

GRAPH

Handwritten Notes of Striver(TUF) Playlist

by: Aashish Kumar Nayak

NIT Srinagar



AASHISH KUMAR NAYAK



aashishkumar.nayak

- (1.) Introduction to graph | Types | Conversion from word.
- (2.) Graph representation | Two ways
- (3.) What are connected components.
- (4.) BFS (Breadth first search)
- (5.) DFS (Depth first search)
- (6.) Number of provinces | connected components
- (7.) Number of islands | Number of connected components in a graph
- (8.) Flood fill Algorithm
- (9.) Rotten oranges
- (10.) Detect cycle in an undirected graph using BFS
- (11.) Detect cycle in an undirected graph using DFS
- (12.) Distance of nearest cell having 1 in 0/1 matrix.
- (13.) Surrounded Regions | Replace 0's with X's.
- (14.) Number of enclaves | Multi-source BFS.
- (15.) Number of distinct island | constructive thinking + DFS
- (16.) Bipartite graph | BFS
- (17.) Bipartite graph | DFS
- (18.) Detect cycle in a directed graph using DFS
- (19.) Find eventual safe states - DFS
- (20.) Topological sort Algorithm | DFS
- (21.) Kahn's Algorithm | Topological sort Algorithm | BFS
- (22.) Detect a cycle in a directed graph | Topological sort | Kahn's algorithm | BFS
- (23.) Course schedule I and II | Pre-requisite Task | Topological sort
- (24.) Find eventual safe states | BFS | Topological sort
- (25.) Allen Dictionary | Topological sort
- (26.) Shortest path in Directed Acyclic graph | Topological sort.
- (27.) Shortest path in undirected graph with unit weights
- (28.) Word Ladder - I | shortest paths
- (29.) Word Ladder - II | shortest paths
- (30.) Dijkstra's Algorithm - using priority queue
- (31.) Dijkstra's Algorithm - using set
- (32.) Dijkstra's Algorithm why PQ and not Q, intuition | Time complexity Analysis
- (33.) Print shortest path - Dijkstra's algo
- (34.) Shortest distance in a binary maze
- (35.) Path with minimum Effort
- (36.) Cheapest flight with K stops
- (37.) Minimum multiplication to reach end
- (38.) No. of ways to arrive at destination
- (39.) Bellman Ford Algorithm
- (40.) Floyd Warshall Algorithm
- (41.) Find the city with smallest no. of neighbours at a threshold distance
- (42.) Minimum Spanning Tree - Theory
- (43.) Prim's Algorithm - minimum spanning tree
- (44.) Disjoint set union by rank | Union by size | Path compression
- (45.) Kruskal's Algorithm - minimum spanning tree
- (46.) Number of provinces - Disjoint set
- (47.) Number of operations to make network connected | DSU
- (48.) Account merge | DSU
- (49.) Number of Island-II - Online queries - DSU
- (50.) Making a large island - DSU
- (51.) Most stones removed with same row or column - DSU
- (52.) Strongly connected components - Kosaraju's Algorithm
- (53.) Bridges in graph - using Tarjan's algorithm of time in and low time
- (54.) Articulation point in graph

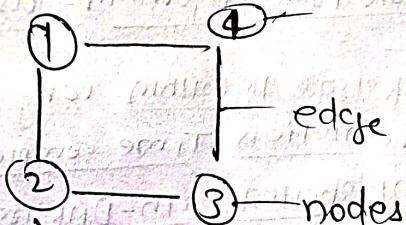
Vec 1

EN GRAPH

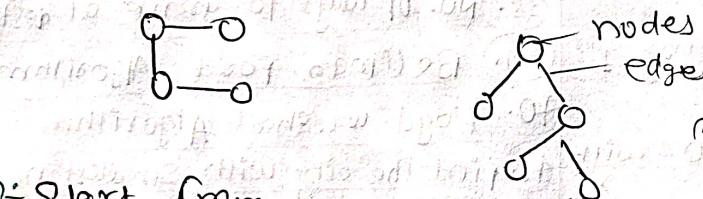
01-04-2023

There are two types of graph

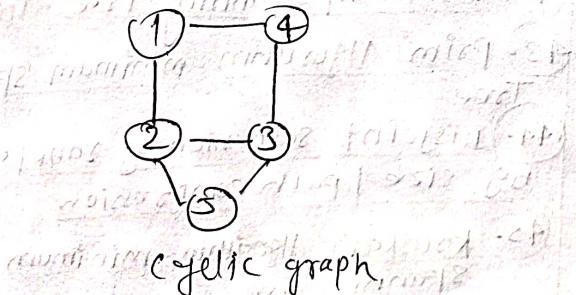
- ① undirected graph



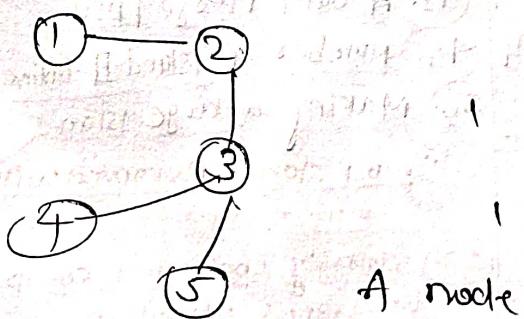
cycle in a graph



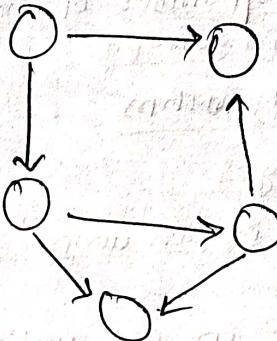
Defn: Start from a node and end at that node is called cycle.



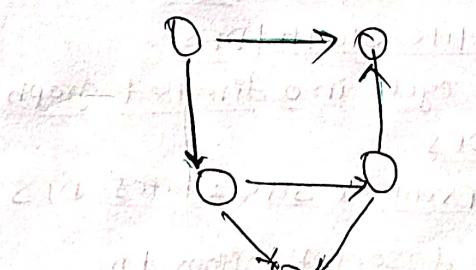
Path: Contains a lot of nodes and each of them are reachable.



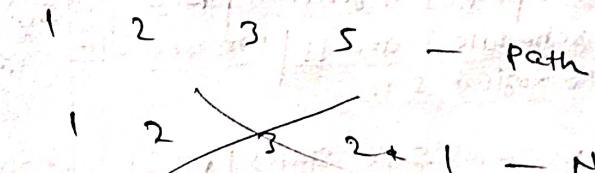
- ② directed graph



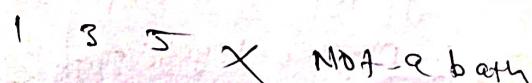
This is also graph



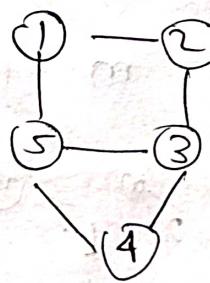
Directed Acyclic Graph (DAG)



can't appear twice in a path



Degrees on Graph



$$D(3) \rightarrow 3 \quad \text{for an } \cancel{\text{undirected}}$$

$$D(4) \rightarrow 2$$

$$D(5) \rightarrow 3$$

$$D(2) \rightarrow 2$$

$$D(1) \rightarrow 2$$

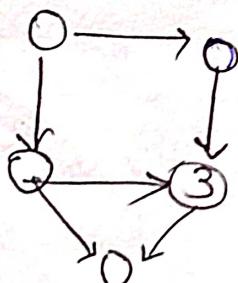
for an undirected graph the no. of edges that are attached to it is called degree.

$$\boxed{\text{Total Degree of a graph} = 2 \times \text{no. of edges}}$$

for directed graph

Indegree (node)

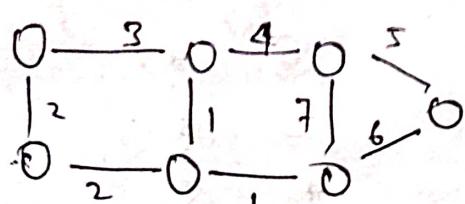
Outdegree (node)



$$\text{Indegree}(3) = 2$$

$$\text{Outdegree}(3) = 1$$

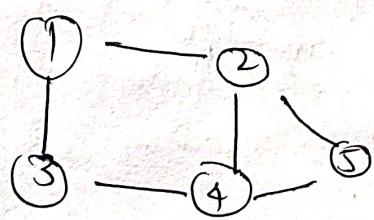
Edge weights



if weight is not assign
unit weight

Rec 2

Lec

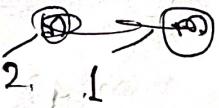


input

n-nodes, m-edges

5

6



m lines

↓
Represent
edges

[

1	3
2	4
3	4
2	5
4	5

5 6 ✓ m

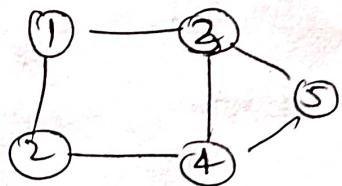
m lines

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

Two way to store

① Matrix

② List



① matrix

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

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

Space $n \times n$
Costly

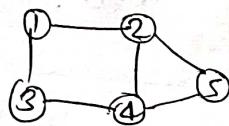
Code :-

```

int main()
{
    int n, m;
    cin >> n >> m;
    int adj[n+1][m+1];
    for (int i=0; i<m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u][v] = i;
        adj[v][u] = i;
    }
    return 0;
}

```

(ii) List :-



$1 \rightarrow \{2, 3\}$
 $2 \rightarrow \{1, 4, 5\}$
 $3 \rightarrow \{1, 4\}$
 $4 \rightarrow \{2, 3, 5\}$
 $5 \rightarrow \{2, 4\}$

Code :-

```

int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> adj[n+1];
    for (int i=0; i<m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    return 0;
}

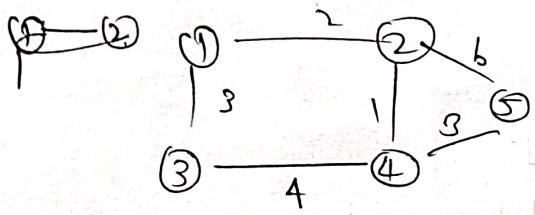
```

for directed graph

$\text{adj}[u].push_back(v);$

T.S.C. : $O(E \log V)$

Weighted graph :-



Matrix

0	1	2	3
0			
1		2	
2	2		
3			

$\text{adj}[v][v] = \text{weight}$.

list

we will store pair.

0

1

2

3

4 → {2, 3, 5}

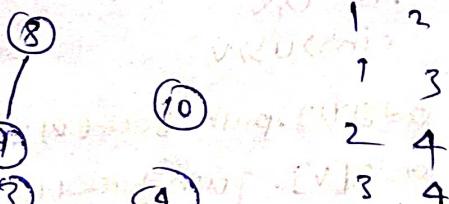
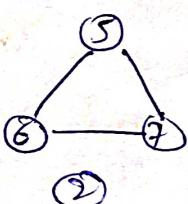
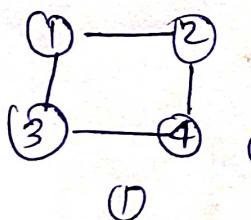
5

instead of this store this
{(2, 1), (3, 4), (5, 3)}

3. rect

Connected components :-

$N=10, M=8$



1	2
1	3
2	4
3	4
5	6

9 different components of a single graph

it can be also 4 different graph.

But according to input/question it is a single graph.

vis []

If we start from 1st component then we will never reach to the 2nd component so for that we will use something called visited array.

vis



N=11

for ($i = 1 \rightarrow 10$)

if (!vis[i])

traversal(i);

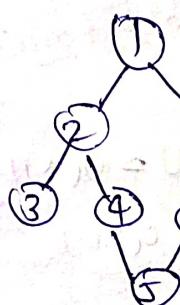
for any traversal.

if node not visited.

// then call traversal algo from this node.

A. Lees.

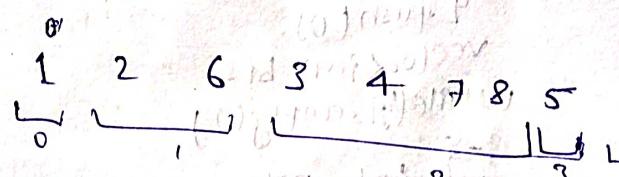
BFS (Breadth First Search)



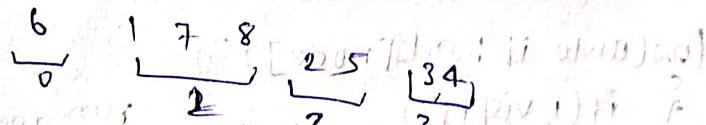
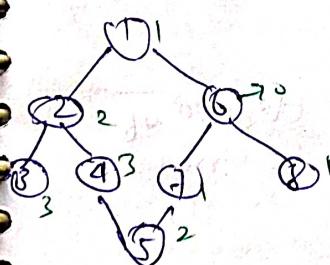
level

Breadth

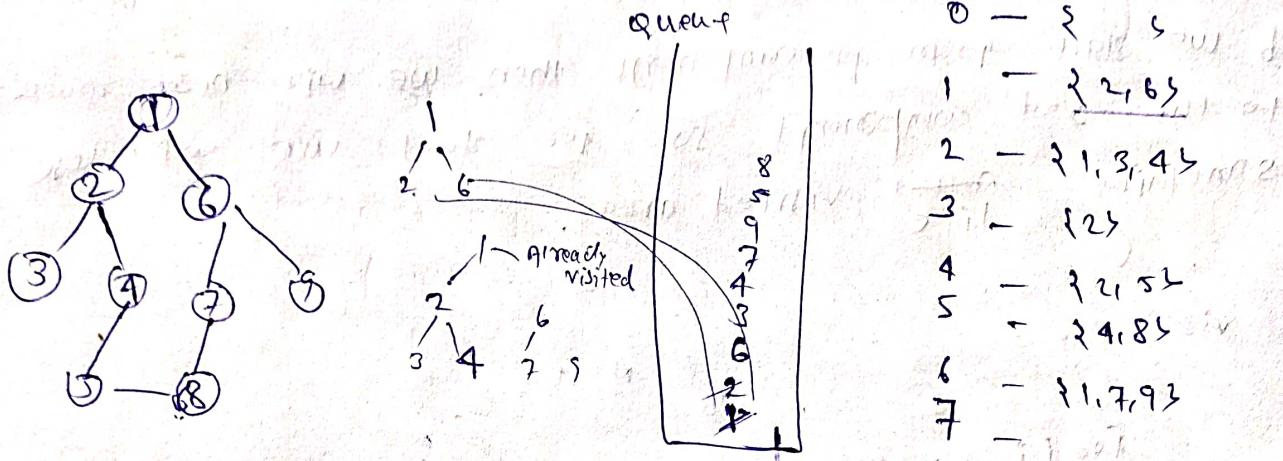
levelwise



let say 6 is starting node



pt is same for like level order traversal. not pre-order



vis [0|1|1|1|1|1|1|1|1|1|1|1]

Output
1 2 6 3 4 7 9 5 8
 |
 0 1 2 3 4 5 6 7 8

Code :-

vector<int> bfsOfUngraph(int v, vector<int> adj[])

{ int vis[n] = {0};

vis[0] = 1;

queue<int> q;

q.push(0);

vector<int> bfs;

while(!q.empty())

{ int node = q.front();

q.pop();

~~bfs.push_back()~~

bfs.push_back(node);

for(auto it : adj[node])

{ if(!vis[it])

{ vis[it] = 1;

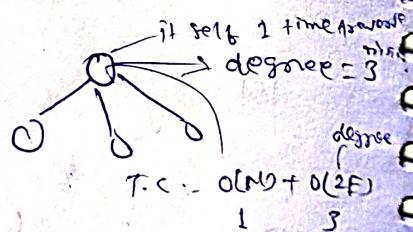
q.push(it);

~~if(vis[i])~~ i + 1

S.C. - $O(3N) \approx O(N)$

T.C. - $O(N) + O(2E)$

$2E$ = Total degree



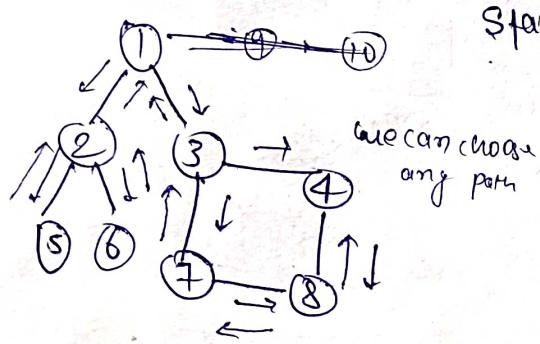
T.C. - $O(N) + O(2F)$

run for degree of node

return bfs;

5- Lec 6

DFS (Depth First Search) Recursion



Starting node = 1

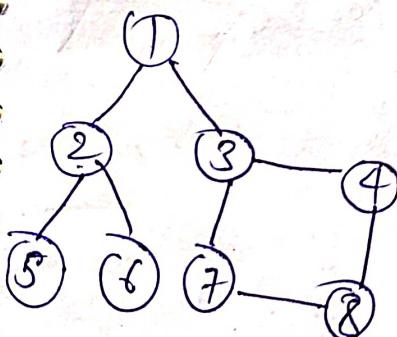
1 2 5 6 3 7 8 4

Main thing is, traversing in depth.

If Starting node = 3

Recursion is an algo which goes till depth calculate and returns back.

So we use recursion in DFS.



dfs(1)
dfs(2)
dfs(3)

vis []
Adjacency list :-

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

`vector<int> dfsOfGraph(int V, vector<int> adj[])`

```
{  
    int vis[v] = {0};  
    int start = 0;  
    vector<int> ls;  
    dfs(start, adj, vis, ls);  
    return ls;  
}
```

void dfs(int node, vector<int> adj[], int vis[], vector<int> &ls)

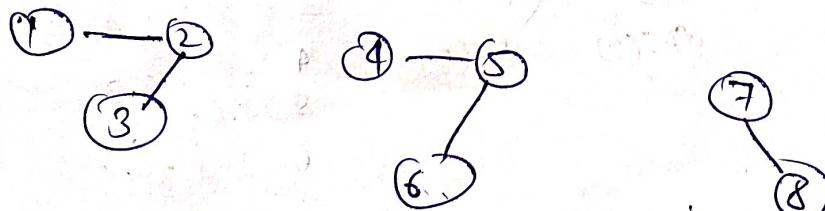
```
{  
    vis[node] = 1;  
    ls.push_back(node);  
    for (auto it : adj[node])  
    {  
        if (!vis[it])  
        {  
            dfs(it, adj, vis, ls);  
        }  
    }  
}
```

S.C. \rightarrow $O(N) + O(N) + O(N)$

$\approx O(N)$
Recursion
T.C. $= O(N) + O(2E)$
degree

For directed graph
T.C. $= O(E)$

Q. :- Number of provinces :-



No. of provinces = 3

From node 3 we can go to 2, 3 or 4. From 2 we can go to 1, 3, 4. From 3 \rightarrow 1, 2 so 1 \rightarrow 2 this is 1 province.

3	x
2	x
1	x

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

We will have multiple starting point

Start = 1/2/3

Start = 4/5/6

Start = 7/8

class

void

```
dfs (int node, vector<int> adjls[], int vis[])
{
    vis[node] = 1;
    for (auto it : adjls[node])
        if (!vis[it])
            dfs(it, adjls, vis);
}
```

T.C. $O(N) + O(V+2E)$
 $\approx O(N)$

S.C. $O(N)$

Recursion
space

int numProvinces (vector<vector<int>> adj, int v)

vector<int> adjls[v];

for (int i=0; i<v; i++)
 for (int j=0; j<v; j++)

```
        if (adj[i][j] == 1 && i != j)
            adjls[i].push_back(j);
            adjls[j].push_back(i);
```

int vis[v] = {0};

int cnt = 0;

for (int i=0; i<v; i++)

```
    if (!vis[i])
        cnt++;
        dfs(i, adjls, vis);
return cnt;
```

@Aashish Kumar Nayak

No. of component
Number of Islands

@Aashish Kumar Nayak

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

No. of Island

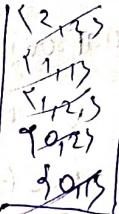
3. Starting Nodes

8 Island

using BFS

Starting → 3014

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



adjacency list

They can be neighbour

so we will go in all 8-direction.



Then after traversal the queue will be empty

Now starting point will change and process will be same.

so we will use three starting points.

for finding starting points we will traverse from

(0,0) to (n-1, n-1) and if we find 0 then ignore while

if we find them call traversal also.

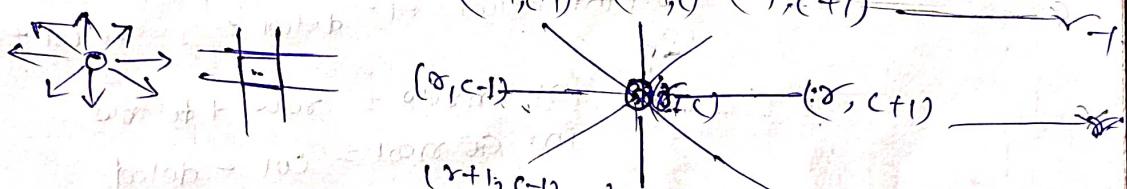
pseudo code :-

```
for(row → 0-n)
  {
    for(col → 0-m)
      {
        if(!vis[row][col])
          {
            bfs(row, col);
            count++;
          }
      }
  }
```

for BFS traversal

nodes.

we have to visit all the neighbour



for that we can write

a code

```
for(i = 1 to n)
  {
    for(j = 1 to m)
      {
        new-row = row + delrow;
        new-col = col + delcol;
      }
  }
```

```
Void bfs(int row, int col, vector<vector<int>> &vis,  
vector<vector<char>> &grid)
```

{

```
    vis[row][col] = 1;
```

```
    queue<pair<int, int>> q;
```

```
    q.push({row, col}); int n = grid.size();
```

```
    while(!q.empty()) { int m = grid[0].size();
```

{

```
        int row = q.front().first;
```

```
        int col = q.front().second;
```

```
        q.pop();
```

// traverse in the neighbours & mark them visited

```
        for(int delrow = -1; delrow <= 1; delrow++)
```

{

```
            for(int delcol = -1; delcol <= 1; delcol++)
```

{

```
                int nrow = row + delrow;
```

```
                int ncol = col + delcol;
```

```
                if(nrow >= 0 & nrow < n &
```

```
                ncol >= 0 & ncol < m & grid[nrow][ncol] == '1' &
```

```
                !vis[nrow][ncol])
```

{

```
                    vis[nrow][ncol] = 1;
```

```
                    q.push({nrow, ncol});
```

```
int numIslands(vector<vector<char>> &grid)
```

```
int n = grid.size();
```

```
int m = grid[0].size();
```

```
vector<vector<int>> vis(n, vector<int>(m, 0));
```

```
int cnt = 0;
```

```
for (int row = 0; row < n; row++)
```

```
    for (int col = 0; col < m; col++)
```

```
        if (!vis[row][col] && grid[row][col] == '1')
```

```
            cnt++;
```

```
bfs(row, col, vis, grid);
```

```
return cnt;
```

In matrix queue BFS

Time Complexity: $O(N^2) + O(N^2)$
Matrix Queue $\approx O(N^2)$

Space Complexity: $N^2 \times 9$

$N^2 + N^2 \times 9$
matrix 9
queue 9
BFS

$O(N^2)$

Lec 9

Flood Fill Algorithm

Ques

By DFS

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

sr, sc starting coordinate. DFS(2, 0)
initial = 2

new color = 3

DFS(1, 0)

DFS(1, 1)

DFS(2, 0)

DFS(2, 1)

DFS(2, 1)

We can choose BFS / DFS
any of them

But we choose DFS here for no reason

Initial color will be $\text{image}[sr][sc]$

Code :-

```
vector<vector<int>> floodFill (vector<vector<int>> &image,  
int sr, int sc, int newcolor)
```

```
{ int init_color = image[sr][sc];
```

```
vector<vector<int>> ans = image;
```

```
int delRow[] = {-1, 0, +1, 0};
```

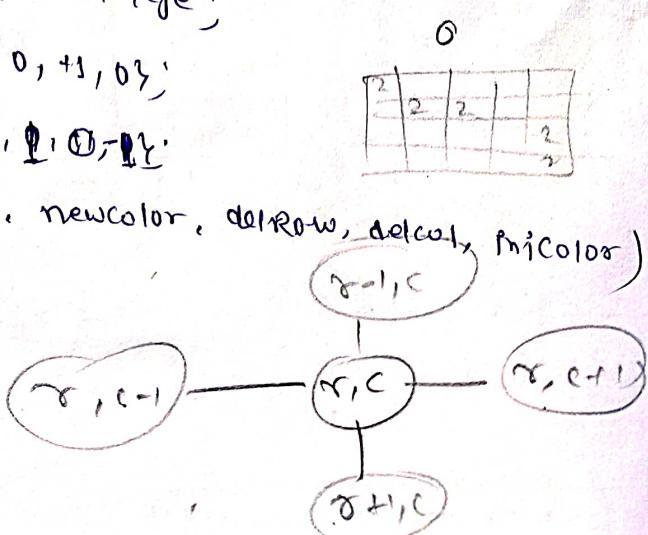
```
int delCol[] = {0, 1, 0, -1};
```

```
dfs(sr, sc, ans, image, newcolor, delRow, delCol, init_color)
```

```
return ans;
```

}

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



```
void dfs(int row, int col, vector<vector<int>> &ans, vector<
```

```
<vector<int>> &image, int delRow[], int delCol[])
```

```
ans[row][col] = newColor;
```

```
for (int i = 0; i < 4; i++) { int m = image[i].size();
```

```
int nrow = row + delRow[i];
```

```
int mcol = col + delCol[i];
```

```
if (nrow >= 0 & nrow < m & mcol >= 0 &
```

```
nrow < m & image[nrow][mcol] == newColor)
```

```
= initiator & ans[nrow][mcol] != newColor)
```

```
dfs(nrow, mcol, ans, image, newColor, delRow,
```

```
delCol);
```

matrix for every
 $(N \times M) \times 4$ 4 neighbour nodes

$$= (M \times N) + (N \times M) \times 4$$

call dfs
if

 $\approx O(N \times M)$

So C. $O(N \times M) + delRow$
 $\approx O(N \times M)$

5	9	2	9
5	3	6	5
8	0	9	0
5	0	0	0

Rotten Oranges

We have to find the minimum time to rotten all the oranges.

