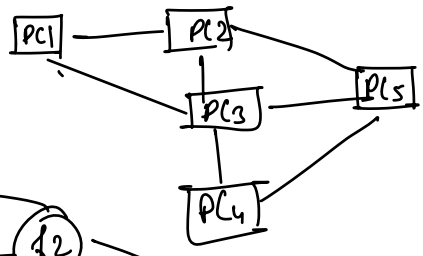
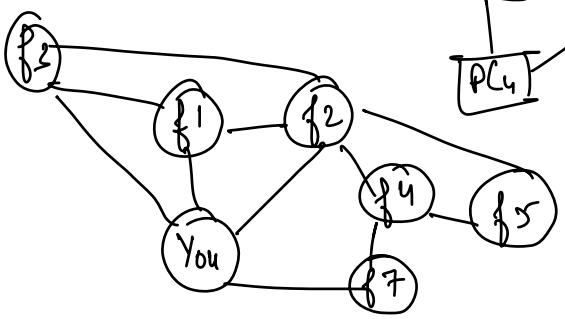
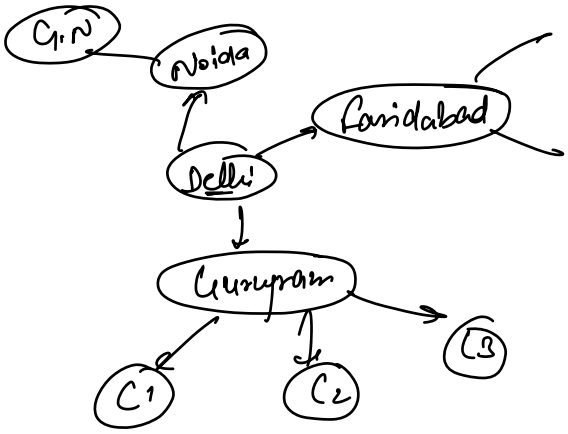
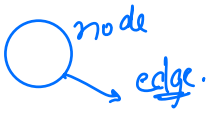
Graph → network.

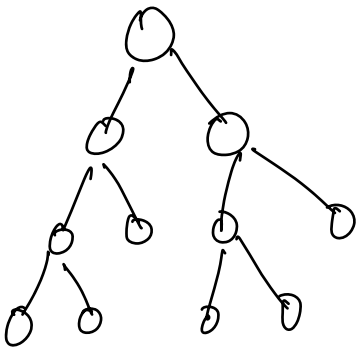


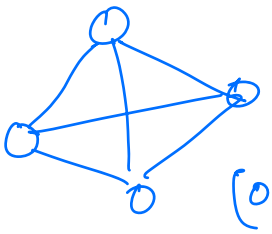






Graph → collection of nodes and edges.





[0 to $^NC_2$]

$$^NC_r = \frac{N!}{(N-r)!\,r!}$$

$$= \frac{4!}{2!\,2!}$$

$$= \frac{\cancel{4}^{2} \times 3 \times \cancel{2!}}{\cancel{2!} \times \cancel{2!}}$$

$$= 6.$$

① N nodes → N-1 edges.

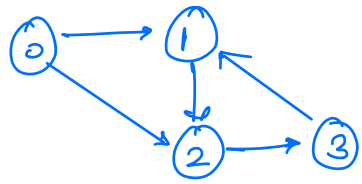
② Tree always has root node

③ Cycle will not be present in tree.

∴ Tree is a subset of graph.

How to store graph in code?

[5 edges
4 nodes]



① 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

$m[i][j] \rightarrow = 1$ edge from i to j

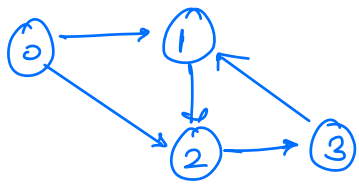
$\searrow = 0$ no edge from i to j

N nodes → S.C ⇒ $O(N^2)$

② Adjacency List

[Array of lists]

[5 edges
4 nodes]



0 → {1, 2}

1 → {2}

2 → {3}

3 → {1}

[A[i] → list of nodes i is pointing to.]

