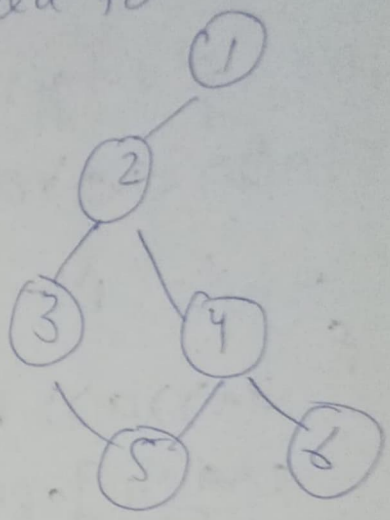
Graph representation

i) Adjacency Matrix

current node no. ↓ / connected to →	1	2	3	4	5	6
1	0	1	0	0	0	0
2	1	0	1	1	0	0
3	0	1	0	0	1	0
4	0	1	0	0	1	1
5	0	0	1	1	0	0
6	0	0	0	1	0	0



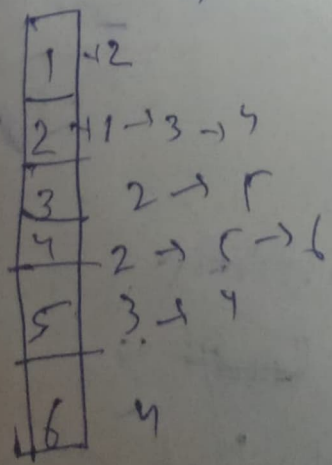
1 means connected 0 means not connected

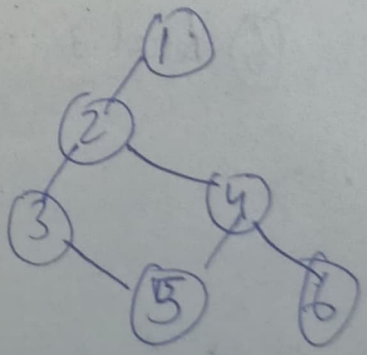
This representation require a lot of space $O(n^2)$

1) BFS or DFS is (time complexity → $O(n^2)$)

3) Adding new node is not easy

ii) Adjacency List





~~vector<int>~~

no of nodes : N

No of edges : M

vector <int> ar[N+1];

while(M--)

{

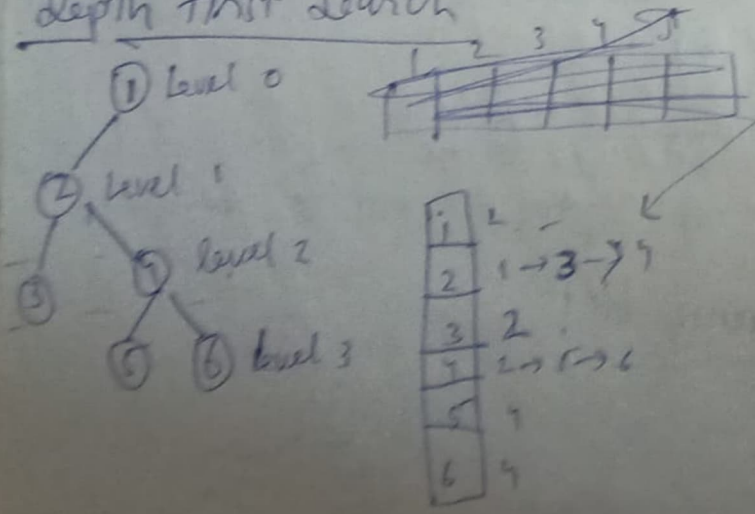
cin>>a>>b

ar[a].push_back(b);

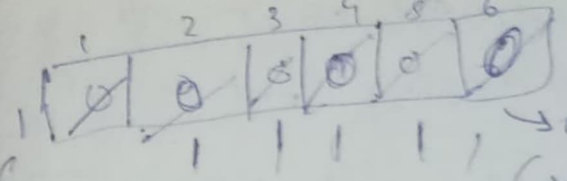
ar[b].push_back(a);

}

Depth First Search

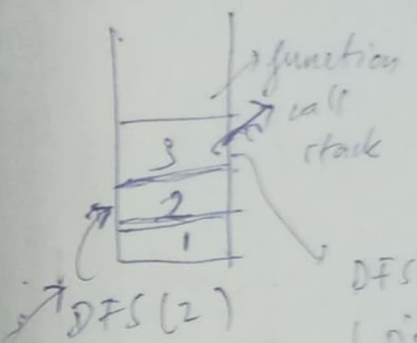


In this technique we first traverse all nodes at ~~level~~ a level then go to another level that only if it is ~~not~~ visited (level order traversal)



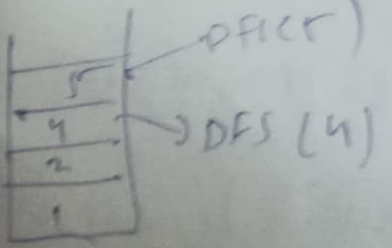
DFS(1)

(if node is visited it will marked 1 else 0)



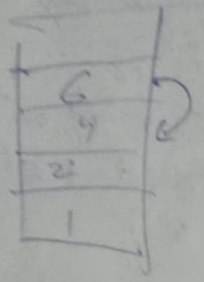
(DFS(3) will all call DFS(2) but DFS(2) is all ready visited so we won't go there)

so there is nothing after DFS 3 so it will pop out return.



