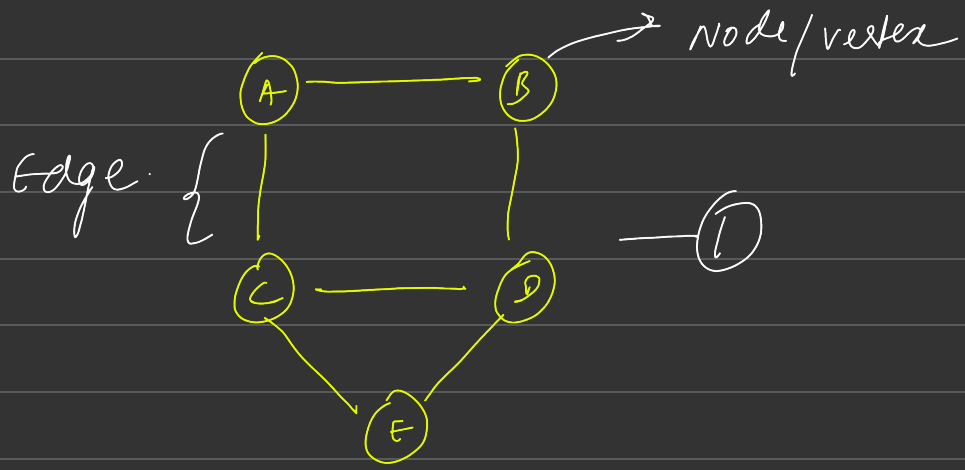
GRAPHS

→ Index:

Graph Datastructure

→ A Graph is a datastructure that consists of a set of nodes (vertices) connected by edges



→ Types

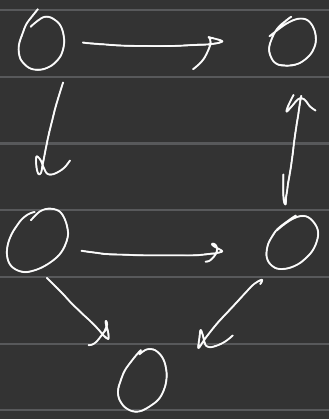
(a) undirected Graph — (1)

(b) Directed Graph

↳ All edges are directed

(c) Cyclic Graph

↳ starting & ending at same node.



- Directed Graph.

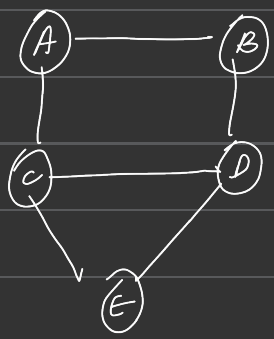
→ Rules for a DS to be called as a Graph

⇒ Degree of a Graph

→ Degree of a graph is total number of edges connected to it.

→ In a directed graph degree is Equal to the sum of incoming edges + sum of outgoing edges.

Ex:-



Deg(A) = 2
Deg(D) = 3.
Deg(E) = 2.

(a) Indegree(V) :- no. of incoming edges to node V
(b) outdegree(V) :- no. of outgoing edges from node V.

⭐ ⇒ Handshaking Lemma (for degree).

→ In an undirected graph the sum of degrees of all nodes is Equal to twice the number of edges.

$$\sum deg(V) = 2E$$

→ Why?

→ In an undirected graph each edge is connected to 2 nodes.

→ for directed Graph.

$$\sum Indegree = \sum Outdegree = E$$

for all nodes.

⋆ ⇒ Representation of a Graph [C++].

→ Input format :-

(i)

[n, m]

Nodes — Edges.

(ii)

N-edges

1 2
1 3
3 4

→ Ways to store the graph input

Adjacency Matrix — Adjacency List.

→ if a[i][j] = 1

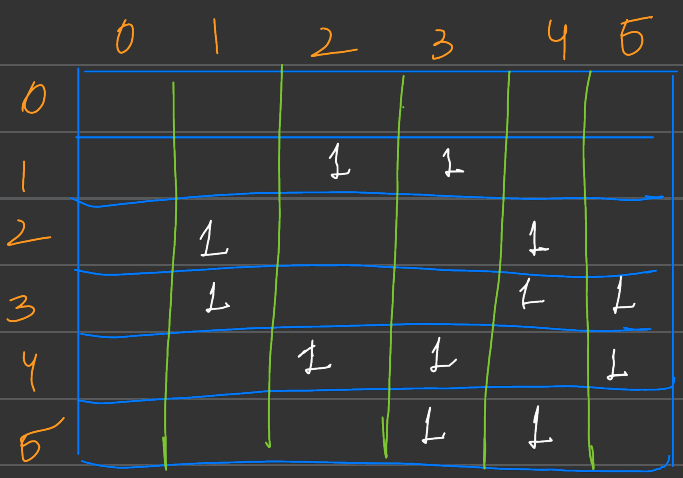
There is an edge b/w i → j.

→ Example :-

(a) Input :- $n = 5$, $m = 6$.

Edges :-

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



(very costly).

(N^2)

A → code for adjacency matrix

```
int n, m;
cin >> n >> m;
int adj[n+1][m+1];

for (i = 0 → m) {
    int u, v;
    cin >> u >> v;
    adj[u][v] = 1;
    adj[v][u] = 1;
}
```

☆ ⇒ Adjacency List.

SC :- O(2E).

→ Representation :-

```
vector<int> adj[N+1];
```

→ Example :-

Input :- $n = 5$, $m = 6$.

Edges :-

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

List ⇒

0

1 → {2, 3}

2 → {1, 4, 5}

3 → {1, 4}

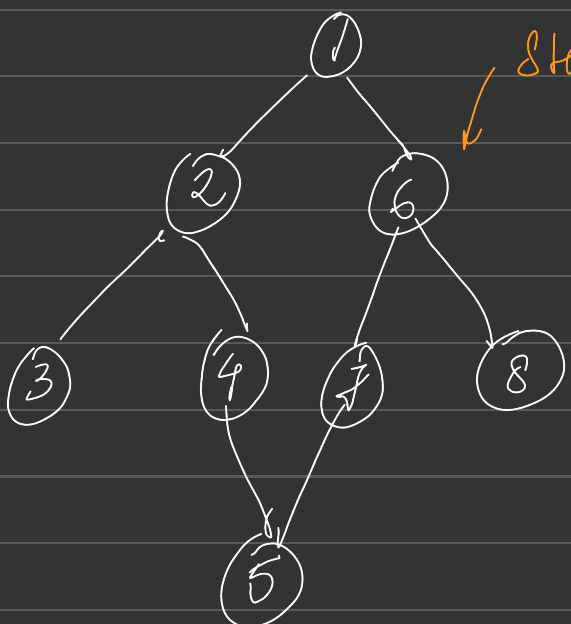
4 → {3, 2, 5}

→ code

```
int n, m;
cin >> n >> m;

vector<int> adj[n+1];

for (i = 0 → m) {

    int u, v;
    cin >> u >> v;

    adj[u].push_back(v);
    adj[v].push_back(u);

}
```

⇒ Graph Traversals.

(A) Breadth first search (BFS).



BFS :- 6 1 7 8 2 5 3 4

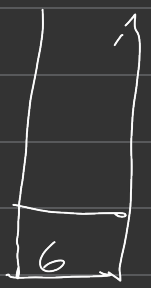
Starting node L1 L2 L3

→ Algorithm

1. Initialise queue with starting node & initialize visited array with false

vis:- | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
0 1 2 3 4 5 6 7 8

→ output :- [6]

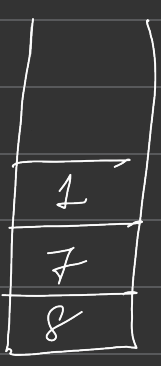


2. pop all queue Elements and push the popped elements adjacent nodes.

