

OVER GRAPH SERIES

G-1: INTRO TO GRAPH | Types | Conventions Used

Two types of data structure?

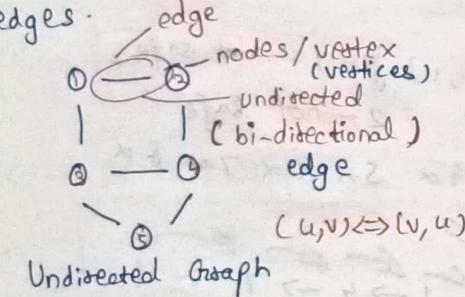
1. Linear (Arrays, stacks, queues, LL)
2. Non-Linear
(Tree, BST, Graphs)

In fact, a tree is a special type of graph with some restrictions.

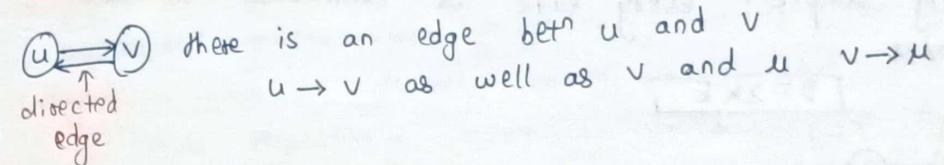
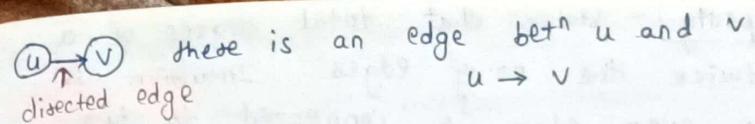
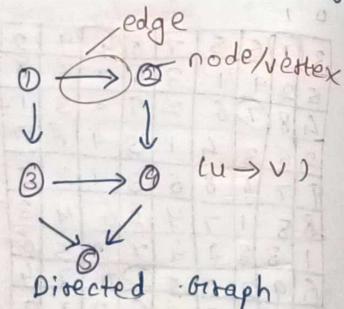
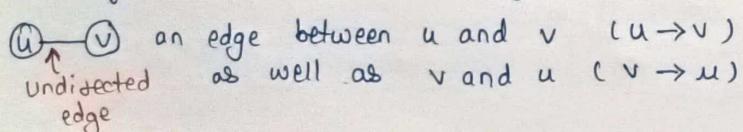
GRAPHS

Data structures - that have a wide-ranging appn in real life. These include analysis of electrical ckt's, finding the shortest routes betn 2 places, building navigation (places) system like G-map, even social media use graphs to store data abt. each user, etc.

A graph is a non-linear DS consists of nodes that have data and are connected to other nodes through edges.



numbering of nodes can be done in any order.
pair of vertices are called edges.

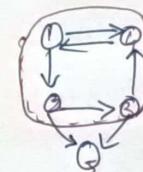


Cycles

- 3 cycles Undirected cyclic graph



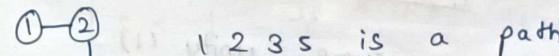
start at one node & end at that node



— directed cyclic graph (DAG)

No cycle 1 cycle
(DAG) (directed acyclic graph)

* PATH - lot of nodes and each pair is reachable

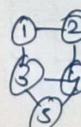


1 2 3 4 5 is a path

1 2 3 2 1 (NO, bcz a node can't appear twice in path)
1 3 5 (not a path, there's no edge betn 1 and 3)

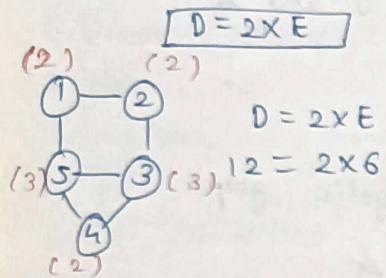
* Degree - no. of edges going inside or outside that node

For undirected graphs, the Degree (D) is no. of edges attached to node



$$D(3) = 3 \quad D(5) = 2 \\ D(4) = 3$$

A very imp. **Property** - states that total degree of a graph is twice the no. of edges. Intuition is very simple - every edge is connected to two nodes.



$$D = 2 \times E$$

$$D = 2 \times E$$

$$12 = 2 \times 6$$

For directed graphs, we have **indegree** and **outdegree**.
(Incoming) (Outgoing)

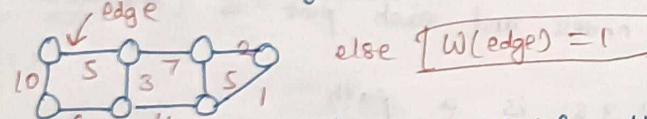
$$\begin{array}{l} 1 \rightarrow 2 \\ \downarrow \\ 4 \quad 3 \end{array}$$

$$\text{InD}(2) = 2$$

$$\text{OD}(2) = 0$$

Edge Weight

graph may have weights assigned to its edges often told as cost of the edge.



$$\text{else } W(\text{edge}) = 1$$

If weights not assigned, we assume unity (1).
In appn, weight may be a measure of the cost of a route.

for eg - If A and B, represent towns in road network, weight of edge AB may be cost of moving from A to B, and viceversa.

Gr-2: Graph Representation in C++ | Two Ways

In question, they will mention whether it is directed or undirected graph. The first line contains two integers n and m denoting no. of nodes and the no. of edges respectively. Next m lines contain two integers u and v representing edge. In UND graph if $u \rightarrow v$ then $v \rightarrow u$, now question arises is there any boundation on the no. of edges i.e. the value of m? Ans is NO. If we add more edges, value of m \uparrow .

1. Adjacency matrix

2. Adjacency lists

$n=5 \quad m=6$

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

↳ two-dimensional array of size $n \times n$, for 1-based indexing, last node is 5 and $n=5$, so define 2D array of size $[n+1][n+1]$.

If there's edge betn 1 and 2 mark at (1, 2) as well as (2, 1) for undirected graph.

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

all remaining are left as it is

Space - $O(N \times N)$ costly as n^2 locations are consumed

TC - $O(m)$

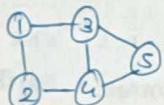
to optimise space, we use

We normally use an array here

vector<int> adj[n+1] // Array of vectors/ vectors of vectors

Now every index contains an empty vector/list.

Consider an undirected graph



$$0 - \{\}$$

$$1 - \{2, 3\}$$

1 2 node 2 appears in 2 - {1, 4}

1 3 node 1, 3 - {1, 4, 5}

2 4 and 1 appears in the list 4 - {2, 3, 5}

3 4 of node 2 5 - {4, 3}

4 5

3 5 1 edge is connected to 2 nodes

Space complexity = $O(2xE)$

which is much much better than n^2 locations, and most of them are unused.

For directed graphs, $u \rightarrow v$ so, $SC = O(E)$

bcz vice-versa = false

Weighted Graph Representation

If weights are mentioned,

for adjacency matrix

$adj[u][v] = w$ } both for undirected
 $adj[v][u] = w$ } graph

else for directed, $adj[u][v] = w$

But, how to implement in adjacency list?

Earlier, we were storing integers in each index, but for weighted graphs, we will store pairs (node, edge weight)

vector<pair<int, int>> adj[n+1]

edge 4 → { (2, 1), (3, 4), (5, 3) }
(4, 2) wt. edge (4, 2)

Gr-3: Java

Gr-4: Connected Components in Graphs

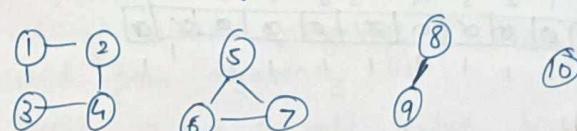
Graphs can be connected or can be like a binary tree.

But,



4 diff. graphs as they are not connected.

Given an undirected graph with 10 nodes and 8 edges. Edges are (1,2), (1,3), (2,4), (4,3), (5,6), (5,7), (6,7), (8,9). The graph will be



Apparently, it's a graph, which is in 4 pieces. So, the graph is broken down into 4 diff. connected components.

If we see 2 diff. components, then I can't say it can't be a single graph.

Graph Traversal

Any traversal algo. will use a visited array.

0 1 2 3 4 5

0	0	0	0	0	0
---	---	---	---	---	---

for 1-based : size of vis array $(n+1)$

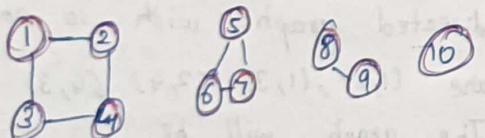
0-based : size = (n)

If a node is ! visited, call the traversal algo.

```
for i:=1 to 10
    if(!vis[i])
        traversal(i)
```

We need to do it in loop

Why not one call for traversal cuz, traversal(1) will only traverse connected components i.e. nodes 1, 2, 3 and 4, but not the connected components.



visited

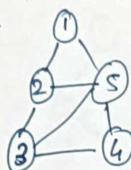
0	1	2	3	4	5	6	7	8	9	10
0	1	0	1	0	1	0	1	0	1	0

TC $\rightarrow O(N)$

SC $\rightarrow O(N)$

GS: Breadth-First Search | Level Order Traversal

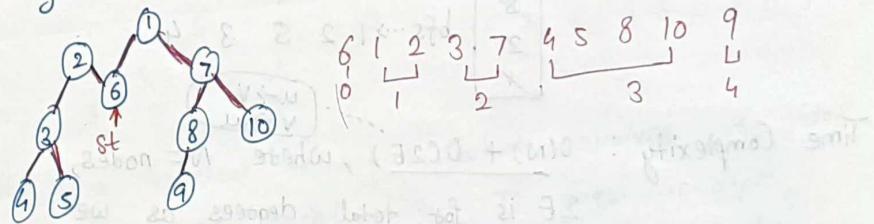
Eg -



ans 1 2 3 4

If starting node is not 1 or initial one, then remember you need to go breadth-wise

always



Initial configuration -

- Queue Data Structure : follows FIFO, and will always contain the starting node.
- visited array : an arr initialized to 0
- Steps - 1. we start with 'st' node, mark it as visited, and push it into queue DS.

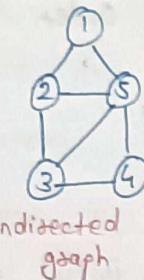
2. In every iteration, we pop out node 'v' and put in the bfs soln vector, as we are traversing the node.

3. All the unvisited adjacent nodes from 'v' are visited next and are pushed into the queue.

The list of adjacent neighbors of the node can be accessed from adjacency list.

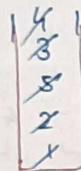
4. Repeat 2 and 3 until queue becomes empty.

Day Run



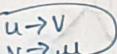
Undirected
graph

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



Adjacency list

bfs \rightarrow 1 2 5 3 4



Time Complexity : $O(N) + O(2E)$, where $N = \text{nodes}$,

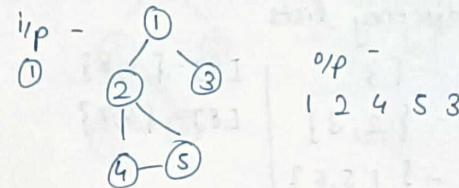
$2E$ is for total degrees as we traverse all adjacent nodes.

Space complexity : $O(3N) \approx O(N)$, space for Queue, visited array and an adjacency list.

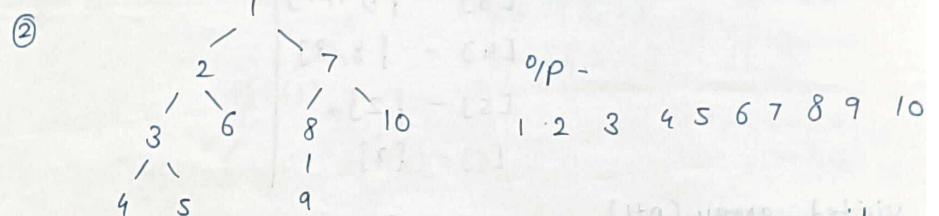
For directed graph, TC $\rightarrow O(N) + O(E)$

$$\text{Degree} = \text{Edges} \\ (u \rightarrow v)$$

Q6: Depth First Search (DFS)



O/P -
1 2 4 5 3



O/P -

1 2 3 4 5 6 7 8 9 10

DFS is a traversal technique which involves the idea of recursion and backtracking. It goes in-depth, i.e. traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

dfs()

{ vis[node] = 1

ls.add(node)

for(auto it: adj[node])

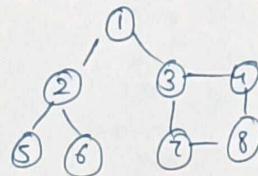
{ if(!vis[it])

dfs(it)

}

}

way, all nodes
manner.

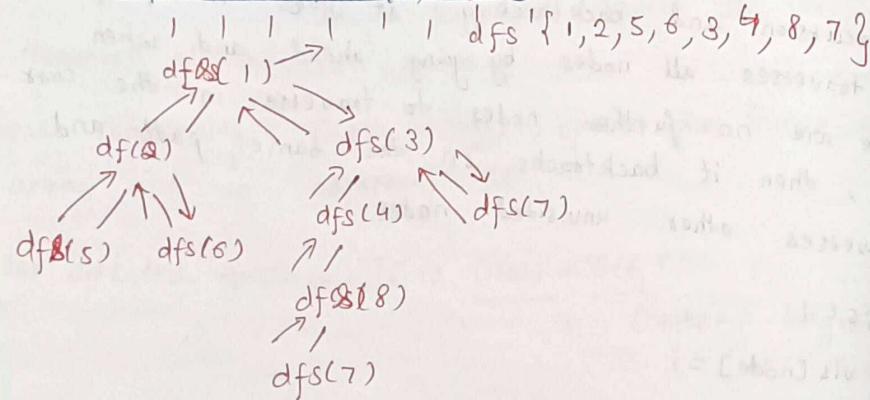


adjacency lists

[0] - {3}	[7] - {3, 8}
[1] - {2, 3}	[8] - {4, 7}
[2] - {1, 5, 6}	
[3] - {1, 4, 7}	
[4] - {8, 3}	
[5] - {2}	
[6] - {2}	

visited array ($n+1$)

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



Time Complexity $\rightarrow O(N) + O(2E)$, for a directed graph
 $O(N) + O(E)$ bcz for every node we are calling the
 recursive function once, the time taken is $O(N)$
 and $2E$ is for total degrees as we traverse for all
 adjacent nodes.

$$SC \rightarrow O(N) + O(N) + O(N) \approx O(N)$$

\uparrow visited array \uparrow dfs stack space \uparrow adjacency list (dfs)

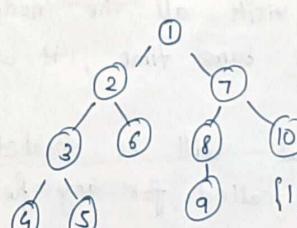
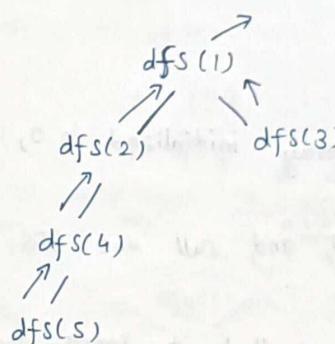
skewed graph
 then stack space $\approx O(n)$



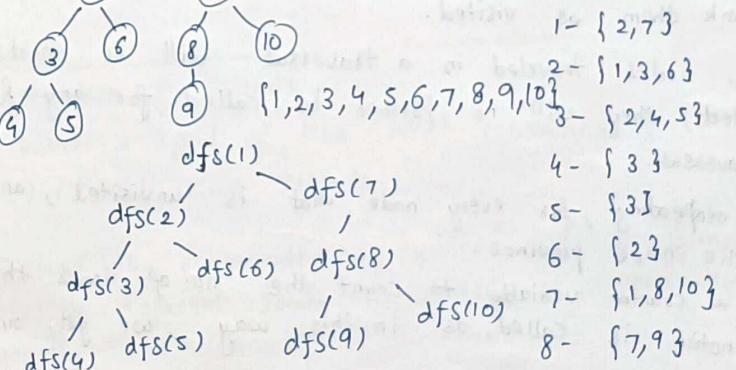
Visited	1	2	3	4	5
	0	1	0	0	0

adj

dfs	1 2 4 5 3
-----	-----------



vis	1	2	3	4	5	6	7	8	9	10
	0	1	1	0	1	1	1	1	1	1

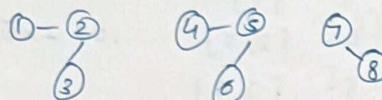


1 -	{2, 7, 3}
2 -	{1, 3, 6, 3}
3 -	{2, 4, 5, 3}
4 -	{3, 3}
5 -	{3, 3}
6 -	{2, 3}
7 -	{1, 8, 10, 3}
8 -	{7, 9, 3}
9 -	{8, 3}
10 -	{7, 3}

G7: Number of Provinces

PS => Given an Undirected Graph. we say two vertices u and v belong to a single province if there is a path from $u \rightarrow v$ or $v \rightarrow u$. Your task is to find the number of provinces.

Ex -



ans = 3

Algorithm :-

Step - 1: We need a visited array initialized to 0, represent nodes that ain't visited yet.

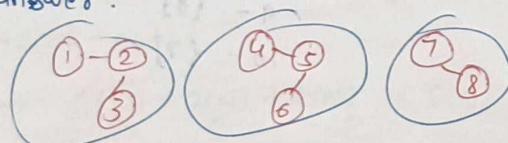
2: Run a for loop from 0 to N , and call the DFS for the first unvisited node.

3: DFS funcⁿ call will make sure that it starts the DFS call from that unvisited node, and visits all the nodes that are in that province, and at same time, it will also mark them as visited.

4: Since, nodes traveled in a traversal will be marked as visited, they will no further be called for any further DFS traversal.

5: Keep repeating, for every node that is unvisited, and visit the entire province.

6: Add a counter variable to count the no. of times the DFS function is called, as in this way, we get our answer.

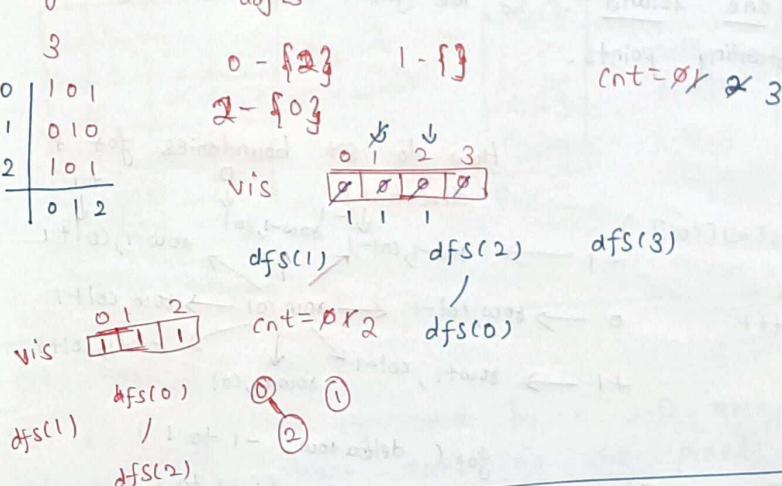


ans = 3

TC $\rightarrow O(N) + O(V+2E)$, where $O(N)$ is for outer loop and inner loop runs in total a single DFS over entire graph, and we know DFS takes a time of $O(V+2E)$.

SC $\rightarrow O(N) + O(N)$, space for recursion stack & space and visited array.

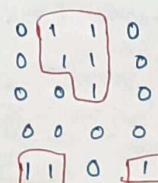
Day Run



G8: Number of Islands

PS - Given a grid of size $N \times M$ consisting of 0's (water) and 1's (land). Find Islands.

An Island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e. in all 8 directions.



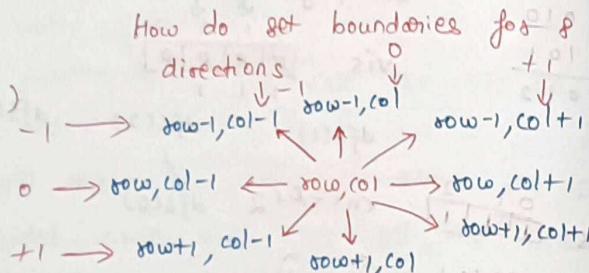
ans = 3

Intuition:



If we start a trav. algorithm, from a land(1) it will make sure it traverses all 8 direcⁿ in nearest form. So, one Traversal covers one island. If we do 3 traversals then we must have 3 starting nodes, anyone can be considered, and it'll make sure it visits everyone. The basic idea is "one starting node represents one island". So, we just need to figure out no. of starting points.

```
for (row)
    for (col)
        if (!vis[row][col])
            bfs
            cnt++
```



for (deltaRow: -1 to 1)

TC $\rightarrow O(N^2 + N \times M \times 9)$,
 N^2 for nested loops and $N \times M \times 9$ for the overall DFS of the matrix, that will happen throughout if all the cells are filled with 1.

SC $\rightarrow O(N^2)$ for visited array, max queue space, if all marked as 1 then max. queue space will be N^2 .

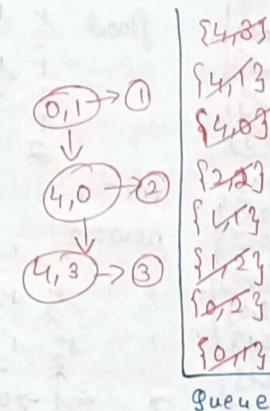
Day Run -
 $cnt = \emptyset \times 3$

0	1	2	3
0	1	1	0
0	1	1	0
0	0	1	0
0	0	0	0

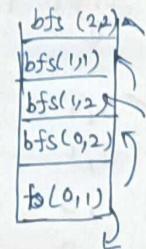
✓	✓
✓	✓
✓	
✓	✓

visited array

starting node: {0,1}



call stack



G9: Flood fill Algorithm

PS: An image is represented by a 2-D array of integers, each integer representing the pixel value of the image. Given a co-ordinate (st, sc) representing the starting pixel (s and c) of flood fill, and a pixel value $newColor$, "flood fill" the image.

To perform a 'flood fill', consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel are colored with the new color ' $newColor$ '. (replace) those same.

