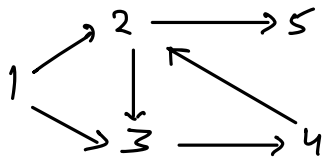
Welcome ☺

Agenda: Topological sort
DSU

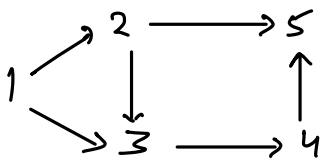
Q Given N courses with pre-requisites of each course. Check if it is possible to complete all courses.



Cyclic graph ⇒ Ans = false.

Acyclic graph ⇒ Ans = True.

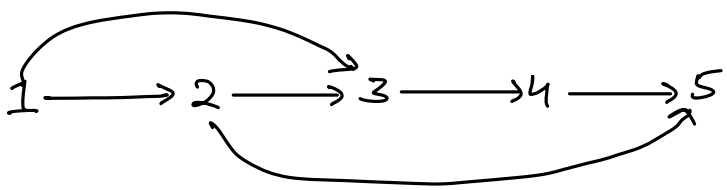
Q If it is possible to complete all the courses, find any one order in which you can complete all the courses.

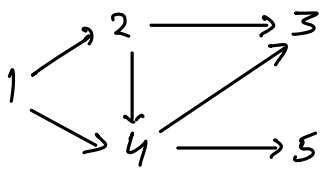


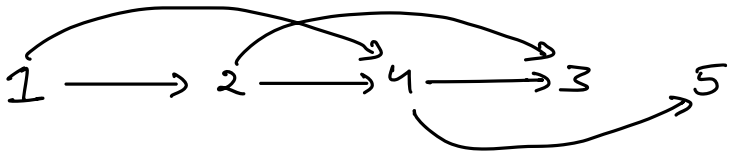
Ans = 1 → 2 → 3 → 4 → 5

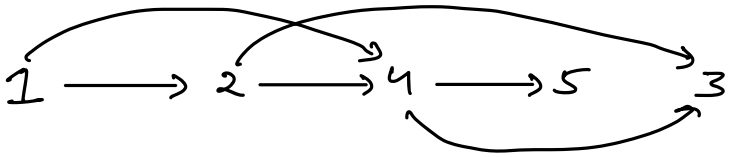
Directed Acyclic Graph (DAG)

Topological sort ⇒ linear ordering of nodes s.t if there is an edge from node i → j then i will be on left of j





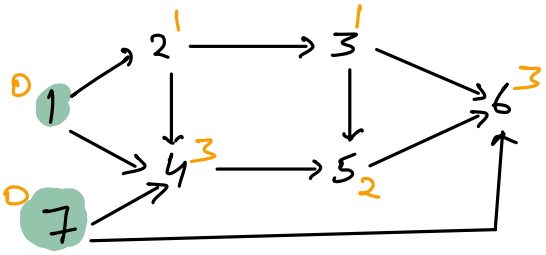




There can be more than 1 topological sort / order for any DAh.

Find Topological sort.

1) Left to Right



1) Compute indegrees ∀ nodes.

```
∀i, in[i] = 0
for ( u → 1 to N)
{
    for ( v : adj[u] ){ u ⟶ v
        in[v]++
    }
}
```

T.C O(N+E)

2) Insert all nodes with indegree 0 in set / array / queue.

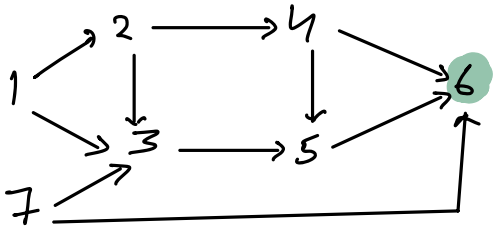
3) Get any node from the above set/array, print it (and) & update the indegree of adjacent nodes (decrease by 1)

4) If updated indegree of any adjacent node becomes 0, insert that nodes in set/array/queue and repeat from step 3 till set/array is empty.

T.C → O(N+E)
S.C → O(N)

---

2) Right to left



Topological sort can end if outdegree = 0 for that node.

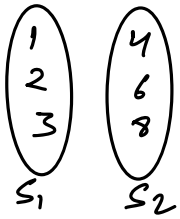
→ length of adj. list

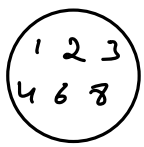
```
∀i  vst[i] = false.
for ( i → 1 to N)
{
    if ( !vst[i]) dfs(i)
}

void dfs(u)
{
    vst[u] = true
    for( v : adj[u])
    {
        if ( !vst[v]) dfs(v)
    }
    print(u) // prints right to left // for L→R store in stack.
}
```

T.C → O(N+E)

S.C → O(N)

Disjoint Set Union (DSU)



