...VER GRAPH SERIES

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

Two types of data structure ?

1. Linear (Arrays, Stacks, Queues, LL)
2. Non-Linear

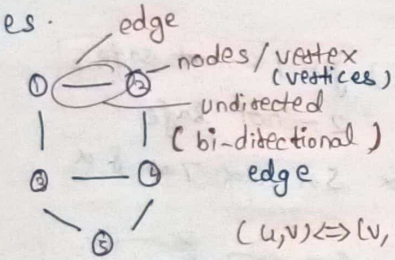
(Tree, BST, Graphs)

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

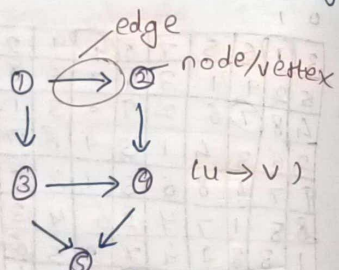
GRAPHS

Data Structures - that have a wide-ranging app^n in real life. These include analysis of electrical ckts, 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.



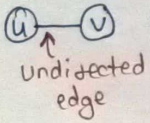
Undirected Graph



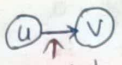
Directed Graph

numbering of nodes can be done in any order.

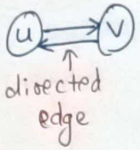
pair of vertices are called edges.



an edge between u and v (u → v)
as well as v and u (v → u)

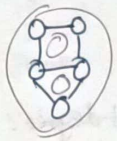


there is an edge betn u and v
u → v

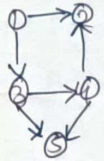


there is an edge betn u and v
u → v as well as v and u v → u

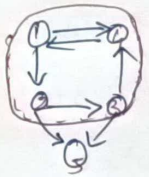
✱ Cycles



- 3 cycles Undirected Cyclic graph
start at one node & & end at that node



No cycle



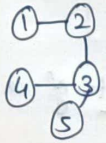
- directed cyclic graph (DCG)

1 Cycle

└ DAG (directed acyclic graph)

✱ PATH

- lot of nodes and each pair is reachable



1 2 3 5 is a path

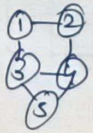
1 2 3 2 1 (NO, bcz a node can't appear twice in path)

1 3 5 (not a path, there's no edge betn 1 and 3)

✱ Degree

- no. of edge going inside or outside that node

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

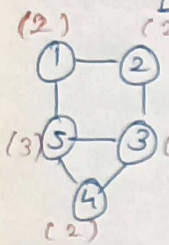


D(3) = 3 D(5) = 2

D(4) = 3

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.

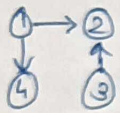
$D = 2 \times E$



$D = 2 \times E$

$12 = 2 \times 6$

For directed graphs, we have indegree and outdegree.
(incoming) (outgoing)

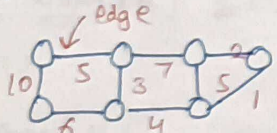


$InD(2) = 2$

$OD(2) = 0$

Edge Weight

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



else $W(edge) = 1$

If weights not assigned, we assume unity (1).

In app^n, weight may be a measure of the cost of a route.

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

G-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 ---> v then v ---> 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 ↑.

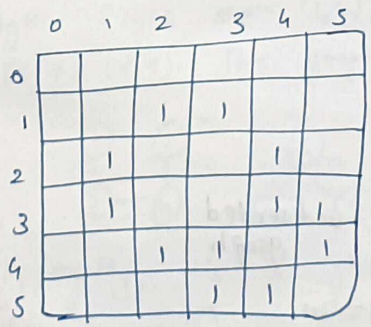
1. Adjacency matrix
2. Adjacency lists

$n = 5 \quad m = 6$

mlines
$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 4 \\ 3 & 4 \\ 3 & 5 \\ 4 & 5 \end{bmatrix}$$

--> two-dimensional array of size n x 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.



all remaining are left as it is

Space - O(N x N) costly as n^2 locations are consumed

TC - O(m)

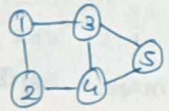
to optimise space, use ~~—~~ g

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}
2 - {1, 4}
3 - {1, 4, 5}
4 - {2, 3, 5}
5 - {4, 3}

1 2 → node 2 appears in node 1, and 1 appears in the list of node 2
1 3
2 4
3 4
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 ....> v so, SC = O(E) bc2 vice-versa = false

Weighted Graph Representation

If weights are mentioned,

for adjacency matrix

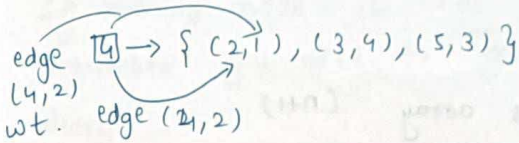
adj[u][v] = w
adj[v][u] = w } both for undirected graph

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]



G-3: Java

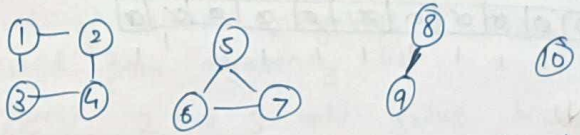
G-4: Connected Components in Graphs

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)

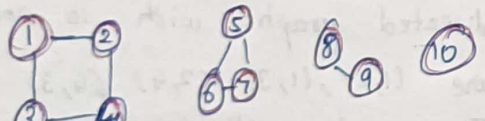
0-based : size = (n)

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 cu2, travel(1) will only traverse connected components i.e. nodes 1, 2, 3 and 4, but not the connected components.



visited

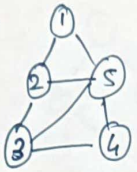
0	1	2	3	4	5	6	7	8	9	10
0	ø	ø	ø	0	ø	ø	ø	ø	ø	ø
	1	1	1	1	1	1	1	1	1	1

TC → O(N)

SC → O(N)

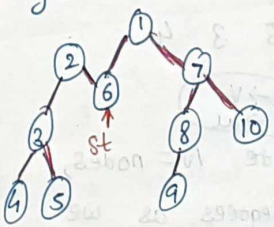
Gs: Breadth - First Search | Level Order Traversal

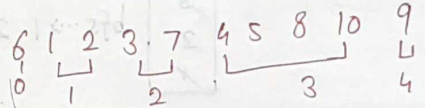
Eg -



ans 1 2 5 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
- visited array : an arr initialized to 0

Steps - 1. we start with 'st' node, mark it as visited, and push it into queue DS.

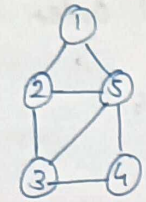
2. In every iteration, we pop out node 'v' and put in the bfs soln vector, as we are traversing the nod

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.

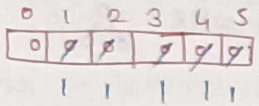
Dry Run

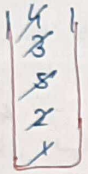


undirected graph

0 - { }
1 - { 2, 5 }
2 - { 1, 5, 3 }
3 - { 2, 4, 5 }
4 - { 3, 5 }
5 - { 1, 2, 3, 4 }

Adjacency list





bfs → 1 2 5 3 4

Time Complexity : O(N) + O(2E), where N = nodes, 2E is for total degrees as we traverse all adjacent nodes.

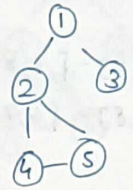
Space complexity : O(3N) ≈ O(N), space for Queue, visited array and an adjacency list.

For directed graph, TC → O(N) + O(E)

Degree = Edges (u ····> v)

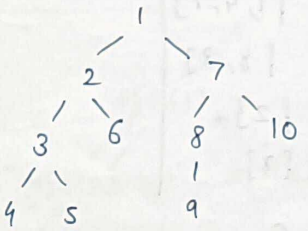
G6 : Depth First Search (DFS)

i/p -



o/p -
1 2 4 5 3

②

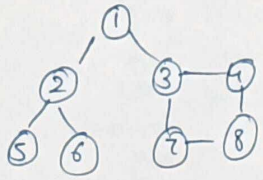


o/p -
1 2 3 4 5 6 7 8 9 10

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 curr 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)
   }
}
```

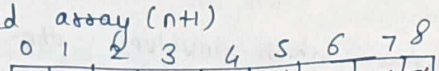
manner.



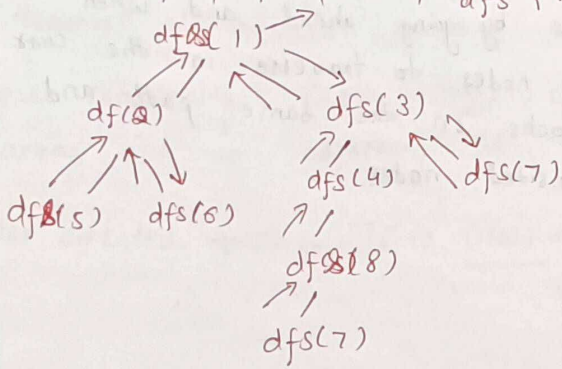
adjacency lists

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

visited array (n+1)



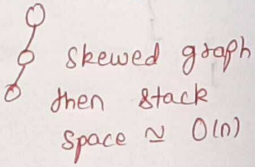
dfs { 1, 2, 5, 6, 3, 4, 8, 7 }

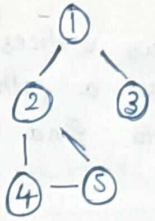


Time Complexity → O(N) + O(2E), for a directed graph O(N) + O(E) bcz for every node we are calling the recursive function once, the time taken is O(N) and 2E is for total degrees as we traverse for all adjacent nodes.

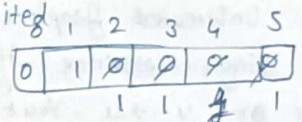
SC → O(N) + O(N) + O(N) ≃ O(N)

visited array — dfs stack space — adjacency list (dfs)





visited

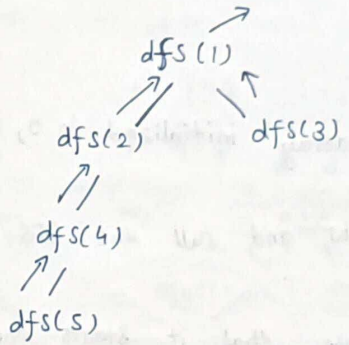


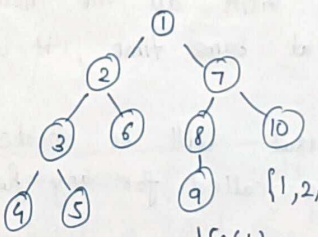
adj

1 — { 2, 3 }
2 - { 1, 4, 5 }
3 - { 1 }
4 - { 2, 5 }
5 - { 2, 4 }

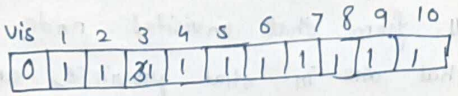
10 = 2xE

dfs 1 2 4 5 3

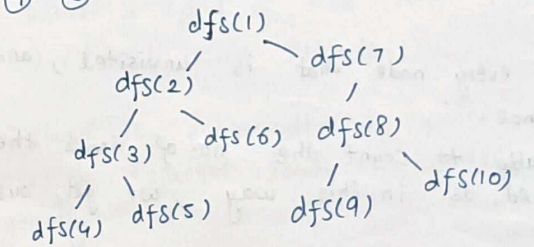




{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}



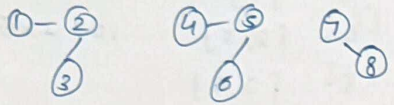
1 - { 2, 7 }
2 - { 1, 3, 6 }
3 - { 2, 4, 5 }
4 - { 3 }
5 - { 3 }
6 - { 2 }
7 - { 1, 8, 10 }
8 - { 7, 9 }
9 - { 8 }
10 - { 7 }



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 ---> v or v ---> 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.

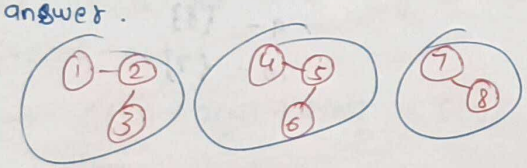
2: Run a for loop from 0 to N, and call the DFS for the first unvisited node.

3: DFS func^n 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.

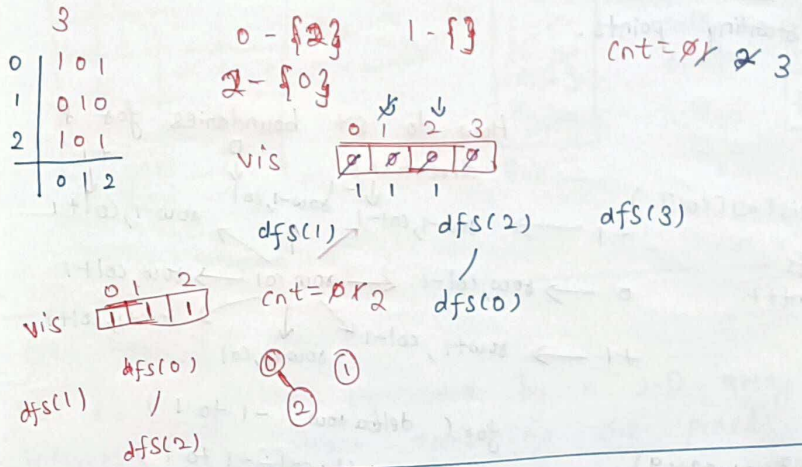


ans = 3

TC -> 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 -> O(N) + O(N), space for recursion stack space and visited array.

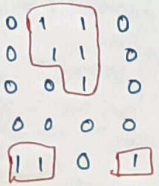
Dry Run



G8: Number of Islands

PS - Given a grid of size NxM 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.



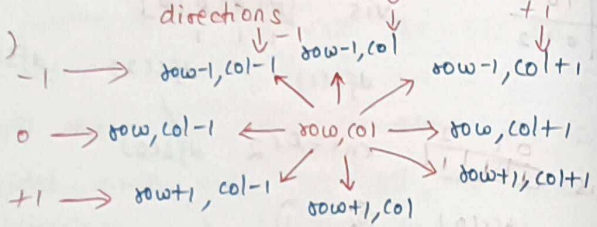
ans = 3

Intuition:

If we start a trav. algorithm, from a land(1) it will make sure it traverses all 8 direcn in nearest loom. 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[o][col])
      bfs
      cnt++
```

How do we set boundaries for 8 directions



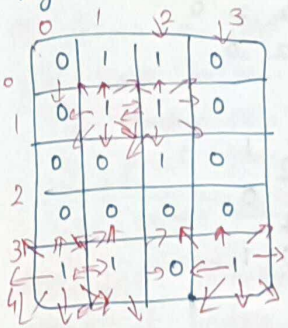

```
for( deltarow: -1 to 1 )
  for (deltacol: -1 to 1 )
    nrow = row + deltarow
    ncol = col + deltacol
```

TC → $O(N^2 + N \times M \times 9)$,
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.

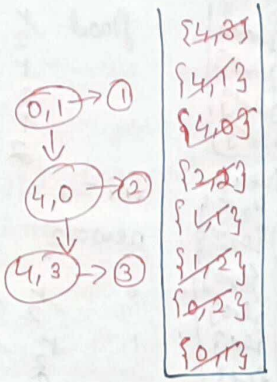
SC → $O(N^2)$ for visited array, max queue space, if all marked as 1 then max. queue space will be N^2.