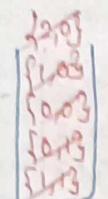
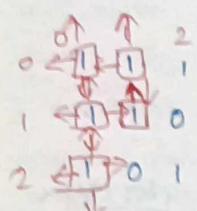
N - no. of rows

M - no. of columns

$1/p$	0	1	1	1
1	1	1	0	
2	1	0	1	

$$Sx = 1$$

$$\begin{array}{r} \text{P/P} \\ \text{---} \\ \begin{array}{r} 2 & 2 & 2 \\ 2 & 2 & 0 \\ \hline 2 & 0 & 1 \end{array} \end{array}$$



flood	$\begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$
-------	--	--	--

Queue
5204
5023
5003
5103
5013
4613

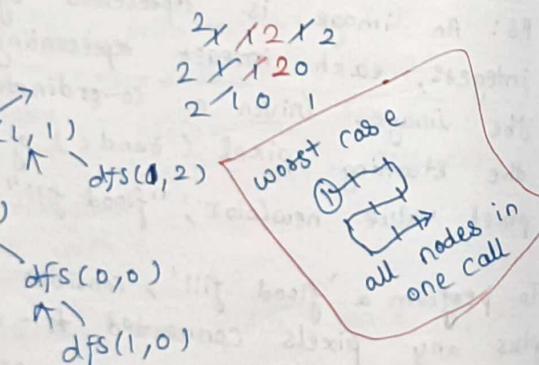
$$\begin{array}{cccc} \text{ini} & = & 1 & \\ \text{new} & = & 2 & 1 \quad 2 \\ \hline 0 & & x_2 & x_2 & x_2 \\ 1 & & x_2 & x_1 & 0 \\ 2 & & \leftarrow x_2 \rightarrow 0 & & 1 \end{array}$$

✓	0, 1	-1, 1
2, 1	1, 1	
1, 0	0, 0	
1, 2	0, 2	
0, 0	✓	
2, 0	✓	
1, -1		✓
1, 2		✗

if(valid and image is initial
and ans(node) is not newColor)
call bfs || dfs

$$\begin{array}{ccc} 1 & 1 & 1 \\ 1 & \textcircled{1} & 0 \\ 1 & 0 & 1 \end{array}$$

ini = 1
npw = 2



$$TC \rightarrow O(NXM) + O(NXM \times b)$$

TC \rightarrow $O(NXM) + O(NV)$
 For the worst case, all of the pixels will have some colour, so DFS func will be called for every node takes $O(NXM \times 4)$ time
 SC \rightarrow $O(NXM) + O(NXM)$ copied ans array rec. stack space

G-10: Rotten Orange

PS: Given a grid of $N \times M$ where each cell in the grid have values 0, 1 and 2 which means

o: empty cel

1: fresh orange

2: rotten orange

2: ~~bottom orange~~
 determine min. time to get all oranges. A bottom
 orange at index $[i, j]$ can get other in 4-dim.
 \uparrow in unit time.



$$t = \alpha x \beta y u$$

Rottenness happens simultaneously.

1 P 0 2 2 + x

6

2-1

Now, question arises which algo. to use?
 A rotten one can set fresh orange neighbours that are at a distance 1 or at the same level. It means each of them got rotten at a similar level, implying we need to visit the same level at the same time.
 Hence, level-order traversal is BFS.

Why not DFS? bcz. visiting depth-wise. But here it's not the case to sort all oranges, we need to find minimum time to sort them all, which is possible only when we are in neighbouring directions at an equal pace.

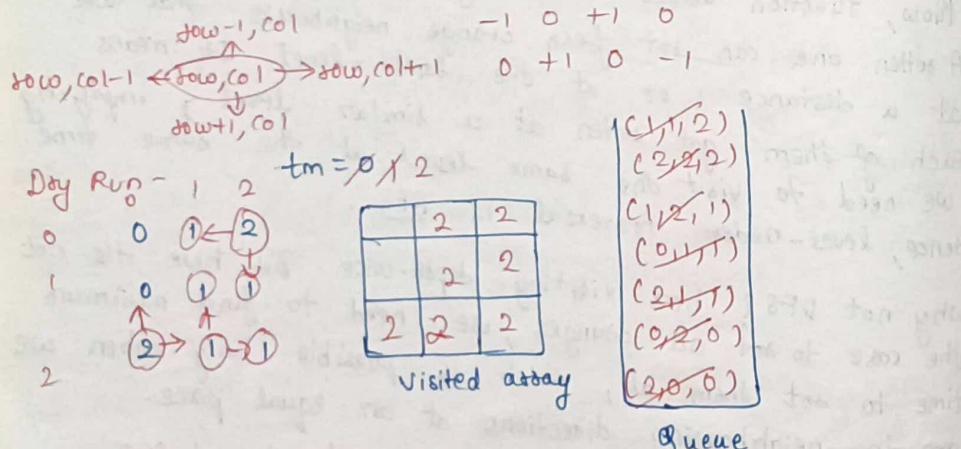
all rotten in 1 sec

Approach:

Initial configuration : Queue, Visited Array

Algo -

1. Store pair of cell no. and time $\langle \langle i, j \rangle, t \rangle$ in queue and marked them as rotten.
2. While BFS, pop out an element and travel to all its neighbours. In a graph, we store the list of neighbours in an adjacency list but here we know Nb are in 4-dimensions.
3. We use nested loops to visit neighbours.
4. BFS call make sure that it starts the BFS call from rotten, and set all valid fresh ones and put them in queue with time increase of 1 unit.
5. Pop-out another rotten and repeat until g.bec. w.
6. Add a cnt variable to store max.time and return if any of the fresh orange isn't remaining else return -1.

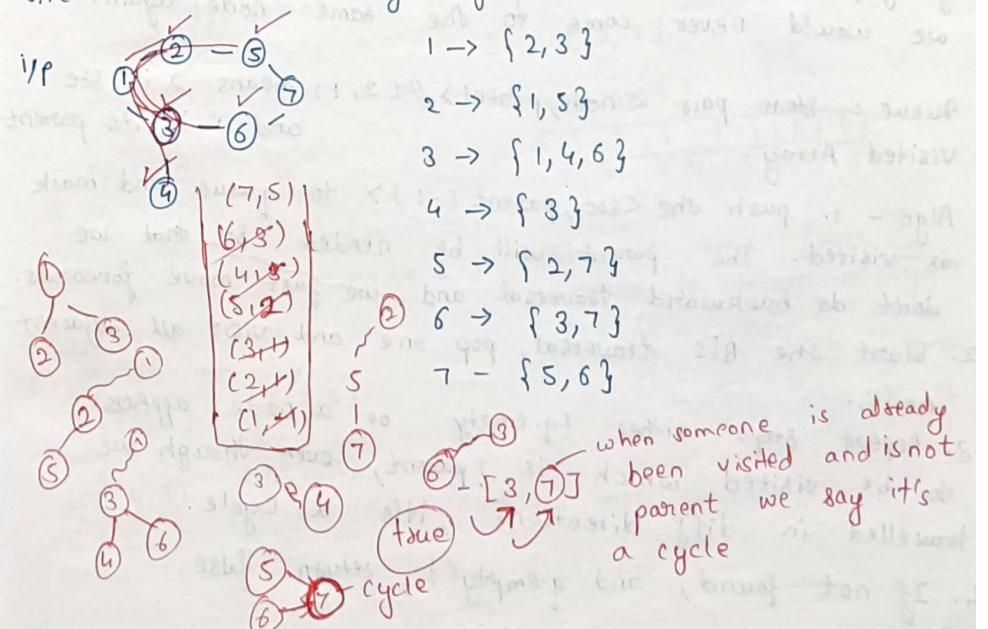


$TC \rightarrow O(NXM + NXMX4) \sim O(NXM)$, for the worst case, all of the cells will have fresh oranges, so the BFS funcn will be called for NXM nodes and for every node, we are traversing u neighbours \leftrightarrow , it'll take $O(NXM \times u)$ time

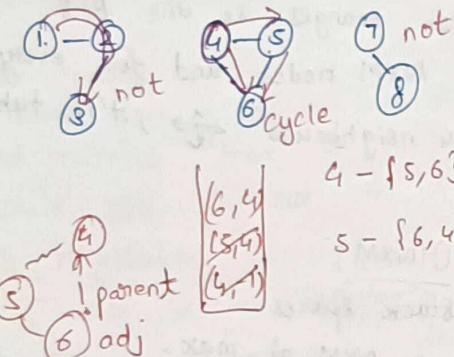
$SC \rightarrow O(NXM) + O(NXM) \sim O(NXM)$
 recursive stack spaces
 for copied i/p array takes up NXM at max.

G-11: Detect a cycle in an undirected graph \rightarrow BFS (adj. List)

PS - Given an UD Graph with V vertices and E edges, check whether it contains any cycle or not. Graph is in the form of adjacency list where $adj[i][j]$ contains all the nodes j th is having edge with.



Input can be connected



Intuition - we start from a node, and start doing BFS level-wise, if somewhere down the line, we visit a single node 2x, it means we came two paths to end up at the same node. It implies there is a cycle in the graph bcz we know that we start from diff. directions but can arrive at the same node only if graph is connected or contains a cycle, otherwise we would never come to the same node again.

Queue: store pair <node, parent> // (2, 1) means 2 is site and 1 is its parent
Visited Array

1. push the <site, parent(-1)> to queue and mark as visited. The parent will be needed so that we don't do backward traversal and we just move forwards.
2. Start the BFS traversal, pop one and visit all adjacent nodes.
3. Repeat steps either !q.empty() or a node appears to be visited which is !parent, even though we travelled in diff. directions, it's a cycle.
4. If not found, and q.empty(), return false.

```

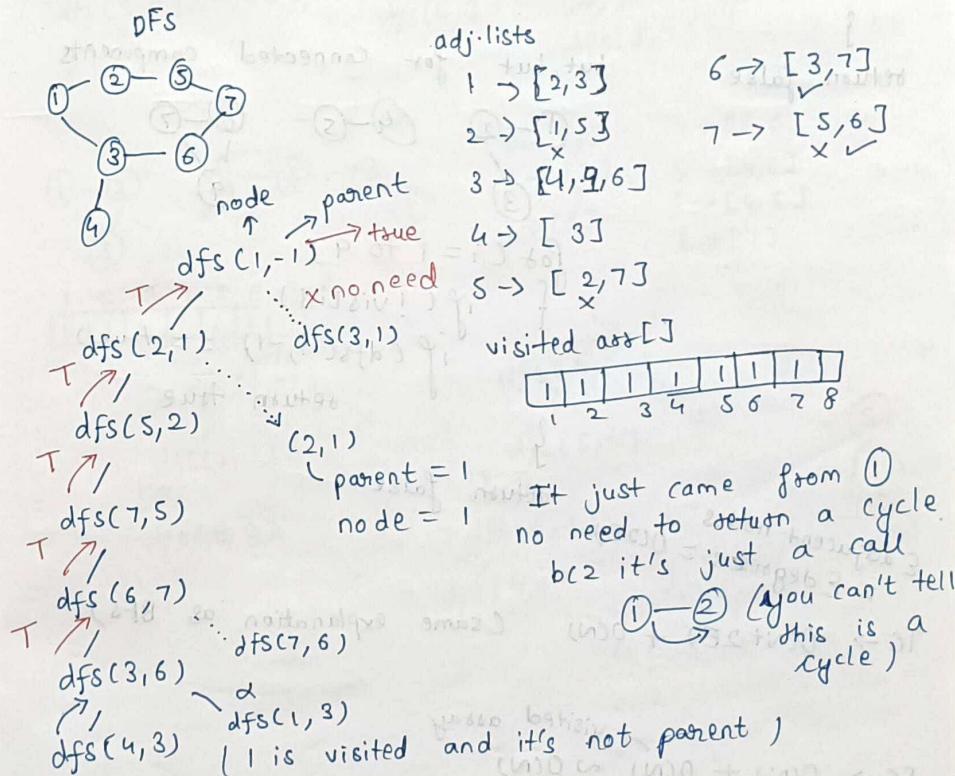
for(i=1 to n) {
    if(!vis[i])
        detect(vis[i])
    return true
}
return false
    
```

Pseudo-code

$T.C \rightarrow O(N+2E)$ + $O(N)$, where $N = \text{node}$
 $2E$ is total degrees as we traverse all adjacent nodes. In the case of connected components of graph, it takes another $O(N)$ time.

$S.C \rightarrow O(N) + O(N) \sim O(N)$, space for queue data structure and visited array.

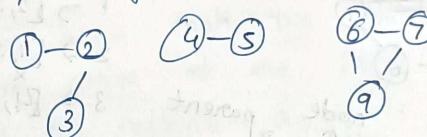
G-12: Detect a cycle in an undirected graph using DFS



Pseudocode :-

```
dfs(node, parent)
    vis[node] = 1
    for auto it : adj[node]
    {
        if (!vis[it])
            if (dfs(it, node))
                return true
        else if (it != parent)
            return true
    }
    return false
```

but, but for connected components



```
for (i = 1 to 9)
{
    if (!vis[i])
        if (dfs(i, -1) == true)
            return true
}
```

return false

$$\sum \text{adjacent nodes} = \sum \text{degrees} = O(2E)$$

$$TC \rightarrow O(N+2E) + O(N) \quad (\text{same explanation as BFS})$$

visited array

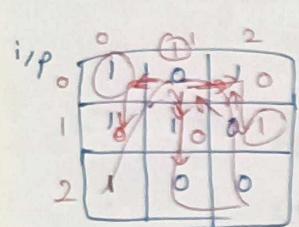
$$SC \rightarrow O(N) + O(N) \sim O(N)$$

for skewed graph, we may end up having a recursive stack space of $O(n)$

Q13: Distance of nearest cell having 1

PS - Given a 0/1 grid $N \times M$. Find the distance of the nearest 1 in the grid for each cell.

The distance is calculated as $|i_1 - i_2| + |j_1 - j_2|$, where i_1, j_1 are row no. and col no. of current cell, and i_2, j_2 are row no. and col no. of nearest cell having value 1.



i/p
0 1 1
0 0 0
1 0 1
2 1 0 0

$$i_1, j_1 \\ 0, 0 \\ 1, 0 \\ 2, j_2$$

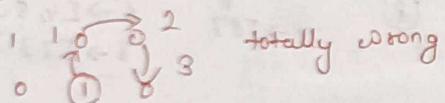
$$i_2, j_2 \\ 0, 1 \\ 0, 0 \\ 0, 1$$

answer

0,0 nearest is himself

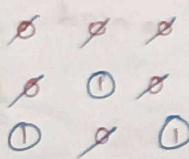
What algorithm?

DFS



totally wrong

BFS (level-wise) steps



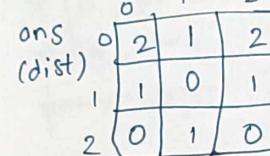
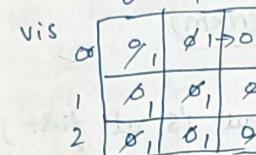
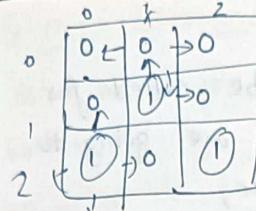
1 1 1

✓ 1 step

nearest 0's (0's are at dist. 1)

✓ 1 step

next set of 0's (dist. 2)



(0,0), 2
(0,1), 2
(1,0), 1
(2,1), 1
(1,2), 1
(0,2), 1
(2,2), 0
(2,0), 0
(1,1), 0

(1,1), 0
(0,1), 1
(1,2), 1

! queue.empty()

Intuition: BFS will take a step from cells containing 1 and will reach out to all zeros at a distance of one. Apparently, we can say that the nearest 1 to the 0s is at a distance of one. Again if we take another step, we will reach the next set of zeros, for these 1 is at a distance of 2. similarly, we can reach all the 0's possible.

Algo - (Init-Config: Queue <coordinates, steps> Visited Array)

- Push the pair of 1's in Q and mark them as visited.
- Start BFS, pop out from queue, and travel to all its neighbors having 0.
- For every neighbor unvisited 0, we can mark the dist to be +1 of curr node dist and store it in the dist. 2D arr, at same time insert {<row, col>, steps+1}
- Repeat until !q.empty

$$TC \rightarrow O(NXM + NXM \times 4) \sim O(NXM)$$

for worst case, the BFS func will be called for (NXM) nodes, and for every node, we are going to its 4 neighbors, so it takes $(NXM \times 4)$

$$SC \rightarrow O(NXM) + O(NXM) + O(NXM) \sim O(NXM)$$

↑ ↑ ↑
Visited array distance steps Queue DS (if all 1's at first)

day sun :-

0	1	2
1	0	1
2	1	0
0	0	0

0	1	0
0	0	1
0	1	2

steps

(2,2), 2
(2,1), 1
(1,2), 1
(0,2), 1
(2,0), 0
(1,1), 0
(1,0), 0
(0,1), 0
(0,0), 0

Queue

1	1	1
1	1	1
1	1	1

visited

0	1	2
1	0	1
2	1	0

0	1	2
1	0	1
2	1	0

steps

(0,0), 2
(0,1), 2
(1,0), 1
(1,1), 1
(0,2), 1
(2,0), 0
(2,1), 0
(1,2), 0

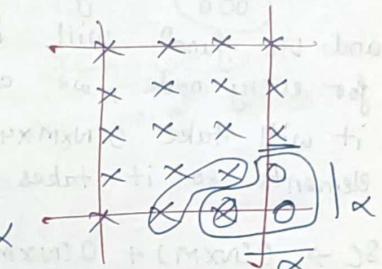
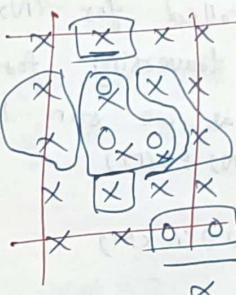
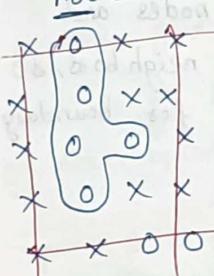
Queue

0	1	0
0	0	0
1	0	0

visited

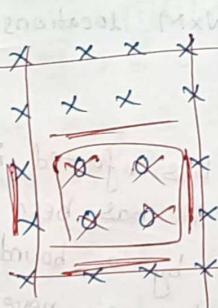
G-14: Surrounded Regions | Replace 0's with X's

PS - $N \times M$ grid, Replace all '0' with 'X', that is surrounded by 'X'. An '0'(or a set of '0') is considered to be surrounded by 'X' if there are 'X' at locations just below, just \leftarrow , just \rightarrow .
 A L R
 not connected



Intuition:

$$\begin{aligned} & \text{row} = 0, \text{col} = 0 - m-1 \quad \checkmark \\ & \text{row} = 0 - n-1, \text{col} = 0 \quad \text{row} = n-1, \text{col} = 0 - m-1 \quad \checkmark \\ & \text{row} = 0 - n-1, \text{col} = m-1 \quad \text{row} = 0, \text{col} = m-1 \quad \text{stable} \end{aligned}$$



Algo - Start from boundary
mark all connected 0's

Intuition :-

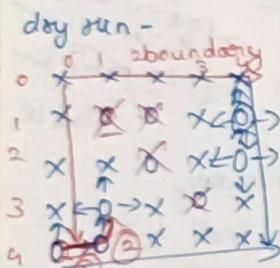
The boundary elements in matrix can't be replaced with 'X' as, they are not surrounded by 'X' from all 4 directions. This means if '0' (or a set of '0') is connected to a boundary '0' then it can't be replaced with 'X'.

The intuition is that we start from boundary elements having '0' and go through its neighboring 0's in 4 directions and mark them as visited to avoid replacing them with 'X'.

$-1, 0$
 \uparrow
 $\text{row}, -1 \leftarrow \text{row}, \text{col} \rightarrow \text{row}, \text{col} + 1$
 \downarrow
 $\text{row} + 1, \text{col}$

$T_C \rightarrow O(N \times M \times 4) + O(N) + O(M) \sim O(N \times M)$ for worst case
 every elem. will be marked as '0' in matrix
 and DFS funcⁿ will be called for $(N \times M)$ nodes and
 for every node, we are traversing for 4 neighbours, so
 it will take $O(N \times M \times 4)$, as we run loops for boundary
 elements so it takes $O(N) + O(M)$

$S_C \rightarrow O(N \times M) + O(N \times M) \sim O(N \times M)$
 \uparrow
 visited array
 \uparrow
 auxiliary stack space takes $N \times M$ locations at max



done dfs(4, 0)

dfs(4, 1) \times

dfs(3, 1)

29
14

	0	1	2	3	4
0	o	x	x	x	x
1	x	x	x	x	o
2	x	x	x	x	o
3	x	x	x	x	x
4	x	x	x	x	x

dfs(1, 4)

dfs(2, 4)

$\begin{matrix} 8 & 1 \\ 4 & x \\ 4 & 0 \end{matrix}$
 total = 11

someone has infected it
 meaning it has been
 touched by a boundary
 cell which can never
 ever be replaced
 with X's.

G1-15: Number of Enclaves

PS - Given an $n \times m$ (0/1) grid, where 0 - sea and 1 - land.

A move consists of walking from one land to another adjacent (4-direction) land cell or walking off boundary of the grid.

Find no. of land cells in grid for which we can't walk off the boundary of the grid in any no. of moves.

i/p - $\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$

bound

jump

off

cnt = 3

$\begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix} \rightarrow X$
 $\begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix} \rightarrow X$
 $\begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix} \rightarrow X$
 $\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix} \rightarrow X$

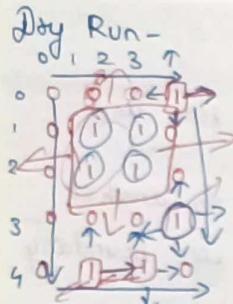
cnt = 4

one thing I can surely say is, the cell at boundary having 1 (land) will never be my answer apparently, all the 1's connected to it can jump off the bdt, and which not account to our answer.

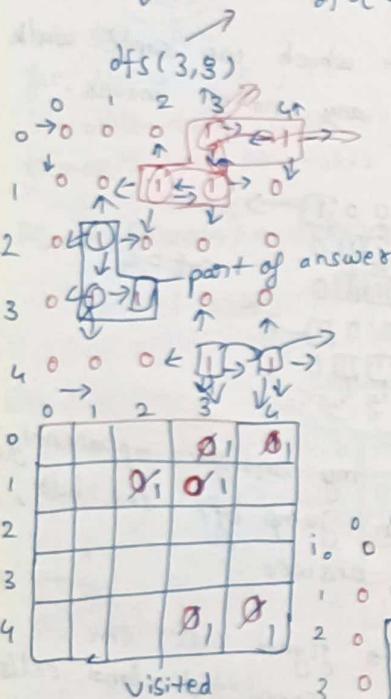
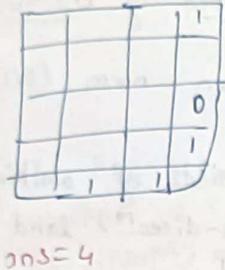
The intuition is that we need to figure out the boundary land cells, go through their connected land cells and mark them as visited. The sum of all tem. lands will be the answer.

