GRAPH

Handwritten Notes of Striver(TUF) Playlist

by: Aashish Kumar Nayak

NIT Srinagar

AASHISH KUMAR NAYAK

aashishkumar.nayak

1. Introduction to graph | Types | conversion used.
2. Graph representation | Two ways
3. What are connected components
4. BFS (Bredth first search)
5. DFS (Depth first search)
6. Number of Provinces | connected components
7. Number of islands | Number of connected components in 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 0/1 matrix.
13. Surrounded Regions | Replace O's with X's.
14. Number of enclaves | Multisource BFS.
15. Number of distinct island | constructive thinking + DFS
16. Bipertite Graph | BFS
17. Bipertite 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. Alien Dictionary | Topological sort
26. Shortest path in Directed Acyclic graph - Topological sort.
27. shortest path in undireted 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, intution 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 spaming 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. Accounts 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

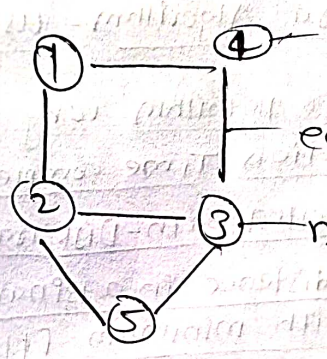
Lec 1

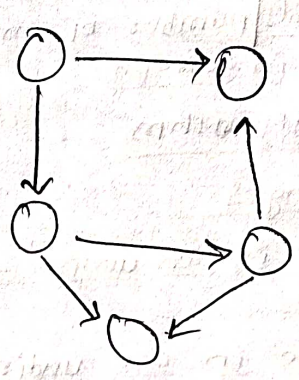
IN GRAPH

01-/04/2023

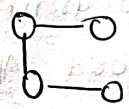
There are two types of graph

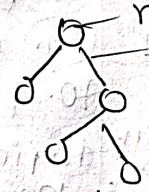
(i) undirected graph & (ii) Directed graph





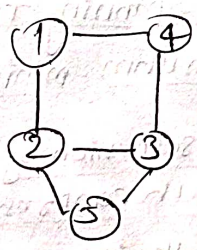
cycle in a graph



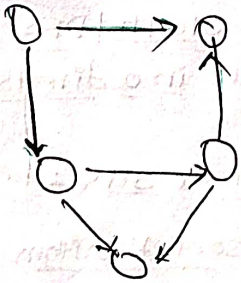


This is also graph

Def^n: Start from a node and end at that node it called cycle.

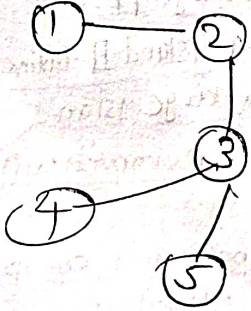


cyclic graph



Directed Acyclic graph (DAG)

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



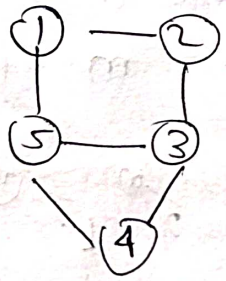
1 2 3 5 — Path

1 2 3 2 1 — Not a path

A node can't appear twice in a path.

1 3 5 ✗ Not a path

Degrees on Graph :-



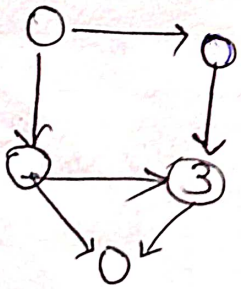
D(3) → 3
D(4) → 2
D(5) → 3
D(2) → 2
D(1) → 2

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

> Total Degree of a graph = 2x no. of edges

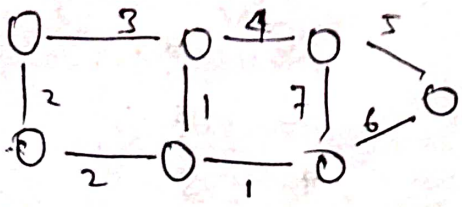
for directed graph :-

Indegree (node)
Outdegree (node)



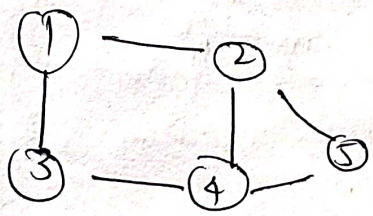
Indegree (3) = 2
Outdegree (3) = 1

edge weights :-



if weight is not assign unit weight

Lec 2



Input

n-nodes, m- edges

5 6

m lines → Represent edges

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

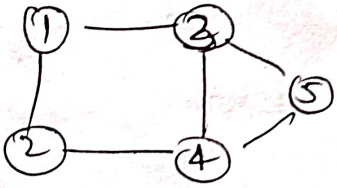
5 6

m lines

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

Two way for store

① Matrix ② List



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

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

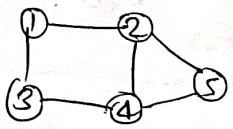
Space $n \times n$

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] = 1;
     adj[v][u] = 1;
   }
} return 0;
```

(ii) List :



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

Code :

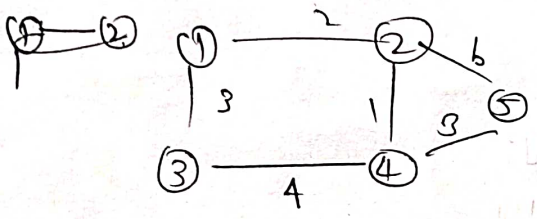
```
int main()
{ int n, m;
  cin>>n>>m;
  int ~~adj~~ vector<int> adj[n+1];
  for(int i=0; i<m; i++)
   { int u,v;
     cin>>u>>v;
     adj[u].push_back(v);
     adj[v].push_back(u);
   }
  return 0;
```

for directed graph

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

S.C. : O(Edges)

weighted graph :-



Matrix

	0	1	2	3
0				
1			2	
2		2		
3				

adj[v][v] = weight;

List

0 we will store pair.

1

2

3

4 → { 2 3, 5 } → instead of this store this

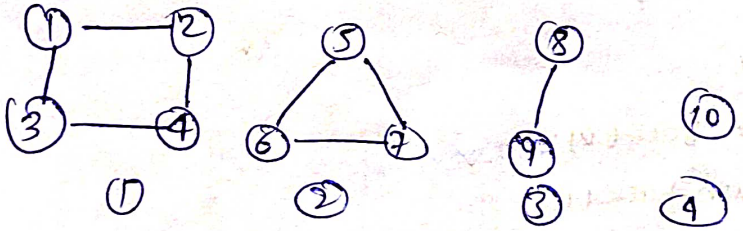
5

{ (2,1), (3,4), (5,3) }
weight

3. Lec 4.

Connected components :-

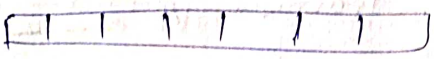
N = 10, M = 8



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

4 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 like visited array.

vis

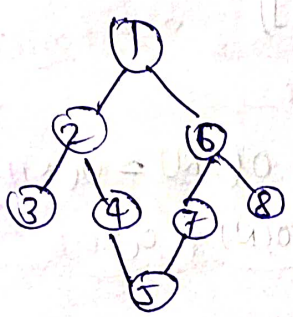
N=11

```
for (i=1 → 10)
    if(!vis[i])
        traversal(i);
```

for any traversal if node is not visited. // then call traversal also from this node.

A. Lees.

BFS (Breadth First Search)



level

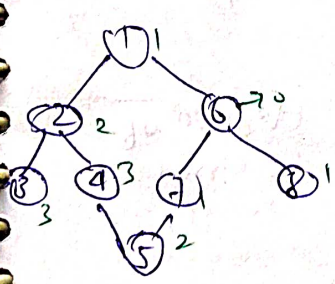
0
1
2
3

Breadth → level wise

1 2 6 3 4 7 8 5

0 1 2 3

let say 6 is starting node

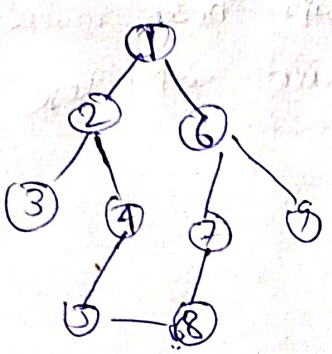


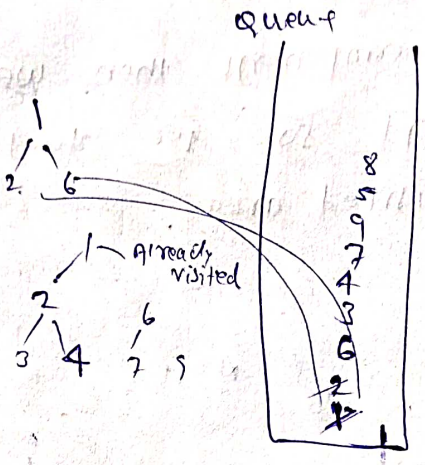
6 | 1 7 8 | 2 5 | 3 4

0 1 2 3

@Aashish Kumar Nayak

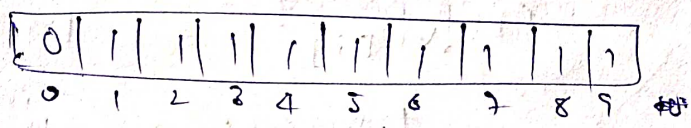
It is same as like level order traversal of an array/list



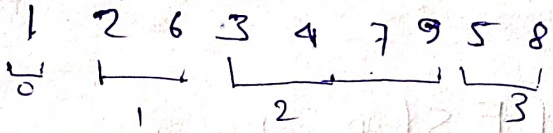


0 — {2, 6}
1 — {2, 6}
2 — {1, 3, 4}
3 — {2}
4 — {2, 5}
5 — {4, 8}
6 — {1, 7, 9}
7 —

vis



output



Code :

```
vector<int> bfsOfGraph(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(node);

        for(auto it : adj[node])
        {   if(!vis[it])
            {   vis[it] = 1;
                q.push(it);
            }
        }
    }
    return bfs;
}
```

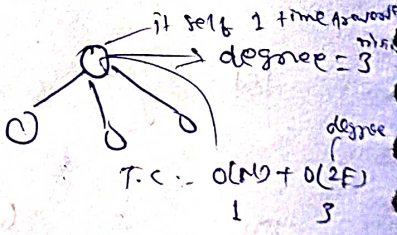
O(N) + O(2E)

~~N + 2E~~ 1 + (2E)

S.C. — O(3N) ≃ O(N)

T.C. — O(N) + O(2E)

2E = Total degree

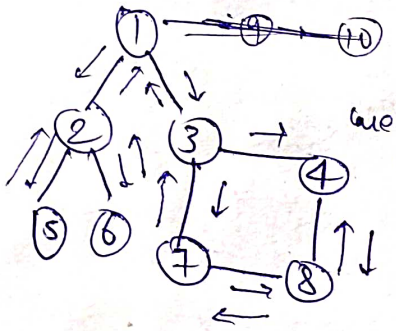


T.C. — O(N) + O(2E)
1 3

run for degree of node

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

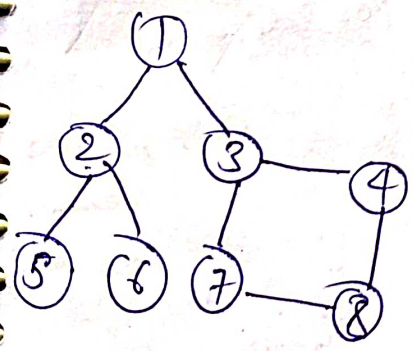
3 4 8 7 1 2 5 6

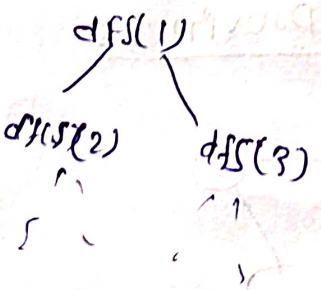
Recursion is an algo which goes till depth calculate and returns back.

So we use Recursion in DFS.

vis

Adjacency list :=





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

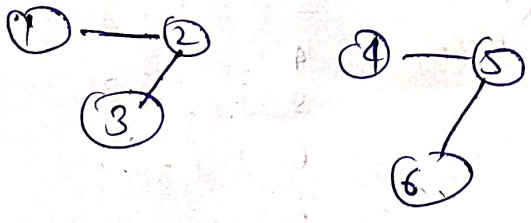
@Aashish Kumar Nayak

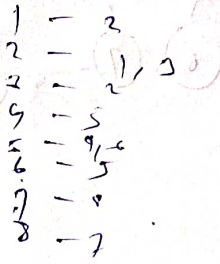
```
vector<int> dfsOfGraph(int V, vector<int> adj[])
{
    int vis[V] = {0};
    int start = 0;
    vector<int> ls;
    dfs(start, adj, vis, ls);
    return ls;
}

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. → O(N) + O(N) + O(N)
dfs visited Recursion
≈ O(N)

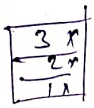
T.C. → O(N) + O(2E)
degree
↓ for directed graph
T.C. → O(E)

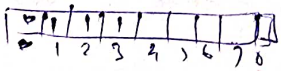
Ques. Number of Provinces :-





No. of provinces = 3





From node 1 we can go 2, 3 or, from ② → 1, 3, from ③ → ①, ② so ① — ② this is 1 province.
③

@Aashish Kumar Nayak

we will have multiple starting point

start = 1/2/3
start = 4/5/6
start = 7/8

~~Code~~

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

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;
}
```

T.C. $O(N) + O(V+2E)$ — for all dfs node
$\approx O(N)$

S.C. — $O(N) + O(N)$
Recursion stack space

1 — 2
2 — 1, 3
3 — 2

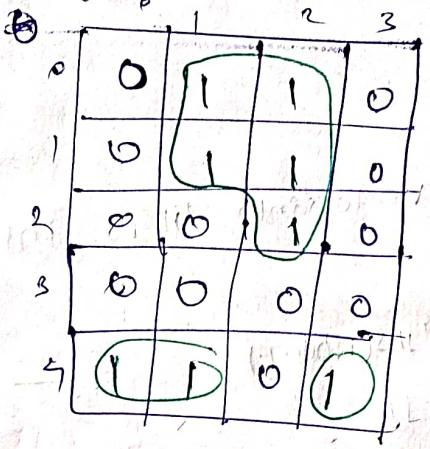
adjLs[i].push_back(j);

@Ashish Kumar Nayak

~~No. of component~~

Number of Islands

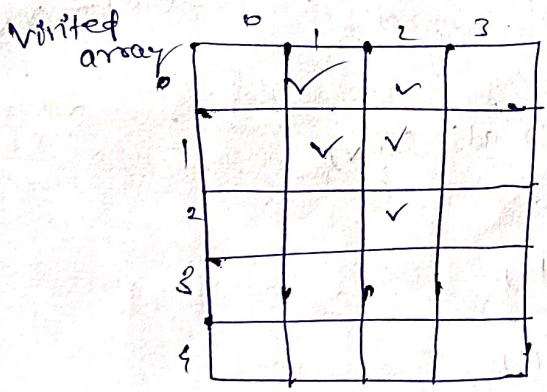
@Aashish kumar Nayak

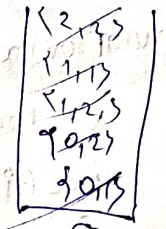


No. of Island
3. Starting Nodes
& Island

Using BFS

Starting → {0,1}





adjacency list

They can be neighbour so we will go in all 8-direction.

Then after 1 traversal the queue will be empty

Now Starting point will change and process will be same.

So we will use three starting points.

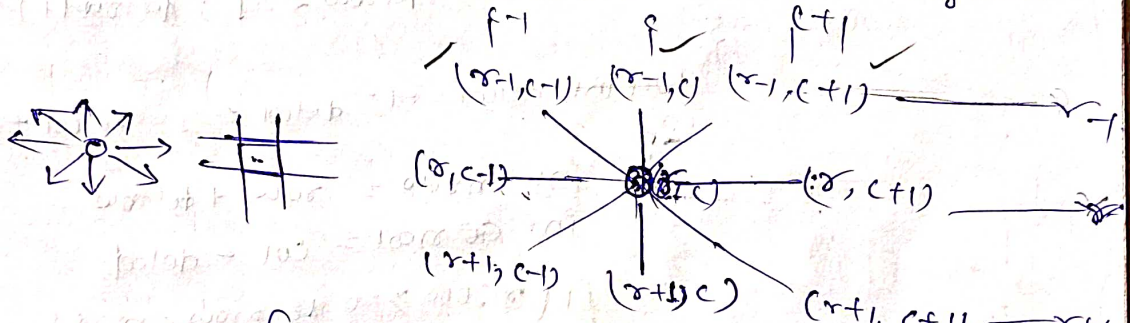
for finding starting points we will traverse from (0,0) to (n-1, n-1) and if we find 0 then ignor while

if we find 1 then call traversal also.

pseudo code :-

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

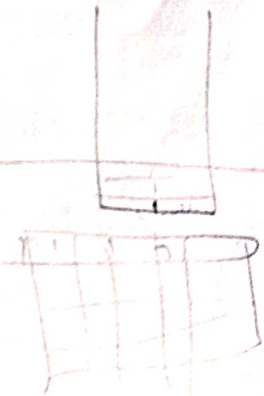
for BFS traversal we have to visit all the neighbour nodes.



for that we can write a code