→ pop 6 , push (1) , push (7) , push (8).

→ mark pushed nodes as visited

vis:- | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
0 1 2 3 4 5 6 7 8



3. Repeat same process for all remaining nodes.

pop(1), pop(7), pop(8).

push(2)
push(6) ✗
already visited

push(5)

Output :- [6, 1, 7, 8]

pop(2)

push(3) push(4)

pop(5)

✗

output :- [6, 1, 7, 8, 2, 5]

pop(3) pop(4)

x x

→ Queue is empty now

→ Output :- [6, 1, 7, 8, 2, 5, 3, 4].

#.

* TC :- O(N) + O(2E).

* SC :- O(N) + O(N) ≃ O(N)

queue vis

★ Depth - first search (DFS).

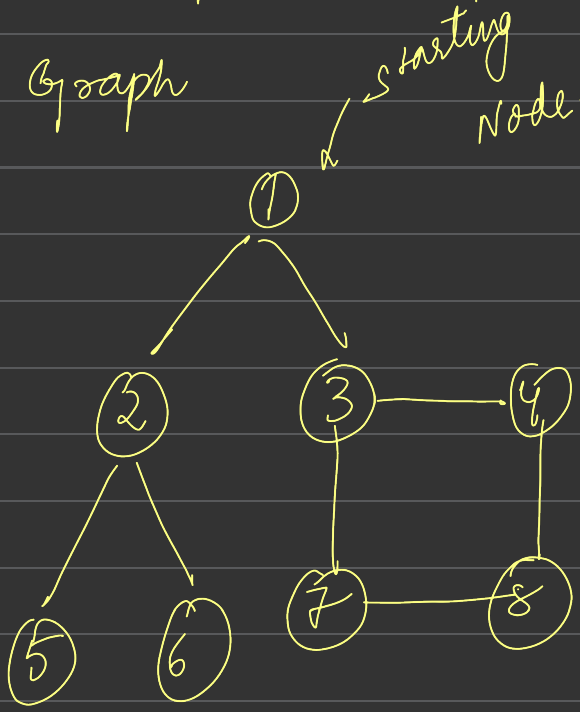
→ Instead of traversing level wise i.e., completing all nodes at a level, start from one node and start visiting all of its adjacent, then adjacent nodes adjacent and so on . . . . .

→ Algo / process -

1. visit the starting node, then start to visit its adjacent nodes as specified in the adjacency list
2. when visiting any node, mark it visited
   → Iterate through adjacency list, if neighbour is not visited then visit that node else skip.

→ Example :-

Graph



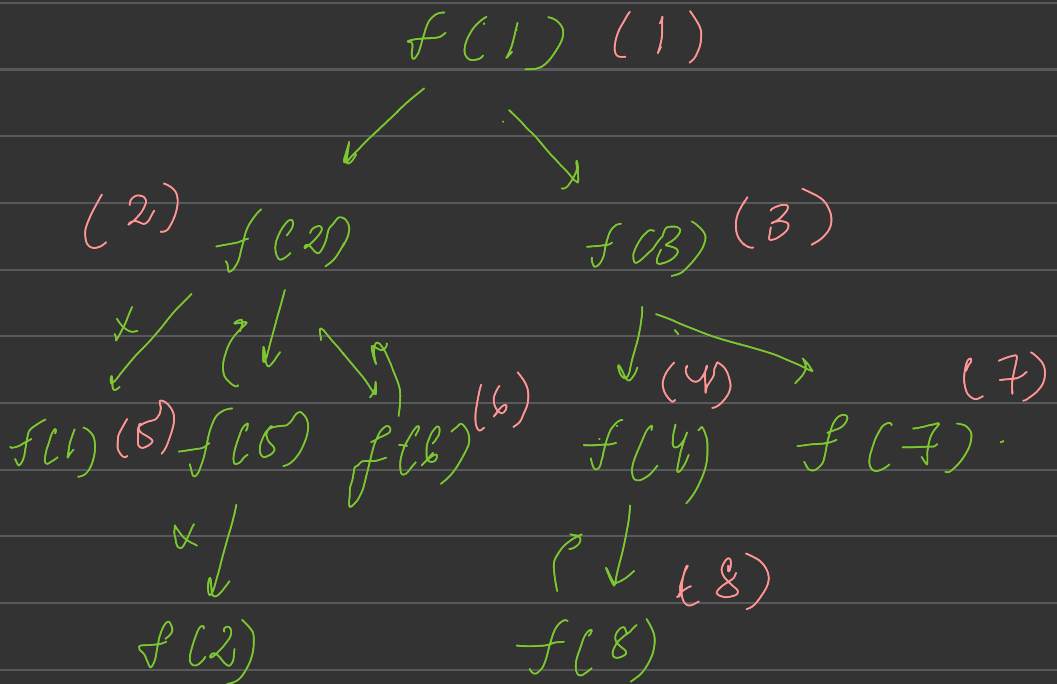
Adj List

1 → {2, 3}
2 → {1, 5, 6}
3 → {1, 4, 7}
4 → {3, 8}
5 → {2}
6 → {2}
7 → {3, 8}
8 → {4, 7}.

vis :-

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

Recursion Tree



1 → 2 → 5 → 6 → 3 → 4 → 8 → 7

* Code.

```
void dfs(int node) {

    vis[node] = true
    dfs.add(node);

    for(auto it : adj[node]) {

        if(!vis[it]) dfs(it);

    }

}
```

$\sum$ neighbour = $\sum$ degree = $2E$

* TC :- $O(N) + O(2E)$

* SC :- $O(N) + O(N) \simeq O(N)$.

visited R.S.S

☆ Topological sort.

(DAG).

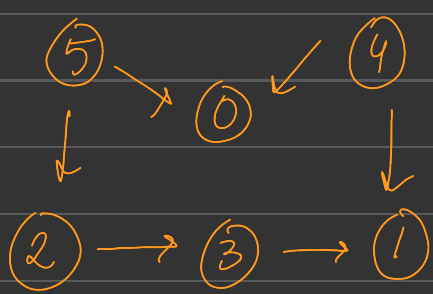
→ A Toposort exists only for Directed Acyclic Graph

⇒ DAG.

→ A directed graph without any cycles.

⇒ Topo sort :- Linear ordering of vertices such that if there exists an edge between u and v, u always appears before v in that ordering.

Ex :-



Linear ordering :-

(a) 5 → 4 → 2 → 3 → 1 → 0

(b) 4 → 5 → 2 → 3 → 1 → 0

→ directed

TC :- O(V + E)

SC :- O(V) + O(V) ≃ O(V).

↓ stack ↳ visited Array

⇒ Process:

→ Same approach as Dfs algo, but before finishing up the function call, push the current dfs node into stack.

⇒ Why Stack?

→ When we perform Dfs for a node :-

(i) we go deep down its descendants
(ii) once all nodes are fully processed we safely include that node in our ordering

→ This post-order Dfs gives us the reverse order of topological sorting ↘ Lifo

⇒ Bfs Approach (Kahn's Algorithm)

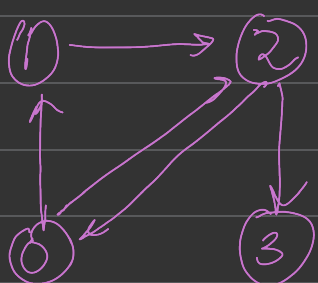
(i) Maintain an Indegree array
(ii) Start Bfs for nodes whose indegree = 0
(iii) for any adjacent node, reduce indegree
if after reduction indegree == 0
↳ push to queue

(iv) Repeat until Q is empty.