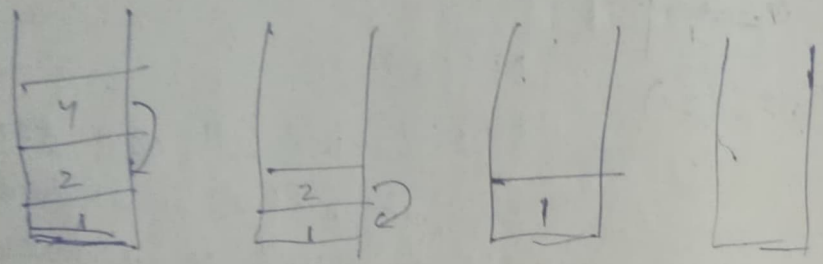
(~~node~~ DFS(5) will call DFS(4) but node 4 is ~~all~~ ready visited)

DFS(5) is completed so it will track back to DFS(4) and pop out



DFS(6) call DFS(4)

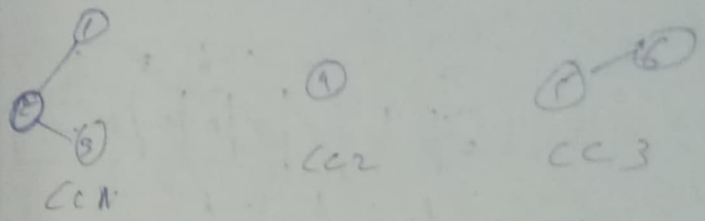
~~DFS~~ but it is visited




```
void dfs(int v)
{
  vis[v] = 1;
  cout << v << " -> ";
  for(int i=0; i<arr[v].size(); i++)
  {
    int child = arr[v][i];
    if(vis[child] == 0)
      dfs(child);
  }
}
```

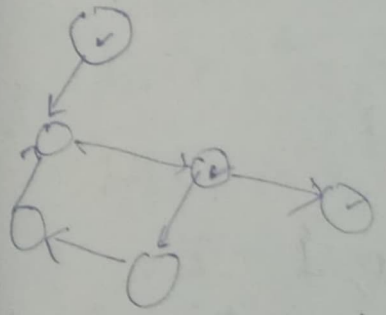
Vis is visited array and ~~arr~~ arr is adjancy list.

Connected component in graph



3 connected components

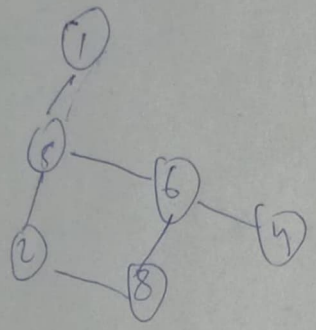
Strongly connected components

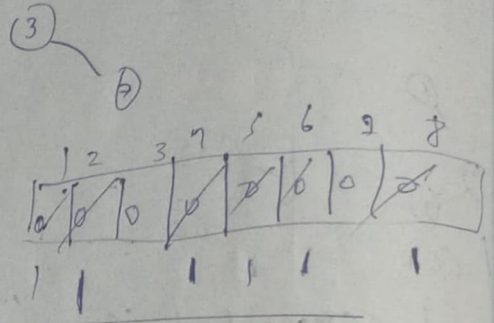


3 strongly cc

On chosing any two node in directed graph in order to be it a strongly connected components ~~if~~ there exist a path between one to another there should be no ~~path~~ ~~back~~ path to go back to starting node.

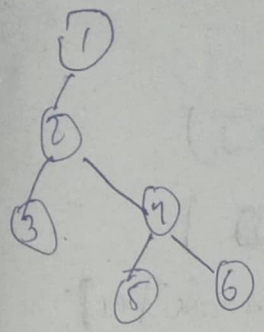
Counting number of CC

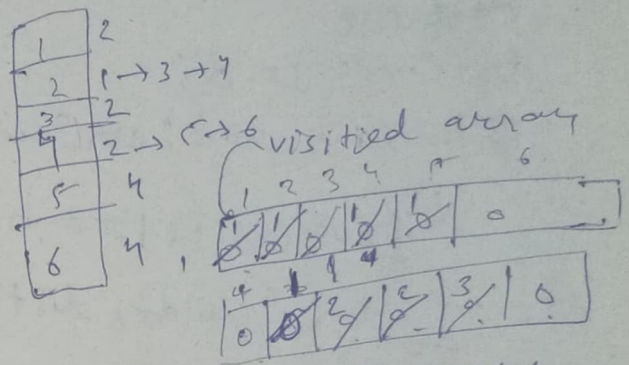





```
int main()
{
  :
  :
  int cc_count = 0;
  for(int i=1; i<=n; i++)
  {if(vis[i]==0)
    {dfs(i); cc_count++;}
  cout<<" # of cc = "<<cc_count;
}
```

Single Source Shortest path (on trees) using DFS





∴ Let 1 is source

dfs(1, 0) distance from source.

node is source

dfs(2, 0+1)

dfs(3, 1+1)

dfs(4, 1+1)

dfs(5, 2+1)