```
for(delrow -1 to 1)
  {
    for(delcol -1 to 1)
      {
        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});
            }
        }
    }
}
```



@Aashish Kumar Nayak

```
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;
}
```

S.C. :- $O(N^2) + O(N^2)$
matrix queue
$\approx O(N^2)$

T.C. :- ~~N~~

$N^2 + N^2 \times 9$
matrix check for 1 / BFS

$\approx O(N^2)$

In matrix ~~grid~~ BFS

Lec 9

Flood Fill Algorithm

Sol by DFS

sr, sc — starting coordinate. DFS(2,0)

initial = 2

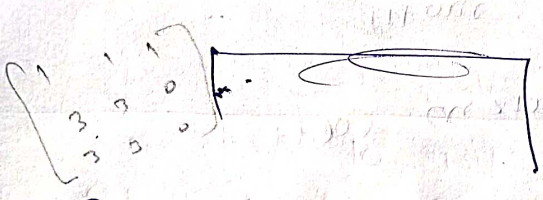
new color = 3

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

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

DFS(1,1)

DFS(2,1)



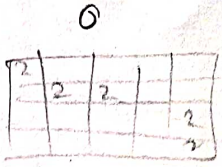
We can choose BFS / DFS any of them

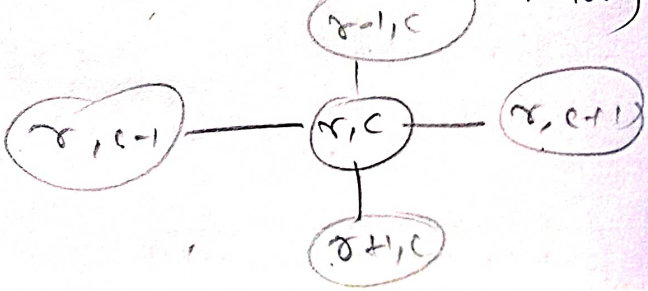
But we choose DFS here for no reason

Initial color will be image[sr][sc]

Code :-

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

{ int iniColor = image[sr][sc];   // 2
  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, iniColor);
  return ans;
}
```





```
void dfs(int row, int col, vector<vector<int>> &ans, vec
vector<vector<int>> &image, int delrow[], int delcol[]
{
    ans[row][col] = newColor;
    int n = image.size();  int m = image[0].size();
    for(int i = 0; i<4; i++)
    {
        int nrow = row + delrow[i];
        int ncol = col + delcol[i];
        if(nrow >=0 && nrow<n && ncol >=0 &&
        nrow<m && image[nrow][ncol]
==inicolor && ans[nrow][ncol] != newcolor)
        {
            dfs(nrow, ncol, ans, image, newcolor, delrow,
            delcol);
        }
    }
}
```

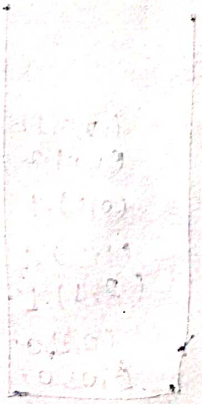
T.C. $(N \times M) \times 4$ — matrix for every, 4 neighbours node

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

call dfs for $N \times M$ $\approx O(N \times M)$

S.C. $O(N \times M) + delrow + delcol$

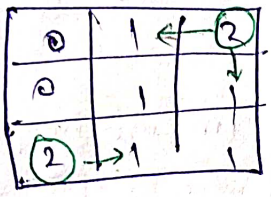
$\approx O(N \times M)$



@Aashish Kumar Nayak

Rotten oranges :-

we have to ~~find~~ the minimum time to rotten all the oranges.



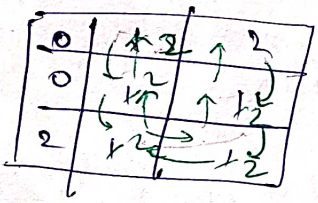
0 — Empty box
1 — Fresh orange
2 — rotten orange

* in 1 unit of time if it can rott ← ↑ 0 ↓ → 4 oranges if available.

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

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

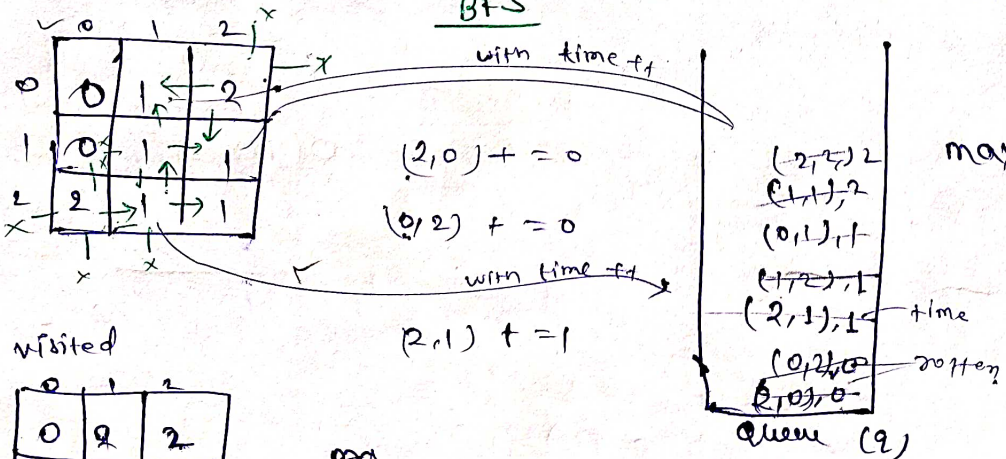
if we use DFS



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 equal pace.

BFS simultaneously rotten



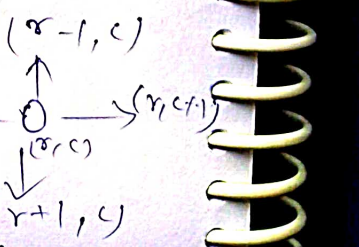
visited

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

ma

Max time = 2 sec

return max time,



@Aashish Kumar Nayak

@Aashish Kumar Nayak

```
int orangesRotting(vector<vector<int>>& grid)
{
    int n = grid.size();
    int m = grid[0].size();

    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 dcol[] = {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 + dcol[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. $O(N \times M)$ + $O(N \times M) \times 4$
(loop; every node; neighbours)
$\approx O(N \times M)$

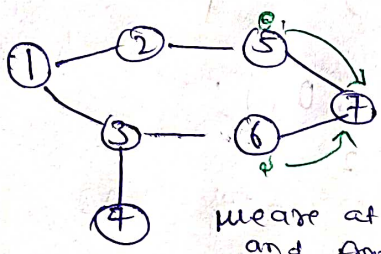
S.C. $O(N \times M) + O(N \times M)$
(vis; queue)
$\approx O(N \times M)$

```
if(cnt != cntFresh)
    return -1;

return tm;
}
```

Detect a cycle in an undirected graph :-

Using BFS

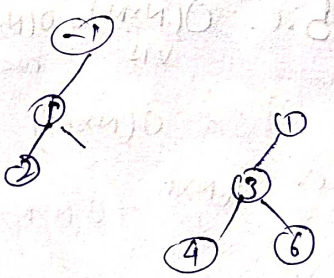
if you are starting traverse and you are able to come back at that particular node then there will be cycle in the graph.

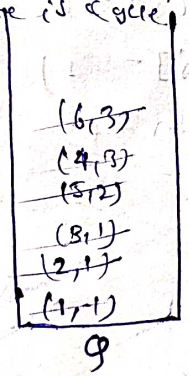


meare at level 2 ⇒ 5,6 and from both side if we touched 7 it means there is cycle

adj list

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





vis

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

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. - $O(N) + O(N)$ (queue, visited)

$\simeq O(N)$

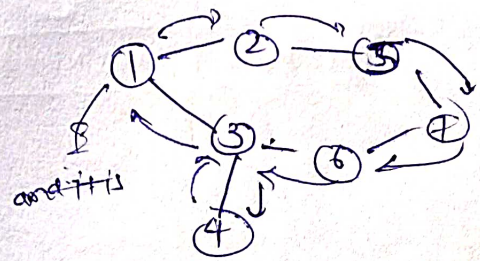
@Aashish Kumar Nayak

```
bool detect(int src, vector<int> adj[], int vis[])
{
    vis[src] = 1;
    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;
}
```

node

parent

Cycle detection in undirected graph → DFS (using)



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

and it should not be parent of that node.

Adj List

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

vis

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

DFS(1, -1)

normal node, parent node

no need for further call because we got True

DFS(2,1)

DFS(3,1)

we will go this call after completing the DFS(2,1) call

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

DFS(5,2)

DFS(7,5)

DFS(6,7)

for 7 there will be no call because it is parent

DFS(3,6)

DFS(4,3)

DFS(1,3) return true,

3 is marked visited and also it is not parent, so we got the cycle.

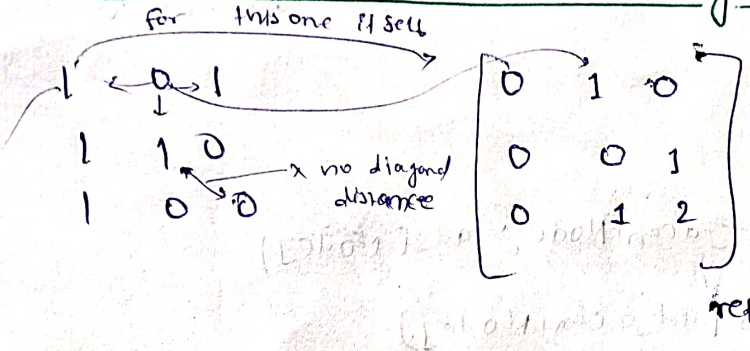
∴ return true.

```
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;
}
```

Sc. - O(N) (Recursion) + O(N) (Vis) ≈ O(N)

T.C. - DFS Traversal
O(N + 2E) + O(N) (for loop)
≈ O(N + 2E)
Not O(N) — But it will depend on components & O(3)

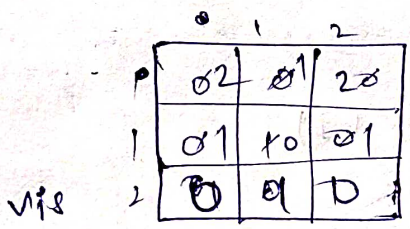
Distance of nearest cell having 1

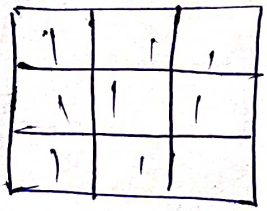


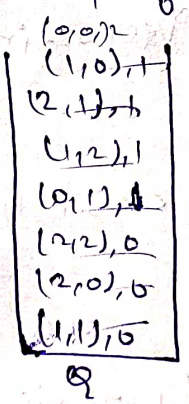
∵ This is something related to graph, and also we have to find min steps or min distance, it means it is a graph problem.

which algo — BFS comes in my mind.

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







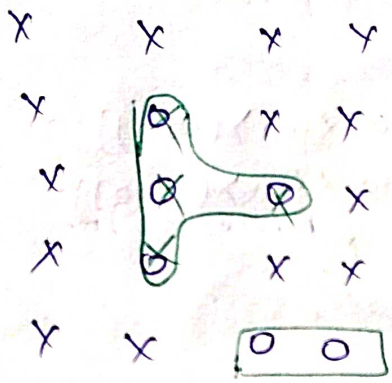
T.C. — $O(n \times m) \times 4 \approx O(n \times m)$

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

@Aashish Kumar Nayak

Replace O's with X's

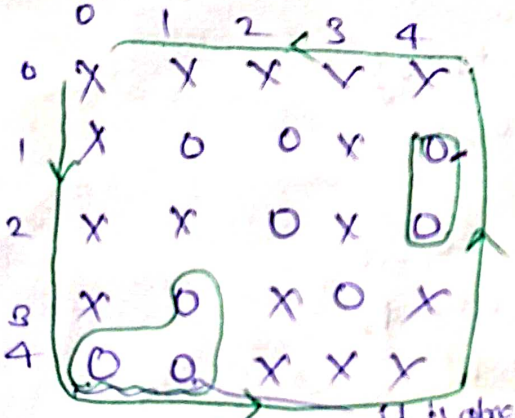


only those ~~ze~~ O's which are surrounded by X in top, down, left, right that can be converted into X. rest ~~are~~ will not converted.

Observation : ~~If someone is on the boundary they.~~ Can a set of X connected to a boundary 'O' cannot be converted.

DFS

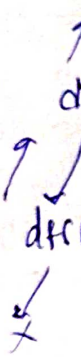
① go to the boundary and bind if there is zero if we got zero cw then call the dfs ~~for~~ from that index



dfs(4,0)

dfs(4,1)

dfs(3,1)

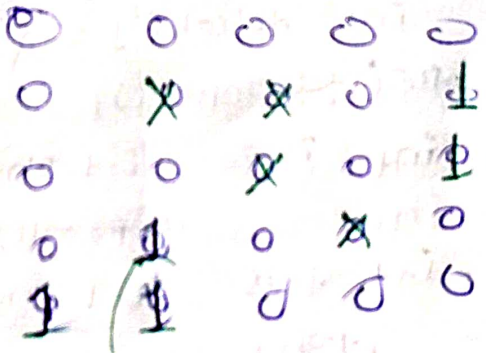


It is already marked no need to traverse again

dfs(1,4)

dfs(2,4)

vis



Some visited from Boundary so can't convert.

@Aashish Kumar Nayak

Algo:-

→ from boundary, find '0's & traverse & mall 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;
  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] && mat[nrow][ncol] == '0')
    { dfs(nrow, ncol, vis, mat, delrow, delcol);
    }
  } }

vector<vector<char>> fill(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 j=0; j<m; j++)
    { if(! vis[0][j] && mat[0][j] == '0')
      { dfs(0, j, vis, mat, delrow, delcol);
      }
      if(! vis[n-1][j] && mat[n-1][j] == '0')
      { dfs(n-1, j, vis, mat, delrow, delcol);
      }
    }
for(int //traver

//traverse for 1st col & last col
```

@Aashish Kumar Nayak

```
for(int i= 0; i<n; 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;
}
```

dfs — 4 neighbours — loop — Boundary

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

$\approx O(N \times m)$

S.C.

$O(N \times m)$

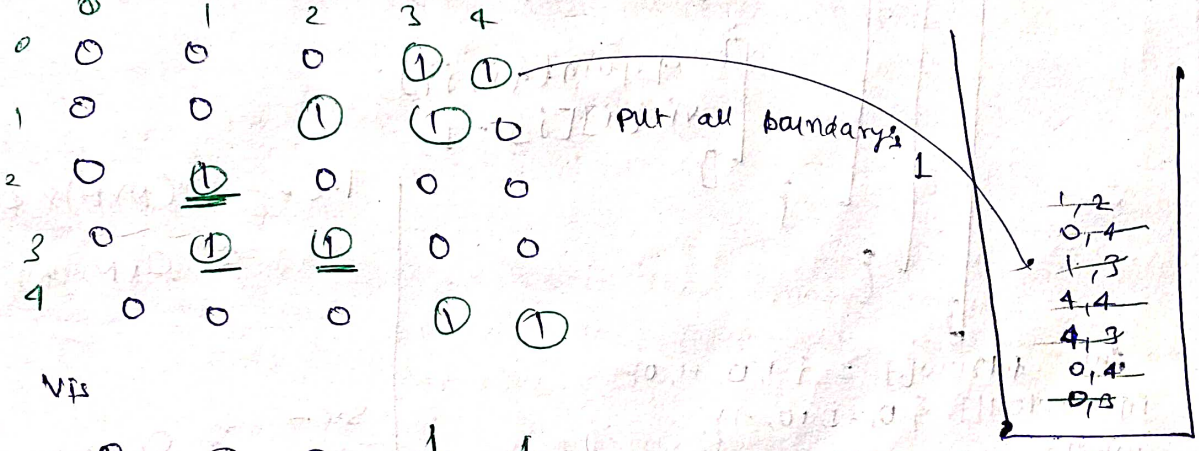
Number of Enclaves || Multi-source BFS

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

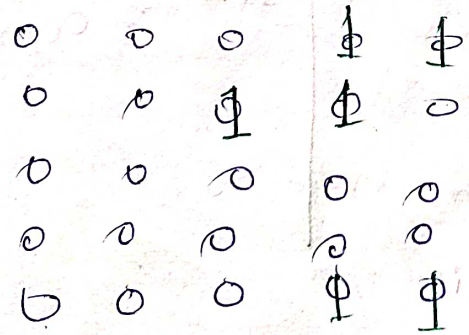
(B means B. you can't count 1's which is on boundary and also the 1's which is connected to the boundary 1's)

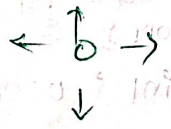
so then the remaining 1's will be our answer.

lets try BFS for no reason just to get clarity



Vis





Once the queue is empty

now count the unvisited 1's and return the ans.