(Q) Detect a cycle in directed Graph

PS:- Given a directed graph detect if there exists any cycle in that graph.

Input :-



output :-

True.

→ Toposort Approach.

→ Toposort ordering exists only for a DAG.

∴ If we take toposort ordering for a graph with cycles, toposort() = N

∴ Use this to check if toposort == N

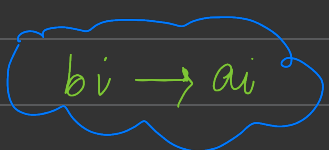
T ↙ ↘ F

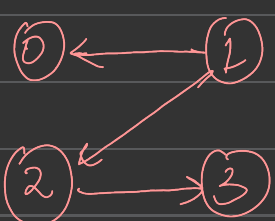
No cycle cycle.

(Q) course schedule.

→ Ps:- you are given 'N' courses to take, but there are some pre-requisites also given

→ pre-req [ai, bi] indicates that to take course ai, you need to first finish course bi





→ Input :- N=4, p=3

pre-req :- [[0,1], [2,1], [3,2]].

→ output :- True (1 → 0 → 2 → 3).

→ Approach (# Toposort).

→ Why Toposort?

→ we are dealing with orderings

(i) use Kahn's algo (Toposort + Bfs) to check if there exists any cycle or not

(ii) if no cycle, course order is possible else not.

(Q) course schedule-2.

→ same question as before but just instead of checking if topo sort is possible or not, instead return the ordering of topo sort.

(Q) Find Eventual Safe States

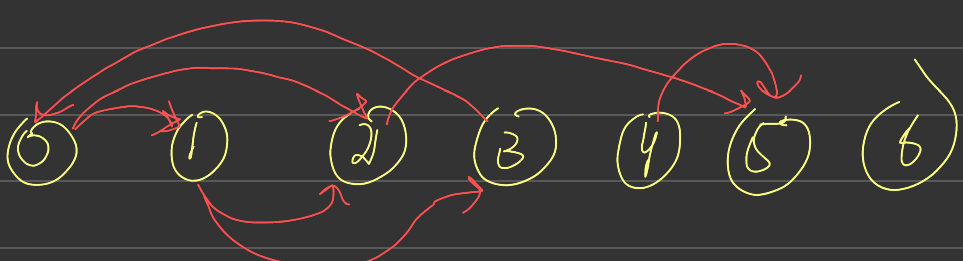
Ps:- Given a graph, return the list of all nodes.

→ A node is said to be safe node iff

(i) The node is a terminal node, meaning there are no outgoing edges from that node

(ii) A node whose all outgoing edges leads to safe nodes.

→ Input :- graph :- [[1,2], [2,3], [5], [0], [5], [], []].



→ output :- [2, 4, 5, 6].

→ Approach :-

(Q) Why topoSort ?

→ The Indegree concept used in Kahn's algorithm (Bfs) can help us find terminal nodes and path leading to terminal node.

→ Reverse all the edges

→ find indegree :- (Actually gives all terminal nodes).

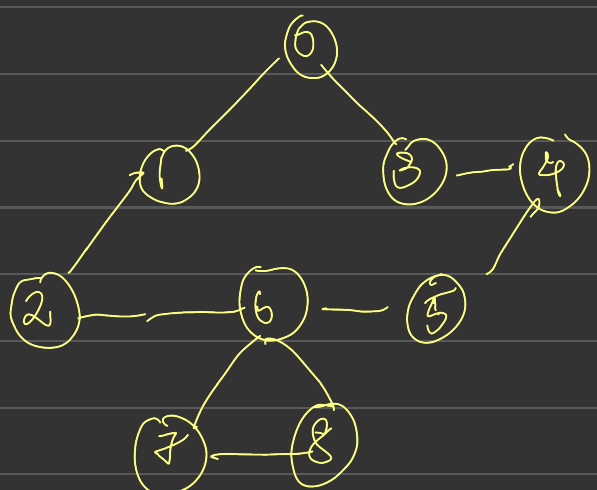
→ Apply topoSort.

☆ Shortest Path Algorithm

(Q.) Shortest path in undirected graph having unit weight

PS:- Given an undirected graph, having unit distance to each of its adjacent vertices find the shortest path source node to all the other nodes.

→ Input :-



→ output :- [0, 1, 2, 1, 2, 3, 3, 4, 4].

→ Approach -

→ Use standard Bfs / Dfs with some tweaks

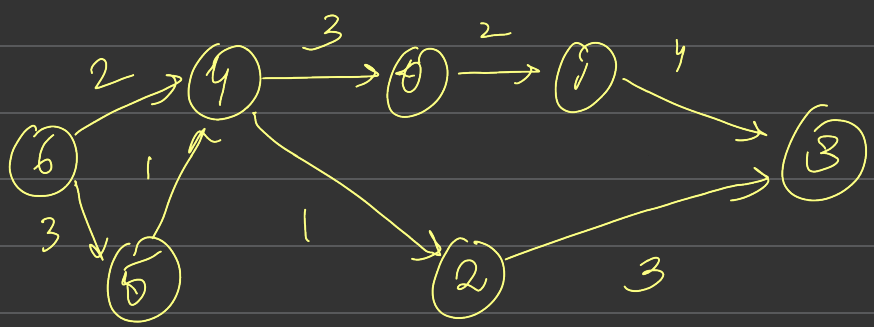
(i) Instead of vis array use dist {∞} array
(ii) If dist at current node is less than previously visited node update distance

```
if ( dist[node] + 1 < dist[adjNode]){

dist[adjNode] = 1 + dist[node];

}-
```

(Q) Shortest path in a DAG:

PS :- Given a DAG, find the shortest path from source node to all the other nodes.

→ Input :- srcNode = 6.



→ Output :- 5 7 3 6 2 3 0

(0 1 2 3 4 5 6)

→ Approach

(i) Step 1 :- find the topoSort of given DAG.

Why TopoSort ?

→ Before going to the next adjacent nodes, if we can have visited all the preceeding nodes and also calculate the weights for those nodes, from current node it is then easier to calculate the weights for next non visited adjacent nodes.

(ii) Relax the edges.

→ simple weight updation

```
if ( dist[node] + wt < dist[v] ){
    dist[v] = wt + dist[node];
}
```

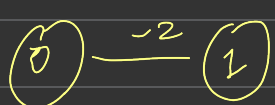
Dijkstra's Algorithm

→ DJ's algorithm is used to find the shortest path from source node to all the other reachable nodes in an undirected graph without any negative weights

→ Why does it not work in case of -ve weights?

→ If we apply DJ's algo on negative weights, we will fall under ∞ loop, because the weight will keep on reducing

Ex:-



$0 \xrightarrow{-2} 1$, $1 \xrightarrow{-4} 0$

$\rightarrow -4, -6 \cdots \infty$.

→ DJ's algorithm is plain BFS, with :-
  ↳ instead of Q, we use priority queue
  ↳ some extra computations to calculate min. distance.

→ why priority queue instead of regular Q?

  → when we use priority queue with different rating factor as distance, we can do better computations with small distances
  → Reduces numbers of computations of different paths.

