : al){
        if (path[nbr] == true) return true;
        if (vis[nbr] == false){
            if (dfs (graph, nbr, vis, path) == true) return true;
        }
    }
    path[src] = false;
    return false;
}
```

TC: O(V+E)

SC: O(V)