Dry Run -

cnt = 0 ✗ ✗ 3



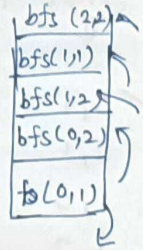
starting node: {0,1}



Queue

visited array

call stack



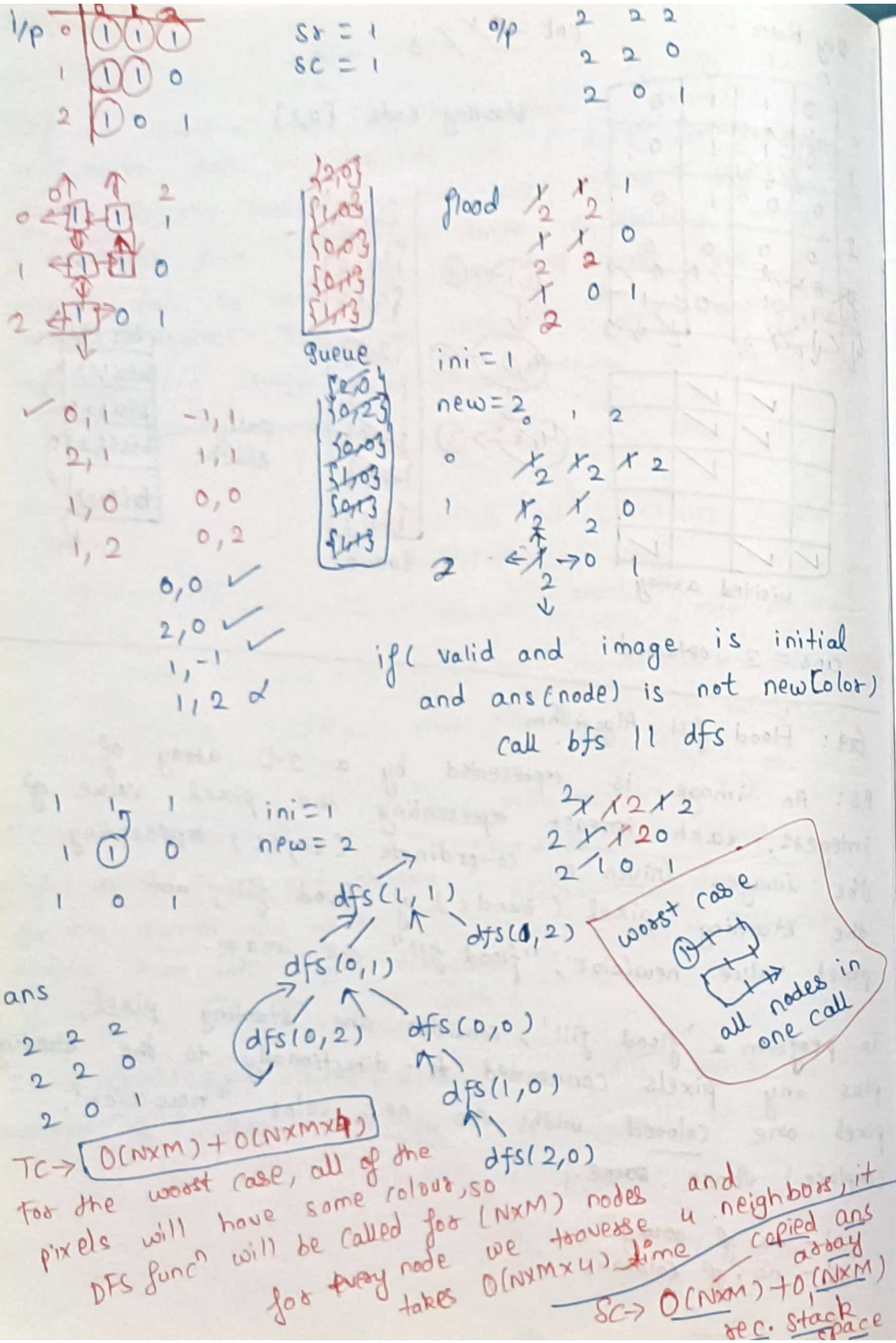
ans = 3 returned

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 (r 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



G-10: Rotten Oranges

PS: Given a grid of NxM where each cell in the grid have values 0, 1 and 2 which means

0: empty cell
1: fresh orange
2: rotten orange

Determine min. time to rot all oranges. A rotten orange at index [i,j] can rot other in 4-dim. ↔ in unit time.

i/p

t = 0 x 2 3 4

Rottening happens simultaneously.

i/p

t = 0 1

Now, Question arises which algo. to use? (↔ (4-direc^n) that are A rotten one can rot fresh orange neighbours 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 rot all oranges, we need to find minimum time to rot them all, which is possible only when we are in neighbouring directions at an equal pace.

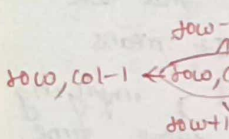
all rotten in 1 sec

Approach:

Initial configuration : Queue, Visited Array

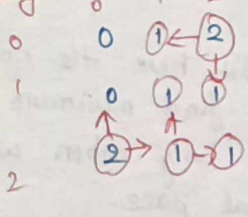
Algo -

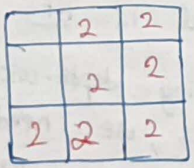
1. Store pair of cell no. and time $<<r,c>,t>$ in queue and marked them as rotten
2. While BFS, pop out an element and travel to all it's neighbours. In a graph, we store the list of neighbours in an adjacency list but here we know Nb are in 4-dimensions.
3. We use nested loops to visit neighbours.
4.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



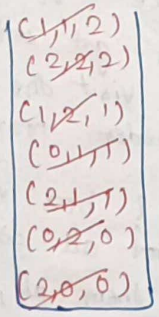
-1	0	+1	0
0	+1	0	-1

Dry Run -





Visited array



Queue

TC → O(NxM + NxMx4) ~ O(NxM), for the worst case, all of the cells will have fresh oranges, so the BFS func^n will be called for NxM nodes and for every node, we are traversing 4 neighbours ↔, it'll take O(NxMx4) time

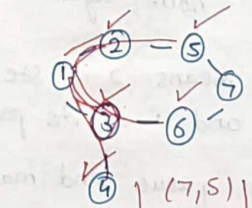
SC → O(NxM) + O(NxM) ~ O(NxM)

for copied i/p array — recursive stack spaces takes up NxM at max.

G-11 : Detect a cycle in an undirected graph → 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] contains all the nodes i^{th} is having edge with.

i/p



1 → {2, 3}

2 → {1, 5}

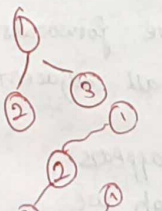
3 → {1, 4, 6}

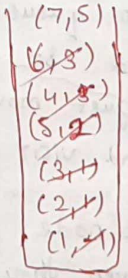
4 → {3}

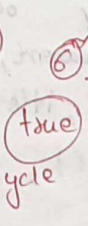
5 → {2, 7}

6 → {3, 7}

7 - {5, 6}



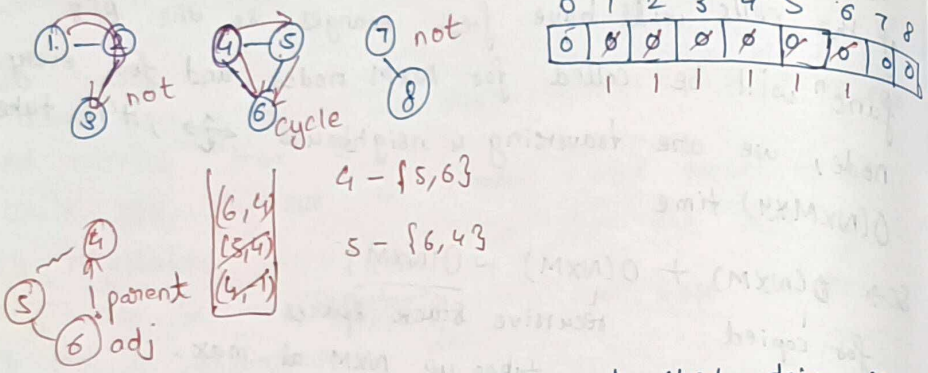




[3, 7]

when someone is already been visited and is not parent we say it's a cycle

Input can be connected comp



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 ste and 1 is its parent
Visited Array

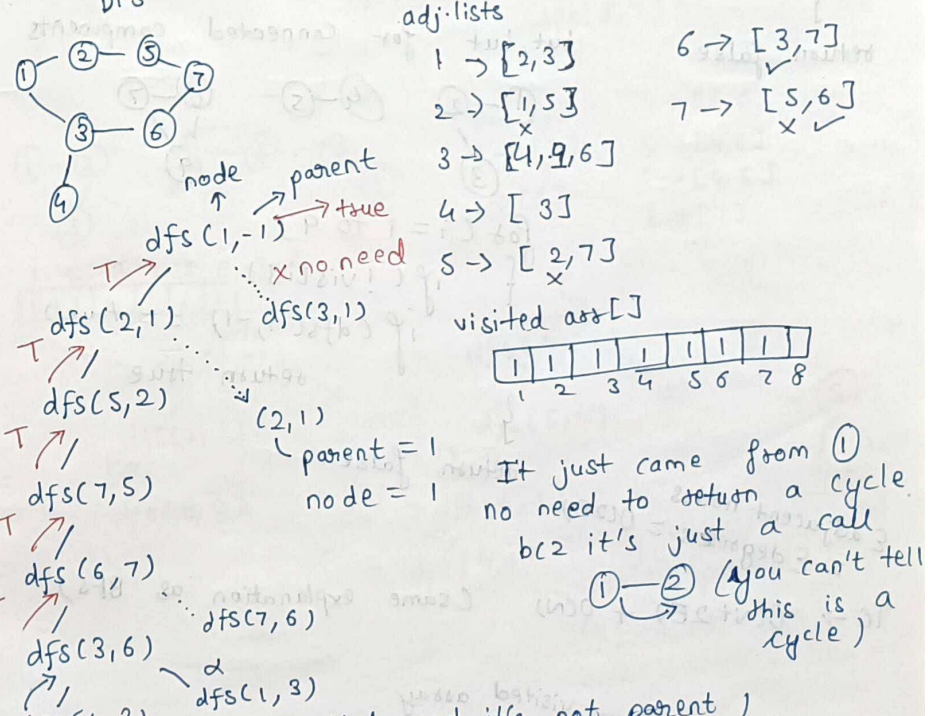
Algo - 1. push the <src, parent(-1)> to queue and mark as visited. The parent will be needed so that we don't do backward traversal and we just move forwards.
2. Start the BFS traversal, pop one and visit all adjacent nodes.
3. Repeat steps either !q.empty or a node appears to be visited which is !parent, 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

TC → O(N+2E) + O(N), where N=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.

SC → O(N) + O(N) ~ O(N), space for queue data structure and visited array.

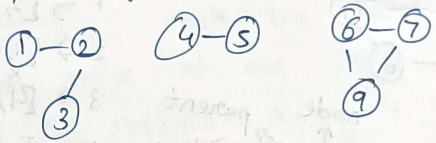
G-12: Detect a cycle in an undirected graph using DFS