0	1	2
0	1	1
2	1	1

0 - Empty box

1 - Fresh orange

2 - Rotten orange

* In 1 unit of time if corn come with 4 oranges if available.

* If you can't rotten all oranges then return -1.

Sol) we have to visit some level nodes at same time & BFS is the only traversal algo which traverse level wise

If we use DFS

0	1	2
0	1	1
2	1	1

it will take 5 sec

But we have to do it in min time and that is only possible if we move in the neighbouring direction with queue

BFS simultaneously rotten

0	1	2
0	1	2
1	0	1

BFS

with time t_1

$$(2,0) + = 0$$

$$(0,2) + = 0$$

with time t_2

$$(2,1) + = 1$$

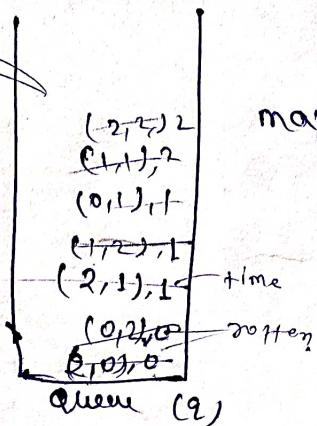
visited

0	1	2
0	2	2
2	2	2

ma

Max time = 2 sec

return max time, $(0, 0)$



max time = 2 sec

$(x-1, c)$

$(0, c)$

$(r+1, c)$

neighbour nodes

```
int orangesrotting(vector<vector<int>> &grid)
```

```
{
```

```
    int n = grid.size();
```

```
    int m = grid[0].size();
```

① Aashish Kumar Nayak

```
queue<pair<pair<int, int>, int> q;
```

```
int vis[n][m];
```

```
int cntfresh = 0;
```

```
for (int i=0; i<n; i++)
```

```
{ for (int j=0; j<m; j++)
```

```
{ if (grid[i][j] == 2)
```

```
{
```

```
    q.push({{i, j}, 0});
```

```
    vis[i][j] = 2;
```

```
}
```

```
else {
```

```
    vis[i][j] = 0;
```

```
if (grid[i][j] == 1)
```

```
{ cntfresh++;
```

```
}
```

```
}
```

```
}
```

```
}
```

```
int tm = 0;
```

```
int drow[] = {-1, 0, +1, 0};
```

```
int dcov[] = {0, +1, 0, -1};
```

```
int cnt = 0;
```

```
while (!q.empty())
```

```
{ int r = q.front().first.first;
```

```
int c = q.front().first.second;
```

```
int t = q.front().second;
```

```
tm = max(tm, t);
```

```
q.pop();
```

```
for (int i=0; i<4; i++)
```

```
{ int nrow = r + drow[i];
```

```
int ncol = c + dcov[i];
```

```
if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && vis[nrow][ncol] == 0
```

```
&& grid[nrow][ncol] == 1) { q.push({{nrow, ncol}, t+1});
```

```
vis[nrow][ncol] = 2; cnt++; }
```

T.C. $O(N \times M) + O(N \times M)$
vis usage

$\approx O(N \times M)$

T.C. $O(N \times M)$ + $O(N \times M) \times 4$
loop /
every node /
neigh bu

$\approx O(N \times M)$

```

if(cnt == cntFresh)
    return -1;

```

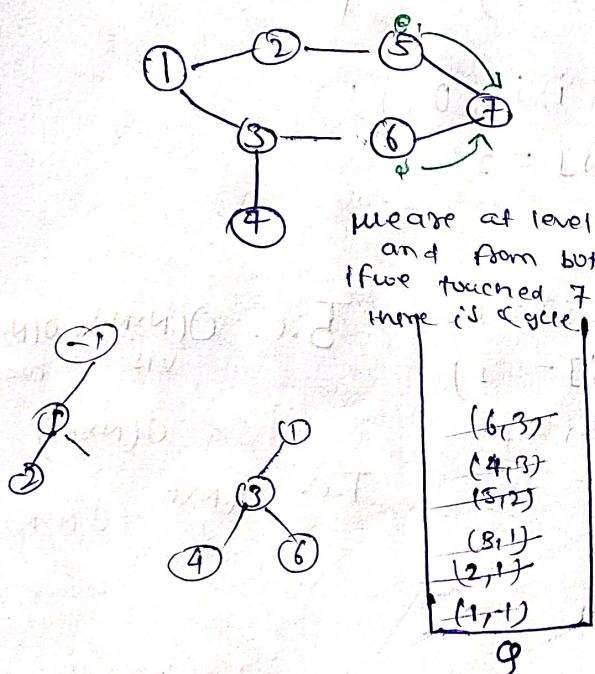
```

return tm;

```

Detect a cycle in an undirected graph :-

if you are starting traverse and you are able to come back at that particular node then there will be cycle in the graph.



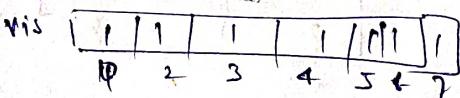
means at level 2 $\{5, 6\}$
and from both side
if we touched 7 it means

there is a cycle

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

using BFS

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



for Component graph

```

for (i = 1; i <= n; i++)
{
    if (!vis[i])
    {
        if (detectCycle(i) == true)
            return true;
    }
}
return false;

```

component
thing

$T.C. = O(N + 2E) + O(N)$

$S.C. = O(N) + O(N)$

where N = Vis

$\approx O(N)$

```

bool detect(int src, vector<int> adj[], int vis[])
{
    vis[src] = 1; // Node parent
    queue<pair<int, int>> q;
    q.push({src, -1});
    while (!q.empty())
    {
        int node = q.front().first;
        int parent = q.front().second;
        q.pop();
        for (auto adjacentNode : adj[node])
        {
            if (!vis[adjacentNode])
            {
                vis[adjacentNode] = 1;
                q.push({adjacentNode, node});
            }
            else if (parent != adjacentNode)
            {
                return true;
            }
        }
    }
    return false;
}

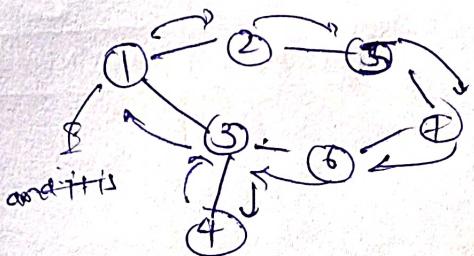
```

```

bool isCyclic(int v, vector<int> adj[])
{
    int vis[v] = {0};
    for (int i = 0; i < v; i++)
    {
        if (!vis[i])
        {
            if (detect(i, adj, vis))
                return true;
        }
    }
    return false;
}

```

Cycle detection in un-directed graph \rightarrow DFS



start from somewhere and if we got a node which is previously visited then we can say there is cycle.

and it should be different from parent of that node.

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

normal node
parent node

DFS(1, -1)

no need for further call

because we got true

we will do this call

after completing

the DFS(2, 1)

call

DFS(1, 1)
already but
visited parent is
same as node
so, it is not
cycle.

DFS(3, 6)

DFS(4, 3)

DFS(1, 3)

for (7 frame will be

zero call because it is parent)

return false,

3 is marked visited and also it is

not parent, so we got the

return true.

Adj List

1 - 2, 3

2 - 1, 5

3 - 1, 4, 6

4 - 3

5 - 2, 7

6 - 3, 7

7 - 5, 6

```

bool dfs(int node, int parent, vector<int>& vis[], vector<int>
adj[])
{
    vis[node] = 1;
    for (auto adjacentNode : adj[node])
    {
        if (!vis[adjacentNode])
        {
            if (dfs(adjacentNode, node, vis, adj) == true)
                return true;
        }
        else if (adjacentNode != parent)
            return true;
    }
    return false;
}

```

```

bool isCyclic(int v, vector<int> adj[])
{
    int vis[v] = {0};
    for (int i = 0; i < v; i++)
    {
        if (!vis[i])
        {
            if (dfs(i, -1, vis, adj) == true)
                return true;
        }
    }
    return false;
}

```

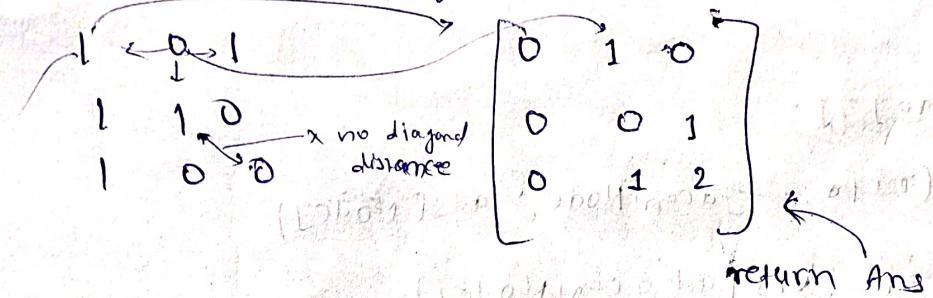
Recursion Vis

S.C. - $O(N) + O(N) \approx O(N)$

T.C. - DFS Traversal
 $O(N + 2E) + O(N)$
 $\approx O(N + 2E)$ for loop
 Not $O(N)$ but it will depend on comp.
 P.S. ! $\approx O(3)$ nonis = ~~is~~

Distance of nearest cell having 1

for this one it self



∴ This is something related to graph, and also we have to min kind min steps or min distance, it means it is a graph problem.

which algo - BFS comes in my mind.

Because it will simultaneously travels in all direction and always minimum count.

	0	1	2
0	02	01	20
1	01	10	01
2	0	0	0

vis

1	1	,
1	1	
1	1	

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

Q

$1 \rightarrow 0$ instead of
 $1 \rightarrow 1$

$1 \leftarrow 0$
 $1 \leftarrow 1$

$1 \rightarrow 0$

$$T.C. = O(m \times m) \times 4 \approx O(n \times m)$$

$$S.C. = O(n \times m)$$

```
vector<vector<int>> nearest (vector<vector<int>> grid)
```

```
{ int n = grid.size();  
int m = grid[0].size();
```

```
vector<vector<int>> vis(n, vector<int>(m, 0));
```

```
vector<vector<int>> dist(n, vector<int>(m, 0));
```

```
queue<pair<pair<int,int>,int>> q;
```

```
for (int i=0; i<n; i++)
```

```
{ for (int j=0; j<m; j++)
```

```
{ if (grid[i][j] == 1)
```

```
{ q.push({{i,j},0});
```

```
vis[i][j] = 1;
```

```
else
```

```
vis[i][j] = 0;
```

```
int delrow[] = { -1, 0, +1, 0};
```

```
int delcol[] = { 0, +1, 0, -1};
```

```
while (!q.empty())
```

```
{ int row = q.front().first.first;
```

```
int col = q.front().first.second;
```

```
int steps = q.front().second;
```

```
q.pop();
```

```
dist[row][col] = steps;
```

```
for (int i=0; i<4; i++)
```

```
{ int nrow = row + delrow[i];
```

```
int ncol = col + delcol[i];
```

```
if (nrow >= 0 & nrow < n & ncol >= 0 & ncol < m & vis[nrow][ncol] == 0)
```

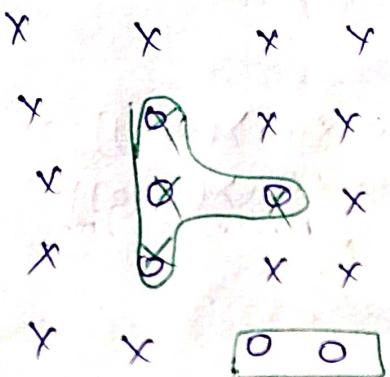
```
{ vis[nrow][ncol] = 1;
```

```
q.push({{nrow, ncol}, steps+1});
```

```
}
```

```
return dist;
```

Replace O's with X's



only those ~~O's~~ which are surrounded by X in top, down, left, right that can be converted into X.
rest are not converted.

Observation : If someone is on the boundary they,

can't be part of a set connected to a boundary so cannot be converted.

DFS

① go to the boundary and bind it if there is zero
If we got zero then call the DFS for ~~from~~ that index

0	1	2	3	4	NIS
X	X	X	Y	X	0 0 0 0 0
X	0	0	X	0	0 X X 0 1
X	X	0	X	0	0 0 X 0 0
X	0	X	0	X	0 1 0 X 0
0	0	X	X	Y	1 1 0 0 0

dfs(4,0)

It is already marked
no need
to come again

dfs(4,1)

Some visited from

dfs(3,1)

Boundary so can't convert.

dfs(1,4)

dfs(2,4)

X

Algo :-

→ from boundary, find '0's & traverse & mark all '0's connected to it.
then → Rest of zero can be converted into X.

Code :-

```
void dfs(int row, int col, vector<vector<int>> &vis, vector<vector<char>> &mat, int delrow[], int delcol[])
{
    vis[row][col] = 1;
    int n = mat.size();
    int m = mat[0].size();
    int rrow, rcol;
    for (int i = 0; i < 4; i++)
    {
        rrow = row + delrow[i];
        rcol = col + delcol[i];
        if (rrow >= 0 && rrow < n && rcol >= 0 && rcol < m
            && !vis[rrow][rcol] && mat[rrow][rcol] == '0')
        {
            dfs(rrow, rcol, vis, mat, delrow, delcol);
        }
    }
}
```

```
vector<vector<char>> ffill(int n, int m, vector<vector<char>> mat)
{
    int delrow = {-1, 0, 1, 0};
    int delcol = {0, 1, 0, -1};
    vector<vector<int>> vis(n, vector<int>(m, 0));
    for // traverse first & last row,
        for (int i = 0; i < m; i++)
    {
        if (!vis[0][i] && mat[0][i] == '0')
        {
            dfs(0, i, vis, mat, delrow, delcol);
        }
        if (!vis[n - 1][i] && mat[n - 1][i] == '0')
        {
            dfs(n - 1, i, vis, mat, delrow, delcol);
        }
    }
    for // traverse
        for (int i = 1; i < n - 1; i++)
    {
        if (!vis[i][0] && mat[i][0] == '0')
        {
            dfs(i, 0, vis, mat, delrow, delcol);
        }
        if (!vis[i][m - 1] && mat[i][m - 1] == '0')
        {
            dfs(i, m - 1, vis, mat, delrow, delcol);
        }
    }
}
```

// traverse for 1st col & last col

```

for(int i=0; i<m; i++)
{
    if(!vis[i][0] && mat[i][0] == '0')
        dfs(i, 0, vis, mat, delrow, delcol);

    if(!vis[i][m-1] && mat[i][m-1] == '0')
        dfs(i, m-1, vis, mat, delrow, delcol);
}

```

```

for(int i=0; i<n; i++)
{
    for(int j=0; j<m; j++)
    {
        if(!vis[i][j] && mat[i][j] == '0')
            mat[i][j] = 'X';
    }
}
return mat;

```

4 neighbors

(loop Bound)

T.C. $O(N \times M) \times 4 + O(N) + O(M)$

$$\approx \underline{\underline{O(N \times m)}}$$

S.C.

$$= \underline{\underline{O(N \times m)}}$$

Number of Enclaves with single-source BFS

count the no. of '1's or land from where you can not go outside the matrix. (inside 1's)

(B means B. you can't count 1's which is on boundary and also the 1's which is connected to the boundary 1's)

then the remaining 1's will be our answer.

lets try BFS for no reason just to get clarity

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

Vf

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

put all boundary

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

$\leftarrow b \rightarrow$

Once the queue is empty now count the unvisited 1's and return ans.

ans ans = 3

Code :-

@Aashish Kumar Nayak

```

int number_of_enclaves(vector<vector<int>> &grid)
{
    queue<pair<int, int>> q;
    int n = grid.size();
    int m = grid.size();
    int vis[n][m] = {0};

    for(int i=0; i<n; i++)
    {
        for(int j=0; j<m; j++)
        {
            if(i==0 || i==n-1 || j==0 || j==m-1)
            {
                if(grid[i][j] == 1)
                {
                    q.push({i, j});
                    vis[i][j] = 1;
                }
            }
        }
    }

    int delrow[] = {-1, 0, +1, 0};
    int delcol[] = {0, +1, 0, -1};

    while(!q.empty())
    {
        int row = q.front().first;
        int col = q.front().second;
        q.pop();

        for(int i=0; i<4; i++)
        {
            int nrow = row + delrow[i];
            int ncol = col + delcol[i];
            if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
               vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1)
            {
                q.push({nrow, ncol});
                vis[nrow][ncol] = 1;
            }
        }
    }

    int cnt = 0;
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<m; j++)
        {
            if(grid[i][j] == 1 && vis[i][j] == 0) cnt++;
        }
    }
    return cnt;
}

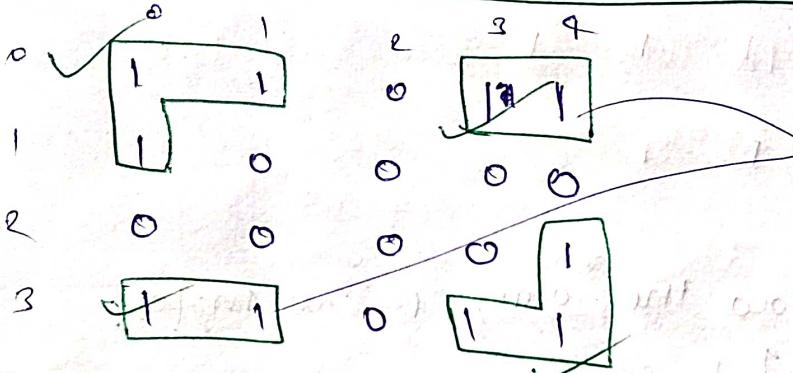
```

$$T.C. = O(N \times M) \times 4 \\ \approx O(N \times M)$$

$$S.C. \approx O(N \times M)$$

Number of distinct islands

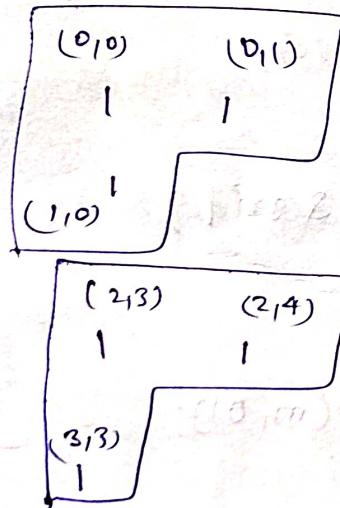
@ Ashish Kumar Nagarkar



These are identical so it will be counted as 1.

There are 3 distinct islands.

1 way if we can store the shapes in a set so that it will store only unique shapes.



How to store these shapes so that we get such that these two come out to be identified

$\{(0,0), (0,1), (1,0)\}$
 $\{(2,3), (2,4), (3,3)\}$ } These are not identical

so make starting point as base origin

$$(0,0) - (0,0) = (0,0)$$

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

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

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

$$(2,3) - (2,3) = (0,0)$$

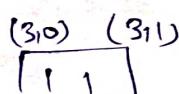
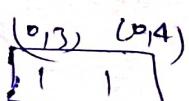
$$(2,4) - (2,3) = (0,1)$$

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

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

Now there are

identical



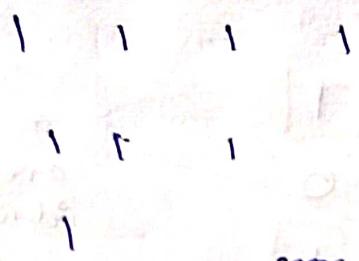
$$(0,3) - (0,3) = (0,0)$$

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

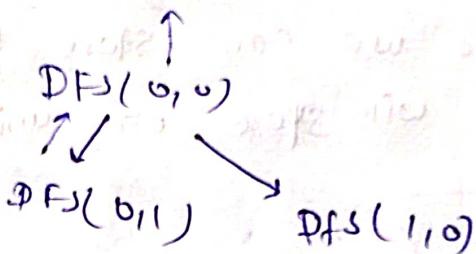
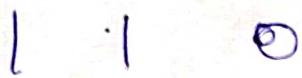
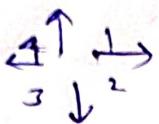
$$(3,0) - (3,0) = (0,0)$$

$$(3,1) - (3,0) = (0,1)$$

but what if



so must follow the same order for all shapes.