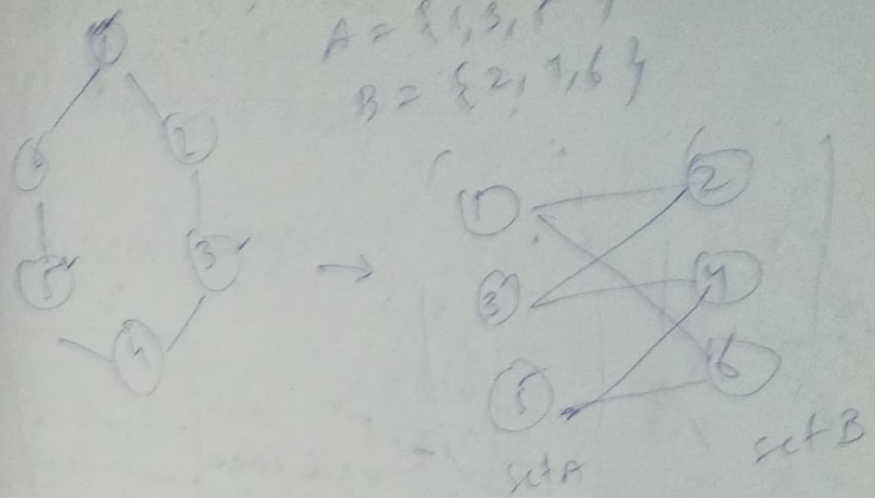
dfs(6, 2+1)

```
void dfs (int node, int d)
{
  vis[node] = 1;
  dist[node] = d;
  for(int child : adj[node])
  {
    if(vis[child] == 0)
      dfs(child, dist[node]+1);
  }
}
```

Bipartite Graph

A graph can be divided into 2 disjoint sets such that:

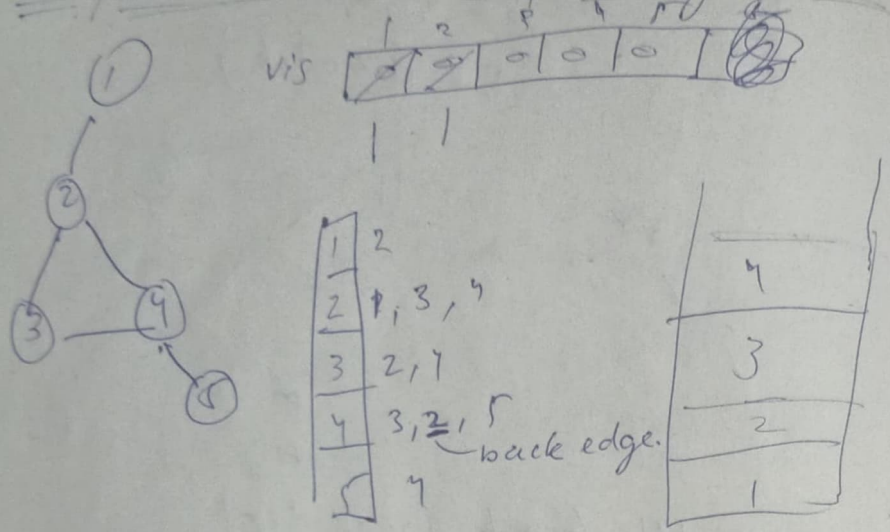
1) Each vertex belongs to exactly one of the 2 sets.
2) Each edge connects vertices of 2 different sets.



Bipartite Graph test

```
bool dfs(int v, int c)
{
  vis[v] = 1;
  col[v] = c;
  for(int child : adj[v])
  {
    if(vis[child] == 0)
    {
      if(dfs(child, c^1) == false)
        return false;
    }
    else
    if(col[v] == col[child])
      return false;
  }
  return true;
}
```

Cycle Detection in Graph Using DFS

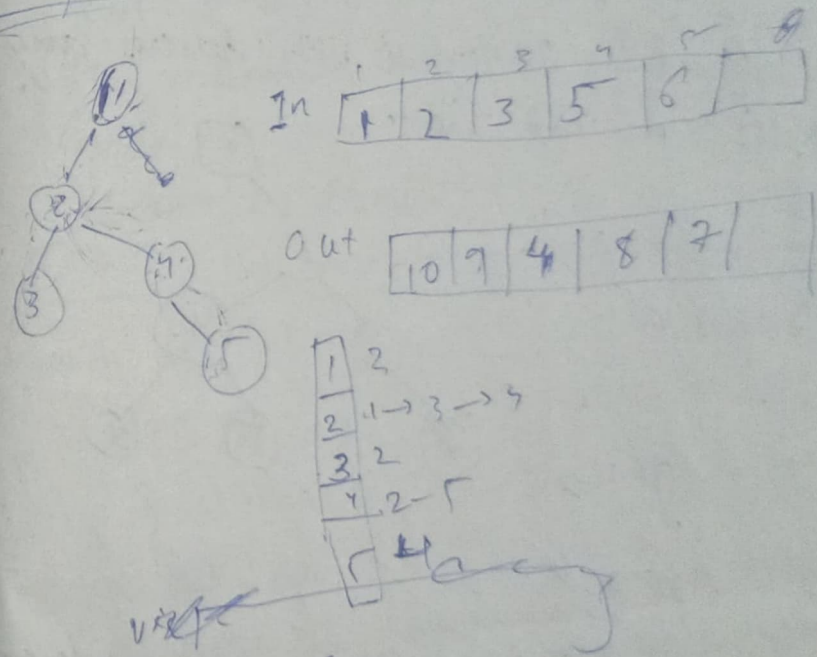


we would check if the edge connects parent to child if yes then it is not a backedge (& backedge occurs only in cyclic graph)

```
bool dfs(int node, int par)
{ vis[node] = 1;
  for(int child : ar[node])
  {
    if(vis[child] == 0)
    {
      int res = dfs(child, node);
      if(res == true)
        return true;
    }
```

```
    else
    if(child != par)
      return true;
  }
  return false;
}
```

In/out time of Nodes

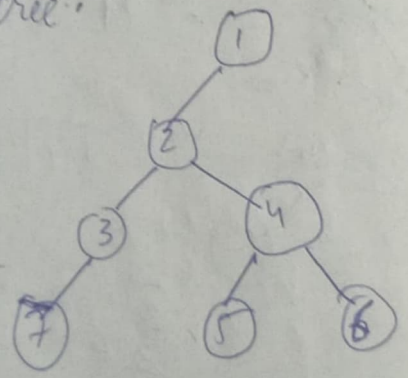



```
int timer = 1;
bool dfs(int v)
{ vis[v] = 1;
  in[v] = timer++;
  for(int child : ar[v])
  { if(vis[child] == 0)
```

```
    dfs(child);
  }
 }
 out[v] = timer++;
}
```

Diameter of Tree

It is defined as the longest path between any 2 nodes in the tree.