$T_C \rightarrow O(N \times M \times 4)$ - for worst case, assuming all pieces as land, the (BFS) funcⁿ will be called for $(N \times M)$ nodes, and for DFS every node its 4 neighbours.

$S_C \rightarrow O(N \times M)$, $O(N \times M)$ for visited array, and queue/DFS stack space takes up $N \times M$ locations at max.



dfs(0, 3)
dfs
dfs(4, 1)
dfs(4, 2)



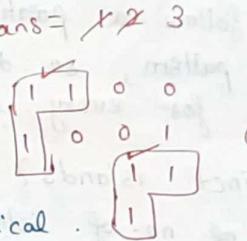
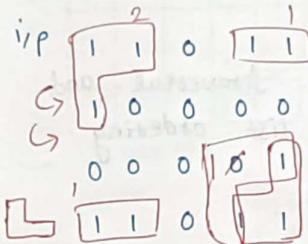
dfs(0, 3)
dfs(0, 4)
dfs(1, 3)
dfs(1, 2)

Queue

(1,2)
(1,3)
(2,4)
(2,3)
(3,4)
(3,3)

Q1-16: Number of Distinct Islands | CT+DFS | Expansion of No. Islands)

PS → Given a boolean 2D grid, you have to find no. of distinct islands where a grp of connected 1s (horizontally or vertically) forms an island. Two islands are considered to be distinct if and only if one island is not equal to another (not rotated or reflected)



If rotated, not identical.

Depending on shape of island formed, we count the no. of islands.

The question arises how to store these shapes?

We can store the shapes in set DS, then it will return unique islands. We can store coordinate in a vector or a list.

(0,0), (0,1)

{ (2,3), (2,4) }

(1,0)

{ (2,3), (2,4), (3,3) }

But, we can call one of starting points a base, and subtract the base coordinates from the land's coordinates (cell coordinates - Base coordinates). Now the list will be similar as illustrated.

$(0,0)$	\rightarrow	$(0,1)$	Coordinate-base
\downarrow	\downarrow	$(0,0) - (0,0) = (0,0)$	
$(1,0)$		$(0,1) - (0,0) = (0,1)$	
		$(1,0) - (0,0) = (1,0)$	
\rightarrow			
$(2,3)$	\rightarrow	$(2,4)$	$(2,3) - (2,3) = (0,0)$
\downarrow		$(2,4) - (2,3) = (0,1)$	$\left\{ \text{similar} \right\}$
$(3,3)$		$(2,4) - (3,3) = (0,1)$	
		$(3,3) - (2,3) = (1,0)$	

NOTE: make sure to follow a particular traversal and a particular order pattern, so that list ordering remains the same for every cell.

How to state distinct islands?

This is expansion of no. of islands.

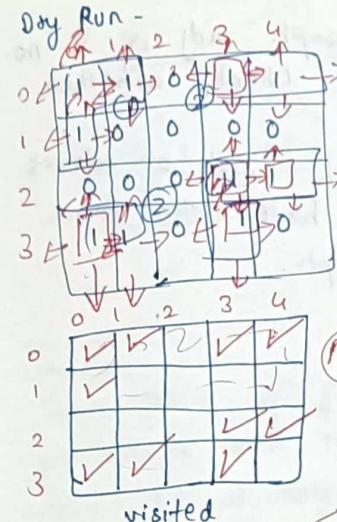
TC $\rightarrow O(nxm) + O(nxm \times 4)$ also if we use set other than unordered set then, $O(nxm \times \log(nxm)) + O(nxm \times 4)$

$\sim O(NXM)$

For worst case, assuming all pieces as land, the DFS funcn will be called for (NXM) nodes, and for every node, we are traversing for 4-neighbours, it takes $O(NXM \times 4)$ time. Set at max will store the complete grid, so it takes $\log(NXM)$ time.

SC $\rightarrow O(NXM) + O(NXM) \sim O(NXM)$

visited array takes up NXM locations at max



0	✓	✓	✓	✓
1	✓	-	-	-
2	✓	✓	✓	✓
3	✓	✓	✓	

visited

ans

0,0

0,1

0,2

0,3

0,4

1,0

1,1

1,2

1,3

1,4

2,0

2,1

2,2

2,3

2,4

3,0

3,1

3,2

3,3

3,4

4,0

4,1

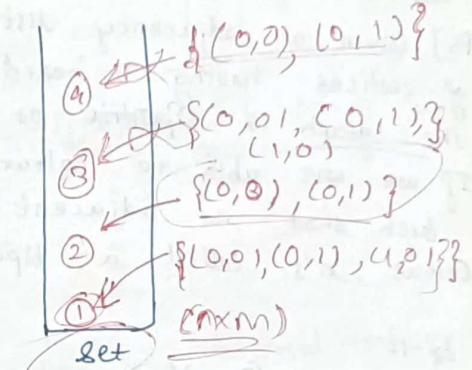
4,2

4,3

4,4

ans

2



dfs(0,0) - 0,0

dfs(0,1) - 0,0

dfs(0,2) - 0,0

dfs(0,3) - 0,0

dfs(0,4) - 0,0

dfs(1,0) - 0,0

dfs(1,1) - 0,0

dfs(1,2) - 0,0

dfs(1,3) - 0,0

dfs(1,4) - 0,0

dfs(2,0) - 0,0

dfs(2,1) - 0,0

dfs(2,2) - 0,0

dfs(2,3) - 0,0

dfs(2,4) - 0,0

dfs(3,0) - 0,0

dfs(3,1) - 0,0

dfs(3,2) - 0,0

dfs(3,3) - 0,0

dfs(3,4) - 0,0

dfs(4,0) - 0,0

dfs(4,1) - 0,0

dfs(4,2) - 0,0

dfs(4,3) - 0,0

dfs(4,4) - 0,0

dfs(5,0) - 0,0

dfs(5,1) - 0,0

dfs(5,2) - 0,0

dfs(5,3) - 0,0

dfs(5,4) - 0,0

dfs(6,0) - 0,0

dfs(6,1) - 0,0

dfs(6,2) - 0,0

dfs(6,3) - 0,0

dfs(6,4) - 0,0

dfs(7,0) - 0,0

dfs(7,1) - 0,0

dfs(7,2) - 0,0

dfs(7,3) - 0,0

dfs(7,4) - 0,0

dfs(8,0) - 0,0

dfs(8,1) - 0,0

dfs(8,2) - 0,0

dfs(8,3) - 0,0

dfs(8,4) - 0,0

dfs(9,0) - 0,0

dfs(9,1) - 0,0

dfs(9,2) - 0,0

dfs(9,3) - 0,0

dfs(9,4) - 0,0

dfs(10,0) - 0,0

dfs(10,1) - 0,0

dfs(10,2) - 0,0

dfs(10,3) - 0,0

dfs(10,4) - 0,0

dfs(11,0) - 0,0

dfs(11,1) - 0,0

dfs(11,2) - 0,0

dfs(11,3) - 0,0

dfs(11,4) - 0,0

dfs(12,0) - 0,0

dfs(12,1) - 0,0

dfs(12,2) - 0,0

dfs(12,3) - 0,0

dfs(12,4) - 0,0

dfs(13,0) - 0,0

dfs(13,1) - 0,0

dfs(13,2) - 0,0

dfs(13,3) - 0,0

dfs(13,4) - 0,0

dfs(14,0) - 0,0

dfs(14,1) - 0,0

dfs(14,2) - 0,0

dfs(14,3) - 0,0

dfs(14,4) - 0,0

dfs(15,0) - 0,0

dfs(15,1) - 0,0

dfs(15,2) - 0,0

dfs(15,3) - 0,0

dfs(15,4) - 0,0

dfs(16,0) - 0,0

dfs(16,1) - 0,0

dfs(16,2) - 0,0

dfs(16,3) - 0,0

dfs(16,4) - 0,0

dfs(17,0) - 0,0

dfs(17,1) - 0,0

dfs(17,2) - 0,0

dfs(17,3) - 0,0

dfs(17,4) - 0,0

dfs(18,0) - 0,0

dfs(18,1) - 0,0

dfs(18,2) - 0,0

dfs(18,3) - 0,0

dfs(18,4) - 0,0

dfs(19,0) - 0,0

dfs(19,1) - 0,0

dfs(19,2) - 0,0

dfs(19,3) - 0,0

dfs(19,4) - 0,0

dfs(20,0) - 0,0

dfs(20,1) - 0,0

dfs(20,2) - 0,0

dfs(20,3) - 0,0

dfs(20,4) - 0,0

dfs(21,0) - 0,0

dfs(21,1) - 0,0

dfs(21,2) - 0,0

dfs(21,3) - 0,0

dfs(21,4) - 0,0

dfs(22,0) - 0,0

dfs(22,1) - 0,0

dfs(22,2) - 0,0

dfs(22,3) - 0,0

dfs(22,4) - 0,0

dfs(23,0) - 0,0

dfs(23,1) - 0,0

dfs(23,2) - 0,0

dfs(23,3) - 0,0

dfs(23,4) - 0,0

dfs(24,0) - 0,0

dfs(24,1) - 0,0

dfs(24,2) - 0,0

dfs(24,3) - 0,0

dfs(24,4) - 0,0

dfs(25,0) - 0,0

dfs(25,1) - 0,0

dfs(25,2) - 0,0

dfs(25,3) - 0,0

dfs(25,4) - 0,0

dfs(26,0) - 0,0

dfs(26,1) - 0,0

dfs(26,2) - 0,0

dfs(26,3) - 0,0

dfs(26,4) - 0,0

dfs(27,0) - 0,0

dfs(27,1) - 0,0

dfs(27,2) - 0,0

dfs(27,3) - 0,0

dfs(27,4) - 0,0

dfs(28,0) - 0,0

dfs(28,1) - 0,0

dfs(28,2) - 0,0

dfs(28,3) - 0,0

dfs(28,4) - 0,0

dfs(29,0) - 0,0

dfs(29,1) - 0,0

dfs(29,2) - 0,0

dfs(29,3) - 0,0

dfs(29,4) - 0,0

dfs(30,0) - 0,0

dfs(30,1) - 0,0

dfs(30,2) - 0,0

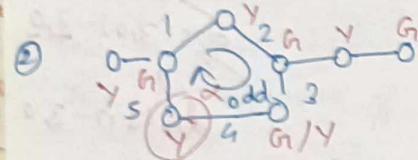
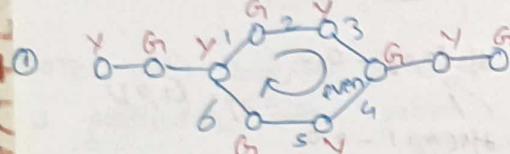
dfs(30,3) - 0,0

dfs(30,4) - 0,0

Q-17: Bipartite Graph BFS

PS] Given an adjacency list of a graph adj of V no. of vertices having 0 based index. Check whether the graph is bipartite or not.
If we are able to colour a graph with 2 colours such that no adjacent nodes have the same colour, it's called a bipartite graph.

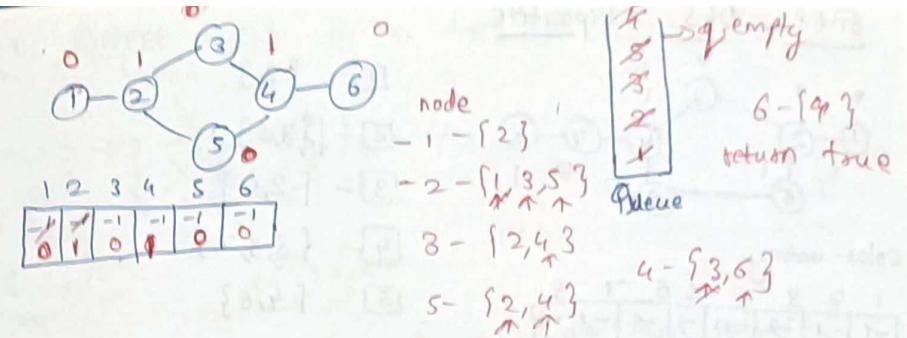
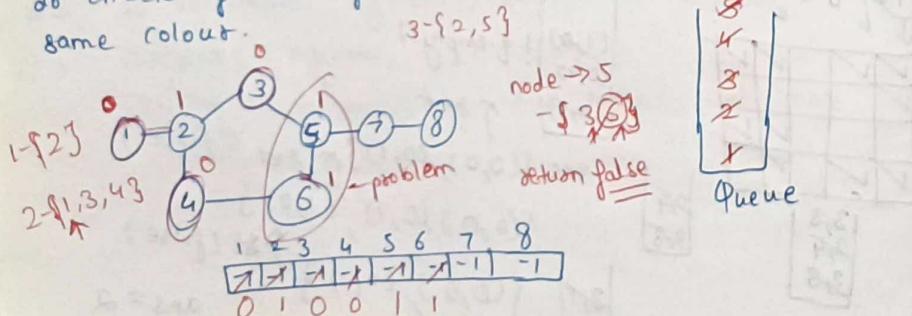
Ex-1-



③ linear graph having no cycle are always bipartite.

conclusions - Even length cycles graph is bipartite.
Odd length cycles graph is not bipartite.

Brute Force, go move and colour the adjacent and do check if two of them are not coloured with the same colour.



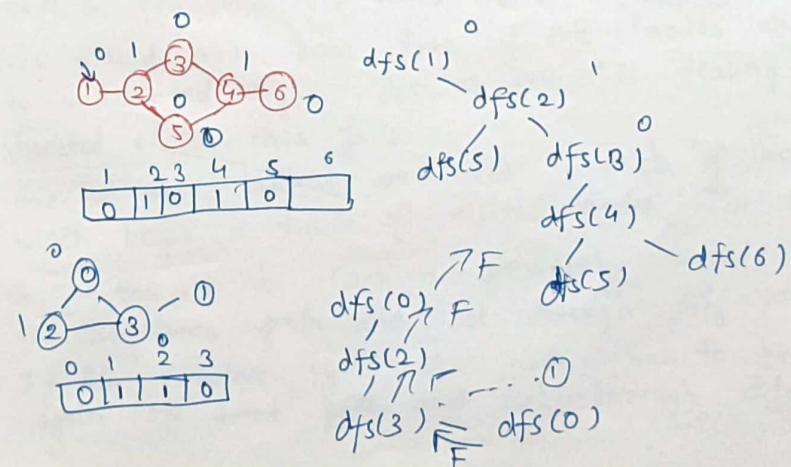
Intuition -

Brute force of fill colours, using any traversal technique just make sure no two adj. nodes have the same colour. If at any moment of traversal, we find the adj. nodes to have same color, it means that there is an odd cycle, or it can't be a bipartite graph.

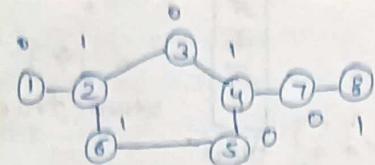
Larger test case Fails - Connected Components

TC $\rightarrow O(V+2E)$, where V = vertices, $2E$ is total degrees as we traverse all adjacent nodes

SC $\rightarrow O(3V) \sim O(V)$, space for Queue DS, colour array and an adjacency list.



G-18: DFS Bipartite



color array

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

DFS(1, 0)

DFS(2, 1)

DFS(3, 0)

DFS(1, 0)
0 == 1 False X

DFS(4, 1) bcz if (dfs == 0) return 0

DFS(5, 0) True

DFS(7, 0) True

TC \rightarrow O(V+2E)

SC \rightarrow O(N)

DFS(6, 1) 2nd 6 have same color
vis[2] = 1 == 1 return false

1 2 3

2 3 6

3 2 4

4 3 5 7

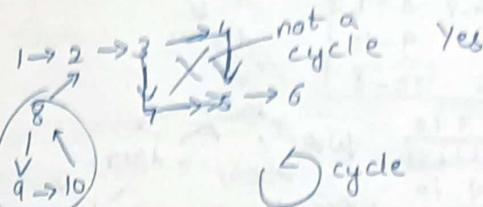
5 4 6

6 2 5

7 4 8

8 7 3

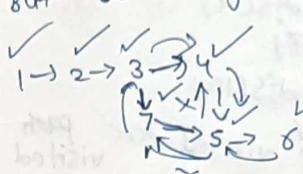
G-19: Detect cycle in a directed graph using DFS



cycle

for an undirected graph, if we visit an adjacent node which is not the parent and already visited then we say it's a cycle.

But UDG algo won't work here,

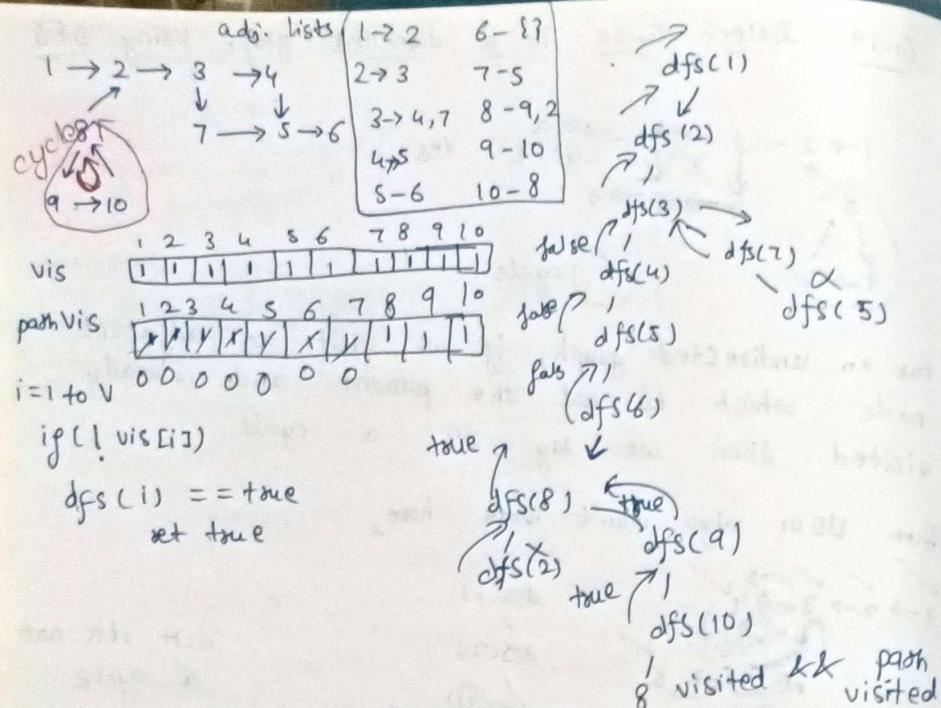


dfs(1)
dfs(2)

dfs(3)
dfs(4)
dfs(5)
dfs(6)

But it's not a cycle

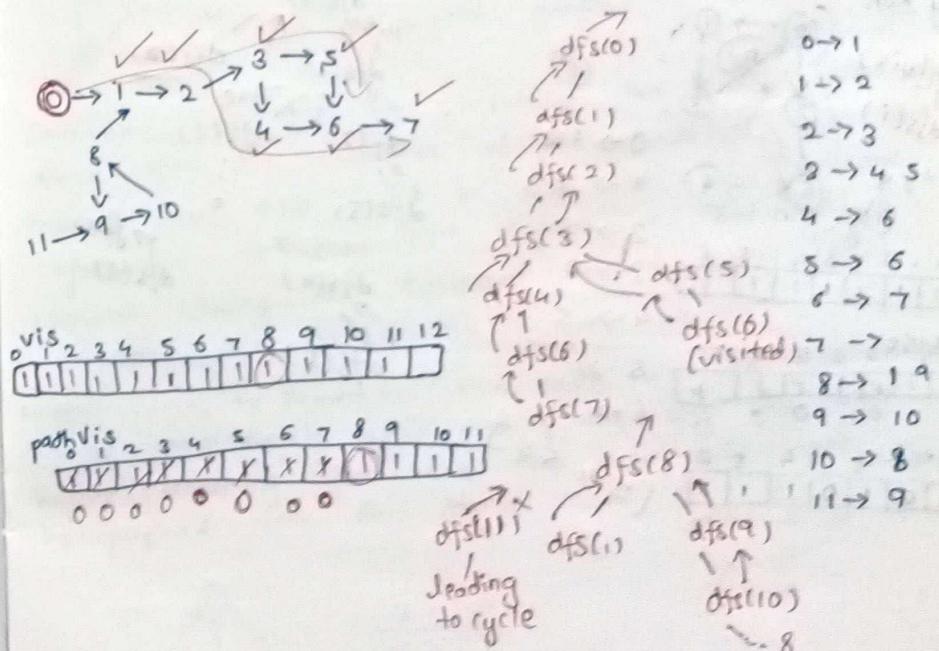
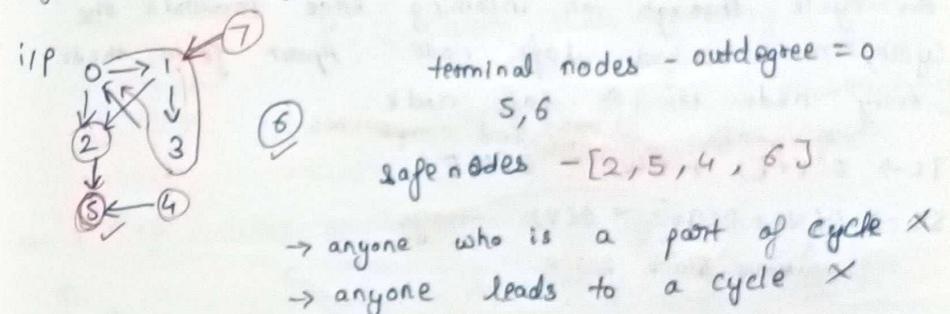
- node 5 has been vis. twice foll 2 diff paths
- this would have been true if the nodes are connected to undirected edges. But as we are dealing with directed edges this fails
- due to above reason, we need to think of an algo, which keeps a track of visited nodes, in traversal on
- so, intuition is to reach a previously visited node again on the same path and not through diff. paths.
- If DG contains cycle, the node has to be visited again on same path and not through diff. paths.



TC $\rightarrow O(V+E) + O(V)$ where, N - no. of vertices and E - edges. These can be at most V components so another $O(V)$ TC

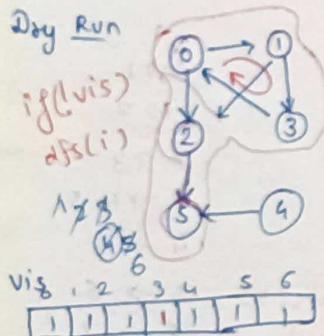
SC $\rightarrow O(2N) + O(N) \approx O(V) \dots O(2V)$ for vis and samePath and $O(V)$ for rec. stack space

- 6-20: Eventual Safe Nodes
- A node is terminal if there are not outgoing edges.
 - A node is safe if every possible path starting from that node leads to a terminal node.
 - Return all array safe nodes. The answer should be in ascending order.



