Agenda:

1. Intro to Graphs
2. Types of Graphs
3. Traversal: DFS
4. Detect cycle in directed graph

Additional Mocks ?

1. Mandatory Skill Evaluation DSA 2.5 hrs

2. Mock Interview (1 hr)

   Intro (5 min)

   Qus and Ans (25 min)

   Feedback (5 min)

vertices

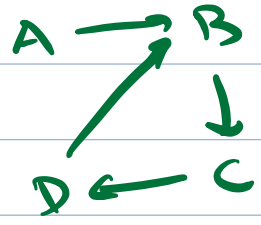
Graphs : Collection of nodes and edges

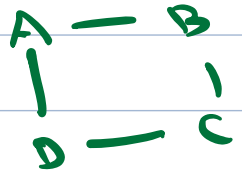
Real life examples of Graph :

1. Computer Networks
2. Google Maps
3. Social Media

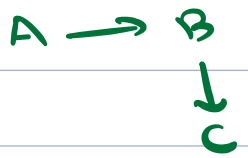
Types of Graph

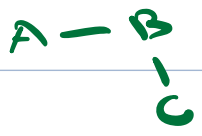
1. Cyclic Graph - Graph which contains atleast one cycle. If we start from a node and return to same node without repeating an edge → cycle
↓
closed path



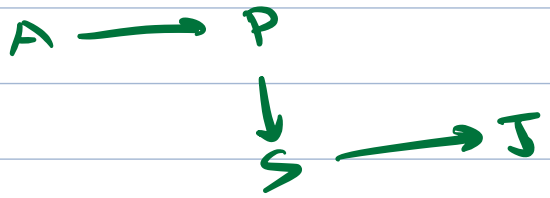


2. Acyclic Graph : Graph with no cycles
↓
No closed paths

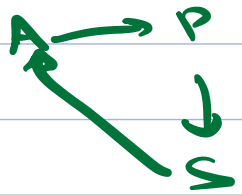




3. **Directed Graph**: Edges have a direction, indicating one way relationship b/w nodes

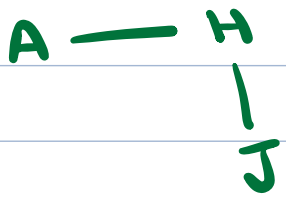


Directed Acyclic

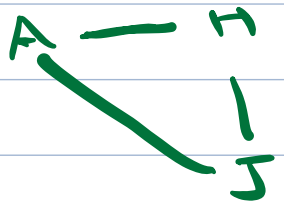


Directed Cyclic

4. **Undirected Graph**: Edges have no direction, representing symmetric relationship b/w nodes

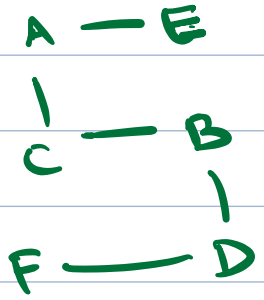


Undirected Acyclic



Undirected cyclic

5. **Connected Graph**: A connected graph has a path b/w every pair of nodes. There are no isolated nodes.

