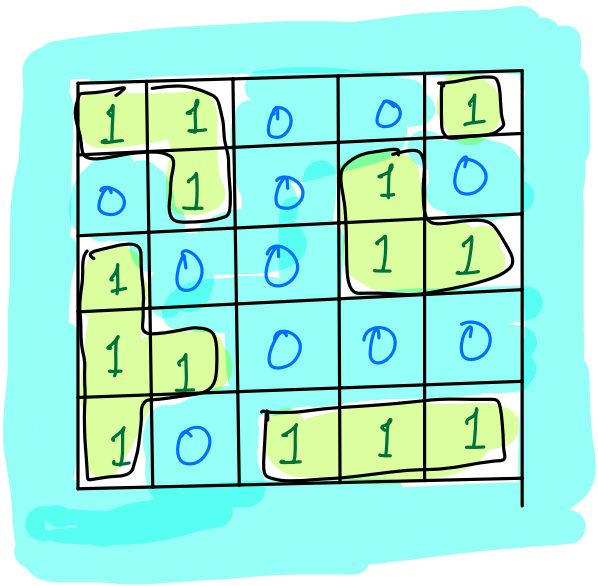
Today's Agenda

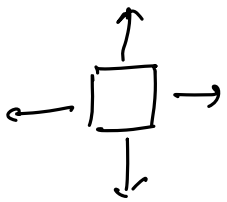
- No. of Islands
- Topological Order → 2 methods
- D.S.U. [Disjoint set Union]
- Application for D.S.U.

Number Of Islands



1 → land

0 → water.



ans = 5.

Every cell with '1' as node of a graph.

∴ No. of islands = no. of components in the graph.

Idea → From every unvisited '1' call dfs / bfs.

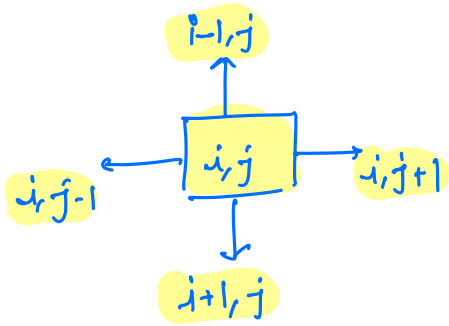
code →

```
visited[N][M];  // visited[i][j] = false

count = 0

for(i=0; i<N; i++){
    for(j=0; j<M; j++){
        if(arr[i][j] == 1 && visited[i][j] == false){
            dfs(arr[][], visited[][], i, j);
            count++;
        }
    }
}

return count;
```



dx[7] → {-1, 0, +1, 0}
dy[7] → {0, -1, 0, +1}

```
void dfs(arr[N][m], visited[N][m], i, j){
    visited[i][j] = true;
    dx[7] → {-1, 0, +1, 0}
    dy[7] → {0, -1, 0, +1}                    i, j
    for( k=0; k<4; k++){
        ni = i + dx[k];
        nj = j + dy[k]
        if(ni ≥0 && ni < N && nj ≥0 && nj < m &&
           arr[ni][nj] == 1 && visited[ni][nj] == false){
            dfs( arr[][], visited[][], ni, nj);
        }
    }
}
```

T.C → O(nodes + edges)
↓ ↓
N*m + 4*N*m

[T.C → O(N*m)
S.C → O(N*m)]

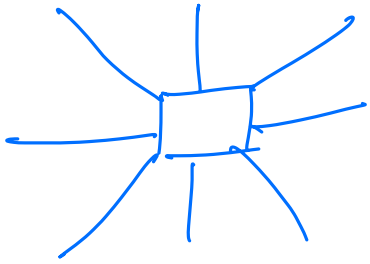
↓

visited[N][m] +
recursive st. size.