2 → 3 → 2 → 4 → 6
4 edges

7 → 3 → 2 → 4 → 5
4 edges

Novice Approch

We would run DFN, N times (for each node)
In each iteration, we would set ith node as root and find distance of farther node.

Running time $O(n^2)$

Better Approch

We can find diameter in only 2 DFS run.

Take any arbitrary node as root and run dfs from it and find the farthest node, let this node be x.

Run a dfs from node x and find the maximum distance from this node to any other node, this distance is diameter.

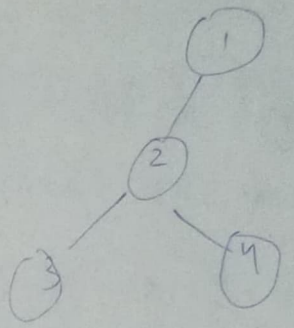
```
void dfs(int node, int d)
{ vis[node]=1;
  if(d>maxD){maxD=1, maxNode=node}
  for(int child: ar[node])
  { if(vis[child]==0)
    dfs(child, d+1)
  }
}
```

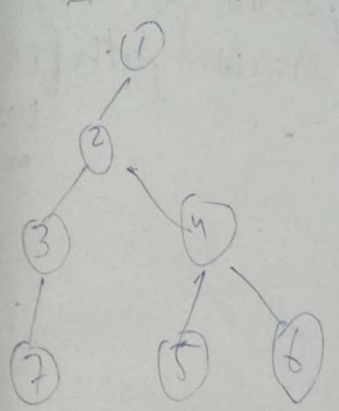
maxD = -1
dfs(1, 0); ← finding maximum from first node (it could be any node but the selected 1)
maxD = -1;
dfs(maxnode, 0)
cout << maxD;
}

int main()
{
:
:
maxD = -1;
dfs(1, 0); ← finding node with max dist from this node (Taking 1 - is not necessary)
vis
for(i=0; i<n; i++)
vis[i] = 0; ← Clearing vis array so second dfs call can run properly
maxD = -1
dfs(maxNode, 0)
cout << maxD;
}

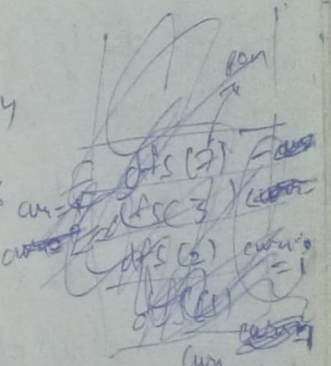
Calculating Subtree size using DFS in O(N)

subsize[1] = 4
subsize[2] = 3
subsize[3] = 1
subsize[4] = 1





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



int dfs(int node)
{
vis[node] = 1;
int curr_size = 1;
for(int child : adj[node])
if(vis[child] == 0)
curr += dfs(child);
}
subsize[node] = curr_size;

vis

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

subsize

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

return curr_size;
}

~~Breadth First Search~~

things we can do using dfs

1) count no of counted component
2) detect a cycle.
3) find if graph is Bipartite or not
4) find single source shortest path (in tree only)
5) In/Out time
6) Diameter of tree.
7) calculate sub tree size

$U = XY$

$$\frac{\partial U}{\partial x} = Y\frac{\partial X}{\partial x} \qquad \frac{\partial U}{\partial y} = X\frac{\partial Y}{\partial y}$$

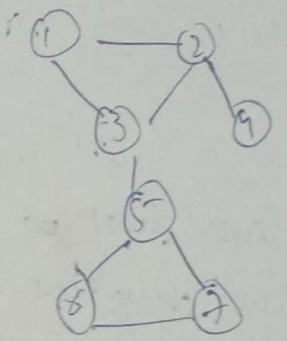
$$y^3 Y\frac{\partial X}{\partial x} + x^2 X\frac{\partial U}{\partial y} = 0$$

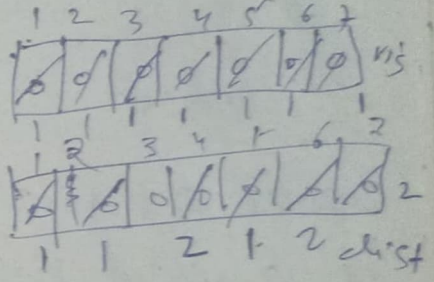
$$y^3 Y\frac{\partial X}{\partial x} = -x^2 X\frac{\partial Y}{\partial y} = k$$

Breth First search

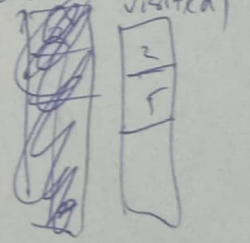
also called level order search
(level - distance from selected node (root node))

(distance - no of edges need to be traveled)