ans = 3

@Aashish kumar Nayak

Code :-

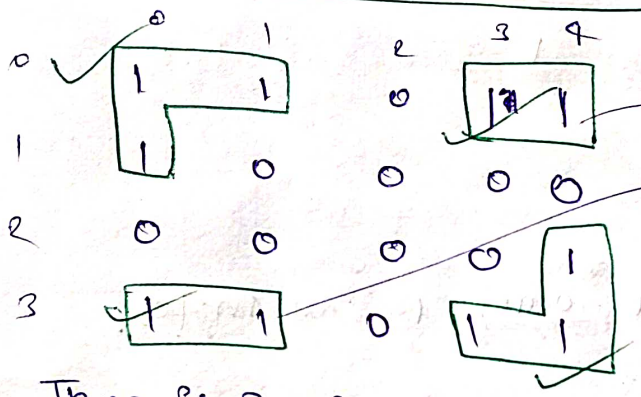
```
int numberOfEnclaves(vector<vector<int>> &grid)
{ queue<pair<int,int>> q;
  int n = grid.size();
  int m = grid.size();
  int vis[n][m] = {0};
  for(int i=0; i<n; i++)
  { for(int j=0; j<m; j++)
    { if(i==0 || j==0 || i==n-1 || j==m-1)
      { if(grid[i][j] == 1)
        { q.push({i,j});
          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

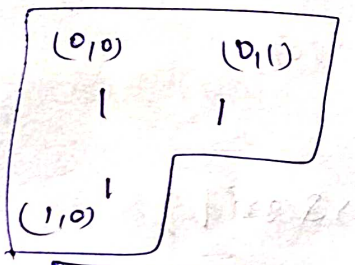
@Ashish Kumar Mayak

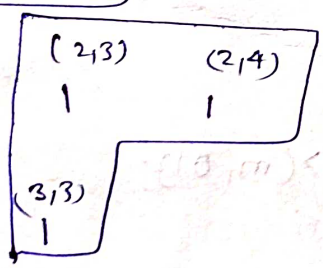


These are identicle so it will be count as 1.

There is 3 distinct island.

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





How to store these shapes ~~so that~~ we such that these two come out to be identical

{(0,0) (0,1) (1,0)}
{(2,3) (2,4) (3,3)} } These are not identicle

so make starting point as base origin

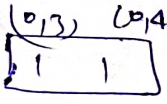
(0,0) - (0,0) : (0,0)
(0,1) - (0,0) : (0,1)
(1,0) - (0,0) : (1,0)

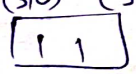
= {(0,0) (0,1) (1,0)}

(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 these are identicle





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

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

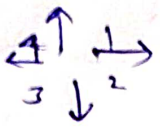
But what if

1 1 1 1

1 1 1

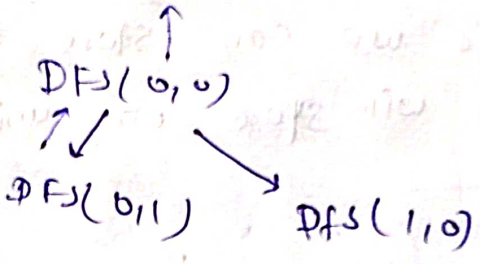
1

So must Follow the same order for all shapes.



1 1 0

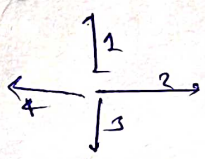
1 0 0



Code :-

```
int countDistinctIslands( vector<vector<int>>& grid )
{ int n=grid.size();
  int m = grid[0].size();
 vector<vector<int>> vis(n, vector<int>(m,0));
  set<vector<pair<int,int>>> st;
  for(int i=0; i<n; i++)
  {
    for(int j=0; j<m; j++)
    {
      if(!vis[i][j] && grid == 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
row0, int col0){
    vis[row][col] = 1; vec.push_back({row-row0, col-col0});
    int n = grid.size();
    int m = grid[0].size();
    vec.push_back({row-row0, col-col0});
    int delrow[] = {-1, 0, 1, 0};
    int delcol[] = {0, -1, 0, 1};
    for(int i=0; i<4; i++)
    {
        int nrow = row+delrow;
        int ncol = col + delcol;
        if(nrow >=0 && nrow <n && ncol >=0 && ncol <m
        && !vis[nrow][ncol] && grid[nrow][ncol ==1]
    {
        dfs(nrow, ncol, vis, grid, vec, row0, col0);
    }
    }
}
```



for loop dfs x neighbour

T.C. ~~O(NXM)~~ + O(NXM)x4

$O(N \times M \times \log(N \times M)) \approx \sim\sim O(N \times M) \sim\sim = O(N \times M \log(N \times M) + (N \times M) \times 4$

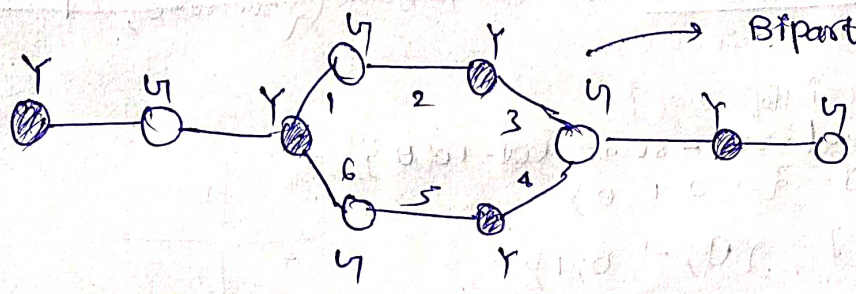
S.C. $\approx O(N \times M)$

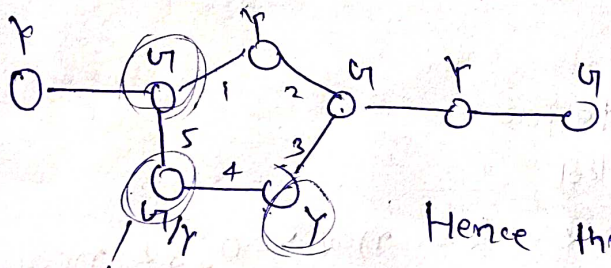
@Aashish kumar Nayak

Bipartite graph

@Aashish Kumar Nayak

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





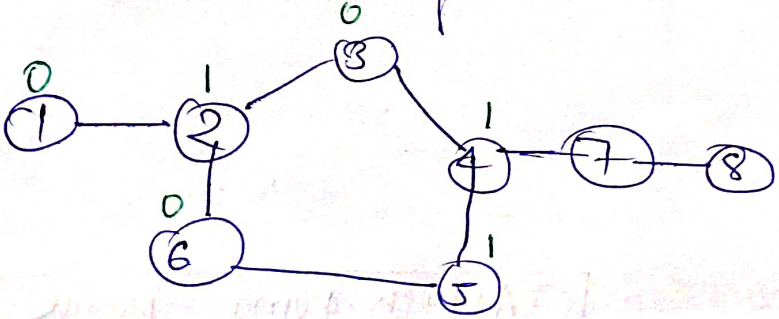
Hence this is not a Bipartite graph.

* Linear graphs with no cycles are always Bipartite graph.

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

We will solve this problem using BFS

∆ Instead of visited we will take colour array.



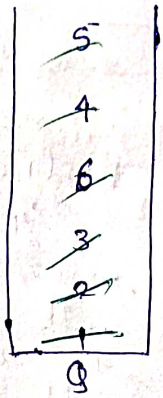
adj. list

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

colour

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

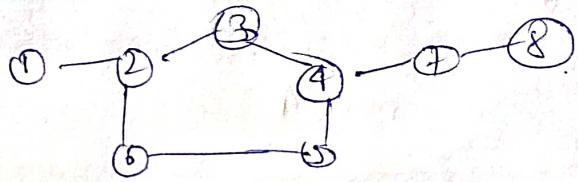
node = 4

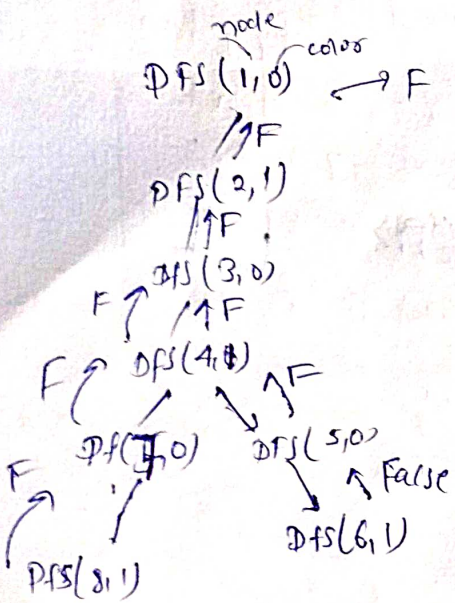


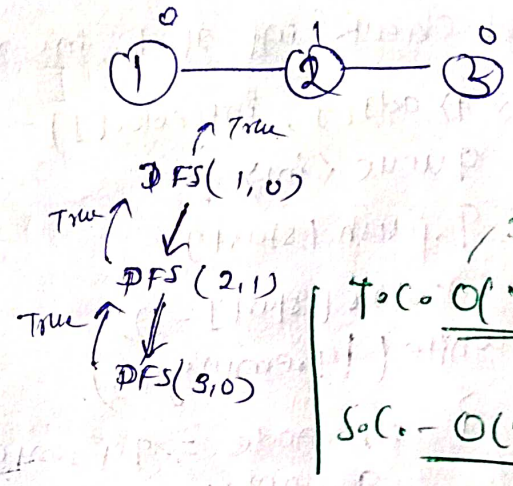
code :-

```
bool cheak(int start, int V, 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] = !color[node];
        q.push(it);
      }
      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(cheak(i, V, adj, color) == false)
      { return false;
      }
    }
  }
  return true;
}
```

Bipertite Graph using DFS :







DFS Algo

T.C. - O(V+2E)

S.C. - O(V)
color

code :

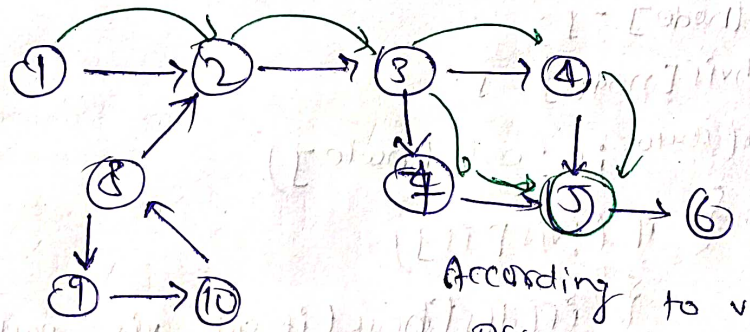
```
bool dfs(int node, int col, int color[], vector<int> adj[])
{ color[node] = col;
  for(auto it : adj[node])
  { if(colour[it] == -1)
    { if(dfs(it, !col, color, adj) == false)
        return false;
    else if( color[it] == col)
      { return false;
      }
  return true;
}
bool isBipertite(int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

@Aashish Kumar Nayak

Detect cycle in a directed graph

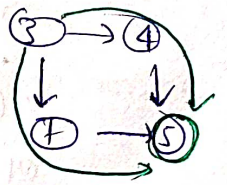
visited algorithm using the DFS will not work here this with algo will work only in undirected graph

for ex:-



dfs(1)
dfs(2)
dfs(3)
dfs(4)
dfs(5)
dfs(7)

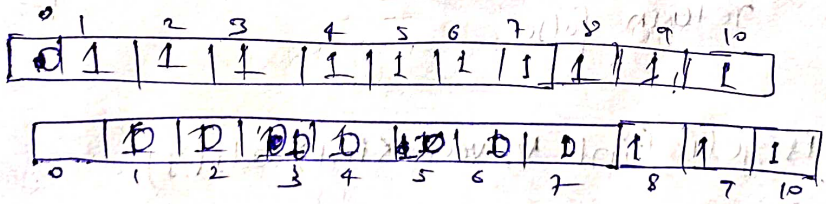
According to visited algo using DFS



This is a cycle but this is not true.

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

vis —
path vis —

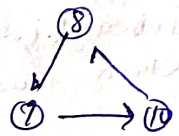


when we are returning then make path unvisited visited array.

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

dfs(3)
dfs(7)

dfs(8)
dfs(9)
dfs(10)
dfs(8)



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

we got 8 which is visited & pathvisited
return true;

code :-

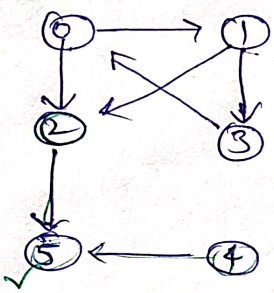
```
bool dfscheak(int node, vector<int>adj[], int vis[], int
pathvis[])
{   vis[node] = 1;
    pathvis[node] = 1;
    for( auto it : adj[node])
      { if (!vis[it])
          { if(dfsCheak(it;adj, vis , pathVis) ==true)
                return true;
          }
        else if (pathvis[it])
          {   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(dfscheak(i, adj, vis, pathvis) == true)
              return true;
        }
    }
    return false;
}
```

T.C. - O(V+E)

Sec. - O(2N)
≈ O(N)

@Aashish Kumar Nayak

Find Eventual Safe States :-



We have to find which are the safe node →

terminal nodes

↓

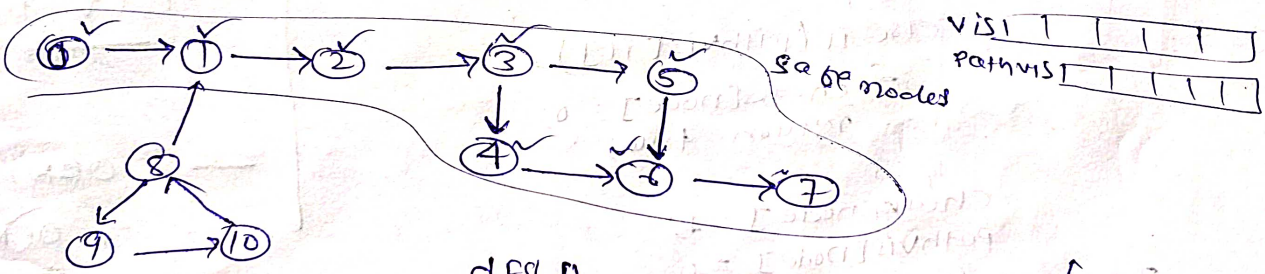
Outdegree ⇒ 0

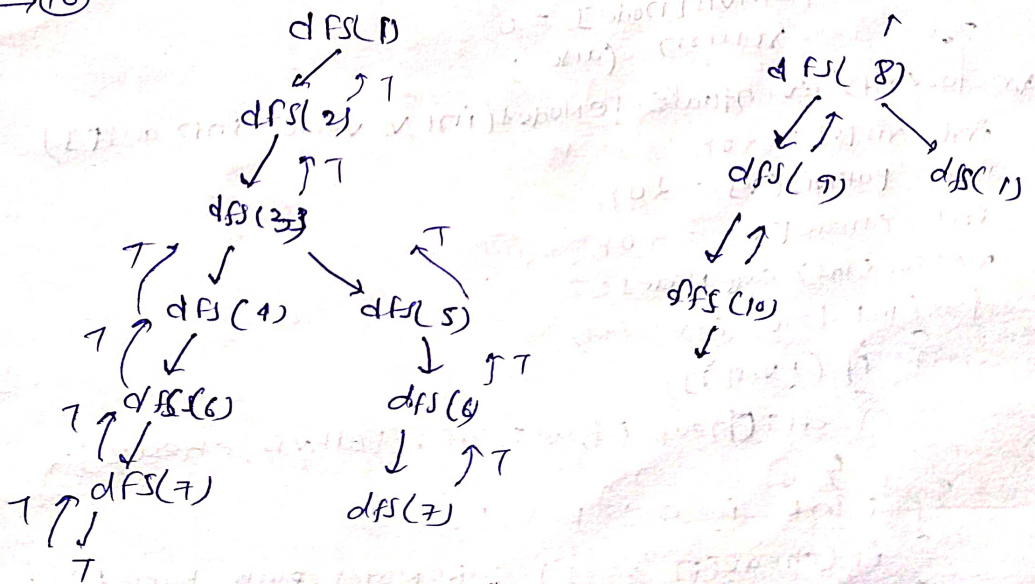
Nodes which are ending at terminal node

Safe Node arr → [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.





Code:

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

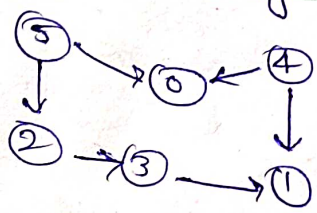
T.C. – O(V+E)

S.C. – O(2N) ≈ O(N)

@Aashish kumar Nayak

Topological Sort (DFS) :

(DAG) / ↳ Linear ordering of vertices such that if there is an edge between u & v, u appears before v in that ordering.



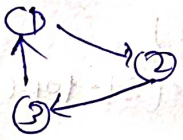
5 4 2 3 1 0

4 5 2 3 1 0

only possible in (DAG) Directed Acyclic graph

(i) should be directed

(ii) should be no cycle.