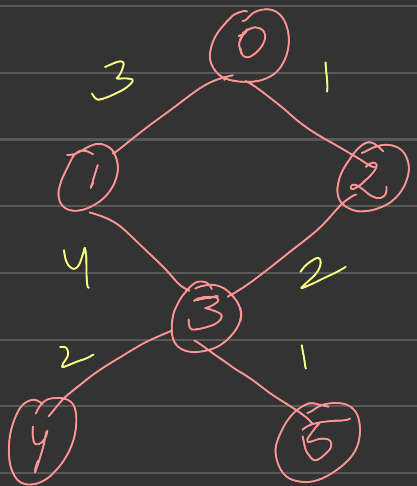
↳ Pseudo code

```
while ( !pq.empty ) {                    → O(V)
    dis, node = top();                   → log(heapsz)
    for (adjNode) & Nedges
        if (dist check())
            update
        { q.push(); log(hsz); }
}
```

TC :- E log V

→ Dry-run :-

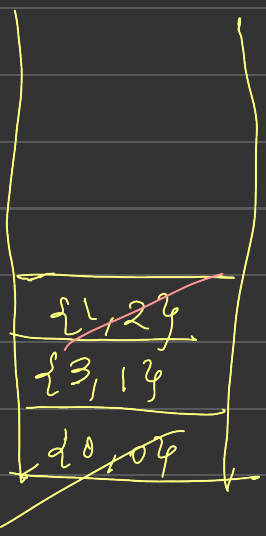
* Graph :-



→ dist :-

0	∞	∞	∞	∞	∞
0	1	2	3	4	5

→ PQ (min Heap)



dist = 0

node = 0

+3 +1

node = 1
dist = 3
dist[1] = 3

node = 2
dist = 1
dist[2] = 1

{1,2} will be picked first

dist = 1

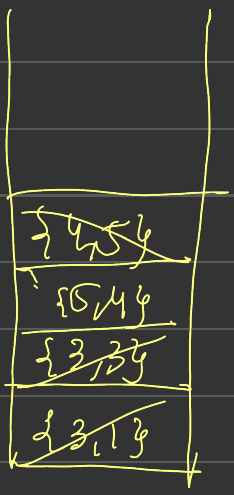
node = 2

+1 +2

discarded as higher weight

node = 0
dist = 2

node = 3
dist = 3
dist[3] = 3



dist = 3

node = 1

+4

node = 0

✗

node = 3

node = 3

dist = 7 (X).

dist = 4

node = 5

✗

dist = 5

node = 4

✗

dist = 3

node = 3

+2

+1

node = 4

dist = 5

dist[4] = 5

node = 5

dist = 4.

dist[5] = 4.

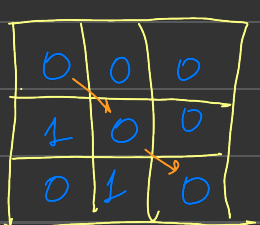
dist :-

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

(Q) Shortest path in a Binary maze

→ PS:- Given a $n * n$ binary maze, return the shortest path length required to reach from $(0,0) \rightarrow (n-1, n-1)$ when we are allowed to travel only cells with '0' values in all 8 directions.

→ Input :-



→ output :- 3

→ Approach:

→ Apply DJ's algorithm with changes:-

(i) Instead of adj list traverse all 8 directions
(ii) only visit cells with 0 value.

(Q) Path with minimum Effort [Min-Max model]

(0,0)

PS:- Given that we have to reach from source to destination $(n-1, m-1)$

→ The requirement is that we need to find the path whose max (absolute different between any 2 adjacent cells) is minimum.

→ Input :-

1	2	3
3	8	4
5	3	5

→ output :- 1

(i) $1 \xrightarrow{1} 2 \xrightarrow{1} 3 \xrightarrow{1} 4 \xrightarrow{1} 5 \rightarrow \max(1, \cdots 1) = \boxed{1}$

(ii) $1 \xrightarrow{2} 3 \xrightarrow{5} 8 \xrightarrow{5} 3 \xrightarrow{2} 5 \rightarrow \max(2,5,5,2) = \boxed{5}$

$\boxed{\min = 1}$

→ Approach → TC: $O((4+m+n)\log(m+n))$ (E, V)

→ Standard DJ's algorithm with some changes

(i) finding max absolute difference

```
maxDiff = max(curDiff, h[nR][nC] - h[r][c]);
```

(ii) Picking the min. of max absolute diffs.

```
if (maxDiff < dist[nR][nC]) {
    maxDiff = dist[nR][nC];
}
```

(Q) Cheapest flight within K stops.

→ Ps:- we are given with source, destination and K → maxStops.

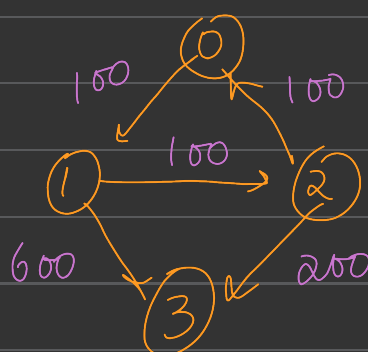
→ we need to travel via flight from src → dest with at most K stops and minimum price possible.

→ Input :-

src = 0

dst = 3

K = 1.



→ output :-

700

(0 → 1 → 3).

→ Approach:

→ DJ's algorithm with changes as:-

(i) distance will not be the primary key
it can happen that we reach the destination with minimal cost but exceed the stops required

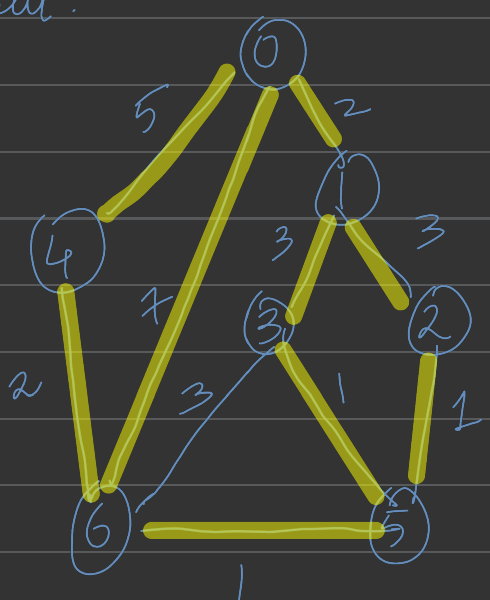
(ii) stops will be the primary key.

(iii) instead of priority_queue we can use regular queue here, as stops
↓
stops+1

(Q) Number of ways to arrive at destination =

→ PS :- Given a 0-based indexed undirected graph
Return the number of ways required to reach destination (n-1) with minimum distance.

→ Input :-

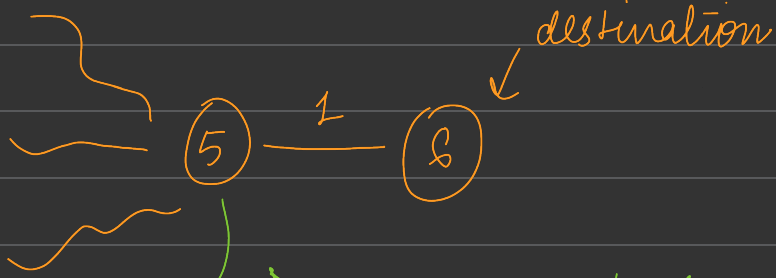


→ output :- 4.

(minDistance = 7).

→ Approach

* The main intution here is that we should not only just track the number of ways required to reach the destination
  BUT also the ways required to reach a node that lead to destination.



→ No. of ways to reach 5 at min distance = 3

⇓

No. of ways to reach 6 at min Distance = 3

→ Above approach can be implemented using an array :- numWays and updation would happen at :-

```
if ( dist [node] + edgeW == dist [adyNode]) {

    dist [adyNode] = dist[node] + edgeW;
    numWays [adyNode]
                =
    numWays [adyNode] + numWays [node];

}
```

(Q.) Minimum multiplications to reach end

→ PS:- Given start, end, arr. at each step start is multiplied with any number in the array. for any numbers of times and we apply mod 10^5 after that.
→ Return the min number of steps required to reach from start to end

→ Input :- start = 7 end = 66175
arr :- [3, 4, 65].