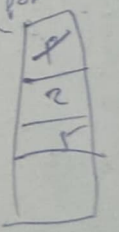




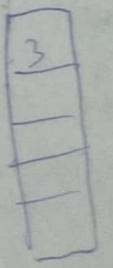
curr = 1
(2 and 3 are visited)



current = 3
pop



queue

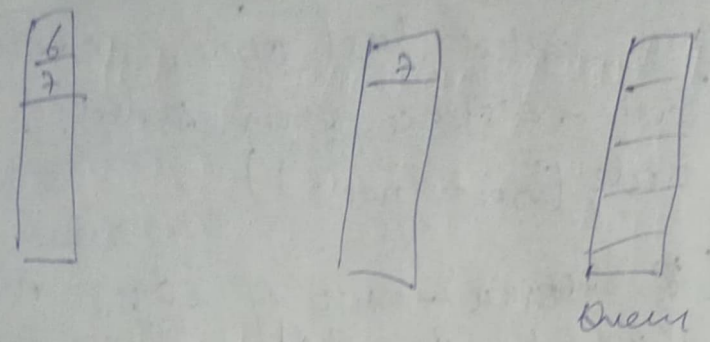


queue.

curr = 2

curr = 5

curr = 4 curr = 6 curr = 2



Reason why BFS is level order traversal

because element in lower level will be computed first then only the higher level element will be computed this is enforced by queue, element with lower level are more close to front.

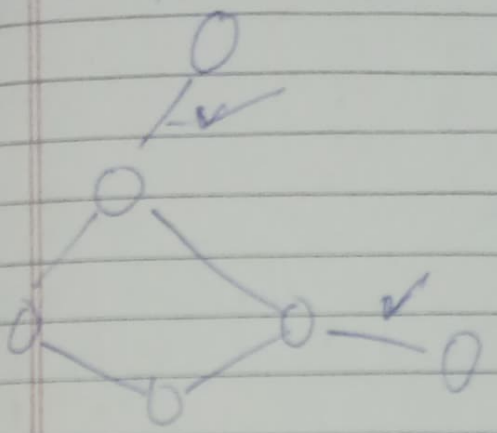
```
void BFS(int src)
{
  queue<int> q;
  q.push(src);
  vis[src] = 1
  dist[src] = 0;
```

```
while(!q.empty())
{ int curr = q.front();
  q.pop()
  for(int child : arr[curr])
    if(vis[child] == 0)
    {
      q.push(child);
      dist[child] = dist[curr] + 1;
      vis[child] = 1;
    }
}
}
```

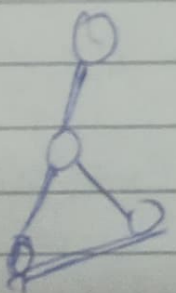
Date / / Page

Bridges in graph.

A bridge is defined as an edge which when removed makes graph disconnected or more precisely it increases the number of connected components

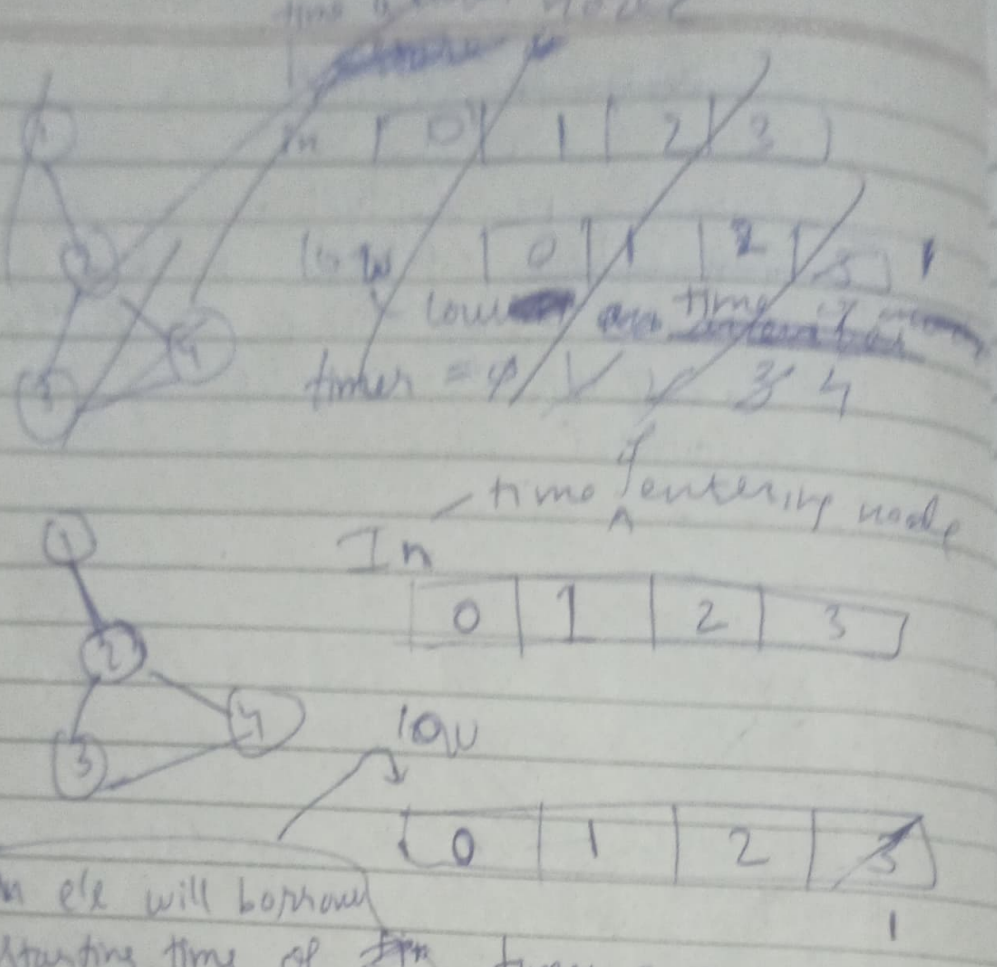


A Back edge & An edge connects ~~connects~~ ancestor node with its ancestor) it can not be a bridge



