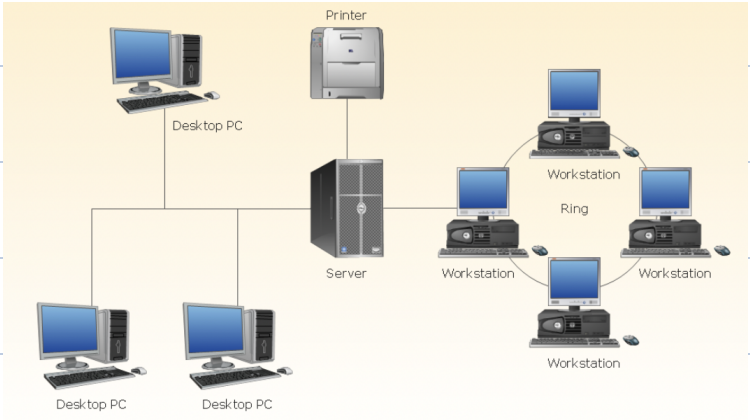
Current PSP 52.9 → 58

Nov23_PSP_3May
Gobika K
Vijay V A
kameswarreddy Yeddula
Piyush Kumar
Sawan
Sai Sharath
Manjunatha I
Harshil Dabhoya
Rajeev
Kevin Theodore E
Yash Malviya
kumkum

Nov23_PSP_3May
sudhakar venkatachalam
Suraj Devraye
Mohammad Mateen
MD JASHIMUDDIN
Vigneshwaran K
manikandan m
Prashant Kumar Soni
Pradeep Kumar Chandra
Shaurya Srivastava
Mayur Hadawale
SIJU SAMSON
Mohammed Arshad

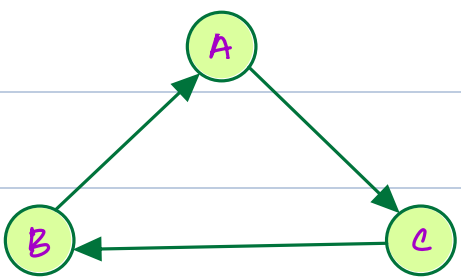
What is graphs ?

It is simply nothing but collection of nodes, connected to each other using edges