→ Output :- 4

$7 * 3 = 21$, $21 \% 10^5 = 21$
$21 * 3 = 63$, $63 \% 10^5 = 63$
$63 * 65 = 4095$, $4095 \% 10^5 = 4095$
$4095 * 65 = 266175$, $266175 \% 10^5$
$= 66175$

→ Approach :-

→ How graph ?

→ Start is one of the nodes
→ and we can get upto 9999 nodes upon multiplying start to numbers present in the array

∴ Nodes :- [start .... 9999].

→ Now the goal is to reach end in min steps, with multiplication & mod ops....

→ Apply DJ's.

(Q) Swim in Rising Water

→ PS:- Given a square grid of size $n * n$, the goal is to swim from $(0,0)$ to $(n-1, n-1)$

→ But we can only swim ⇔ adjacent cell if the elevation of that cell <= current Time

→ Each cell represents an elevation Time

→ Input:-

0	1	2	3	4
24	23	22	21	5
12	13	14	15	16
11	17	18	19	20
10	9	8	7	6

→ ouput :- 16 (from elevation 16 we can reach the destination via specified route).

☆ => Approach:

→ Standard DJ's algorithm with changes in relaxation

→ Relaxation changes

↳ Here instead of checking for minDistance, we need to see, to travel to adjacent cell, if we the right Elevation

=> `int newTime = max(curTime, grid[nR][nC])` ← from Heap.

```
if (newTime < dist[nR][nC]) {
    dist[nR][nC] = newTime;
    pq.push({newTime, {nR, nC}});
}
```

* BELLMAN FORD ALGORITHM.

→ Single source shortest path.

→ Works in the case of -ve weight

→ Bellman-ford only works for directed graph.

→ TC :- O(V*E), SC : O(V).

↳ dist[].

⇒ Process.

→ Do the relaxation of all the edges for N-1 times sequentially.

Vertices ↙

→ why N-1?

→ Each iteration will help to compute the distance of nodes.

→ The first node leading to calculate distance in worst case might be the last edge

→ At max it will take us N-1 edges to reach from start to end.

Ex :-

Edges

(u, v, wt)

Iteration 1

(3, 4, 1)

dist[3] + 1 < dist[4]
∞ skip

(2, 3, 1)

dist[2] + 1 < dist[3]
∞ skip

(1, 2, 1)

dist[1] + 1 < dist[2];
∞ skip

(0, 1, 1).

dist[0] + 1 < dist[1]
↳ dist[1] = 1

1st iteration

→ in 2nd iteration dist[1] will be used to get dist[2]

⋮

→ in 4th iteration dist[3] will be used to get dist[4]

N-1

✱ Detecting - ve weight cycles?

→ If Negative cycle exists, even after the $N-1^{th}$ iteration, we would still be updating distance array.

✱ Code.

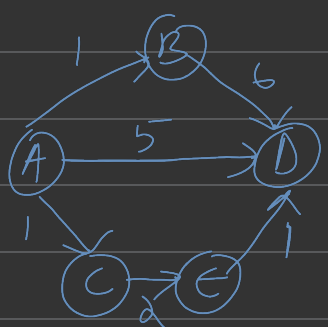
```
for (i = 0 → V) {
    for (auto it : edges) {
        int u = it[0]; int v = it[1];
        int wt = it[2];

        if (dist[u] != 1e9 &&
                dist[u] + wt < dist[v]) {
            dist[v] = dist[u] + wt;
        }
    }
}
for (auto it : edges) {
    if (dist[u] != 1e9 && dist[u] + wt < dist[v])
        { return -1; }
}
return dist[];
```

☆ FLOYD-WARSHALL ALGORITHM

Multi-Source shortest path.

→ for directed graph

→ find all the distances from one source to a destination and take the minimum



Best = 4

(A → C → E → D)

dist[i][j] = dist[i][k] + dist[k][j];

Ex:- [1][3] via 0 → [1][0] + [0][3];

→ TC :- $O(N^3)$.

→ SC :- $O(N^2)$.

→ How to detect -ve cycle.

```
for(node; 0 → V) {
→ of cost[node][node] < 0
}.                    ∴ -ve cycle.
```

⇒ Code :-

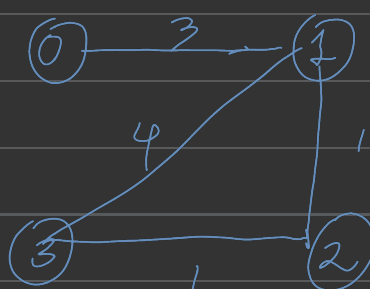
```
for(i=0 → n) {
    for(J=0 → n) {
        if (cost[i][J]==-1) cost[i][J]=1e9;
} }  if (i==J) cost[i][J]=0;
for(k=0 → n) {
    for(i=0 → n) {
    for(J=0 → n) {
        cost[i][J]= min(cost[i][J],
        cost[i][k]+cost[k][J]);
} } }
for(i=0 → n) {
    for(J=0 → n) {
} } if(cost[i][J]==1e9) cost[i][J]=-1;
return cost[][];
```

(Q) Find the city with smallest number of neighbours at a threshold distance

→ PS:- We are given $0 \to n-1$ cities, distance required to reach from city A → city B, we are also given with threshold distance.

→ We need to find a city with minimum number of adjacent cities that can be visited with distance <= distanceThreshold.

→ Input :-



distanceThreshhold = 4.

→ output :- 3
1

3 → city 1 city 2
d = (4) (1).

→ Approach:

→ TC :- O(N3).

(i) Apply floyd warshall to find the min distance of all the cities to all of its adjacent cities

(ii) Iterate through the costMatrix and find reachable cities under threshold distance.

→ for above example: costMatrix :-

	0	1	2	3	
0	0	3	4	5	→ 3 cities under threshold
1	3	0	1	2	→ 4
2	4	1	0	1	→ 4
3	5	2	1	0	→ 3

min (3, 4, 4, 3) = 3

→ city 3

✱ Minimum Spanning Tree

→ Spanning Tree

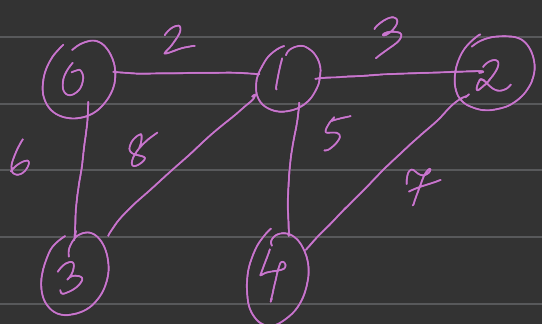
↳ A Tree with 'N' nodes and 'N-1' edges

↳ All nodes are reachable from one another.

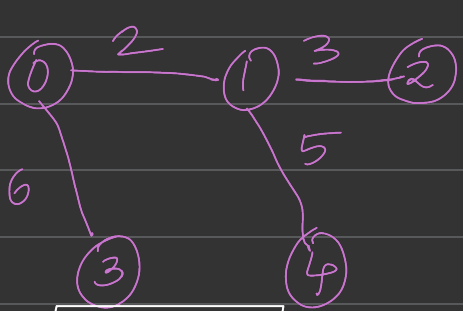
→ Minimum Spanning Tree :-

↳ A Spanning Tree with minimum Total edge weight

→ Tree :-



→ MST.



Wt = 16

✱ Prim's Algorithm

→ DS required :-
  ↳ priority queue ({wt, node, Parent})
  ↳ vis Array.