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
```

Σ adjacent nodes
$= \Sigma$ degrees $= O(2E)$

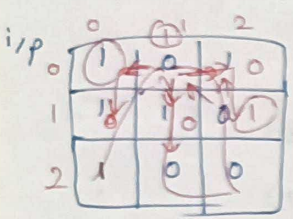
TC → $O(N+2E) + O(N)$ (same explanation as BFS)

SC → $O(N)$ (visited array) $+ O(N) \sim O(N)$
└ for skewed graph, we may end up having a recursive stack space of $O(n)$

G-13: Distance of nearest cell having 1

PS - Given a 0/1 grid N*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.



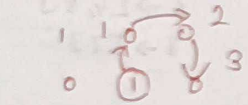
i1 j1
0,0 |0-1| + |0-0|
1,0 1+0 = 1
i2 j2

0 1 0
0 0 1 answer
0 1 2

0,0 nearest is himself

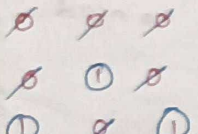
What algorithm?

DFS

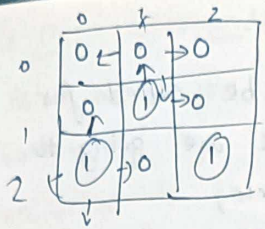


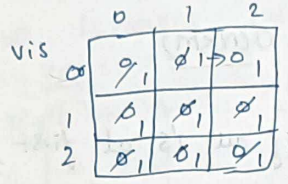
totally wrong

BFS (level-wise) steps

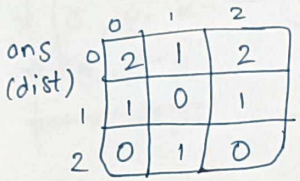


1 1 1
↓ 1 step
nearest 0's (0's are at dis. 1)
↓ 1 step
next set of 0's (dist. 2)





vis



ans (dist)

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

ans //

(0,0),2
(0,2),2
(1,0),1
(2,1),1
(1,2),1
(0,1),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 - (Ini-Config: Queue < coordinates, steps >
Visited Array)

1. Push the pair of i's in Q and mark them as visited.
2. Start BFS, pop out from Queue, and travel to all its neighbors having 0.
3. 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}
4. Repeat until !q.empty

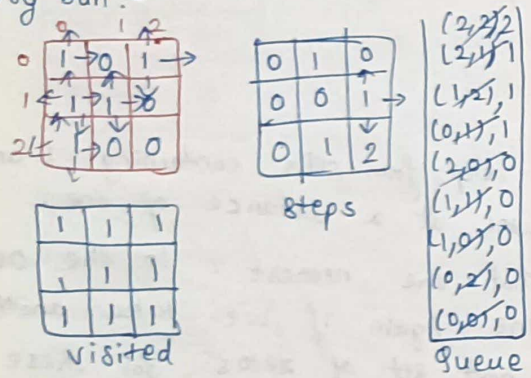
TC → $O(N \times M + N \times M \times 4) \sim O(N \times M)$

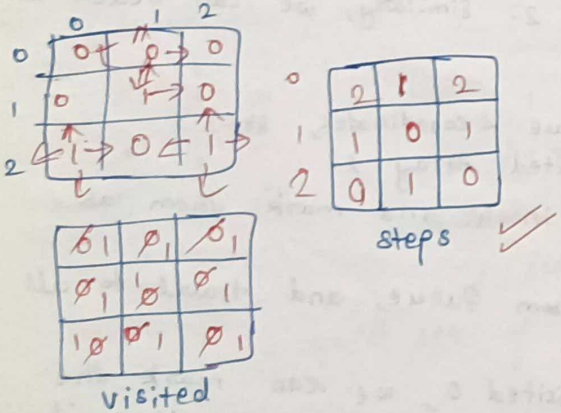
for worst case, the BFS func^n will be called for $(N \times M)$ nodes, and for every node, we are going to it's 4 neighbors, so it takes $(n \times m \times 4)$

SC → $O(N \times M) + O(N \times M) + O(N \times M) \sim O(N \times M)$

- visited array
- distance steps ans
- Queue DS (if all 1's at first)

dry run :-

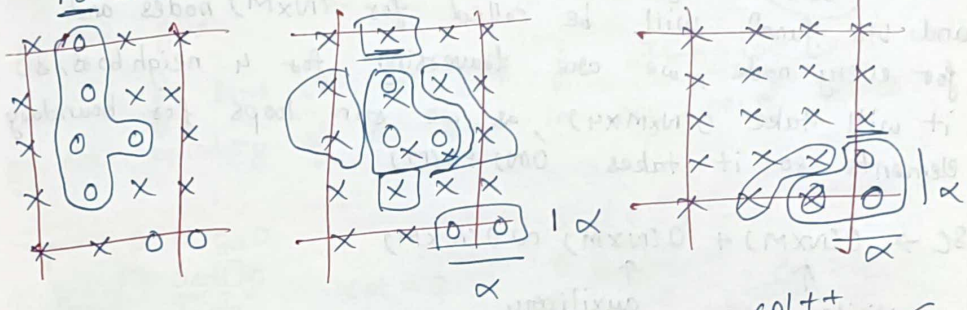


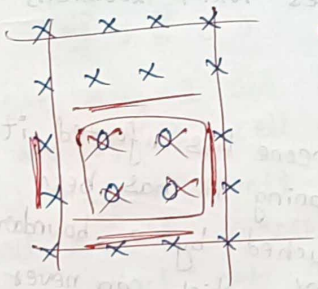


G-14: Surrounded Regions | Replace O's with X's

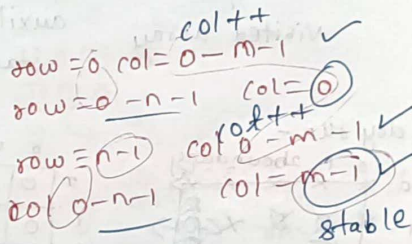
Ps - N×M grid, Replace all 'O' with 'X', that is surrounded by 'X'. An 'O' (or a set of 'O') is considered to be surrounded by 'X' if there are 'X' at locations just below, just ↑ ← → (A L R).

not connected





Intuition :

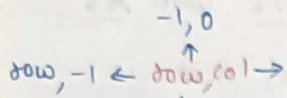


Algo - Start from boundary mark all connected O's

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 'O' (or a set of 'O') is connected to a boundary 'O' then it can't be replaced with 'X'.

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

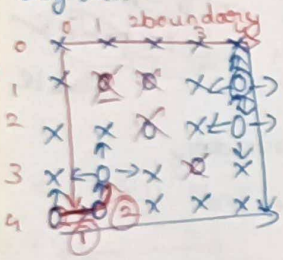


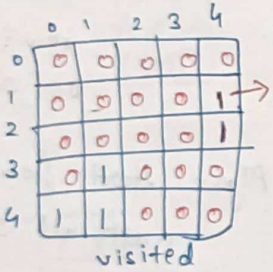
TC → O(N×M×4) + O(N) + O(M) ~ O(N×M) for worst case (000 / 000) every elem. will be marked as '0' in matrix and DFS func^n will be called for (N×M) nodes and for every node, we are traversing for 4 neighbors, so it will take O(N×M×4), as we run loops for boundary elements so it takes O(N) + O(M)

SC → O(N×M) + O(N×M) ~ O(N×M)

↑ visited array ↑ auxiliary stack space takes N×M locations at max

dry run -





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

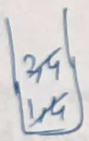
done dfs(4,0)

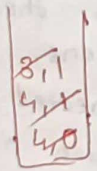
dfs(4,1)

dfs(3,1)

dfs(1,4)

dfs(2,4)





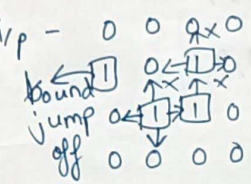
G-15: Number of Enclaves

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

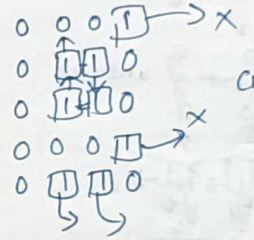
A move consists of walking from one land to another adjacent (4-direc^n) land cell or walking off boundary of the grid.

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 -



cnt = 3



cnt = 4

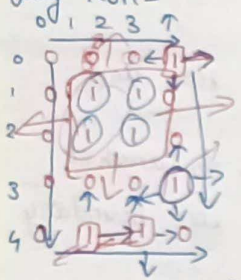
one thing I can surely say is, the cell at boundary having 1 (land) will never be my answer apparently, all the 1's connected to it can jump off the bdt, and which not account to our answer.

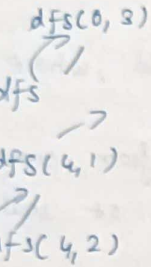
The intuition is that we need to figure out the boundary land cells, go through their connected land cells and mark them as visited. The sum of all non-lands will be the answer.

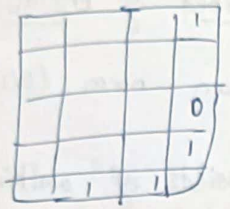
TC → O(N×M×4) - for worst case, assuming all pieces as land, the (BFS) func^n will be called for (N×M) nodes, and for ~~DFS~~ every node its 4 neighbors.

SC → O(N×M), O(N×M) for visited array, and queue/DFS stack space takes up N×M locations at max.

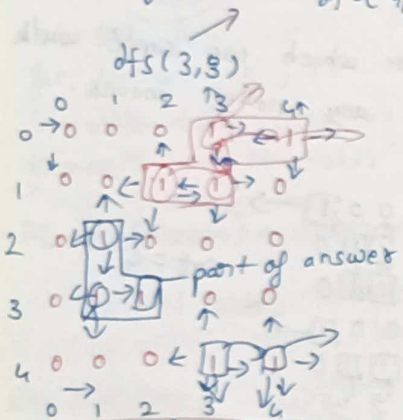
Dry Run -

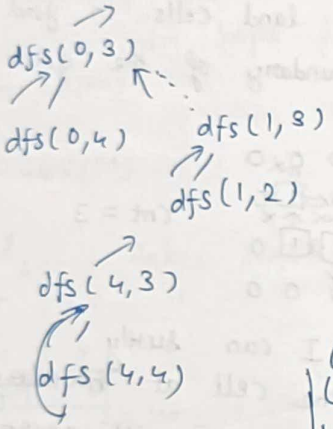


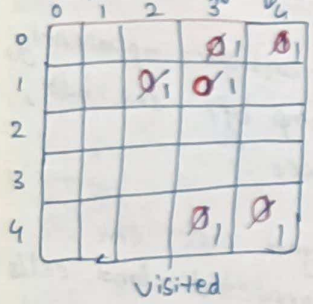


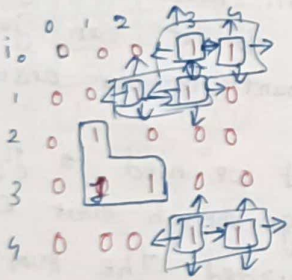


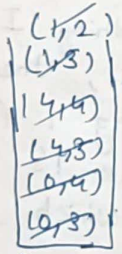
ans = 4





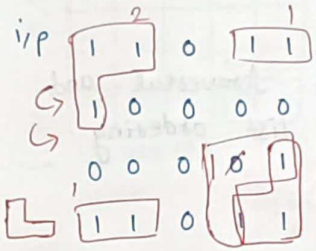




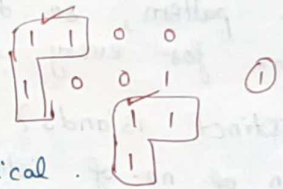


G-16 : Number of Distinct Islands | CT + DFS | (Expansion of No. Islands)

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)



ans = ~~2~~ 3



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 set will return unique islands. We can store coordinates in a vector or a list.

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

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

{ (0,0), (0,1), (1,0) }

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

X does not match

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) → (0,1)
↓ ↓
(1,0)

coordinate - base

(0,0) - (0,0) = (0,0)
(0,1) - (0,0) = (0,1)
(1,0) - (0,0) = (1,0)

(2,3) → (2,4)
↓
(3,3)

(2,3) - (2,3) = (0,0)
2,4 - 2,3 = (0,1)
3,3 - 2,3 = (1,0)

similar

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 store distinct islands?

This is expansion of no. of islands.

TC → O(n×m) + O(n×m×4) also if we use set other than unordered set then, O(n×m × log(n×m)) + O(n×m×4)

~ O(N×M)

For worst case, assuming all pieces as land, the DFS func^n will be called for (N×M) nodes, and for every node, we are traversing for 4-neighbours, it takes O(N×M×4) time. Set at max will store the complete grid, so it takes log(N×M) time.

SC → O(N×M) + O(N×M) ~ O(N×M)

visited array — set takes up N×M locations at max

Dry Run -

visited

dfs(0,3) - 0,3
dfs(0,4) - 0,3

dfs(2,3) - 2,3
dfs(2,4) - 2,3

dfs(3,3) - 2,3

ans = 2

{(0,0), (0,1)}
{(0,0), (0,1), (1,0)}
{(0,0), (0,1)}
{(0,0), (0,1), (1,0)}
(n×m)
set

dfs(0,0) - 0,0
dfs(0,1) - 0,0
dfs(1,0)

dfs(3,0) - 3,0
dfs(3,1) - 3,0

set {
vec { (0,0), (0,1), (1,0) }

(1,0)
(0,1)
(0,0)
! q.empty()
↑ set

vec {(0,0), (0,1)} ↑ set (false)
{(0,0) (0,1), (1,0)}
↑ st (false)
{(0,0), (0,1)}

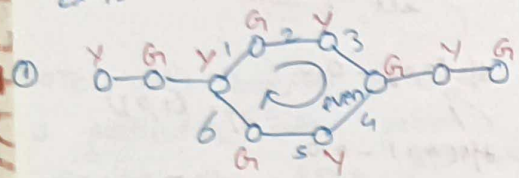
ans = 2

G-17: Bipartite Graph BFS

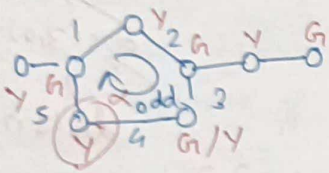
PS] Given an adjacency list of a graph adj of V no. of vertices having 0 based index. Check whether the graph is bipartite or not.

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.

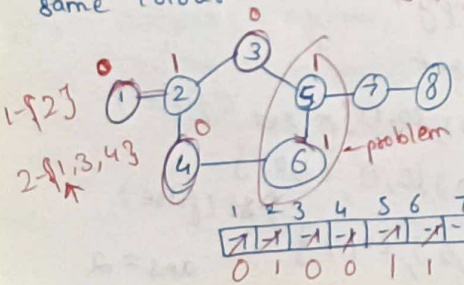


① ② ③ linear graph having no cycle are always bipartite.

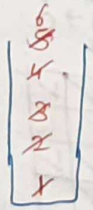


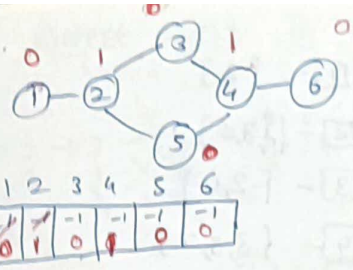
Conclusions - Even length cycles graph is bipartite.
Odd length cycles graph is not bipartite.

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



node → 5
-{3,6}
return false





node
1 - {2}
2 - {1,3,5}
3 - {2,4}
5 - {2,4}

sq empty
6 - {4}
return true

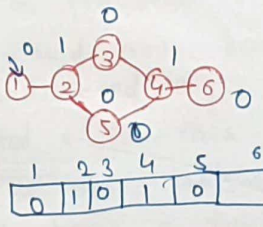
4 - {3,6}

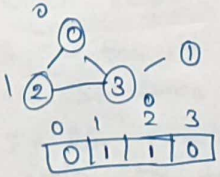
Intuition -
Brute force of fill colours, using any traversal technique just make sure no two adj. nodes have the same colour. If at any moment of traversal, we find the adj. nodes to have same color, it means that there is an odd cycle, or it can't be a bipartite graph.

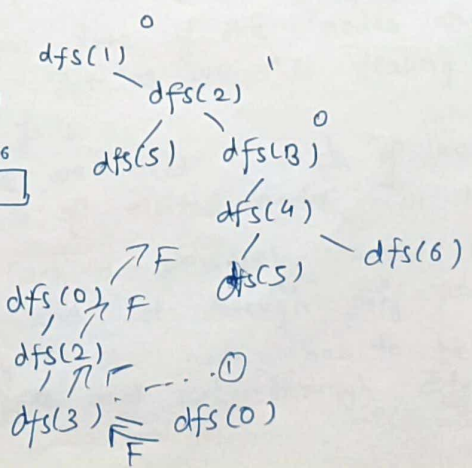
Larger Test Case Fails - Connected Components

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

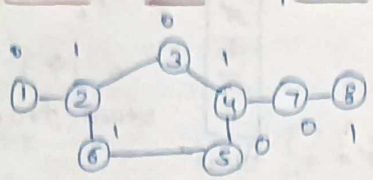
SC → O(3V) ~ O(V), Space for Queue DS, colour array and an adjacency list.







G-18: DFS Bipartite



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

Color array

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

F DFS(1,0) → False

DFS(2,1)

DFS(3,0)

DFS(4,1) → bc2 if(dfs==0) return 0

DFS(1,0) == 1 False X

DFS(5,0) True DFS(7,0) True DFS(8,1)

DFS(6,1)

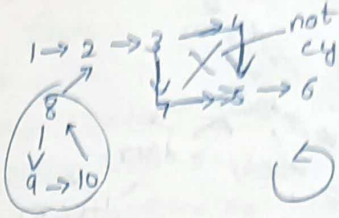
2 and 6 have same color

vis[2] == 1 ==1 return false

TC → O(V+2E)

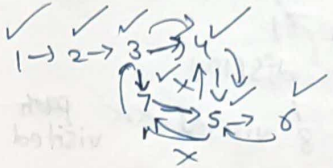
SC → O(V)

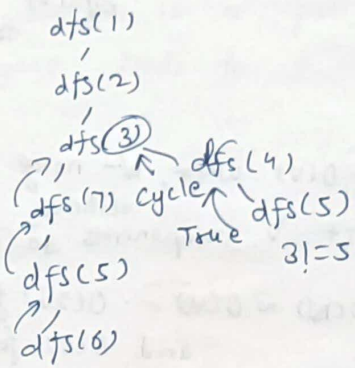
G-19: Detect cycle in a directed graph using DFS



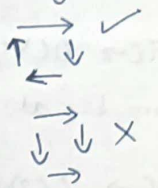
For an undirected graph, if we visit an adjacent node which is not the parent and already visited then we say it's a cycle.

But UDG algo won't work here,

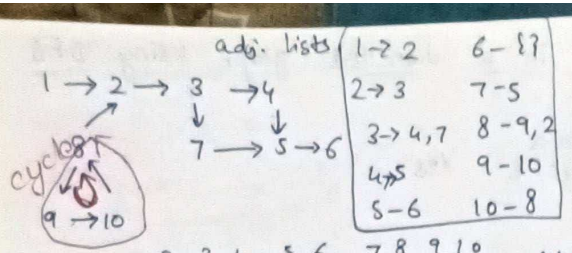


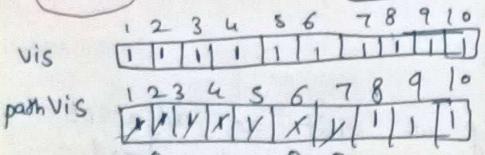


But it's not a cycle



- 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.



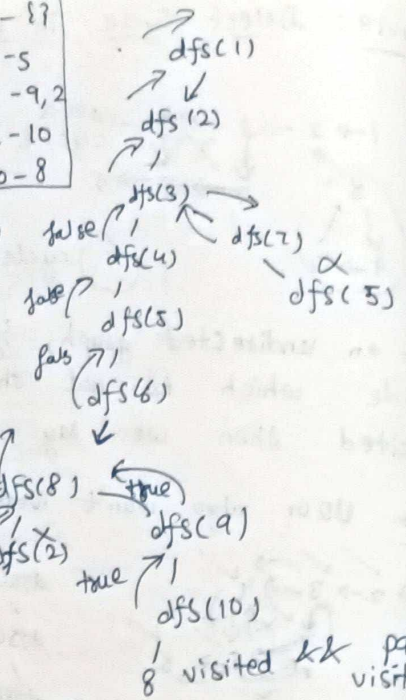


i=1 to V

if(! vis[i])

dfs(i) == true

set true

