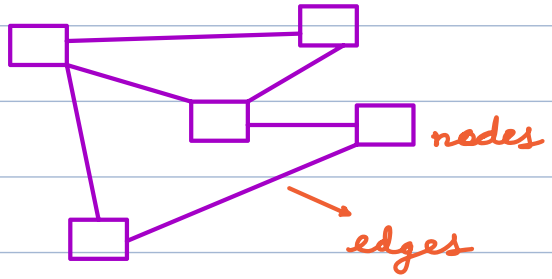
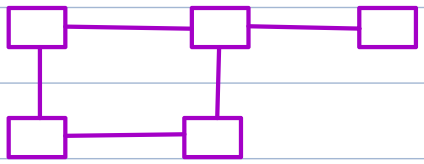
Graphs → collection of nodes & edges.

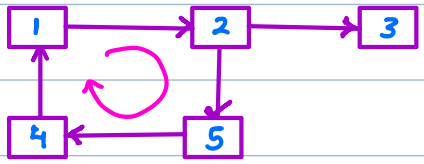


Types of Graphs

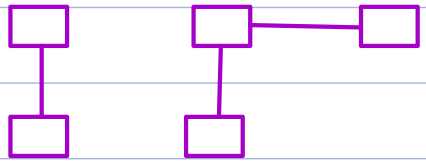
1) **Cyclic**

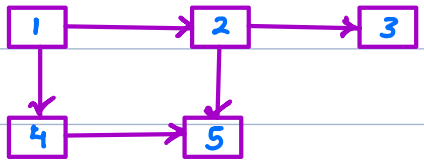


There is a path from one node to itself.

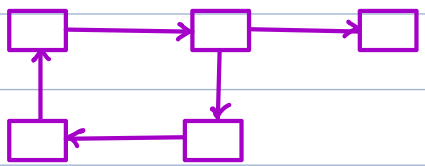


Acyclic



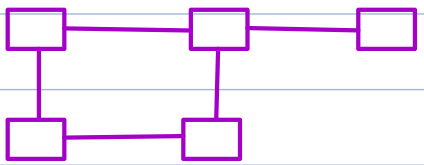


2) **Directed**

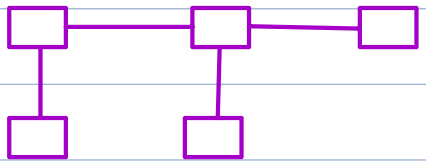


Single direction when travelling from one node to another.

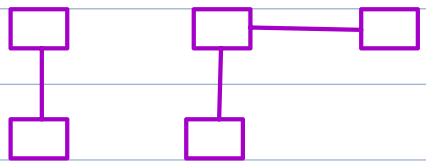
Undirected



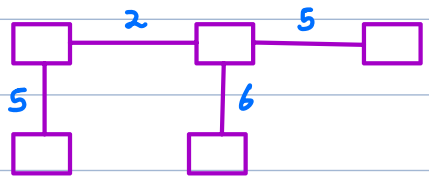
3) **Connected**



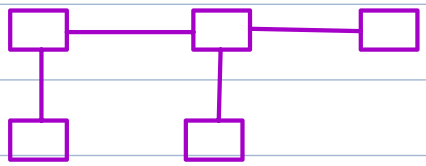
Disconnected



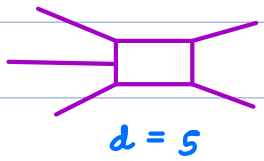
4) **Weighted**



Unweighted

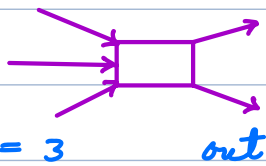


5) **Degree**



degree(node) = # edges connected to that node.

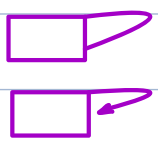
In/Out Degree



indegree → # incoming edges

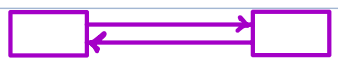
outdegree → # outgoing edges

6) Simple Graph → A connected graph without self loops & multi edges.



Both ends of edge connecting same node.

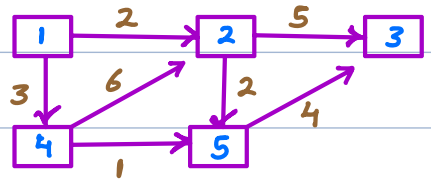
Multiple edges having same usecase.



(Not a multiedge)

How to store graphs ?

1) Adjacency Matrix



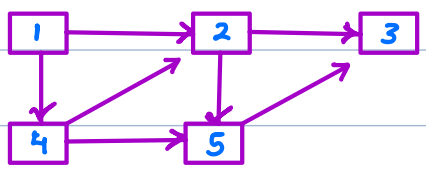
A[i][j] = 1 , $i \rightarrow j$
0 , no edge

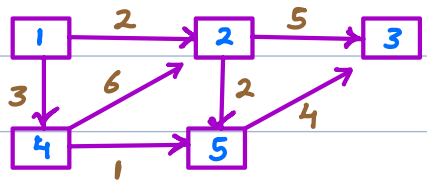
A[i][j] = wt , $i \xrightarrow{wt} j$
0 , no edge

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

SC = $O(N^2)$

2) Adjacency List





∀i, A[i] = list of nodes 'i' is pointing to.