→ Process :

Why not mark vis during push?
→ we may get less wt later from other node

  ↳ Insert any node to minHeap ({0, 0, -1})
  ↳ Take out Top Element
    ↳ if unvisited
      - Mark visited + Add node to MST [].
      - Add edgeWt to Sum
      - Traverse and add adjacent nodes to minHeap
  ↳ Repeat for all nodes until minHeap is empty.

→ TC :- $O(E \log V)$ → SC :- $O(V-1) + O(V)$.

Disjoint Set

→ Background:-

→ If we have to find if two Elements belongs to same component or not we can do it in linear $O(N+E)$ time using Dfs

→ Same thing can be done in constant time using Disjoint set $O(1)$.

→ operations / functions
- findParent()
- Union (connect nodes).
  - Size
  - Rank

⋆ ⇒ Union By Rank.

→ Required :- → Rank Array (OutDegree) level wise
→ Parent Array
(To find Parent).

→ Pseudocode : union (u, v).

(i) find ultimate parent of u, v [pu, pv].
(ii) find ranks of pu & pv
(iii) Always connect smaller rank to larger

→ TC :- $O(4\alpha) \simeq O(1)$.

Why?
→ To not increase rank/height

Example:

→ Union :- (1,2) (2,3) (4,5) (6,7) (5,6) (3,7).

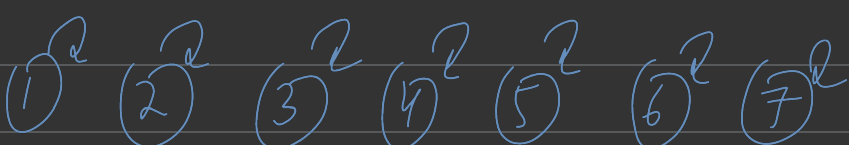
→ Initial config:-

rank :-

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

parent :-

1	2	3	4	5	6	7
1	2	3	4	5	6	7

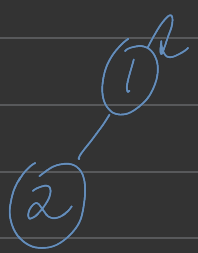


(i) (1,2)

pU(1) = 1 pV(2) = 2

r = 0 r = 0

=> rank same connect anyhow.



parent[2] = 1

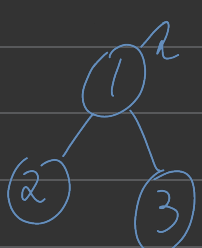
rank[1] = 1

(ii) (2,3)

pU(2) = 1 pV(3) = 3

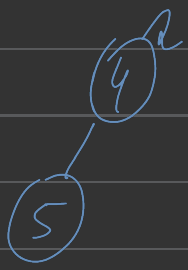
$r_1 = 1$ $r_1 = 0$

$r_1 > r_2$ ∴ connect 3 to ultimate parent of 2.



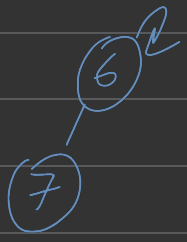
parent[3] = 1

(iii) (4, 5)



rank[4] = 1
parent[5] = 4

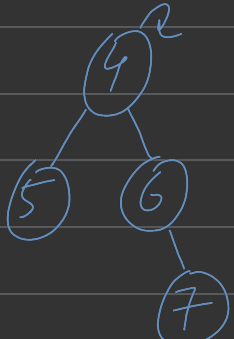
(iv) (6, 7)



rank[6] = 1
parent[7] = 6

(v) (5, 6)

pU(5) = 4 pV(6) = 6
r = 1 r = 0
connect 6 → 4



rank[4] = 2
parent[6] = 4

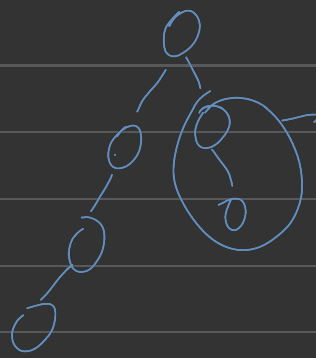
→ Check if (5) & (7) belong to same component

(i) find ultimate parents of u, v
(ii) if same up & uv then same component else not

→ Path compression

→ A Node directs to ultimate parent instead of parents

→ We cant reduce the rank during path compression

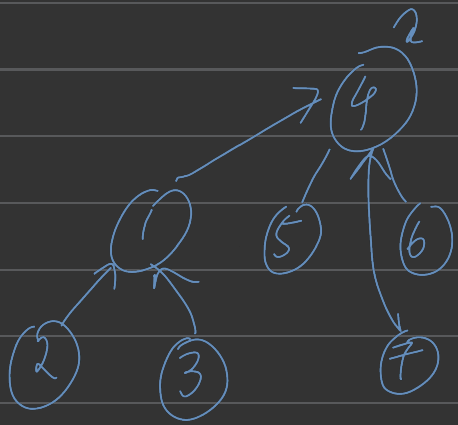


→ path compression → only here ⇒ rank remains intact

(vi) (3, 7)

$pu(3) = 1$ $pu(7) = 4$

$r = 1$ $r = 2$



=> Path compression

```
findParent(u)
{
    if (u == parent[u])
        return u;

    return parent[u] = findParent(parent[u]);
}
```

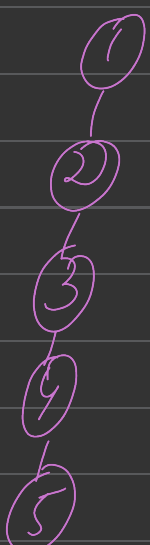
f(5)
↓ ↑ P[5]=1

f(4) P[4]=1
↓ ↑ 1

f(3) P[3]=1
↓ ↑ 1

f(2) P[2]=1
↓ ↑ 1

f(1)



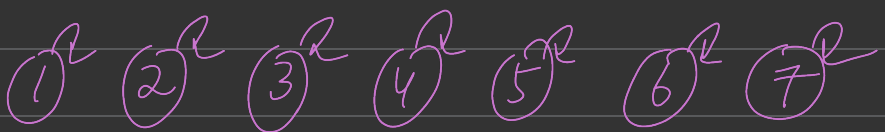
⭐ => Union By Size :

→ Config :- Size :-

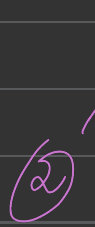
1	1	1	1	1	1	1
1	2	3	4	5	6	7

parent :-

1	2	3	4	5	6	7
1	2	3	4	5	6	7



(i) (1,2)



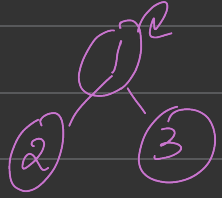
p[2] = 1

s[1] = 2

(ii) (2,3).

pU = 1 pV = 3

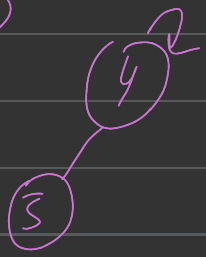
s = 2 s = 1



s[1] = 3

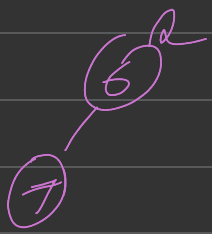
p[3] = 1

(iii) (4,5)



p[5] = 4 s[4] = 2

(iv) (6,7)

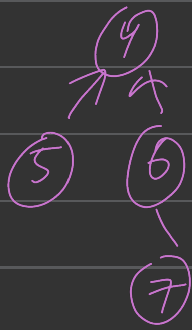


p[7] = 6 s[6] = 2

(v) (5,6)

pu = 4 s = 2

pv = 6 s = 2

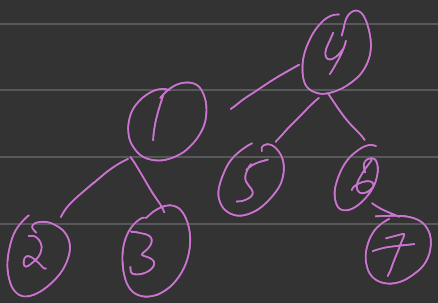


s[4] = 4

(+2)

(vi) (3,7) pu = 1 s = 2

pv = 4 s = 4



s[4] += 3

s[4] = 7

✱ Kruskal's Algorithm

→ Helps to find the Minimum Spanning Tree.