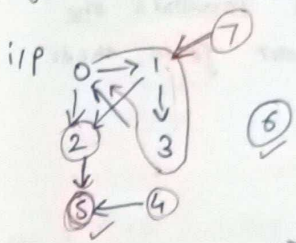


TC → O(V+E) + O(V) where, V - no. of vertices and E - edges. There can be at most V components so another O(V) TC

SC → O(2V) + O(N) ~ O(V) ... O(2V) for vis and same Path and O(V) for rec. stack space

G-20: Eventual Safe States

- 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.



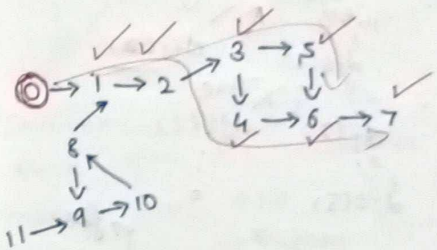
terminal nodes - outdegree = 0

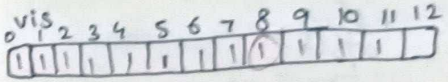
5, 6

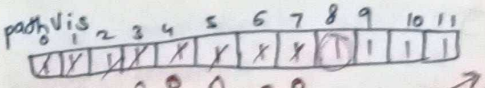
safe nodes - [2, 5, 4, 6]

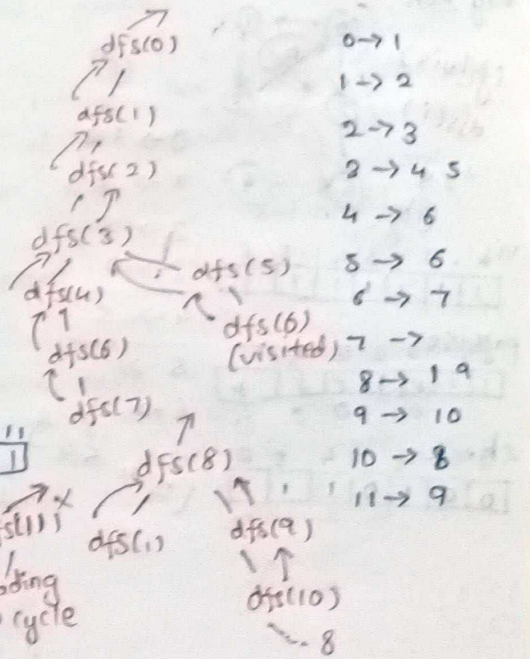
→ anyone who is a part of cycle ✗

→ anyone leads to a cycle ✗









So, the intuition is to find out 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 DG.

→ 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.

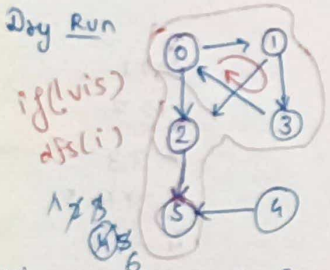
dfs — at most connected components

TC → O(V+E) + O(V) ~ O(V+E)

SC → O(V) + O(3V) ~ O(V) — arrays

recursive stack space

Dry Run



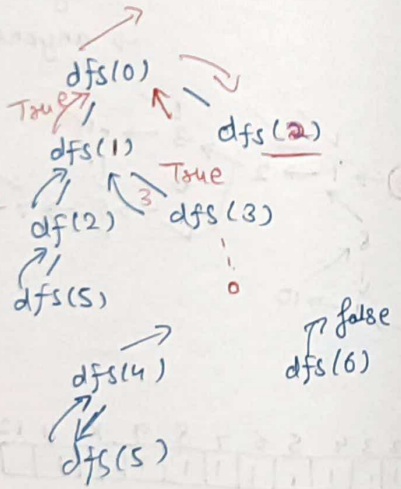
vis : 1 2 3 4 5 6

path_vis : 0 2 3 4 5 6

check : 2 3 4 5 6

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

[2 4 5 6]

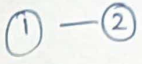


G-21 : Topological Sort Algorithm (DAG) — Directed Acyclic Graph

Any linear ordering such that u always appears before v, if there's an edge betn u and v.

Why Topo Sort only exists in DAG :

Case 1 (If the edges are undirected)



1 → 2
2 → 1

1 2 ✗
2 1 ✗

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

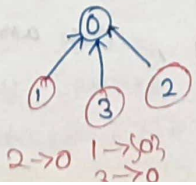
Case 2 (If directed graph contains a cycle)

1 2
2 3
3 1

cyclic dependency, so only for acyclic graph.

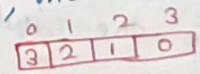
Intuition : Since we are inserting the nodes into the stack after the completion of the traversal, we are making sure, there will be no one who appears afterward but may come before in ordering as everyone during the traversal would have been inserted into the stack.

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.



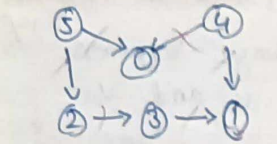
2 → 0 1 → 0 or 3
3 → 0

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



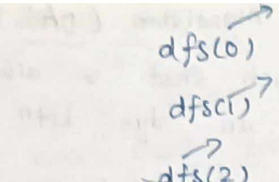
i = 0 - V

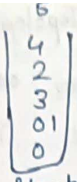
1 → 0 < map[0]
map[1] > map[0]
1st index < 0th



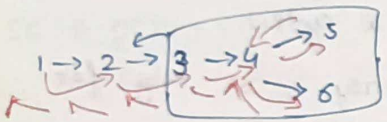
vis: 0 1 2 3 4 5

0 → {3}
1 → {3}
2 → {3,3}
3 → {1,3}
4 → {0,1,3}
5 → {2,0}





⇒ 5 4 2 3 1 0 topo sort



stack: 3, 4, 5, 6

when DFS call for 3 is over, we must have nodes 3,4,5 and 6 linearly ordered in the stack.

TC → O(V+E) + O(V)
dfs — at most V connected components
V - no. of nodes
E - no. of edges

SC → O(2N) + O(N) ~ O(2N)
vis and stack — rec. stack space

G-22: Kahn's Algorithm | Topological Sort | BFS

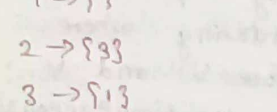
Intuition: we will slightly modified version of BFS where we will be requiring Queue and an Array to store the indegree of each node.

(2) ← indegree = no. of incoming edges = 2

Initial Config - Queue, Indegree Array, Ans array

when a node's ... nodes are decreased by 1 and if any of them 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.

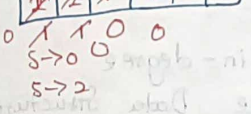
Dry Run :-



adj. List
5 → {0,2}
4 → {0,1}
3 → {1}
2 → {3}
1 → {}
0 → {}

ans {5 4 2 0 3 1}

Indegree



Because, 5 has no indegree so one points to 5 hence, no one will appear before 5 in the ordering thus we'll append it to our answer.

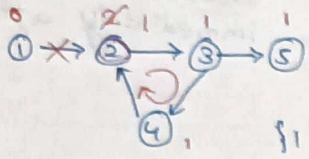
TC - O(V+E), where V - no. of nodes, E - no. of edges.
This is a Simple BFS Algorithm

SC → O(N) + O(N) ~ O(2N), O(N) for indegree array, and O(N) for queue data structure used in BFS

G-23: Detect Cycle in DG | 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 DCG, 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 ind[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 → O(V+E) V - vertices, E - edges

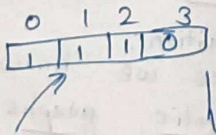
SC → O(N) + O(N) ~ O(N) ... O(N) - for in-degree
O(N) - Queue Data Structure
N → no. of nodes

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

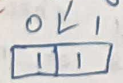
There are a total of N tasks, labeled from 0 to N-1. Some tasks may have prerequisites, for an eg - To do task 0 you have to first complete task 1, which is expressed as {0,1}

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

① [1,0] 3 → 2 → 1 → 0 (no cycle) ✓ YES
[2,1] 1→0
[3,2] 2→1



② [1,0] 1→0 (cycle) ✗ NOT POSSIBLE
[0,1]



Striver -

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 → 0 u → 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 → v) where u and v are two nodes.
Each task → node, Each pair - directed edge betn 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 → O(V+E) V - vertices, E - edges

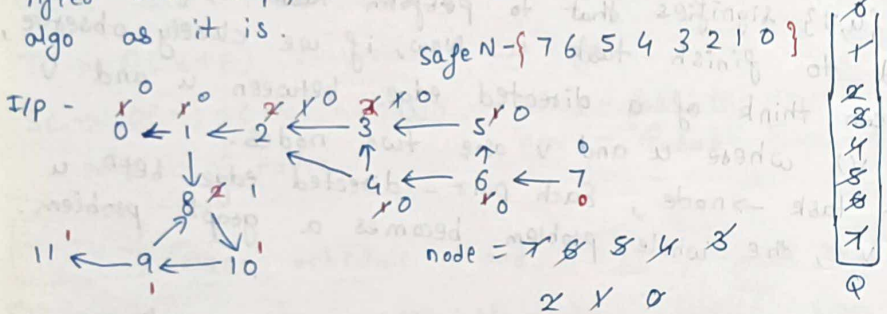
SC → O(N) + O(N) ~ O(N) - one for indegree array
+O(N) and one for Queue DS and
↳ extra O(N) for Topo Sort Array.

Gr-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 → {1,3} 2→1, 2→3
1 → {2 3}
3 → {2 3} indegree[it]++ (2)

TC → O(V+E) + O(N*logN), where, V- no. of nodes, E - no. of edges. Extra O(N*logN) for sorting safeNodes, where N is no. of safe nodes.

SC → O(N) + O(N) + O(N) ≈ O(3N), indegree, Queue and adjacency reverse list.

Gr-26 : Alien Dictionary

PS => 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.

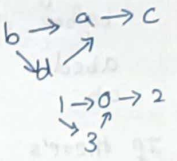
a, b, c, d ..... x, y, z → 26

i/p N=5, k=4

baa
abcd
abca
cab
cad

b<a (b→a)
d<a (d→a)
a<c (a→c)
b<d (b→d)

bdac (4) order



caa
aaa
aab

c<a
a<b

c→a→b

cab (3) order

If a<b then a must appear before b, meaning I can correspond to an directed edge between a and b (a→b) such that a appears before 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→1, c→2 ....

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

Edge Case: for 3rd i/p if k=5 no word having 'e'
we add a separate node 'e'

b → a → c
 ↘ d ↗
e

1 → 0 → 2
 ↘ 3 ↗
4

Interview
Follow-Ups:

- When the ordering is not possible?

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

② If there's a cyclic dependency
abc
bat
ade
a→b, b→a, 0→1, 1→0
a<b<a (not possible)

We apply Kahn's Algorithm to solve

Dry Run -
baa
abcd
abca
cab
cad

b→a
1→0
d→a
3→0
e→c
0→2

4 → {0,3}
3 → {0}
0 → {2}
2 → {}

b→d
1→3

ans = "bdac"

0 1 2 3
X 0 X X
X 0 0 X
Queue

1 → 0 → 2
 ↘ ↗
3

While storing just do char - 'a' to get no. and afterwards (add + 'a') to get char

TC → O(N * len) + O(K+E), N - no. of words, 'len' is upto index where first inequality occurs, K - no. of nodes, E - no. of edges

SC → O(K) + O(K) + O(K) + O(K) ≈ O(4K)
↑ adj. List ↑ Indegree ↑ Queue ↑ ans to store the order

Ways Swap Two Numbers in 5 diff. ways?

① temp = a, a = b, b = temp

② b = a+b
a = b-a
b = b-a
or
b = a*b
a = b/a
b = b/a

③ In-built swap(a, b)

④ a = a^b
b = b^a
a = a^b
(^-OR)
a = 2, b = 3
b = 3+2 = 5
a = 5-2 = 3
b = 5-3 = 2

a = 2^3 = 1
b = 1^3 = 2
a = 1^2 = 3

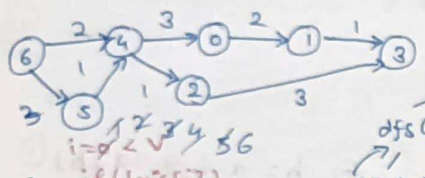
b = -1+3 = 2
a = b = 3

⑤ b = a-b + (a=b) (optimal)

④ b = 6
a = 6/2 = 3
b = 6/3 = 2

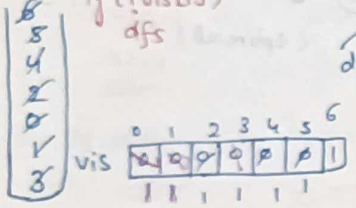
G-27: Shortest Path in Directed Acyclic Graph - Topological Sort

From any source node -



i=0 1 2 3 4 5 6

if(!vis[i])
dfs



dfs(0)
dfs(1)
dfs(3)
dfs(2)
dfs(4)
dfs(5)
dfs(6)

src = 6

adj. List
0 -> {1, 2} (1)
1 -> {3, 1} (2)
2 -> {3, 3} (3)
3 -> {3} (4)
4 -> {0, 3}, {2, 1} (5)
5 -> {4, 1} (6)
6 -> {4, 2}, {5, 3} (7) (8)

Algorithm -

Step 1 - Do a Topo Sort on the Graph

Step 2 - Take the nodes out of stack & relax the edges

dist[] = 0 1 2 3 4 5 6 -> imp
∞ ∞ ∞ ∞ ∞ ∞ 0
{5 7 3 6 2 3 0} (src - src)

src = 6
0

node = 6, dis = 0
+2 / +3
node = 4 / node = 5

node = 5, dist = 3
+1
node = 4

node = 2, 3
+3
node = 3,

node = 4, dist = 2
+3 / +1
node = 0 / node = 2

node = 0, dist = 5
+2
node = 1

node = 1, dist = 7
+1
3 (8)

node = 3
✗

If any remains as ∞, we make it -1 so as to denote it's unreachable.

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.

Djikstra's is useful for graphs that contain cycles, becz they can't be topologically sorted.

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

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
~ O(N+M)

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

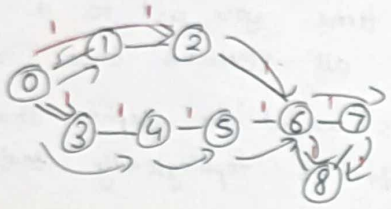
G-28: Shortest Path in an Undirected Graph with Unit Weights.

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

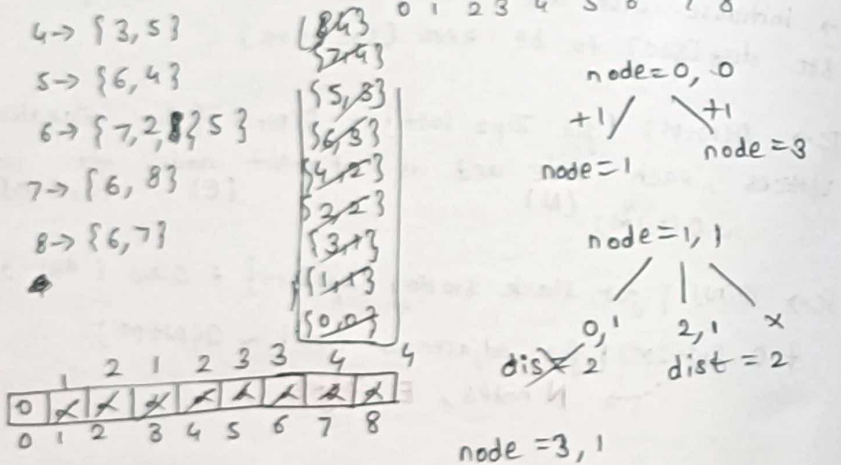
{0,1} {0,3} {3,4} {4,5} {5,6} {1,2} {2,6} {6,7} {7,8} {6,8}

0 → {1,3}
1 → {0,2}
2 → {1,6}
3 → {4,0}
4 → {3,5}
5 → {6,4}
6 → {7,2,8,5}
7 → {6,8}
8 → {6,7}



O/P - 0 1 2 1 2 3 3 4 4

dist: 0 1 2 3 4 5 6 7 8



Intuition: we will calculate the shortested 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 (∞), mark src as 0
- perform BFS
  - if dist[node] + 1 < dist[adjacent]
    dist[adjNode] = dis[node] + 1