Small variation

1	1	0	0	1
0	1	0	1	0
1	0	0	1	1
1	1	0	0	0
1	0	1	1	1

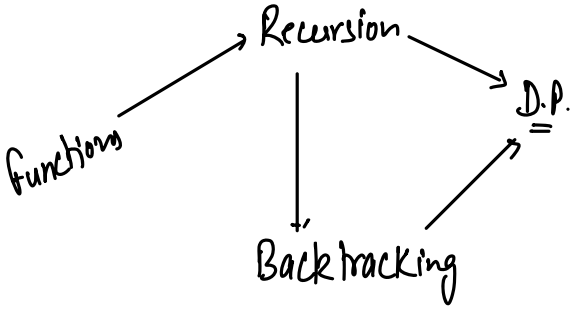
0 → water
1 → land



ans = 2.

dx[7] → { -1, -1, 0, +1, +1, +1, 0, -1 }

dy[7] → { 0, -1, -1, -1, 0, +1, +1, +1 }



Q) Given N courses with pre-requisite of each course. Check if it is possible to finish all the courses.

N=5.

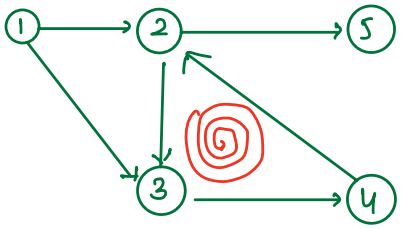
x is a pre-requisite of y ⇒ $x \rightarrow y$

1 ⟶ 2, 3

2 ⟶ 3, 5

3 ⟶ 4

4 ⟶ 2



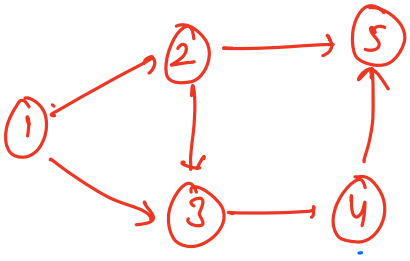
1, 2, 3, 4, 5.

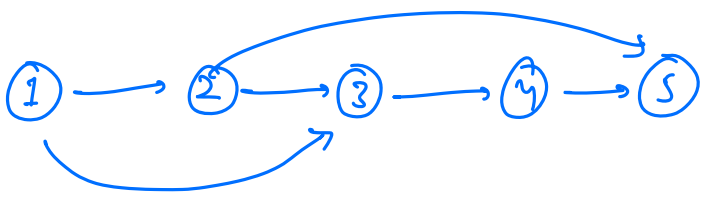
Solution → check if cycle is present or not.

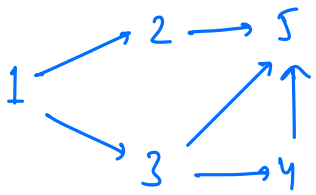
Topological Order / Sorting →

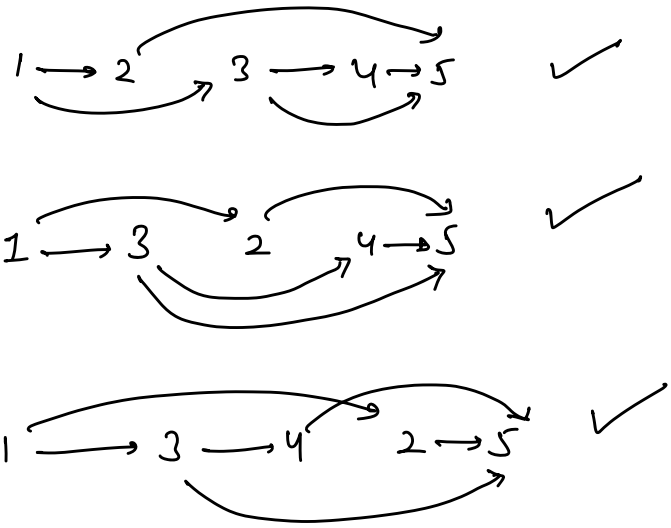
Linear ordering of nodes such that if there is an edge from i to j, then i should be present on l.h.s of j. (i < j)

Directed Acyclic Graph









∴ multiple topological orders are possible.