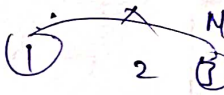


1 - 2

2 - 3

3 - 1

① — ② 1 2 / 2 1 ✗ not possible

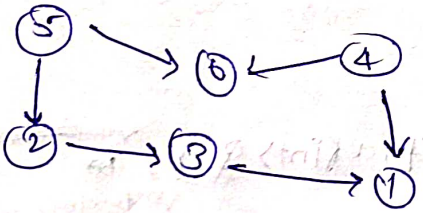


Not possible

1 should be before 3

3 should be before 1

both at same time. ✗ not possible



Adjacency list

0 - { }

1 - { }

2 - {3}

3 - {1}

4 - {0, 1}

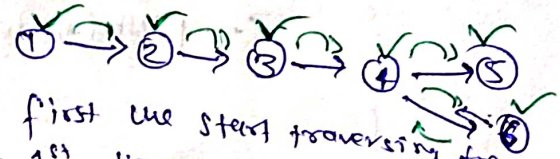
5 - {0, 2}

vis | 1 | 1 | 1 | 1 | 1 | 1 |

st.top():

{ 5 4 2 3 1 0 } — Ans

Intuition :



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	Who's dfs is completed
2	just store it on stack.
3	
4	
5	
6	

@Aashish kumar Nayak

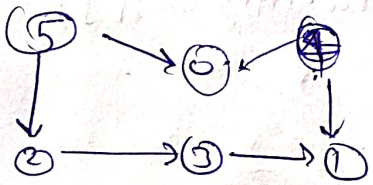
Code :-

```
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) (vis) + O(N) (stack)
≈ O(N)

T.C. = O(V + E)
↓
directed graph

Kahn's Algorithm (Topological sort) :-



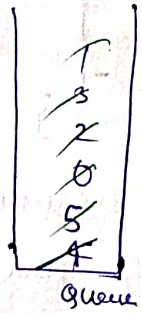
We will use Queue & Indegree array

Indegree

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

There is no element which comes to 4 & 5 or Indegree = 0

so 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 → 3 it means indegree of 3 is 1

so we will increase ++ indegree.

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

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. $O(N) + O(N)$ (indegree array) $\approx O(N)$

Detect a cycle in directed graph using topological sort → BFS

in previous ~~con~~ we used vis, pathvisited and when we were returning then we ~~as~~ reseting the pathvisitid array.

But in BFS we can't do that like backtracking

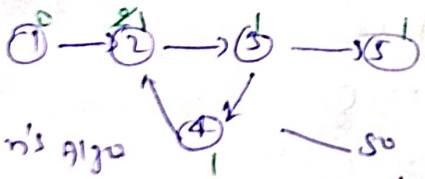
So we use kahn's Algo

This is a direct ~~Acyclic~~ graph

1 — 2
2 — 3
3 — 4, 5
4 — 2
5 — ↛

Toposort is only applicable for DAG

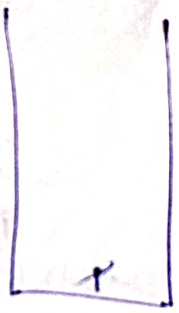
Lets try to find out the topological sort for this Direct cyclic gra.



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

& if ~~si~~Toposort has less than N elements then it must have cycle.

Code:- Code is same as Kahn's Algo

we will just cheak size of toposort array in last if size of toposort == V then there is no cycle and if size is less than V 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;
}
```

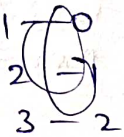
T.C. - O(V+E)

S.C. - O(N)

@Aashish Kumar Nayak

Course Schedule - I & II

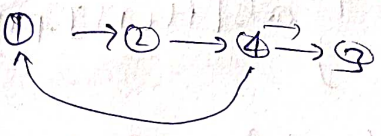
pre requisite task



1,0
2,1
3,2

3 2 1 0 Yes

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



No

This is happing because cycle is present.

We can use topological sort.

u → v (u,v)

u must have before v

Topo sort

Direct & Acyclic

Code :

```
bool isPossible(int V, vector<pair<int,int>> &prerequisites)
{
    vector<int> adj[V];
    for(auto it: prerequisites)
        { adj[it.first].push_back(it.second);
    }
    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);
    }   }
    if(topo.size()==V) return true;
    return false;
}
```

course schedule =

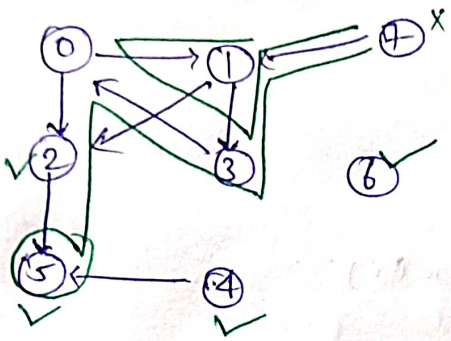
Code

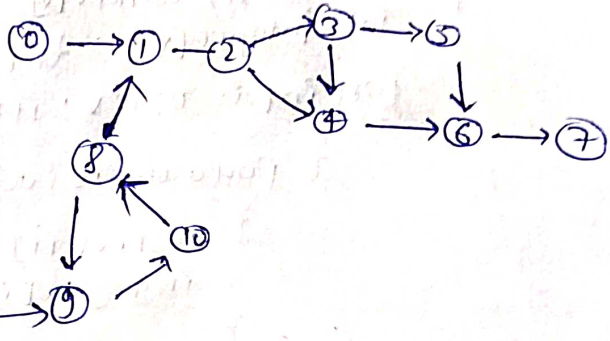
```
vector<int> findOrder(int V, int m, vector<vector<int>>& prerequisites)
{ for(  vector<int> adj[V];
    for( auto it : prerequisites)
      { adj[it[1]].push_back(it[0]);
      }

  for(  int indegree[V] = {0};
    for(int i=0; i<V; i++)
      {
        for(auto it : adj[V])
        {
          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--;
        if(indegree[it] == 0) q.push(it);
    }
    if(topo.size() == V) return topo;
    return {};
}
```

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

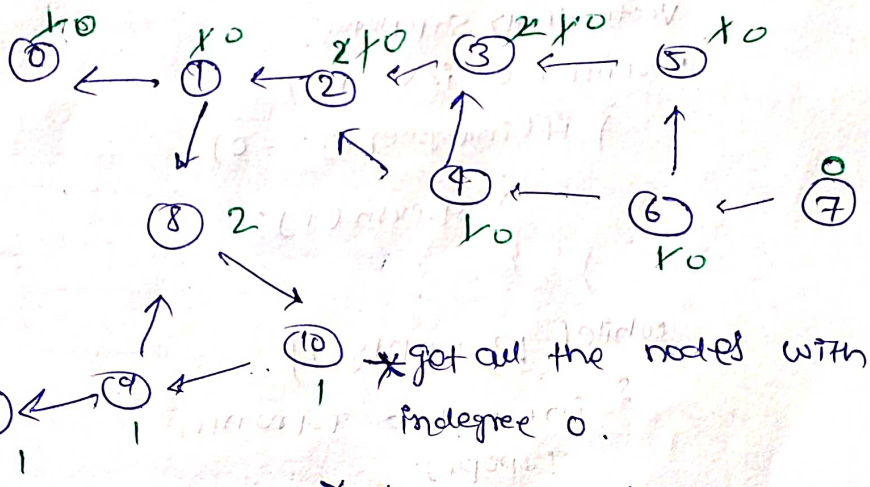
@Aashish Kumar Nayak

Eventual Safe States using BFS Topo sort





→ Reverse all the edges



* get all the nodes with indegree 0.

* do a ~~traversal~~ removal of edges of adjacent nodes.

Safe Node :- 7 6 5 4 3 2 1 0

then we will sort the safe node array.

then return

Code :-

```
vector<int> eventualSafeNodes(int V, vector<int> adj[])
{
    vector<int> adjRev[V];
    int indegree[V] = {0};
    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)

S.C. - $\sim O(V)$

@Ashish Kumar Nayak

Alien Dictionary | Topological sort

Alien

[baa
[abcd
abca
cab
cad

Normal a b c d e f g h - - -

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

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

baa
abcd
abca
cab
cad

figure out where char is not equal and then parse DLL according to that

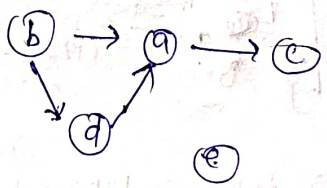
K = 5

a b c a c
0 1 2 3 4 5

j =

$s_1 = arr[i]$;
$s_2 = arr[i+1]$;
compare s_1 & s_2

[b d a c] — Toposort return arr



If we got unequal char then make directed graph then after completing the graph. We will do toposort then return the toposort.

* what if order is not possible. like abcd $-s_1$
abc $-s_2$

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:- a b c
b a t
a d e

wrong dictionary a → b
a < b
✗ b < a
not possible

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

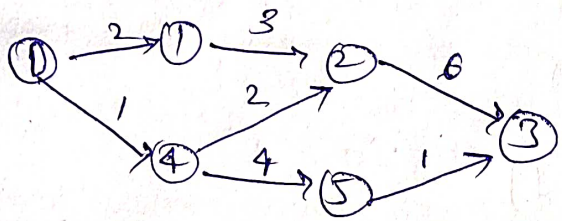
```
{
  vector<int> adj[K];
  for( int i=0; i<N-1; i++)
  {
    string s1 = dict[i];
    string s2 = dict[i+1];
    for int len = min(s1.size(), s2.size());
    for( int ptr=0; ptr<len; ptr++)
    {
      if( s1[ptr] != s2[ptr] )
      {
        adj[s1[ptr]].push_back(s2[ptr]);
        break;
      }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

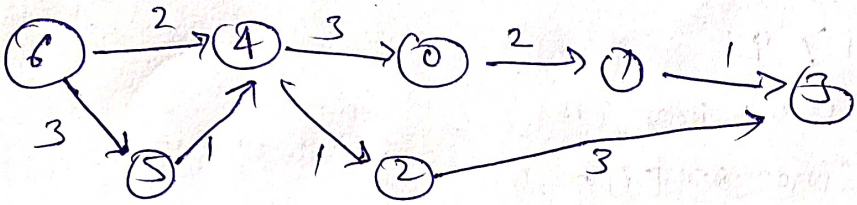
@Aashish Kumar Nayak

Sor Shortest path in Directed Acyclic Graph:-

Topological sort:-



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



adj list :-

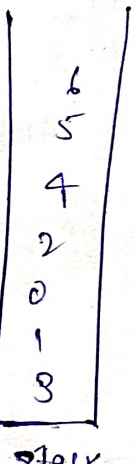
node weight

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

Step 1 :- Do a toposort on the graph.

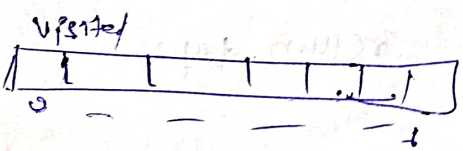
DFS ways
BFS ways

DFS Method



this stack will store the toposort

Stack



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
+3
node = 5
dist = 3

node = 4

dist[] = | 5 | 7 | 3 | 6 | 2 | 3 | 0 |
0 1 2 3 4 5 6

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

5 → {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};
  stack<int> st;
  for(int i = 0; i<N; i++)
  { dist[i] = 1e9; }
  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;
}

for(int i = 0 ; i<N ; i++ )
{ if( !vis[i] )
  { toposort(i,adj,vis,st);
  }
}
```

@Ashish Kumar Nayak

```
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.C. ≈ O(N+M)

S.C. ≈ O(N)

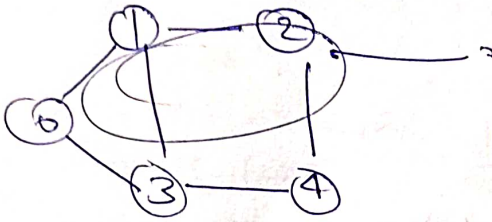
intuition:

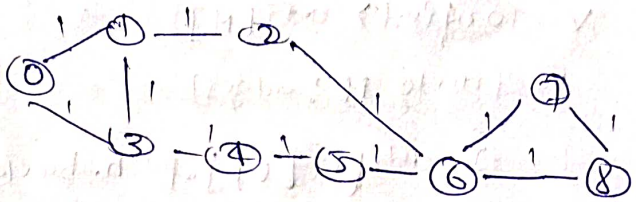
Finding the shortest path to a vertex is easy if you already know the shortest path of all the vertices that can precede it. Finding the longest path to a vertex in DAG is easy if you already know the longest path to all the vertices that can precede it.

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 topological sorted.

Shortest path in undirected graph having unit distance :





dist → | 0 | 1 | 2 | 1 | 2 | 3 | 3 | 4 | 4 |
0 1 2 3 4 5 6 7 8

1 < 2 so no updation

0 — 1, 3
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

node = 0 dist = 0

node = 1 dist = 1 | node = 3 dist = 1

node = 1, dist = 1

+1 → node 0 dist = 2 | +1 → node 2 dist = 2 | +1 → node 3 dist = 2 discarded

queue {node, dist}

we are looking for shortest path

node = 3, dist = 1

+1 → node = 0 dist = 2 | +1 → node = 4 dist = 2

node = 2, dist = 2

node = 1 dist = 3 | node = 6 dist = 3

node = 4 dist = 2

node	3	5
dist	3	3

node = 6, dist = 3

node	2	5	7	8
dist	4	4	4	4

node = 5, dist = 3

node	4	6
dist	4	4

node = 7, dist = 4

node	6	8
dist	5	5

node = 8, dist = 4

node	6	7
dist	5	5

@Aashish Kumar Nayak

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] = 1e9;
}
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] != 1e9)
    {
        ans[i] = dist[i];
    }
}
return ans;
}
```

T.C. - BFS algo
$\approx O(V+2E)$

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

@Aashish Kumar Nayak

WORD Ladder - I

Hard problem

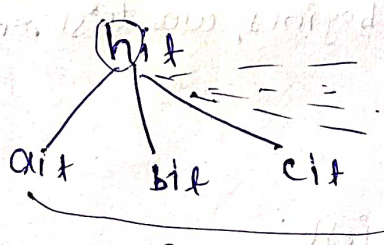
beginWord = "hit" endWord = "cog"

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

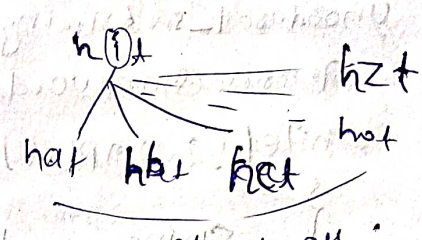
hit → hot → dot → dog → log

Minimum no. of transformations

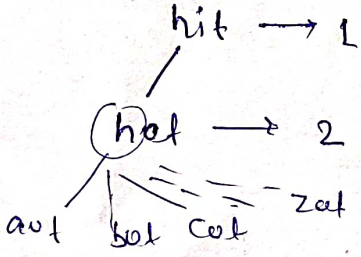
Brute Force



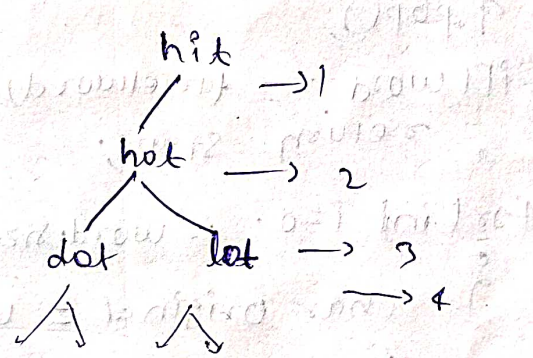
check all in word list



check all.
we got hot



lot & dot exist



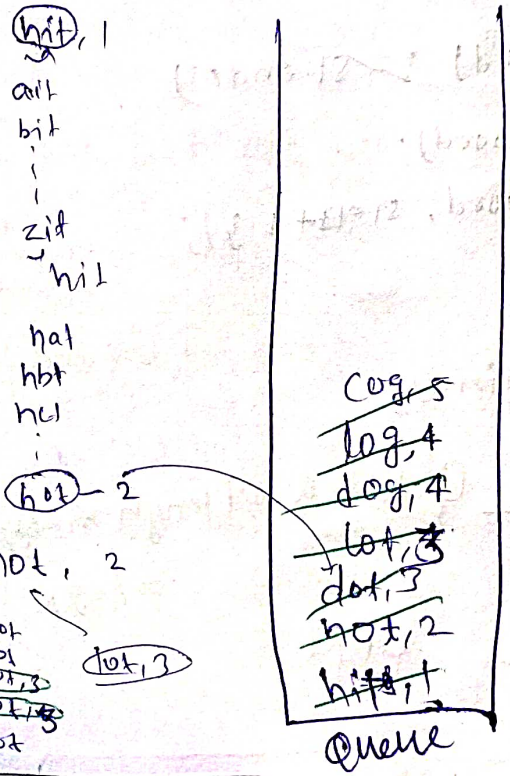
log → 5 → shortest sequence

BFS

we will remove words from wordlist when we will get word.

and if we got same word in next iteration we will not take 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});
unoodered_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;
}
```

set

T.C. O(N * word.length * 26 * logN)

S.C O(N)

@Aashish Kumar Nayak

Word Ladder II

Hard

beginning word = bat end word = coz

wordlist = [~~pat~~, ~~bot~~, pot, poz, coz]

bat → pat → pot → poz → coz

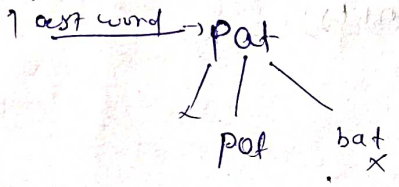
bat → bot → pot → poz → coz

[~~pat~~, ~~bot~~, ~~pot~~, ~~poz~~, coz]

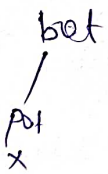
Erase only when entire level completed.