⇒ Process

(i) Sort all the edges by weight
(ii) perform union of above edges, if they dont belong to same component.
↳ Add weight to MST Wt.
(iii) Repeat until all edges are exhausted.

⇒ Time complexity :- $O(N+E) + O(E \log E) + O(2*(4\alpha))$.

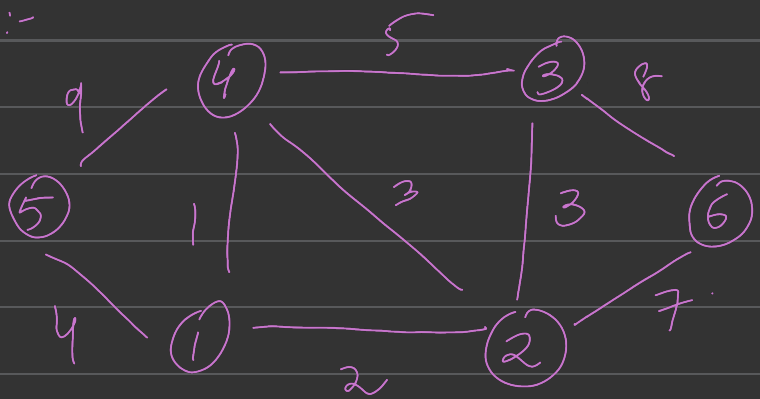
$O(N+E)$ ← Preparing edges
$O(E \log E)$ ← Sorting
$O(2*(4\alpha))$ ↓ Disjoint set
Union.

⇒ Space :- $O(V) + O(V)$

$O(V)$ ↓ Parent
$O(V)$ ↓ Rank / Size

→ Example :-

Graph :-



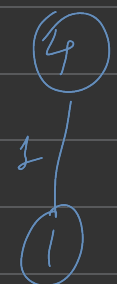
→ Edges sorted by weight :-

(wt, u, v) :

(1, 1, 4)
(2, 1, 2)
(3, 2, 3)
(3, 2, 4)
(4, 1, 5)
(5, 3, 4)
(7, 2, 6)
(8, 3, 6)
(9, 4, 5)

(i) (1, 1, 4).

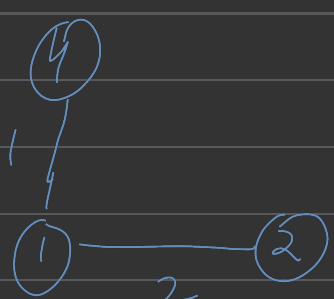
u = 1, v = 4, dont belong to same component



mstWt = 1

(ii) (2, 1, 2).

1, 2 doesnt belong to same component



mstWt = 1+2

(iii) (3,2,3)

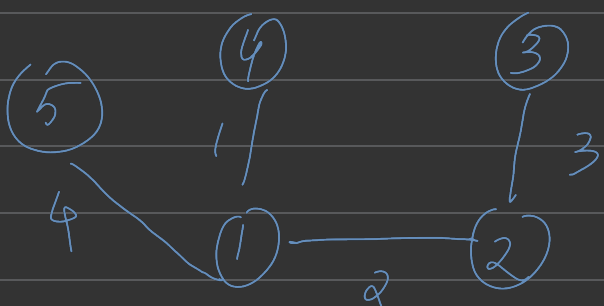
2,3 doesnt belong to same component

mst wt = 1+2+3

(iv) (3,2,4) 2,4 belong to same component

X

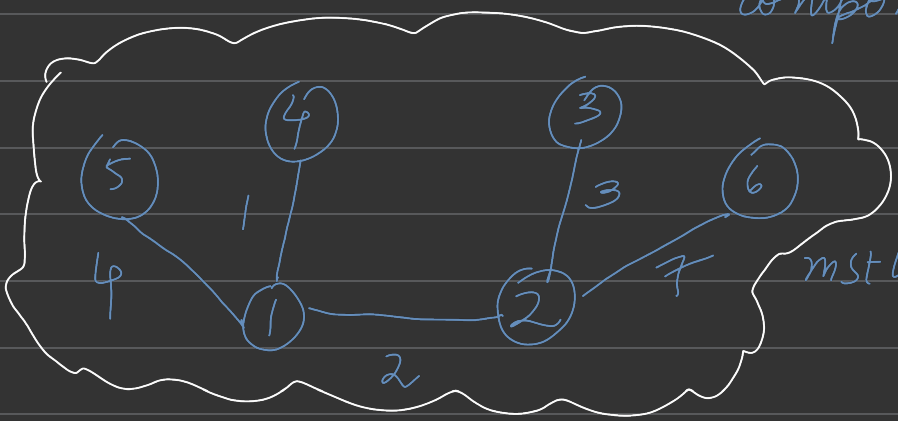
(v) (4,1,5) 1,5 dont belong to same component



mst wt = 1+2+3+4

(vi) (5, 3, 4) ×

(vii) (7, 2, 6) 2, 6 dont belong to same component



mst wt = 1+2+3+4+7

(viii) (8, 3, 6) ×

(ix) (9, 4, 5) ×

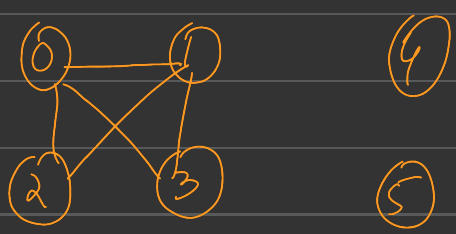
mst wt = 17

(Q) Number of operations required to make Graph connected

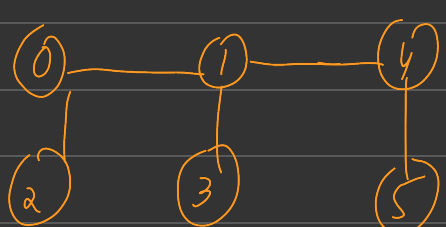
→ PS:- Given a Graph with 'n' components, return the minimum number of edges required to connect all components.

→ Note that we cannot create new edges, we need to reuse existing extra edges.

→ Input :-



→ Output :- 2

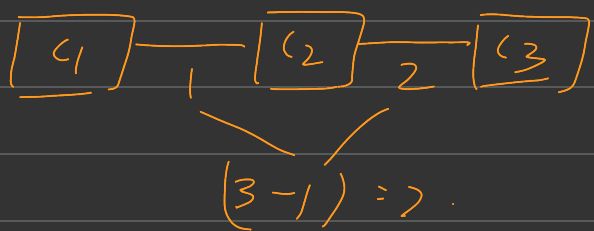


=> Approach:- → TC :- $O(E * 4\alpha) + O(N * 4\alpha)$.

To find ExtraEdges ↑ To find total components.

→ Min. no of edges required to connect n components = $n-1$

Ex :-



=> Steps

(i) find total extra Edges (cycle's)
(ii) find total components (i == p[i]).
(iii) required Edges = Total Components - 1
(iv) if extraEdges >= required Edges
↳ return required Edges
↳ else return -1;

→ why requiredEdges and not ExtraEdges?