So, the intuition is to find all the nodes which are either a part of a cycle or incoming to the cycle. We can do this easily using the cycle detection technique that was used previously to detect cycle in a DAG.

→ Points to remember :-
 Any node which is a part of a cycle or leads to the cycle through an incoming edge towards the cycle, can't be a safe node. Apart from these, every node is a safe node.
 $T_C \rightarrow O(V+E) + O(V) \sim O(V+E)$
 $SC \rightarrow O(V) + O(3V) \sim O(V)$ arrays
 recursive Stack space

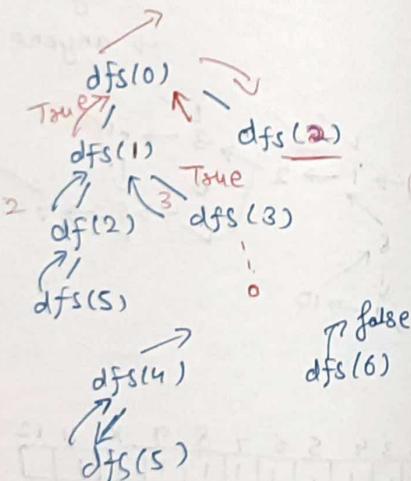


path, vis

2	3	4	5	6
0	0	0	0	0

check

2	3	4	5	6	
0	0	1	0	1	1



G-21 : Topological Sort Algorithm (DAG) directed Acyclic Graph
 Any linear ordering such that u always appears before v, if there's an edge b/w u and v.

Why Topo Sort only exists in DAGs:

Case 1 (If the edges are undirected)

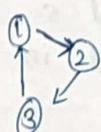
① — ②

1 → 2
2 → 1

1 2 X
2 1 X

It is practically impossible to write such ordering where u appears before v and v appears before u simultaneously so, only directed graphs.

Case 2 (If directed graph contains a cycle)

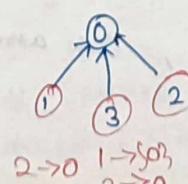


1 2
2 3
3 1

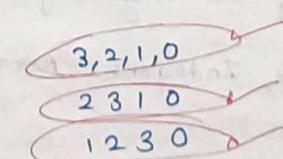
cyclic dependency, so only for acyclic graph.

Intuition: Since we are inserting the nodes into the stack after the completion of the traversal, we are making sure, there will be no one who appears afterward but may come before in ordering as everyone during the traversal would have been inserted into the stack.

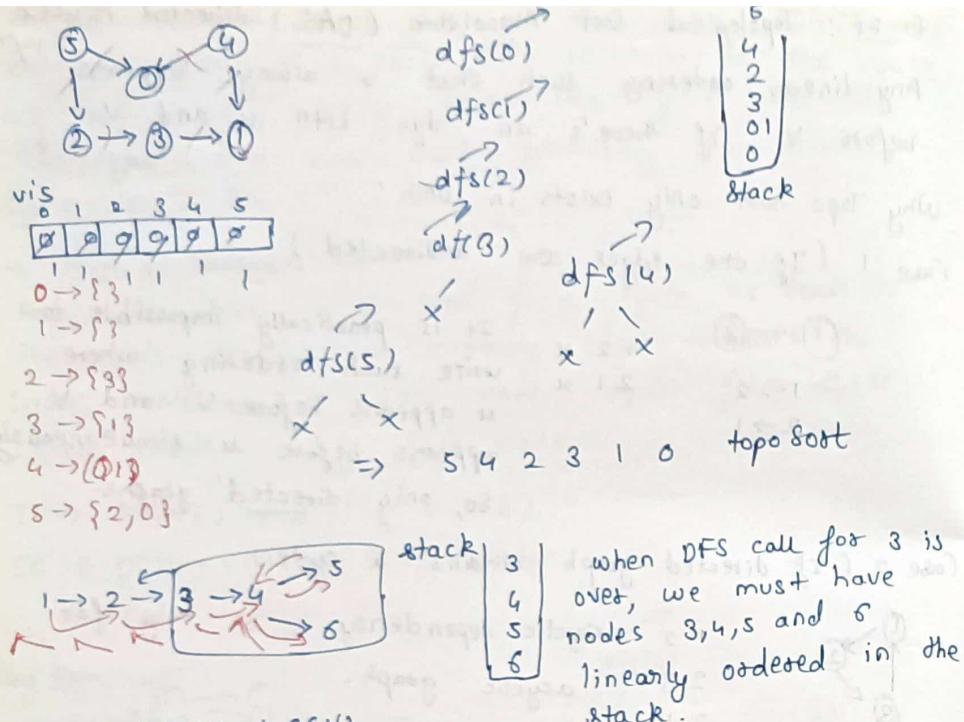
Note: Points to remember, that node will be marked as visited immediately after making the DFS call and before returning from the DFS call, the node will be pushed into the stack.



2 → 0
1 → 0
2 → 0



i = 0 - V
 ① → 0 < map[0]
 map[1] > map[0]
 1st index < 0

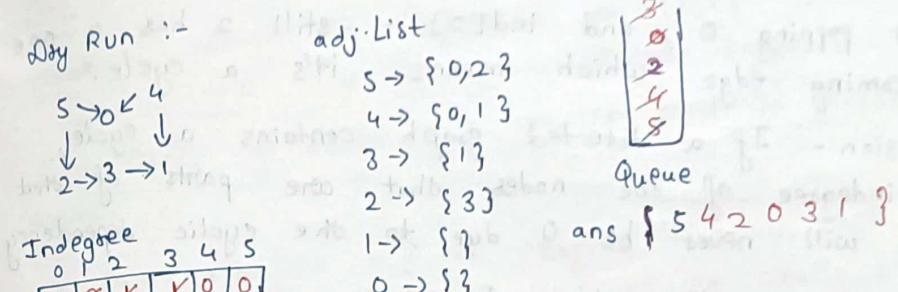


TC $\rightarrow O(V+E) + O(V)$
 $V - \text{no. of nodes}$ $E - \text{no. of edges}$
 $O(V)$ $O(E)$
 at most V connected components
 $SC \rightarrow O(2N) + O(N) \sim O(2N)$
 vis and stack

G-22: **Kahn's Algorithm** | Topological Sort | BFS
 Intuition: we will slightly modify BFS where we will be keeping an array to store the indegree of each node.

(2) indegree = no. of incoming edges = 2
 Initial config - Queue, Indegree Array, Ans array

When a node's indegree becomes zero we will push them into the queue. Meanwhile, we include the curr one in answer immediately after it's popped out of queue.

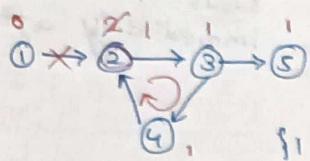


Because, s has no indegree so one points to s hence, no one will appear before s in the ordering thus we'll append it to our answer.

TC $\rightarrow O(V+E)$, where V - no. of nodes, E - no. of edges.
 $SC \rightarrow O(N) + O(N) \sim O(2N)$, $O(N)$ for indegree array, and $O(N)$ for queue data structure used in BFS

G-23: Detect Cycle in DAG | Kahn's Algo
 Intuition:
 - Since we know topo sort is only possible for directed acyclic graphs (DAGs) if we apply Kahn's Algo in a DAG, it will fail to find the Topo Sort.
 - So we finally check the sorting to see if it contains all V vertices or not. If result does not include all V vertices, we can conclude that there is a cycle.

We will check if size of the final Topological Sort equals V (no. of vertices or nodes) or not.



Size != V
↳ true (cycle)
else false
(acyclic)

After popping 0, and $\text{ind}[2] = 2$ still 2 has a one incoming edge which means it's a cycle.

Conclusion - If a directed graph contains a cycle, the indegree of the nodes that are parts of that cycle will never be 0 due to the cyclic dependency.

$TC \rightarrow O(V+E)$ V-vertices, E-edges

$SC \rightarrow O(N) + O(N) \sim O(N)$... $O(N)$ - for in-degree
 $O(N)$ - Queue Data Structure
 $N \rightarrow$ no. of nodes

G-24 : Course Schedule I and II | Topological Sort

There are a total of N tasks, labeled from 0 to $N-1$. Some tasks may have prerequisites, for eg - To do task 0 you have to first complete task 1, which is expressed as $\{0, 1\}$

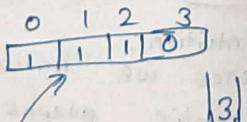
Given N tasks and list of PreReq: P, find if its possible to finish them.

① $[1, 0]$
[2, 1]
[3, 2]

$3 \rightarrow 2 \rightarrow 1 \rightarrow 0$

(no cycle)

YES



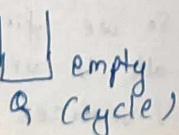
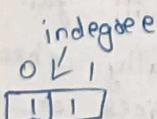
[3]

② $[1, 0]$
[0, 1]

$1 \rightarrow 0$

(cycle)

X NOT POSSIBLE



empty
(cycle)

Stated -

Whenever you see something before something that is when doing Topo sort should hit your brain.

PS-2 : To do task 0 you have to first complete task 1, which is expressed as $[0, 1]$

$1 \rightarrow 0$ $u \rightarrow v$ u appears before v .
Find Order of tasks you should find to finish all tasks.

pair u, v signifies that to perform task v , first we need to finish task u . Now, if we closely observe, we can think of a directed edge between u and v ($u \rightarrow v$) where u and v are two nodes. Each task \rightarrow node, Each pair - directed edge b/w u and v , the whole problem becomes a graph problem.

Intuition :

For problem 2, the intuition is to find the linear ordering in which the tasks will be performed if it is possible to perform all the tasks otherwise, to return an empty array.

For P1, the intuition is to find if it's possible to perform all tasks. (i.e. The graph contains a cycle or not).

We can now solve both using Topological Sort or Kahn's Algorithm. $TC \rightarrow O(V+E)$ V-vertices, E-edges

$SC \rightarrow O(N) + O(N) \sim O(N)$ - one for indegree array
+ $O(N)$ and one for queue DS and extra $O(N)$ for Topo Sort Array.

G-25 : Eventual Safe States using BFS

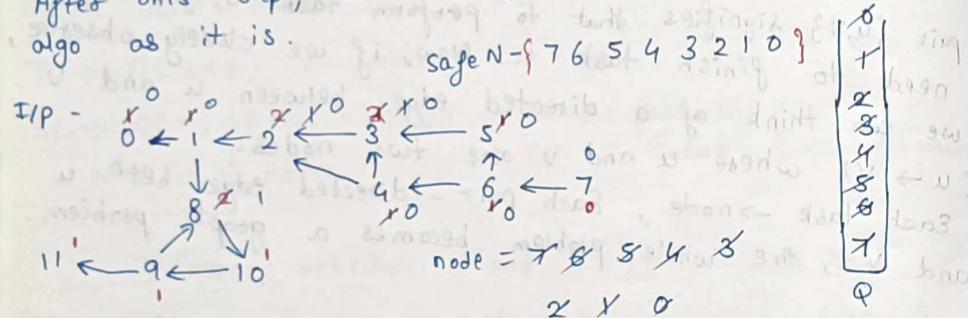
Terminal Node - no outdegree (safe node)

Safe Node - Every outdegree leads to a Terminal Node.

Approach:

The node with outdegree 0 is considered to be a terminal node but the Topo Sort algorithm deals with the indegrees of the nodes. So, to use the Topological sort, we will reverse every edge of the graph. Now, the nodes with indegree 0 become the terminal nodes.

After this step, we will just follow the Topo sort algo as it is.



i → it indegree(i)++ ;
it → i

$2 \rightarrow \{1, 3\}$ $2 \rightarrow 1$
 $2 \rightarrow 3$

$1 \rightarrow \{2, 3\}$
 $3 \rightarrow \{2, 3\}$ indegree[2]++
 (2)

TC $\rightarrow O(V+E) + O(N \log N)$, where, V - no. of nodes, E - no. of edges. Extra $O(N \log N)$ for sorting safeNodes, where N is no. of safe nodes.

SC $\rightarrow O(N) + O(N) + O(N) \approx O(3N)$, indegree, Queue and adjacency reverse list.

G-26 : Alien Dictionary

PS \Rightarrow Given a sorted dictionary of an alien language having N words and k starting alphabets of a standard dictionary. Find order of characters in the alien language.

a, b, c, d ... x, y, z \rightarrow (26)

i/p N=5, k=4

baa	b < a (b \rightarrow a)
ab c d	d < a (d \rightarrow a)
ab c a	a < c (a \rightarrow c)
cab	b < d (b \rightarrow d)
ca d	order

$b \rightarrow a \rightarrow c$
1 \rightarrow 0 \rightarrow 2
 3

(oldizing len)

caa	c > a (c \rightarrow a)
aaa	a < b (a \rightarrow b)
aab	cab (3)

c \rightarrow a \rightarrow b

Reason why need not check

caa and aab (1 and 3) :-

bcz we have figured out why 'caa' appears before 'aaa'. So, by convention if 'aaa' before 'aab' and 'caa' appears before 'aaa', obviously 'caa' appears before 'aab'. Thus, we check pair wise.

To further simplify, we will map with numbers like 'a' with 0, 'b' \rightarrow 1, 'c' \rightarrow 2 ...

Note: The intuition is to check every consecutive pair of words and find out the diff. factors. With these factors, we will form a directed graph, and the whole problem boils down to a Topo-Sort of problem.

Edge Case : for 3rd hyp if $K=5$ no word having 5
we add a separate node 'e'

$$b \rightarrow a \rightarrow c$$

$$\searrow d$$

③

$$1 \rightarrow 0 \rightarrow 2$$

$$\searrow 3$$

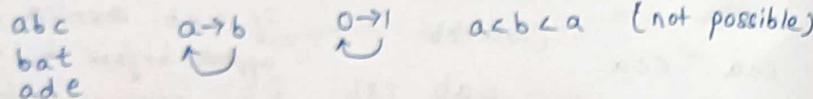
④

Follow-ups:

• When the ordering is not possible?

- ① If every char. matches and longest word appears before the shortest word
 abcd "Not Possible" (if all same, shorter should come before longer)

- ② If there's a cyclic dependency



We apply Kahn's Algorithm to solve

try run -

baa

$b \rightarrow a$

$1 \rightarrow \{0, 3\}$

$\begin{matrix} 0 & 1 & 2 & 3 \\ \times & 0 & x & x \end{matrix}$

abcd

$d \rightarrow a$

$3 \rightarrow \{0\}$

$\begin{matrix} 0 & 1 & 2 & 3 \\ x & 0 & 0 & 0 \end{matrix}$

abca

$d \rightarrow a$

$3 \rightarrow \{2\}$

$\begin{matrix} 0 & 1 & 2 & 3 \\ x & 0 & 0 & 0 \end{matrix}$

cab

$a \rightarrow c$

$0 \rightarrow \{2\}$

$\begin{matrix} 0 & 1 & 2 & 3 \\ x & 0 & 0 & 0 \end{matrix}$

cad

$0 \rightarrow a$

$2 \rightarrow \{3\}$

$\begin{matrix} 0 & 1 & 2 & 3 \\ x & 0 & 0 & 0 \end{matrix}$

$b \rightarrow d$
 $1 \rightarrow 3$

ans = bdac

TC $\rightarrow O(n \times \text{len}) + O(K+E)$, N - no. of words, 'len' is upto index where first inequality occurs, X - no. of nodes, E - no. of edges

SC $\rightarrow O(K) + O(K) + O(K) + O(K) \approx O(4K)$

↑
Odg. List
↑
Indegree Queue
↑
and to
state the order

Swap Two Numbers in 5 diff. ways ?

① temp = a, a = b, b = temp

② b = a+b
a = b-a

or
b = a * b
a = b/a

b = b-a
a = b/a

③ In-built swap(a,b)

⑤ b = a-b + (a=b)

(Optimal)

④ a = a ^ b
b = b ^ a

a = a ^ b

a = 2, b = 3

④ b = 3+2 = 5
a = 5-2 = 3

b = 5-3 = 2

② a = 2^3 = 1

b = 1^3 = 2

a = b = 3

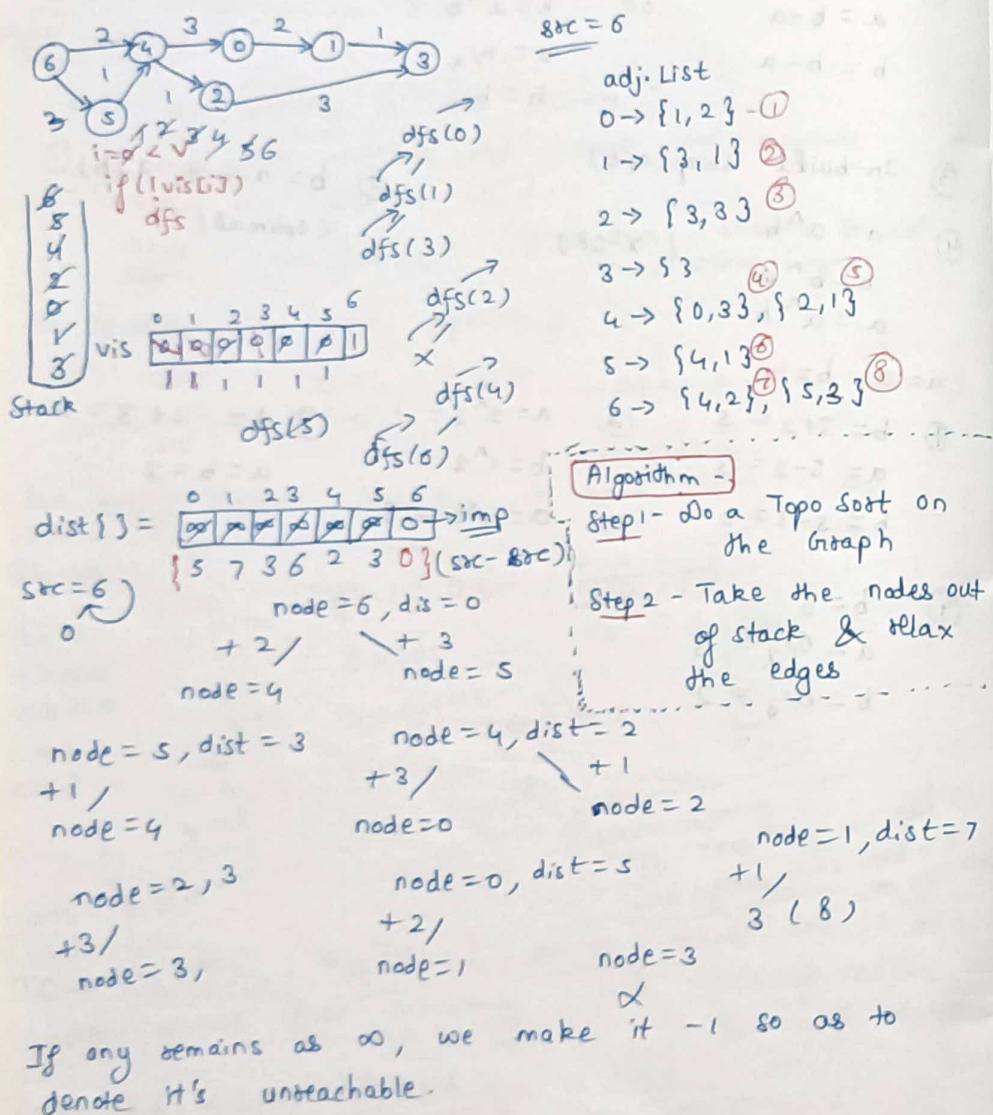
④ b = 6

a = 6/2 = 3

b = 6/3 = 2

G-27: Shortest Path in Directed Acyclic Graph - Topological Sort

From any source node -



Intuition: Finding shortest path to a vertex is easy if you already know the shortest paths to all the vertices that can precede it. Finding the longest path to a vertex in DAG is easy if you already know the longest path to all the vertices that can precede it. Processing the vertices in topological order ensures that by the time you get to a vertex, you've already processed all vertices that can precede it. Dijkstra's is useful for graphs that contain cycles, bcz they can't be topologically sorted.

Preconfiguration - Adj. List, Visited Array, Stack, Distance Array
 → initialise dist. arr with max INT value, update the scc dist[scc] to be zero (scc → scc)

TC → $O(N+M)$ {for Topo Sort} + $O(N+M)$ {for relaxation of vertices, each node and its adjacent nodes get traversed} (N) (E) ... directed $\sim O(N+M)$

SC → $O(N)$ {for stack storing TopoSort} + $O(N)$ {dist-} + $O(N)$ vis. + $O(N+2M)$ {for adjacency list} $\sim O(N+M)$ N nodes, E edges

Gr-28: Shortest Path in an Undirected Graph with Unit Weights.

PS- You are given an undirected graph having unit weight, find shortest path from src to all the vertex and if it is unreachable to reach any vertex, then return -1 for that vertex.

Ex-1. $n=9, m=10$

$$\{0,1\} \quad 0 \rightarrow \{1,3\}$$

$$\{0,3\} \quad 1 \rightarrow \{0,2,7\}$$

$$\{3,4\} \quad 2 \rightarrow \{1,6\}$$

$$\{4,5\} \quad 3 \rightarrow \{4,0\}$$

$$\{5,6\} \quad 4 \rightarrow \{3,5\}$$

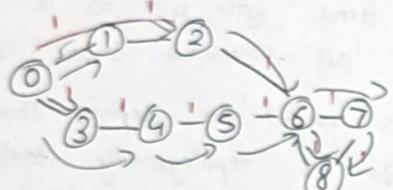
$$\{1,2\} \quad 5 \rightarrow \{6,4\}$$

$$\{2,6\} \quad 6 \rightarrow \{7,2,5\}$$

$$\{6,7\} \quad 7 \rightarrow \{6,8\}$$

$$\{7,8\} \quad 8 \rightarrow \{6,7\}$$

$$\{6,8\} \quad 9$$



O/P -

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

$\begin{array}{c} \{8,4\} \\ \{2,4\} \\ \{5,8\} \\ \{5,6\} \\ \{4,2\} \\ \{4,1\} \\ \{3,2\} \\ \{3,1\} \\ \{1,2\} \\ \{0,2\} \end{array}$

node = 0, 0
+1 / +1
node = 1
node = 1, 1
+1 / +1
node = 2, 1
dist = 2

dist

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

node = 2, 2
+1 / +1
node = 1
dist = 3

node = 3, 1
+1 / +1
node = 4
dist = 2

node = 4, dist = 2
node = 5, dist = 3
node = 3, dist = 3
node = 4, dist = 4
node = 5, dist = 4
node = 6, 3
node = 7, 2
node = 8, 1
node = 9, 0

Intuition: we will calculate the shortest path using BFS. BFS is traversal technique where we visit the nodes level-wise i.e. visits the same level nodes simultaneously, and then moves to the next level.