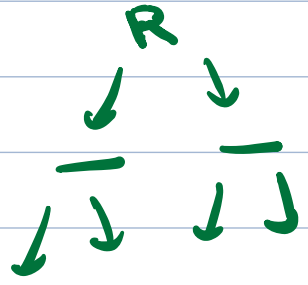


6. Disconnected Graph : Graph with atleast 2 disconnected components (meaning there is no path b/w them)

Q. Is tree a graph ? Yes

① Directed Acyclic Graph

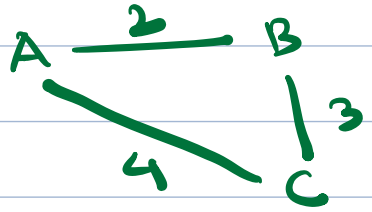
Tree → DAG



② Connected

③ N nodes, N-1 edges

7. Weighted Graph : Edges have associated weights → used to represent distances/ time / costs or other metrics.



8. Unweighted Graph : No associated weights on edges

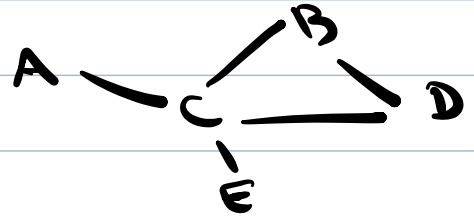
① Directed undirected

② Cyclic Acyclic

③ Weighted unweighted

Degree of a vertex : No. of edges connected to vertex

Degree → Undirected graph



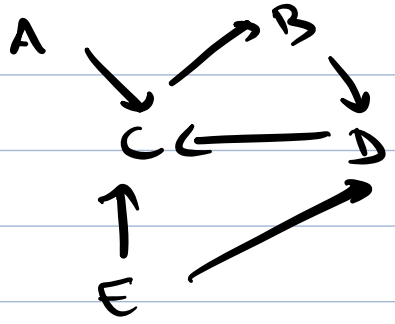
A 1
B 2
C 4
D 2
E 1

Indegree and outdegree of a vertex

Directed Graph

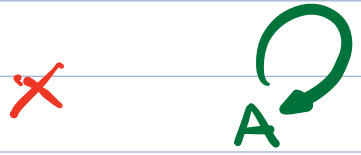
No. of incoming edges for a vertex

No. of outgoing edges for a vertex

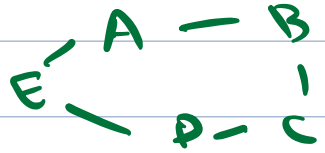


	Indegree	Outdegree
A	0	1
B	1	1
C	3	1
D	2	1
E	0	2

Simple Graph: No self loops or multiple edges b/w same pair of nodes

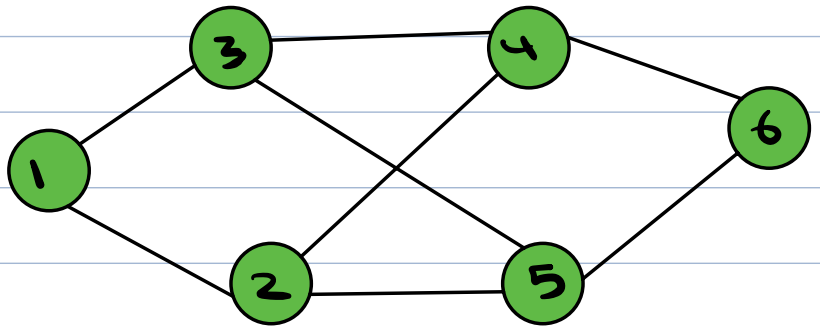


→ No self loop



How to store graph? — Adjacency Matrix / Adjacency List

① Adjacency Matrix



N E

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

Edges

① N → 1 to 6

② adj[1][1] = <0>

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
2	0	1	0	0	1	1	0
3	0	1	0	0	1	1	0
4	0	0	1	1	0	0	1
5	0	0	1	1	0	0	1
6	0	0	0	0	1	1	0

① N nodes, 1 to N → adj[N+1][N+1]

② N nodes, 0 to N-1 → adj[N][N]

Q. Given array of edges, construct adjacency matrix

① // I/P

```
input << N << edges
int adj[N+1][N+1] = {0}
for (i=0 ; i < edges ; i++) {
    input (u and v)
    adj[u][v] = 1
    adj[v][u] = 1
}
```

② // N, edges[][]

```
[[1,3]
 [2,5]
 [4,3]]
```

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

Directed (u → v)	Undirected (u – v)
u v adj[u][v] = 1	u v adj[u][v] = 1 adj[v][u] = 1
Directed weighted	**Undirected weighted**
u v w adj[u][v] = w	u v w adj[u][v] = w adj[v][u] = w

Advantage : Easy to add | delete | update edges.