{bat} word = bat

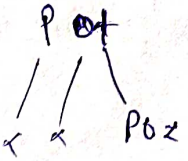
{bat, pat}



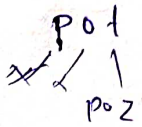
{bat, bot} —



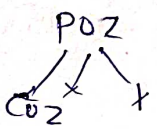
{bat, pat, pot}



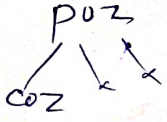
{bat, ~~bot~~, pot}



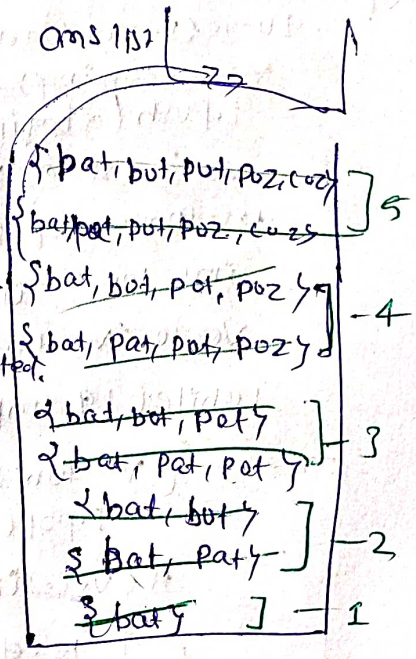
{bat, pat, pot, poz} —



{bat, bot, pot, poz}



Ans list



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<strings>> 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);
      }
    }
```

(1)

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

①

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

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

@Aashish Kumar Nayak

~~Word ladder~~ : II (~~optimised~~ approach)

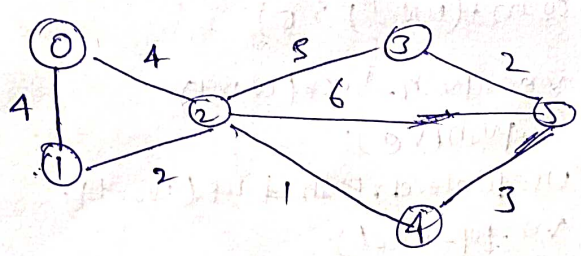
only for C++

DIJKSTRA'S Algorithm using priority queue

we always wanted minimum distance at 1st.

Reason for using Priority-Queue

Dijkstra's Algorithm → shortest path



0 → 0 d=0

→ 1 d=4

2 d=4

3

Adj list

node, weight

0 → {1,4}, {2,4}

1 → {0,4} {2,2}

2 → {0,4}, {1,2}, {3,3}, {4,1} {5,6}

3 → {2,3}, {5,2}

4 → {2,1}, {5,3}

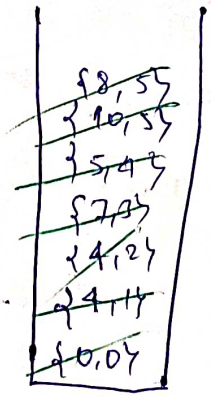
5 → {2,6}, {3,2}, {4,3}

There is two methods in Dijkstra's Algo

① priority queue take less time

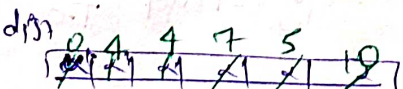
② Set — fastest

③ Queue — ✗ take more time



min heap

{dist, node}



d=0 node = 0

node 1 2

distance = 0+4 0+4

d=4 node = 1

node 0 2 5

dist 4+4 =8 4+2 =6

dist=4 node = 2

node 5 0 1 3 4

dist 4+6 =10 4+4 =8 4+2 =6 3+4 =7 1+4 =5

node = 4 dist = 5

node 2 5

dist = 5+1 =6 5+3 =8

dist=7 node=3

node 2 5

dist 7+3 =10 7+2 =9

dist=8 node=5

node 2 3 4

dist 14 10 17

dist=10 node=5

2 5 4

16 12 13

@Aashish Kumar Nayak return dist; Ans

Neo code :-

@Ashish kumar Nayak

~~vector<int> dijkstra(int V, vector<vector<pair<int,int>>>~~

```
vector<int> dijkstra( int v , vector<vector<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]=1e9;
  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;
}
```

T.C. $E \log V$ (E: edges, V: nodes)

≡ Dijkstra does not works in → negative weight
→ negetive cycle

0 —(-2)— 1

dist | 0 | -2 |

-2 0 1

-4

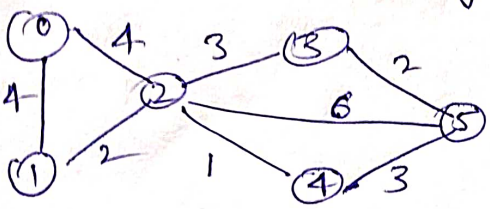
Infinite loop

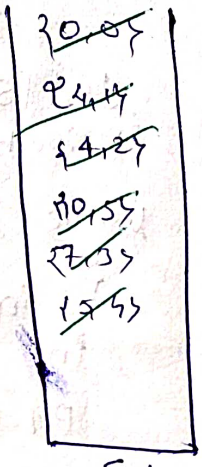
Dijkstra's is applicable for the +ve weights.

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)





node = 0 —+4→ node = 1, d = 4

node = 0 —+4→ node = 2, d = 4

node = 4, dist = 4 —+4→ node = 0, d = 8

node = 4, dist = 4 —+2→ node = 2, d = 6

node = 4, d = 5 —+1→ node = 2, d = 6

node = 4, d = 5 —+3→ node = 5, d = 8

{dist, node}

dist[] =

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

st.erase(); — Log(n)

But we are saving iterations

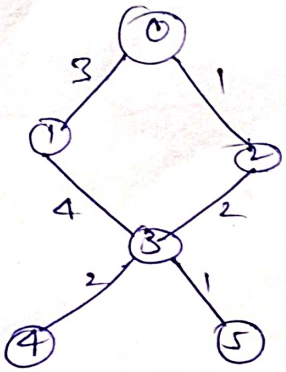
Code :-

```
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, intution, Time Complexity Derivation

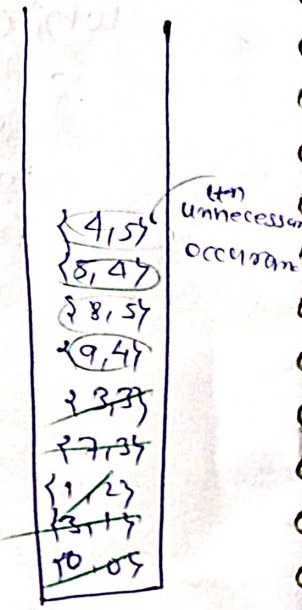
TIME complexity $O(E \log V)$



dist[]

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

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



node = 0
d = 0
+3 / +1
node = 1, d = 3
node = 2, d = 1

node = 1
d = 3
+0 / +4
node = 0, d = 3 ×
node = 3, d = 7

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

node = 3
d = 7
4 / 2 / 2 / 1
node 1, d = 11 ×
2, 9 ×
4, 9
5, 8

node = 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 mor time complexity.

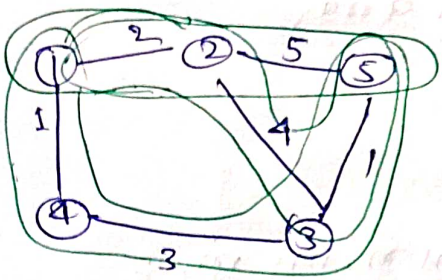
@Aashish Kumar Nayak

if we use priority queue we explore minimal-minimal - minimal - - - - - -

T.C. $E \log V$

E → no. of edges, V → no. of nodes

print shortest path - Dijkstra's Algorithm :-



src = 1
dst = 5

8 paths -

	sum
1-2-5	7
1-2-3-5	7
1-4-3-5	6 ✓ shortest path

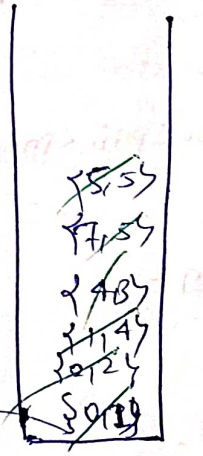
we need to print path

1-4-3-5

HINT

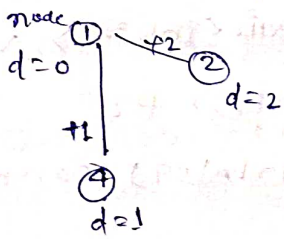
remember where I am coming from
then backtrack the path

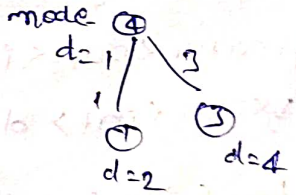
priority-queue

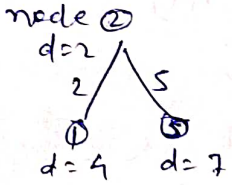


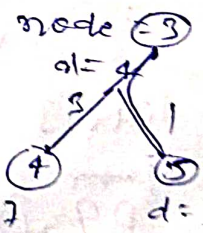
parent [1 | 1 | 4 | 3 | 5]

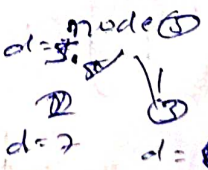
dis[] [0 | 2 | 4 | 1 | 7]
1 2 3 4 5

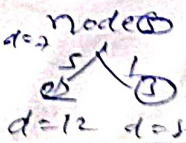












@Aashish Kumar Nayak

dist = 5

1 → 4 → 3 → 5

5 come from 3

go to 3 and check where did 3 coming from

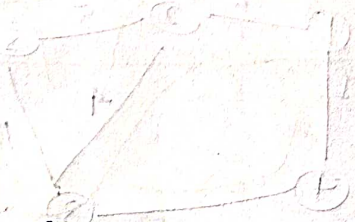
dist[1] = 0 then stop

or if node equals to src then stop

→ then we will store it 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[n+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(n+1, 1e9), parent(n+1);
  for(int i=1; i<=n; i++)
  {
    parent[i] = i;
  }
  dist[1] = 0;
  pq.push({0,1});
```

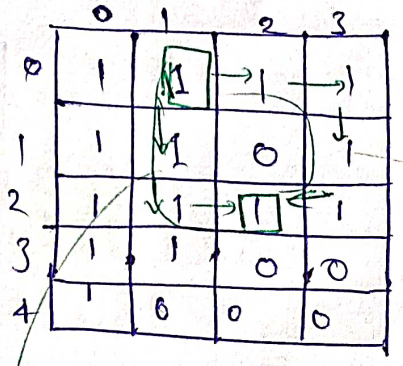
```
while(!pq.empty())
{
    auto it : pq.top();
    int node = it.second;
    int dis = it.second
              it.first;
    pq.pop();
    for(auto it : adj[node])
    {
        int adjNode = it.first;
        int edw = it.second;
        if(dis + edw < dist[adjNode])
        {
            dist[adjNode] = dis + edw;
            pq.push({dist + edw, adjNode});
            parent[adjNode] = node;
        }
    }
}
if(dist[n] == 1e9)
    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;
}
}
```

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

$\approx O(E \log V)$

@Aashish Kumar Nayak

Shortest Distance in a Binary Maze



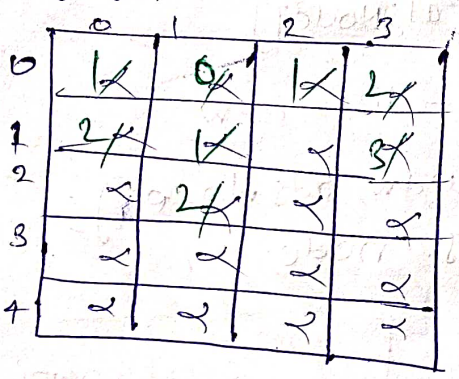
src = {0,1}
dest = {2,2}

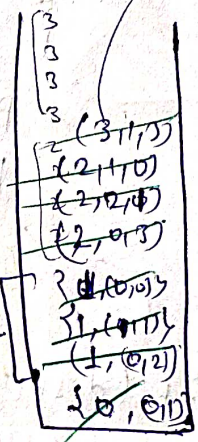
5 steps

3 steps

Dijkstra's Algo

dist[]



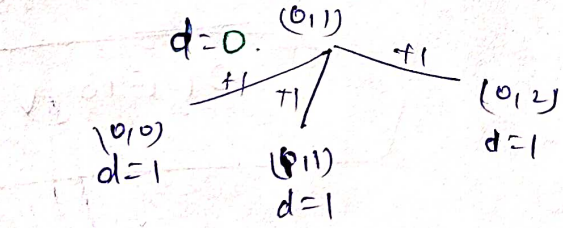


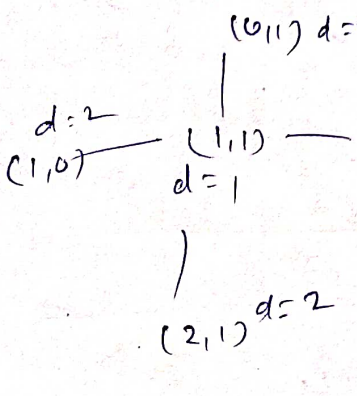
return dist = 3
Ans

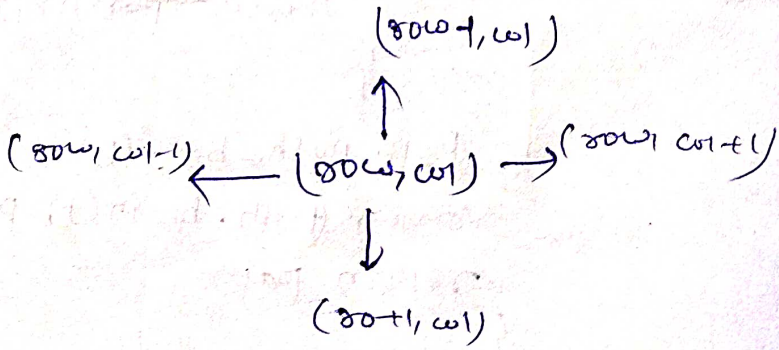
we are using pq to take out minimal of the distance but here we will find no diff bcoz in Q or PQ so we will use Q here so save T.C log(H)

pq

dalist, {row, col}







@Aashish kumar Nayak

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(n, vector<int>(m,1e9));
dist[source.first][source.second] = 0;
q.push({0,{source.first, source.second}});
int dr[] = {-1,0,1,0};
int dc[] = {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+dr[i];
int newc = c+dc[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({1+dis,{newr,newc}});
}
}
return -1;
}
```

@Aashish Kumar Nayak

Path with Minimum Effort

1	2	2
3	8	2
5	3	5

1 → 2 → 2 → 2 → 5
1 → 0 → 0 → 3 = max → 3 effort

1 → 3 → 8 → 2 → 5
2 → 5 → 6 → 3 = max → 6 effort

1 → 3 → 5 → 3 → 5
2 → 2 → 2 → 2 = max → 2 effort

figure out ~~minimum~~ max 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**

~~dist[][]~~

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

dist[][]

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

pq

(2,2,2)
(7,2,1)
(7,2,0)
(5,1,1)
(3,2,2)
(1,1,2)
(6,1,1)
~~(1,0,2)~~
(2,1,0)
~~(1,0,1)~~
(0,0,0) ← (dist, r, c)

(0,0) diff = 0 { max of all the diff. on the path }
(0,1) diff = 1
(1,0) diff = 2

diff = 1
(0,1) — (0,2) diff = 0
(0,1) — (1,1) diff = 6
new max(6,1) = 6

diff = 1
(0,2) — (1,2) d = 0

@Aashish Kumar Nayak

```
int MinimumEffort(vector<vector<int>>& heights)
{
priority_queue<pair<int,pair<int,int>>, vector<pair<int,pair<int,int>>>,
greater<pair<int,pair<int,int>>> > pq; // {diff, {row, col}}

int n = height.size();
int m = height[0].size();

vector<vector<int>> dist(n, vector<int>(m,1e9));
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)
{ int 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.
}
```

T.C. $O(E \log V)$ — edges, nodes

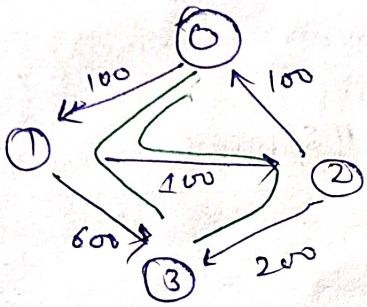
$\geq$ ~ $O(N \times M)$

@Aashish kumar Nayak

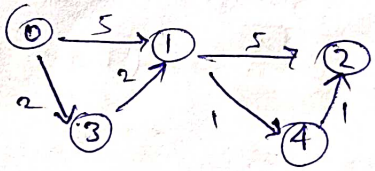
Cheapest Flights within K stops :-

src = 0, dest = 3, K = 1

at max we can take K stops



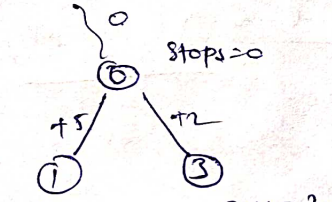
let's see why simple Dijkstra's will fail

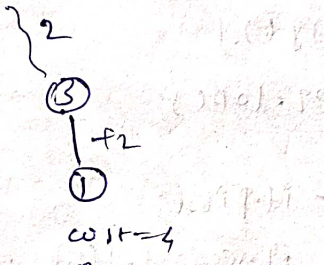


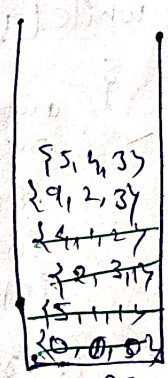
cost
dist[]

0	5	9	2	5	∞
0	1	2	3	4	

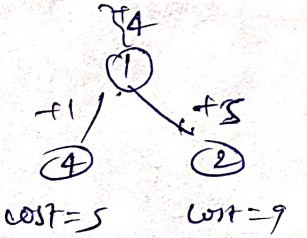
src = 0
dest = 2
K = 2

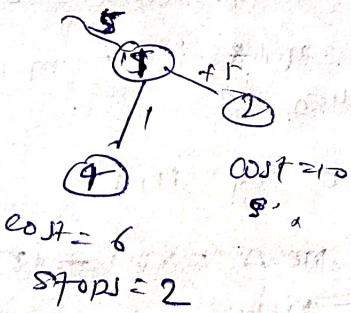






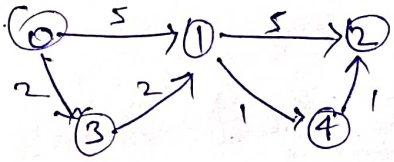
pq
{dist, node, stops}

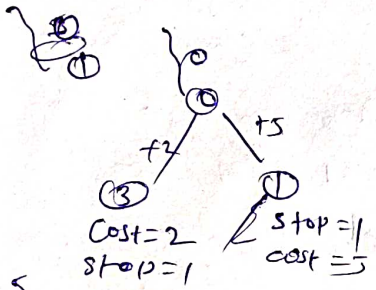


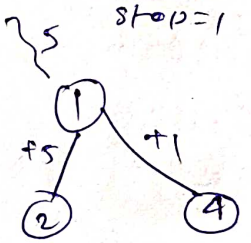


This path has sortest distance but more stops so we can't choose this.