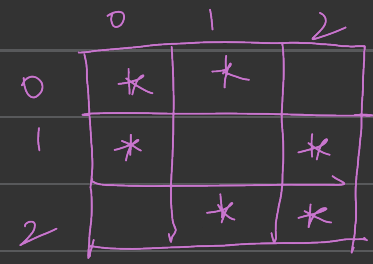
↓

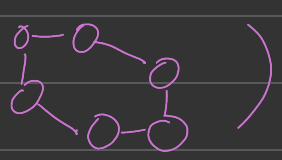
Represents minEdges to connect components which is possible iff we have extra edges.

(Q) Most Stones removed within same row/col.

→ PS :- Given 'n' stones and stone [xi, yi] representing the position of stone [row, col].
→ A stone can be removed if there is yet another stone in the same row or col as that of the existing stone
→ Return the max number of stones that can be removed

→ Input :- stones = [[0,0], [0,1], [1,0], [1,2], [2,1], [2,2]].



→ output :- 5 ()

⋆ ⇒ Approach :

(i) Treat each row / col where Stone is present as Node

(ii) Treat Group of Stones as a Component

↓

(Stones in Same row / col)

(iii) From each component of size 'c', we can at max remove $c-1$ nodes

↙

(only one node at max will be left having no adjacent row / col).

(iv) $\boxed{\text{Max Stone Removal} = N - \text{Num Of Components}}$

($i == p[i]$)

→ Row/col mapping to form node:-

(i) Ex:- [[0,0], [0,1]]

⇒ That there is a stone at row=0, col=0

row → row : 0

col → (col + maxRow + 1) : 2

∴ (0,0) ⇒ Union (0,2)

(Q) Accounts Merge (Hard) (⋆⋆⋆⋆)

→ PS:- Given a list of account names and emails for that account name
  → Merge the emails to similar account
  → We can merge emails of 2 account names iff there is at least 1 common mail in both account names

→ Input :- accounts :- [["John", "JohnSmith@m.co", "John_newyork@m.co"], ["John", "JohnSmith@m.co", "John00@m.co"] ["Mary", "mary@m.co"] ["John", "Johnnybravo@m.co"]].

→ Output :- [["John", "John00@m.co", "John_newyork@m.co", "JohnSmith@m.co"], ["Mary", "mary@m.co"], ["John", "Johnnybravo@m.co"]].

→ Hint :

(i) Treat each name Mails(i) as a Node

(ii) To merge do the Union

★ => Approach

(i) prepare a map having the mapping of mail → nodeIndex

↳ while preparing a map if we find the node is already present in map, => merge candidate, do Union();

(ii) Merge all required nodes from above node by checking their Uparent

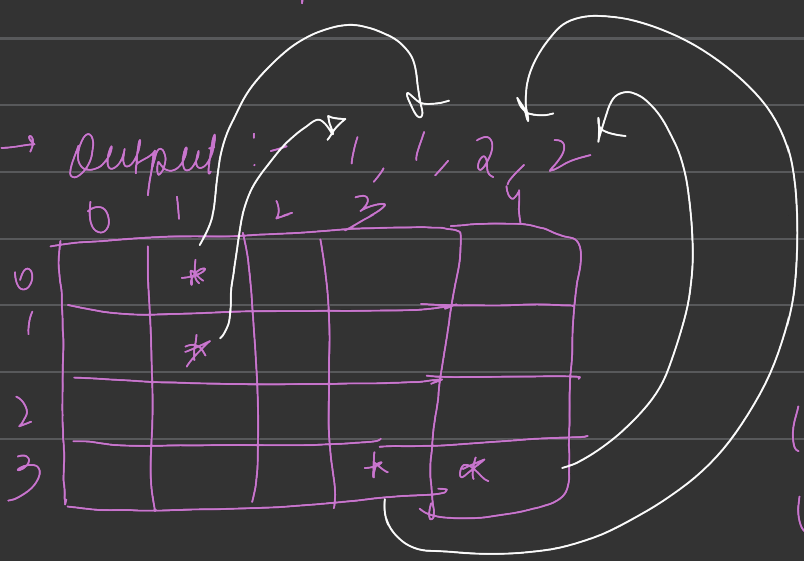
(iii) from above (ii) aggregate the ans in sorted fashion & attach to the name.

(Q) Number of Islands - 2 (online queries)

→ PS:- given the row / col size, K operations return the numbers of connected islands after each operation
- → Initially all the values are 0
- → After each operations [r, c], mark [r, c] = 1 => Island

→ Input :- n = 5, m = 4, k = 4
operations = [[1,1], [0,1], [3,3], [3,4]]

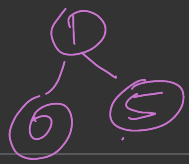
→ Output :- 1, 1, 2, 2



(i) (1,1) → 1 island
(ii) (0,1) → 1 component
(1-1) → (0,1)
(iii) (3,3) a comp's
(iv) (3,3) ↔ (3,4) a comps.

(1,1) → row = 1
col = 4+1 => 5

(0,1) → row = 0
col = 5



☆ Approach:

(i) we can have duplicate operations, to keep track of those we can use vis Array
↳ if already vis, numOfIslands remain same
(r, c)

(ii) (cell = row * maxCol + col)

(iii) After marking an island visited, increase the count of currentIsland's assuming, it doesn't have any neighbouring islands
(↔)
↳ Check if there are any neighbouring islands
↳ if (vis[nR][nC] == 1)

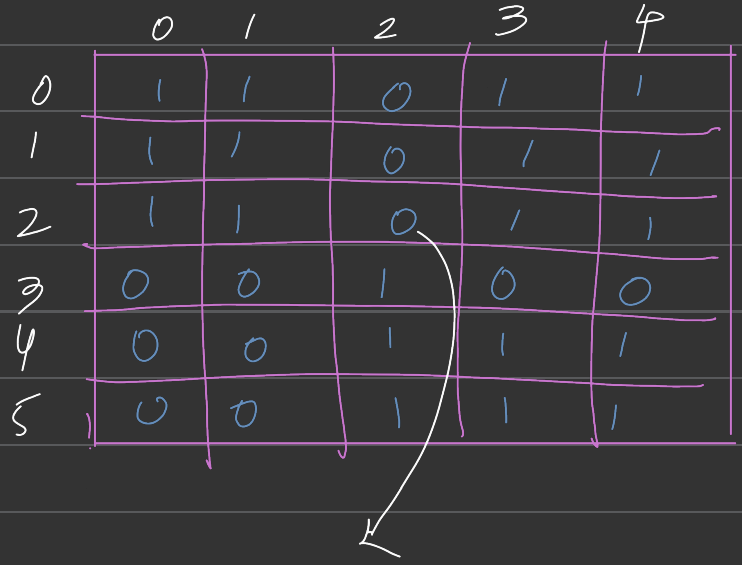
↳ if we find any neighbouring island, check if they are not connected (ultimate parents)
↳ if they are not connected
↳ connect them, count--;

(iv) Repeat for all operations

(Q) Making a Large island

→ PS :- Given a grid containing 0, 1
  → Return the largest island we can form by converting at most one 0 to 1
  → Size of island includes adjacent islands (↔↕)

→ Input :-



→ Output :- 20 (if we convert (2,2) to 1)

☆ Approach

(i) Connect individual components, find the size of individual components after they are connected

(ii) simulate flipping of 0 → 1 for all 0's

↳ find the total component's size's of all adjacent components to that '0'

↳ The MaxSize is candidate to flip

↳ Note that when cheking the adjacent component size, there can be a case wherein

$L \in$ Same Comp $\leftarrow$ $\in$ Same Comp

∴ Do the checking based on a Parent

(iii) Edge Case :- of grid has all 1's & no 0's return the largest connected component size.