// normal edge

& // back edge

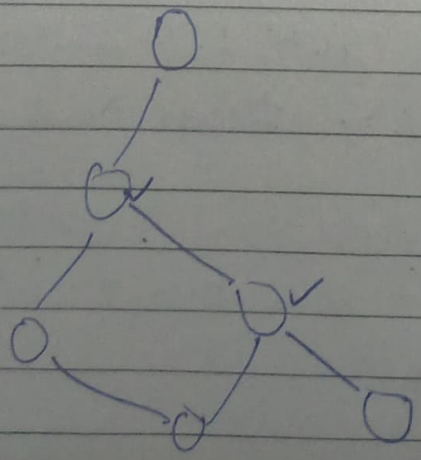



```
void dfs (int node, int par)
{
  vis[node] = 1;
  int[node] =  low[node] = time;
  timer++;
  for (int child : ar[node])
  {
    if (child == par) continue;
    if (vis[child] == 1)
    {
```

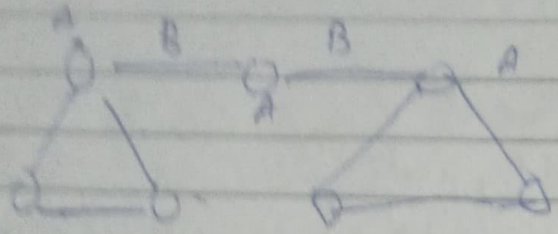
```
low[node] = min(low[node], in[child]);
}
else
{
    dfs(child, node);
    if(low[child] > in[node])
        cout << node << " - " << child <<
            " is a bridge";
    low[node] = min(low[node],
                    low[child]);
}
}
```

Articulation points

A vertex which when removed makes graph disconnected.



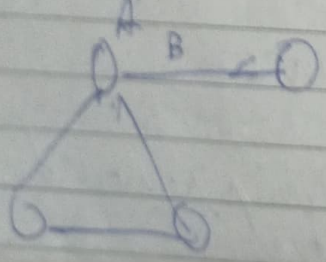
Relation between Bridges and articulation point.



B – Bridge
A – Articulation point

An end point of bridge is articulation point.

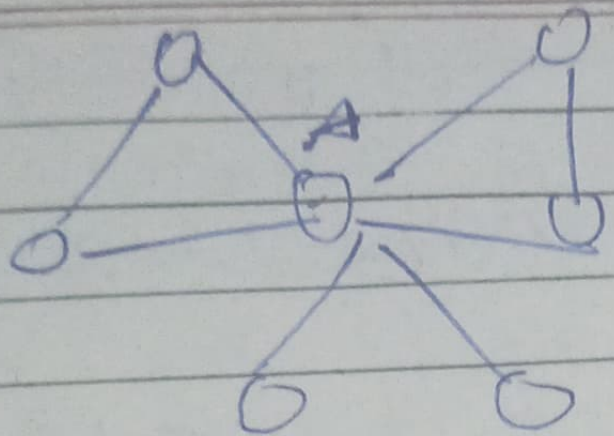
But not all end point of bridge is necessarily an articulation point.



degree of an articulation point should be more than one.

Articulation point may also exist without bridge.

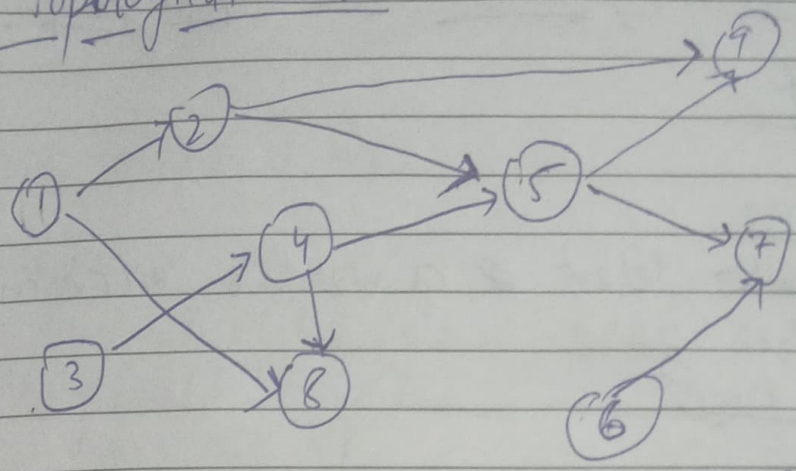
Date: __/__/__
Page:



Cut-vertex.

```
int vis[100000];
set<int> s;
int in[100000];
int low[100000];
int timer;
void dfs(int v, int p=-1){

vis[v] = 1;
in[v] = low[v] = timer;
timer++;
int children = 0;
for(int to: adj[v]){
    if(to==p) continue;
    if(vis[to]==1)
    {
        low[v] = min(low[v], in[to]);
    }else
    {
        dfs(to, v);
        low[v] = min(low[v], in[v]);
        if(low[to]>=in[v] && p!=-1)
            s.push_back(v);
        ++children;
    }
}
if(p==-1 && children>1)
    s.push_back(v);
}
```