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

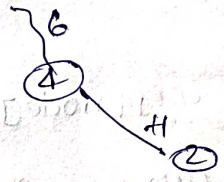






cost = 10
stops = 2

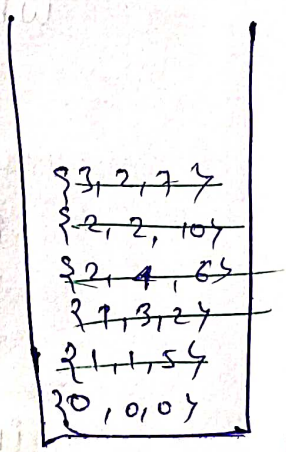
cost = 6
stops = 2



cost = 7
stop = 3

dist[]" =

0	5	7	2	6
0	1	2	3	4



{stops, node, dist}

~~use~~ there no need of priority queue because ~~#~~stops are increasing constantly.

we can extra O(logN)

Code :-

```
int CheapestFlights( 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;
```

@Aashish Kumar Nayak

```
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. $O(E \log V)$ ~~X~~

But we are not using PQ

so $O(E)$

Sc. Flight.size();

Minimum multiplication to reach End :

start = 3 end = 30

arr[] = {2,5,7}

start = 3 $\xrightarrow{\times 2}$ 6 $\xrightarrow{\times 5}$ 30

(2) Ans

start = 7 end = 66175

arr[] = {3,4,65}

7 × 3 = 21

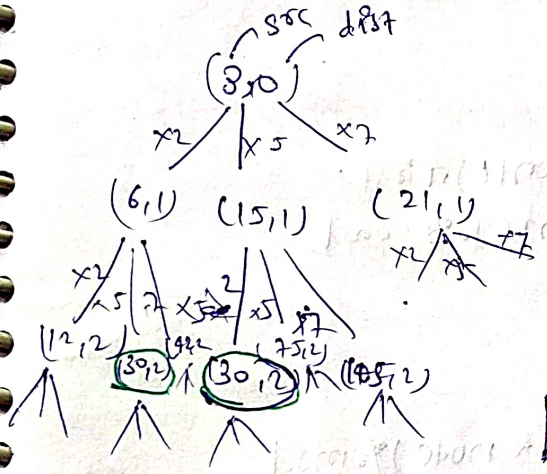
21 × 3 = 63

63 × 65 = 4095

4095 × 65 = 266175

266175 % 100000 = 66175 Ans

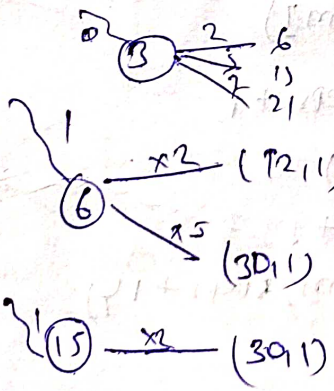
(4) Ans



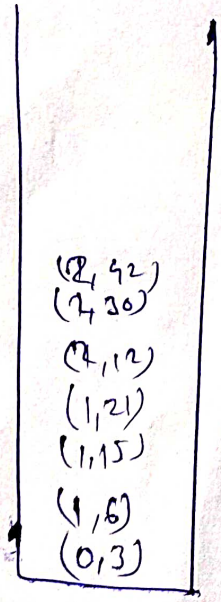
Dijkstras Algo

dist[]

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



∵ no need to use pq we will use queue.



Steps, numy

min-heap

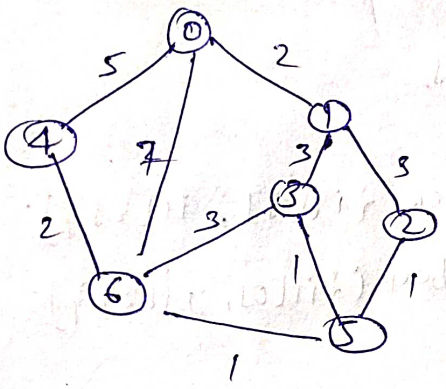
Code :-

```
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



n = 7, src = 0, dest = 6

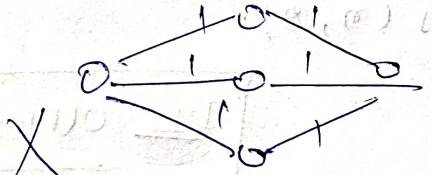
0 → 6

0 → 4 → 6

0 → 1 → 2 → 5 → 6

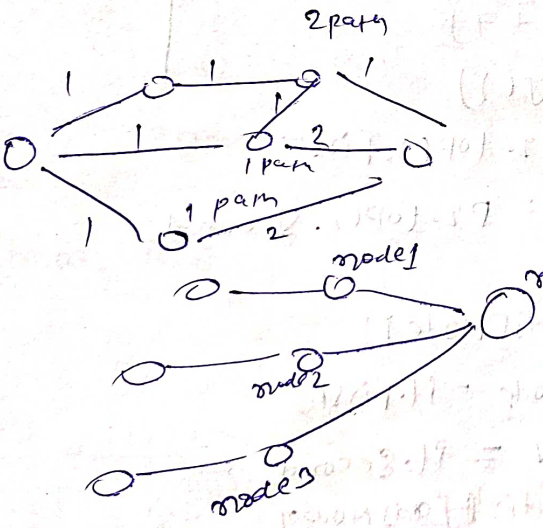
0 → 1 → 3 → 5 → 6

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

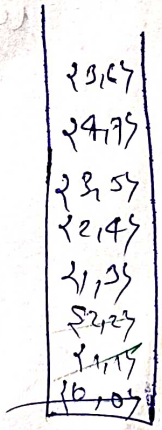


minimum distance 3 times we ended

But this is wrong

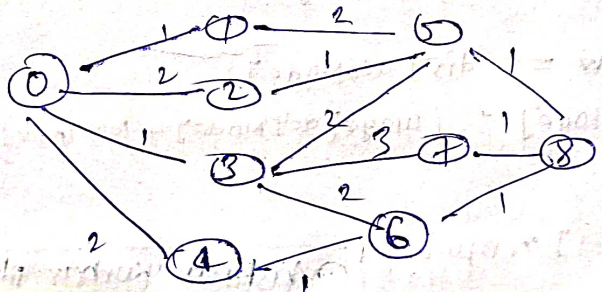


4 path

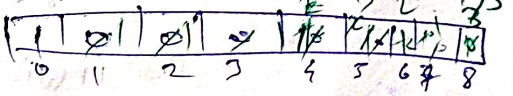


{dist, node}

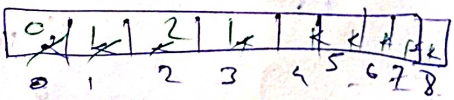
ways[node] = ways[node1] + ways[node2] + ways[node3]



ways



dis



@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 mode = (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;
}
```

T.C. O(E log V)

SC

Belman ford Algorith (Algorithm for finding shortest path)

Need of Belman ford Algorithm :-

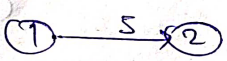
It helps us to detect negative cycles.

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

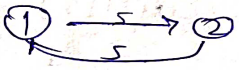
But in Belman ford Algo we can calculate path with -ve weight or -ve cycle.

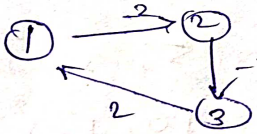
This Algo is Applicable for → DG

Ex:-



if we have 

then change it to 

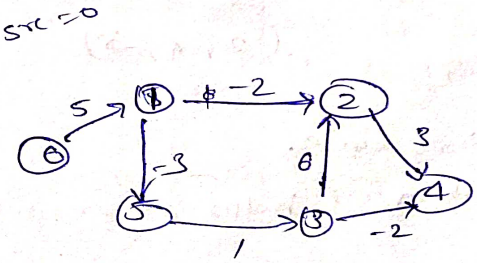


→ Dijkstra will fall in TLE or in infinite loop.

path weight = 2 − 2 − 1 + 1 = −1

if path weight < 0 it has negative cycle.

Belman ford Algo :-



adj list

(u, v, wt)
(3, 2, 6)
(5, 3, 1)
(0, 1, 5)
(1, 5, -3)
(1, 2, -2)
(3, 4, -2)
(2, 4, 3)

this operation for all edges and it will count 1st iteration

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

Ⓧ Ⓧ This is called Relax

```
if( dist[u] + wt < dist[v] ).
    dist[v] = dist[u] + wt
```

@Aashish Kumar Nayak

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

(u, v, wt)

1st iteration

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] + 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)

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

2nd iteration

dist[3] + 6 < dist[2] → (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

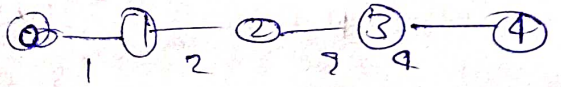
1) Why N-1 cycles

2) How we detect negative cycle.

→ 0 helps to calculate 1, then 1 helps, 5 ... so on

the are N nodes

fifth (N-1) iteration



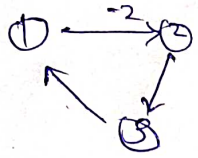
(N-1) edges

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.

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



On Nth iteration the relaxation will be done and the distance array get reduced.

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

Code:

```
vector<int> bellman_ford(int V, vector<vector<int>> &edges, int S)
{   vector<int> dist(V, 1e8); dist[S] = 0;
    for(int 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]; int v = it[1]; int wt = it[2];
       if(dist[u] != 1e8 && dist[u] + wt < dist[v])
       {return {-1};}
    }
    return dist;
}
```

T.C. $\simeq O(V \times E)$ (V: nodes, E: edges)

@Aashish Kumar Nayak

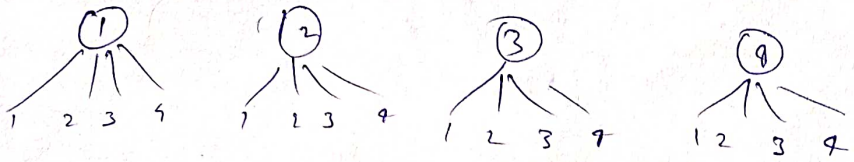
floyd warshall Algorithm (Algorithm for shortest path)

This is very-very different from Dijkstra's & Belaman ford.

in these two previous Algo src same point is 1.

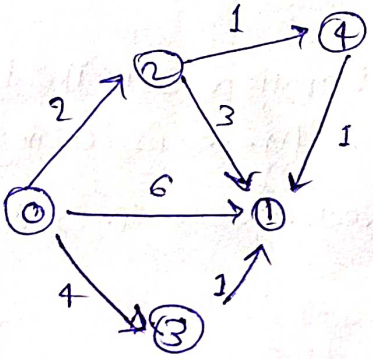
But in floyd warshall algo there is multiple source point.

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.



0 1

$0 \to 1 = 6$

$(0 \to 2) + (2 \to 1) = 5$
2 3

$(0 \to 3) + (3 \to 1) = 5$
4 1

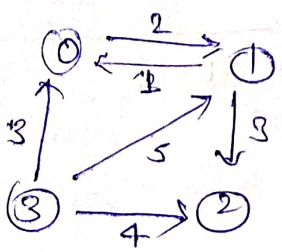
$(0 \to 4) + (4 \to 1) = 4$
3 1

$mm(d[i][k] + d[k][j])$

UCn → Dh 0 —

0 ⊥ 5 0 ⊥ 5

0 ↑ 5



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

cost

[0][1p] = 0→0 + 0→1

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

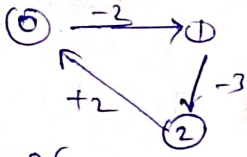
[1][2] = [1][0] + [0][2]
1 + ∝

@Ashish Kumar Nayak

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 intution, it is a simple brute on all paths. Focus on the three for loops.

(*) How to detect -ve cycle.



$-2 + 2 - 3$

cost $= -3$

if the costing of any node cost[i][i] < 0 (node to node) it means there will have negative cycle.

Code :-

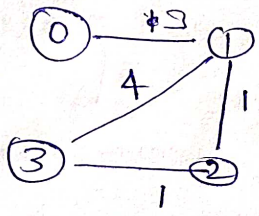
```
void Shortest_distance(vector<vector<int>>& matrix)
{ int n = matrix.size();
  for(int i=0; i<n; i++)
  { for(int j=0; j<n; j++)
    { if(matrix[i][j] == -1)
      { matrix[i][j] = 1e9;
      }
      if(i==j)
      { matrix[i][j] = 0;
      }
    }
  }
  for(int k=0; k<n; k++)
  { for(int i=0; i<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] == 1e9) { matrix[i][j] = -1; }
      if(i==j) { matrix[i][j] = 0; }
    }
  }
}
```

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

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

@Aashish Kumar Nayak

City with the smallest no. of Neighbours at a threshold distance



threshold = 4

city

0 → 1, 2

1 → 0, 2, 3

2 → 0, 1, 3

3 → 1, 2

lowest no. of neighbours

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

0, 3

largest one is 3

∴ Ans = 3

cnt = 0

cntMax = 5 ∵ we have 4 cities

city = -1

for 0 including itself

cnt = 1 + 2 + 3 + 0 ⇒ cnt = 3

cntMax = 3

for 1

cnt = 1 + 2 + 3 + 4

for 2

cnt = 1 + 2 + 3 + 0 ⇒ cnt = 3

for 3

cnt = 1 + 2 + 3 ⇒ cnt = 3

cntMax = min(cnt, cntMax)

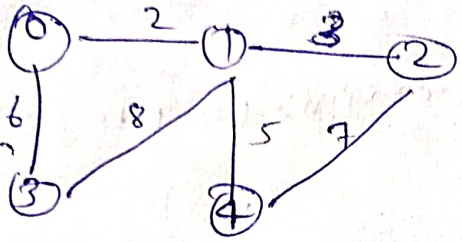
take largest one.

@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++)
  { dist[i][i]=0; }
for(int k=0; k<n; k++)
  { for(int i=0; i<n; i++)
    { for(int j=0; j<n; j++)
      { if(dist[i][k] == INF MAX || dist[k][j] == INT_MAX)
          continue;
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
      } } }
int cntCity = n;
int cityNo = -1;
for(int city = 0; city < n; city++)
  { int cnt = 0;
    for(int adjCity = 0; adjCity < n; adjCity++)
    { if(dist[city][adjCity] <= distanceThreshold)
        cnt++;
    }
    if(cnt <= cntCity)
    { cntCity = cnt;
      cityNo = city;
    }
}
return cityNo;
}
```

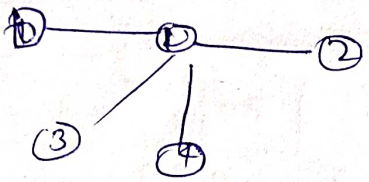
@Aashish Kumar Nayak

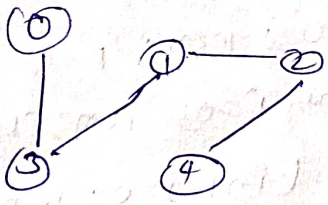
Minimum Spanning Tree (MST) :-

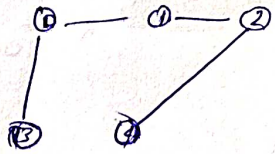


N-nodes (5)
M-edges (6)

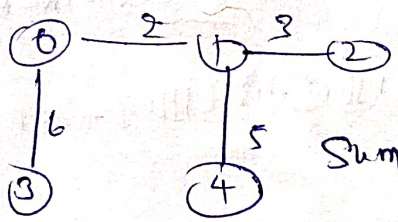
A tree in which have N nodes & N-1 edges & all nodes are reachable from each other.