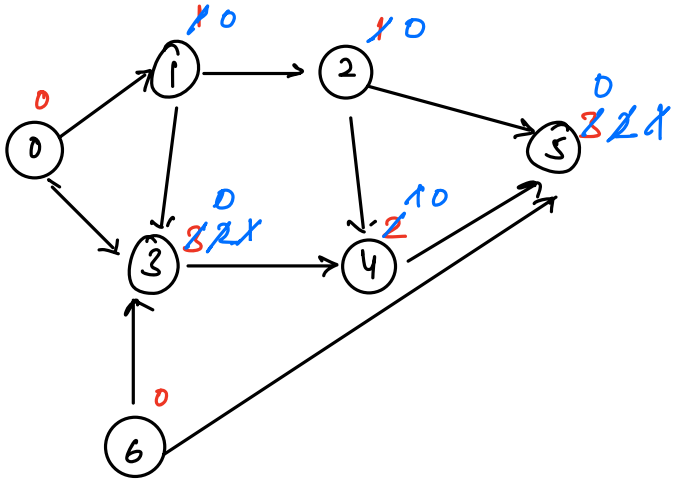
Find topological Order

① Using in-degree

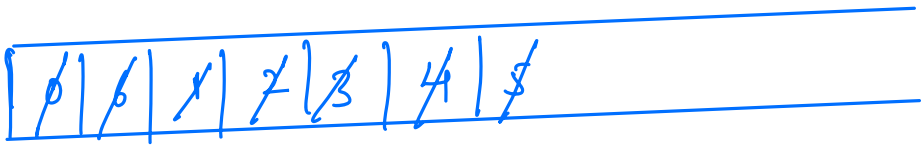
Step-1 Find the in-degree of all the nodes.

```
in[N], ∀i, in[i] = 0;

for( i = 0; i < N; i++) {
    for( int nbr : graph[i]){
        in[nbr] ++;
    }
}
```



Step-2 Insert all the nodes with in-degree 0 in a queue



→ 0, 6, 1, 2, 3, 4, 5

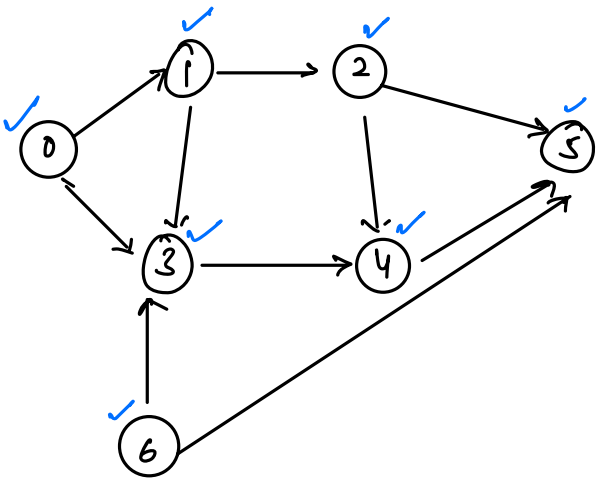
```
for( i=0; i<N; i++){
    if( in[i] == 0) { q.enqueue(i), print(i) }
}
```

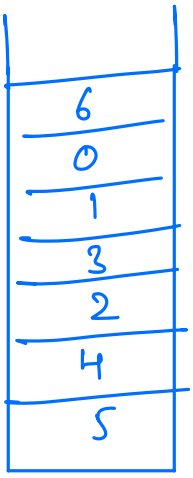
Step-3.

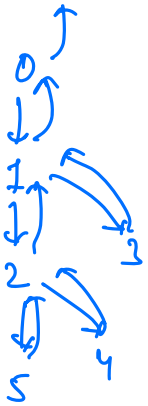
```
while( q.isEmpty() == false){
    x = q.dequeue();
    for( int nbr : graph[x]){
        in[nbr]--;
        if( in[nbr] == 0) { q.enqueue(nbr), print(nbr)};
    }
}
```

T.C → O(N+E)
S.C → O(N)

② DFS + stack.







6

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

```
//Graph → given
boolean visited[N];   // ∀i, visited[i] = false

  Stack<integer> st;

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

//Remove all elements from st & print them.
```

```
void dfs( Graph, src, visited[N], Stack<Integer> st){
    visited[src] = true;
    for( int nbr : graph[src]){
        if( visited[nbr] == false){
            dfs( graph, nbr, visited);
        }
    }
    st.push(src);
}
```

T.C → O(N+E)
S.C → O(N)

↓

visited[N] +
max size of stack +
Stack<Integer> st

Disjoint Set Union

intersection

$S1 \cap S2 \rightarrow \{-\}$

$S1 \cup S2 \rightarrow \{1, 2, 3, 4, 8, 6\}$

union

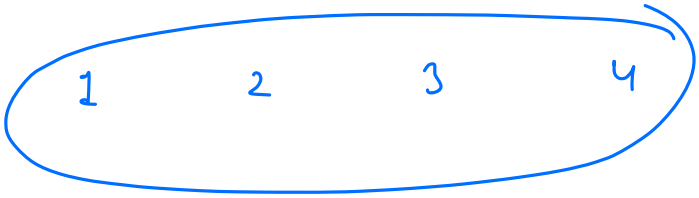
Q Given N elements. Consider each element as a unique set and perform multiple queries.