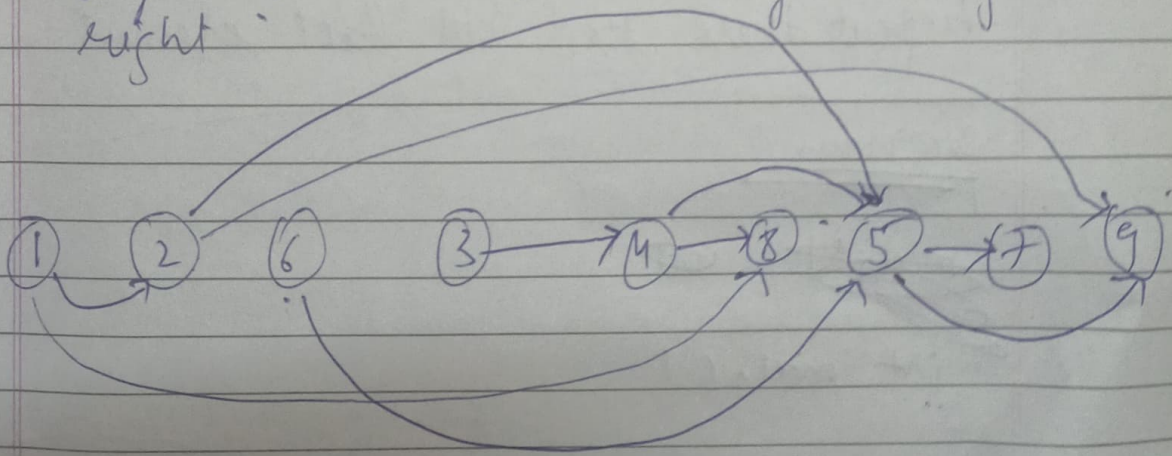
Topological Sort



Top sort = 1, 2, 6, 3, 4, 8, 5, 7, 9

7 depends on 5 and 6
5 depends on 2 and so on.

on listing node in topological order all edges will be directed from left to right.



~~Kahn's Topological~~

Kahn's Algorithm for topological sort

Step one.
Select a vertex with in degree zero
Step two.

Remove it and all edges from it
Step three.

add it in array
Step four.

Repeat these step will last element.

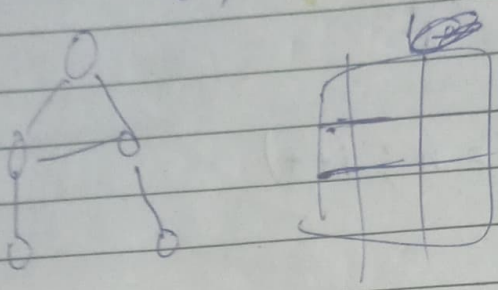
Program

```
int main()
{
int n, m, x, y;
cin>>n>>m;
for(int i=1; i<=m; i++)
  {cin>>x>>y;
arr[x].push_back(y);
in[y]++;  // in array it will in degree of edge.
}
```

```
    kahn(n);
}
void kahn(int n)
{
    queue<int> q;
    for(int i=1; i<=n; i++)
      if(in[i]==0)
        q.push(i);
    while(!q.empty())
    {
      int curr = q.front();
      res.push_back(curr);
      q.pop();
      for(int node : adj[curr])
      {
        in[node]--; // edge removal
        if(in[node]==0)
          q.push(node);
      }
    }

cout<<"Topsort: ";
for(int node : res)
  cout<<node<<" ";
}
```

Graph Algo on 2D Grid

1) Applying dfs on Grids.



1 cell = node
2-sides = Edges.

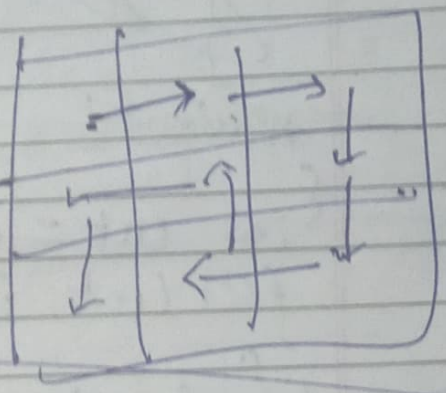
or

3 → sides + corners = Edges.

iff only sides are edges. then there can be only 4 ways to move in grid (so we don't need adjacency list in grid)

```
bool isvalid(int x, int y)
{
    if(x<1 || x>N || y<1 || y>M)
    return false.
    if(vis[x][y] == true)
    return false;

    return true;
}

void dfs(int x, int y)
{
    vis[x][y] = 1;
    cout<<x<<" "<<y<<endl;
```

```
if(isvalid(x-1,y)) // moving up
    dfs(x-1,y);
if(isvalid(x,y+1)) // moving right
    dfs(x,y+1);
if(isvalid(x+1,y)) // moving down
    dfs(x+1,y);
if(isvalid(x,y-1)) // moving left
    dfs(x,y-1);
}
int main()
{
cin>>n>>m
    dfs(1,1);
}
```