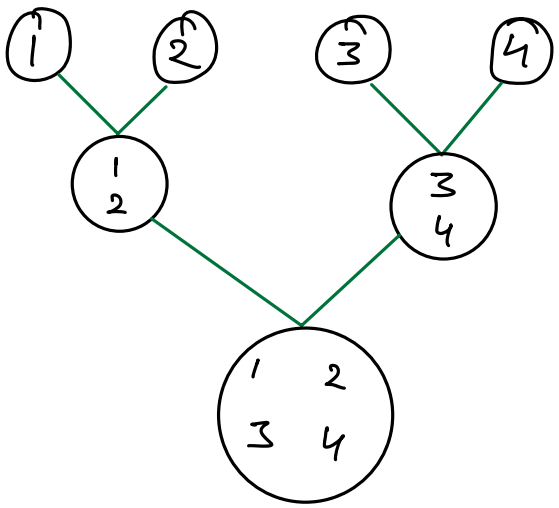
Union $S_1 \cup S_2$ 

Intersection $S_1 \cap S_2 = \phi$ or $\{\}$

disjoint sets

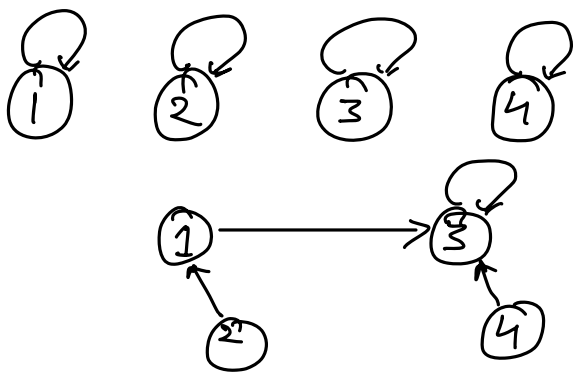
Q Given N elements, consider each element a unique set and perform multiple queries.
In each query, check if (u,v) belongs to different sets.
If yes, merge the 2 sets and return true., else return false.

eg:



Queries.
(1,2) → True.
(3,4) → True.
(1,2) → false
(1,4) → True
(2,3) → false

Idea ⇒ 1) Consider every set as a tree.
2) ∀ nodes, node points to its parent
3) ∵ for root node, there is no parent, root points to itself.



Queries.
(1,2) → True
(3,4) → True.
(1,2) → false
(2,4) → True.

(1,2) ⇒ parent[1] = 2
or
parent[2] = 1

(1,4) ⇒ parent[4] = 1 ✗
or
parent[1] = 4 ✓

⇒ It is only possible to update the parent of root node.

How to find root for a given node.

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

T.C ⇒ O(H) height

worst case O(N)

Check and merge for query (u,v)

```
bool union ( u , v)
{
   int x = root(u)
   int y = root(v)
   if ( x == y) return false.
   parent[x] = y
   return true.
}
```

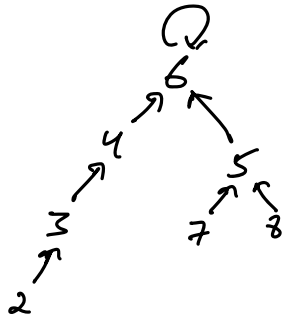
T.C ⇒ O(H+H)

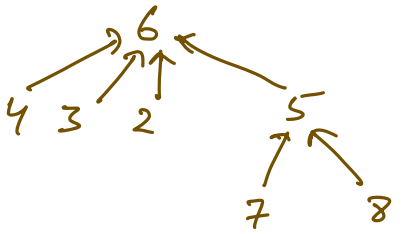
worst ⇒ O(N)

How to optimize. T.C

⇒ 1) Union By Rank. ⇒ (H.W)

✓✓ 2) Path compression





T.C ⇒ O(K) for K nodes

Amortised T.C ⇒ O(1)

Code

```
int root ( int n)
{
    if ( n == parent(n))
        return n
    parent[n] = root ( parent[n])
    return parent[n]
}
```

Applications of DSU

1) Check if the given graph is connected ?

   1) Undirected graph ⇒ Travel complete graph from any node.

   2) Directed graph ⇒

      a) Consider every node as a unique set

      b) ∀ edges (u,v) → take union (u,v)

      c) If root ∀ nodes is same ⇒ connected graph
         else ⇒ disconnected graph.

T.C O(E+N) S.C → O(N)

2) Detecting cycle in undirected graphs ?

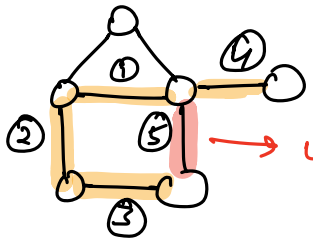
a) Consider every node as unique set

b) #edges (u, v) → take union (u, v)

if for any edge → union returns false

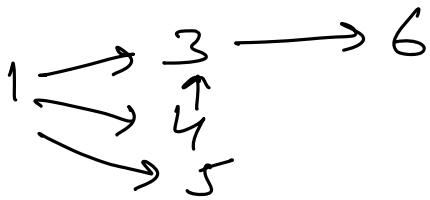
⇓

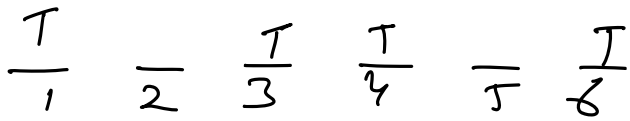
cycle exists.

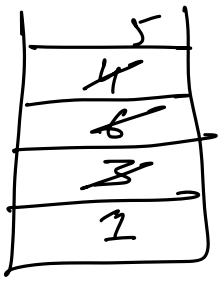


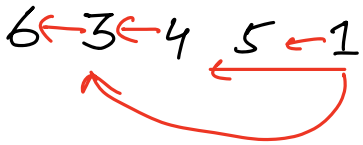
→ will return false.

T.C ⇒ O(N+E) S.C ⇒ O(N)









LIS Binary Search