In each query, check if (u, v) belong to different sets.

if Yes → merge the two sets & return true;

if No → return false

N = 4.



Queries.

(1,2) → true

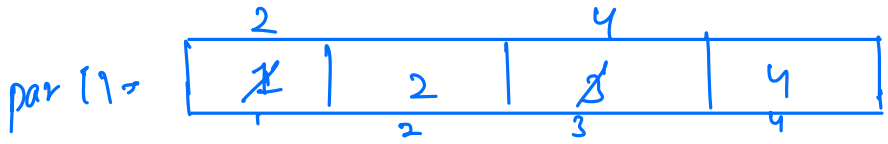
(3,4) → true

(1,2) → false.

(1,3) → true

(2,3) → false

idea → Consider every element as a tree where every node points to its parent node and root points to itself.



Queries.

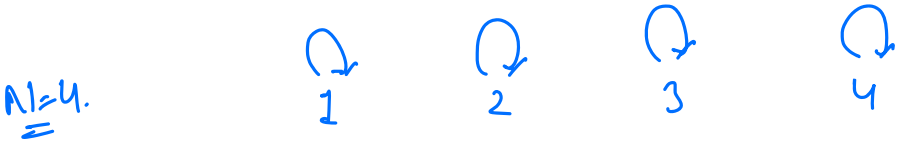
(1,2) → true

(3,4) → true

(1,2) → false

(1,3) → false

(2,3) → false



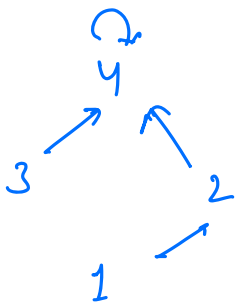
par[1] = 2 or par[2] = 1

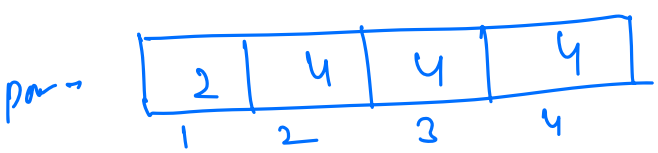
par[3] = 4 or par[4] = 3

par[1] = 3 or par[3] = 1 Both are wrong.

∴ We can only update the parent of root node.

par[2] = 4 or par[4] = 2 ;





```
// par[N], ∀i, par[i] = i;

int root( int x) {
    while( parent[x] != x) {
        x = parent[x];
    }
    return x;
}
```

T.C → O(Ht. of tree)

```
boolean union ( int x, int y) {
    rx = root(x);
    ry = root (y);
    if ( rx == ry ) { return false }
    else {
        par[rx] = ry ;   // or  par[ry] = rx
        return true;
    }
}
```

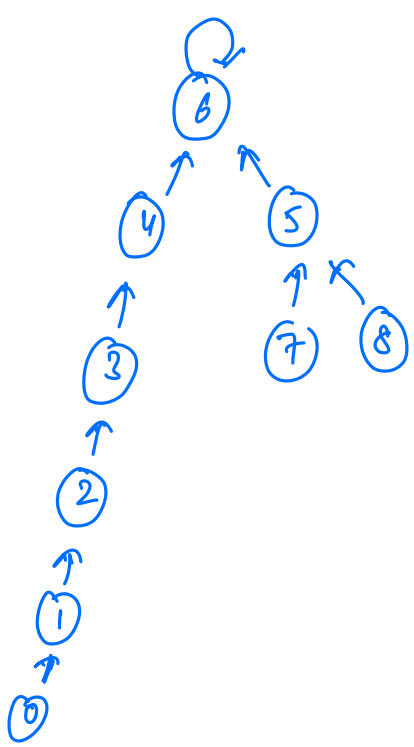
T.C → O(Ht of tree)