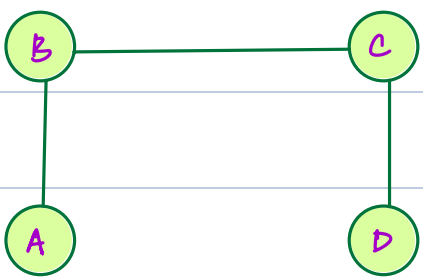


Classification of Graphs

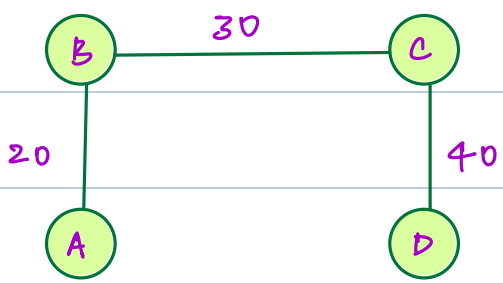
Directed Graphs



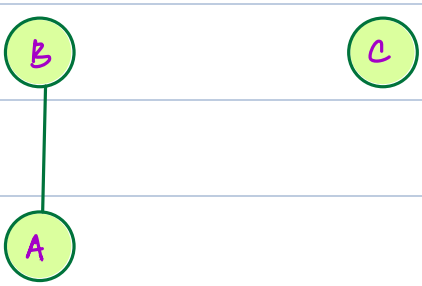
Undirected Graphs



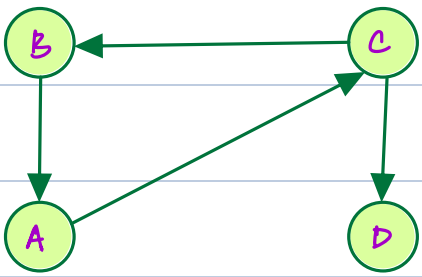
weighted Graphs



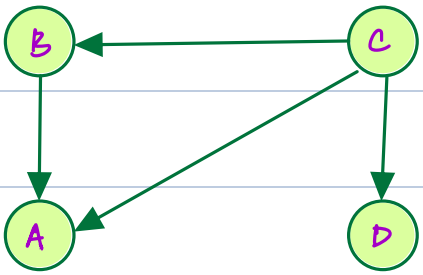
Disconnected Graph



cyclic Graph



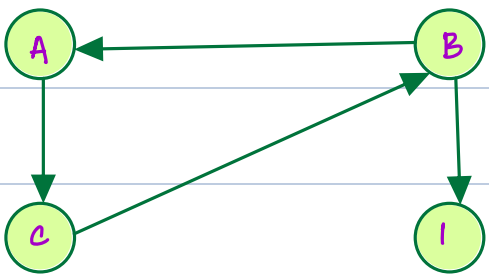
Acyclic

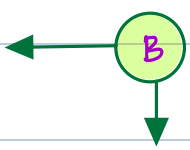


Indegree and Outdegree

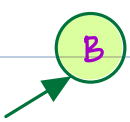
Indegree — No. of incoming edges

Outdegree — No. of outgoing edges





Outdegree = 2



Indegree = 1

Simple Graph

No self loops or multiple edges b/w same nodes

Not allowed

not allowed

Storing a Graph

N → nodes (5) M → edges (7)

start	end
1	4
2	5
3	2
4	3
2	4
3	5
1	2

edges

Adjacency Matrix

	1	2	3	4	5
1		1		1	
2	1		1	1	1
3		1		1	1
4	1	1	1		
5		1	1		

1 → edge

0 → no edge

```
edges [7] [2]

adj_m [n+1] [n+1]
```

no. of edges

```
for (i=0; i< M; i++) {
    start = edge [i] [0]
    end   = edge [i] [1]
    adj_m [start] [end] = 1;
    adj_m [end] [start] = 1;
}
```

s.c = O(n²)

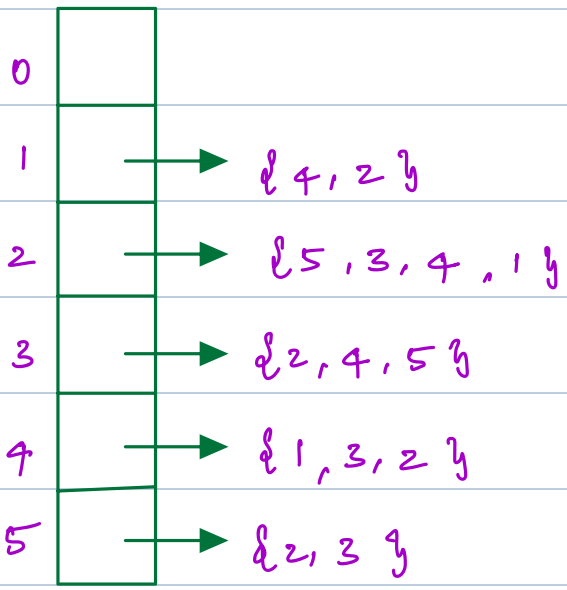
N → nodes M → edges

5 7

start	end
1	4
2	5
3	2
4	3
2	4
3	5
1	2

edges

Adjacency lists



pseudo code

```
AL = new ArrayList <> ();
for (i=0; i< N; i++)
    AL.add (new ArrayList <> ());
for (i=0; i<M; i++) {
    start = edge [i] [0];
    end = edge [i] [1];
    AL.get (start).add (end);
    AL.get (end).add (start);
}
```

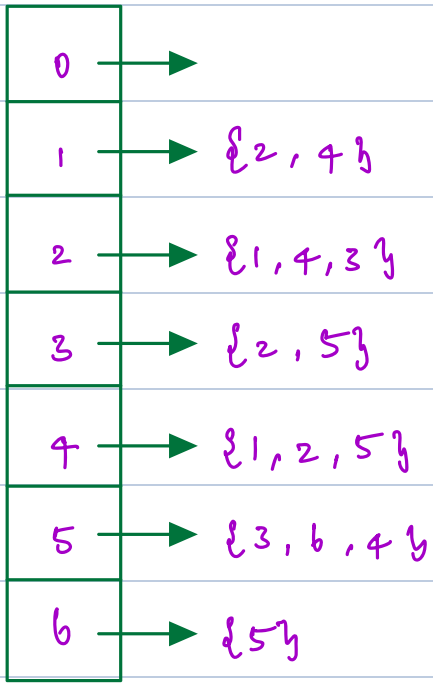
S.C = O(n+e)