Code :-

```
int countDistinctIslands( vector<vector<int>> &grid )  
{  
    int m = grid.size();  
    int n = grid[0].size();  
    vector<vector<int>> vis(m, vector<int>(n, 0));  
    set<vector<pair<int, int>> st;  
    for (int i=0; i<m; i++)  
    {  
        for (int j=0; j<n; j++)  
        {  
            if (!vis[i][j] && grid[i][j] == 1)  
            {  
                vector<pair<int, int>> vec;  
                dfs(i, j, vis, grid, vec, i, j);  
                st.insert(vec);  
            }  
        }  
    }  
    return st.size();  
}
```

```

void dfs(int row, int col, vector<vector<int>> &vis,
         vector<vector<int>> &grid, vector<pair<int, int>> &vec, int
         nrow, int ncol) {
    vis[row][col] = 1; vec.push_back({row, col});
    int n = grid.size();
    int m = grid[0].size();
    vec.push_back({row - nrow, col - ncol});
    int delrow[] = {1, 0, 1, 0};
    int delcol[] = {0, -1, 0, 1};
    for (int i = 0; i < 4; i++) {
        int nrow = row + delrow[i];
        int ncol = col + delcol[i];
        if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m
            && !vis[nrow][ncol] && grid[nrow][ncol] == 1)
            dfs(nrow, ncol, vis, grid, vec, nrow, ncol);
    }
}

```

for loop ~~dfs~~ x neighbour

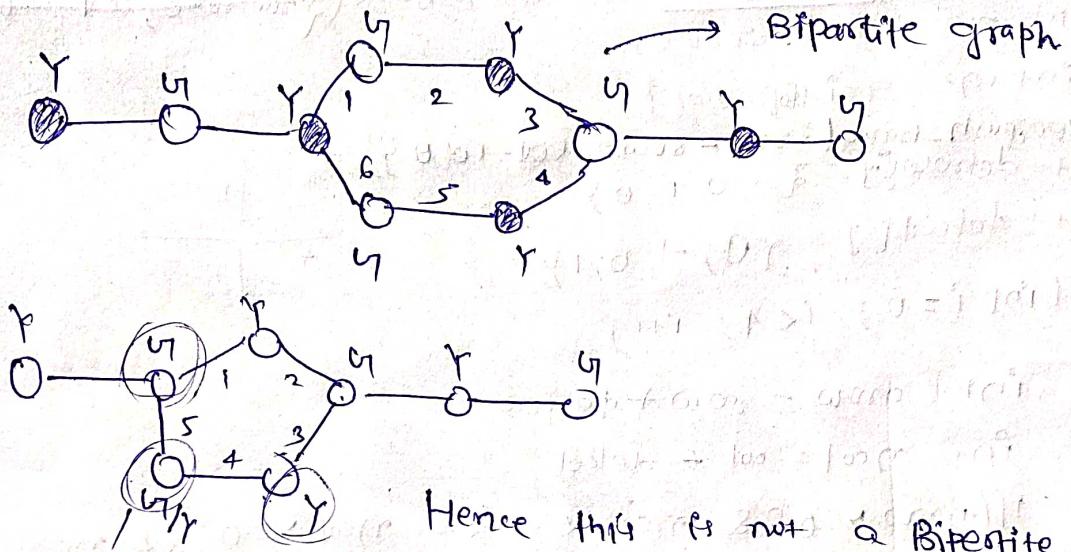
$$\begin{aligned}
 T.C. &= O(NXM) + O(NXM) \times 4 \\
 &\approx O(NXM) = O(NXM \log(NXM) + (NXM) \times 4)
 \end{aligned}$$

S.C. $\approx \underline{\underline{O(NXM)}}$

Bipartite graph

@Aashish Kumar Nayak

Colour the graph with 2 colours such that no adjacent nodes have same colour.



Hence this is not a Bipartite graph.

we can't colour this graph with any two colours

* Linear graphs with no cycles are always Bipartite graph.

* Any graph with even cycle length can also be Bipartite.

* Any graph with odd length cycle can never be a Bipartite.

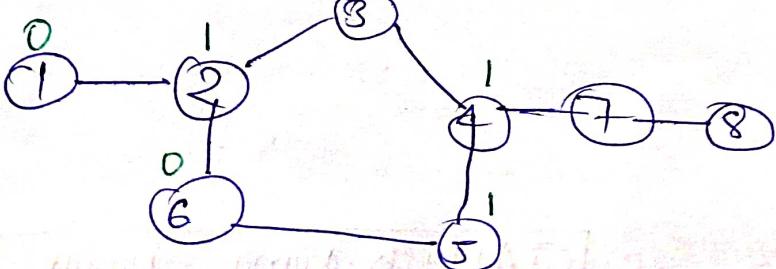
We will solve this problem using

2 instead of visited array we will take

BFS

colour array.

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



Colours

0 -> H. 1 -> 1 -> 1 -> 1 -> 1 -> 1 -> 1 -> 1 -> 1

0 1 2 3 4 5 6 7 8

node = 4

Code :-

bool check(int start, int v, vector<int>

vector<int> adj[], int color[])

{ queue<int> q;

q.push(start);

color[start] = 0;

while (!q.empty())

{ int node = q.front();
q.pop();

for (auto it : adj[node])

{ if (color[it] == -1)

{ color[it] =

-1 * color[node];

}

else if (color[it] == color[node])

{ return false;

}

}

return true;

bool isBipartite (int v, vector<int> adj[])

{ int color[v];

for (int i = 0; i < v; i++)

color[i] = -1;

for (int i = 0; i < v; i++)

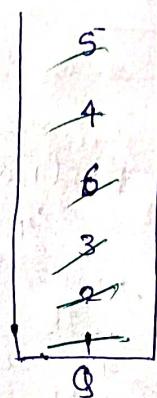
{ if (color[i] == -1)

{ if (check(i, v, adj, color) == false)

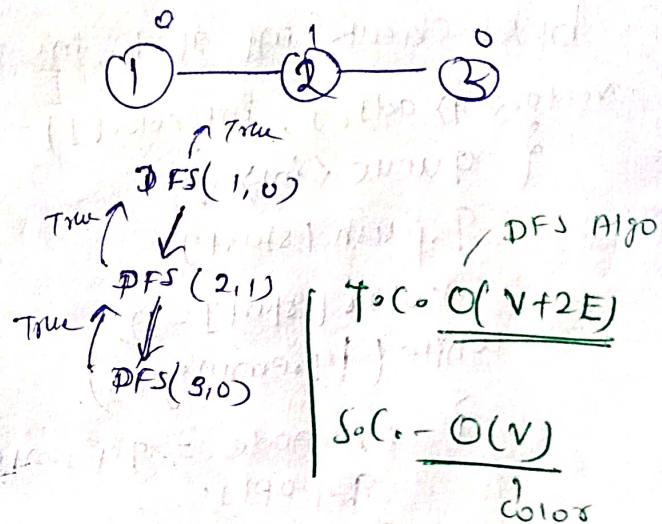
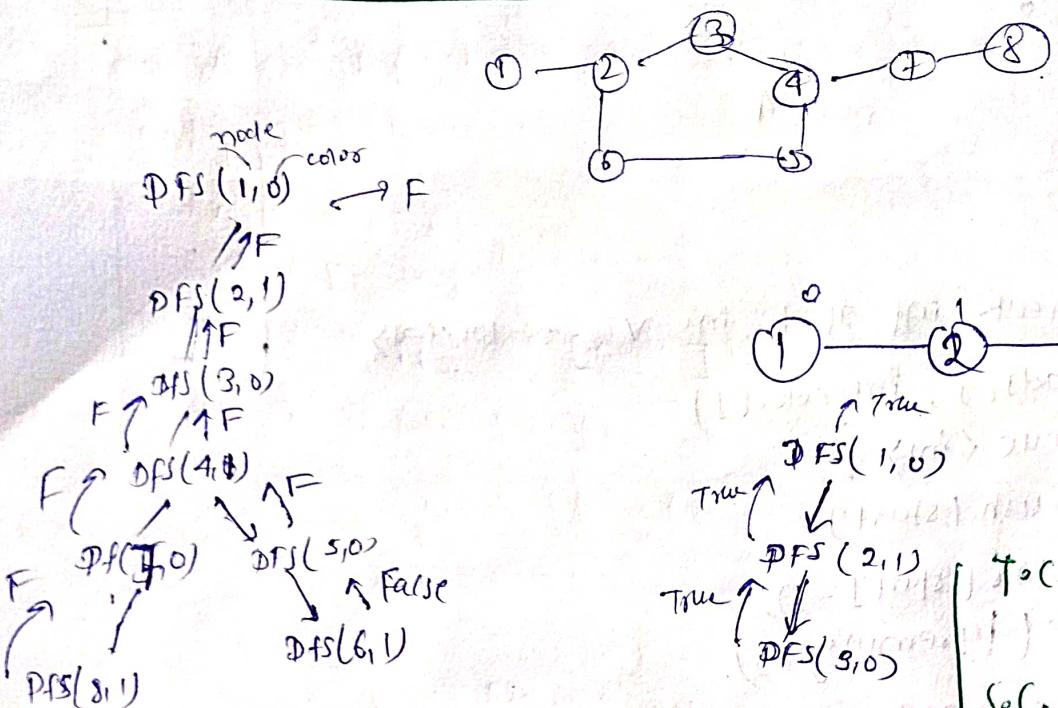
{ return false;

}

return true;



Bipartite Graph using DFS



code :-

```

bool dfs(int node, int col, int color[], vector<int> adj[])
{
    color[node] = col;
    for(auto it : adj[node])
    {
        if(color[it] == -1)
        {
            if(dfs(it, !col, color, adj) == false)
                return false;
            else if(color[it] == col)
                return false;
        }
    }
    return true;
}

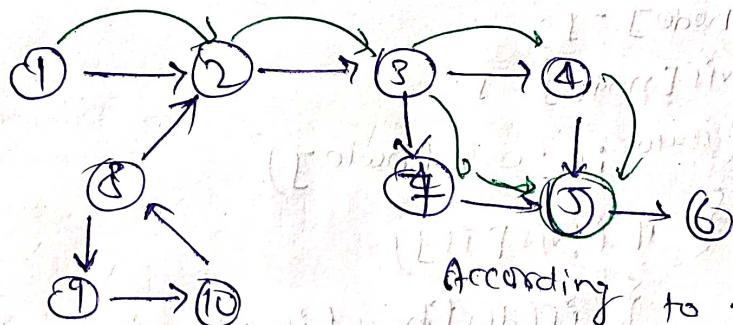
bool isBipartite(int v, vector<int> adj[])
{
    int color[v];
    for(int i=0; i<v; i++)
        color[i] = -1;
    for(int i=0; i<v; i++)
    {
        if(color[i] == -1)
        {
            if(dfs(i, 0, color, adj) == false)
                return false;
        }
    }
    return true;
}

```

Detect cycle in a directed graph

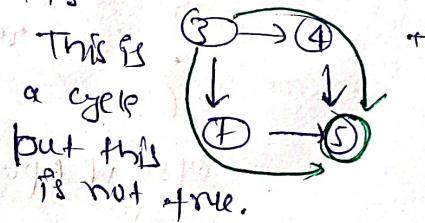
Visited algorithm using the DFS will not work here
this algo will work only in undirected graph

for ex:-



\checkmark dfs(1)
 \checkmark dfs(2)
 \downarrow
 \checkmark dfs(3)
 \downarrow
 \checkmark dfs(4) \rightarrow dfs(7)
 \downarrow
 \checkmark dfs(5)

According to visited algo using DFS



so [on the same path, node has to be visited again]

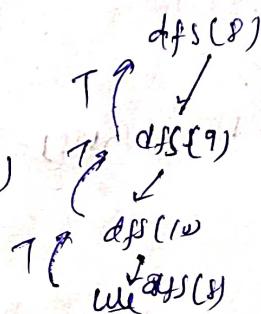
1	2	3	4	5	6	7	8	9	10
0	1	1	1	1	1	1	1	1	1

vis →

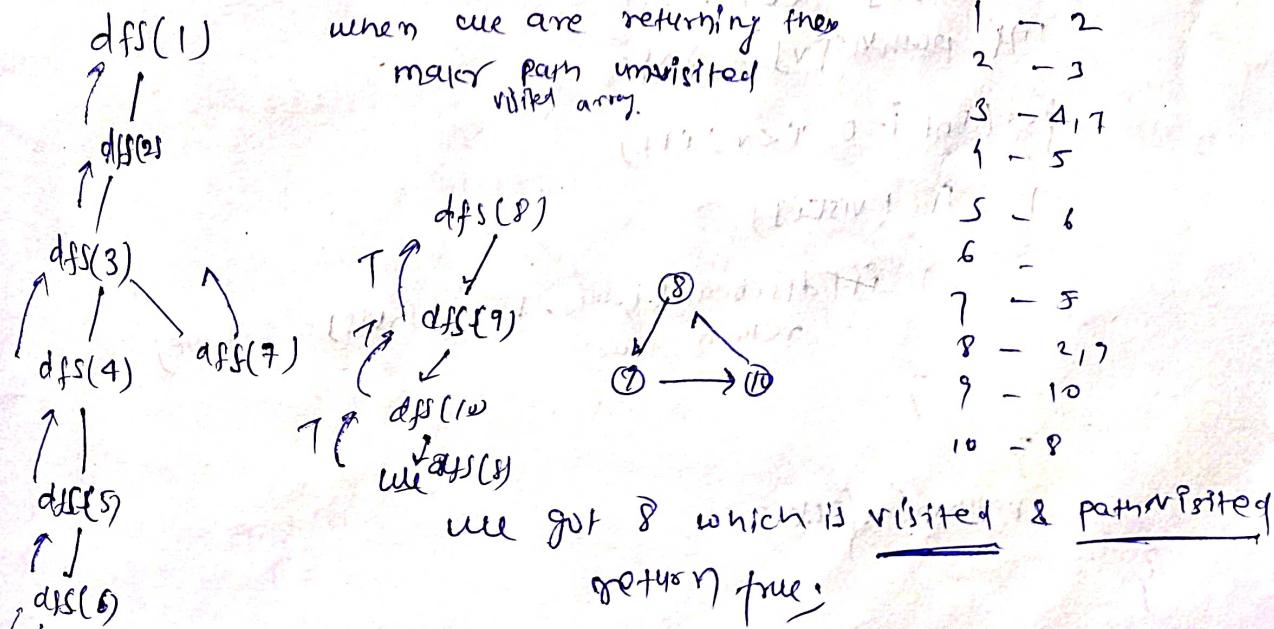
1	2	3	4	5	6	7	8	9	10
0	1	1	0	1	0	1	0	1	1

path vis →

when we are returning from
make path unvisited
visited array.



$1 \rightarrow 2$
 $2 \rightarrow 3$
 $3 \rightarrow 4, 7$
 $4 \rightarrow 5$
 $5 \rightarrow 6$
 $6 \rightarrow -$
 $7 \rightarrow 5$
 $8 \rightarrow 2, 7$
 $9 \rightarrow 10$
 $10 \rightarrow 8$



Code :-

```
bool dfscheck(int node, vector<int> adj[], int vis[], int pathvis[])
```

```
{  
    vis[node] = 1;  
    pathvis[node] = 1;  
    for (auto it : adj[node])  
    {  
        if (!vis[it])  
        {  
            if (dfscheck(it, adj, vis, pathvis) == true)  
                return true;  
        }  
        else if (pathvis[it])  
        {  
            return true;  
        }  
    }  
    pathvis[node] = 0;  
    return false;  
}
```

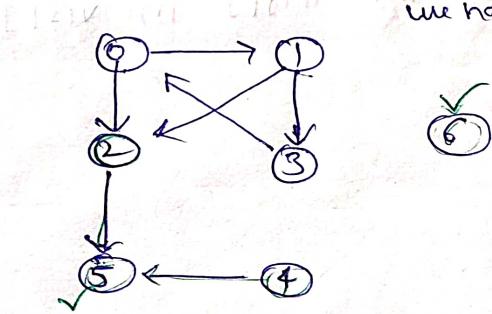
```
bool iscyclic(int V, vector<int> adj[])
```

```
{  
    int vis[V] = {0};  
    int pathvis[V] = {0};  
    for (int i = 0; i < V; i++)  
    {  
        if (!vis[i])  
        {  
            if (dfscheck(i, adj, vis, pathvis) == true)  
                return true;  
        }  
    }  
    return false;  
}
```

$$T.C. = O(V + E)$$

$$\text{Sec.} = O(2V) \approx O(V)$$

Find Eventual Safe States:



We have to find

which are the safe node →

terminal nodes

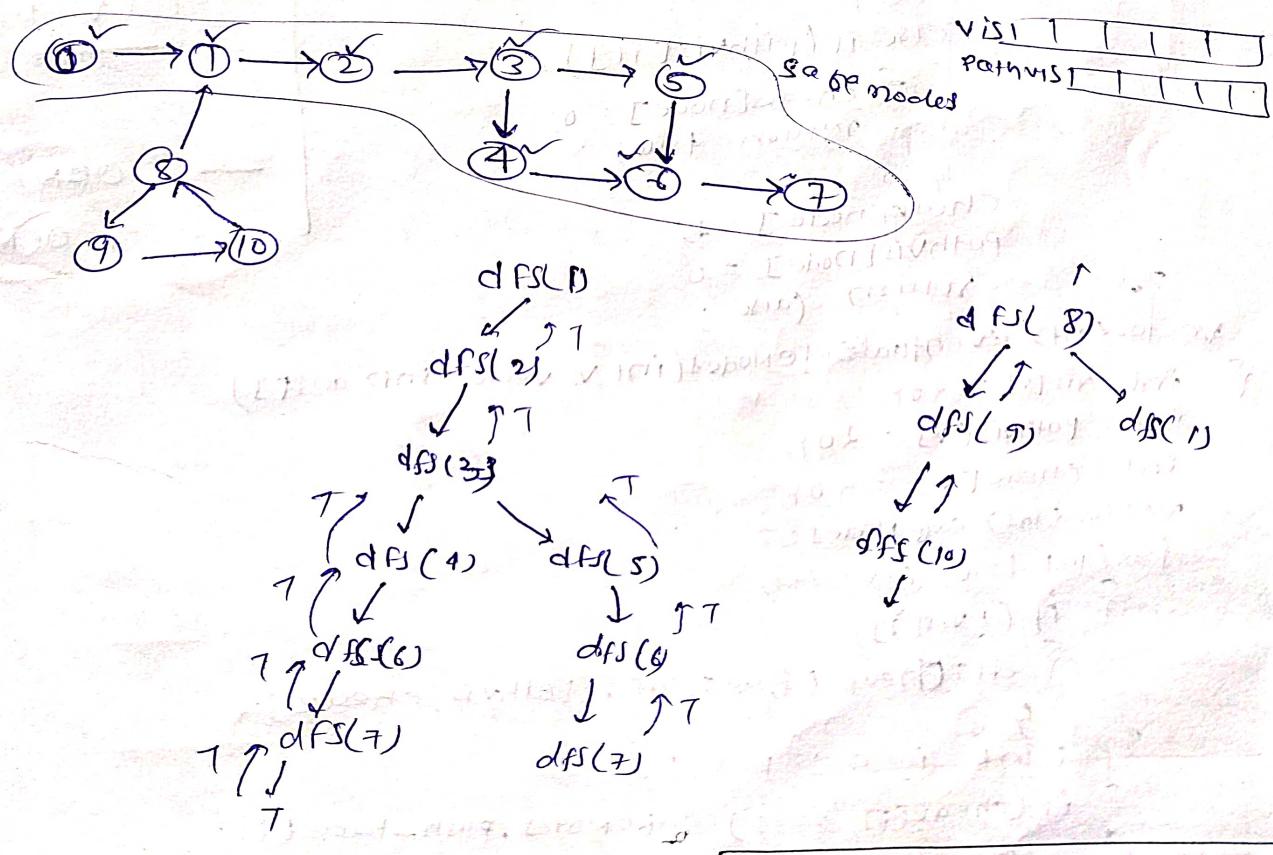
Outdegree $\Rightarrow 0$

Nodes which are ending at terminal node

Safe Node Ans $\rightarrow [2, 4, 5, 6]$

- Any one who is a part of cycle can not be safe node
- Any one that leads to a cycle can not be safe node
- Rest of all are safe nodes

We will solve by cycle detection technique.



@ Ashish Kumar Nayak

Code:

```
bool dfscheck(int node, vector<int> adj[], int vis[],  
    int pathvis[], int check[]){  
    vis[node] = 1;  
    pathvis[node] = 1;  
    check[node] = 0;  
    for(auto it : adj[node])  
    {  
        if(!vis[it])  
        {  
            if(dfscheck(it, adj, vis, pathvis, check)  
                == true)  
            {  
                check[node] = 0;  
                return true;  
            }  
        }  
        else if(pathvis[it])  
        {  
            check[node] = 0;  
            return true;  
        }  
    }  
    check[node] = 1;  
    pathvis[node] = 0;  
    return false;  
}  
  
vector<int> eventualSafeNodes(int v, vector<int> adj[]){  
    int vis[v] = {0};  
    int pathvis[v] = {0};  
    int check[v] = {0};  
    vector<int> safeNodes;  
    for(int i=0; i<v; i++)  
    {  
        if(!vis[i])  
        {  
            dfs(check, i, adj, vis, pathvis, check);  
        }  
    }  
    for(int i=0; i<v; i++)  
    {  
        if(check[i] == 1) safeNodes.push_back(i);  
    }  
    return safeNodes;  
}
```

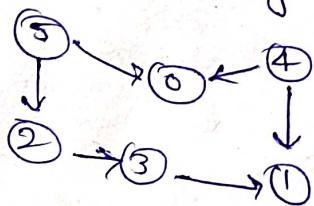
$$\underline{\text{T.C}} \rightarrow \underline{O(V+E)}$$

$$\underline{\text{S.C}} - \underline{O(2N)} \\ \approx \underline{O(N)}$$

Topological Sort (DFS) :-

(DAG)

if there is an edge between $u \rightarrow v$, u appears before v in that ordering.



Only possible in (DAG) Directed Acyclic graph

i) should be directed

ii) should be no cycle.

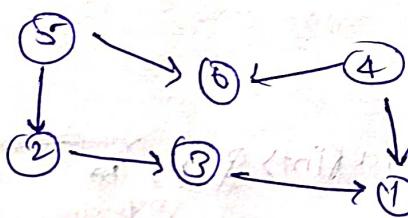
$1 - 2$

$1 - 2$

$2 - 1$ X Not possible

$1 - 2$ X Not possible

1 should be before
3 should be before 1
both X not possible
at some time.



Adjacency list

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

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

VIS [1 1 1 1 1]



st.top(),

{ 5 4 2 3 1 0 } — Ans

@Aashish Kumar Nayak

Institution :-



first we start traversing from 1st then we will go till end & when we are returning back at that time we will store it on stack.

1 2 3 5 C	whose dfs is completed just store it on stack.
-----------------------	--

Code :-

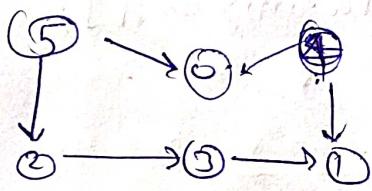
```
vector<int> toposort( int v, vector<int> adj[ ] )  
{  
    int vis[v] = {0};  
    stack<int> st;  
    for( int i = 0; i < v; i++ )  
    {  
        if( !vis[i] )  
        {  
            dfs(i, vis, st, adj);  
        }  
    }  
  
    vector<int> ans;  
    while( !st.empty() )  
    {  
        ans.push_back(st.top());  
        st.pop();  
    }  
    return ans;  
}
```

```
void dfs( int node, int vis[], stack<int> &st, vector<int> adj[ ] )  
{  
    vis[node] = 1;  
    for( auto it : adj[node] )  
    {  
        if( !vis[it] )  
        {  
            dfs(it, vis, st, adj);  
        }  
    }  
    st.push(node);  
}
```

S.C. : $O(N)$ + $O(N)$
 $\approx O(N)$

T.C. : $O(V + E)$
↓
directed
graph

Kahn's Algorithm for topological sort

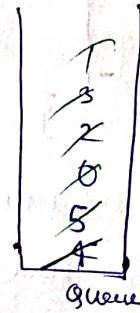


We will use Queue & indegree array

Indegree

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

There is no element which comes to ≥ 2 times
 \Rightarrow we can place 4, 5 in starting.



Step 1: Insert all nodes with indegree 0

Step 2: take them out of the queue.

Step 3: Remove the degree of adjacent nodes.