Algorithm (Pseudo Code)

- make the adjacency list
- create distance array of size $N (\infty)$, mark src as 0
- perform BFS
 - if $dist[\text{node}] + 1 < dist[\text{adjacent}]$
 $dist[\text{adjNode}] = dist[\text{node}] + 1$
 - push if (true) q.push(adjNode, dis[adjNode])
- else after BFS
 some one is unreachable mark it as -1.

TC $\rightarrow O(M) \{ \text{for adj. list creation} \} + O(N+2M) \{ \text{for BFS} \}$
 $+ O(N) \{ \text{for adding } -1 \text{ to unreachable nodes} \}$
 $\sim O(N+2M)$

SC $\rightarrow O(N) \{ \text{for BFS Queue} \} + O(N) \{ \text{for dist. array} \}$
 $+ O(N+2M) \{ \text{for adj List} \} \sim O(N+M)$

where,
 N - no. of vertices
 M - no. of edges

Gr-29: Word-Ladder - I

(Hard) PS- Given two distinct words startword and targetword and a wordlist. Find length of shortest transformation

- A word can only consist of lowercase characters
- only one letter can be changed in each trans.
- each transformed word must exist in wordlist including target word.
- startword may or may not be part of wordlist

i/p - word list - { des, det, dfr, dgt, dfs }

st = des, end = dfs

det → dfr → dfs ans = 3

i/p - word list - { poon, plee, same, poie, plea, plie, poin }

st = poon, end = plea

1 poon
2 poon
3 poin
4 poie - plie - plea
5

ans = 7

word st = 'hit' end = 'cog'

word List = [hot, dot, dog, lot, log, cog]

hit → hot → dot → dog → log → lot → cog

hit → level 1
a-2
hot, hit, hot, hot

hot → 2
dot → 3
dot → 4
dot → 5

hot → 4
hot → 5 { shortest sequence }

Day Run -
set → hot, dot, dog, lot, log, cog

cog, 5
log, 4
dog, 4
lot, 3
dot, 3
hot, 2
hit, 1

DCN

Queue

hit hot
| / \
hot dot lot

dot lot
| / \
dog dot log

dog log
| / \
cog

mumbai - Odisha → hyderabad - Kerala
kolkata - Bangalore - Odisha → hyderabad

'hot' is executed - then it'll not process it further.

Time Complexity - $O(N^*M^*26)$

where, N = size of wordList Array and M = word length of words present in the wordList.

Note that, hashing operations in an unordered set takes $O(1)$ time, but if you want to use set here, then time complexity would increase by a factor of $\log(N)$ as hashing operations in a set take $O(\log(N))$ time.

SC → $O(N)$ { for creating set and copying all wordList words into it }

where, N → size of wordlist array

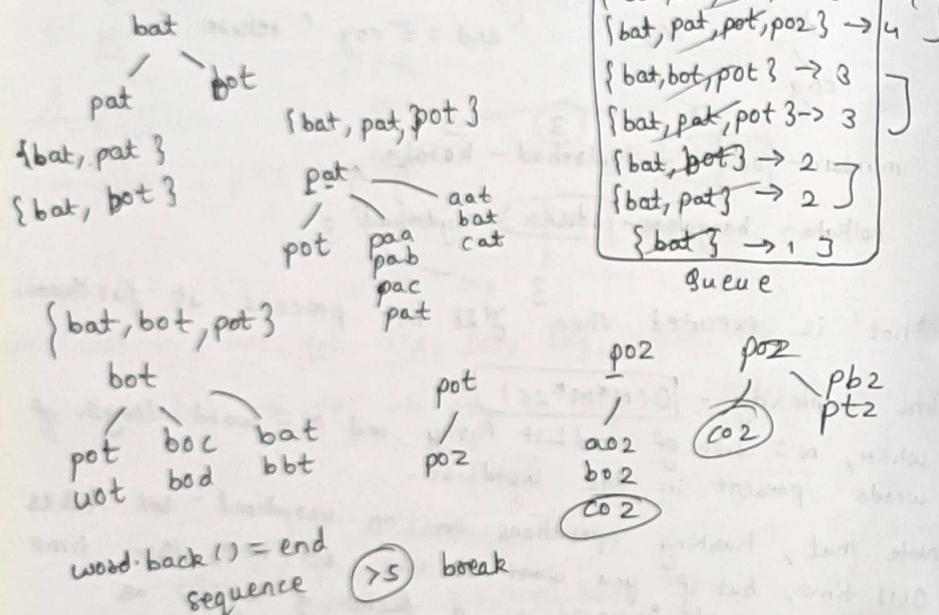
Q-30: Word Ladder - 2

Find all shortest transformation sequence from st to end. You can return them in any order possible.

Return empty list if there is no such transformation sequence.

Q. wordList = [pot, bat, pat, ~~pat~~, ~~bat~~, ~~bat~~]

begin word = bat word = bat



Contrary to previous one, here we don't stop the traversal on the first occurrence of the target word, but continue for as many occurrences of word as possible as we need all the shortest possible sequences in order to reach the destination word. Don't delete a word immediately from list even if it matches bcz, we delete it after the traversal of a particular level meanwhile we can keep a track which guys have to be deleted in the next level.

Algorithm -

Step 1. Store list in a hashset which would make search and delete operations faster.

Step 2. Queue to store paths in form of vector

Step 3. add {stword} to list and also to usedOnLevel arr[]

Step 4. Pop first element (i.e. back of the path)

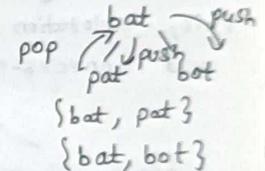
- word - check all its char by 'a'-'z' if present in wordList - push it into the usedOnLevel vector and don't delete immediately

Step 5. Push that word into the sequence and add it to the Queue. and pop out the next step.

Step 5. If vec.size() > level // new level

level++

for(auto a : used)
st.erase(a)



Step 6. If last word present at top of queue is equal to target we push it to ans (list<list>)

Step 7. If ans ! empty then check ans[0].size() == vec.size() this needs to be checked for shortest sequence.

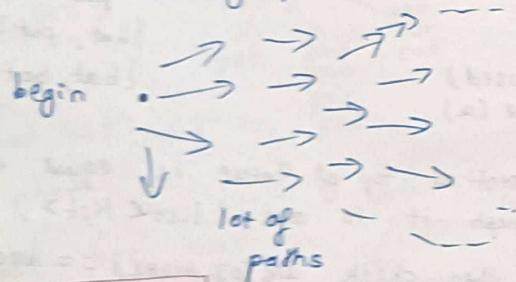
Step 8. In case none, return empty {}.

Tc → Cannot be predicted bcz there can be multiple sequences of transformation from st to end, depends on example. So we can't define a fixed range of time or space in which this program would run, for all test cases.

G-31: Word Ladder - 2 (Optimized Approach)

(TLE) → (Submit) (Leetcode 126.)

Intuition: Now, as the first step instead of storing the sequences, we just store the words as we progress during the BFS traversal. This would give us an idea abt. the length of the shortest sequences possible. We also store words along with level on which they appear during the traversal in a map DS so that we can have count of possible no. of paths to reach target. In the end, we backtrack from end to begin to get the answer sequences. Through this, exploration of paths would be minimal, from back and unnecessary paths wouldn't be explored.



INTERVIEW TIP:

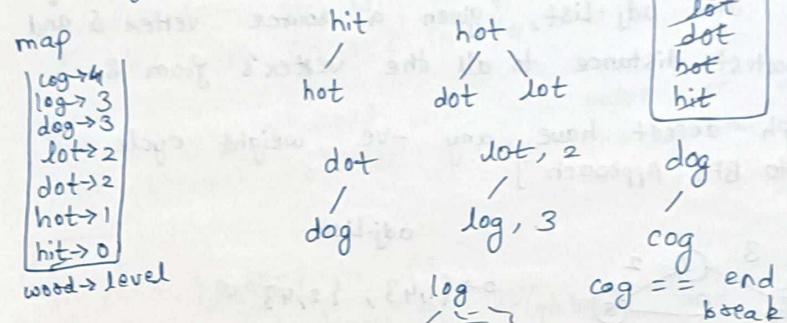
DON'T DECLARE GLOBAL VARIABLES

Initial Configuration:

- 1] Vector (Store sequences)
- 2] Map : (word → level)
- 3] Hash Set (wordList)
- 4] Queue (BFS)

wordList = [bot, dot, dog, hot, hit, cog]

beg = hit, end = cog



④ cog, { cog }

dfs ③ dog, { cog, dog }

② dog, { cog, log, dog }

dfs ② dot, { cog, dog, dot }

① hot, { cog, log, dot, hot }

dfs ① hot, { cog, dog, dot, hot }

pop-back()

⑤ hit, { cog, dog, dot, hot, hit }

dfs == beginword
ans
base case

Time Complexity and Space Complexity →
Cannot be predicted, depends on the test cases we
can't define a fixed range of time or space.

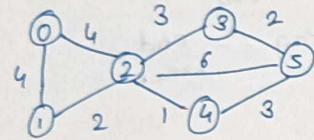
G-32: Dijkstra's Algorithm: Using Priority Queue (~~PQ~~ set)

Ps: Given weighted, undirected and connected graph of V vertices and adj. list, given a source vertex S and find shortest distance to all the vertex's from S .

Note: Graph doesn't have any -ve weight cycle.

[Similar to BFS Approach]

i/p-

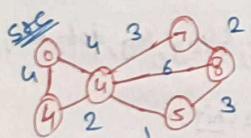


Und.(Cycle).Graph

adj.List

- $0 \rightarrow \{1, 4\}$, $\{2, 4\}$
- $1 \rightarrow \{0, 4\}$, $\{2, 2\}$
- $2 \rightarrow \{0, 4\}$, $\{3, 3\}$, $\{4, 1\}$, $\{1, 2\}$, $\{5, 6\}$
- $3 \rightarrow \{2, 3\}$, $\{5, 2\}$
- $4 \rightarrow \{2, 1\}$, $\{5, 3\}$
- $5 \rightarrow \{2, 6\}$, $\{3, 2\}$, $\{4, 3\}$

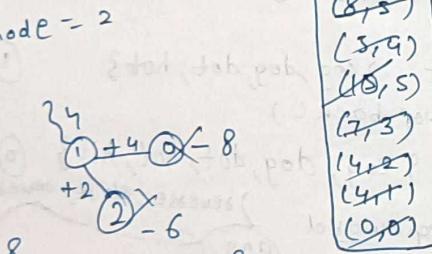
$soc = 0$
 $dis = 0$
 $+h/$
 $node = 1$



$dist$

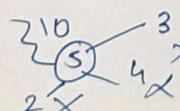
0	4	4	7	5	10
0	1	2	3	4	5

$+5$
 $+3$
 $node = 2$
 $dis = 6$



$\{dist, node\}$

$node = 3, 7$
 $+3/$
 $node = 2$
 $dis = 10$



$+6/$
 $+2/$
 $+3$
 $n=2$
 $n=3$
 $n=4$
 x_0
 x_1

Approach -

Use Priority Queue - store {distance, node}.

Source Node - src will be given or we can take one and push $\{0, src\}$ in PQ

Dist Array - initialized with large value, it will store the shortest distances at the end of the day.

Algo -

1. push $\{0, soc\}$ in PQ
2. pop it and travel its neighbours and if $currDistance + weight < dist[adjNode]$
 - push in PQ $\{currDistance + w, adjNode\}$
 - $dist[adj] = curr + w$
3. Repeat step 2 until PQ is not empty

Dijkstra's Algorithm is not valid for negative weights or negative cycles.

Eg -

$⑥ \rightarrow ①$

$dist$

0	1
0	1

PQ

$⑥ \rightarrow ②$ $⑦ \rightarrow ②$

TC $\rightarrow O(E \log V)$ E - edges, V - vertices

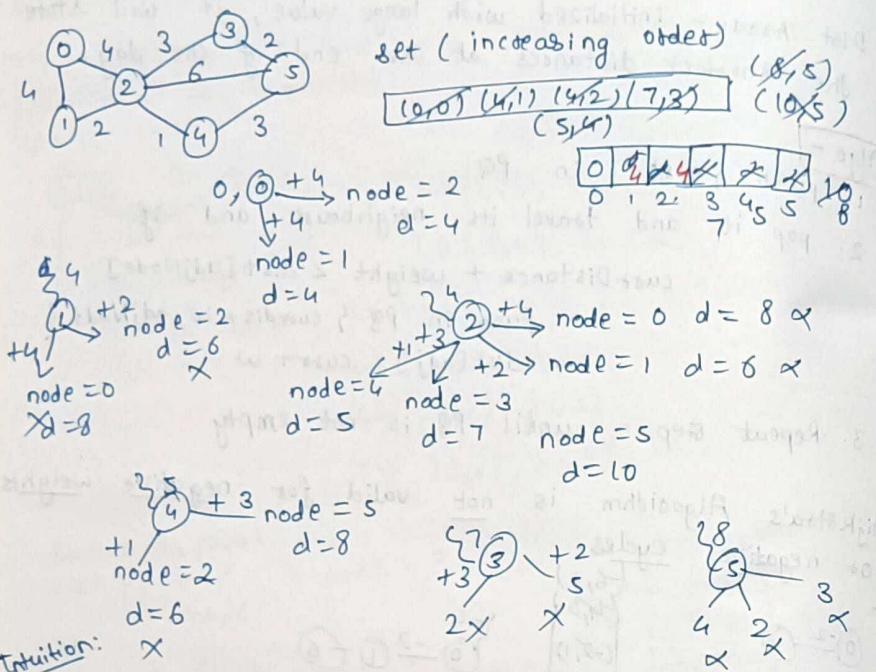
SC $\rightarrow O(|E| + |V|)$, E - edges, V - vertices

$④ \rightarrow ⑤$

Thus, due to negative weight of $④ \rightarrow ⑤$ the distance always decreases, and we get stuck in an infinite loop.

G-33: Dijkstra's Algorithm - Using set - Part 2

(Priority Queue just iterates but with set we have the liberty to erase and find)



The only difference using a PQ and a set is that in a set we can check if there exists a pair with same node but a greater distance than the current inserted node as there will be no point in keeping that node into the set if we come across much better value than that.

i.e. we have node that has been reached by two paths, say with a cost of 7 and 9. It's obvious that the path with a cost of 7 is more optimal than 9. Both ways are fine we can't explicitly say that set is better bcz by saving iterations we still took

$O(\log N)$ for set.erase(), minor-minor difference.

Gr-34: Why PQ and not Q, Intuition, Time Comp. Detiv'n

Difference is we need to traverse all connected nodes of a node and find the minimum among them when we use a normal queue it takes $O(V)$ time but using PQ it takes $\log(V)$

In this example, we travel from 0 to 3, we will explore both the paths 0-1-3 and 0-2-3 but in case of PQ we only explore minimal i.p 0-2-3.

Derivation of $TC \rightarrow$



 Derivation of TC →



 Pseudocode -
 $\text{while } (\text{! pq.empty}()) \rightarrow V \text{ times}$
 $\quad \left\{ \begin{array}{l} \text{dist node} = \text{top}() - \log(\text{Heapsize}) \\ V-1 \rightarrow \text{for all adj nodes} \\ \quad \left(\begin{array}{l} \text{each node} \\ \text{is connected if condition} \\ \text{to other nodes} \end{array} \right) \\ \quad \text{update dist} \\ \quad \text{push in pq} \end{array} \right.$
 $O(V * (\text{pop vertex from pq}) + \text{no. of edges on each vertex} * \text{push in pq})$
 $O(V * [\log(\text{heapsize}) + V-1 \log(\text{Heapsz2})])$

$O(V^* \log(\text{heapsize}) (X+V-X))$

$O(V^*V^* \log(\text{heapSize}))$

Now, at worst case, the heap will be V^2 if we consider pushing adj. elements of a node, at worst case each element will have $V-1$ nodes and they will be pushed in PQ.

$$O(\sqrt{v^2} \log(v^2)) \Rightarrow O(E \cdot \log v)$$

$$O(2^{N^2} \log v) = O(E \cdot \log v^2)$$

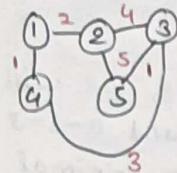
$$= O(E \log V) \text{ worst case}$$

$$Sc \rightarrow O(E + V) \quad O(E \log E) \text{ for } PQ$$

and all to
 $V-1$
 $V^*(V-1)$
 $= V^2$ edges
 or
 E edges

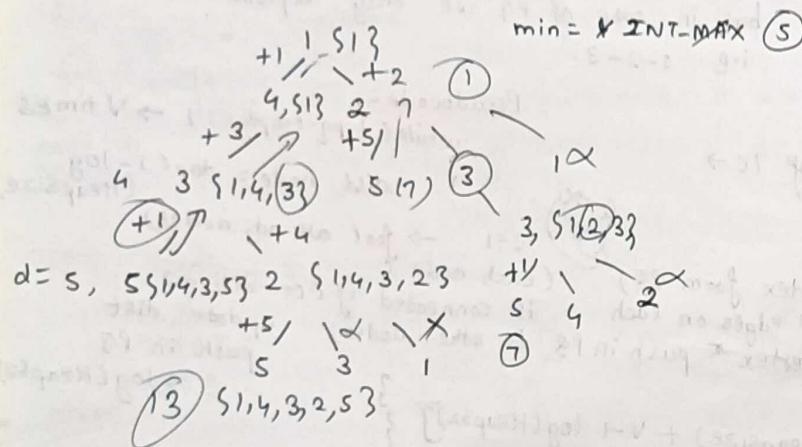
Graph

PS - given a weighted undirected graph having $n+1$ vertices numbered from 0 to n , edges, and m edges. Find shortest path betⁿ vertex 1 and n if not -1.



$$\begin{aligned}1 \rightarrow 2 \rightarrow 5 &= 7 \\1 \rightarrow 2 \rightarrow 3 \rightarrow 5 &= 7 \\1 \rightarrow 4 \rightarrow 3 \rightarrow 5 &= 5\end{aligned}$$

{1, 4, 3, 5}



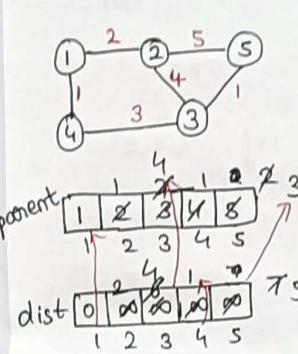
Intuition:

① Intuition behind the above problem is based on Dijkstra Algo. with a combination of little sum (bit) of memoization in order to point the shortest possible path, and not just calculate the shortest distance, in order to point we will try to remember the node from which we came while traversing each node by Dijkstra alongside calculating shortest distance.

An array called 'parent' can be used, which will store the parent node for each node and will update itself if a shorter path from a node is found at some point. Thus, we backtrack through parent array till source node.

Initial Configuration

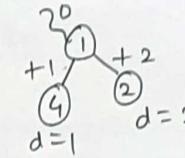
1. Source Node (1)
 2. Priority Queue {dist, node} PQ
 3. Adj. List
 4. Dist Array (∞)
 5. Parent Array
- | | | |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 2 | 3 |



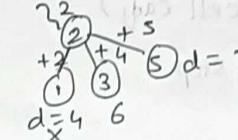
parent [1 2 3 4 5]

dist [0 ∞ ∞ ∞ ∞]

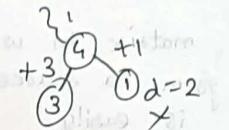
1 2 3 4 5



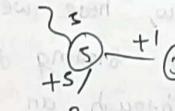
d = 2



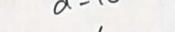
d = 7



d = 6



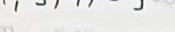
d = 5



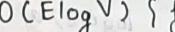
d = 4



d = 3



d = 2



d = 1



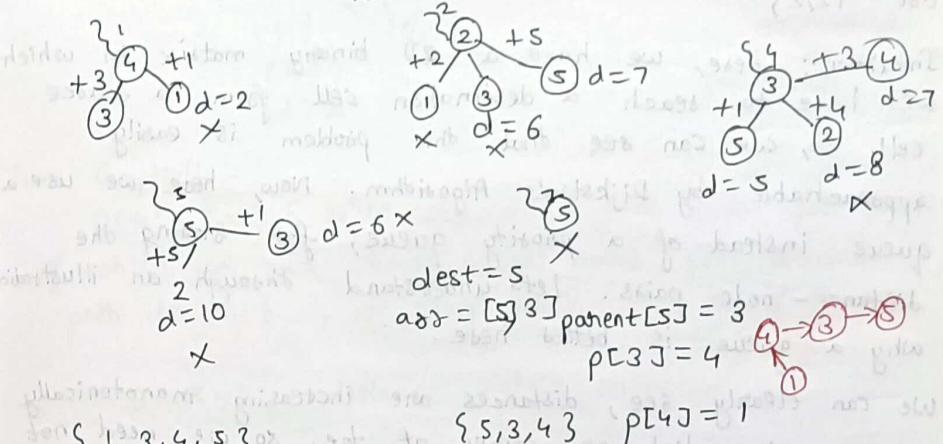
d = 0



d = -1

55, 53
52, 53
54, 53
56, 53
57, 53
52, 23
51, 43
50, 13

PQ {dist, node}



dest = 5

arr = [5] parent[S] = 3

p[4] = 1

p[1] = 1 ! = 1 false

TC $\rightarrow O(E \log V)$ {for Dijkstra's}

+ $O(V)$ {for backtracking in order to find the parent for each node}, where $E = \text{No. of edges}$ and $V = \text{No. of nodes}$

SC $\rightarrow O(E + V)$ {for priority queue and dist array}

+ $O(V)$ {for storing final path}

G1-36: Shortest distance in a Binary Maze

Given an $n \times m$ grid, where each element can either be 0 or 1. You need to find the shortest distance b/w src and dest. If not possible, return -1.