Traversal

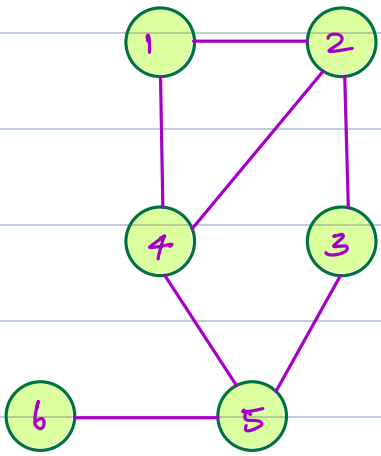
Depth First Traversal { inorder, preorder, postorder }

Node = 6 Edges = 7

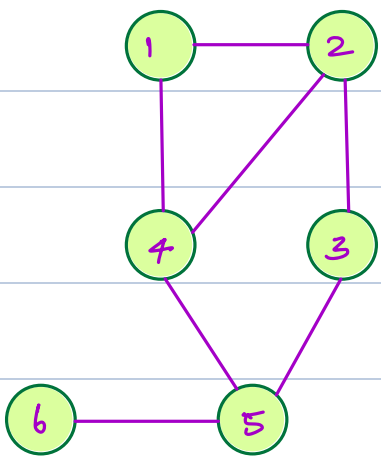
Start	End
1	2
1	4
2	4
2	3
3	5
5	6
4	5

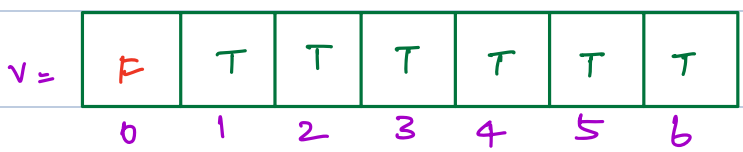


neighbours



Dry Run





Array, Hashmap, HS

4, 5, 6, 3, 2, 1

(1). Start DFS from any node

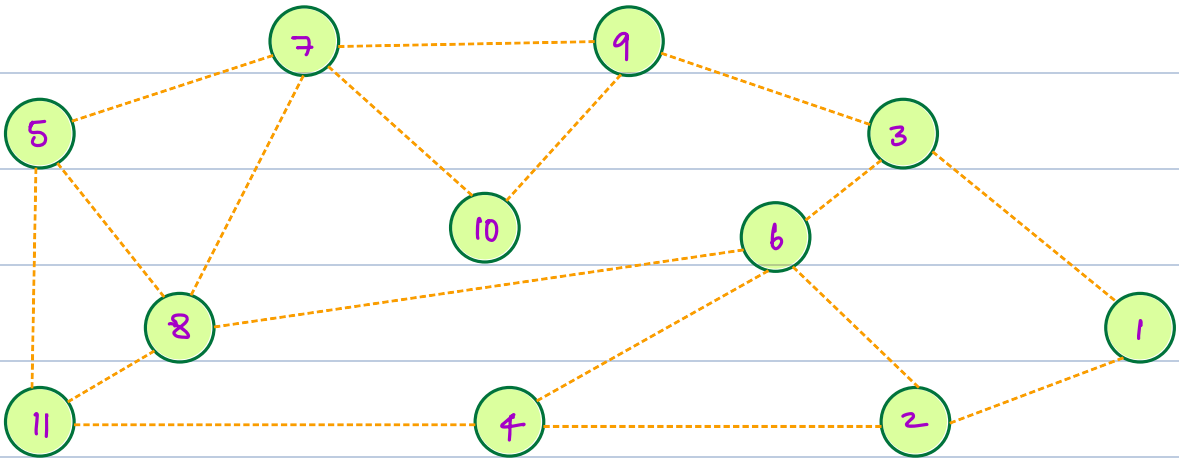
(2) Maintain visited array

pseudo code

```
// given N, M
// construct an adjacency list ⟶ graph
bool [] visited;
for (i = 0; i < n; i++)
    visited [i] = False;
for (j = 0; j < n; j++)
    if (! visited [j]) dfs (j)

void dfs (node) {
    // mark node as visited
    visited [node] = True
    // traverse all unvisited neighbours of node
    for (nei : graph.get (node)) {
        if (! visited [nei])
            dfs (nei);
    }
}
```