Disadvantage : Space wastage bcoz of non existent edges

$N = 10^5$ adj → $10^5 \times 10^5 = 10^{10}$

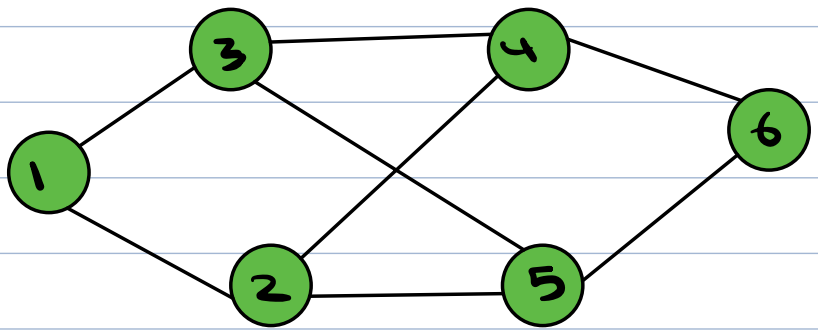
Only $N = 10^3$, we can use adj matrix → $10^3 \times 10^6 = 10^6$

SC : $O(N^2) / O(V^2)$

---

2. Adjacency List

Store neighbours of every node



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

List<int> adj [7]

0 →

1 → 3 2

2 → 5 4 1

3 → 1 4 5
4 → 3 2 6
5 → 2 3 6
6 → 4 5

N/V = 6 (1 to 6)
E = 8

SC: O(V + 2E)
= O(V + E)

① N nodes (1 to N)
list<int> adj[N+1]

② N nodes (0 to N-1)
list<int> adj[N]

10:30

Q. Given array of edges, construct adjacency list

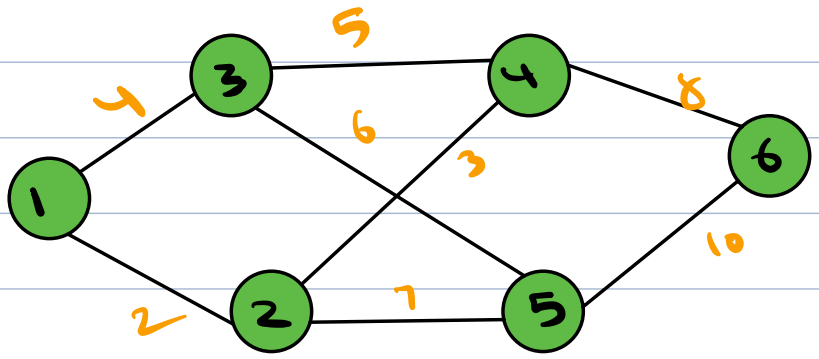
① // I/P

```
input << N << edges
list<int> adj[N+1]
for (i=0 ; i< edges ; i++){
    input (u and v)
    adj[u].add(v)
    adj[v].add(u)
}
```

u - v

adj[u].add(v) → u → v

Weighted graph



u → (v, w)

0 →

1 → (3,4) (2,2)

2 → (5,7) (4,3) (1,2)

3 → 1 4 5

4 → 3 2 6

5 → 2 3 6

6 → 4 5

list <pair <int, int>> adj [N+1]

for (i = 0 ; i < edges ; i++) {

input (u, v and w)

adj[u].add(<v,w>)

adj[v].add(<u,w>)