Move - $\leftarrow \uparrow \rightarrow \downarrow$

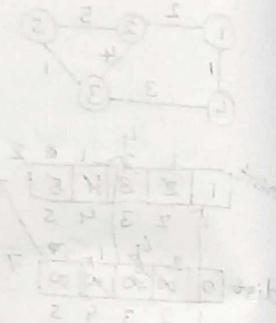
i/p	0	1	2	3
0	1	0	1	1
1	1	1	0	1
2	1	1	1	1
3	1	1	0	0
4	1	0	0	1

src = {0, 1}

dst = {2, 2}

ans = 3

cell \rightarrow cell (adj)
+1 step



Intuition: Here, we have a 2D binary matrix in which we have to reach a destination cell from a source cell. So, we can see that this problem is easily approachable by Dijkstra's Algorithm. Now, here we use a queue instead of a priority queue, for storing the distance-node pairs. Let's understand through an illustration why a queue is better here.

We can clearly see, distances are increasing monotonically since, greater distance comes at top, so we need not do with PQ as pop operation will always pop minimal guy. This helps us to eliminate extra $\log(n)$ of time needed to perform push-pop in PQ.

Initial configuration-

① Src, destination

② Queue {distance, row, col? }

③ Distance matrix [][][] with all values as ∞ but $dist[\&src] = 0$ to start off.

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

0	1	2	3
0	x	0	2
1	1	2	1
2	1	2	1
3	1	2	1
4	1	2	1

grid

distance

2, (0, 3)
2, (1, 0)
2, (2, 1)
1, (0, 0)
1, (1, 1)
1, (2, 2)
0, (0, 1)

Queue (PQ)

$$\{ (0, 1) \xrightarrow{+1} (0, 2) \}$$

$$\{ (0, 0) \xrightarrow{+1} (1, 1) \}$$

$$\{ (1, 1) \xrightarrow{+1} (0, 2) \}$$

$$\{ (1, 0) \xrightarrow{+1} (2, 1) \}$$

$$\{ (1, 0) \xrightarrow{+1} (1, 1) \}$$

$$\{ (1, 0) \xrightarrow{+1} (0, 1) \}$$

$$\{ (1, 0) \xrightarrow{+1} (2, 2) \}$$

Time Complexity -

$O(4 * N * M)$ $\$ N * M$ are the total cells, for each of which we also check 4 adj. nodes for shortest path length 3 , N - no. of rows, M - no. of columns

SC $\rightarrow O(N * M)$ - distance matrix

G-37: Path with Minimum Effort

Ps- You're a hiker preparing for an upcoming hike. You are given heights, a 2D matrix, where a cell depicts height of cell (row, col) . You are situated in the top-left cell $(0,0)$, and you hope to travel to the bottom-right cell $(\text{rows}-1, \text{cols}-1)$. You can go $\uparrow\downarrow$ and find a route that requires min. effort.

A router's effort is the maximum absolute difference in heights between 2 consecutive cells of the route.

i/p

src		
1	2	2
3	8	2
5	3	5

Dest

$$\begin{matrix} 2 & 2 & 2 \\ 1 & 0 & 0 \\ 3 & 0 & 3 \end{matrix} - \max = 3$$

$$\begin{matrix} 3 & 5 & 3 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{matrix} - \max = 2$$

$$\begin{matrix} 3 & 8 & 2 \\ 2 & 5 & 6 \\ 2 & 6 & 3 \end{matrix} - \max = 6$$

Intuition - we need to minimize effort of moving from source to dest. The effort can be calculated as max value of the difference b/w the node and its next node in the path from src to dest. Among all the possible paths, we have to minimize this effort. So, for these types of min. path, there is Dijkstra which we will use to update distance every time we encounter a value of difference less than the previous value. This way, whenever we reach destination we finally return value of difference which is also min. effort.

dry run -

0	1	2	
0	0	2	
1	3	8	2
2	5	3	5

0	1	2	
0	0	1	
1	2	8	5
2	2	2	8

PQ

$(0,0,0)$	$(1,0,2)$	$(3,2,2)$	$(2,2,1)$
$(1,0,1)$	$(6,1,1)$	$(5,1,1)$	$(2,2,1)$
$(2,1,0)$	$(1,1,2)$	$(2,2,0)$	

$$\exists (0,0) \xrightarrow{\quad} (0,1) d=1$$

$$\exists (0,0) \xrightarrow{\quad} (1,0) d=2$$

$$\exists (0,0) \xrightarrow{\quad} (0,1) d=0$$

$$\exists (0,0) \xrightarrow{\quad} (1,1) d=0$$

$$\exists (1,0) \xrightarrow{\quad} (0,0) d=2$$

$$\exists (1,0) \xrightarrow{\quad} (1,1) d=5$$

$$\exists (2,0) \xrightarrow{\quad} (1,0) d=2$$

$$\exists (2,0) \xrightarrow{\quad} (2,1) d=2$$

$$\exists (2,0) \xrightarrow{\quad} (2,2) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=5$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (0,0) d=2$$

$$\exists (0,1) \xrightarrow{\quad} (0,2) d=0$$

$$\exists (0,1) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (1,1) \xrightarrow{\quad} (2,2) d=3$$

$$\exists (1,1) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,0) \xrightarrow{\quad} (1,0) d=2$$

$$\exists (2,0) \xrightarrow{\quad} (2,1) d=2$$

$$\exists (2,1) \xrightarrow{\quad} (2,2) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=5$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (0,0) d=2$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,2) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,0) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (0,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (1,1) d=6$$

$$\exists (2,2) \xrightarrow{\quad} (2,0) d=6$$

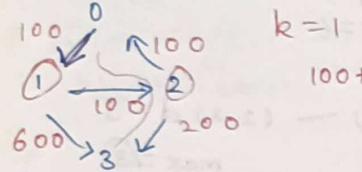
$$\exists (2,2) \xrightarrow{\quad} (2,1) d=6$$

$$\exists (2,2)$$

Q7-38: Cheapest Flights within K stops

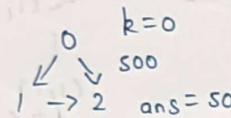
PS - n cities and m edges connected by some no. of flights. You are given an array of flights where $\text{flights}[i] = [\text{from}, \text{to}, \text{price}]$, src, dst and k, return cheapest price from src to dst at most k stops. If no route then return -1.

1/p



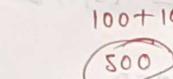
$k=1$

$$100 + 600 = 700$$



$k=0$

$$100 + 100 = 200$$



$k=1$ one
src = 0
dst = 2

$ans = 500$

Intuition:

Since in this, we would find min-cost with K stops so, using Dijkstra's Algorithm.

If we store elements in PQ w.r.t min-distance first, then after a few iterations, our Algo will halt when no. of stops would exceed. Bcz we didn't explore those paths which have more cost but fewer stops than curr.

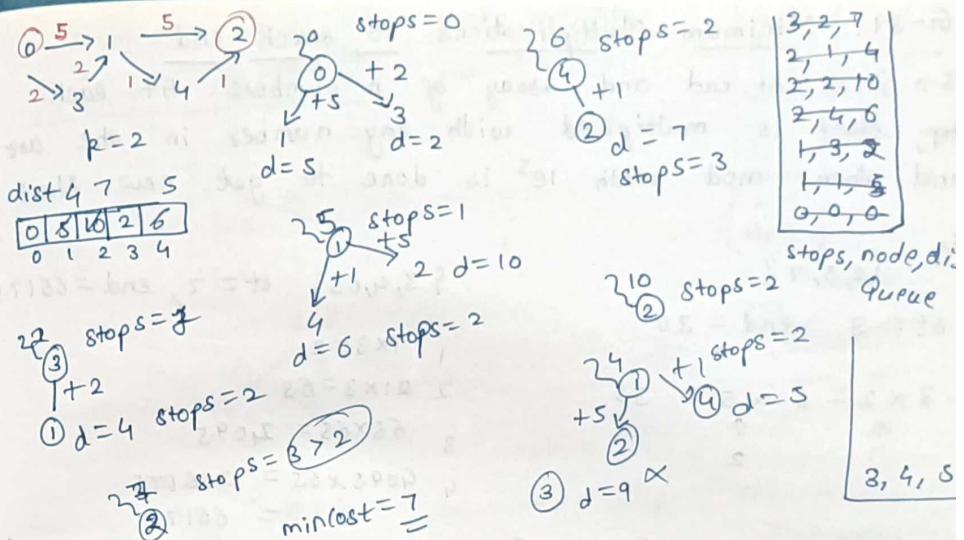
To tackle this, we store in terms of min-stops in PQ. Also, do we need a PQ? NO, stops are monotonically ↑

and when we pop we pop minimal, thus extra $O(\log N)$ of insertion-deletion in PQ will be saved, use a Queue.

Pre-Configuration - 1. Queue { stops, \$node, cost }

2. Distance Array - min cost

3. src & destination



Time Complexity $\rightarrow O(N) \{ \text{addn} \log(N) \}$ bcz we're using a simple Queue rather than a PQ which is usually used in Dijkstra where $N = \text{no. of flights / No. of edges}$

SC $\rightarrow O(|E| + |V|) \{ \text{for adjacency list, PQ, and dist array} \}$
where, E - no. of edges (flights.size()) and V - no. of airports

G-39: Minimum Multiplications to reach end

PS → Given start, end and array of n numbers. At each step, start is multiplied with any number in the arr and then mod with 10^5 is done to get new start.

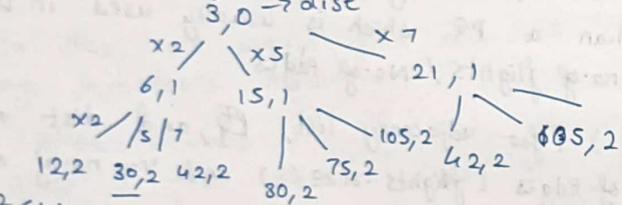
i/p {2, 5, 7}

st = 3 end = 30

$$3 \times 2 = 6 \times 5 = 30$$

st = 3 arr = {2, 5, 7}

$\xrightarrow{3,0} \xrightarrow{x2} \xrightarrow{x5} \xrightarrow{x7}$



$$\text{mod } 100000 - 0, 1, 2, 3, \dots, 99999$$

nodes range always in betn 0 - 99999

We update the distance array, whenever we find a lesser no. of multiplications in order to reach a node, whenever in this way, we reach end, the multiplications needed to reach it would always be minimum.

Queue - {steps, number} at each step, the steps are increasing so no need of PQ a Queue will do.

Distance Array - 0 ... 99999 marked as ∞

Start and End

start = 3 end = 75
 $\xrightarrow{0, 5, 6, 21, 15, 30, 75}$
 $\xrightarrow{1, 11, 11, 11, 21, 2}$

mod = 100000
 $\text{arr} = [2, 5, 7]$

$\xrightarrow{3,0}$
 $\xrightarrow{x2} \xrightarrow{x5} \xrightarrow{x7}$
6,1 15,1 21,1

$\xrightarrow{6,1}$
 $\xrightarrow{x2} \xrightarrow{x5} \xrightarrow{x7}$
12,2 30,2 42,2

$\xrightarrow{15,1}$
 $\xrightarrow{x2} \xrightarrow{x5} \xrightarrow{x7}$
30,2 75,2

node = 75 == end
stop return 2

2,
2,75
2,42
2,30
2,12
1,21
1,15
1,6
0,3

dist, number
Steps
Queue

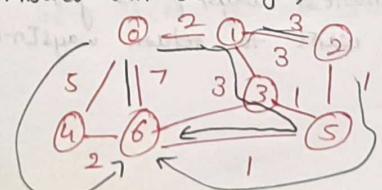
Time Complexity - $O(100000 * N)$ where '100000' are the total possible numbers generated by multiplication (hypothetical) and N = size of the array with no.s of each node could be multiplied.

Space Complexity - $O(100000 * N)$ - size of queue (hypothetical space for dist array is $O(1)$ constant).

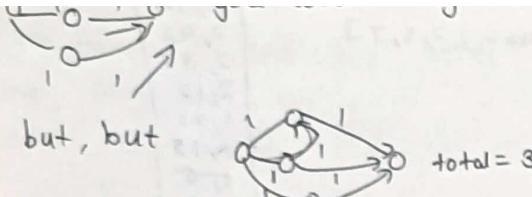
G-40: Number of Ways to arrive at destination

You are in city from 0 to n-1 and bi-directional roads. roads[i] = [u_i, v_i, time_i] means that there is road between intersections u_i and v_i that takes time_i to travel. You need to find in how many ways you can travel from intersection 0 to n-1 in shortest amt. of time.

Answer can be large, return it modulo $10^9 + 7$.

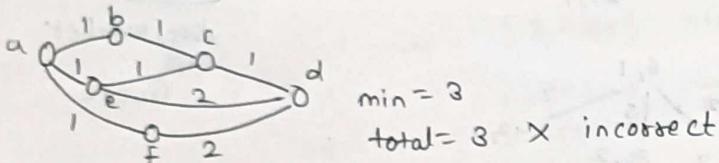


$0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 6$
 $2 \quad 3 \quad 1 \quad 1$
 $2 \quad 3 \quad 3 = 8$
 $0 \rightarrow 6 \quad 0 \rightarrow 4 \rightarrow 6$



but, but

total = 3



min = 3

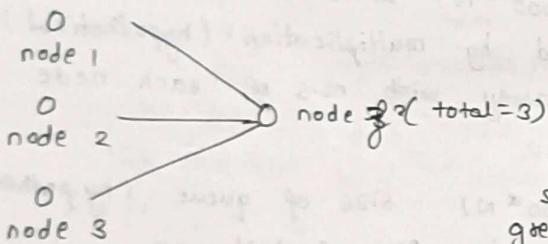
total = 3 X incorrect

$a \rightarrow b \rightarrow c \rightarrow d$ (3)

$a \rightarrow e \rightarrow c \rightarrow d$ (3) total = 4 ways

$a \rightarrow e \rightarrow d$ (3)

$a \rightarrow f \rightarrow d$ (3)



$$\text{waysInode} j = \text{ways}(n_1) + \text{ways}(n_2) + \text{ways}(n_3)$$

and sum of these three shortest paths can be hence greater than 3.

So, we apply Dijkstra and where we count all the possible shortest paths from the source to the destination node.

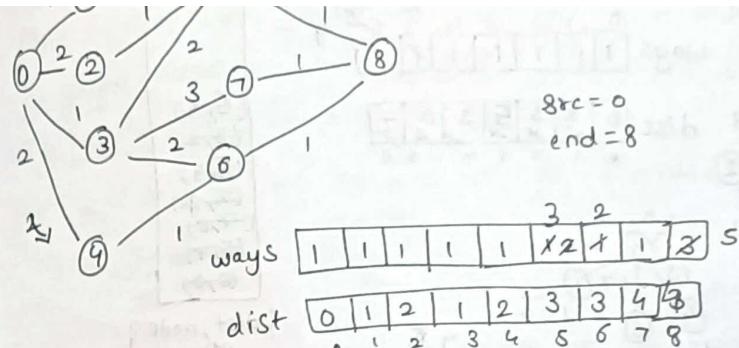
Initial Configuration:

Priority Queue - {dist, node ?}

Distance Array - minimum distance from start node to current node.

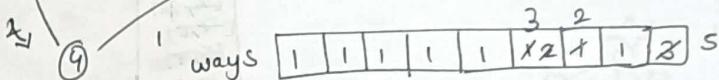
Src Node and End Node

Ways Array - contains possible shortest ways/paths for each node. Eventually, we would want to return ways[n-1] where, n = Number of nodes



src = 0

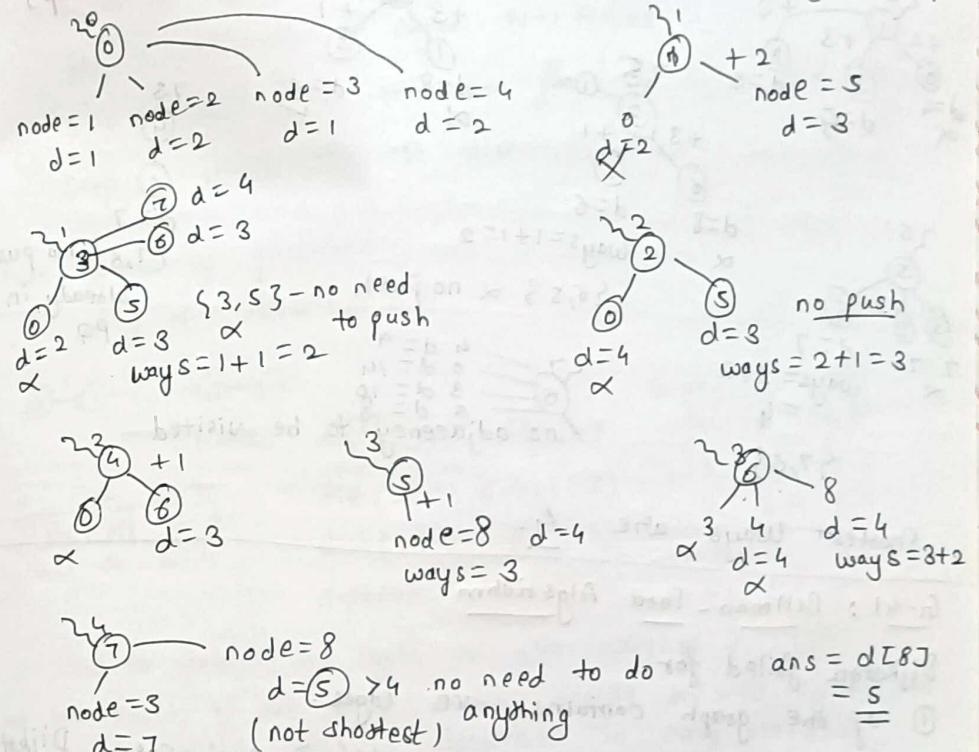
end = 8



dist

(4, 7)
(5, 8)
(3, 6)
(3, 5)
(2, 4)
(1, 3)
(2, 2)
(1, 1)
(0, 0)

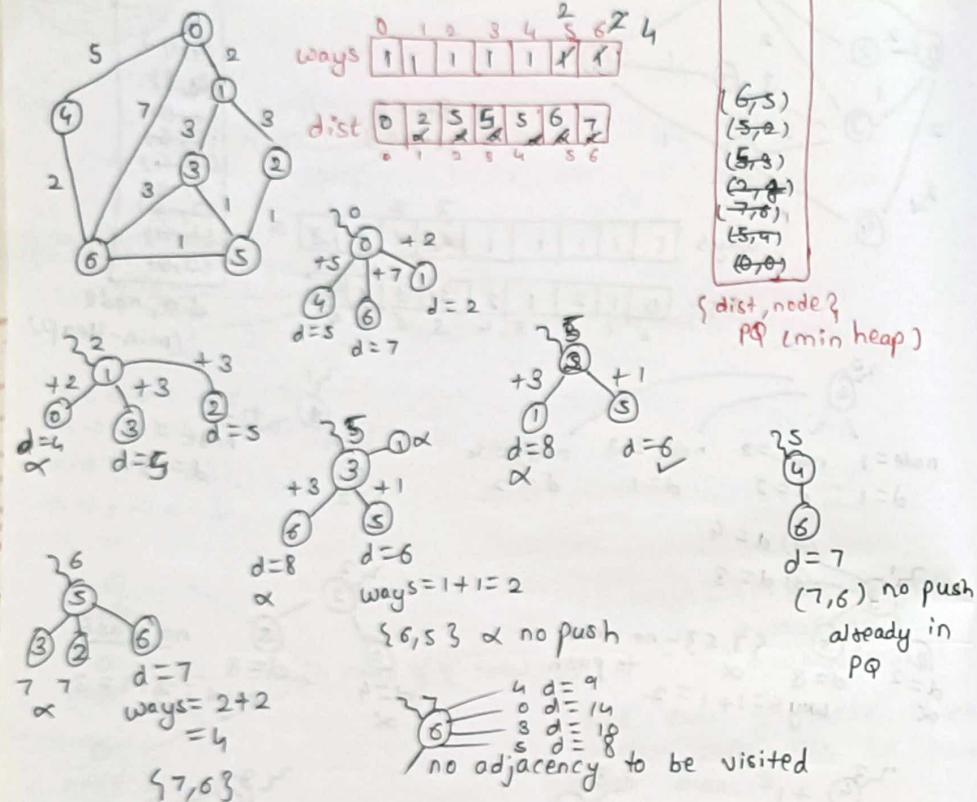
dist, node
(min-Heap)



TC $\rightarrow O(C \log V)$ { As we are using simple Dijkstra }

SC $\rightarrow O(N)$ & for dist array + ways array + approx. complexity for PQ }

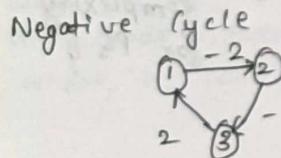
one more day run -



Gr-41 : Bellman-Ford Algorithm

Dijkstra failed for

- ① If the graph contains -ve edges.
- ② If the graph has a negative cycle. In such case, Dijkstra fails to minimize the distance, keeps on running, and goes into an infinite loop. As a result, gives TLE.



$$\text{Sum: } 2 + (-2) + (-1) = -1$$

Bellman Ford successfully solves these problems. It works fine with -ve edges as well as it is able to detect if graph contains a -ve cycle. But this algorithm is only applicable for directed graphs.

for undirected we convert it to directed



Algorithm - Edges can be given in any order

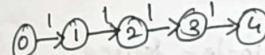
Step 1. Relaxation of edges N-1 times

Step 2. Nth time relaxation for -ve cycle and return f13.

Step 3. return shortest distances

Two follow-up questions

1. Why N-1 iterations?



checks

dist[3]+1 < dist[4]

dist[2]+1 < dist[3]

dist[1]+1 < dist[2]

dist[0]+1 < dist[1]

Given

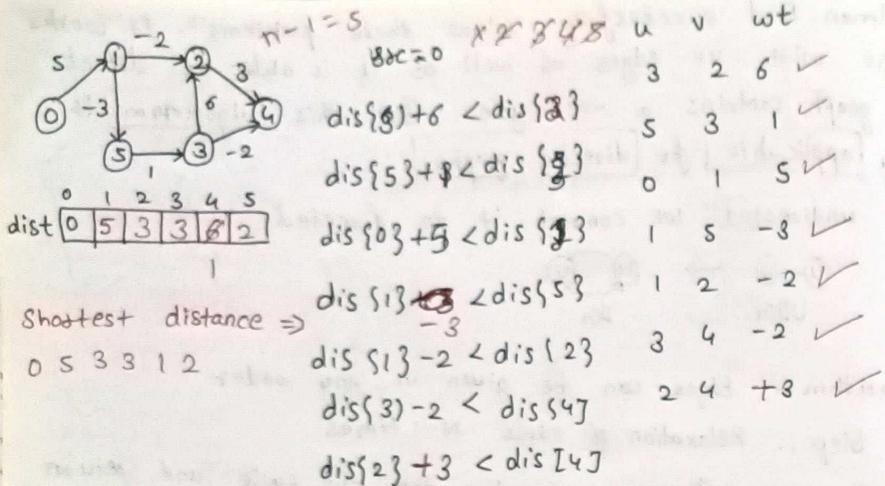
u	v	wt
3	4	1
2	3	1
1	2	1
0	1	1

2. How to detect -ve cycle or -ve weights?

→ we know, if we keep rotating inside a -ve cycle, the path weight will be decreased in every iteration.