How we find indegree

We can do that from adj list

$2 \rightarrow 3$ it means indegree of 3 is 1
 so we will increase its indegree

increase its indegree

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

Code :-

```
vector<int> toposort(int v, vector<int> adj[])
```

```
{ int degree[v] = {0};  
    for(int i=0; i<v; i++)  
    {  
        for(auto it: adj[i])  
        {  
            indegree[it]++;  
        }  
    }
```

```
queue<int> q;
```

```
for(int i=0; i<v; i++)  
{ if(indegree[i] == 0)  
    { q.push(i);  
    } }
```

```
vector<int> topo;  
while(!q.empty())
```

```
{ int node = q.front();  
    q.pop();  
    topo.push_back(node);
```

```
for(auto it: adj[node])  
{  
    indegree[it]--;  
    if(indegree[it] == 0)  
        q.push(it);  
}
```

```
return topo;
```

T.C. $O(V+E)$

S.C. $O(N) + O(N) + O(N)$
 $\approx O(N)$

Detect a cycle in directed graph

using topological sort \rightarrow BFS

In previous form we used vis, pathvisited and when we were returning then we were resetting the pathvisited array.

But in BFS we can't do that like backtracking so we use Kahn's Algo

This is a direct acyclic graph

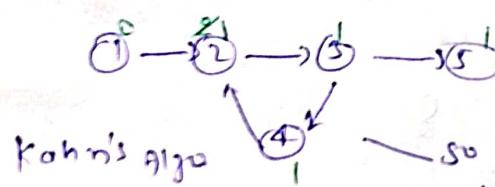


\oplus

1 → 2
2 → 3
3 → 4, 5
4 → 2
5 → 4

Toposort is only applicable for DAG

Let's try to find out the topological sort for this

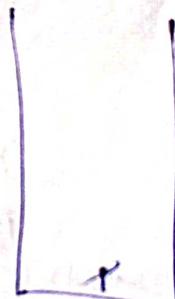


Kahn's algo

Indegree
queue
reduce indegree by 1

so for this
toposort will be

[1]



So if Toposort has N elements then it must be DAG

2. If Toposort has less than N elements then it must have cycle.

Code :- Code is same as Kahn's Algo

We will just check size of topsort array
in last if size of topsort == N then there is no
cycle or and if size is less than N then there is
cycle.

```
bool iscycle(int v, vector<int> adj[])
```

```
{ int indegree[v] = {0};
```

```
for( int i=0; i<v; i++)
```

```
{ for( auto it: adj[i])
```

```
{ indegree[it]++;
}
```

```
}
```

```
Queue<int> q;
```

```
for( int i=0; i<v; i++)
```

```
{ if(indegree[i] == 0)
    q.push(i);
}
```

```
}
```

```
int cnt=0;
```

```
while( !q.empty() )
```

```
{ int node = q.front();
```

```
q.pop();

```

```
cnt++;

```

```
for( auto it: adj[node] )
```

```
{ indegree[it]--;
}
```

```
{ if( indegree[it] == 0 )
    q.push(it);
}
```

```
if( cnt == v )
    return false;
}
```

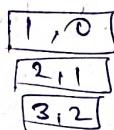
```
return true;
}
```

T.C. - O(N+E)

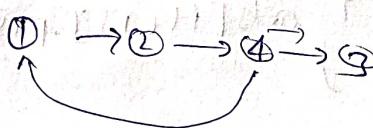
S.C. - O(N)

Course Schedule I & II

Pre-requisite Task



3 2 1 0 Yes



No

This is happening because cycle is present.
We can use topological sort.

$u \rightarrow v \Rightarrow (u, v)$ is order. If u must have before v .

Topo sort

```
Code = do{ if(possible(int v, vector<pair<int, int>> &prereq))  
    { vector<int> adj[v];  
        for(auto it: prereq[v])  
            adj[it.first].push_back(it.second);  
    }  
} while(indgree[v] != 0);
```

```
for(int i=0; i<v; i++) { for(auto it: adj[i])  
    { indgree[it]++; } }
```

queue<int> q;

```
for(int i=0; i<v; i++)  
    { if(indgree[i] == 0)  
        { q.push(i); } }
```

```
vector<int> topo;  
while(!q.empty())
```

```
{ int node = q.front();  
    q.pop(); }
```

```
topo.push_back(node);
```

```
for(auto it: adj[node])  
    { indgree[it]--;  
        if(indgree[it] == 0)  
            q.push(it); }
```

```
if(topo.size() == v) return true;  
return false;
```

Code

Course Schedule

```

Vector<int> findOrder( int n, int m, vector<vector<int>> &
{                                     // prerequisites)
    for( auto v : adj[v] )
        for( auto it : prerequisites )
            if( it[1] == v )
                adj[it[0]].push_back(v);

    for( int i = 0; i < n; i++ )
        for( auto it : adj[i] )
            indegree[it]++;
}

queue<int> Q;
for( int i = 0; i < n; i++ )
    if( indegree[i] == 0 )
        Q.push(i);

while( !Q.empty() )
    int node = Q.front();
    Q.pop();
    topo.push_back(node);

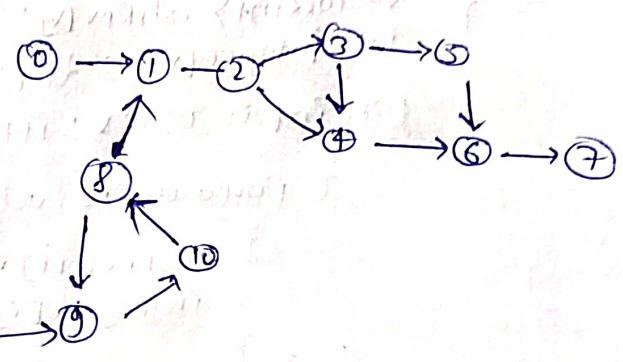
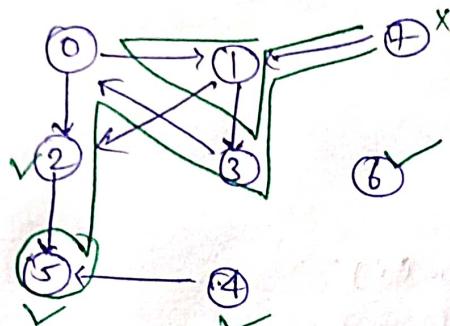
    for( auto it : adj[node] )
        indegree--;
        if( indegree[it] == 0 )
            Q.push(it);

if( topo.size() == n ) return topo;
else return {};
}

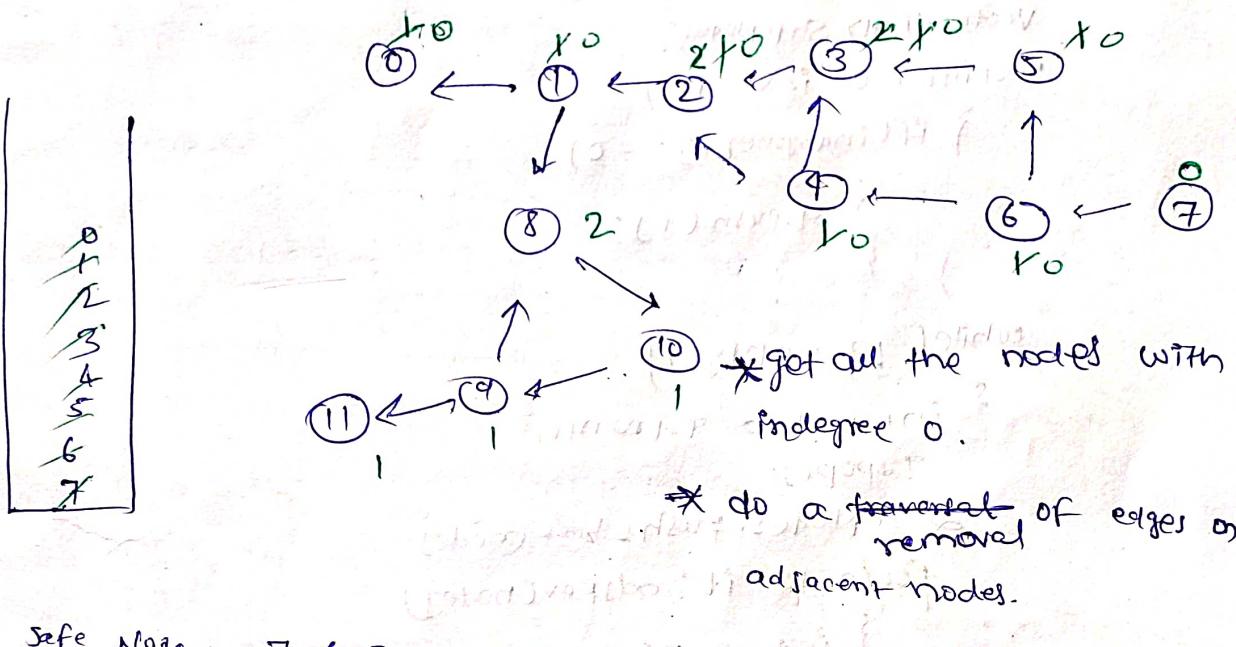
```

* * *
 something is before
 something
 use topo sort in
 these kind of
 problem.

Eventual Safe States using BFS Topo sort



→ Reverse all the edges



* do a ~~forward~~ removal of edges on adjacent nodes.

Safe Node = 7 6 5 4 3 2 1 0

then we will sort the safe node array.

then return

Code 2

```
vector<int> eventualSafeNodes(int V, vector<int> adj[]) {
    vector<int> adjRev[V];
    int indegree[V] = 207;
    for(int i=0; i<V; i++) {
        for(auto it: adj[i]) {
            adjRev[it].push_back(i);
            indegree[i]++;
        }
    }
    queue<int> q;
    vector<int> safenodes;
    for(int i=0; i<V; i++) {
        if(indegree[i] == 0)
            q.push(i);
    }
    while(!q.empty()) {
        int node = q.front();
        q.pop();
        safenodes.push_back(node);
        for(auto it: adjRev[node]) {
            indegree[it]--;
            if(indegree[it] == 0)
                q.push(it);
        }
    }
    sort(safenodes.begin(), safenodes.end());
    return safenodes;
}
```

T.C. = $O(V+E)$ + $O(V \log V)$ sorting

SC. = $\sim O(V)$

Alien dictionary | Topological sort

Alien

[bba
abcd
abca
cab
cad]

Normal

a comes before b
 b comes before c
 c comes before d
 a comes before c

[abc
acd
baa
cab
cad]

a b c d e f g h -

find out the order. find out the alien order?

b, a, c

b d a c

Alien Order

* * * This question says something before something so use Topological sort.

i = 0

[bba
abcd
abca
cab
cad]

figure out where char is not equal

and then arr[i] according to that

i = 1

$s_1 = \text{arr}[i];$
 $s_2 = \text{arr}[i+1];$

compare s_1 & s_2

If we got unequal char

then make directed graph

then after completing the graph.

We will do toposort then return the toposort.

return ans

(@Aashish Kumar Nayak)

* what if order is not possible. like
 $\begin{array}{c} \text{abc d} \\ \text{abc } - s_1 \\ \text{abc } - s_2 \end{array}$
if the largest string is before smallest string and every character of small string is matched.
then its wrong dictionary.

* And also when there is cyclic dependency

ex:-
wrong dictionary $a \rightarrow b$ $\begin{array}{c} a \ b \ c \\ a \not\sim b \\ b \ a \ t \\ x \ b \ a \ l \ a \ d \ e \\ \text{not possible} \end{array}$

String findOrder (string dict[], int N, int K) {
 no. of string
 total char.

```

    vector<int> adj[K];
    for (int i = 0; i < N; i++) {
        string s1 = dict[i];
        string s2 = dict[i + 1];
        for (int len = min(s1.size(), s2.size()); len > 0; len--) {
            for (int p1 = 0; p1 < len; p1++) {
                if (s1[p1] != s2[p1]) {
                    adj[s1[p1]].push_back(s2[p1]);
                    break;
                }
            }
        }
    }
}
    
```

vector<int> topo = toposort(K, adj);

string ans = "";

for (auto it : topo)

ans = ans + char(it + 'a');

return ans;

```

vector<int> topoSort( int v, vector<int> adj[] )
{
    int indegree[v] = 0;
    for( int i=0; i<v; i++ )
    {
        for( auto it: adj[i] )
            indegree[it]++;
    }
}

queue<int> q;
for( int i=0; i<v; i++ )
{
    if( indegree[i] == 0 )
        q.push(i);
}

vector<int> topo;
while( !q.empty() )
{
    int node = q.front();
    q.pop();
    topo.push_back(node);

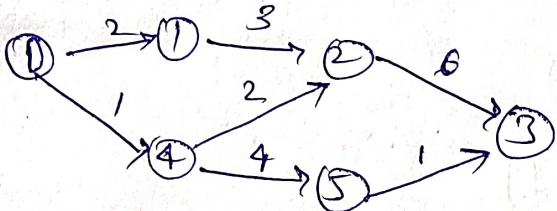
    for( auto it: adj[node] )
    {
        indegree[it]--;
        if( indegree[it] == 0 )
            q.push(it);
    }
}

return topo;
}

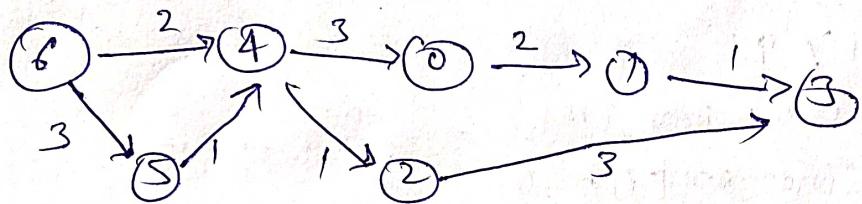
```

SOS Shortest path in Directed Acyclic Graph

Topological sort:-



We have to print the shortest distance for all nodes from 0 node in an array.



adj list :-

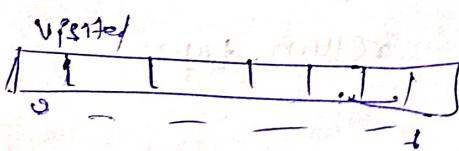
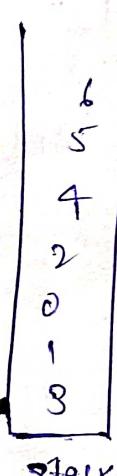
	node	weight
0	{1, 2}	
1	{3, 4}	
2	{3, 5}	
3		
4	{0, 3, 5}	
5	{4, 1, 3}	
6	{4, 1, 2, 3, 5, 3}	

Step 1 :- Do a toposort on the graph.

DFS ways
BFS ways

DFS method

this specie
will store the
toposort



6 | 5 | 4 | 2 | 0 | 1 | 3

@Aashish Kumar Nayak

Step 2: take the nodes out of stack and relax the edges.

node = 6 , dist = 0
 ↗
 node = 5
 dist = 3

↓
 node = 4

dist[] # [5 | 7 | 3 | 6 | 2 | 3 | 0]
 6 2 3 4 5 6

node = 5 , dist = 3
 ↗
 node = 4
 dist = 4

$S \rightarrow \{4, 1\}$

```

Code: vector<int> shortestpath( int N, int M, vector<int> edges[])
{
    vector<pair<int, int>> adj[N];
    for( int i = 0; i < M; i++ )
    {
        int u = edges[i][0];
        int v = edges[i][1];
        int wt = edges[i][2];
        adj[u].push_back({v, wt});
    }
    int vis[N] = {0}; → for( int i = 0; i < N; i++ )
    stack<int> st; → if( !vis[i] ) → toposort(i, adj, vis, st);
    for( int i = 0; i < N; i++ ) →
    {
        if( dist[i] == INT_MAX ) →
        dist[0] = 0; →
        while( !st.empty() ) →
        {
            int node = st.top(); →
            st.pop();
            for( auto it : adj[node] ) →
            {
                int v = it.first;
                int wt = it.second;
                if( dist[node] + wt < dist[v] ) →
                dist[v] = dist[node] + wt;
            }
        }
    }
    return dist;
}

```

```

void toposort(int node, vector<pair<int, int>> adj[],
int vis[], stack<int> &st)
{
    vis[node] = 1;
    for (auto it : adj[node])
    {
        int v = it.first;
        if (!vis[v])
        {
            toposort(v, adj, vis, st);
        }
    }
    st.push(node);
}

```

$$T_{\text{c.}} \approx O(N + M)$$

$$S_{\text{c.}} \approx O(N)$$

Intuition:

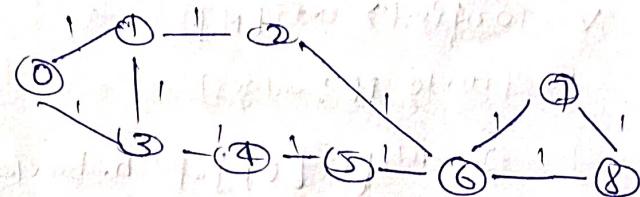
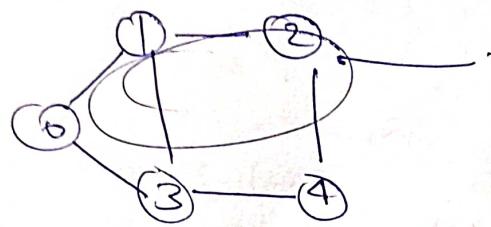
Finding the shortest path to a vertex is easy if you already know the shortest path of all the vertices that can precede it. Finding the longest path to a vertex in DAG is easy if you already know the longest path to all the vertices that can precede it.

Processing the vertices in topological order ensures that by the time you get to a vertex, you have already processed all the vertices that can precede it.

Dijkstra's algorithm is necessary for graphs that can contain cycles because they can't be topologically sorted.

Shortest path in undirected graph

having unit distance.



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

1 & 2 so no updation

node = 0 dist = 0
node = 1 node = 3
dist = 1 dist = 1

node = 1, dist = 1
+1 +1 +1
node 0 node 2 node 3
dist = 2 dist = 2 dist = 2
discarded {node, dist}

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

queue

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

we are looking for shortest path

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

node = 2, dist = 2

node = 1
dist = 3

node = 4 dist = 2

node = 3
dist = 3

mode = 2, dist = 3

node 2 5 7 8
dist 4 4 4 4

mode = 5, dist = 3

node 4 6
dist 4 4

mode = 7, dist = 4

node 6 8
dist 5 5

Node = 8, dist = 4
node 6 7
dist 5 5

Code :-

```
vector<int> shortestPath(vector<vector<int>> &edges, int N,
int M, int src)
{
    vector<int> adj[N];
    for(auto it : edges)
    {
        adj[it[0]].push_back(it[1]);
        adj[it[1]].push_back(it[0]);
    }
    int dist[N];
    for(int i = 0; i < N; i++)
    {
        dist[i] = INT_MAX;
    }
    dist[src] = 0;
    queue<int> q;
    q.push(src);
    while(!q.empty())
    {
        int node = q.front();
        q.pop();
        for(auto it : adj[node])
        {
            if(dist[node] + 1 < dist[it])
            {
                dist[it] = 1 + dist[node];
                q.push(it);
            }
        }
    }
    vector<int> ans(N, -1);
    for(int i = 0; i < N; i++)
    {
        if(dist[i] != INT_MAX)
        {
            ans[i] = dist[i];
        }
    }
    return ans;
}
```

P.C. - BFS algo
 $\underline{\underline{O(V + 2E)}}$

S.C. $\approx \underline{\underline{O(N)}}$

WORD Ladder - I

Hard problem

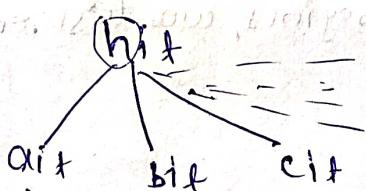
begin word = "hit" end word = "cog"

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

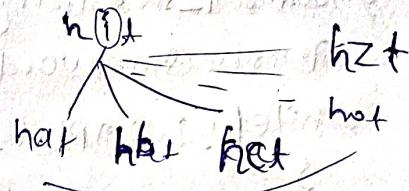
hit → hot → dot → dog → log

Minimum no. of transformation

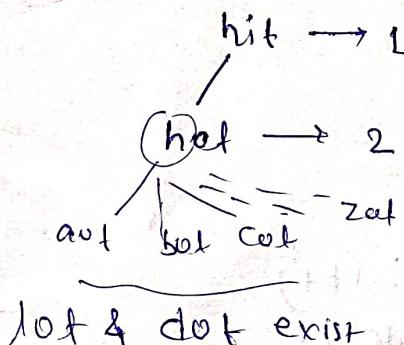
Brute Force



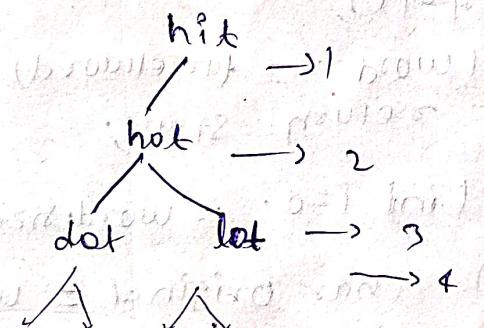
check all in word list



check all
we got hot



lot & dot exist



→ shortest sequence

BFS

hit, 1

aut

bit

;

zit

hit

hot

hbt

;

hot

2

hot, 2

bot

bot

;

bot

bot

;

bot

;

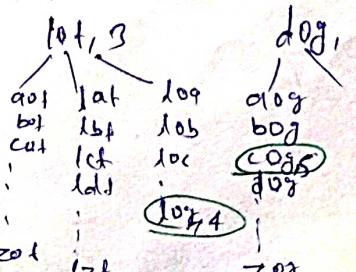
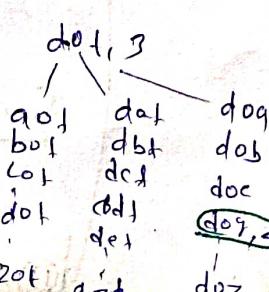
bot

we will remove words from wordList

when we will get word.

and if we get same word in next iteration we will not face it.

[hot, dot, dog, lot, log, cog] — Now we can stop



Ans = 5

Code :-

```
int wordLadderLength(string startword, string targetword,
vector<string> &wordlist)
{
    queue<pair<string, int>> q;
    q.push({startword, 1});
    set<string> st({wordlist.begin(), wordlist.end()});
    st.erase(startword);
    while(!q.empty())
    {
        string word = q.front().first;
        int steps = q.front().second;
        q.pop();
        if(word == targetword)
            return steps;
        for(int i=0; i<word.size(); i++)
        {
            char original = word[i];
            for(char ch='a'; ch<='z'; ch++)
            {
                word[i] = ch;
                if(st.find(word) != st.end())
                {
                    st.erase(word);
                    q.push({word, steps+1});
                }
            }
            word[i] = original;
        }
    }
    return 0;
}
```

T.C. $O(N * \text{word.length} * 26)$
 $\approx 10^8 N$

S.C. $O(N)$

@Aashish Kumar Nayak

Word Ladder II

Hard

beginning word = bat end word = coz

word list = [pat, bot, pot, poz, coz]

bat → pat → pot → poz → coz

bat → bot → pot → poz → coz

[pat, bot, pot, poz, coz]

{ bat } word = bat

Pat bot X

{ bat, pat }

1 last word → pat

pot bat X

{ bat, bot }

bot

pot

{ bat, pat, pot }

pot

pot

poz

{ bat, bot, pot }

pot

poz

{ bat, pot, poz }

poz

coz

{ bat, bot, pot, poz }

poz

coz

Ans 1/12

{ bat, bot, pot, poz, coz } 5

{ bat, pot, poz, coz } 4

{ bat, bot, pot, poz } 3

{ bat, pot, pot } 2

{ bat, pot } 1

levels

Code :-

```
vector<vector<string>> findSequence(string beginword, string endword
vector<string>& wordlist)
{
    unordered_set<string> st(wordlist.begin(), wordlist.end());
    queue<vector<string>> q;
    vector<string> used;
    q.push({beginword});
    vector<string> usedOnLevel;
    usedOnLevel.push_back(beginword);
    int level = 0;
    vector<vector<string>> ans;
    while(!q.empty())
    {
        vector<string> vec = q.front();
        q.pop();
        if(vec.size() > level)
        {
            level++;
            for(auto it : usedOnLevel)
                st.erase(it);
            usedOnLevel.clear();
        }
        string word = vec.back();
        if(word == endword)
        {
            if(ans.size() == 0)
            {
                ans.push_back(vec);
            }
            else if(ans[0].size() == vec.size())
            {
                ans.push_back(vec);
            }
        }
    }
}
```

```

for (int i=0; i<word.size(); i++)
{
    char original = word[i];
    for (char ch='a'; ch<='z'; ch++)
    {
        word[i] = ch;
        if (std::count(word) > 0)
        {
            vec.push_back(word);
            q.push(vec);
            UsedOnLevel.push_back(word);
            vec.pop_back();
        }
        word[i] = original;
    }
}
return ans;
}

```

This soln will not work on LeetCode (T.L.E)
but it is fine and acceptable in enterprise.