} u → v

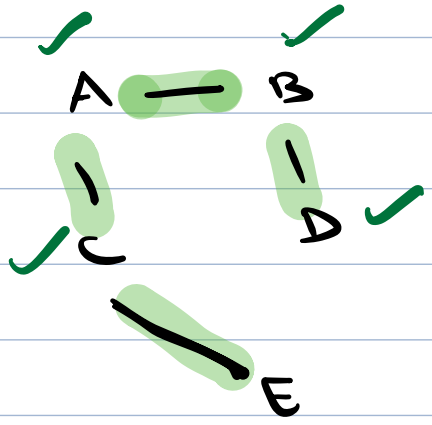
adj[u].add(<v,w>) ? u → v

}

Graph Traversals

1. DFS - Depth First Search
2. BFS - Breadth First Search

DFS — Preorder, Postorder, Inorder } Recursion

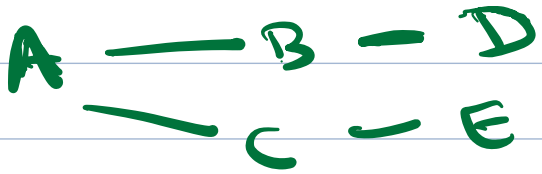


A → B, C
B → A, D
C → A, E
D → B
E → C

Visited

A	B	C	D	E
F T	F T	F T	F T	F T

O/P: A B D C E



```
// N nodes (1 to N)
bool visited[N+1] = {F}
list<int> adj[N+1]

void dfs(int curNode){
    print(curNode)
    visited[curNode] = true

    for(i=0; i<adj[curNode].size(); i++){
        int nbr = adj[curNode][i]
        if(visited[nbr] == false)
            dfs(nbr)
    }
}

void main(){
    // 1 to N
    for(i=1; i<=N; i++){
        if(visited[i] == false)
            dfs(i)
    }
}
```

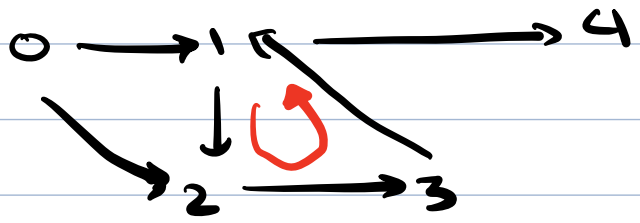
TC: $O(V+2E)$

$\simeq O(V+E)$

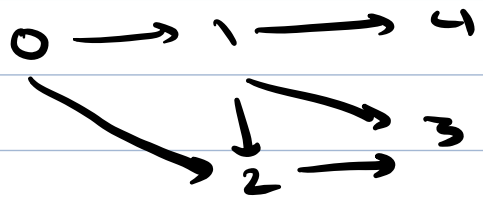
SC: $O(V)$

vis[] and stack

1. Given a directed graph, check if it contains a cycle or not?

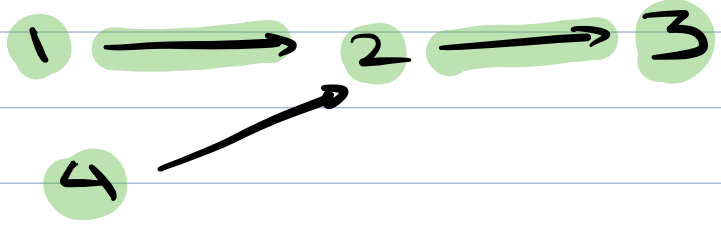


True



False

Approach : Apply DFS and if we come across a visited node, there's a cycle.