→ so after N-1, we check for Nth relax, if found that further relaxation is possible then we conclude that DG has -ve cycle.

Initial Config - distance array - $\text{dist}[src] = 0$



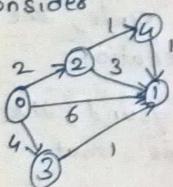
TC $\rightarrow O(V \times E)$ where, V = no. of vertices & E = no. of edges

SC $\rightarrow O(V)$ for the distance array which stores the minimized distances.

G-42: Floyd Warshall Algorithm

Dijkstra's and Bellman Ford algo. are single-source shortest-path algorithms. But in Floyd Warshall Algorithm we need to figure out the shortest distance from each node to every other node.

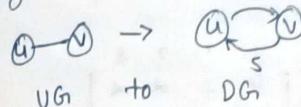
Consider



for pair {0, 1}
To find shortest distances,
 $(0 \rightarrow 1) = 6$
 $(0 \rightarrow 2) + (2 \rightarrow 1) = 5$
 $(0 \rightarrow 3) + (3 \rightarrow 1) = 5$
 $(0 \rightarrow 4) + (4 \rightarrow 1) = 4$
 $(0 \rightarrow 2) + (2 \rightarrow 4) = 3$ (path from $0 \rightarrow 4$)

Thus, we can derive the following formula
 $\text{matrix}[i][j] = \min(\text{matrix}[i][j], \text{matrix}[i][k] + \text{matrix}[k][j])$
where i \rightarrow src node
j \rightarrow dest. node
k \rightarrow node via which we are reaching from i to j

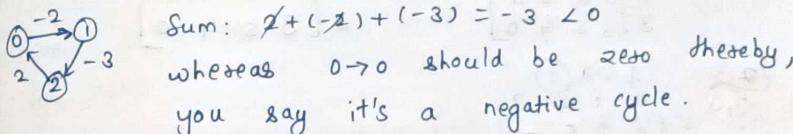
To apply this algo to UG



- * using adjacency matrix for storing
- * $\text{dist}[i][i] = 0$

Follow-up questions

Negative cycle:



What will happen if we will apply Dijkstra's algorithm for this purpose?

- If the graph has a negative cycle: We can't apply Dijkstra's algorithm, it leads to TLE.
- If the graph doesn't contain a negative cycle: We apply Dijkstra for every possible node. TC $\rightarrow O(VE \log V)$

for checking -ve

for (int i=0 ... n
j=0 ... n

if ($\text{matrix}[i][j] < 0$)
"negative"

TC $\rightarrow O(n^3)$... three nested loops
SC $\rightarrow O(n^2)$... storing adjacency matrix.

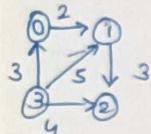
Pseudo Code -

```
for( via = 0...n )
    for( row < n )
        for( col < n )
```

$\rightarrow O(n^3)$

(Brute Force)

$$\text{matrix}[row][col] \\ = \min(m[row][col], m[row][via] + m[via][col])$$



$$\begin{array}{c} 0 \xrightarrow{2} 1 \\ 3 \xleftarrow{5} 0 \\ 3 \xrightarrow{4} 2 \\ 2 \xleftarrow{\alpha} 3 \end{array} \quad \begin{array}{c} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & \alpha \\ 1 & \alpha & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & \alpha \\ 3 & 3 & 5 & 4 & 0 \end{array} \xrightarrow{\text{via}} \begin{array}{c} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & \alpha & \alpha \\ 1 & \alpha & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & 2 \\ 3 & 3 & 5 & 4 & 0 \end{array}$$

$$\begin{array}{c} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & 3 \\ 1 & \alpha & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & \alpha \\ 3 & 3 & 5 & 4 & 0 \end{array} \xrightarrow{\text{via } 1} \begin{array}{c} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & \alpha \\ 1 & \alpha & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & \alpha \\ 3 & 3 & 5 & 4 & 0 \end{array} \xrightarrow{\text{via } 2} \begin{array}{c} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & 5 & \alpha \\ 1 & \alpha & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & \alpha \\ 3 & 3 & 5 & 4 & 0 \end{array} \xrightarrow{\text{via } 3} \begin{array}{c} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & 5 & \alpha \\ 1 & \alpha & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & \alpha \\ 3 & 3 & 5 & 4 & 0 \end{array}$$

$$\begin{array}{c} 1,0 \\ 2,3 \\ 3,0 \\ 1,2 \\ 2,3 \\ 3,1 \\ 1,3 \\ + 3,2 \end{array} \quad \begin{array}{c} 0 & 1 & 2 & 3 \\ 0 & 2 & 5 & \alpha \\ 1 & \alpha & 0 & 3 & \alpha \\ 2 & \alpha & \alpha & 0 & \alpha \\ 3 & 3 & 5 & 4 & 0 \end{array} \quad \checkmark$$

G-43: City with the smallest number of neighbours at a Threshold Distance

You need to find out a city with smallest no. of cities that are reachable through some path and whose distance is at most threshold distance. If there are multiple cities, our answer will be city with greatest number.

$$\begin{array}{c} 0 & 1 & 2 & 3 \\ 0 \rightarrow 0 & 3 & 4 & 5 \rightarrow \text{cnt} = 3 \\ 1 \rightarrow 0 & 1 & 2 \rightarrow 4 & \text{ans} = \underline{\underline{3}} \\ 2 \rightarrow 4 & 1 & 0 & 1 \rightarrow 4 \\ 3 \rightarrow 5 & 2 & 1 & 0 \rightarrow 3 \end{array}$$

threshold

$$\begin{array}{c} 0 & 1 & 2 \\ 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \rightarrow 2 \\ 2 & 2 & 0 & 1 \rightarrow 2 \end{array} \quad \begin{array}{l} 0 \rightarrow 0 \\ 1 \rightarrow 3 \\ 2 \rightarrow 0 \end{array} \quad \text{ans} = \underline{\underline{2}}$$

We know Floyd Warshall helps us to generate a 2D matrix, that stores the shortest distances from each node to every other node. In the generated 2D matrix, each cell $m[i][j]$ represents the shortest distance from node i to node j .

Intuition - For each node, the job is to find the shortest distances to every other node & count no. of adjacent cities, say: cntCity - whose dist. is at most the threshold. Finally, the task is to choose the node with largest value and minCity value.

Pre config - adjacency matrix [adj] - all set to ∞ (n*n)

cntCity - initially set to V - max. nodes possible

cityNo - set to -1. Stores node with largest value and minimum cntCity value.

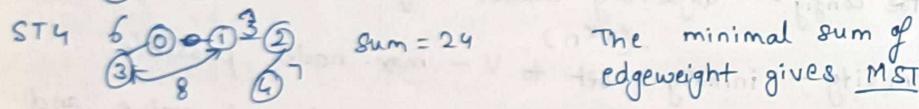
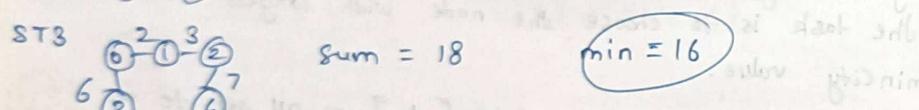
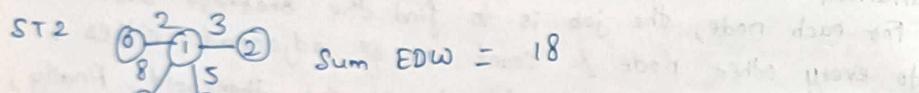
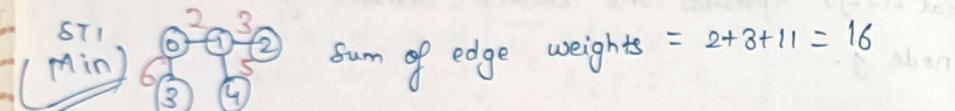
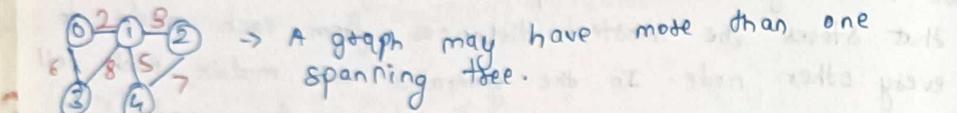
If we need to find out the no. of adjacent cities as well we need to return cntCity-1. This is bcz we have included the node itself. $0 \rightarrow \overset{\uparrow}{0} 1 2 3$ itself which will be always 0 and < threshold

TC $\rightarrow O(V^3)$ - as we have three nested loops, each running for V times.

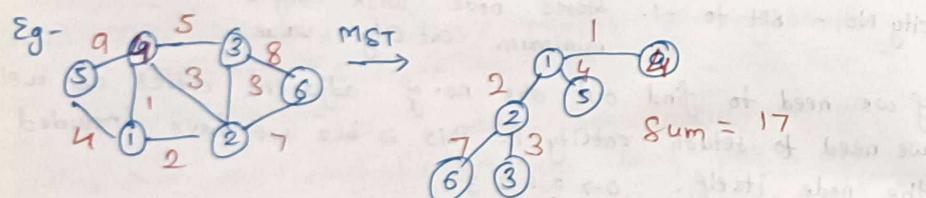
SC $\rightarrow O(V^2)$ - where V - no. of vertices, adjacency matrix

G-44: Minimum Spanning Tree (MST)

A spanning tree is a tree in which we have N nodes (i.e. All the nodes present in the original graph) and N-1 edges and all nodes are reachable from each other.



The minimal sum of edgeweights gives MST



(n-45): Prim's Algorithm → MST

Initial Config: PQ (min-heap) {dist, node}

Visited array

MST (pairs) if we need MST edges

Sum variable

Take any node and push in PQ eg. {0, 0}

Intuition: Greedy Technique

for every node, we select greedily its unvisited adjacent node with minimum edge weights. Doing so for every node, we get sum of all the edge weights of minimum spanning tree and spanning tree itself.

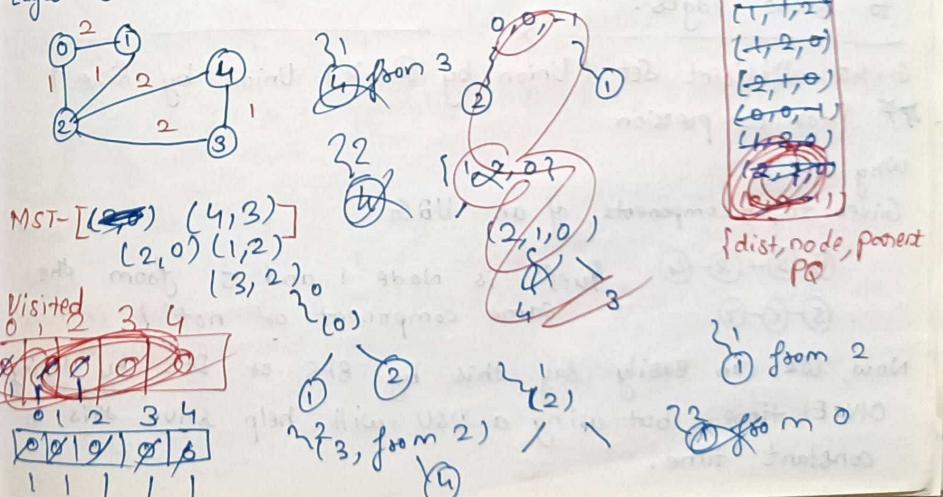
Note: If we only need the sum

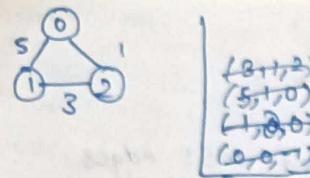
then
→ we need not use triplet of {dis, {node, parent}}
we will just use pair in the format {edw, node}

Basically, we don't need parent node.
→ we need not use MST array in whole algo as well.

$$\text{sum} = 8 \times 6 \times 2 \times 5$$

Edges - 5





vis
 $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$

MST
 $[1,2,0][1,2]$

2^0 of from -1
 $6 \quad 0$

sum = $0 + 4$

{
 1
 2 from 0

{
 1
 3 from 2

{
 1 from 0
(already visited)

$$TC \rightarrow O(E \log E) + O(E \log E) \sim O(E \log E)$$

The max. size of PQ can be E at most E iterations
the pop takes $\log E$... this gives $(E \log E)$. Inside
that for every node, we need to traverse all its
adjacent nodes where no. of nodes can be E and
for push its $\log E \sim (E \log E)$

$$SC \rightarrow O(E) + O(V) \sim E\text{-edges}, V\text{-nodes/vertices}$$

$O(E)$ for PQ \rightarrow for visited array
 $O(V)$ for PQ \rightarrow for visited array

If we need to get mst, then $O(V-1) \sim O(V)$
to store edges.

G-45: Disjoint Set / Union by Rank / Union by Size

Path Compression

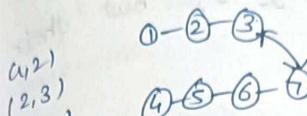
Why DSU?

Given two components of an UDG

$1-2-3-4$ guess is Node 1 and 5 from the
same component or not?

$5-6-7$
Now, we can easily say this by BFS or DFS by taking
 $O(V+E)$ time, but using a DSU will help solve this in

DSU is generally used for dynamic graphs. (run-time)



$(1,2)$
 $(2,3)$
 $(4,5)$
 $(6,7)$
 $(5,6)$
 $(3,7)$

here we conclude 1 & 7 are from same component
At any point, if we try to check u & v are from same
component, DSU will be able to answer this in constant
time.

Functionalities of DSU

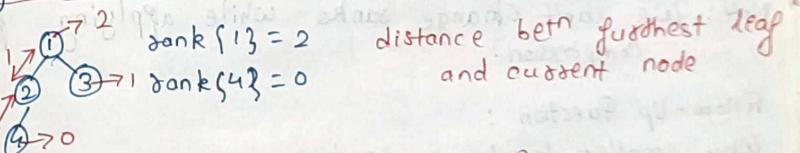
1. find parent for particular node
2. Union() in broad this method basically adds
an edge betn 2 nodes)

↳ By Rank

↳ By size

Union by rank:

Rank:



distance b/w furthest leaf
and current node

Ultimate parent (topmost node or boss)

In this $\text{parent}[6] = 5$ but, $\text{ultimate parent}[6] = 4$.

Algo: Pre-Config : rank array (initially all 0)
parent array (initially $\text{parent}[i] = i$)

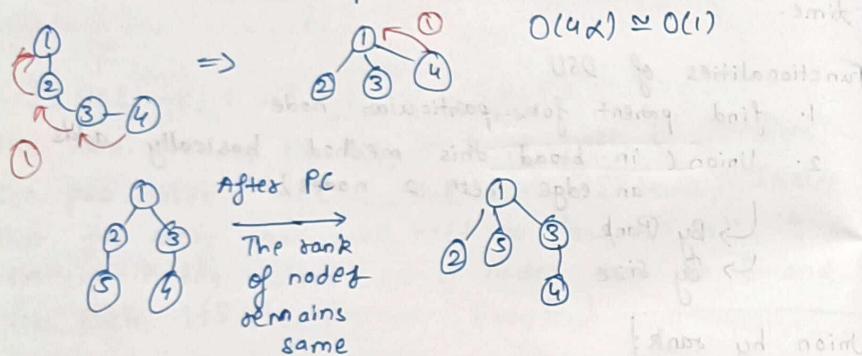
Steps
1. for two nodes (u and v) then, find the ultimate
parent p_u and p_v .

2. After that, we find rank of p_u and p_v .
3. We connect smaller rank to bigger rank.

Observation 1: Why we need the ultimate parents.
So, 5 & 7 have parent as 3 & 6, but their ultimate parent is 1.

thus, we use `findPar()` which find ULTIMATE parent for a node.

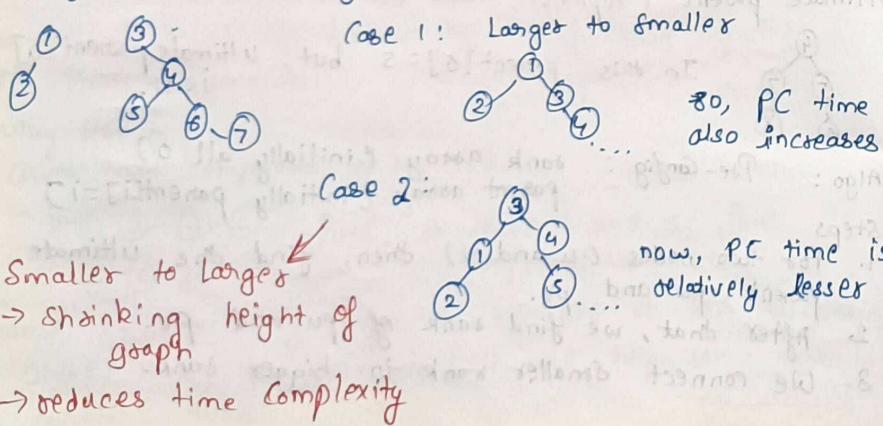
Observation 2: Ultimate parent takes up $O(\alpha \lg N)$. So for $O(1)$ we use Path compression.



Note: We can't change ranks while applying path compression.

Follow-Up Question:

Why smaller to larger rank?



Observation 3: After Path compression, the ranks of the graph becomes distorted. So, we store size of components.

Union by size

size array: initialized to one

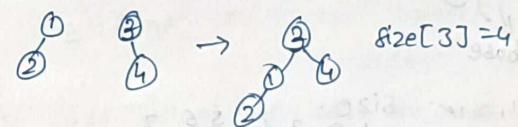
parent array: $\text{parent}[i] = i$

steps → 1. p_u, p_v (ultimate)

2. $\text{size}[p_u]$ and $\text{size}[p_v]$

3. finally we connect ultimate parent with smaller size to larger size. But if equal we connect any to any.

Meanwhile we also increase the size where it's getting attached



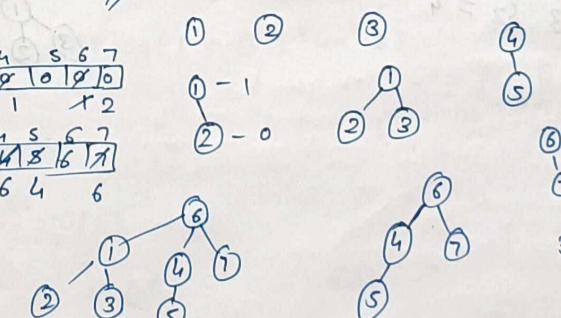
Note: 1. seems much more intuitive than UBR as rank gets distorted after PC

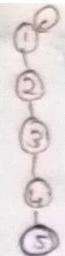
2. `findPar()` remains same

$T_C \rightarrow O(4\alpha) \approx 1, O(1) \approx 0, 4\alpha \approx 1$

Constant Time $O(1)$

	rank	1	2	3	4	5	6	7
	parent	1	2	3	4	5	6	7
(1,2)✓		1	0	0	0	0	0	0
(2,3)✓		1	1	0	0	0	0	0
(4,5)✓		1	2	3	4	5	6	7
(6,7)✓		1	2	3	4	5	7	7
(5,6)✓		1	2	3	4	6	6	7
(3,7)✓		1	2	3	4	6	6	7



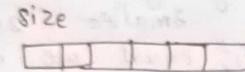
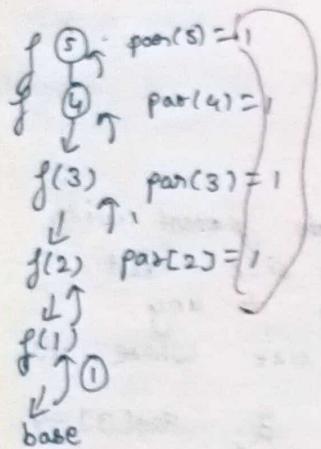


Path Compression

findPar(u)

```
if( u == par[u] )
    return u;
```

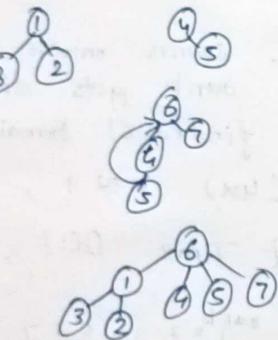
```
return par[u]= findPar(parent[u])
```



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

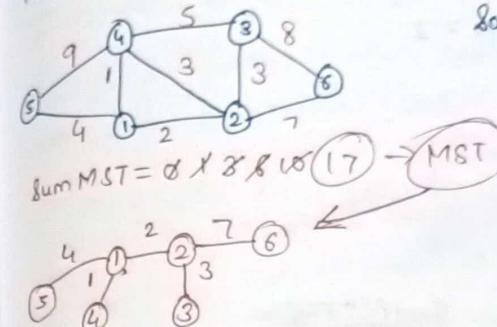
find(3) find(7)

① ⑥
size=3 s2=4



Kruskal's Algorithm - MST

Given weighted, undirected and connected graph of V vertices, task is to find the sum of weights of MST.



Sort all edges according to the weight: (wt, u, v)

- 1, 1, 4 ✓
- 2, 1, 2 ✓
- 3, 2, 3 ✓
- 3, 2, 4 ✗ (same comp.)
- 4, 1, 5 ✓
- 5, 3, 4 ✗
- 7, 2, 6 ✓
- 8, 3, 6 ✗
- 9, 4, 5 ✗

→ Iterate over sorted edges by weight

→ If ultimate parents are same, don't do anything

→ If ULP different, then add weight of edge to our final answer and apply union for path compression and adding edge betn u and v.

Note: As graph contain bidirectional edges, we can get a single edge twice in our array (wt, u, v) and (wt, v, u) but D&U will automatically discard the duplicate

TC $\rightarrow O(N+E) + O(E \log E) + O(E^2 \times 2)$

{for extracting edge from adjacency list} {Sorting array acc. to weight} {loop continues for E times, two operations Path Union() each takes $O(4 \times 2) \approx O(4 \times 2)$ }

SC $\rightarrow O(N) + O(N) + O(E)$

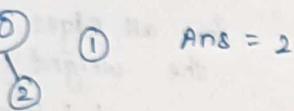
{parent and size} {Storing edges}

$O(4 \times 2) \approx O(4 \times 2) \approx O(E^2)$

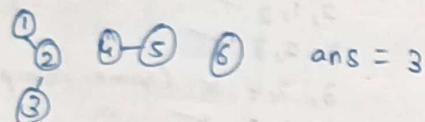
Number of Provinces

Province is a group of directly or indirectly connected cities and no other cities outside of the group.