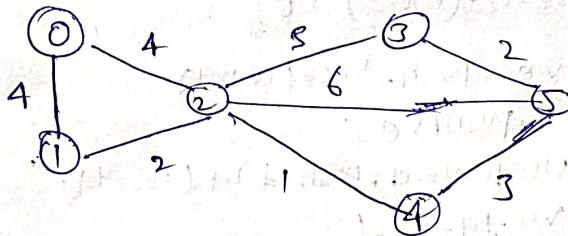
T.C. It will vary with different examples.
It is next to impossible 😊 to find T.C. of this code.

Worst ladder: II (optimised approach)

Dijkstra's

we always wanted minimum distance at 1st.

Reason for using priority-queue



0 → 0 $d=0$

→ 1 $d=4$

$\frac{2}{3}$ $d=4$

Algorithm using priority queue

Algorithm → shortest path

Adj list

$0 \rightarrow \{1, 4\}, \{2, 4\}$

$1 \rightarrow \{0, 4\}, \{2, 4\}$

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

$3 \rightarrow \{4, 3\}, \{5, 2\}$

$4 \rightarrow \{2, 1\}, \{5, 3\}$

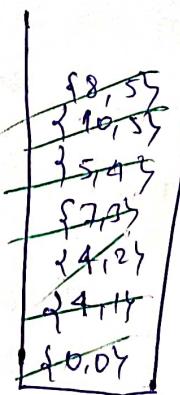
$5 \rightarrow \{4, 6\}, \{3, 2\}, \{4, 3\}$

There are two methods in Dijkstra's algo

① priority queue take less time

② Set → fastest way

③ Queue → take much time



min heap
{dist, node}

$d=0$ node = 0
node 1 node 2
distance = $0+4$ $0+4$

$d=4$ node = 1
node 0 node 2
distance = $4+4$ $4+2$
 $=8$ $=6$

node = 4 dist = 5
node = 2 dist = $5+4$
 $=9$ node = 5
dist = $5+3$
 $=8$

node = 3
node 2 node 5
dist = $7+3$
 $=10$ node = 4
dist = $7+2$
 $=9$ node = 6
dist = $10+3$
 $=13$

dist = 10
node = 5
node 2 node 3
dist = $10+2$
 $=12$ node = 4
dist = $10+3$
 $=13$

Ne code :-

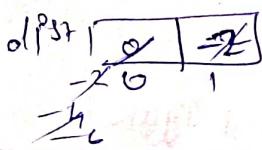
@Aashish Kumar Nayak

```
vector<int> dijkstra(int v, vector<vector<pair<int, int>>> adj[], int s)
{
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> pq;
    vector<int> dist(v);
    for(int i=0; i<v; i++)
        dist[i] = INT_MAX;
    dist[s] = 0;
    pq.push({0, s});
    while(!pq.empty())
    {
        int dis = pq.top().first;
        int node = pq.top().second;
        pq.pop();
        for(auto it : adj[node])
        {
            int edgeWeight = it[1];
            int adjNode = it[0];
            if(dis + edgeWeight < dist[adjNode])
            {
                dist[adjNode] = dis + edgeWeight;
                pq.push({dist[adjNode], adjNode});
            }
        }
    }
    return dist;
}
```

$$\boxed{\begin{array}{ll} \text{Time} & E \log V \\ \text{Space} & \text{edges} \\ & \text{nodes} \end{array}}$$

★ Dijkstra does not work in → negative weight

$$② \xrightarrow{-2} ①$$



Infinite loop

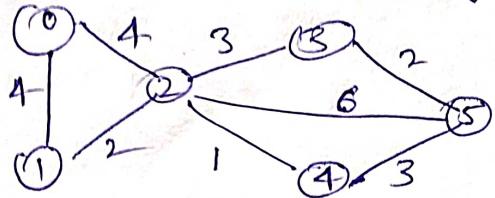
Dijkstra's is applicable for the weights.

Dijkstra's Algorithm (using Set)

We always wanted minimum distance at first, so
we used priority queue in previous approach.

Set : Set stores unique value and smallest value

at the top (ascending order)



30.05
24.44
24.25
10.55
27.33
15.55

Set

node = 0 $\xrightarrow{+4}$ node = 1
 $d = 4$
 $\xrightarrow{+4}$ node = 2
 $d = 4$
node = 4 $\xrightarrow{+4}$ node = 0
 $d = 8$
 $\xrightarrow{+2}$ node = 2
 $d = 6$

node = 4 $\xrightarrow{+1}$
 $d = 5$
 $\xrightarrow{+3}$ node = 5
 $d = 8$

dist. nodes

dist[] = $\boxed{10.55 \times 41 \times 4 \times 7 \times 5 \times 10}$

Step 0: — Logary (2000 iterations)

But we are saving iterations

Code :-

```
vector<int> dijkstra (int v, vector<vector<int>> adj[], int s)
{
    set<pair<int, int>> st;
    vector<int> dist (v, 1e9);
    st.insert ({0, s});
    dist[s] = 0;

    while (!st.empty())
    {
        auto it = * (st.begin ());
        int node = it.second;
        int dis = it.first;
        st.erase (it);

        for (auto it : adj[node])
        {
            int adjNode = it[0];
            int edgw = it[1];

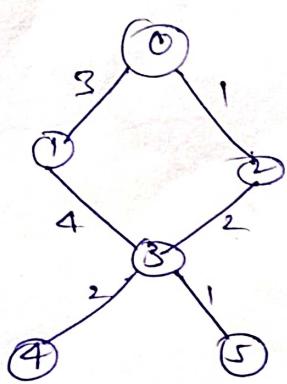
            if (dis + edgw < dist[adjNode])
            {
                if (dist[adjNode] != 1e9)
                    st.erase ({dist[adjNode], adjNode});
                dist[adjNode] = dis + edgw;
                st.insert ({dist[adjNode], adjNode});
            }
        }
    }
    return dist;
}
```

Dijkstra's Algorithm

Why PQ and not Q, intuition, Time Complexity Derivation

TIME complexity

$$O(E \log V)$$



dist[]

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

0 → {1, 3} {2, 1}

1 → {0, 2} {3, 4}

2 → {0, 1} {3, 2}

3 → {1, 4} {2, 2} {4, 2} {5, 1}

4 → {3, 2}

5 → {3, 1}

mode = 0

d = 0

+3
node = 1
d = 3

+1
node = 2
d = 1

mod = 1

d = 3

+0 / +4

mode = 0 mode = 3
d = 3 d = 2

mode = 2

d = 1

1 / 2
node = 0 mode = 3
d = 2 d = 3

mode = 3

d = 7

4 / 2 / 1
node 1 2 4 5
d = 1 9 9 8
x x

mode = 3
d = 3

* If we are using Q it means we are doing Brute force
we are traversing all the path.

* in set or priority queue its like we are using greedy algo.

Q will take lot more time complexity.

@Aashish Kumar Nayak

unnecessary
occurrence

Queue.

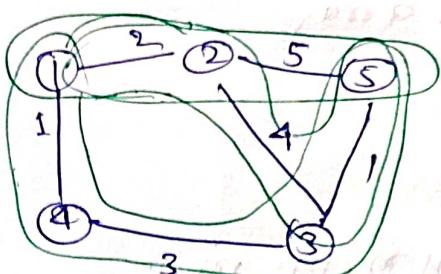
dist, node

If we use priority queue we explore minimal-minimal paths.

T.C. $E \log V$

no. of edges | no. of nodes

Print shortest path - Dijkstra's Algorithm :-



src = 1

dst = 5

8 paths

shortest

1-2-5

1-2-3-5

1-4-3-5

sum

7

7

5

✓
shortest
path

we need to print path

1-4-3-5

AHNT

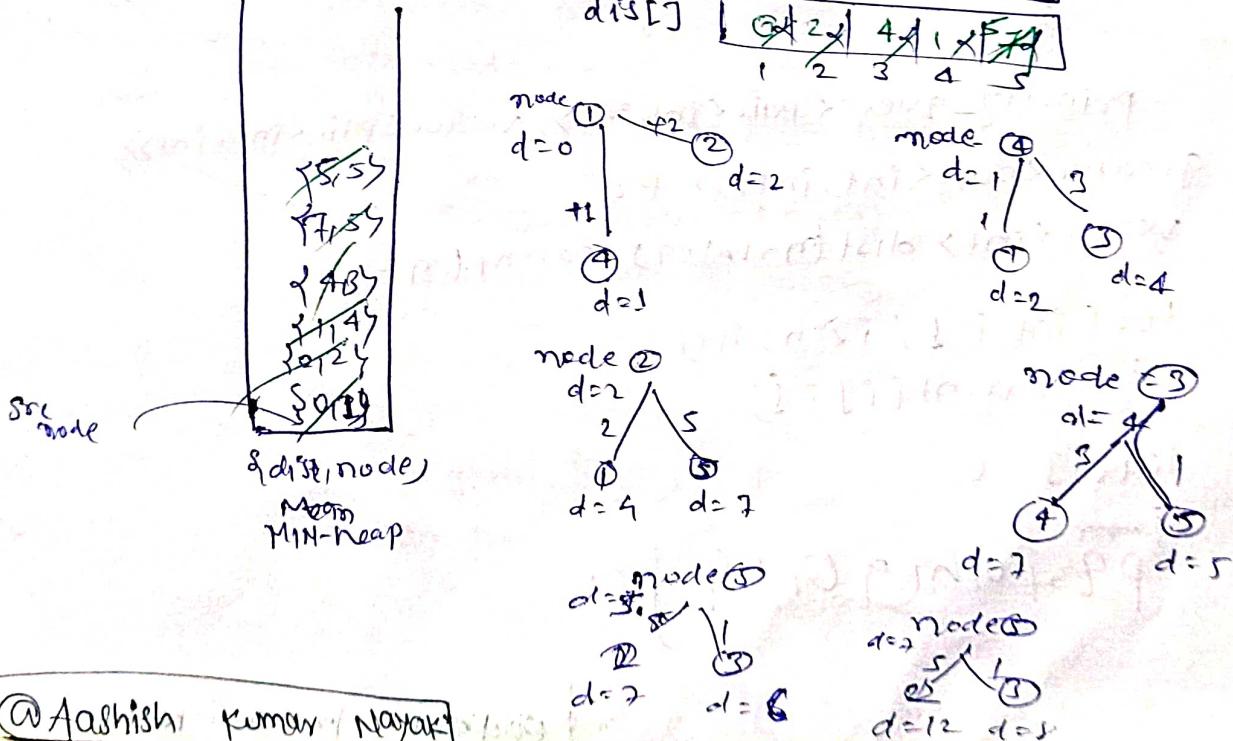
remember where I am coming from

then backtrack the path

priority-queue

parent[]

dist[]



$dist = 5$

$1 \rightarrow 4 \rightarrow 3 \rightarrow 5$
 | come from 3
 $dist[1] = 0$
 then stop
 do
 if node pt equals to src.
 then stop
 then we will store pt in array
 → Reverse the array
 → return the array

Code:

```

vector<int> shortestPath(int n, int m, 
vector<vector<int>> &edges)
{
  vector<pair<int,int>> adj[m+1];
  for (auto it : edges)
  {
    adj[it[0]].push_back({it[1], it[2]});
    adj[it[1]].push_back({it[0], it[2]});
  }

  priority_queue<pair<int,int>, vector<pair<int,int>>, 
  greater<pair<int,int>> pq;
  vector<int> dist(m+1, 1e9), parent(m+1);
  for (int i=1; i<=n; i++)
  {
    parent[i] = i;
  }
  dist[1] = 0;
  pq.push({0, 1});
  while (!pq.empty())
  {
    auto top = pq.top();
    pq.pop();
    int dis = top.first;
    int node = top.second;
    if (dis > dist[node])
      continue;
    for (auto it : adj[node])
    {
      int adjNode = it.first;
      int adjDis = it.second;
      if (adjDis + dis < dist[adjNode])
      {
        dist[adjNode] = adjDis + dis;
        parent[adjNode] = node;
        pq.push({dist[adjNode], adjNode});
      }
    }
  }
}
  
```

```
while(bpq.empty())
```

```
{
```

```
    auto it = pq.top();
```

```
    int node = it.second;
```

```
    int dis = it.second - it.first;
```

```
pq.pop();
```

```
for (auto it : adj[node])
```

```
{
```

```
    int adjNode = it.first;
```

```
    int edw = it.second;
```

```
    if (dis + edw < dist[adjNode])
```

```
{
```

```
        dist[adjNode] = dis + edw;
```

```
        pq.push({dist + edw, adjNode});
```

```
        parent[adjNode] = node;
```

```
}
```

```
,
```

```
if (dist[n] == reg)
```

```
    return -1;
```

```
vector<int> path;
```

```
int node = n;
```

```
while (parent[node] != node)
```

```
{
```

```
    path.push_back(node);
```

```
    node = parent[node];
```

```
}
```

```
path.push_back(1);
```

```
reverse(path.begin(), path.end());
```

```
return path;
```

```
}
```

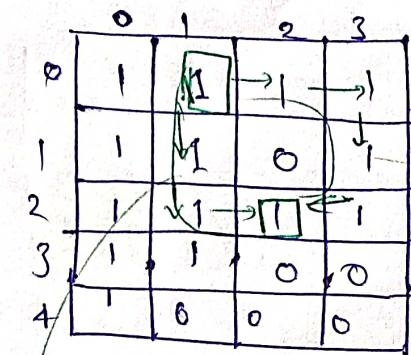
```
}{
```

5	0	1	1	1
1	0	1	1	1
0	0	1	1	1
0	0	0	1	1

T.C. = $O(N) + O(E \log V)$

$\approx O(E \log V)$

Shortest Distance in a Binary Maze



3 steps

src = {0,1}
dest = {2,2}
5 steps

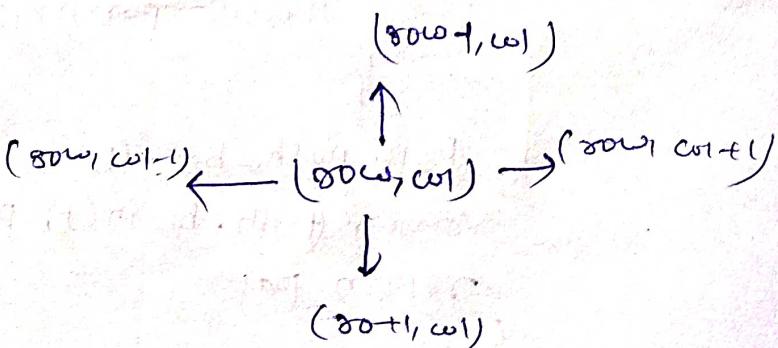
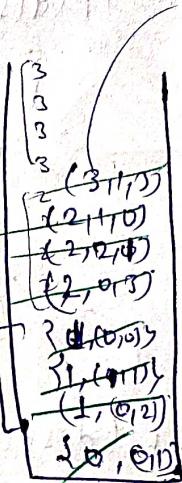
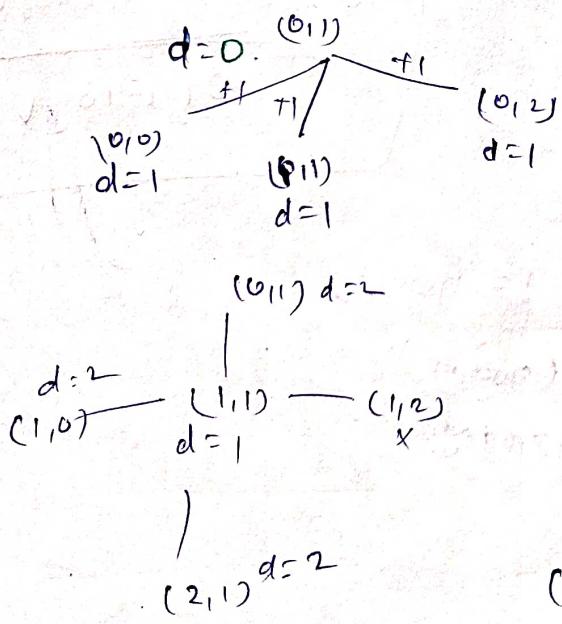
Dijkstra's Algo

dist[]				
0	1	2	3	4
X	0	X	X	X
2	X	X	3	X
X	2	X	X	X
X	X	X	X	X
X	X	X	X	X

return dist[3]

Ans

we are using
pq to take out
minimal optm.
distance but
here we will
find no ribbon
in Q or PQ
so we will
use φ here so
save T.C. $\log(M)$



Code

```
int shortestPath(vector<vector<int>>& grid, pair<int, int> source,
                  pair<int, int> destination)
{
    queue<pair<int, pair<int, int>> q;
    int n = grid.size();
    int m = grid[0].size();
    vector<vector<int>> dist(m, vector<int>(m, INT_MAX));
    dist[source.first][source.second] = 0;
    q.push({0, {source.first, source.second}});
    int dx[] = {-1, 0, 1, 0};
    int dy[] = {0, 1, 0, -1};
    auto it = q.front();
    q.pop();
    int dis = it.first;
    int r = it.second.first;
    int c = it.second.second;
    for(int i=0; i<4; i++)
    {
        int newr = r + dx[i];
        int newc = c + dy[i];
        if(newr >= 0 && newr < n && newc >= 0 && newc < m
           && grid[newr][newc] == 1 && dis + 1 < dist[newr][newc])
        {
            dist[newr][newc] = 1 + dis;
            if(newr == destination.first && newc == destination.second)
                return dis + 1;
            q.push({dis + 1, {newr, newc}});
        }
    }
    return -1;
}
```

Path with minimum Effort

(1)	2	2
3	8	2
5	3	(5)

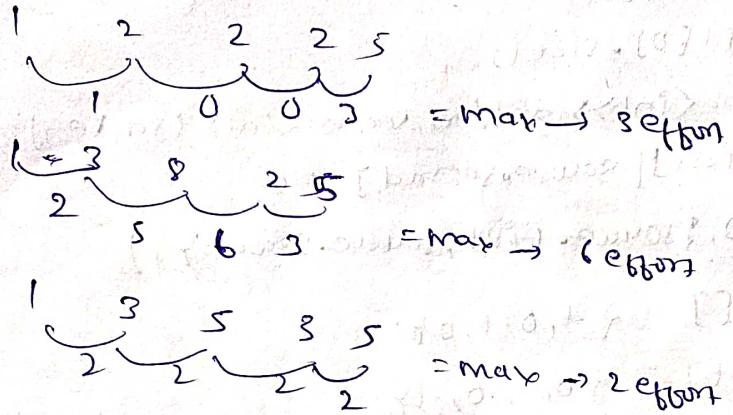


figure out ~~minimum~~ maximum effort from all the path and take the min ~~effort~~ of all max effort.

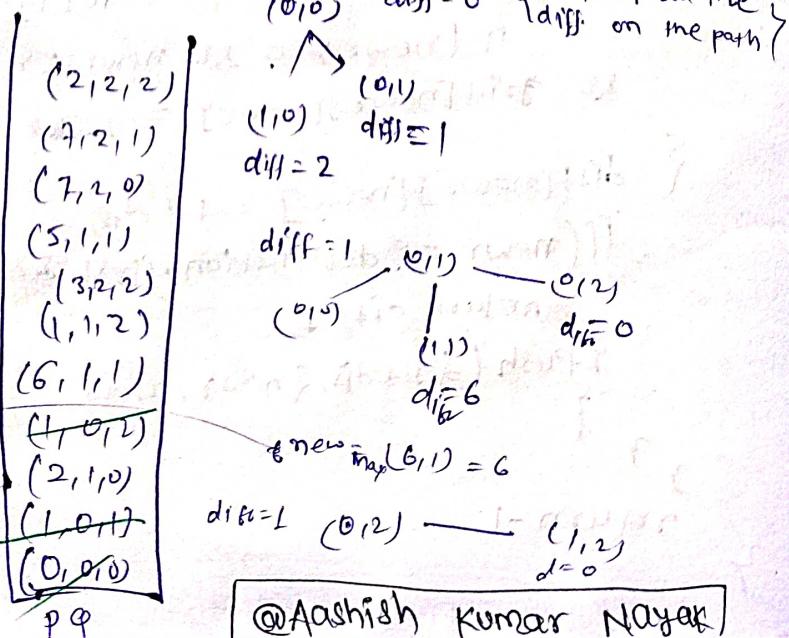
- * Something related to path, minimum, shortest path kind of thing then apply Dijkstra's Algorithm

~~diff[eff]~~

0	1	3
(1)	2	2
3	8	2

dist[i][j]

0	1	2
0	1	1
2	1	1



@Aashish Kumar Nayar

```

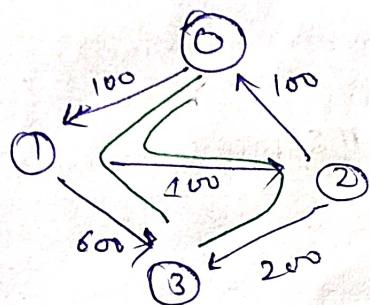
    } int minimumEffort (vector<vector<int>> &heights)
    {
        priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>> pq;
        greater<pair<int, pair<int, int>> > pq; // {diff, from, col}
        int n = heights.size();
        int m = heights[0].size();
        vector<vector<int>> dist (n, vector<int>(m, INT_MAX));
        dist[0][0] = 0;
        pq.push({0, {0, 0}});
        int dr[] = {-1, 0, 1, 0};
        int dc[] = {0, 1, 0, -1};
        while (!pq.empty())
        {
            auto it = pq.top();
            pq.pop();
            int diff = it.first;
            int row = it.second.first;
            int col = it.second.second;
            if (row == n-1 && col == m-1)
                return diff;
            for (int i=0; i<4; i++)
            {
                int newr = row + dr[i];
                int newc = col + dc[i];
                if (newr >= 0 && newc >= 0 && newr < n && newc < m)
                    if (newEffort = max(abs(heights[row][col] - heights[newr][newc]), diff);
                        if (newEffort < dist[newr][newc])
                            dist[newr][newc] = newEffort;
                            pq.push({newEffort, {newr, newc}});
            }
        }
        return 0; // unreachable.
    }