  - push if (true) q.push(adjNode, dis[adjNode])
- else
after BFS
some one is unreachable mark it as -1.

TC → O(M) { for adj. list creation } + O(N+2M) { for BFS }
+ O(N) { for adding -1 to unreachable nodes }
~ O(N+2M)

SC → O(N) { for BFS Queue } + O(N) { for dist. array }
+ O(N+2M) { for adj List } ~ O(N+M)

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

G-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 targetword.
- startword may or may not be part of wordlist

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

st = des , end = dfs

des(1) → dfr(2) → dfs(3) ans = 3

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

st = toon , end = plea

1 toon (1)
2 poon (1)
3 poin
4 poie (1) – plie (5) – plea (6) → plea (7)

ans = 7

word st = 'hit' end = 'cog'

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

hit(1) → hot(2) → dot(3) → dog(4) → log → lot
dog(4) → cog(5)

hit → level 1

'a-z' hat, hbt, ... hit, (hot), h2t

hot → 2
dot → 3
dog → 4
cog → 5 { shortest sequence }

Dry Run -

set → ~~hot~~(1), ~~dot~~(1), ~~dog~~(1), ~~lot~~(1), ~~log~~(1), ~~cog~~(1)

hit
hot
dot
lot
dog
log
cog

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

O(N)

'end == cog' return 5

mumbai – odisha(2) – hydrebad(3) – kerala
kolkata – bangalore – odisha(3) – hydrebad – 3

'hot' is executed then I'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

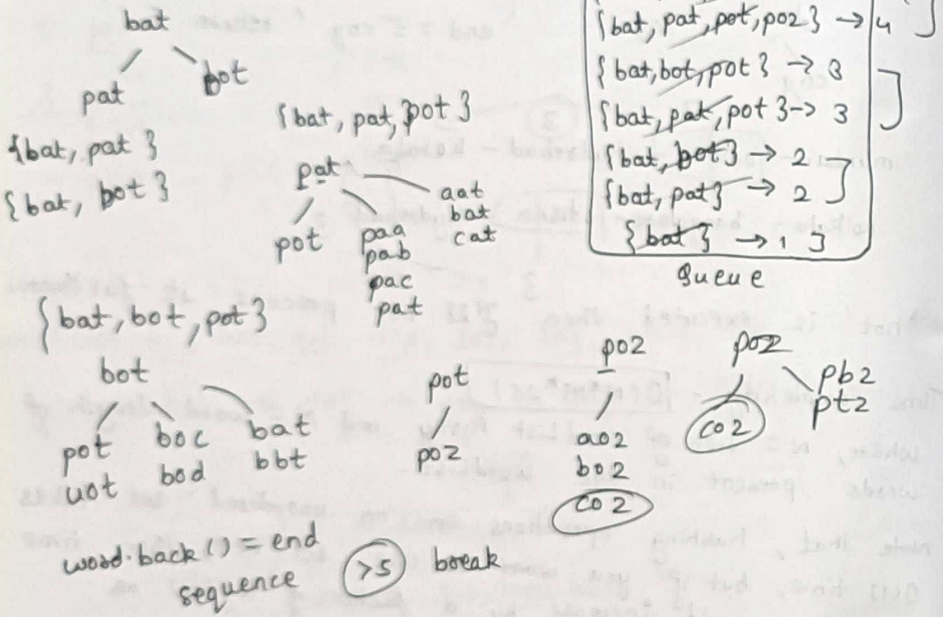
G-30: Word Ladder - 2

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 = [~~pat~~, ~~bat~~, ~~pot~~, ~~po2~~, ~~co2~~]

begin word = bat word = bat



{bat, bat, pot, po2, co2} → 5
{bat, pat, pot, po2, co2} → 5
{bat, bat, pot, po2} → 4
{bat, pat, pot, po2} → 4
{bat, bat, pot} → 3
{bat, pat, pot} → 3
{bat, bot} → 2
{bat, pat} → 2
{bat} → 1

Queue

Contrary to previous one, here we don't stop the traversal on the first occurence of the targetword, but continue for as many occurences 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.

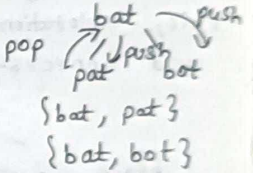
Step 2. Queue to store paths in form of vector

Step 3. add {stWord} to list and also to usedOnLevel as {}

Step 4. Pop first element (i.e. back of the path)
- word - check all its chars 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 {}.

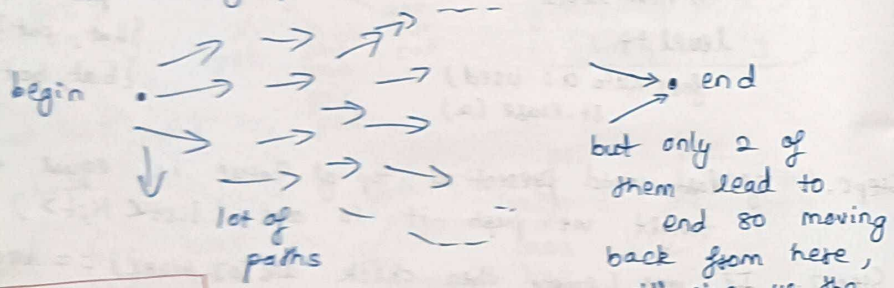
TC → Cannot be predicted bcz there can be multiple sequences of transformation from st to end, depends on example. So we can't define a fixed range of time or space in which this program would run for all test cases.

SC

G-31: Word Ladder - 2 (Optimised 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 cnt 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.



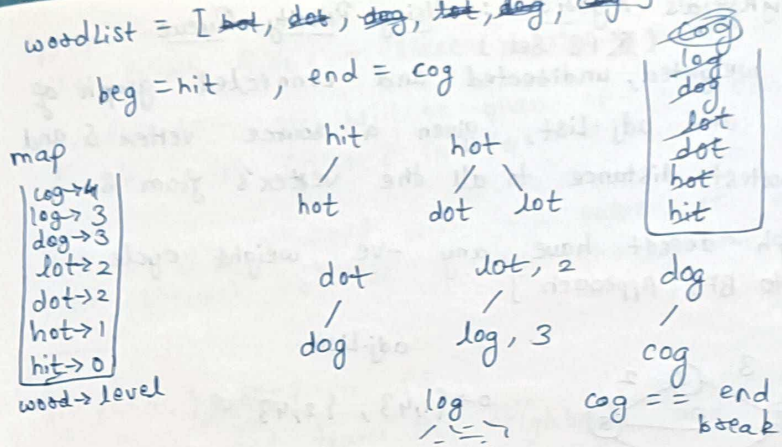
INTERVIEW TIP:
DON'T DECLARE GLOBAL VARIABLES

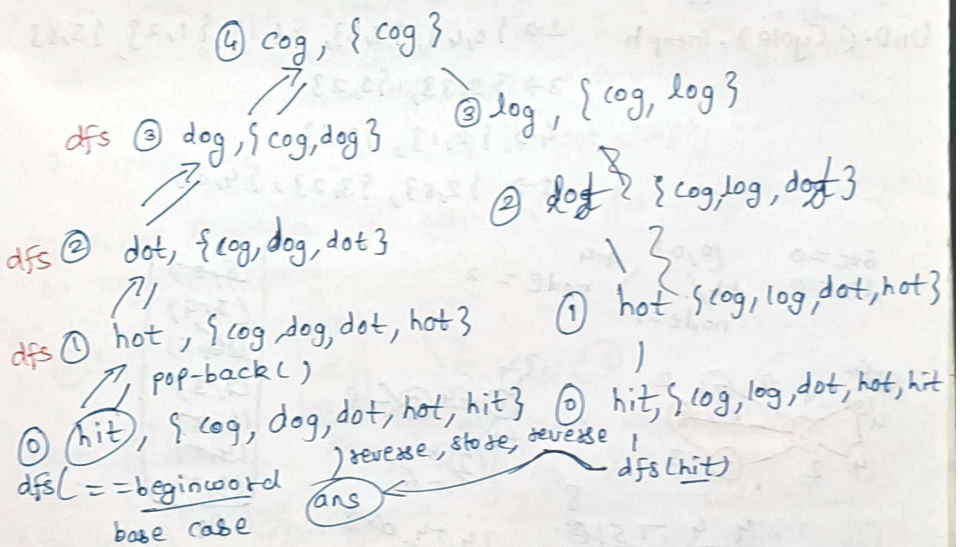
Initial Configuration:

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

wordList = [~~hot~~, ~~dot~~, ~~dog~~, ~~lot~~, ~~log~~, cog]

beg = hit , end = cog





Time Complexity and Space Complexity →

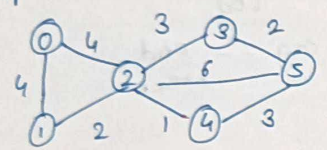
Cannot be predicted, depends on the test cases we can't define a fixed range of time or space.

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]

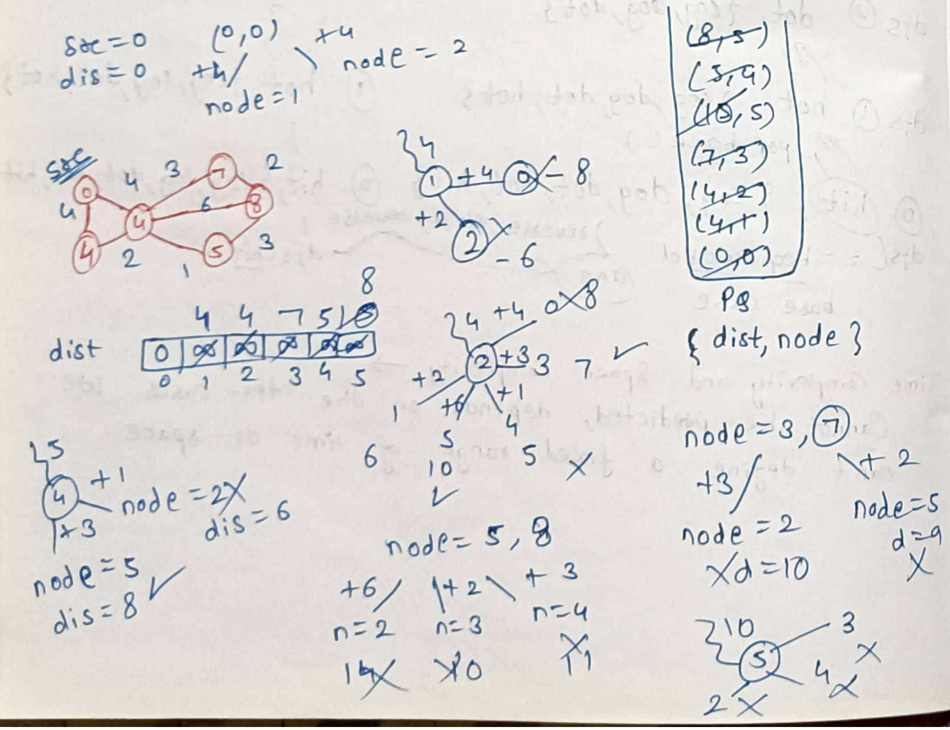
i/p -



UnD. (Cycle). Graph

adj. List

0 -> {1,4}, {2,4}
1 -> {0,4}, {2,2}
2 -> {0,4}, {3,3}, {4,1}, {1,2}, {5,6}
3 -> {2,3}, {5,2}
4 -> {2,1}, {5,3}
5 -> {2,6}, {3,2}, {4,3}



Approach -
Use Priority Queue - store {distance, node}.

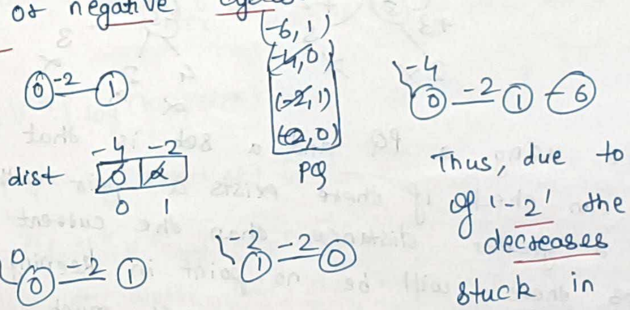
Source Node - src will be given or we can take one and push {0,src} in PQ

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

Algo -
1. push {0, src} in PQ
2. pop it and travel its neighbours and if
   currDistance + weight < dist[adjNode]
   - Push in PQ {currdis + 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 -



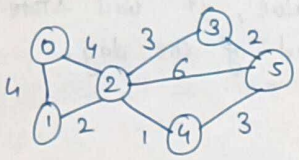
Thus, due to negative weight of '-2' the distance always decreases, and we get stuck in an infinite loop.

TC -> O(E log V)   E - edges, V - vertices

SC -> O(|E| + |V|),   E - edges, V - vertices

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

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



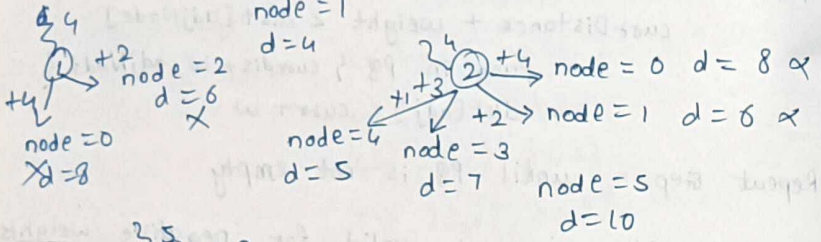
set (increasing order)

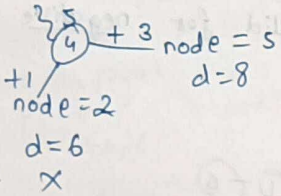
(0,0) (4,1) (4,2) (7,3) (8,5) (10,5)

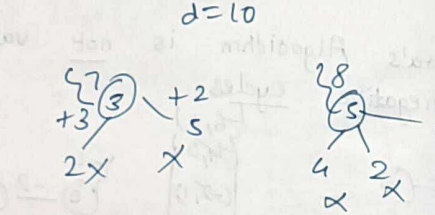
(5,4)

0, 0 → +4 node = 2 d = 4

+4 node = 1 d = 4







Intuition:

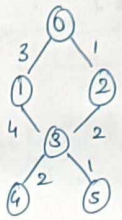
The only difference using a PQ and a set is that in a set we can check if there exists a pair with same node but a greater distance than the current inserted node as there will be no point in keeping that node into the set if we come across much better value than that.

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 explicity say that set is better bcz by saving iterations we still took O(log N) for set.erase(), minor-minor difference.

G-34: Why PQ and not Q, Intuition, Time Comp. Deriv^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.e 0-2-3.

Derivation of TC →

Pseudocode -

while(!pq.empty()) → V times

{ dist node = top() - log (HeapSize)

V-1 → for(all adj nodes)

O(V*(pop vertex from PQ) (each node { if (condition)

+ no. of edges on each is connected update dist

vertex * push in PQ to other nodes) push in PQ

↳ log(HeapSz)

} }

O(V*[log(heapsize) + V-1 log(HeapSz)] }

O(V* log(heapSize) (1+V-1))

O(V*V* log(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

and all to V-1

V*(V-1) = V^2 edges

OR E edges

O(V^2 log(V^2)) => O(E · log V)

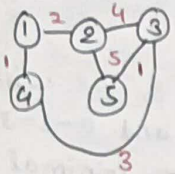
O(2 V^2 log V) = O(E · log V^2) → for set

= O(E log V) worst case → for PQ

Sc → O(E+V) O(ElogE) for PQ

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.

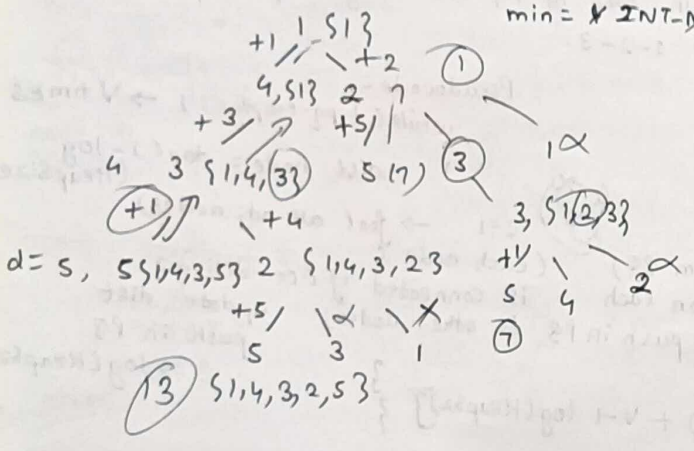


1 → 2 → 5 = 7

1 → 2 → 3 → 5 = 7

1 → 4 → 3 → 5 = 5

{1, 4, 3, 5}



Intuition:

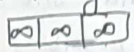
① Intuition behind the above problem is based on Dijkstra Algo. with a combination of little sum (bit) of memoization in order to print the shortest possible path and not just calculate the shortest distance, in order to print we will try to remember the node from which we came while traversing each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

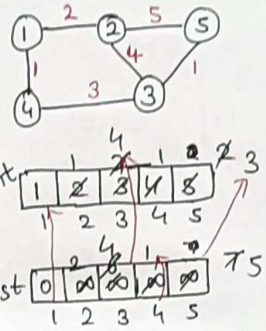
An array called 'parent' can be used, which will store the parent node for each node and will update itself if a shortest path from a node is found at some point Thus, we backtrack through parent array till source node

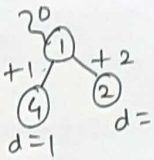
Initial configuration

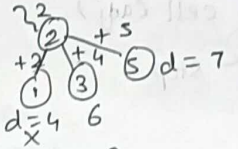
1. Source Node (1) 2. Priority Queue {dist, node} (0, 1) PQ

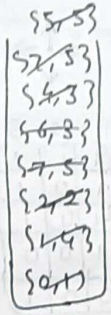
3. Adj list 4. dist Array (∞) 5. Parent Array



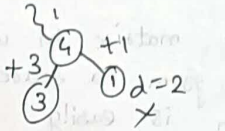


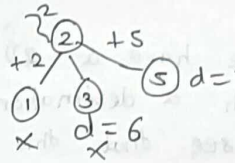


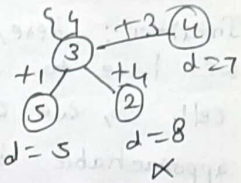


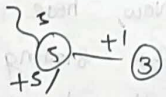


PQ {dist, node}







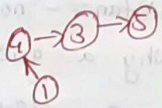


dest = 5

arr = [5 3]

parent[5] = 3

p[3] = 4



{1, 3, 4, 5}

{5, 3, 4} p[4] = 1

p[1] = 1 != 1 false

TC → O(E log V) {for Dijkstra's}

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

SC → O(E+V) {for priority queue and dist array}

+ O(V) {for storing final path}

G-36 : Shortest Distance in a Binary Maze

Given an n*m grid, where each element can either be 0 or 1. You need to find the shortest distance betn src and dest. If not possible, return -1.

Move - ←↑→↓

i/p

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

ans = 3

cell → cell (adj)

(+1) step

src = {0,1}

dst = {2,2}

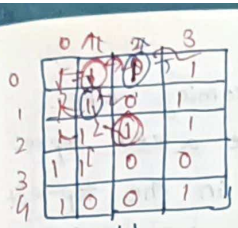
Intuition : Here, we have a 2D binary matrix in which we have to reach a destination cell from a source cell. So, we can see that this problem is easily approachable by Dijkstra's Algorithm. Now, here we use a queue instead of a priority queue, for storing the distance - node pairs. Let's understand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

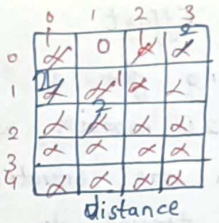
We can clearly see, distances are increasing monotonically since, greater distance comes at top, so we need not do with PQ as pop operation will always pop minimal guy. This helps us to eliminate extra log(N) of time needed to perform push-pop in PQ.

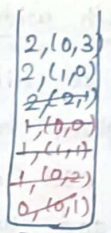
Initial Configuration -

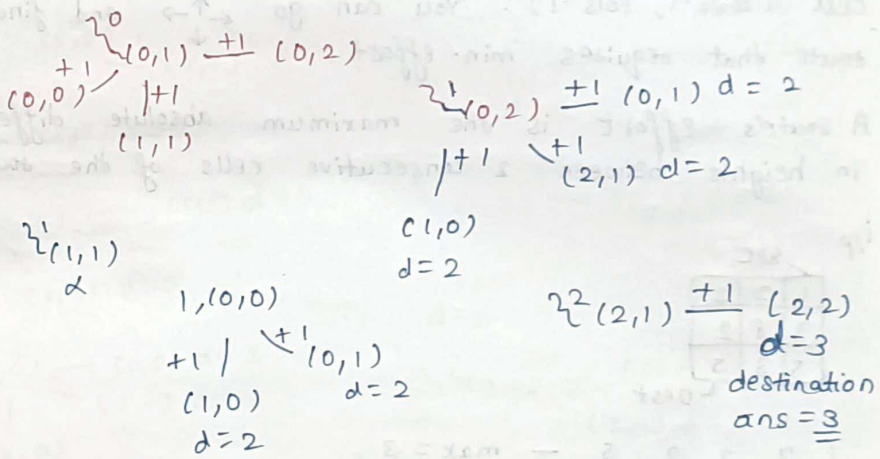
① src, destination ② Queue {distance, {row, col}}

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









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 ; N - no. of rows, M - no. of columns

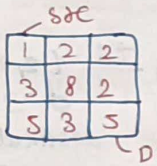
SC → O(N*M) - distance matrix