S.C → $O(N+E)$

$\left[\begin{array}{l} 1 \le N \le 10^5 \\ 1 \le E \le 10^5 \end{array}\right]$

⇓

{Σ length of all lists = edges.}

Properties / Types of Graph

① **Directed**

$i \longrightarrow j$

i to j ✓
j to i ✗

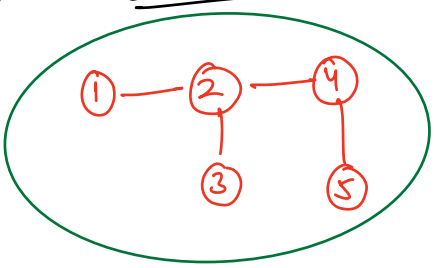
Undirected

$i \text{ ——— } j$

$i \rightleftarrows j$

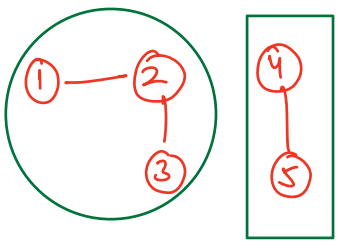
i to j ✓
j to i ✓

② **Connected**



Starting from any node, if you can reach any other node.

Disconnected



③ **Weighted**

$1 \xrightarrow{5} 2$

$m[i][j] \rightarrow w_{ij}$ (weight)

$\searrow$ 0 (no edge b/w i & j)

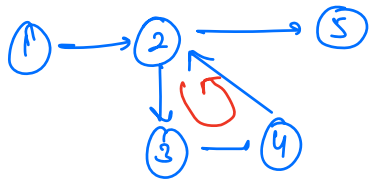
$A[i] \rightarrow \{\{\underset{\text{nbr}}{1}, \underset{\text{edge wt.}}{5}\}, \{2,3\}, --\{\ \}\}$

list of pair.

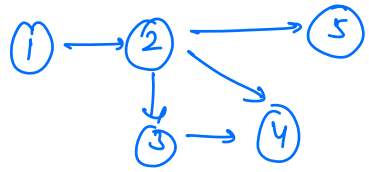
Unweighted

$1 \longrightarrow 2$

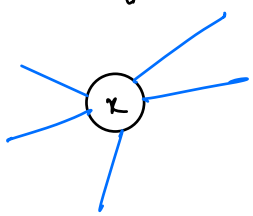
(4) **Cyclic**



Acyclic



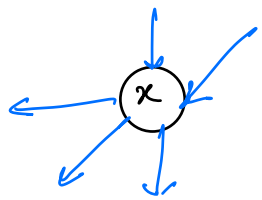
(5) **Degree**



no. of edges connected to a node.

degree (x) = 5

Indegree / Outdegree



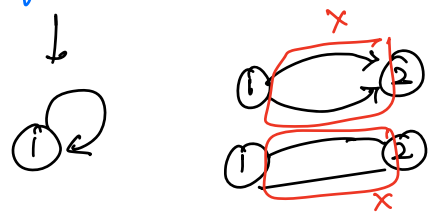
in-degree → incoming edges

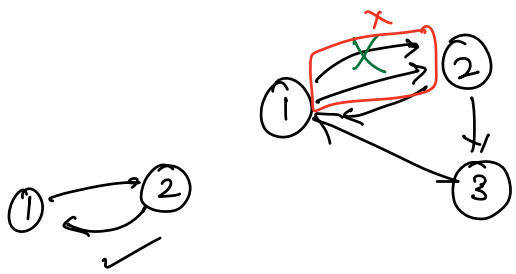
out-degree → outgoing edges.