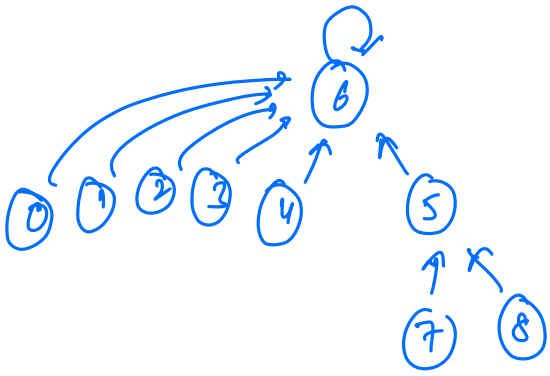
optimise D.S.U

→ Union by Rank $[\log_2 N]$ (#todo)

→ Path Compression - O(1)

path compression





Query → (0, 6)

root(x) → K steps

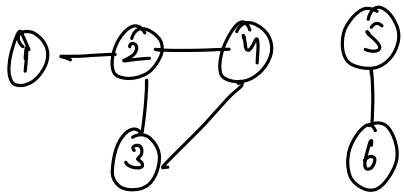
for the next K-nodes → O(1)

```
int root ( int x) {
    if( par[x] == x) { return x }
    res = root( par(x) );
     par [x] = res;
     return res;
}
```

[T.C → Amortized O(1)]

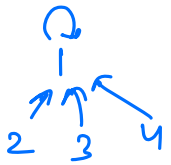
Applications of DSU

① Check if a (undirected) graph is cyclic or not →



→ For all nodes, consider them as independent sets.

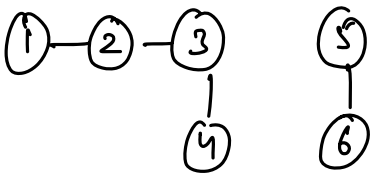
→ For all the edges, take union(u,v)



↓

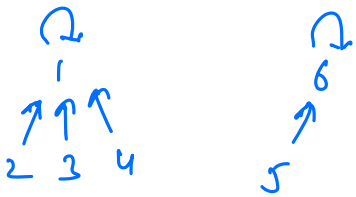
```
if (union(u,v) == false) {
    cycle is present
}
```

② Check if a graph is connected or not →



→ For all nodes, consider them as independent sets.

→ For all the edges, take union(u,v)



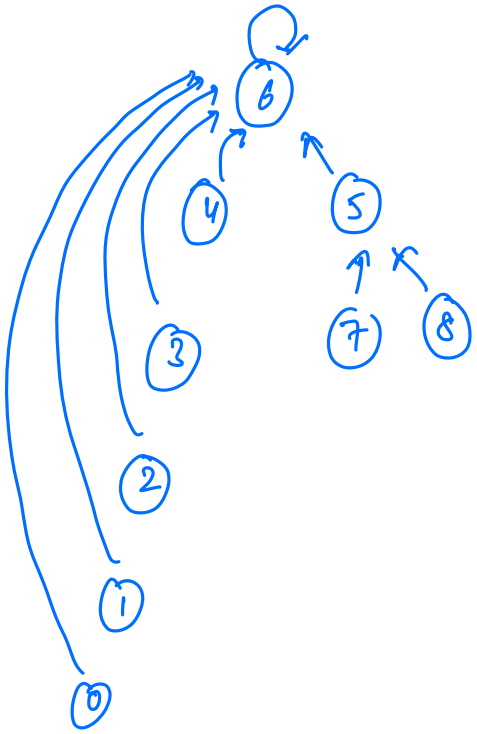
→ if all nodes are having the same root

⇒ graph is connected.

otherwise, ⇒ graph is disconnected.

③ Minimum Spanning Tree -

---- To be continued....




```
int root ( int x) {
    if( par[x] == x) { return x }
    res = root( par(x));
     par [x] = res;
    return res;
}
```

par[] -

0	1	2	3	4	5	6	7	8
6	6	6	6	6	6	6	5	5

root(0) ↑6
↓ ↑6
root(1)
↓ ↑6
root(2)
↓ ↑6
root(3)
↓ ↑6
root(4)
↓ ↑6
root(6)

root(0) → 5 steps
root(1) → 1 "
root(2) → 1 "
root(3) → 1 "
root(4) → 1 "