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 repre. 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 (rows-1, cols-1). You can go ←↑→↓ and find a route that requires min. effort.

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

i/p



1 2 2 2 5 — max = 3
(1) (0) (0) (3)

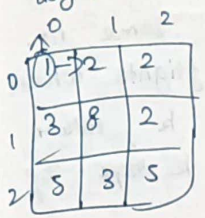
1 3 5 3 5 — max = 2
(2) (2) (2) (2)

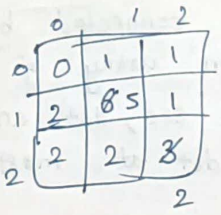
1 3 8 2 5 — max = 6
(2) (5) (6) (3)

min(3,2,6) = 2

Intuition - we need to minimize effort of moving from source to dest. The effort can be calculated as max. value of the difference betn 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 -





pq

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

2^0 (0,0) — (0,1) d=1
|
(1,0) d=2

2^1 (0,1) — (0,2) d=0
|
(1,1)
d=6
max (0,1) = (1)

2^1 (0,2) — (0,1) d=0
\
(1,2) d=0

2^1 (1,2) — (2,2) d=3
|
(1,1)
d=6
max = (3)

2^2 (1,0) — (0,0) d=2
\
(1,1) d=5
|
(2,0)
d=2

2^2 (2,0) — (1,0) d=2
\
(2,1) d=2

2^2 (2,1) — d=5 max(5) 5<5
\
(2,2) d=2 2<3 ✓

2^2 (2,2) end & stop

minimal

max = 2

1 0 3 6
1 2 3

max of path of more than 2

n×m - total nodes
←↑→↓ (4 outdegrees)

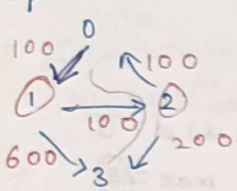
TC → O(ElogV) ~ O(4*N*M log(N*M)) ; N*M are total cells, for each of which we also check 4 adjacent nodes for minimum effort and addn log(N*M) for insertion-deletion operations in priority-queue?

SC → O(N*M) ; distance matrix + PQ (min-Heap) takes O(N*M) in worst case?

Q-38: Cheapest Flights within K stops

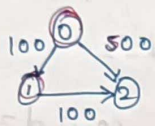
PS - n cities and m edges connected by some no. of flights. You are given an array of flights where flights[i] = [from, to, price], src, dst and k, return cheapest price from src to dst at most k stops. If no route then return -1.

I/p



k = 1

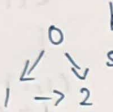
100 + 600 = 700



k = 1 one

src = 0

dst = 2



100 + 100 = 200

500

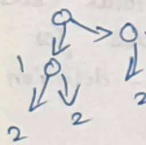
Intuition:

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

If we store elements in PQ wrt 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

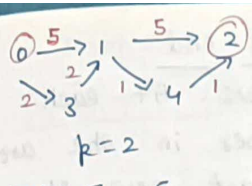
Also, do we need a PQ? (NO), stops are monotonically ↑ and when we pop we pop minimal, thus extra O(log N) of insertion-deletion in PQ will be saved, use a Queue.

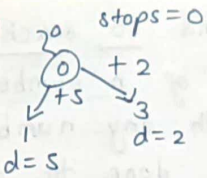


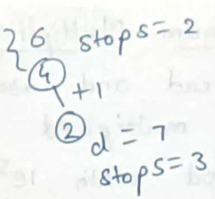
Pre-Configuration - 1. Queue { stops, {node, cost} }

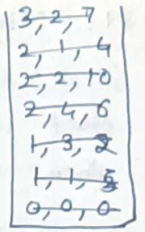
2. Distance Array - min cost

3. src & Destination







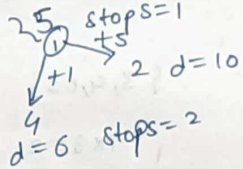


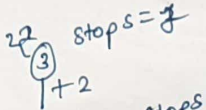
stops, node, di...

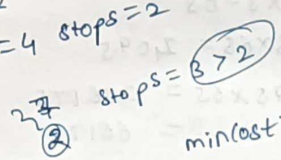
Queue

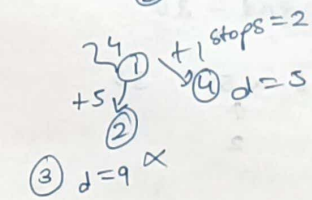
dist 4 7 5

0	5	10	2	6
0	1	2	3	4









3, 4, 5

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

SC → O(|E| + |V|) { 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 st, 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 x 2 = 6 x 5 = 30

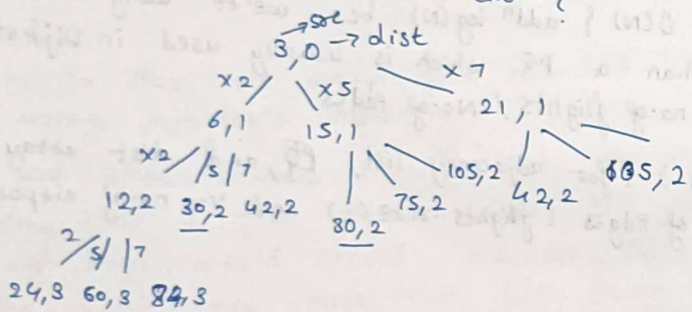
↑ ↑

1 2

{3,4,65} st = 7, end = 66175

1. 7 x 3 = 21
2. 21 x 3 = 63
3. 63 x 65 = 4095
4. 4095 x 65 = 266175 = 66175

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



mod 100000 - 0, 1, 2, 3 99999

nodes range always in betn 0 ... 99999

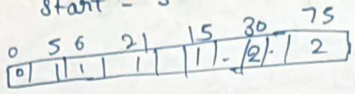
We update the distance array, whenever we find a lesser no. of multiplications in order to reach a node, whenever in this way, we reach end, the multiplications needed to reach it would always be minimum.

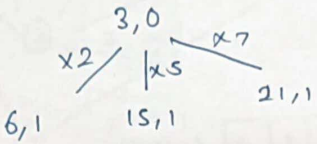
Queue - {steps, number} at each step, the steps are increasing so no need of PQ a Queue will do.

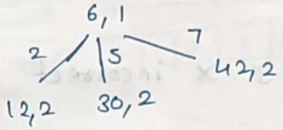
Distance Array - 0 ... 99999 marked as ∞

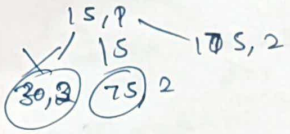
Start and End

start = 3 end = 75 mod = 100000 arr = [2,5,7]









dist, number steps
2,
2, 75
2, 42
2, 30
2, 12
1, 21
~~1, 15~~
~~1, 6~~
~~0, 3~~

Queue

node = 75 == end

stop return 2.

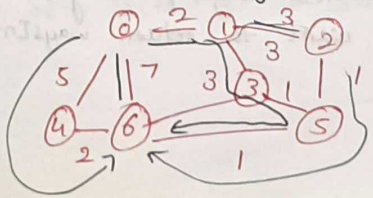
Time Complexity - O(100000 * N) where '100000' are the total possible numbers generated by multiplication (hypothetical) and N = size of the array with no.s of each node could be multiplied.

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-direcn roads. roads[i] = [ui, vi, timei] means that there is road between intersections ui and vi that takes timei 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.



7

0 → 1 → 3 → 6

2 3 3 = 8

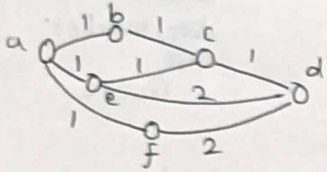
0 → 6 0 → 4 → 6

0 → 1 → 2 → 5 → 6

2 3 1 1

but, but

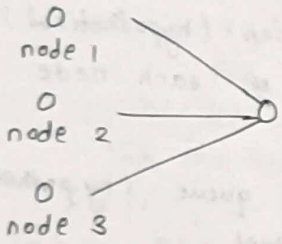
total = 3



min = 3

total = 3 ✗ incorrect

a → b → c → d (3)
a → e → c → d (3) total = 4 ways
a → e → d (3)
a → f → d (3)



node 3 (total = 3)

waysInode] = ways(n1) + ways(n2) + ways(n3)

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.

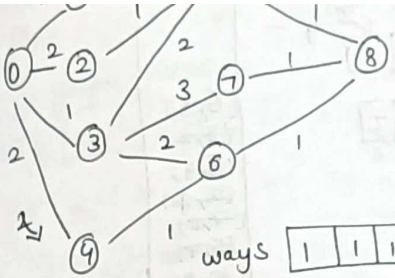
Initial Configuration:

Priority Queue - {dist, node}

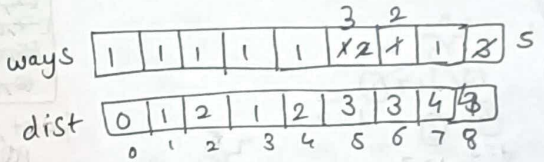
Distance Array - minimum distance from ~~start~~ node to current node

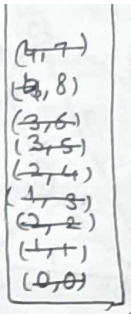
Src Node and End Node

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

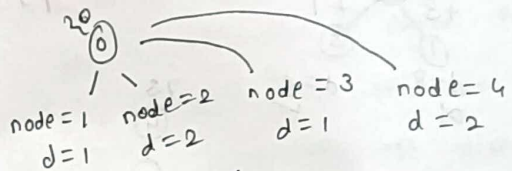


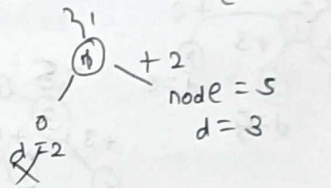
src = 0
end = 8

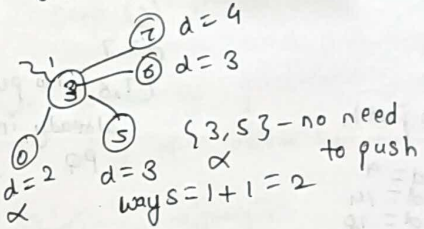


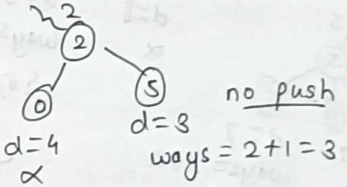


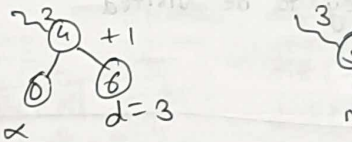
dist, node (min-Heap)

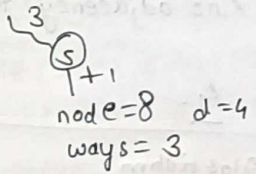


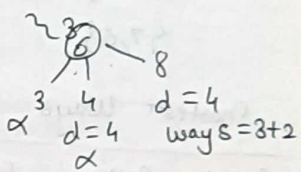


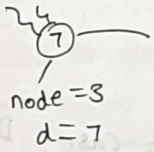












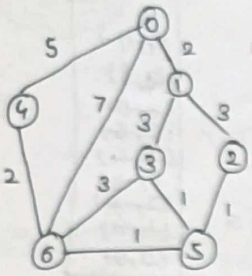
node = 8
d = 5 > 4 no need to do anything
(not shortest)

ans = d[8] = 5

TC → O(ElogV) { As we are using simple Dijkstra }

SC → O(N) { for dist array + ways array + approx. complexity for PQ }

one more dry run -

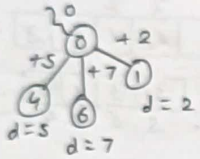


ways | 1 | 1 | 1 | 1 | 1 | 2 | 2

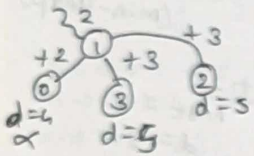
dist | 0 | 2 | 5 | 5 | 5 | 6 | 7

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

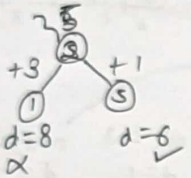
{dist, node}
PQ (min heap)



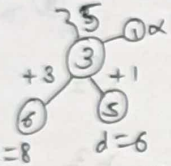
d=2



d=4 ∝ d=5 d=5



d=8 ∝ d=6 ✓



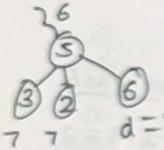
d=8 ∝ d=6

ways = 1+1 = 2

{6,5} ∝ no push

d=7

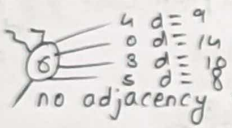
(7,6) no push already in PQ



d=7 ∝

ways = 2+2 = 4

{7,6}



4 d=9
0 d=14
3 d=18
5 d=18

no adjacency to be visited

Shortest ways are 4.

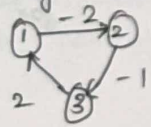
G-41 : Bellman-Ford Algorithm

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).

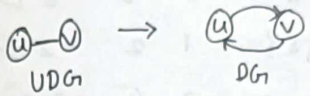
Negative cycle



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



UDG → DG

Algorithm - Edges can be given in any order

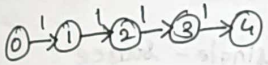
Step 1. Relaxation of edges N-1 times

Step 2. N^{th} time relaxation for -ve cycle and return 1 else 0.

Step 3. return shortest distances

Two follow-up questions

1. Why N-1 iterations ?

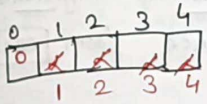


checks dist[3]+1 < dist[4]

dist(2)+1 < dist(3)

dist(1)+1 < dist(2)

dist(0)+1 < dist(1)



Given

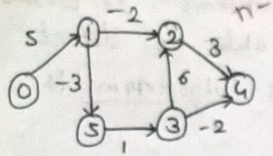
u	v	wt
3	4	1
2	3	1
1	2	1
0	1	1

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 N^{th} relax^n, if found that further relaxation is possible then we conclude that DG has -ve cycle.

Initial Config - distance array - dist[src] = 0



n-1 = 5

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

Shortest distance ⇒
0 5 3 3 1 2

src = 0

dis{3} + 6 < dis{2}
dis{5} + (-2) < dis{3}
dis{0} + 5 < dis{1}
dis{3} - 3 < dis{5}
dis{1} - 2 < dis{2}
dis{3} - 2 < dis{4}
dis{2} + 3 < dis{4}

u	v	wt
3	2	6 ✓
5	3	1 ✓
0	1	5 ✓
1	5	-3 ✓
1	2	-2 ✓
3	4	-2 ✓
2	4	+3 ✓

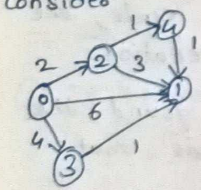
TC → O(V×E) where, V = no. of vertices & E = no. of edges

SC → 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,

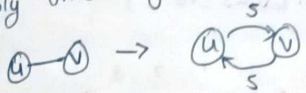
(0→1) = 6
(0→2) + (2→1) = 5
(0→3) + (3→1) = 5
(0→4) + (4→1) = 4
(0→2) + (2→4) = 3 (path from 0→4)

Thus, we can derive the following formula

matrix[i][j] = min (mat[i][j], matrix[i][k] + mat[k][j]

where i → src node
j → dest. node
k → node via which we are reaching from i to j

To apply this algo to UG

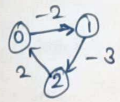


UG to DG

* Using adjacency matrix for storing
* dist[i][i] = 0 (0 ↺ = 0)

Follow-up questions

Negative Cycle:



Sum: x + (-x) + (-3) = -3 < 0
whereas 0→0 should be zero thereby, you say it's a 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 → O(VE log V)

for checking -ve

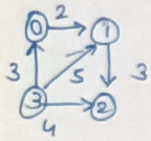
```
for (int i=0...n
        j=0...n
    if( mat[i][j] < 0)
        "negative"
```

TC → O(n³) ... three nested loops
SC → O(n²) .... storing adjacency matrix.

Pseudo Code -