```

TC $O(E \log V)$
edges nodes

$\leq \approx O(N \times M)$

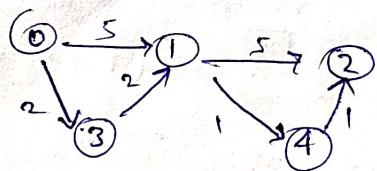
Cheapest Flights within K stops :-



$\text{src} = 0, \text{dest} = 3, K = 1$

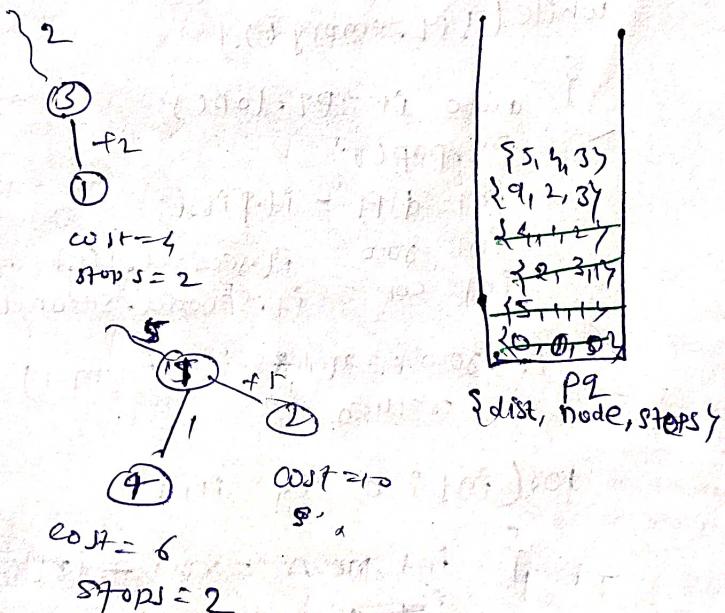
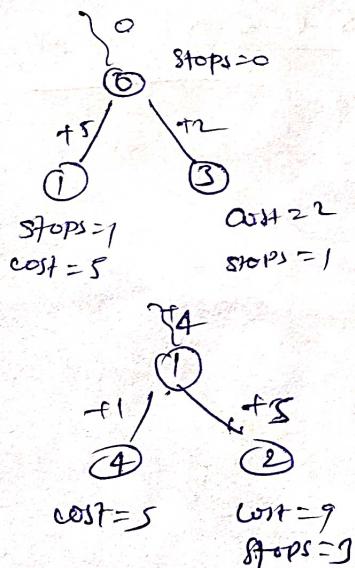
at max we can take K stops

Let's see why simple Dijkstra's will fail



cost	dist[0]	dist[1]	dist[2]	dist[3]	dist[4]
	0	5	9	2	5

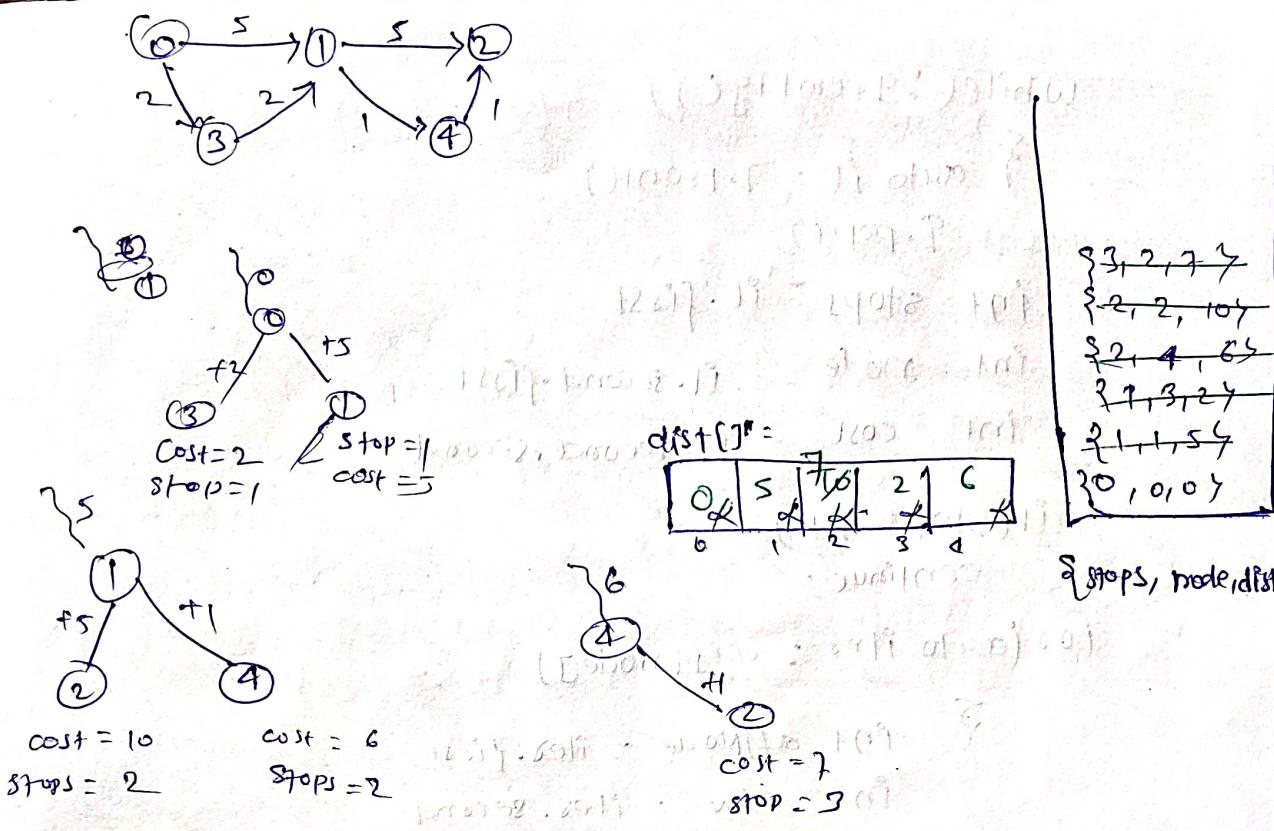
$\text{src} = 0$
 $\text{dest} = 2$
 $K = 2$



{ dist, node, stops }

This path has shorter distance
but more stops so
we can't choose this.

\therefore Dijkstra's Algo store everything in terms of
distance, but in this question we have to
store in terms of stops.



~~offer no need of priority queue because stops are increasing constantly.~~

we can extra $O(\log N)$

Code :-

```

int CheapestFlight( int n, vector<vector<int>> &flights,
                     int src, int dst, int k)
{
    vector<pair<int, int>> adj[n];
    for( auto it : flights)
    {
        adj[it[0]].push_back({it[1], it[2]});
    }
    queue<pair<int, pair<int, int>> q;
    q.push({0, {src, 0}});
    vector<int> dist(n, 1e9);
    dist[src] = 0;
}

```

```

while(!q.empty())
{
    auto it = q.front();
    q.pop();
    int stops = it.first;
    int node = it.second.first;
    int cost = it.second.second;

    if(stops > k)
        continue;

    for(auto iter : adj[node])
    {
        int adjNode = iter.first;
        int edw = iter.second;

        if(cost + edw < dist[adjNode] && stops <= k)
        {
            dist[adjNode] = cost + edw;
            q.push({stops + 1, {adjNode, cost + edw}});
        }
    }

    if(dist[dst] == 1e9)
        return -1;
    else
        return dist[dst];
}

```

T.C ~~O(E log V)~~

But we are not using PQ
so ~~O(E log V)~~

Sol. ~~O(E log V)~~

Flight size,

Minimum Multiplication to reach End

$$\underline{\text{start}} = 3 \quad \underline{\text{end}} = 80$$

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

$$\underline{\text{start}} = 3 \xrightarrow{x^2} 6 \xrightarrow{x^5} 80$$

② Ans

$$\underline{\text{start}} = 7 \quad \underline{\text{end}} = 66175$$

$$\text{arr}[] = \{3, 4, 654\}$$

$$7 \times 3 = 21$$

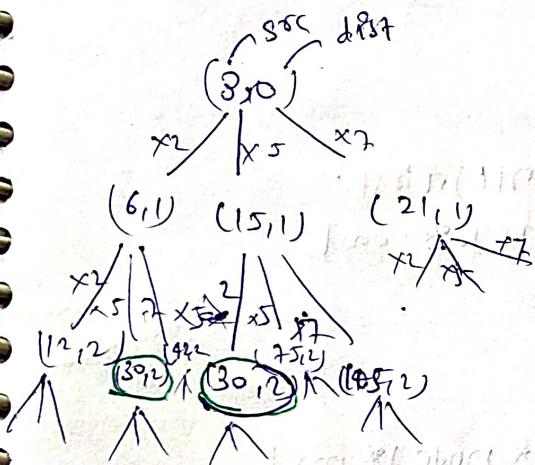
$$21 \times 3 = 63$$

$$63 \times 65 = 4095$$

$$4095 \times 65 = 266175$$

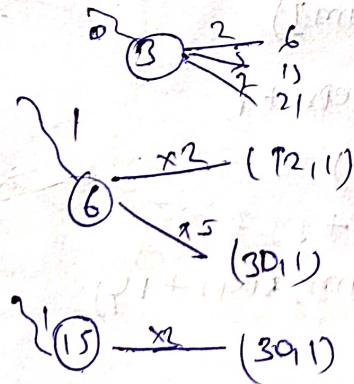
$$266175 \times 100000 = 266175$$

④ Ans



dist[]

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



- (2, 42)
- (4, 20)
- (4, 12)
- (1, 2)
- (1, 15)
- (1, 6)
- (0, 3)

∴ no need to use pq we will use queue.

Steps, num
min-heap

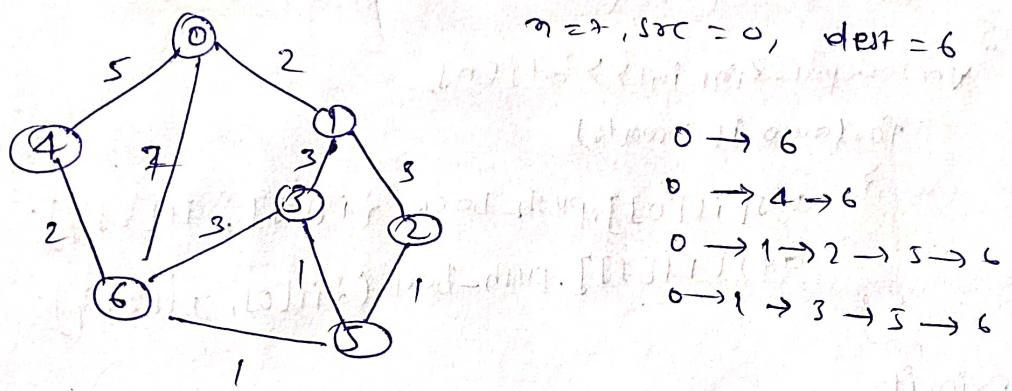
Code :-

```
int minimumMultiplication (vector<int>&arr, int start,
int end)
{
    queue<pair<int, int>> q;
    q.push({start, 0});
    vector<int> dist(100000, 1e9);
    dist[start] = 0;
    int mod = 100000;
    while (!q.empty())
    {
        int node = q.front().first;
        int steps = q.front().second;
        q.pop();
        for (auto it : arr)
        {
            int num = (it * node) % mod;
            if (steps + 1 < dist[num])
            {
                dist[num] = steps + 1;
                if (num == end)
                    return steps + 1;
                q.push({num, steps + 1});
            }
        }
    }
    return -1;
}
```

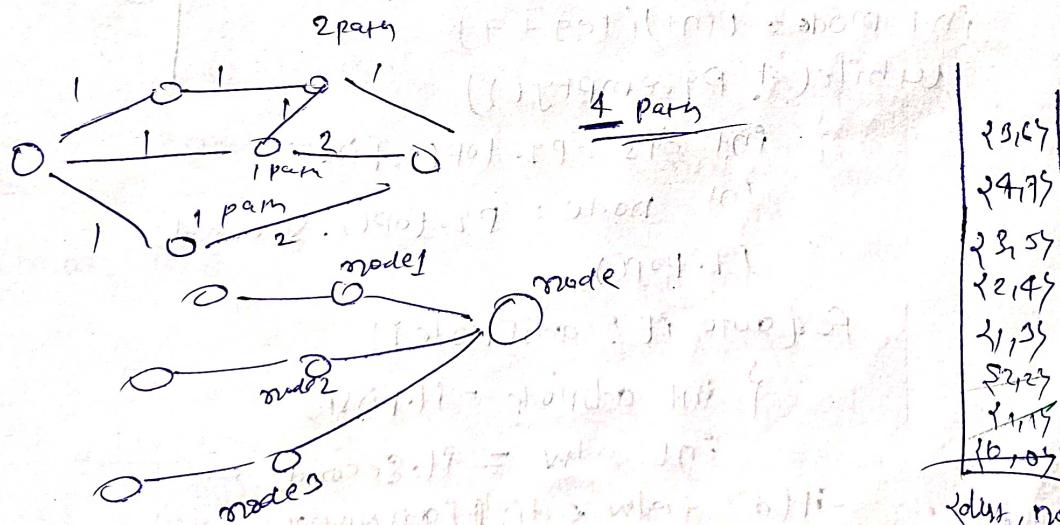
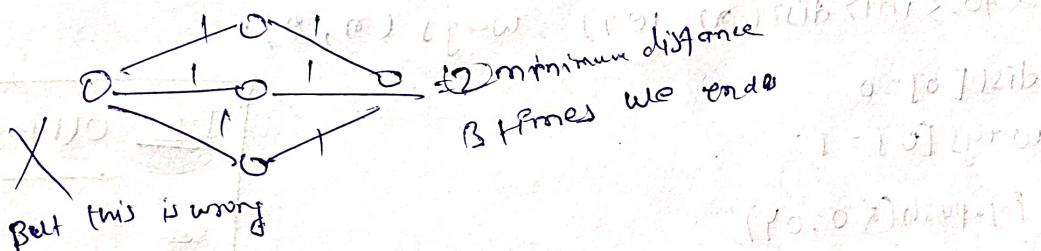
T.C. $O(100000 * N)$

@Aashish Kumar Nayak

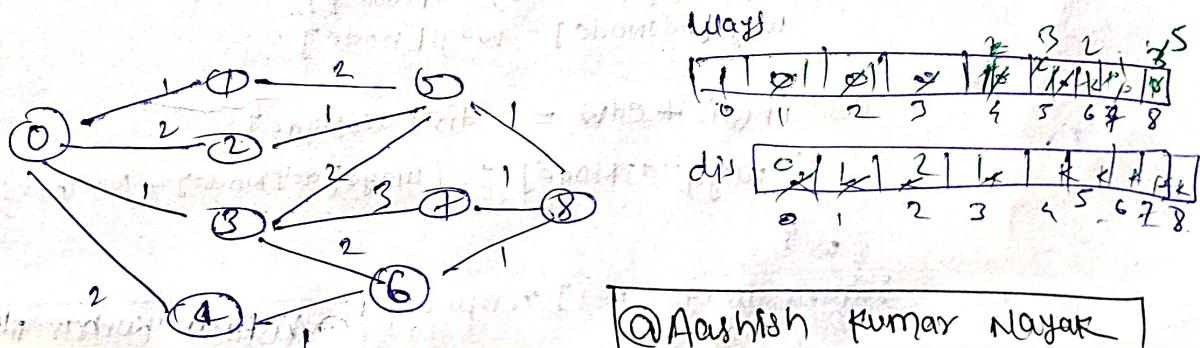
Number of ways to Arrive at Destination



No. of paths which are shortest. (There will be multiple path with same distance)
shortest



$\text{ways}[\text{node}] = \text{ways}[\text{node}[1]] + \text{ways}[\text{node}[2]] + \text{ways}[\text{node}[3]]$



@Aashish Kumar Nayak

Code :-

```
int countPaths(int n, vector<vector<int>> &roads)
```

{
 vector<pair<int, int>> adj[n];
 for(auto it : roads)
 {
 adj[it[0]].push_back({it[1], it[2]});
 adj[it[1]].push_back({it[0], it[2]});
 }
}

Priority Queue (pair<int, int>)
vector<pair<int, int>>, greater<pair<int, int>> pq;
vector<int> dist(n, 1e9), ways(n, 0);
dist[0] = 0
ways[0] = 1;
pq.push({0, 0});
int mod = (int)(1e9 + 7);
while(!pq.empty())
{
 int dis = pq.top().first;
 int node = pq.top().second;
 pq.pop();
 for(auto it : adj[node])
 {
 int adjNode = it.first;
 int edw = it.second;
 if(dis + edw < dist[adjNode])
 {
 dist[adjNode] = dis + edw;
 pq.push({dis + edw, adjNode});
 ways[adjNode] = ways[node];
 }
 else if(dis + edw == dist[adjNode])
 {
 ways[adjNode] = (ways[adjNode] + ways[node]) % mod;
 }
 }
}
return ways[n - 1] % mod;

Time $O(E \log V)$

Bellman Ford Algorithm (Algorithm for finding shortest path)

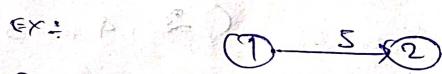
Need of Bellman Ford Algorithm

It helps us to detect negative cycles.

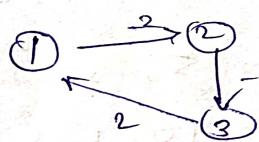
In Dijkstra's we can't calculate distance with -ve weight or -ve cycle.

But in Bellman Ford algo we can calculate path with -ve weight or -ve cycle.

This algo is applicable for \rightarrow DVs



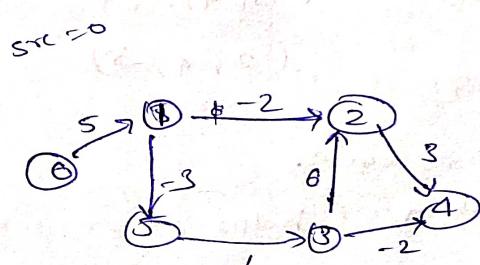
If we name
then change it to



Dijkstra will fail in TLE or infinity w.p.
path weight = $\rightarrow + 2 + 1 + 1 = -1$

if path weight < 0 it has negative cycle.

Bellman Ford Algo



④ This algo says Relax all the edges $N-1$ times sequentially

④ if ($dist[u] + w < dist[v]$).
Relax
 $dist[v] = dist[u] + w$

adj list
 (u, v, w)
 $(3, 2, 6)$
 $(5, 3, 1)$
 $(0, 1, 5)$
 $(1, 5, -3)$
 $(1, 2, -2)$
 $(3, 4, -2)$
 $(2, 4, 3)$

operation
for all edges
and it will
cause 5 iteration

@Aashish Kumar Nayak

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

v_i, v_j, w_{ij}

$$dist[3] + 6 < dist[2]$$

(3, 2, 6)

$$dist[5] + 1 < dist[5]$$

(5, 3, 1)

$$dist[0] + 5 < dist[1]$$

(0, 1, 5)

$$dist[1] - 2 < dist[2]$$

(1, 2, -2)

$$dist[3] - 2 < dist[4]$$

(3, 4, -2)

$$dist[2] + 3 < dist[4]$$

(2, 4, -3)

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

(3, 2, 6)

$$dist[5] + 1 < dist[3]$$

(5, 3, 1)

$$dist[0] + 5 < dist[1]$$

(0, 1, 5)

$$dist[1] - 3 < dist[5]$$

(1, 5, -3)

$$dist[1] - 2 < dist[2]$$

(1, 2, -2)

$$dist[3] - 2 < dist[4]$$

(3, 4, -2)

$$dist[2] + 3 < dist[4]$$

(2, 4, -3)

3rd iteration

① Why $N-1$ cycles

② How we detect negative cycle.

It helps to calculate 1, then it helps, it --- when
there are N nodes

for
(N-1) iteration



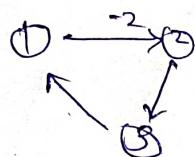
(N-1) edges

Hence $(N-1)$ cycles

since in a graph of N nodes, in worst case, you will take $N-1$ edges to reach from the first to the last, thereby we iterate for $N-1$ iteration.

Try drawing a graph which takes more than $N-1$ edges for any path, it is not possible.

Q. How to detect -ve cycle, because in every iteration we are decreasing path value.



On N th iteration the relation will be alone and the distance array get reduced.

We should get our answer till $(N-1)$ th iteration but if it is still reducing in N th iteration it means there is a negative cycle.

Code: `vector<int> bellman_ford(int v, vector<vector<int>> &edges, int s)`

```
{ dist[s] = 0; vector<int> dist(v, 1e8); dist[s] = 0;
  for (auto i = 0; i < v - 1; i++)
```

```
  { for (auto it : edges)
```

```
    { int u = it[0];
```

```
      int v = it[1];
```

```
      int wt = it[2];
```

```
      if (dist[u] != 1e8 && dist[u] + wt < dist[v])
```

```
        { dist[v] = dist[u] + wt;
```

```
    // Nth iteration to check negative cycle
    for (auto it : edges)
```

```
    { int u = it[0], v = it[1], wt = it[2];
```

```
      if (dist[u] != 1e8 && dist[u] + wt < dist[v])
        { return -1; }
```

```
}
```

```
return dist;
```

$T \approx O(N \times E)$

Nodes, edges

@Aashish Kumar Nayak

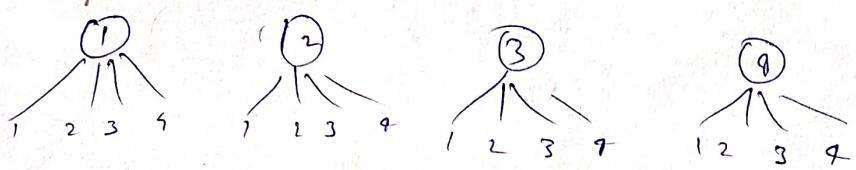
Floyd Warshall Algorithm (Algorithm for shortest path)

This is very-very different from Dijkstra & Belman Ford.

In these two previous Algo same point is 1.

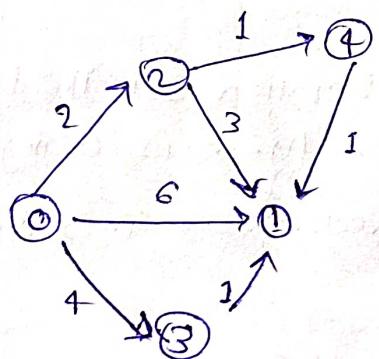
But in Floyd Warshall algo there is multiple source point.

We have to find shortest distance of all node from each node.



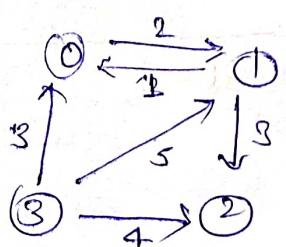
This is also known as multi-source shortest path algorithm.

Note: via every Nodes/vertex.



$$\begin{aligned}
 & (0 \rightarrow 1) = 6 \\
 & (0 \rightarrow 2) + (2 \rightarrow 1) = 2 + 3 = 5 \\
 & (0 \rightarrow 3) + (3 \rightarrow 1) = 4 + 1 = 5 \\
 & (0 \rightarrow 4) + (4 \rightarrow 1) = 1 + 1 = 2
 \end{aligned}$$

$$m \{ d[i][k] + d[k][j] \}$$



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

UOr \rightarrow Dh
 $0 \rightarrow 0$
 $0 \rightarrow 1$
 $0 \rightarrow 2$
 $0 \rightarrow 3$

$$[0][1] = 0 \rightarrow 0 + 0 \rightarrow 1$$

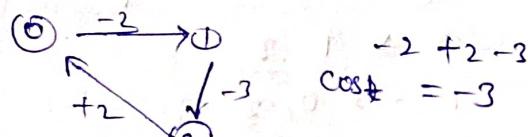
$$[0][2] = [0][0] + [0][2]$$

$$[1][2] = [1][0] + [1][2]$$

This algorithm is not much intuitive as the other ones. It is more of a brute force, where all combination of paths have been tried to get the shortest paths.

Nothing to be panic much on the intuition, it is a simple brute on all paths. Focus on the three for loops.

⑧ How to detect -ve cycle.



If the costing of any node $\text{cost}[i][i] < 0$ (node to node), it means there will have negative cycle.