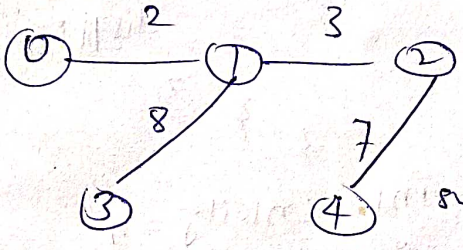




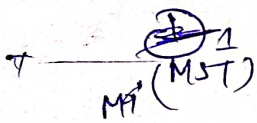
For this graph



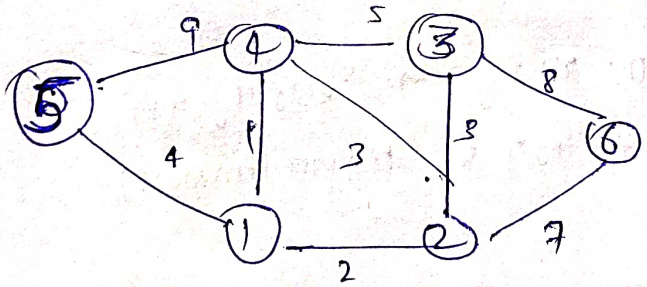
Sum = 16



sum = 20



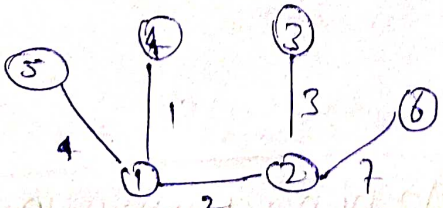
Both are spanning tree but just one is Minimum spanning tree.



for finding MST we have

(i) prim's Algorith
(ii) Kruskal's Algorith
↳ Disjoint set

spanning tree are :-

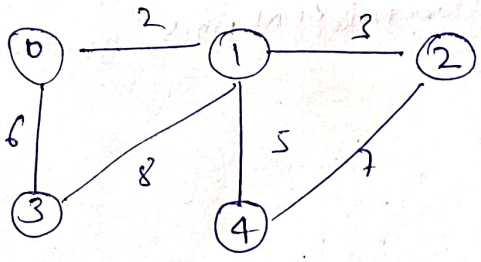


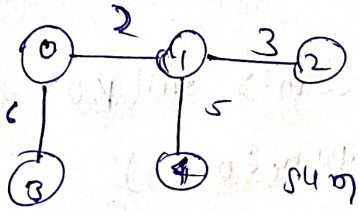
sum = 17

@Aashish Kumar Nayak

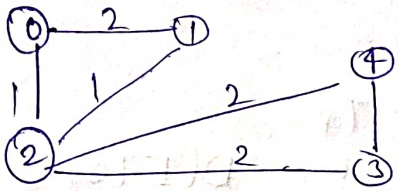
Prim's Algorithm :

It helps to find MST (Minimum spanning Tree).





sum = 16



vis[]

1Ø	1Ø	1Ø	ØL	Ø1
0	1	2	3	4

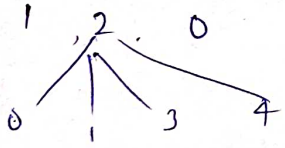
(wt, node, Parent)
Min-heap

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

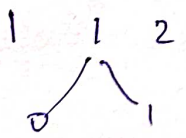
least wt always

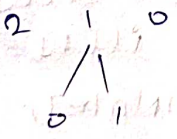
node parent — it means its a 1st node so dont add in MST

look at adjacent of node = 0

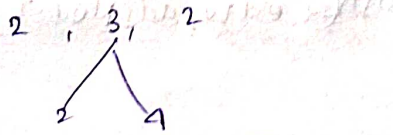


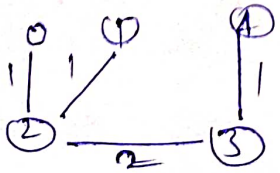
sum = ~~0~~ + 2





MST → [(0,2) (1,2) (2,3) (3,4)]





@Aashish Kumar Nayak

Code :-

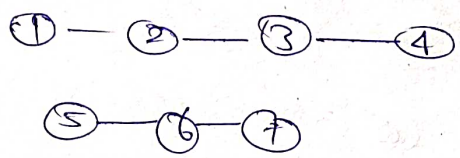
```
int spanningTree(int V, vector<vector<int>> adj[])
{
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<int,int>> pq;
    vector<int> vis(V,0);
    pq.push({0,0});
    int sum = 0;
    while(!pq.empty())
    {
        auto it = pq.top();
        pq.pop();
        int node = it.second;
        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;
}
```

T.C- O(E log E)

Disjoint Set Data structure :

important for Online Assessments wise.

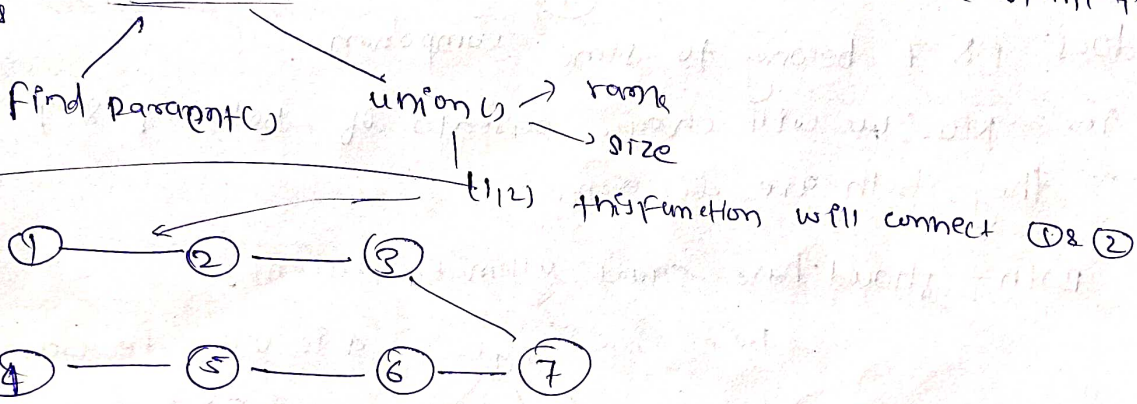
@Aashish Kumar Nayak



1 & 5 are connected or not?

→ No (use BFS/DFS Traversal to find)
takes O(N+E)
i.e. Brute force =

But disjoint set. will do this same on constant time.
It gives



graph is changing and we can find parent or union at any stage. is two nodes are ~~com~~ part Connected or not

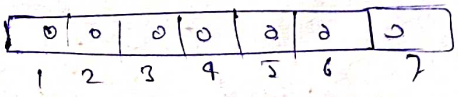
Dynamic

union() → rank
→ size

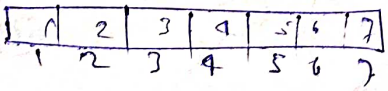
we can implement union in two ways rank and size.

① union() by rank

rank array



parent

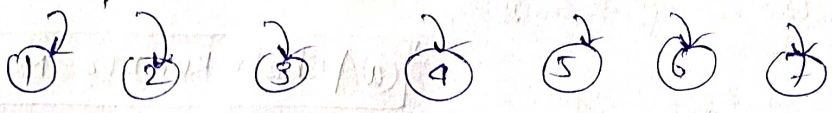


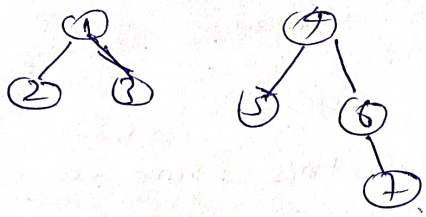
union(u,v)

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

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

parents | 1 | 2 | 3 | 4 | 4 | 4 | 6 |
1 2 3 4 5 6 7



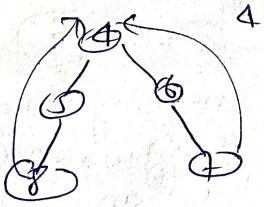


does 1 & 7 belong to same component?

Ans - No., we will check parents of both 1 & 7

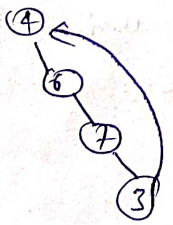
∴ they both are different

Both should have equal ultimate parent.

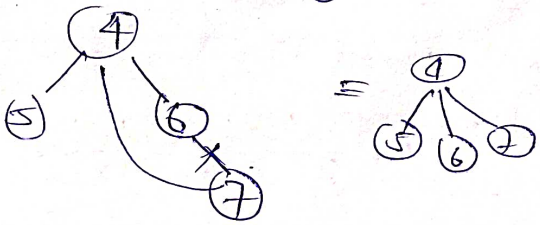
like 

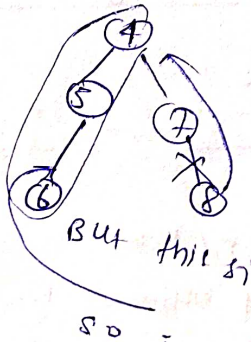
4 is ultimate parent of 8 & 7

path compression



we will store ultimate parent of each node.





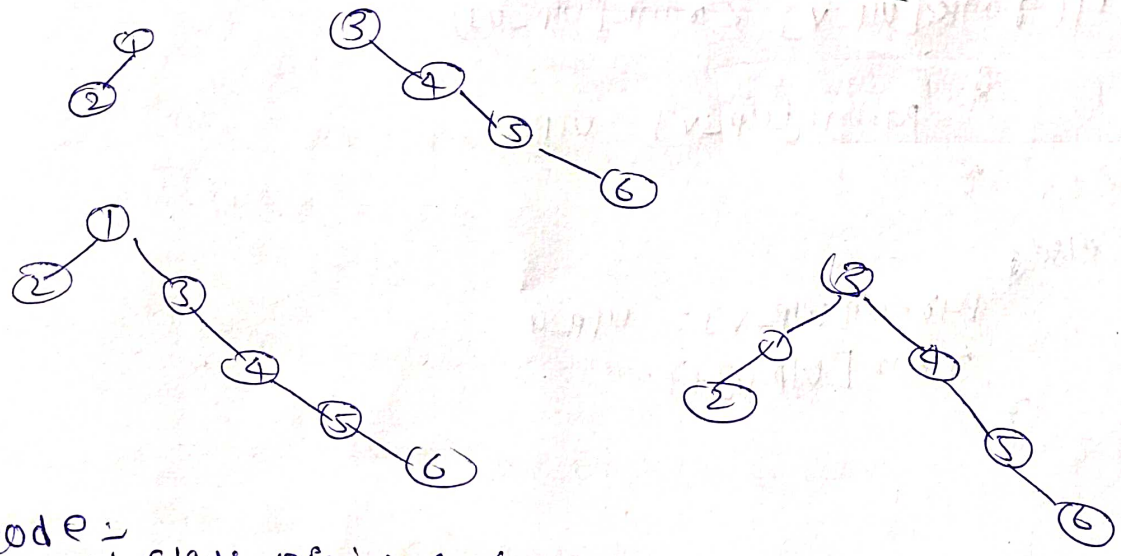
But this side is still same so it can't be reduced

* Heuristic it is called rank.

not height

@Aashish Kumar Nayak

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(rank[ulp_v~~
      if(ulp_u == ulp_v)
        return;
      if(rank[ulp_u] < rank[ulp_v])
        { parent[ulp_u] = ulp_v;
        }
```

@Aashish Kumar Nayak

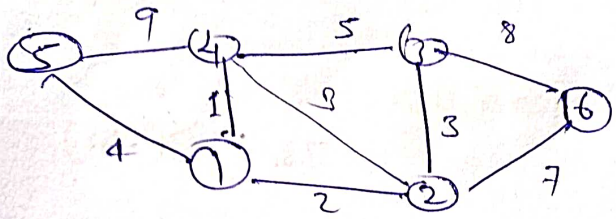
```
else if( rank[ulp_v] < rank[ulp_u])
        {
          parent[ulp_v] = ulp_u;
        }
    else{
          parent[ulp_v] = ulp_u;
          rank[ulp_u]++;
        }
  }
};
int main()
  {
    DisjointSet ds1(7), ds2(8);
      ds.unionByRank(1,2);
      ds.unionByRank(2,3);
        //          (4,5);
        //          (6,7);
        //          (5,6);
    if(ds.findUPar(3) == ds.findUPar(7))
      { cout<<"same\n";
      }
    else    cout<<"Not same\n";
    // ds.unionByRank(3,7);
    // ds.unionBySize(3,7);  > dont use Both simultaneously.
      if(ds.findUPar(3) == ds.findUPar(7))
        { cout<<"same\n";
        }
      else cout<<"Not same\n";
      return 0;
}
```

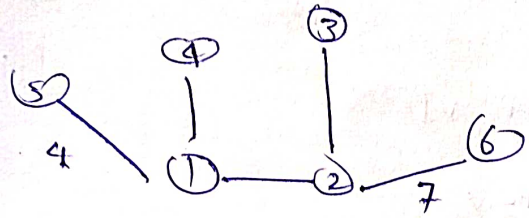
```
void unionBySize(int u, int v)
{
    int ulp_u = findUpar(u);
    int ulp_v = findUpar(v);
    if(ulp_u == ulp_v)
        return;
    if(rank[ulp_u] < rank[ulp_v])
        {
            parent[ulp_u] = ulp_v;
    else if(rank[ulp_v] <= rank[ulp_u])
        {
            parent[ulp_v] = ulp_u;
  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

T.C.: $O(4\alpha)$ ≈ constant

@Aashish Kumar Nayak

Kruskal's Algorithm





Sort all the edges acc to wt.

(wt, u, v)

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

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 wt = it[1];
            int node = i;
            edges.push_back({wt, {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.findUPar(u) != ds.findUPar(v))
    {
        mstWt += wt;
        ds.unionBySize(u,v);
    }
}
return mstWt;
}
```

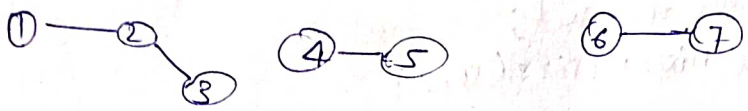
T.C.

$\approx O(M \times 4 \times 4 \times 2)$

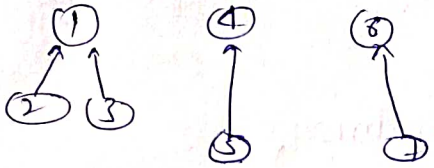
$\approx O(M)$

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

No. Number 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	0	1
7	0	0	0	0	0	1	0



Count the no. of unique ultimate parent.

or if count findUP(u) == u then cnt++

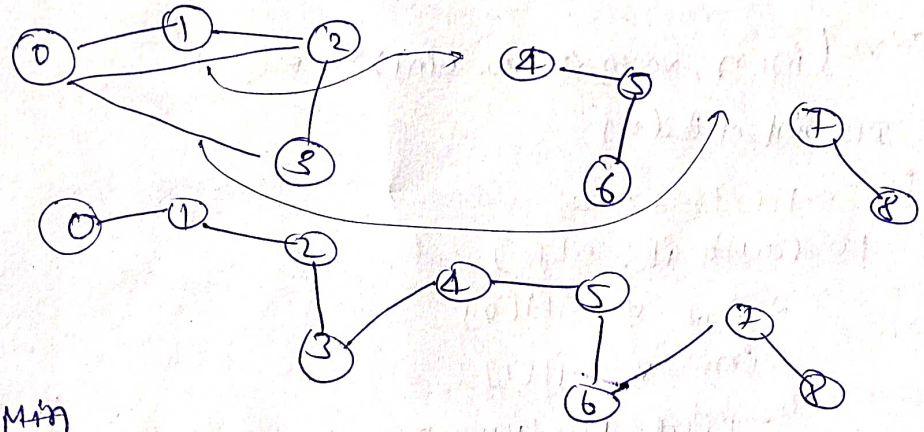
Code :-

```
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);
          }
       }
    }
    int cnt = 0;
    for(int i=0; i<V; i++)
    {  if(ds.parent[i] == i)
          cnt++;
    }
    return cnt;
}
```

```
class DisJointSet {
    {
    };
```

@ Aashish kumar Nayak

Min no. of operations to make a graph connected



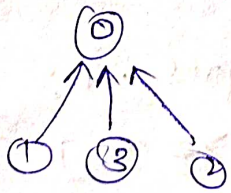
Min
for n components we require (n-1) edges

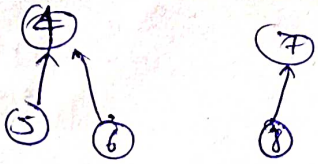
no. of connected components -> nc

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

```
if(extraedges >= ans)
        return ans;
else
        return -1;
```

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





extra = 2

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

@Ashish Kumar Nayak