```
for ( via = 0 ... < n )
    for ( row < n )
    for (col < n )
    matrix [row] [col]
= min ( m[row] [col], m[row] [via] + m [via] [col] )
```

→ $O(n^3)$ (Brute Force)



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

via 0 →

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

↓ via 1

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

0,2 ↳ 0,1 + 1,2

via 2 ←

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

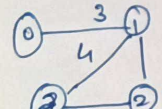
↓ via (3)

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

1,0 2,3 3,0 1,2 2,3 3,1 1,3 + 3,2

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

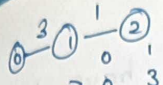
You need to find out a city with smallest no. of cities that are reachable through some path and whose distance is at most threshold distance, If there are multiple cities, our answer will be city with greatest number.



	0	1	2	3	
0 →	0	3	4	5	→ cnt = 3
1 →	3	0	1	2	→ 4
2 →	4	1	0	1	→ 4
3 →	5	2	1	0	→ 3

ans = 3

threshold = 2



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

ans = 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 mat[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.

Base 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.

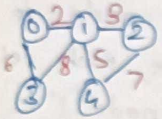
If we need to find out the no. of adjacent cities as well we need to return cntCity - 1. This is bcz we have included the node itself. 0 → 0 1 2 (0 itself which will be always 0 and ≤ threshold)

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

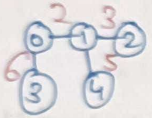
SC → $O(V^2)$ - where V - no. of vertices, adjacency matrix

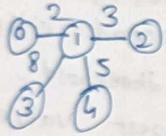
G-44: Minimum Spanning Tree (MST)

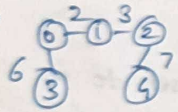
A spanning tree is a tree in which we have N nodes (i.e. All the nodes present in the original graph) and N-1 edges and all nodes are reachable from each other.

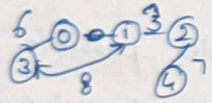


→ A graph may have more than one spanning tree.

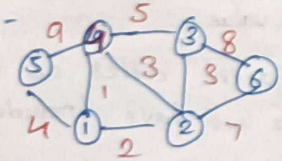
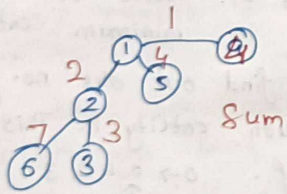
ST1 (Min)  Sum of edge weights = 2+3+11 = 16

ST2  Sum EDW = 18

ST3  Sum = 18 (min = 16)

ST4  Sum = 24

The minimal sum of edgeweight gives MST

Eg-  MST →  Sum = 17

To find MST we use ① Prim's Algorithm (G-45)
② Kruskal's Algorithm
↳ Disjoint Set (first this) then

G-45: Prim's Algorithm → MST

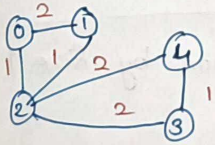
Initial Config: PQ (min-heap) {dist, node}
visited array
MST (pairs) if we need MST edges
Sum variable

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.

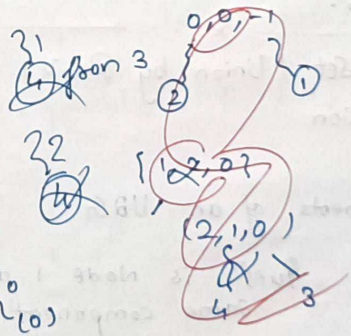
Note: If we only need the sum then
→ we need not use triplet of {dis, {node, parent}}
we will just use pair in the format {edw, node}
Basically, we don't need parent node.
→ we need not use MST array in whole algo as well.

Edges - 5 sum = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4 5



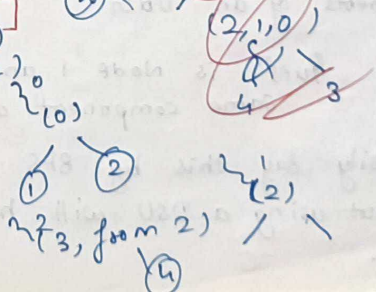
MST- [~~(0,0)~~ (4,3) (2,0) (1,2) (3,2)]

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

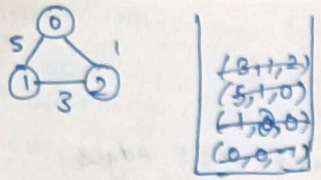


(1,4,3) ~~(1,4,3)~~ (2,3,2) ~~(2,3,2)~~ (2,4,2) ~~(2,4,2)~~ (1,1,2) ~~(1,1,2)~~ (1,2,0) ~~(1,2,0)~~ (2,1,0) ~~(2,1,0)~~ (0,0,-1) ~~(0,0,-1)~~ (4,3,0) ~~(4,3,0)~~ (2,1,0) ~~(2,1,0)~~ (3,-1,-1) ~~(3,-1,-1)~~

{dist, node, parent} PQ



1 from 2
2 from 0
3, from 2
4



(3,+,2)
(5,+,0)
(1,3,0)
(0,3,1)

sum = 0 + 4

vis

1	1	1
0	1	2

MST
[(2,0),(1,2)]

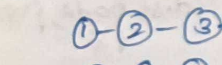
2 → 0 from -1
2 → 1
1 → 2 from 0
1 → 3
3 → 1 from 2
5 → 1 from 0 (already visited)

TC → O(ElogE) + O(ElogE) ~ O(ElogE)
The max. size of PQ can be E at most E iterations the pop takes log E ... this gives (E*logE). 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 ~ (ElogE)

SC → O(E) + O(V) ~ E - edges, V - nodes/vertices
O(E) for PQ → for visited array O(V-1) ~ O(V)
If we need to get mst, then to store edges.

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

Why DSU?
Given two components of an UBG



Ques" is Node 1 and 5 from the same component or not?

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)

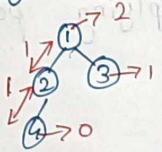
(1,2)
(2,3)
(4,5)
(6,7)
(5,6)
(3,7) ✓ here we conclude 1 & 7 are from same component

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:

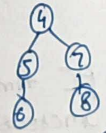
Rank:



rank{1} = 2
rank{4} = 0

distance betn furthest leaf and current node

Ultimate parent (topmost node or boss)

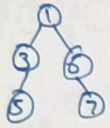


In this parent[6] = 5 but, ultimate parent[6] is 4.

Algo: Pre-Config: rank array (initially all 0)
parent array (initially parent[i] = i)

Steps
1. For two nodes (u and v) then, find the ultimate parent pu and pv.
2. After that, we find rank of pu and pv.
3. We connect smaller rank to bigger rank.

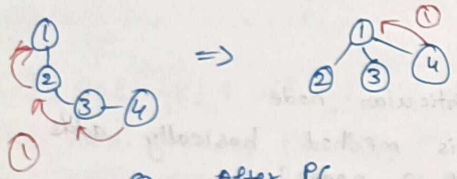
Observation 1: Why we need the ultimate parents.



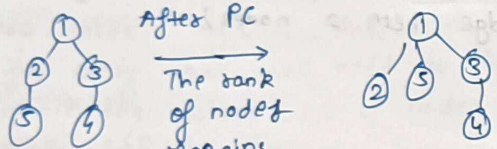
So, 5 & 7 have parent as 3 & 6, but their ultimate parent is 1.

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

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



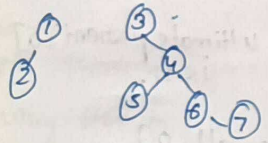
$O(4\alpha) \cong O(1)$



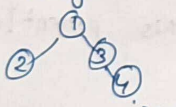
After PC → The rank of node remains same

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

Follow-Up Question:
Why smaller to larger rank?

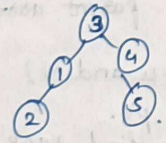


Case 1: Larger to smaller



So, PC time also increases

Case 2:



now, PC time is relatively lesser

Smaller to Larger
→ Shrinking height of graph
→ reduces time complexity

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

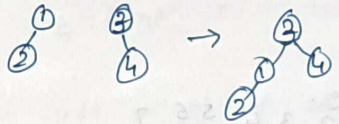
Union by Size

size array: initialized to one

parent array: parent[i] = i

Steps → 1. pu, pv (ultimate)
2. size[pu] and size[pv]
3. finally we connect ultimate parent with smaller size to larger size. But if equal we connect any to any.

Mean While we also increase the size where it's getting attached



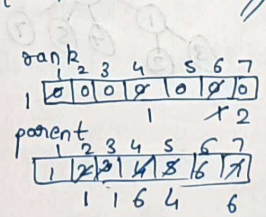
size[3] = 4

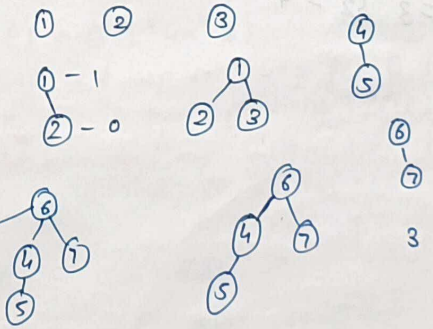
Note: 1. seems much more intuitive than UBR as rank gets distorted after PC
2. findPar() remains same

TC → $O(4\alpha)$ $\alpha \sim 1$, $O(1)$ so, $4\alpha \sim 1$

Constant Time $O(1)$ //

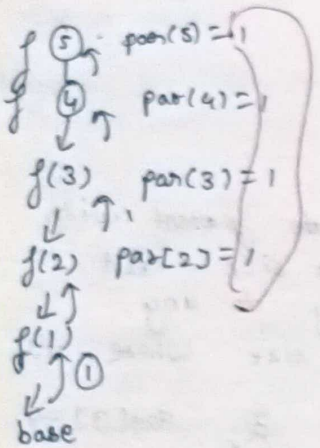
(1,2) ✓
(2,3) ✓
(4,5) ✓
(6,7) ✓
(5,6) ✓
(3,7)

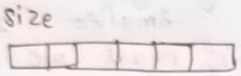




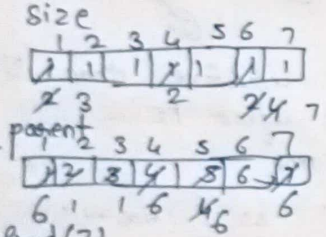
Path Compression

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



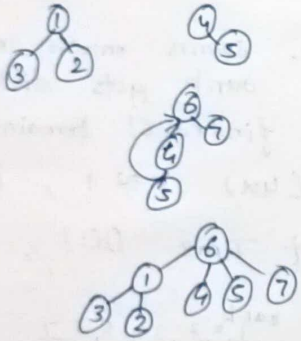


(1,2)
(2,3)
(4,5)
(6,7)
(5,6)
(3,7)



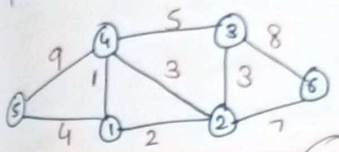
find(3) → ① size = 3

find(7) → ⑥ s2 = 4

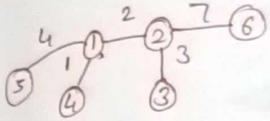


Gr-47: Kruskal's Algorithm – MST

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



Sum MST = 8 + 7 + 8 + 10 = 17 → 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.

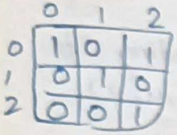
Note: As graph contain bidirectional edges, we can get a single edge twice in our array (wt, u, v) and (wt, v, u) but D&U will automatically discard the duplicate

TC → O(N+E) + O(E log E) + O(E * 4α * 2)
{ for extracting edge from adjacency list } { sorting array acc. to weight } { loop continues for E times, two operations Par() Union() each takes 4α so 4α * 2 ≈ 4 * 2 ≈ (E * 8) }

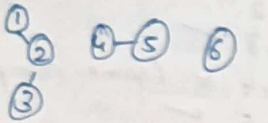
SC → O(N) + O(N) + O(E)
{ parent and size } { storing edges }

Number of Provinces

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



Ans = 2



ans = 3

findP(6) = 6
findP(7) = 6

findP(1) = 1
findP(2) = 1
findP(3) = 1

findP(4) = 4
findP(5) = 5

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

→ Apply DSU on edges
→ Count bosses and return

TC → O(V²*4α) + O(V*4α) ≈ O(V²)
(DSU) (counting bosses)

SC → O(2N)
(parent, size)

G-49: Number of Operations to make network connected

minimum = 2 but extra edges from the graph not imaginary

□ □ □ □ → □-□-□-□

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

□ - for single, 0 edges

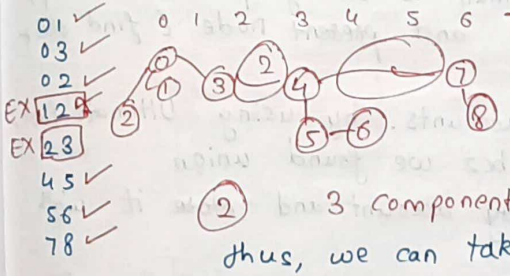
thus, If we have nC-1 extra edges (atleast) then we can connected with nC-1 operations.

→ 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 >= nC-1 ? nC-1 : -1

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



thus, we can take out those edges

TC → O(E*4α) + O(N*4α)
{Union and extra edges} {for counting number of components}

SC → O(2N) where, N = 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.

['John', 'J1@com', 'J2@com', 'J3@com'],
['John', 'J4@com'],
['Bru', 'Bru1@com', 'Bru2@com'],
['John', 'J1@com'],
['Bru', 'Bru2@com'],
['mary', 'mary1@com']]

→ John - J1, J2, J3, J4
John - J4, Bru - Bru1, Bru2
Mary - 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 UltimatePar() method for an email we found union
→ Sort the emails for every account and store it with names.

John, J1@com, J2com, J3com 0
John, J4com 1
bru, bru1com, bru2com, bru3com 2
mary, m1com 3
bru, bru2com, bru8com 4
John, J2@com 5

J1 → 0
J2 → 0
J3 → 0
J4 → 1
b1 → 2
b2 → 2
b3 → 2
b6 → 4
m1 → 3

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

finPar(4) = 2

0 → [J1@com, J2@com, J3@com]
1 → [J4@com]
2 → [b1@com, b2, b3, b6]
3 → [m1]
4 →
5 →

TC → O(N+E) + O(E*4α) + O(N*(ElogE + E)) where, N - no. of indices, E - no. of emails
(visiting, merging, storing)

SC → O(N) + O(N) + O(2N) ≈ 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.

n=4
m=5
k=4

[1,1] → 1
[0,1] → 1
[3,3] → 2
[3,4] → 2

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

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

-1 -m +1
+m

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

Observ^n 2: Optimizing repeating operations by an visited array

Observ^n 3: How to connect cells ?

We number from 0 to n*m - 1

5 X 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)

row * m + col

Total no. of columns

eg. (2,1)

2 x 5 + 1

= 11

Observ^n 4: How to count No. of Islands ?

If cell is unvisited, we mark 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 (nrow, ncol) to adjComponent by above formula.

TC → O(Q * 4α) where, Q - no. of Queries

SC → O(Q) + O(N*M) + O(N*M)

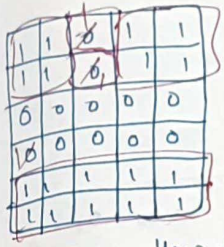
O(Q) → answer

O(N*M) → 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.

→ dynamically (real time) graph - DSU

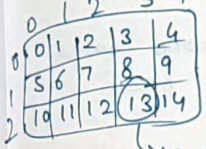


4+4+1 = 9

1+10 = 11

} max (9,11) = 11

Observ^n 1: How to connect them in same group ?



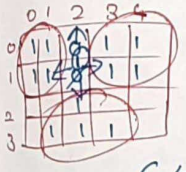
(row, col) → number of Node

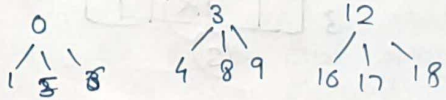
(row * columns + col)

(2,3) → (2x5+3) = 13

because DSU forms a union of single node not a (i,j) pair

To put 0 and get sizes of it's adjacent Group of Islands





6 ← 7 → 8

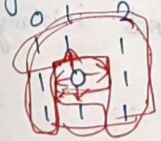
↓

12

so it gets size of ultimate parent

1 + size[UPar(8)] + size[findP(12)] + size[findP(12)] = 1 + 4 + 4 + 4 = 13

But there's an edge case



3 ← 4 → 5

↓

7

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

P(5) → 0 P(3) → 0 P(1) → 0

P(7) → 0

so we get 1 + size[0]

1 + 8 = 9

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.

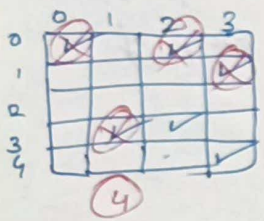
TC → $O(n^2 \times 4) + O(n^2 \times 4 \times 4\times) \sim O(n^2)$

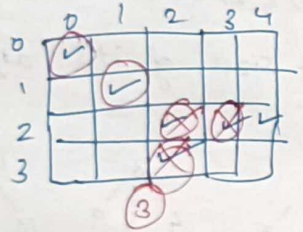
SC → $O(2n^2)$ ... parent and size

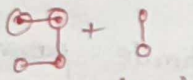
G-53: Most Stones Removed with same row or column

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.



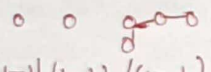




$(4-1)+(2-1)$

$(3+1)$

$= 4$



$(1-1)+(1-1)+(4-1)$

$0+0+3 = 3$

So, for components

C_1 C_2 C_3 C_4

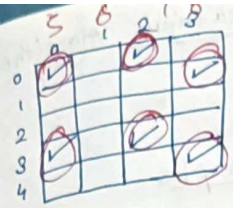
(x_1) (x_2) (x_3) (x_4) — stones removed initially

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

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

$= (n-k)$

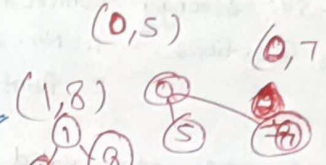
Ans = Stones − no. of components



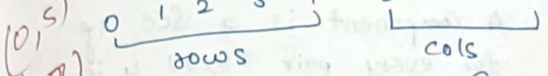
stones = 6

nC = 2

6 − 2 = (4)



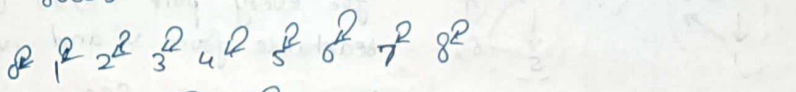
(0,5) (0,7) (1,8) for nodes in DS



(0,5) (0,7)

0 1 2 3 4 — rows

5 6 7 8 — cols



5 7 3 8 4

parent[i] == i && size > 1

How to connect →

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+5) and (2+5) 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.

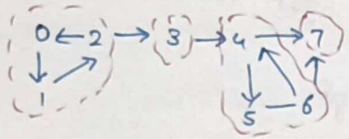
TC → O(N) where, N − Total no. of stones

SC → O(2 * (maxRowIndex + maxColIndex)) for parent and size array.

G-54: Strongly Connected Components - Kosaraju's Algorithm

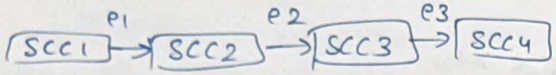
Two Questions - 1. No. of SCCs
2. Print SCCs

SCC's are only valid for directed graphs

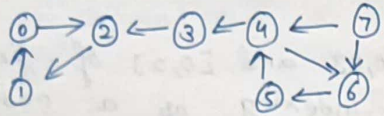


A component is a SCC if for every pair (u,v) u is reachable from v and vice-versa.

By defn, single vertex is also a SCC.

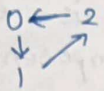


What If we reverse the edges,



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 SCCs.

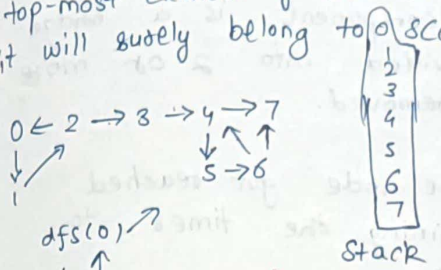
But we don't know SCCs, so we reverse all edges.

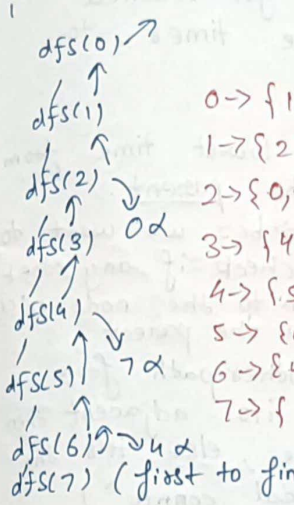
But  transforms to  so, reversing edge doesn't affect SCCs.

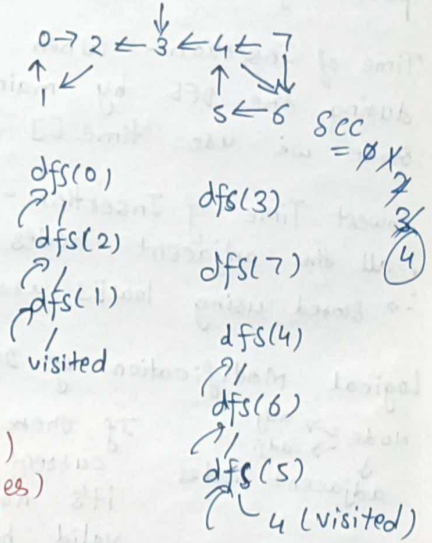
Algorithm -

(•) Sort all the nodes acc. to finishing time
(•) Reverse edges of entire graph
(•) Perform DFS and count no. of DFS calls to get the no. of SCCs

Note: Sorting of the 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.







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

TC -> O(V+E) + O(V+E) + O(V+E) ~ O(V+E)
- O(V+E): Simple DFS (sorting)
- O(V+E): reversing edges
- O(V+E): final DFS

SC -> O(V) + O(V) + O(V+E)
- O(V): stack
- O(V): visited array
- O(V+E): reversed 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.
↳ bcz we want to check if any other path to the node exists via the parent
∴ stored using low[] array.

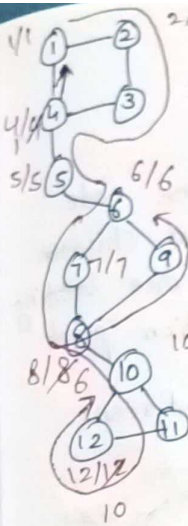
Logical Modification of DFS

Node → adj
→ adj.
↓
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 connec^{n})

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] )
}
```