in(x) = 2 , out(x) = 3

(6) **Simple Graph**

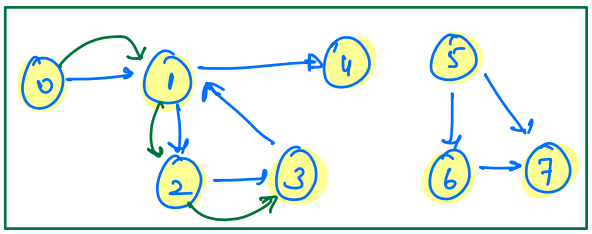
connected graph without self-loops & multi-edges





Traversals

① **Depth First Search** ⇒ Go deep till possible, once a path is complete backtrack to alternate path

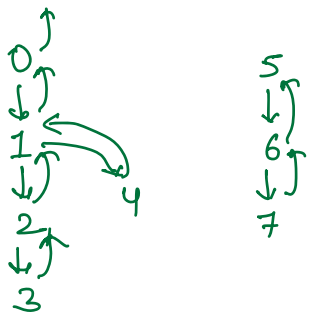


Keep track of visited nodes.

visited →

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

idea → for every unvisited node, you need to call dfs();



```
∀i, visited[i] = false.
for( i=0; i < N; i++){
        if (visited[i] == false){
                dfs(i)
        }
}
```

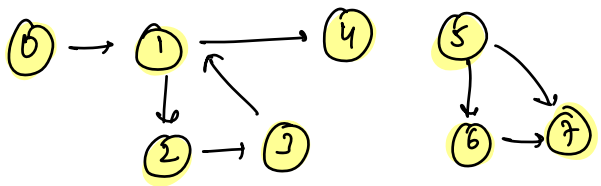
```
void dfs( int src){
        print(src)
        visited[src] = true;
        for( int nbr : A[src]){
                if( visited[nbr] == false) { dfs(nbr) }
        }
}
```

T.C → O(N+E)
S.C → O(N)
⇓
visited[] → ∞
recursive stack → N

||

```
for( i=0; i < A[src].size(); i++){
    nbr = A[src].get(i)
    if(visited[nbr] == false) { dfs(nbr) }
}
```

Breadth first Search → level order traversal



→ Keep track of visited nodes.

visited →

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

$\forall i$, visited[i] = false.

```
for( i=0; i < N; i++){
    if (visited[i] == false){
        bfs(i)
    }
}
```

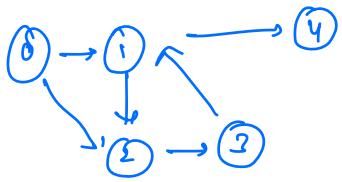
0	1	2	4	3	5	6	7

op. – 0, 1, 2, 4, 3, 5, 6, 7

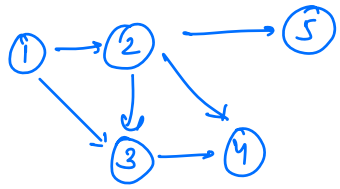
T.C → O(N+E)
S.C → O(N)

```
void bfs( int src) {
    q.enqueue(src), print(src), visited[src] = true;
    while ( q.isEmpty() == false ){
        x = q.dequeue();
        for( int nbr : A[x]){
            if (visited[nbr] == false){
                q.enqueue(nbr), print(nbr), visited[nbr] = true;
            }
        }
    }
}
```

Q: Check if the simple directed graph has a cycle?



ans → true



ans → false

→ if a visited node is encountered again → cycle ✗

→ if a visited node in current path is encountered again → cycle ✓

```
∀i, visited[i] = false.
for( i=0; i < N ; i++){
        if (visited[i] == false && dfs(i))
            return true
}
return false;
```

```
boolean dfs ( int src){
        visited[src] = true
        path[src] = true
for( int nbr : A[src]){
        if( path[nbr] == true){
            return true
        }
        if( visited[nbr]== false &&
            dfs(nbr) == true){
            return true
        }
}
path[src] = false;
return false;
}
```

$\boxed{\begin{array}{l} T.C \to O(N+E) \\ S.C \to O(N) \end{array}}$

✓No. of islands.