Code: void shortest_distance (vector<vector<int>> &matrix)

```
{ int n = matrix.size();
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<n; j++)
        {
            if(matrix[i][j] == -1)
                matrix[i][j] = INT_MAX;
            if(i==j)
                matrix[i][j] = 0;
        }
    }
}
```

T.C. - $O(N^3)$

S.C. $O(N^2)$

matrix

```
for(int k=0; k<n; k++)
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<n; j++)
        {
            matrix[i][j] = min( matrix[i][j], matrix[i][k] +
                                matrix[k][j] );
        }
    }
}
```

// for negative cycle

```
for( int i=0; i<n; i++)
{
    if( matrix[i][i] < 0 )
        { // we can return or print something. }
```

```
for( int i=0; i<n; i++)
{
    for( int j=0; j<n; j++)
    {
        if( matrix[i][j] == INT_MAX )
```

```
        if( i==j )
            matrix[i][j] = -1;
```

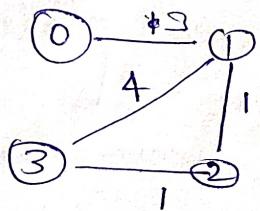
```
    } }
```

~~if(i==j)~~ matrix

@Aashish Kumar Nayak

City with the smallest no. of Neighbours

at a threshold distance.



threshold = 4

city

$$0 \rightarrow \boxed{1, 2}$$

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

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

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

lowest no. of neighbours

largest one is 1 & 3

Ans = 3

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

CNT = 0

CNTMAX = 5 \because we have 4 cities

CITY = -1

CNT MAX = 3

for 0 including itself

CNT = 1 & 3, so CNT = 3

for 1

CNT = 1 & 3, so CNT = 4

CNTMAX = MAX(CNT, CNTMAX)

for 2

CNT = 1 & 3, so CNT = 4

for 3

CNT = 1 & 3, so CNT = 3

CNTMAX = min(CNT, CNTMAX)

take largest one.

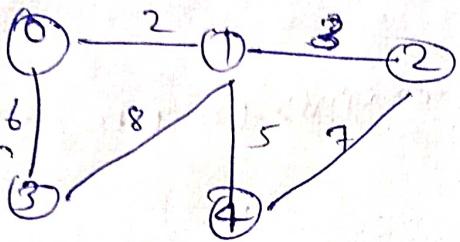
@Aashish Kumar Nayak

```

int findCity( int n, int m, vector<vector<int>> &edges,
int distanceThreshold )
{
    vector<vector<int>> dist( n, vector<int>(n, INT_MAX));
    for ( auto it : edges)
    {
        dist[ it[0] ][ it[1] ] = it[2];
        dist[ it[1] ][ it[0] ] = it[2];
    }
    for ( int i = 0; i < n; i++)
    {
        if ( dist[i][i] == 0 )
        {
            for ( int k = 0; k < n; k++)
                for ( int j = 0; j < n; j++)
                    if ( dist[i][j] == INT_MAX || dist[k][j] == INT_MAX )
                        continue;
                    else
                        dist[i][j] = min ( dist[i][j], dist[i][k] + dist[k][j] );
        }
    }
    int cntCity = n;
    int cityNo = -1;
    for ( int cityJ = 0; cityJ < m; cityJ++)
    {
        int cnt = 0;
        for ( int adjCity = 0; adjCity < m; adjCity++)
            if ( dist[cityJ][adjCity] <= distanceThreshold )
                cnt++;
        if ( cnt <= cntCity )
            cntCity = cnt;
            cityNo = cityJ;
    }
}

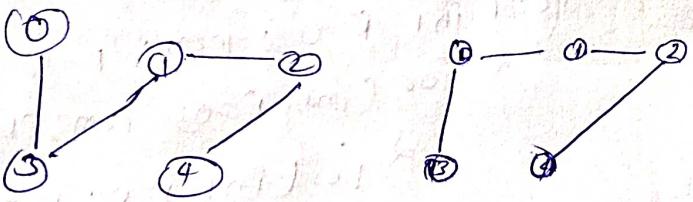
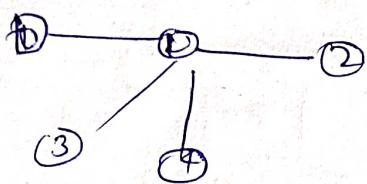
```

Minimum Spanning Tree (MST) :-

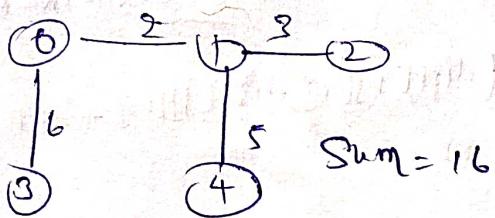


N-nodes (5)
M-edges (6)

A tree in which we have N nodes & $N-1$ edges & all nodes are reachable from each other.

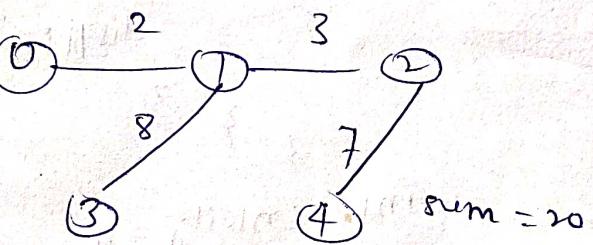


For this graph

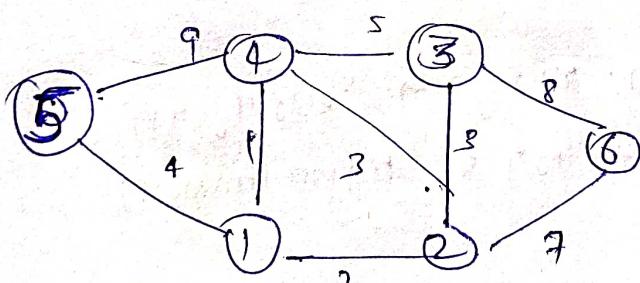


$$\text{Sum} = 16$$

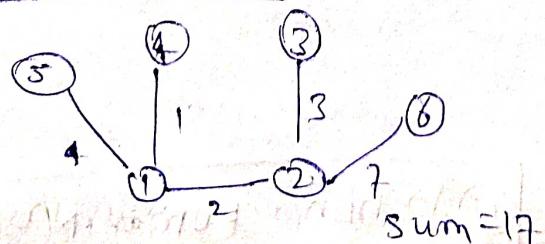
~~MST~~ Both are
Minimum spanning tree.



Spanning tree but just one is



Spanning trees are :-

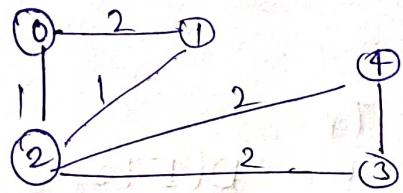
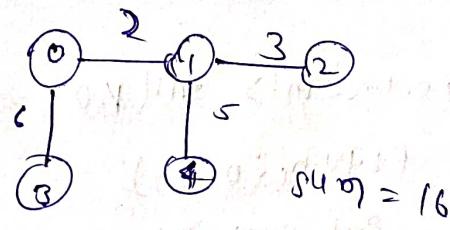
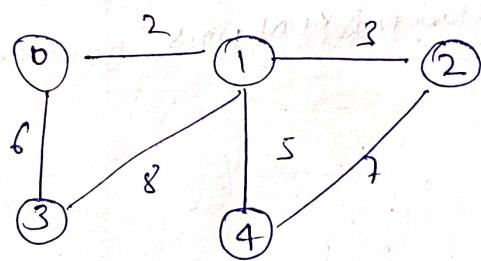


$$\text{sum} = 17$$

for finding MST
we have
 (i) Prim's Algorithm
 (ii) Kruskal's Algorithm
 ↓ Disjoint set

prim's Algorithm :-

It helps to find MST (Minimum Spanning Tree).



node parent it means its a 1st node
look at adjacent of node = 0
so don't add in MST

vis[]

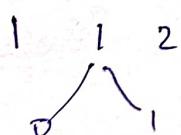
1	0	1	0	1
0	1	2	3	4

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

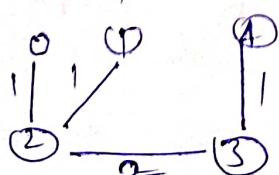
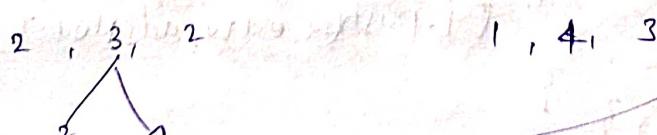


$$\text{sum} = 0 \times 2$$

least wt
always
(wt, node, parent)
Min-Heap



$$\text{MST} \rightarrow [(0,2)(1,2)(2,3)(3,4)]$$



Code :-

```
int spanningTree(int V, vector<vector<int>> adj[])
```

```
{ priority_queue<pair<int,int>, vector<pair<int,int>>, greater<int,int>> pq;
```

```
vector<int> vis(V, 0);
```

```
pq.push({0, 0});
```

```
int sum = 0;
```

```
while(!pq.empty())
```

```
{ auto it = pq.top();
```

```
pq.pop();
```

```
int node = it.second;
```

```
int wt = it.first;
```

```
if(vis[node] == 1)
```

```
continue;
```

```
vis[node] = 1;
```

```
sum += wt;
```

```
for(auto it : adj[node])
```

```
{ int adjNode = it[0];
```

```
int edw = it[1];
```

```
if(!vis[adjNode])
```

```
{ pq.push({edw, adjNode});
```

```
} }
```

```
return sum;
```

3

T.F

T.C.

$O(E \log E)$

Disjoint Set Data Structure

Important for Online Assessments.



1 & 5 are connected or not?

→ No (use BFS/DFS Traversal to find).

Time $O(N^2)$

i.e. Brute Force.

But Disjoint set will do this same on Constant time.
It gives

find parent()

constant time

for each

node

rank

size

t(1,2)

This function will connect ① & ②

(1,2)

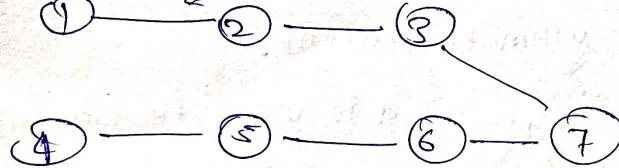
(2,3)

(4,5)

(6,7)

(5,6)

(3,7)



graph is changing and we can find parent or union or is two nodes are connected or at any stage.

Union() → rank
→ size we can implement union in two ways
rank and size.

① Union() by rank

rank array

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

parent

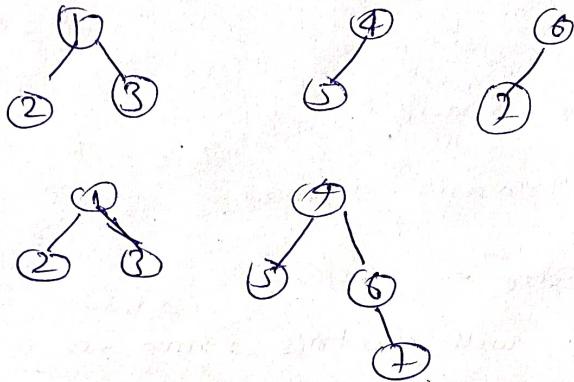
1	2	3	4	5	6	7
1	2	3	4	5	6	7

union(u, v)

1. find ultimate parent of u & v, p_u, p_v
2. find rank of ultimate parents
3. connect smaller rank to larger rank always.

same $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
 $\begin{smallmatrix} 9 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{smallmatrix}$

parents $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}$
 $\begin{smallmatrix} 8 \\ 7 \\ 6 \\ 5 \\ 4 \\ 3 \\ 2 \end{smallmatrix}$

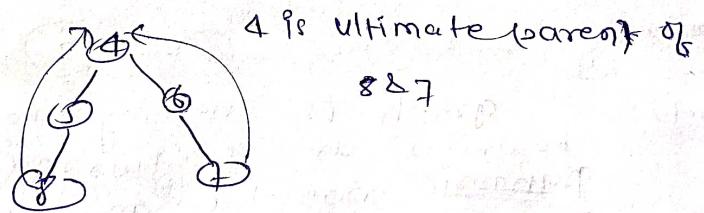


does 1 & 7 belong to same components?

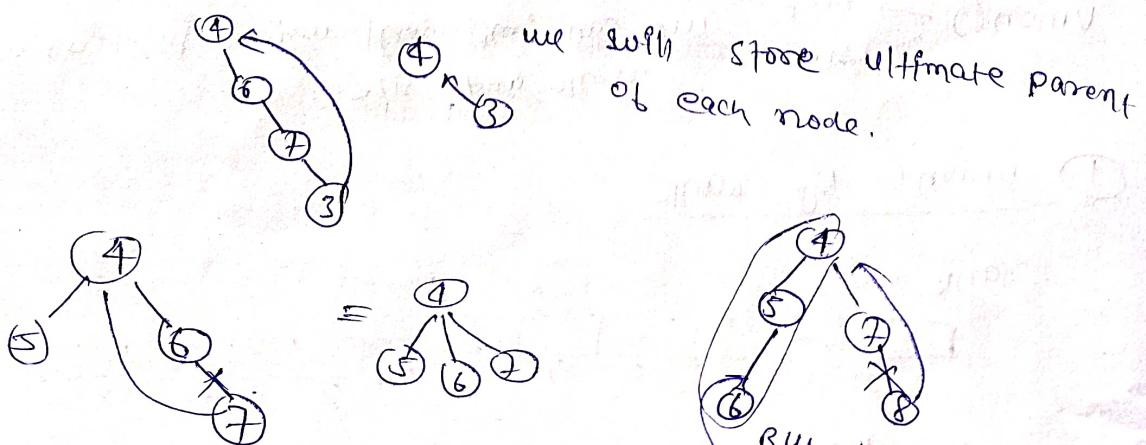
Ans = No., we will check parents of both 1 & 7
 \because they both are different

Both should have equal ultimate parent.

like

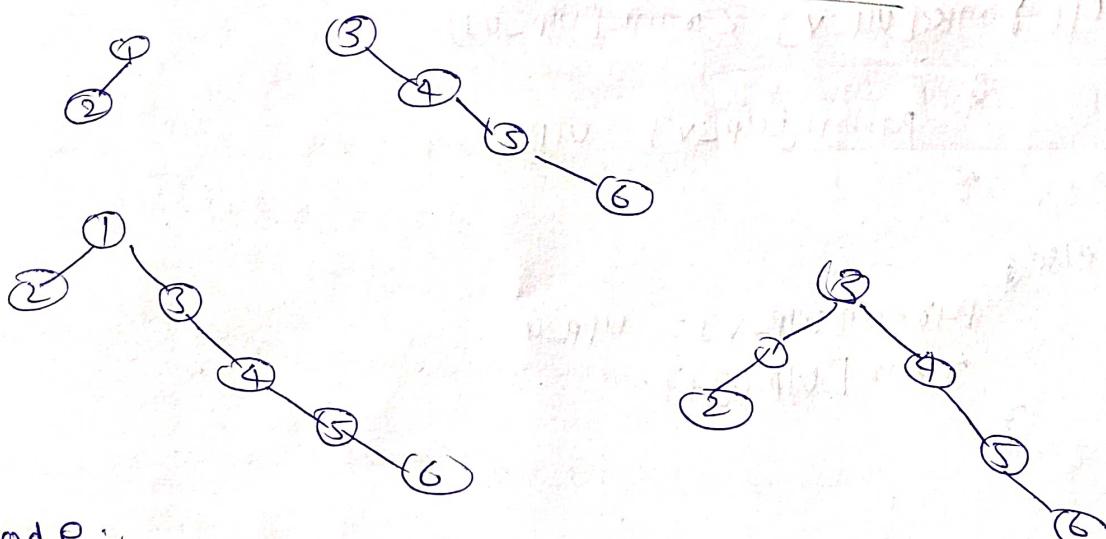


Path compression



* intuitively it is called rank.
 so it can't be reduced
 multi-height

Why connected to smaller to larger



Code :-

```
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n+1, 0);
        parent.resize(n+1, 0);
        size.resize(n+1);
        for(int i=0; i<n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if(node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void UnionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if(ulp_u == ulp_v)
            return;
        if(rank[ulp_u] < rank[ulp_v])
            parent[ulp_u] = ulp_v;
        else if(rank[ulp_u] > rank[ulp_v])
            parent[ulp_v] = ulp_u;
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }
};
```

```
else if( rank[ulp_v] < rank[ulp_u])
```

```
    {  
        parent[ulp_v] = ulp_u;  
    }
```

```
else,  
    parent[ulp_v] = ulp_u;  
    rank[ulp_u]++;
```

```
};
```

```
int main()
```

```
{
```

```
DisjointSet ds1(7), ds2(8);
```

```
ds1.unionByRank(1, 2);
```

```
ds1.unionByRank(2, 3);
```

```
    // (4, 5);
```

```
    // (6, 7);
```

```
    // (5, 6);
```

```
if(ds1.findUPar(3) == ds1.findUPar(7))
```

```
    cout << "Same";
```

```
else
```

```
    cout << "Not Same";
```

```
// ds1.unionByRank(3, 7);
```

```
// ds1.unionBySize(3, 7); > don't use Both simultaneously.
```

```
if(ds1.findUPar(3) == ds1.findUPar(7))
```

```
    cout << "Same";
```

```
}
```

```
else
```

```
    cout << "Not Same";
```

```
return 0;
```

```
}
```

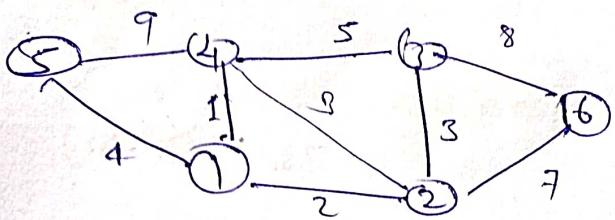
```

void unionBySize(int u, int v)
{
    int ulp_u = findUpPar(u);
    int ulp_v = findUpPar(v);
    if(ulp_u == ulp_v)
        return;
    if(rank[ulp_u] < rank[ulp_v])
    {
        parent[ulp_u] = ulp_v;
    }
    else if(rank[ulp_v] < rank[ulp_u])
    {
        parent[ulp_v] = ulp_u;
    }
    else
    {
        parent[ulp_v] = ulp_u;
        size[ulp_u] += size[ulp_v];
    }
}

```

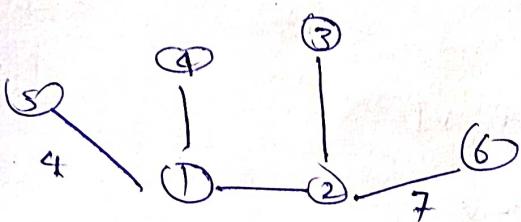
T.C. $O(4\alpha)$
 $\approx \text{constant}$

Kruskal's Algorithm



Sort all the edges w.r.t to w_i .
 (w_e, u, v)

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



Code :-

```

int spanningTree (int V, vector<vector<int>> adj[])
{
    vector<pair<int, pair<int, int>> edges;
    for (int i = 0; i < V; i++)
    {
        for (auto it : adj[i])
        {
            int adjNode = it[0];
            int weight = it[1];
            int node = i;
            edges.push_back({weight, {node, adjNode}});
        }
    }
    DisjointSet ds(V);
    sort(edges.begin(), edges.end());
    int mstWt = 0;
    for (auto it : edges)
    {
        int wt = it.first;
        int u = it.second.first;
        int v = it.second.second;
        if (ds.findPar(u) != ds.findPar(v))
        {
            ds.unionBySize(u, v);
            mstWt += wt;
        }
    }
    return mstWt;
}

```

```

if(ds.findUPar(u) == ds.findUPar(v))
{
    mstWt += wt;
    ds.unionBySize(u,v);
}
return mstWt;
}

```

class DisjointSets

```

vector<int> rank, parent, size;
public:

```

```
DisjointSet(int n)
```

```
{ parent.resize(n+1, 0);
```

```
rank.resize(n+1, 0);
```

```
size.resize(n+1, 1);
```

```
for(int i=0; i<n; i++)

```

```
{ parent[i] = i;
```

```
size[i] = 1;
```

```
int findUPar(int node)
```

```
{ if(node == parent[node])

```

```
    return node;
}
```

```
return parent[node] = findUPar(parent[node]);
```

```
void unionByRank(int u, int v)
```

```
{ int ulp_u = findUPar(u);
```

```
int ulp_v = findUPar(v);
```

```
if(ulp_u == ulp_v) return;
```

```
if(size[ulp_u] < size[ulp_v])

```

```
{ parent[ulp_u] = ulp_v;
```

```
size[ulp_v] += size[ulp_u];
```

```
else if(parent[ulp_v] == ulp_u)

```

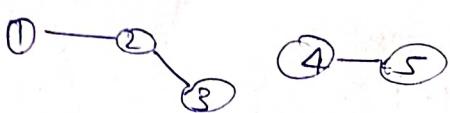
```
{ size[ulp_u] += size[ulp_v];
```

```
}
```

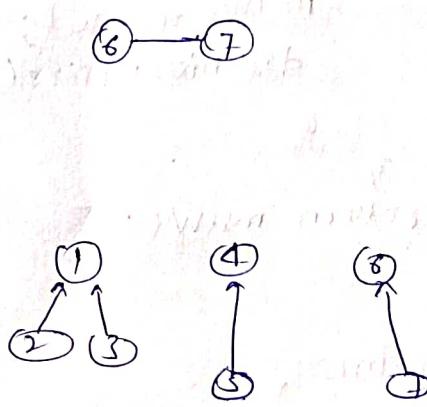
$\approx O(M \times 4 \times \alpha \times \gamma)$

$\approx O(M)$

No. of provinces



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



Count the no. of unique ultimate parents.

or if $\text{findUP}(v) == v$ then count

Code :- int numProvinces(vector<vector<int>> adj, int v)

{ DisjointSet ds(v);

for (int i = 0; i < v; i++)

{ for (int j = 0; j < v; j++)

{ if (adj[i][j] == 1)

{ ds.unionBySize(i, j);

}

cnt = 0;

for (int i = 0; i < v; i++)

{ if (ds.parent[i] == i)

cnt++;

}

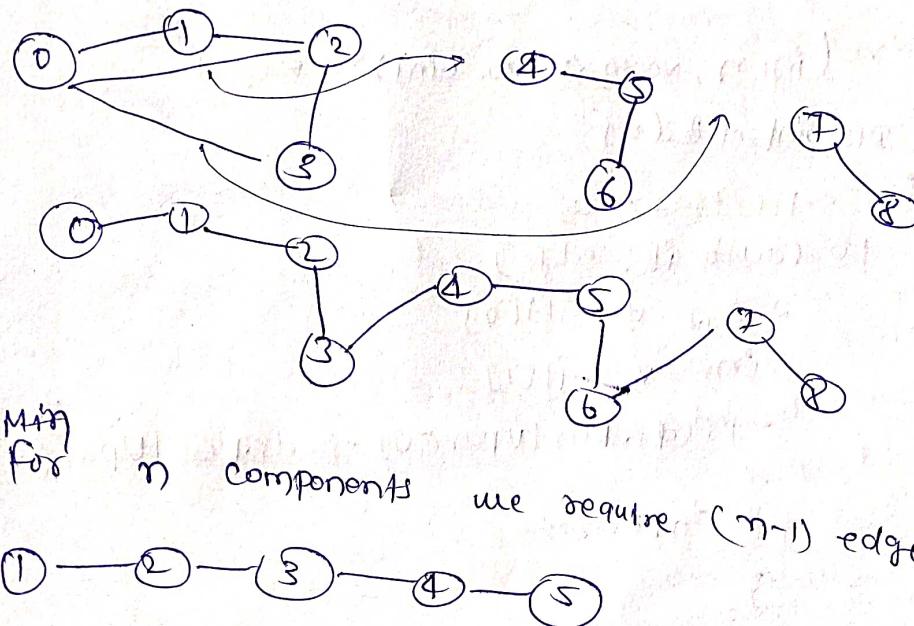
return cnt;

}

class DisjointSet

{}

Min no. of operations to make a graph connected



No. of connected components $\rightarrow nc$

then ans = $nc - 1$ and it will be always minimum.