T/L
8 8/6
10 10/10

T=8 < L(10)
(Bridge)

T/L
5 5/5
6 6/6 → lowest
5<6
(Bridge)

4 4/4
5 5/5
4 < 5 (Bridge)

5 can't reach 4 bcz lowest Time of Insertion is 5.

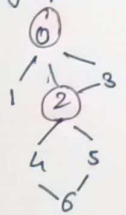
[time of Insertion[node] < low[adjNode] Time of Insert^{n}]

TC → O(V+2E) → DFS

SC → O(V+2E) + O(3N)
↓ Adjacency List
↳ vis, time, lowest Time of Insert^{n}

G-56: Articulation in Graph

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



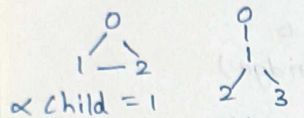
(•) Time of Insertion
(•) Lowest Time of Insertion: min possible from the adjacent nodes excluding parent and visited nodes

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:

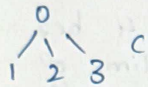
if (low [it] >= tin[node] && parent != -1)

For starting node:



∝ child = 1 ∝ child = 1

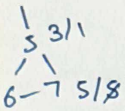
We check the childs of the node. If more than 1 child and children are not connected, then the starting node is definitely an articulation point



child = 3 ✓ So, increment child, if adjacent is not visited

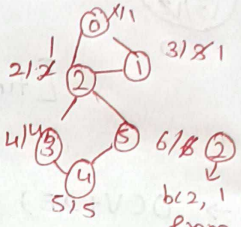
Also, if adjacent is visited, we update

low[node] = min(low[node], tin[it])

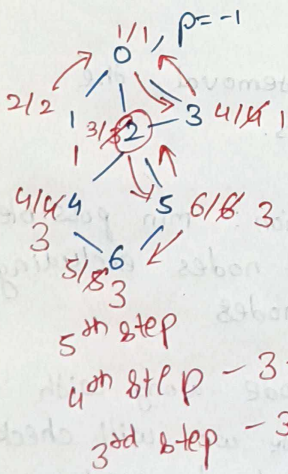


so if 5 is removed

~~7~~ can reach lowest to 3 not above it.



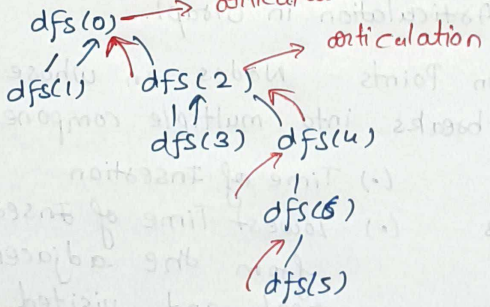
b(2, 1 is from above if 2 is removed you can't go back



5th step

4th step - 3 → 4

3rd step - 3



(child = 2, parent == -1) articulation

articulation

TC → O(V+2E) - dfs

SC → O(3V) - tin, low and vis

1. max no of edges with n nodes?
ans: n*(n-1)/2 since every node can connect with other nodes.
2. find total number of graphs with n nodes?
ans: number of edges can be from 0 to max no_of_edges with n nodes.
if nodes = 4,
edges can be
0 1 2 3 4 5 6
number of graphs with edges
6c0 6c1 6c2 6c3 6c4 6c5 6c6
or using formula =
2^ max_no_of_edges.