```
class DisjointSet {
    ==
};

int solve(int n, vector<vector<int>>& edges)
{
    DisjointSet ds(n);
    int cntExtras = 0;
    for(auto it : edge)
    {
        int u = it[0];
        int v = it[1];
        if(ds.FindUPar(u) == ds.FindUPar(v))
        {
            cntExtras++;
        }
        else
        {
            ds.unionBySize(u, v);
        }
    }
    int cntC = 0;
    for (int i=0; i<n; i++)
    {
        if(ds.parent[i] == i)
            cntC++;
    }
    int ans = cntC - 1;
    if(cntExtras >= ans)
        return ans;
    return -1;
}
```

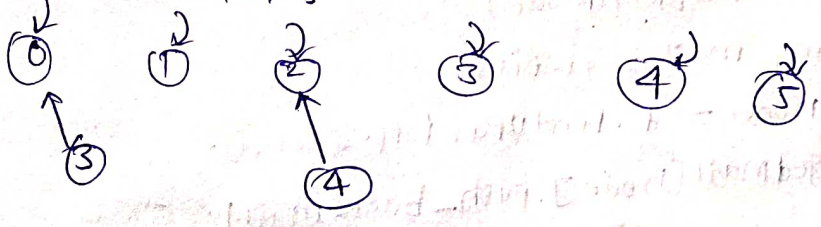
Account Merge - DSU

["John", "J1@gmail.com", "J2@com", "J3@com", "J5@com"] — 0
["John", "J4@com"] — 1
["raj", "R1@com", "R2@com", "R3@com"] — 2
["John", "J1@com", "JJ@com"] — 3
["raj", "R2@com", "R3@com"] — 4
["mary", "M1@com"] — 5

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

J1@com → 0
J2@com → 0
J3@com → 0
J4@com → 1
R1@com → 2
R2@com → 2
J5@com → 3
R3@com → 4
M1@com → 5

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



0 → J1@com, J2@com, J3@com, J5@com
1 → J4@com
2 → R1@com, R2@com
3 →
4 →
5 → M1@com

@Aashish Kumar Nayak

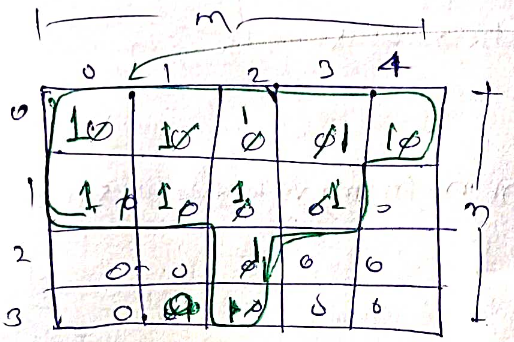
@Ashish Kumar Nayak

Code :-

```
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=1; j<details[i].size(); j++)
        {
            if(mapMailNode.find(mail) == mapMailNode.end())
            {
                mapMailNode[mail] = i;
            }
            else
            {
                ds.unionBySize(i, mapMailNode[mail]);
            }
        }
    }
    vector<string> mergeMail[n];
    for(auto it : mapMailNode)
    {
        string mail = it.first;
        int node = ds.findUPar(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



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

{ 1 1 2 1 1 2 2 2 3 3 1 1]

⇒ formula (row, col) = (row × m) + col
↓
node

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

cnt: ~~0~~ 1 ~~2~~ ~~3~~ ~~2~~ 1 ~~2~~ ~~1~~ ~~2~~ 2 ~~3~~ 2 ~~3~~ ~~4~~ 3 ~~4~~ ~~3~~ ~~2~~ 1 ~~2~~ (1)

class DisjointSet {

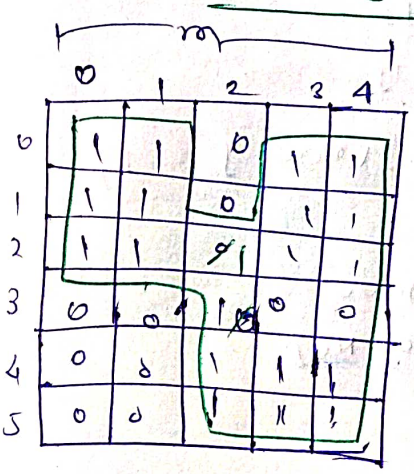
}

@Aashish Kumar Nayak

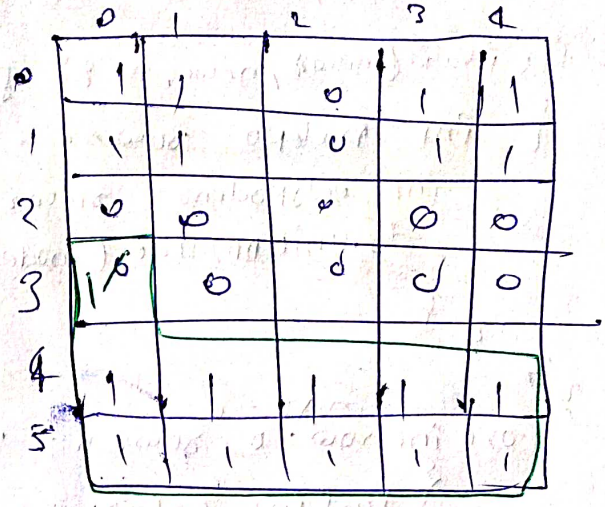
```
bool isvalid(int adjr, int adjc, int n, int m)
{ return adjr >=0 && adjr < n && adjc >= 0 && adjc < m;
}
public: vector<int> numOfIslands(int n, int m, vector<vector<int>>
&operators)
{ DisjointSet ds(n * m);
  int vis[n][m];
  memset(vis, 0, sizeof vis);
  int cnt = 0;
  vector<int> ans;
  for(auto 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 adjc = col + dc[ind];
    if(isvalid(adjr, adjc, n, m))
      { if(vis[adjr][adjc] == 1)
        { int nodeNo = row * m + col;
          if(ds.findUpar(
          int adjNodeNo = adjr * m + adjc;
          if(ds.findUpar(nodeNo) != ds.findUpar(adjNodeNo))
            { cnt--;
              ds.unionBySize(nodeNo, adjNodeNo);
            }
        }
      }
  }
  ans.push_back(cnt);
}
return ans;
}
```

@Aashish Kumar Nayak

Making ~~Connect~~ a Large Island - DSU



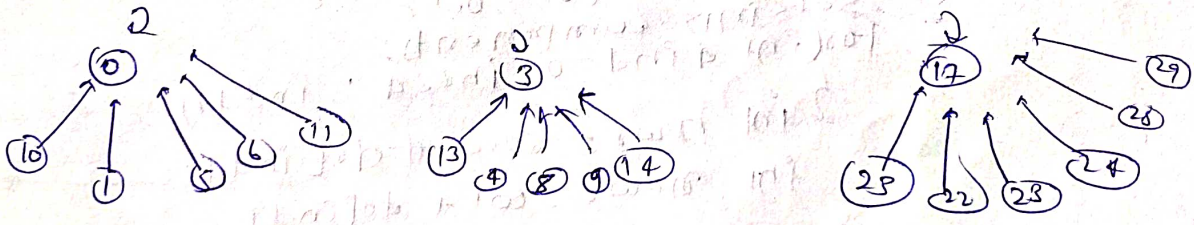
size = 6 + 6 + 7 + 1
= 20



size = 10 + 1
= 11

use for using Disjoint set we have to convert our indexes into a single no. so that we can map it and use.

formula = (row * m) + col




```
class DisjointSet {
    ==
    ==
};
class solution {
public:
int maxConnection(vector<vector<int>>& grid)
{ int n=grid.size();
  DisjointSet 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

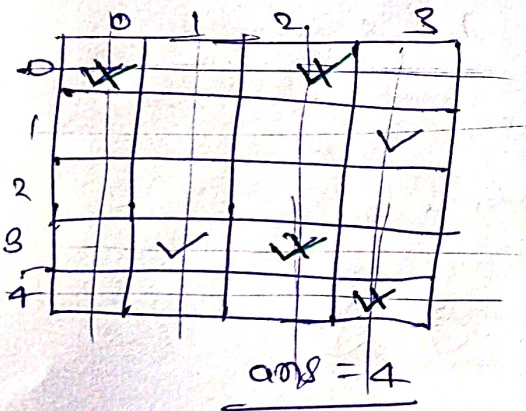
```
for(int ind = 0; ind<4; ind++)
    { int newr = row + dr[ind];
    int newc = col + dc[ind];
if( isValid(newr, newc, n) && grid[newr][newc] ==1)
{ int nodeNo = row*n + col;
  int adjNodeNo = newr*n + newc;
  ds.unionBySize( nodeNo, adjNodeNo);
  }
  }
}
  int mx = 0;
for(int row =0; row<n; row++)
  {
    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 ind=0; ind<4; ind++)
        { int newr = row + dr[ind];
          int newc = col + dc[ind];
          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);
}}
```

@Aashish Kumar Nayak

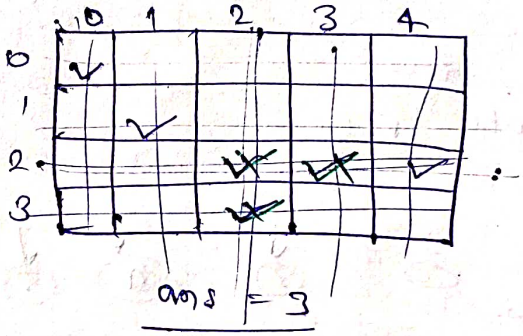
```
for(int cellNO=0; cellNo < n*n ; cellNO++)
{
  Mx = Max(Mx, ds.size[ds.findUpar(cellNo)]);
}
return mx;
}
```

T.C. $O(N^2 * \alpha)$

Most Stones Removed with same row or column

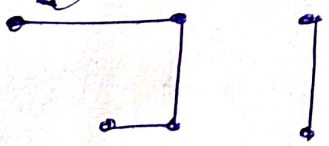


ans = 4

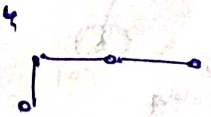


ans = 3

Components ↓ Remove 1 element in each component



= (4-1) + (2-1)
= 3 + 1
= 4



= (1-1) + (1-1) + (4-1)
= 0 + 0 + 3
= 3

Q.
If there are n stones.

C_1	C_2	C_3	C_4
x_1 stones	x_2 stones	x_3 stones	x_4 stones

$x_1 + x_2 + x_3 + \dots = n$ stones

R stones are removed $(x_1 - 1) + (x_2 - 1) + (x_3 - 1) + \dots =$

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

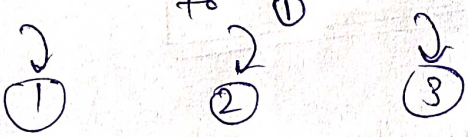
ans = n − no. of components.

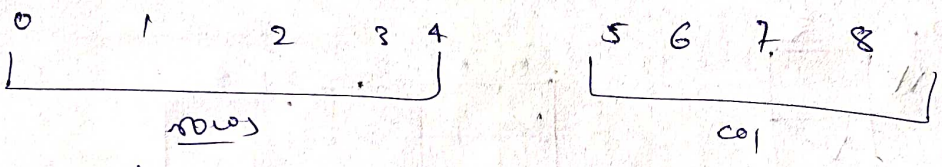
n = no. of stones

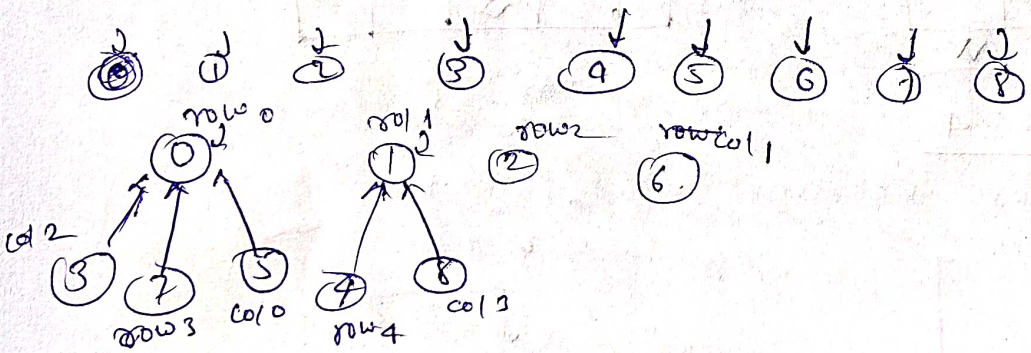
Here we are connecting stones to make node.
so we use disjoint set data structure.

we will treat row as a node

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







2 connected components

for all the stone there is 2 ultimate parents

ans = 6 − 2 = 4 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.

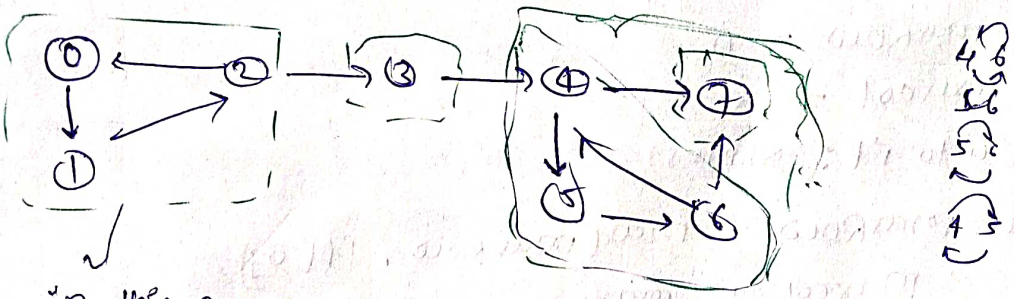
@Aashish Kumar Nayak

Code :-

```
class DisjointSet {
};
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] = 1;
        stoneNodes[nodecol] = 1;
    }
    int cnt = 0;
    for(auto it : stoneNodes)
    {
        if(ds.findUPar(it.first) == it.first)
        {
            cnt++;
        }
    }
    return n - cnt;
}
```

Strongly connected components (SCC) :-

only valid for Directed graph (Kosaraju's Algorithm)



in this component every pair is reachable

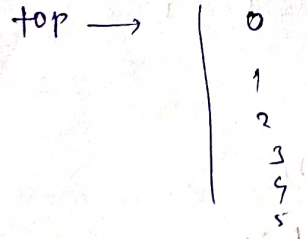
0 2
2 0
1 2
2 1
1 0
0 1

Strongly connected component

0 1 2 | 3 | 4 5 6 | 7

4 strongly connected component

*) sort all the edges according to finishing time.
*) Reverse the graph
*) Do a dfs.



Code :-

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

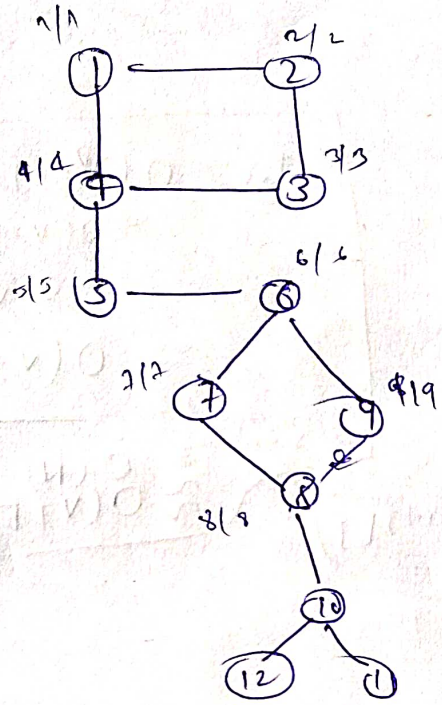
```
void dfs3(int node, vector<int> &vis, vector<int> adjT[])
{
  vis[node] = 1;
  for(auto it : adjT[node])
  {
    if(!vis[it])
    {
      dfs3(it, vis, adjT);
    }
  }
}

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];
  for(int i=0; i<V; i++)
  {
    vis[i] = 0;
    for(auto it : adj[i])
    {
      adjT[it].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;
}
```

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

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

@Aashish Kumar Nayak

Bridges in Graph :-



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

4 - 5
5 - 6
10 - 8

disc[] → DFS time inserts on

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

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

DFS(8)
⇅
DFS(9)

Code :-

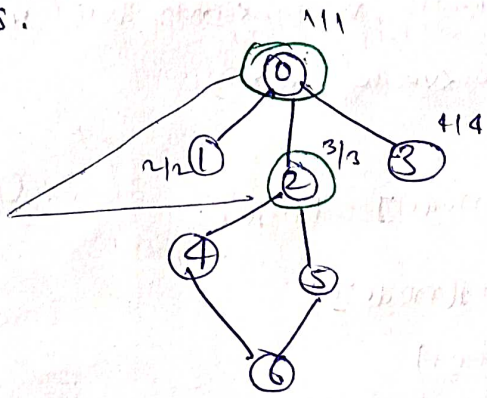
```
int timer = 1;
void dfs( int node, int parent, vector<int> &vis, vector<int> adj[],
int tin[], int low[], vector<vector<int>> )
{   vis[node] = 1;
    tin[node] = low[node] = timer;
    time++;
    for(auto it: adj[node])
    { if(it == parent)
        continue;
      if(vis[it] == 0)
      { dfs(it, 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;
}
```

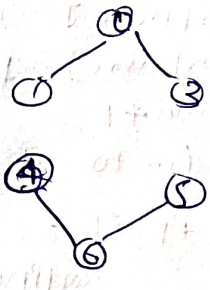
T.C ≃ O(V+2E)

@Aashish Kumar Nayak

Articulation point :-

Nodes on whose removal the graph breaks into multiple components.



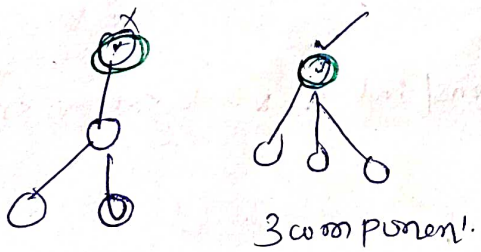


Time of insertion Tin[] → store the time of insertion during the DFS.

low[] → min of all adjacent nodes apart from the parent & visited nodes.

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

dfs(2) → 3
dfs(3) → 1 it can reach to 3.



1 component

3 components.

component should be >= 2

Code :-

```
int timer = 0;
void dfs(int node, int parent, vector<int> &vis, int tin[], int
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]);
            if(low[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);
        }
    }
```

T.C. ≈ O(V + 2E)
S.C. ≈ O(N)

@Aashish Kumar Nayak

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

END

@Aashish Kumar Nayak