A[1] → { 2, 4 }
A[2] → { 3, 5 }
A[3] → { }
A[4] → { 2, 5 }
A[5] → { 3 }

∀i, A[i] = list of (node, wt) 'i' is pointing to.

A[1] → { (2, 2), (4, 3) }
A[2] → { (3, 5) , (5, 2) }
A[3] → { }
A[4] → { (2, 6) , (5, 1) }
A[5] → { (3, 4) }

SC = $O(N + E)$

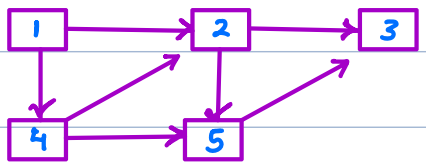
$N, E <= 10^5$ ⇒ use Adjacency List

Traversals → DFS ✓ (Preorder / Inorder / Postorder)

BFS (Level Order Traversal)

Next lecture

Depth First Search → Start from a node & travel down the path till it is possible, if blocked backtrack one step.



start 5, 1

o/p → 5 3 1 2 4

A[1] → { 2, 4 }

A[2] → { 3, 5 }

A[3] → { }

A[4] → { 2, 5 }

A[5] → { 3 }

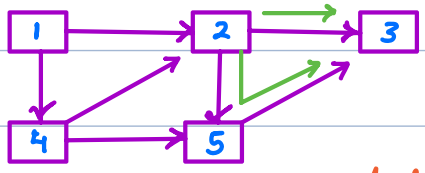
```
∀i, vst[i] = false
for i → 1 to N {
    if (!vst[i]) dfs(i)
}
```

```
void dfs (u) {
    vst[u] = true
    print(u)
    for ( v : Adj[u] ) {   // For-each loop
        if (!vst[v]) dfs(v)
    }
}
```

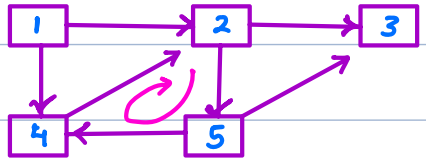
}

TC = O (N+E) SC = O(N) (recursion + |vst|)

Q → check if the given graph is a cyclic graph.

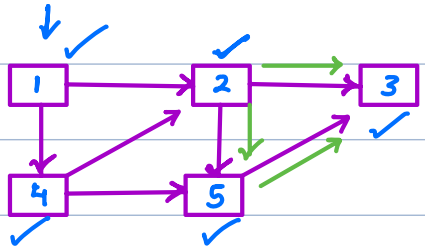


Ans = false



Ans = true

If we reach a visited node again ⇒ cyclic graph. ✗



	1	2	3	4	5
vst = [	t	t	t	t	t]
path = [	f	f	f	f	f]

```
∀i, vst[i] = false
∀i, path[i] = false
for i → 1 to N {
    if (!vst[i] && dfs(i))
        return true
}
return false
```

TC = O (N+E) SC = O(N)

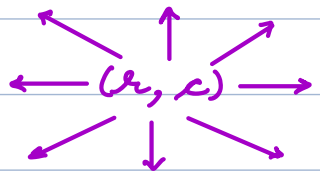
```
boolean dfs (u) {
    vst[u] = true
    path[u] = true
    for (v : Adj[u]) {   // For-
        if (path[v]) return true
        if (!vst[v] && dfs(v))
            return true
    }
    path[u] = false
    return false
}
```

Q → Given a map in the form of matrix where

A[i][j] → 1 ⇒ land
→ 0 ⇒ water

A group of connected lands (1 or more) is called island.

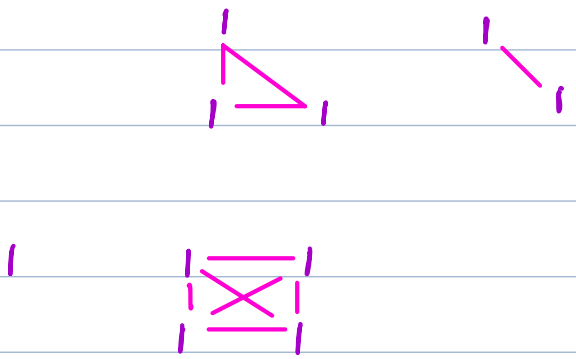
Find the count of island in the given matrix.



outdegree = 8
indegree

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

Ans = 4



Ans = # connected components

vst[i][j] → (A[i][j] = 2)
update input

```
ans = 0
for i → 0 to (N-1) {
    for j → 0 to (M-1) {
        if (A[i][j] == 1) {   // unvisited land
            ans++
            dfs(i,j)
        }
    }
}
return ans
```

TC = O(N * M) SC = O(N * M)

```
void dfs(r, c) {
    dr = [-1  -1  -1  0  0  1  1  1]
    dc = [-1  0  1  -1  1  -1  0  1]
    A[r][c] = 2  // visited land
    for k → 0 to 7 {
        u = r + dr[k]
        v = c + dc[k]
        if (0 <= u && u <= N-1 &&
            0 <= v && v <= M-1 && A[u][v] == 1)
                dfs(u, v)
    }
}
```