T.C : O(N+E)

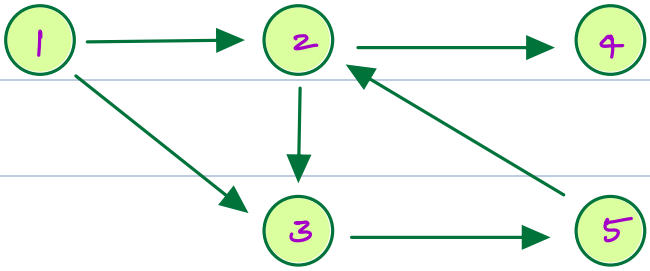


visited

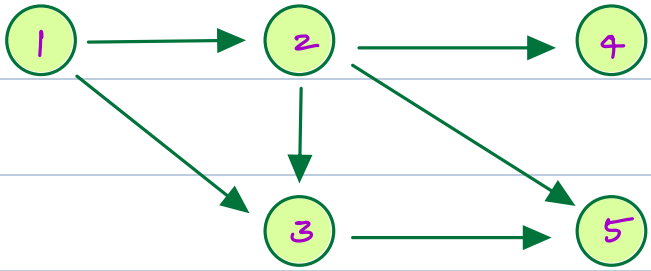
F	T	T	T	T	T	T	T	T	T	T	T
0	1	2	3	4	5	6	7	8	9	10	11

Output: 10 9 3 6 2 1 4 11 8 5 7

Detecting cycle in a directed graph



return True

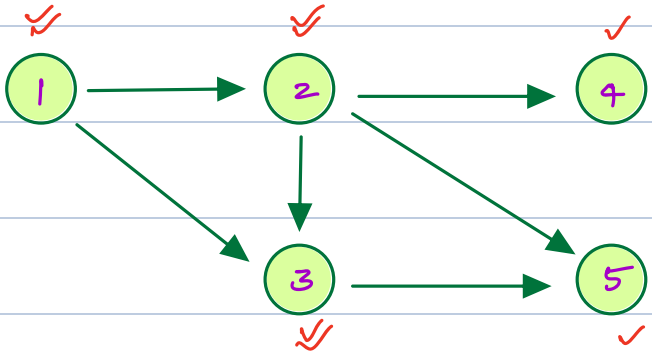


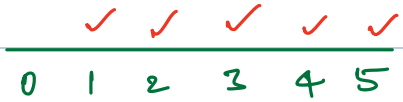
return False

Approach 1

If neighbour is a node that is already visited

→ cycle is present { Does not work }

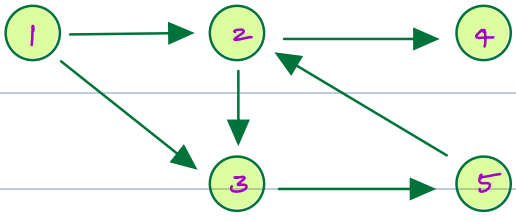




Approach 2

// Maintain current path

path = [] is cycle = False

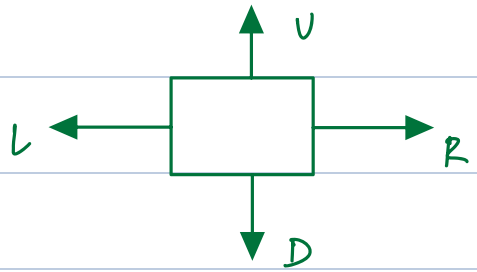



```
void dfs (node) {
    path.append(node);
    // mark node as visited
    visited[node] = True
    // traverse all unvisited neighbours of node
    for (nei : graph.get(node)) {
        If (path.contains(nei)) is cycle = True;
        If (!visited[nei])
            dfs(nei);
    }
    path.remove(node);
```

T.C = O(V+E)