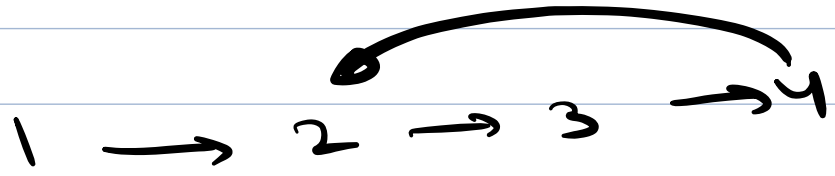


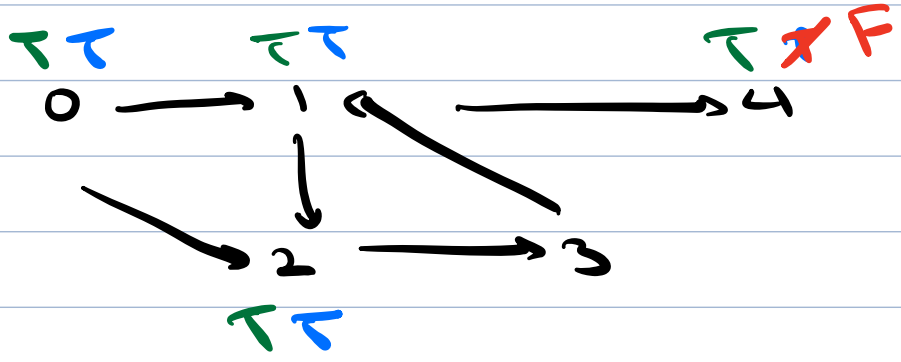
1 → ~~2~~
2 → ~~3~~
3 →
4 → 2

vis 1 2 3 4

~~F~~ T ~~F~~ F ~~F~~ T ~~F~~ T

1 → 2 → 3
4 → 2



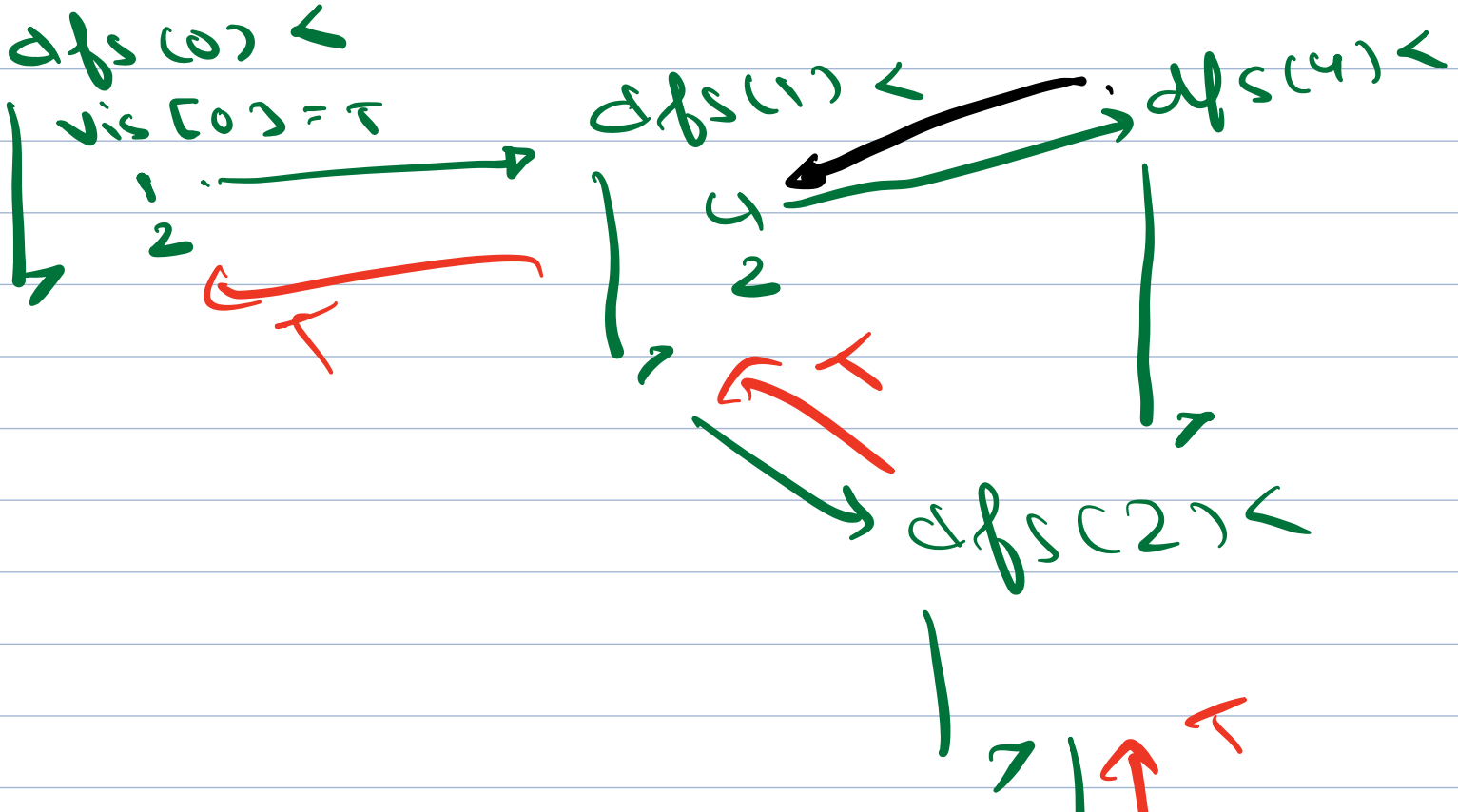


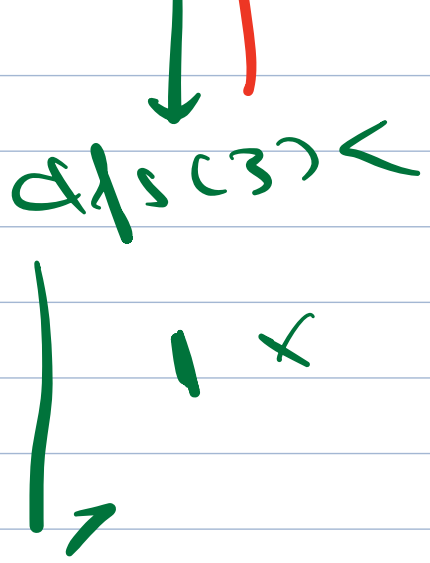
vis 0 1 2 3 4

Green → vis

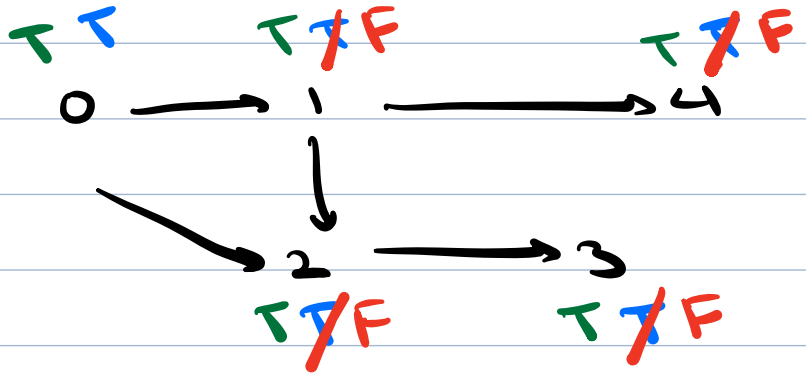
Blue → Path

path 0 1 2 3 4

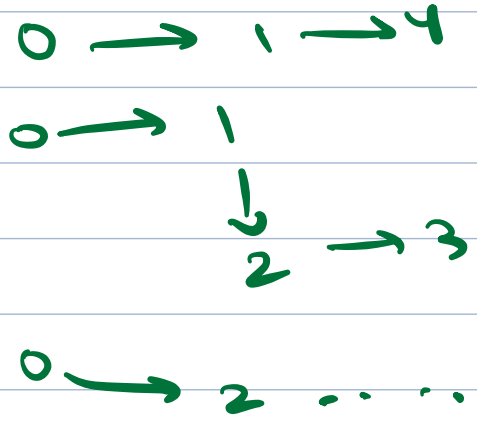




Correct Approach: Look for an already visited node in same path → cycle



False



```
// return true if curNode has a cycle
   or any of the future neighbours

bool dfs(int curNode) {
    vis[curNode] = true
    path[curNode] = true

    for(i=0; i<adj[curNode].size(); i++){
        int nbr = adj[curNode][i]
        if(visited[nbr] == false){
            if(dfs(nbr) == true)
                return true
        }
        else {
            if(path[nbr] == true)
                return true
        }

        path[curNode] = false
        return false
    }
```

TC: O(V+E)

SC: O(V)

3V

↓

vis[], path[], stack