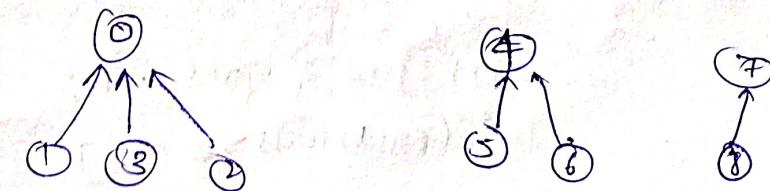
if (extraedges \geq ans)

return ans;

else

return -1;

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



Anyone who is pointing itself is an ultimate parent or counted as component.

class DisjointSet {

====

-}

int

solve (int n, vector<vector<int>> &edges)

{ DisjointSet ds(n);

int countExtras = 0;

for (auto it : edges)

{ int u = it[0];

int v = it[1];

if (ds.findUpPar(u) == ds.findUpPar(v))

{ countExtras++;

}

else

{ ds.unionBySize(u, v);

}

}

int countC = 0;

for (int i = 0; i < n; i++)

{ if (ds.parent[i] == i)

countC++;

}

int ans = countC - 1;

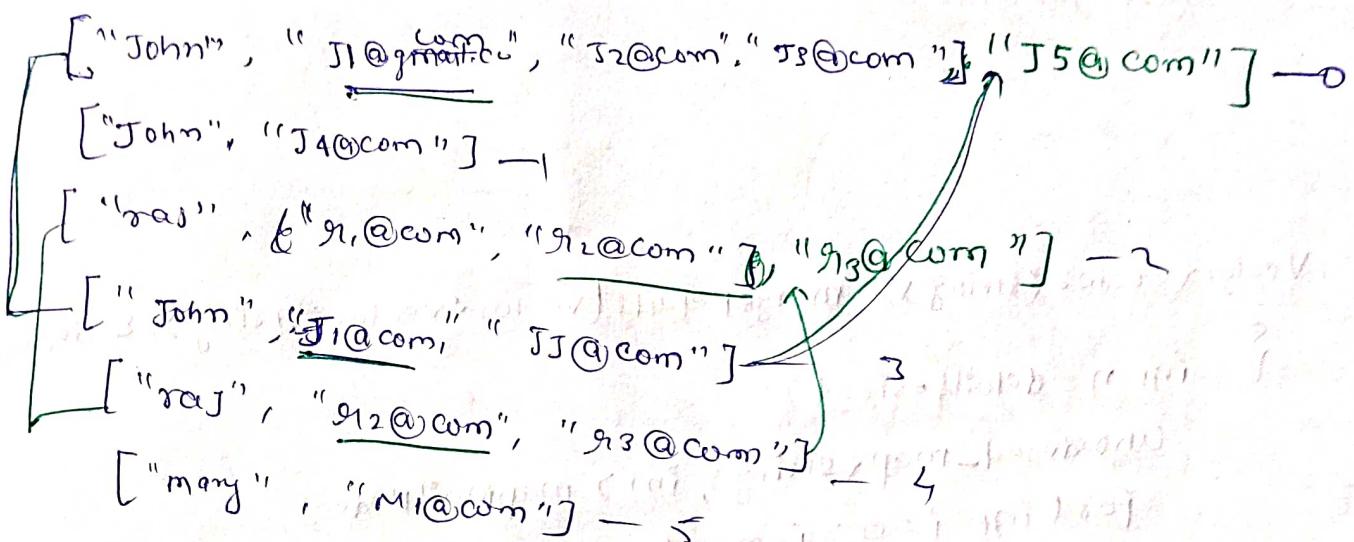
if (countExtras >= ans)

return ans;

return -1;

}

Account Merge - DSU



We are merging and disjoint data structure helps us to merging.

J1@com → 0

J2@com → 0

J3@com → 0

J4@com → 1

raj@com → 2

g2@com → 2

J5@com → 3

g3@com → 4

M1@com → 5

↓ ↓ ↓ ↓ ↓

① ② ③ ④ ⑤

↑ ↑ ↑ ↑ ↑

③ ④ ⑤ ① ②

0 → J1@com, J2@com, J3@com, J5@com

1 → J4@com

2 → raj@com, g1@com

3 → g2@com

4 → M1@com

When we are traversing in 3 index we saw "J1@com" but it is already a part of Index 0 so all rest of account of index 3 we will merge at index 0.
And so on.

Code :-

[@Apshish Kumar Nayak]

```
class DisjointSet*
```

```
{  
    =  
    }  
  
vector<vector<string>> mergeDetails(vector<vector<string>> &details)  
{  
    int n = details.size();  
    unordered_map<string, int> mapMailNode;  
    for (int i = 0; i < n; i++)  
    {  
        for (int j = 0; j < details[i].size(); j++)  
        {  
            if (mapMailNode.find(details[i][j]) == mapMailNode.end())  
            {  
                mapMailNode[details[i][j]] = i;  
            }  
            else  
            {  
                ds.unionBySize(i, mapMailNode[details[i][j]]);  
            }  
        }  
    }  
  
    vector<string> mergedMail[n];  
    for (auto it : mapMailNode)  
    {  
        string mail = it.first;  
        int node = ds.findUpPar(it.second);  
        mergedMail[node].push_back(mail);  
    }  
  
    vector<vector<string>> ans;  
    for (int i = 0; i < n; i++)  
    {  
        if (mergedMail[i].size() == 0) continue;  
        sort(mergedMail[i].begin(), mergedMail[i].end());  
        vector<string> temp;  
        temp.push_back(details[i][0]);  
        for (auto it : mergedMail[i])  
        {  
            temp.push_back(it);  
        }  
        ans.push_back(temp);  
    }  
    return ans;  
}
```

Number of Islands - II - online queries - DSU

	m				
0	1	2	3	4	
0	10	10	0	01	10
1	1	10	10	01	
2	0	0	0	0	0
3	0	0	10	0	0

groups

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

formula $(row, col) \equiv (row \times m) + col$

↓ node

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

ans: 11 2 3 2 3 3 2 4
 8 1 2 2 3 3 3 3 3 3
 2 1 2 2 3 3 3 3 3 3
 1 1 1 1 1 1 1 1 1 1

Class DisjointSet {

```

    i
    _____
    F
    _____
    y
    _____
    
```

```
bool isValid(int adjr, int adsc, int c, int m)
```

```
{ return adjr >= 0 && adsc < n && adsc >= 0 && adsc < m;
```

```
}
```

```
public: vector<int> numoflands(int n, int m, vector<vector<int>> operators)
```

```
{ DisjointSet ds(n * m);
```

```
int vis[n][m];
```

```
memset(vis, 0, sizeof vis);
```

```
int cnt = 0;
```

```
vector<int> ans;
```

```
for (auto it : operators)
```

```
{ int row = it[0];
```

```
, int col = it[1];
```

```
if (vis[row][col] == 1)
```

```
{ ans.push_back(cnt);
```

```
continue;
```

```
}
```

```
vis[row][col] = 1;
```

```
cnt++;
```

```
int dR[] = {-1, 0, 1, 0};
```

```
int dC[] = {0, 1, 0, -1};
```

```
for (int ind = 0; ind < 4; ind++)
```

```
{ int adjr = row + dR[ind];
```

```
int adsc = col + dC[ind];
```

```
If (isValid(adjr, adsc, n, m))
```

```
{ if (vis[adjr][adsc] == 1)
```

```
{ int nodeNo = row * m + col;
```

```
if (ds.findUparr() !=
```

```
int adjNodeNo = adjr * m + adsc;
```

```
if (ds.findUparr(nodeNo) != ds.findUparr(adjNodeNo))
```

```
{ cnt--;
```

```
ds.unionBySize(nodeNo, adjNodeNo);
```

```
ans.push_back(cnt);
```

```
return ans;
```

Making ~~connected~~ a Large Island - DSV

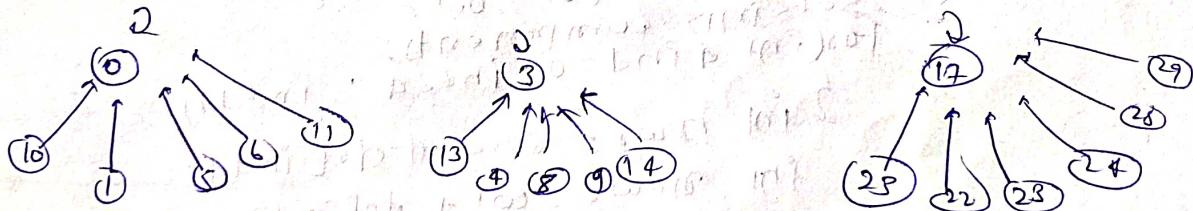
	0	1	2	3	4
0	1	1	0	1	1
1	1	1	0	1	1
2	1	1	1	1	1
3	0	0	1	1	0
4	0	1	1	1	1
5	0	0	1	1	1

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

$$\text{size} = 6 + 6 + 7 + 1 \\ = \underline{\underline{20}}$$

$$\text{size} = 10 + 1 \\ = \underline{\underline{11}}$$

We are using Disjoint Set into a single no. so that we can map it and use.
 formula = $(\text{row} * \text{m}) + \text{col}$



Class DisjointSet

Class Solution

public:

int maxConnection (vector<vector<int>> &grid)

{ int n = grid.size(); }

DSpointset ds(n*n);

for (int row = 0; row < n; row++)

{ for (int col = 0; col < n; col++)

{ if (grid[i][j] == 0)

continue;

int dr[] = {-1, 0, 1, 0};

int dc[] = {0, -1, 0, 1};

@Ashish Kumar Nayak

```

for(int int = 0; int < 4; int++)
{
    int newRow = row + dr[int];
    int newc = col + dc[int];
}

if(isValid(newr, newc, n) && grid[newr][newc] == -1)
{
    int nodeNo = row * n + col;
    int adjNodeNo = newr * n + newc;
    ds.unionBySize(nodeNo, adjNodeNo);
}

for(int int = 0; int < 4; int++)
{
    for(int col = 0; col < n; col++)
    {
        if(grid[row][col] == -1)
        {
            continue;
        }
        int dr = {-1, 0, 1, 0};
        int dc = {0, -1, 0, 1};
        set<int> components;
        for(int i = 0; i < 4; i++)
        {
            int newr = row + dr[i];
            int newc = col + dc[i];
            if(isValid(newr, newc, n) && grid[newr][newc] == -1)
            {
                int nodeNo = newr * n + newc;
                int adjNodeNo = newr * n + newc;
                ds.unionBySize(nodeNo, adjNodeNo);
            }
        }
    }
}

if(isValid(newr, newc, n) && grid[newr][newc] == -1)
{
    int nodeNo = row * n + col;
    int adjNodeNo = newr * n + newc;
    ds.unionBySize(nodeNo, adjNodeNo);
}

if(isValid(newr, newc, n))
{
    if(grid[newr][newc] == -1)
    {
        components.insert(ds.findUPar(newr * n + newc));
    }
    int sizeTotal = 0;
    for(auto it : components)
    {
        sizeTotal += ds.size[it];
    }
    mx = max(mx, sizeTotal);
}

```

for (int cellNO=0; cellNO < n*n; cellNO++)

{

 mxR = max(mr, ds.size[ds.findUPar(cellNO)]).

}

return mxR;

3: 3

$$T \leq O(N^2 \times \alpha)$$

Most Stones Removed with same row or column

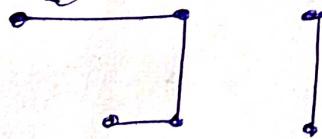
0	1	2	3
0	X		X
1			
2			
3	✓	X	
4			X

$$\text{ans} = 4$$

0	1	2	3	4
0	✓			
1		✓		
2			X	X
3			X	
4				✓

$$\text{ans} = 3$$

components Remove 1 element in each component



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



$$\begin{aligned}
 &= (1-1) + (1-1) + (4-1) \\
 &= 0 + 0 + 3 \\
 &= 3
 \end{aligned}$$

Q: If there are η stones

x_1
stones

x_2

stones

x_3

stones

x_4

stones

$$x_1 + x_2 + x_3 + \dots = \eta \text{ stones}$$

stones
are removed

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

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

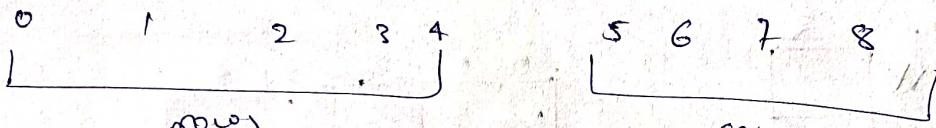
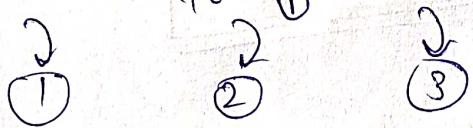
$$\text{ans} = \eta - \text{no. of components.}$$

$$\eta = \text{no. of stones}$$

Here we are connecting stones to make nodes.
so we use disjoint set data structure.

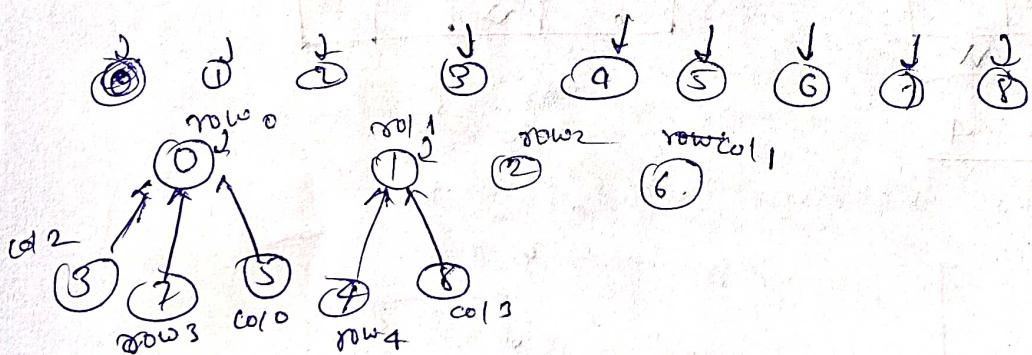
We will treat row as a node

If there is 1 stone in a row we will say it belongs to ①



Nodes in DS

Nodes in DS



2 connected components

For all the stones there is 2 ultimate parents

$$Ans = 6 - 2 = 4 \underline{Ans}$$

Code

We just need the nodes in disjoint set which are involved in having a stone.

So we store the rows and columns in map as they will have stones. And we just need to count them once for ultimate parents.

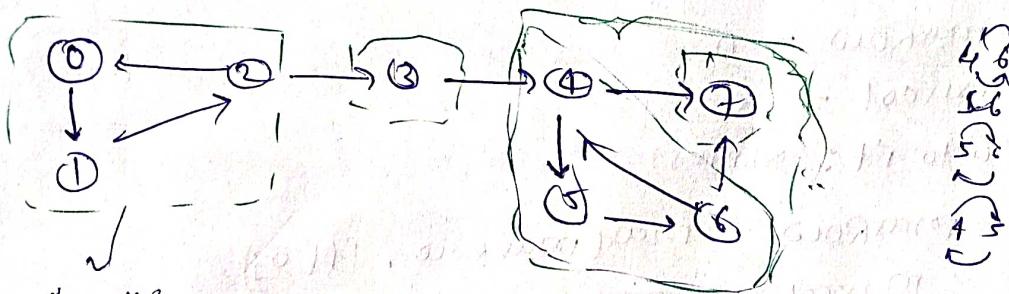
Code = Class DisjointSet
 fnt(mo)

```

int maxRemove(vector<vector<int>> &stones, int n)
{
  int maxRow = 0;
  int maxCol = 0;
  for (auto it : stones)
  {
    maxRow = max(maxRow, it[0]);
    maxCol = max(maxCol, it[1]);
  }
  DisjointSet ds(maxRow + maxCol + 1);
  unordered_map<int, int> stoneNodes;
  for (auto it : stones)
  {
    int nodeRow = it[0];
    int nodeCol = it[1] + maxRow + 1;
    ds.unionBySize(nodeRow, nodeCol);
    stoneNodes[nodeRow] = it;
    stoneNodes[nodeCol] = it;
  }
  int cnt = 0;
  for (auto it : stoneNodes)
  {
    if (ds.findUpPar(it.first) == it.first)
      cnt++;
  }
  return n - cnt;
}
  
```

Strongly connected components (SCC) :-

only valid for DG(Directed Graph) Kosaraju's Algorithm



in this component

every pair is reachable

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

Strongly connected component

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

4: Strongly connected component

- Sort all the edges according to finishing time.
- Reverse the graph
- Do a dfs.

top →

0

1

2

3

4

5

6

7

Code :-

```

void dfs(int node, vector<int>&vis, vector<int>&vt, vector<int> adj[])
{
    vis[node] = 1;
    for(auto id : adj[node])
    {
        if(!vis[id])
        {
            dfs(id, vis, vt, adj);
        }
    }
    vt.push_back(node);
}

```

```

void dfs3(int node, vector<int> &vis, vector<int> adjT[])
{
    vis[node] = 1;
    for (auto id : adjT[node])
        if (!vis[id])
            dfs3(id, vis, adjT);
}

```

$$\begin{aligned}
 \text{T.C.} &= O(N+E) \\
 &+ O(V+E) \\
 &+ O(V+E) \\
 &\approx O(N+E)
 \end{aligned}$$

S.C. $\approx O(N)$
 $\approx O(N+E)$

```

int Kosaraju(int v, vector<int> adj[])
{

```

```

    vector<int> &vis(v, 0);

```

```

    stack<int> st;
    for (int i = 0; i < v; i++)
        if (!vis[i])

```

```

            dfs(i, vis, adj, st);

```

```

    vector<int> adjT[v];
    adjT[v] = adj[v];

```

```

    for (int i = 0; i < v; i++)

```

```

        vis[i] = 0;

```

```

        for (auto id : adj[i])

```

```

            adjT[id].push_back(i);

```

```

    }

```

```

    int scc = 0;

```

```

    while (!st.empty())

```

```

        int node = st.top();

```

```

        st.pop();

```

```

        if (!vis[node])

```

```

            scc++;

```

```

        dfs3(node, vis, adjT);

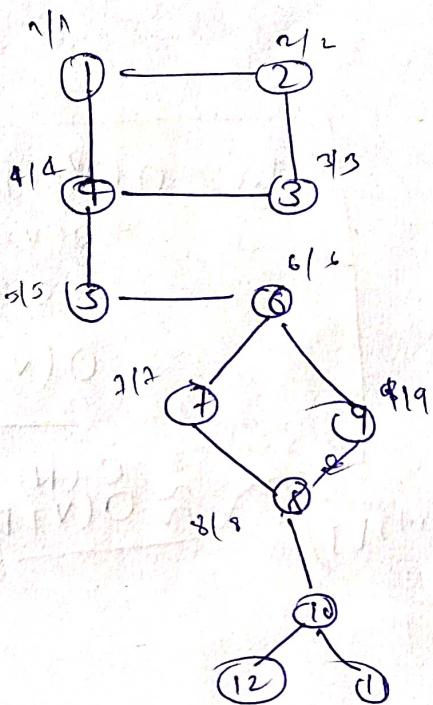
```

```

    }
    return scc;
}

```

Bridges in Graph



Any edge on whose removal the graph is broken down into two or more components.

4 → 5

5 → 6

10 → 8

$dfn[\cdot]$ → DFS time inserts on

$low[\cdot]$ → Min lowest time insertion of all adjacent nodes apart from parent.

DFS(8)

↓

DFS(9)

If I can reach you then it is not a bridge.

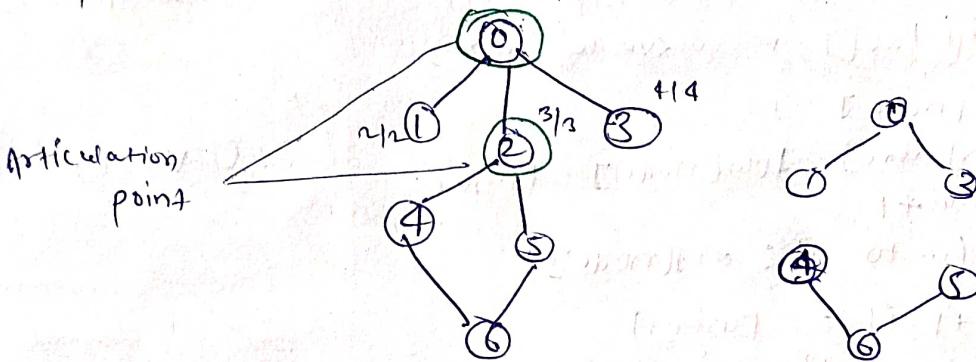
Code :-

```
int timer = 1;  
void dfs(int node, int parent, vector<int> &vis, vector<int> adj[],  
int tin[], int low[], vector<vector<int>> bridges,  
{  
    vis[node] = 1;  
    tin[node] = low[node] = timer;  
    time++;  
    for (auto it : adj[node])  
    {  
        if (it == parent)  
            continue;  
        if (vis[it] == 0)  
        {  
            dfs(it, node, vis, adj, tin, low);  
            low[node] = min(low[node], low[it]);  
            if (low[it] > tin[node])  
                bridges.push_back({it, node});  
        }  
        else  
            low[node] = min(low[node], low[it]);  
    }  
}  
vector<vector<int>> criticalConnections(int n, vector<vector<int>>&  
connections) {  
    vector<int> adj[n];  
    for (auto it : connections)  
    {  
        adj[it[0]].push_back(it[1]);  
        adj[it[1]].push_back(it[0]);  
    }  
    vector<int> vis(n, 0);  
    int tin[n], low[n];  
    vector<vector<int>> bridges;  
    dfs(0, -1, vis, adj, tin, low, bridges);  
    return bridges;  
}
```

$$T.C \approx O(V^2 E)$$

Articulation point :-

Nodes on whose removal the graph breaks into multiple components.



Time of insertion

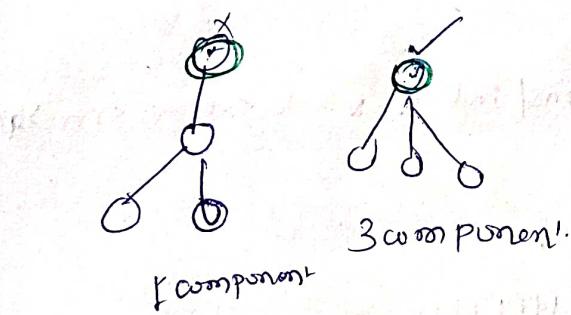
$T_{in}[] \rightarrow$ store the time of insertion during the BFS.

low

$low[] \rightarrow$ min of all adjacent nodes apart from the parent & visited nodes.

$\text{if } (low[i] \geq tm[\text{node}] \text{ & parent } b = -1)$

$dfs(2) \rightarrow 3$
 $dfs(3) \rightarrow 1$ if can reach to 3.



component should be ≤ 2

Code :-

```

int timer = 0;

void dfs(int node, int parent, vector<int> &vis, int tin[], int
low[], vector<int> &mark, vector<int> adj[])
{
    vis[node] = 1;
    tin[node] = low[node] = timer;
    timer++;
    int child = 0;

    for(auto it : adj[node])
    {
        if(it == parent)
            continue;
        if(!vis[it])
        {
            dfs(it, node, vis, tin, low, mark, adj);
            low[node] = min(low[node], low[it]);
        }
        else if(tin[it] >= tin[node] && parent != -1)
        {
            mark[node] = 1;
            child++;
        }
    }
    else
    {
        low[node] = min(low[node], tin[it]);
    }
}

if(child > 1 && parent == -1)
    mark[node] = 1;
}

public : vector<int> articulation_points(int n, vector<int>
adj[])
{
    int tin[n];
    int low[n];
    vector<int> mark(n, 0);
    for(int i=0; i<n; i++)
    {
        if(!vis[i])
            dfs(i, -1, vis, tin, low, mark, adj);
    }
}

```

```
vector<int> ans;
for (int i = 0; i < n; i++) {
    if (mark[i] == s)
        ans.push_back(i);
}
if (ans.size() == 0)
    return -1;
}
return ans;
```

END

@Aashish Kumar
Nayak