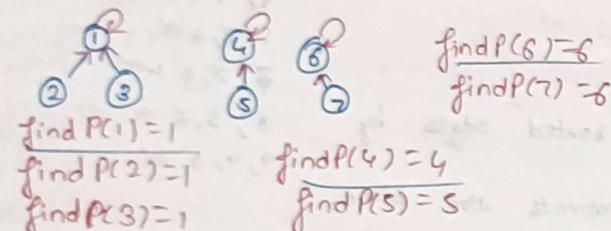
0	1	2
1	0	1
2	0	1



Ans = 2



ans = 3



Number of bosses is the number of provinces.
(Ultimate Parent(node) == node)

→ Apply DSU on edges

→ Count bosses and return

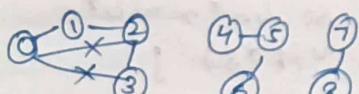
$$TC \rightarrow O(V^2 * 4\alpha) + O(4 * 4\alpha) \sim O(V^2)$$

(DSU) (counting bosses)

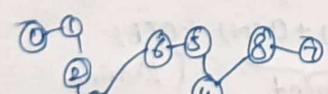
$$SC \rightarrow O(2N)$$

(parent, size?)

G-4a: Number of Operations to make network connected



minimum = 2 but extra edges from the graph not imaginary



$$\square \square \square \rightarrow \square - \square - \square - \square$$

So, we can say if a graph has nC connected components then we need a min. of $nC-1$ edges.

→ for single, 0 edges

thus, If we have $nC-1$ extra edges (atleast) then we can connect with $nC-1$ operations.

→ Apply DSU on edges dynamically as well as check for extra edges

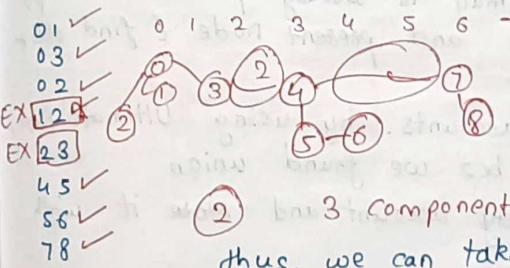
if (Parent(u) != Parent(v))
extra++
else union them (u,v)

→ Count Connected Components

(parent(i) == i) nC++;

→ check extra $\geq nC-1$? $nC-1 : -1$

If unable to have minimum extra edges then it's not possible to connect thus, we return -1.



(2) 3 components, 2 edges

thus, we can take out those edges

$$TC \rightarrow O(E * 4\alpha) + O(N * 4\alpha)$$

{Union and extra edges} {for counting number of components}

$$SC \rightarrow O(2N) \text{ where, } N = \text{no. of nodes}$$

(rank and parent array)

G-50: Accounts Merge - DSU (Merging details)

Two accounts belong to the same person if there is some common email to both a/cs. Merge them, first name then emails in the sorted order.

$\left[\text{[John, J1@com, J2@com, J3@com]}, \right]$
 $\left[\text{[John, J4@com]}, \right]$
 $\left[\text{[Bau, Bau1@com, Bau2@com]}, \right]$
 $\left[\text{[John, J1@com]}, \right]$
 $\left[\text{[Bau, Bau2@com]}, \right]$
 $\left[\text{[Mary, Mary1@com]}, \right]$

$\rightarrow \text{John} - \text{J1, J2, J3, J4}$

$\text{John} - \text{J4}, \text{Bau} - \text{Bau1, Bau2}$ between them
 $\text{Mary} - \text{Mary1}$

Algo-

- Create a map for storing each email with index (node) they belong to
- While doing so, if an email is already present in map, then merge that node and present node (find their union)
- After this, merge the accounts. by using Union-Find method for an email bcz we found union
- Sort the emails for every account and store it with names.

John, J1@com, J2@com, J3@com 0

$J1 \rightarrow 0 \quad m_1 \rightarrow 3$

John, J4@com 1

$J2 \rightarrow 0$

Bau, Bau1@com, Bau2@com, Bau3@com 2

$J3 \rightarrow 0$

Mary, Mary1@com 3

$J4 \rightarrow 1$

Bau, Bau2@com, Bau3@com 4

$b1 \rightarrow 2$

John, J2@com 5

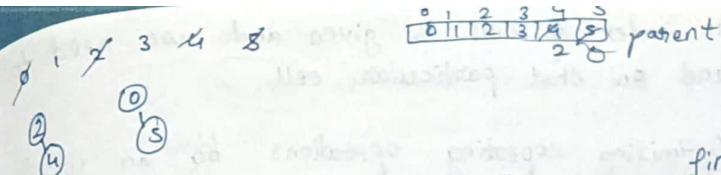
$b2 \rightarrow 2$

Bau, Bau3@com 6

$b3 \rightarrow 2$

John, J1@com 7

$b6 \rightarrow 4$



$0 \rightarrow [\text{J1@com, J2@com, J3@com}]$

$1 \rightarrow [\text{J4@com}]$

$2 \rightarrow [\text{bau1@com, bau2, bau3, bau6}]$

$3 \rightarrow [\text{em1}]$

$4 \rightarrow$

$5 \rightarrow$ visiting merging storing

$T.C \rightarrow O(N+E) + O(E*4x) + O(N*(ElogE+E))$ where, N - no. of indices, E - no. of emails

$S.C \rightarrow O(N) + O(N) + O(2N) \approx O(N)$ where, N = no. of nodes
 parent size ans & merged Accounts

G-51: Number of Islands - II - Online Queries

Return how many islands after each operation.

An island means group of 1s such that they share a common side.

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

$[1,1] \rightarrow 1$
 $[0,1] \rightarrow 1$
 $[3,3] \rightarrow 2$
 $[3,4] \rightarrow 2$

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

$(0,0) \rightarrow 1$
 $(0,0) \rightarrow 1$
 $(1,1) \rightarrow 2$
 $(1,0) \rightarrow 2$
 $(0,1) \rightarrow 1$
 $(0,3) \rightarrow 2$
 $(1,3) \rightarrow 2$

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

$(0,0) \rightarrow 1$
 $(3,2) \rightarrow 3$
 $(2,2) \rightarrow 8$
 $(1,2) \rightarrow 12$
 $(0,2) \rightarrow 1$

Observeⁿ 1: An index of cell is given and we need to add an island on that particular cell

Observeⁿ 2: Optimizing repeating operations by an visited array

Observeⁿ 3: How to connect cells?

We number from 0 to $n \times m - 1$

$s \times 3$	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14

If we want to connect (2,0) and (1,0), we just need to perform a union (10,5)

For component from (row, col)
 \downarrow
 $\boxed{\text{row} * m + \text{col}}$
 \downarrow
 Total no. of columns

$$\text{eg. } (2,1) \\ 2 \times 5 + 1 \\ = 11$$

Observeⁿ 4: How to count No. of Islands?

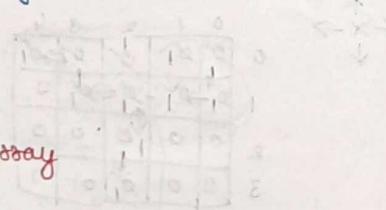
\leftrightarrow If cell is unvisited, we make it and increase the counter by 1.

If adjacent sides are island, then we check if they are connected so, decrease cnt by 1 and do a union. for that we convert the (row, ncol) to adj component by above formula.

TC $\rightarrow O(Q * 4\alpha)$ where, Q - no. of Queries

SC $\rightarrow O(Q) + O(N \times M) + O(N \times M)$

\downarrow
 answer
 for parent and
 size/ rank array



Q-52: Making a Large Island - DSU

Binary matrix, change at most one zero to 1. Find the largest group of connected 1's.

\rightarrow dynamically (real time) graph - DSU

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

$$4+4+1 = 9 \\ 1+10 = 11 \\ \left. \begin{array}{l} \end{array} \right\} \max(9, 11) = 11$$

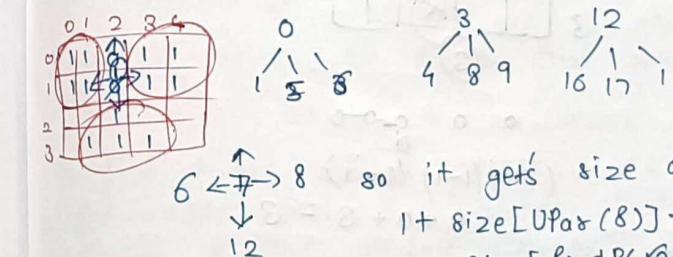
Observeⁿ 1: How to connect them in same group?

0	1	2	3	4
0	1	2	3	4
1	0	1	2	3
1	5	6	7	8
2	10	11	12	13
3	14	15	16	17

(row, col) \rightarrow number of Node

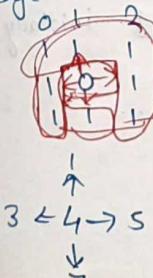
(row * columns + col), because DSU forms a union of single node not a (i,j) pair

To put 0 and get sizes of its adjacent group of Islands



$$6 \xleftarrow{\uparrow} 8 \xrightarrow{\downarrow} 12 \quad \text{so it gets size of ultimate parent} \\ 1 + \text{size}[Upar(8)] + \text{size}[findP(12)] \\ + \text{size}[findP(12)] = 1 + 4 + 4 + 4 \\ = 13$$

But there's an edge case



$$3 \xleftarrow{\uparrow} 4 \xrightarrow{\downarrow} 5$$

Here, 0 will get $1 + 8 + 8 + 8$ whereas it should add only 8 because they all from same component. the base is same, so here use a set data structure.

$$\boxed{1} \quad p(8) \rightarrow 0 \quad p(8) \rightarrow 0 \quad \text{so we get} \\ p(7) \rightarrow 0 \quad p(1) \rightarrow 0 \quad 1 + R = 9 \\ \boxed{1} \quad \text{+ size}[0]$$

Also, what if we are not able to place a zero

Eg Here, we expect answer as 4.

So, finally we check each node's ultimate Parent size.
and maximum is opted.

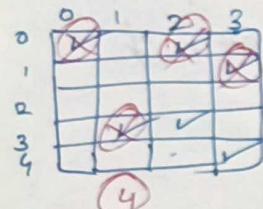
$$TC \rightarrow O(n^2 \times 4) + O(n^2 \times 4 \times 4) \approx O(n^2)$$

$$SC \rightarrow O(2n^2) \dots \text{parent and size}$$

G-53: Most Stones Removed with same row or column

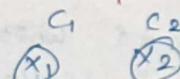
Given 2D plane, remove some stones

A stone can be removed if it either shares same row or column with a stone that hasn't been removed.



$$\begin{aligned} & (4-1) + (2-1) \\ & (3+1) \\ & = 4 \end{aligned}$$

So, for components



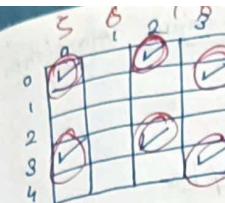
x_1, x_2, x_3, x_4 stones remained initially

$$(x_1-1) + (x_2-1) + (x_3-1) + (x_4-1) \dots$$

$$= (x_1+x_2+x_3+\dots) - (1+1+1+\dots)$$

$$= (n - k)$$

$$\text{Ans} = \text{stones} - \text{no. of components}$$



(0,1)
(0,1)

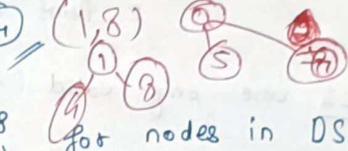
$$8\text{stones} = 6$$

$$nC = 2$$

$$6-2 = 4$$

(0,5)

(0,7)



for nodes in DS

rows

cols

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

⑥ ③ ② ⑥
1 7 3 8 4
parent[i][j] == i && size > 1

How to connect \rightarrow

If two stones are at [0,0] and [0,2] of first row, then bcz of 0-based indexing as a node we convert column to (0+s) and (2+s) and take union of (0,5) and (0,7)

Thus, we connect all stones that are in same row or same column to form different components.

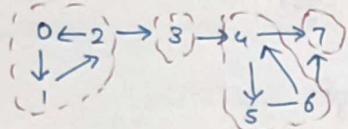
To find Unique ultimate parent, we will store the nodes that had stones in a map and check if that node's ultimate parent is equal to node and thus increment our component counter by 1.

TC $\rightarrow O(N)$ where, N - Total no. of stones

SC $\rightarrow O(2 * (\text{maxRowIndex} + \text{maxColIndex}))$ for parent and size array.

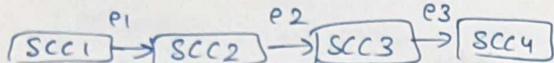
6-54: Strongly Connected Components - Kosaraju's Algorithm
 Two Questions - 1. No. of SCs
 2. Print SCs

SCCs are only valid for directed graphs

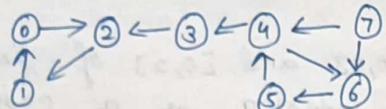


A component is a SCC if for every pair (u, v) u is reachable from v and vice-versa.

By defn, single vertex is also a SCC.



What If we reverse the edges,

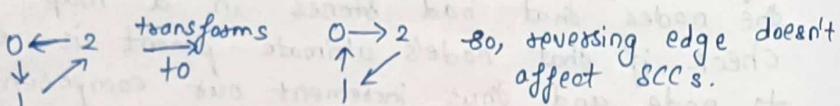


Now, if we do a dfs from node 0 we will only visit nodes of SCC1. Similarly for node 3 then node 4 and node 7. So, eventually, no. of DFS calls is our no. of SCs.

But we don't know SCs, so we reverse all edges.



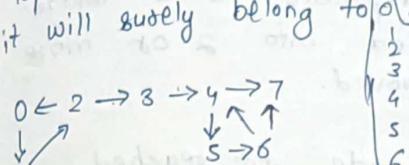
But



Algorithm -

- (1) Sort all the nodes acc. to finishing time
- (2) Reverse edges of entire graph
- (3) Perform DFS and count no. of DFS calls to get the no. of SCs

Note: Sorting of nodes acc. to finishing time is very important. By performing this step, we get to know where we should start our DFS calls. The top-most element of the stack will finish last and it will surely belong to SCC1.



dfs(0) ↗

dfs(1) ↗

dfs(2) ↗

dfs(3) ↗

dfs(4) ↗

dfs(5) ↗

dfs(6) ↗

dfs(7) ↗

0 → {1, 2}

1 → {2, 3}

2 → {0, 3}

3 → {4}

4 → {5, 7}

5 → {6}

6 → {4, 7}

7 → { }

O(Degree)

= O(Edges)

(first to finish)

Stack ↘

dfs(0)

dfs(2)

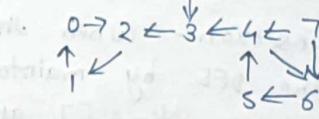
dfs(1)

dfs(4)

dfs(6)

dfs(5)

4 (visited)



dfs(3) ↗

dfs(7) ↗

dfs(1) ↗

dfs(4) ↗

dfs(6) ↗

dfs(5) ↗

4 (visited)

SCC = \emptyset

2
3
4

we can even store nodes of a SCC during dfs and print.

$$TC \rightarrow O(N+E) + O(V+E) + O(V+E) \sim O(N+E)$$

↓
Simple
DFS
(Sorting)
reversing
edges
final
DFS

$$SC \rightarrow O(V) + O(V) + O(V+E)$$

↓
Stack
Visited
array
deversed
adj. List

G-55: Bridges in Graph - Using Tarjan's Algorithm of time in and low time (LeetCode)

(Critical Connections in a Network)

Bridge - Any edge in the component is a bridge if the components get divided into 2 or more parts if the edge is removed.

Time of Insertion - when the node got reached during the DFS by maintaining the timer, to store we use time[] array.

Lowest Time of Insertion - minimum lowest time from all the adjacent nodes except the parent.
→ stored using low[] array. ↗ bcz we want to check if any other path to the node exists via the parent

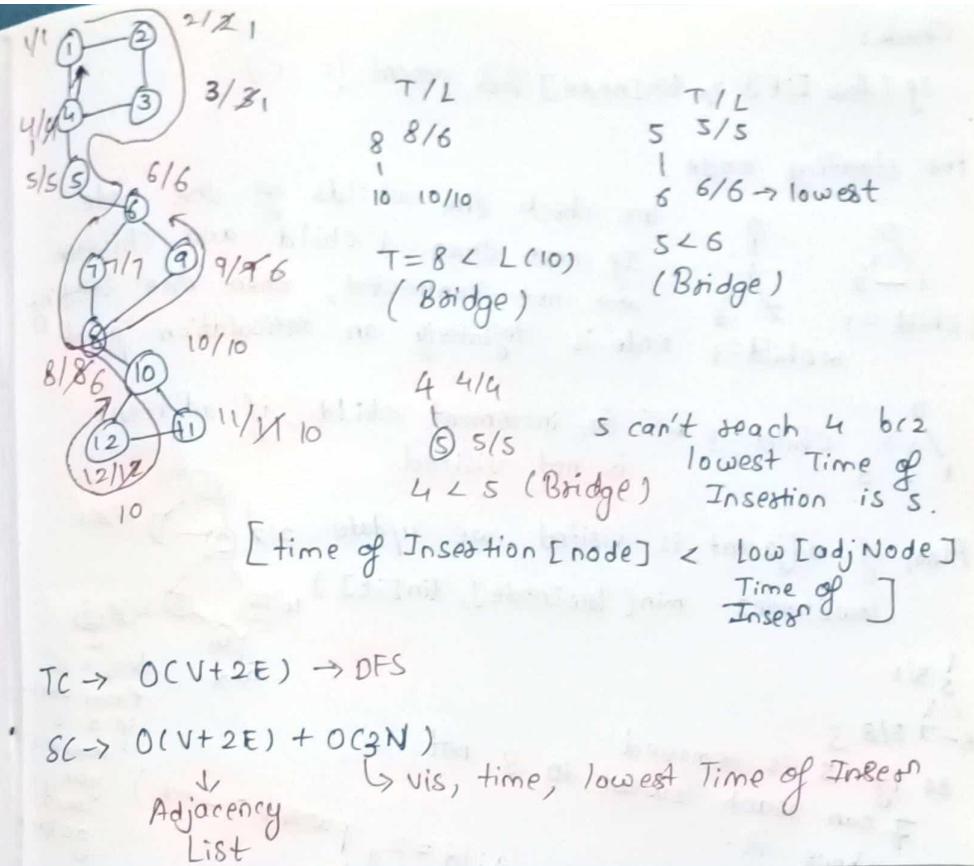
Logical Modification of DFS

Node $\xrightarrow{\text{adj}}$
 \downarrow
adjacent nodes

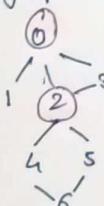
If there is any other path from current node to its adjacent then it's not a bridge, else it's the valid bridge (critical connection)

Algo-

```
DFS() {
    vis[node] = 1
    low[node] = time[node] = timer++
    for adj: node
        if (adj == node's parent) continue
        if (not visited[adj])
            dfs(adj, node) ...
            take minimum low time for node, after DFS
            lowestTime[adj] > time[node]
            store it's a bridge
        else
            take low[node] = min(low[node], low[adjNode])
}
```



Articulation Points - Nodes on whose removal, the graph breaks into multiple components.



(-) Time of Insetion

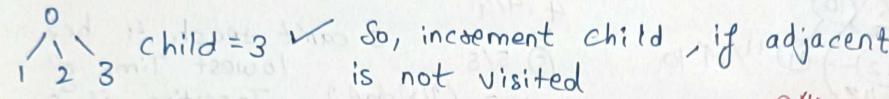
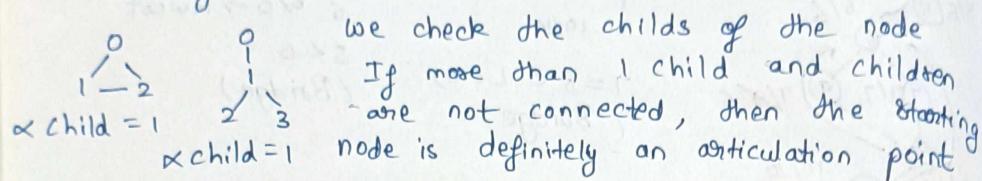
(-) Lowest Time of Insetion: min possible from the adjacent nodes excluding parent and visited nodes

Here, we are trying to remove curr. node along with edges linked to it. For that reason, here we will check if there exists any path from adjacent to previous node of curr. node. Also, the node can't be a starting node.

Check:

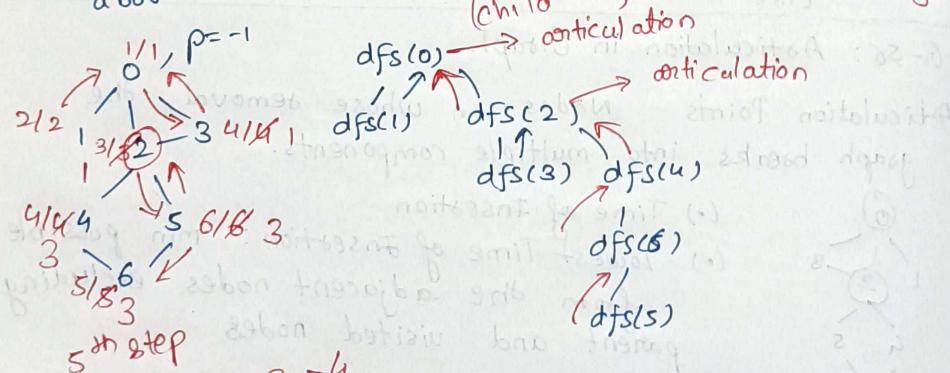
$\text{if } (\text{low}[\text{it}] > \text{tin}[\text{node}] \& \& \text{parent} \neq -1)$

for starting node:



Also, if adjacent is visited, we update
 $\text{low}[\text{node}] = \min(\text{low}[\text{node}], \text{tin}[\text{it}])$

so if 5 is removed
7 can reach lowest to 3 not
above it.
(child = 2, parent = -1) you
can't go bad



4th step $P - 3 \rightarrow 4$ $T C \rightarrow O(V+2E) - \text{dfs}$

3rd step $- 3 \rightarrow 3$ $S C \rightarrow O(3V) - \text{tin}, \text{low} \text{ and } \text{vis}$