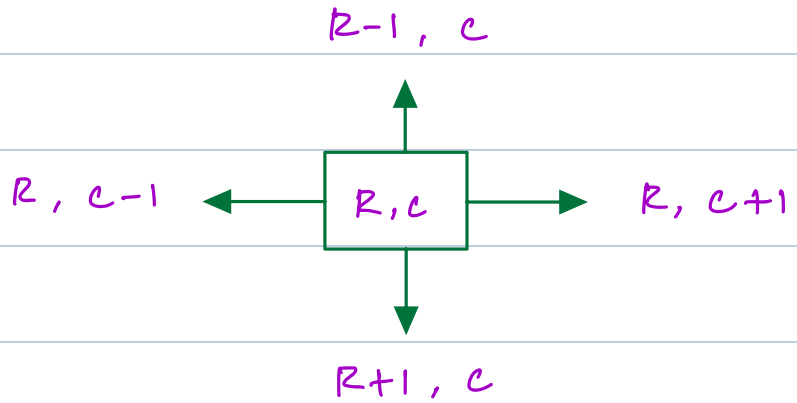
Q → Find the no. of Island in the grid

	0	1	2	3	4
0	1	1	0	0	1
1	0	1	0	1	0
2	1	0	0	1	1
3	1	1	0	0	0
4	1	0	1	1	1



ans = 5

neighbours



Observation 1

A[r][c] == 1 && visited[r][c] == 0

only in this criteria we go in depth of neighbours

Observation 2

nr, nc are index of neighbour not visited, land

#pseudo code

```
// R, C  no. of rows, no. of cols
visited[R][C] = False;
Island = 0;
for (i=0; i<R; i++) {
    for (j=0; j<C; j++) {
        if (A[i][j]==1 && visited[i][j]==False)
            dfs(i, j);
            Island += 1;
    }
}

dir_x = {-1, 0, 1, 0}      dir_y = {0, 1, 0, -1}

void dfs(r, c) {
    visited[r][c] = 1;
    for (i=0; i<4; i++) {
        nr = r + dir_x[i];   nc = c + dir_y[i];
        if (0<=nr<R && 0<=nc<C &&
            !visited[nr][nc] && A[nr][nc]==1) {
```

dfs (nr , nc) ;

	0	1	2	3	4
0	1	1	0	0	1
1	0	1	0	1	0
2	1	0	0	1	1
3	1	1	0	0	0
4	1	0	1	1	1

T.C = (R*C)

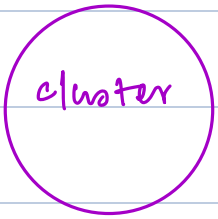
S.C = (R*C)

LinkedIn maximizing the Reach

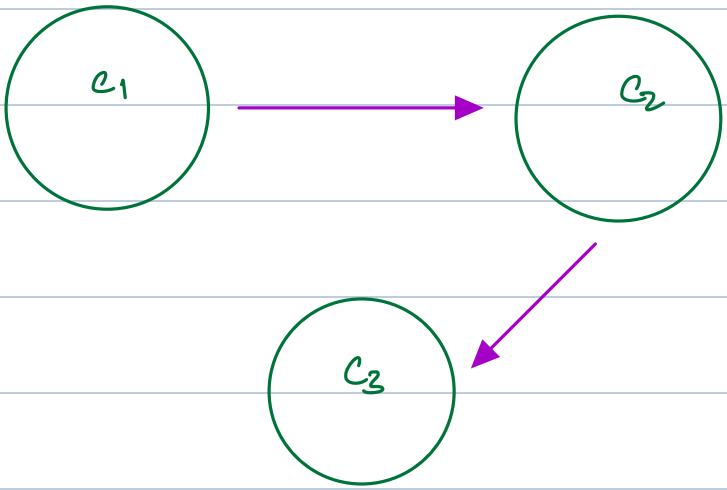
In the LinkedIn ecosystem, understanding the structure of professional networks is crucial for both users and the platform itself.

A new feature is being developed to visualize the network of a user in terms of **"clusters"**. These clusters represent groups of LinkedIn users who are all connected to each other, either directly or through mutual connections, but do not have connections to users outside their cluster. This visualization aims to help users to increase their **Reach**.

Given **A** denoting the total number of people and matrix B of size Mx2, denoting the bidirectional connections between users, and an Integer C denoting the user ID of each connection joins two users by their user IDs, the target person, find out the number of connections that this person should make in order to connect with all the networks.



→ connected directly or through mutual connection



ans = { no. of cluster − 1 }