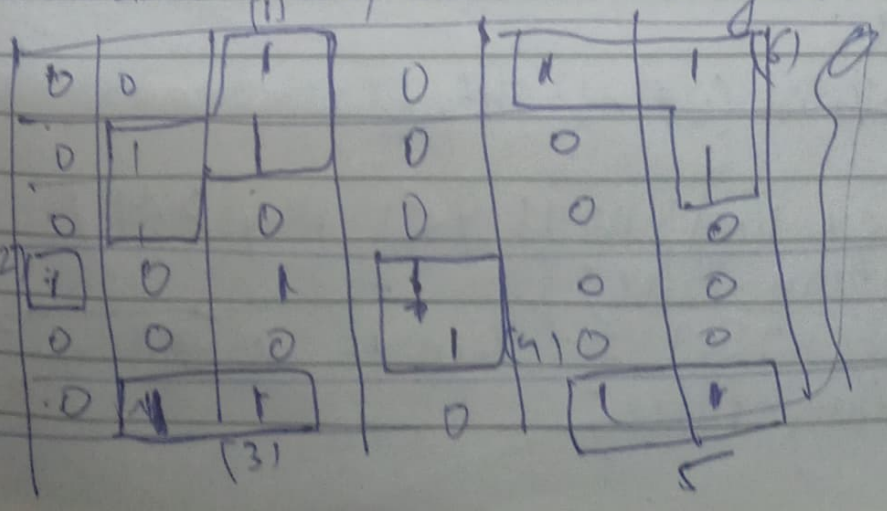


output

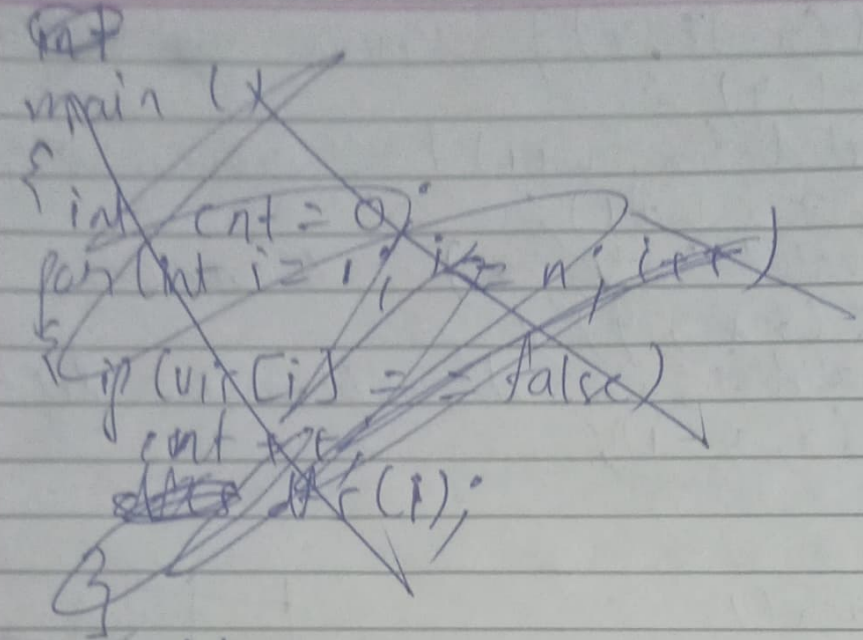
1,1
1,2
1,3
2,3
3,3

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

Connected component in grid.



6. connected component.

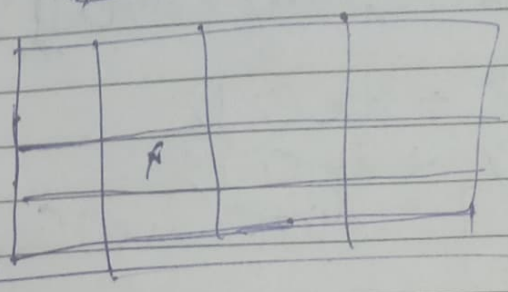


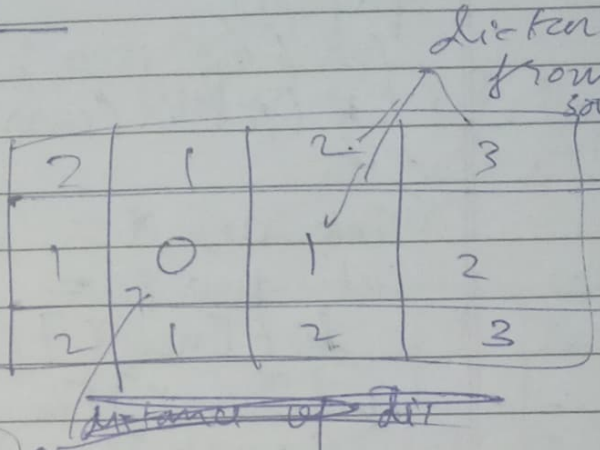
```
main()
{int cnt=0;
for(int i=1; i<=N; i++)
 { for(int j=1; j<=M; j++)
    if(arr[i][j]==1 and vis[i][j]==false
      {
       cnt++;
       dfs(i,j);
      }
 }

bool isValid(int x, int y)
{
 if(x<1 || x>N || y<1 || y>M)
  return false;
 if(vis[x][y]==true || arr[x][y]==0)
  return false;
```

return true;
}

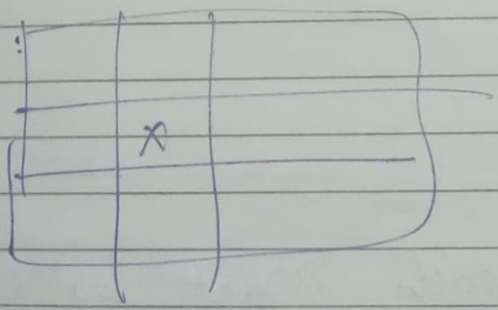
BFS on 2D Grid

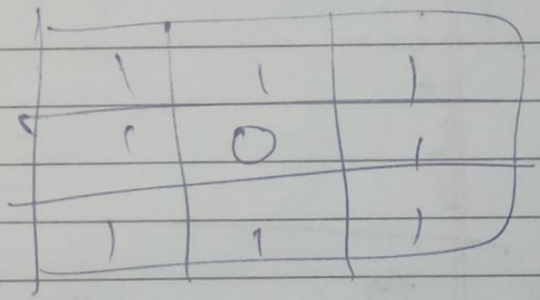
~~1) common sides~~





1) common sides as edges:-





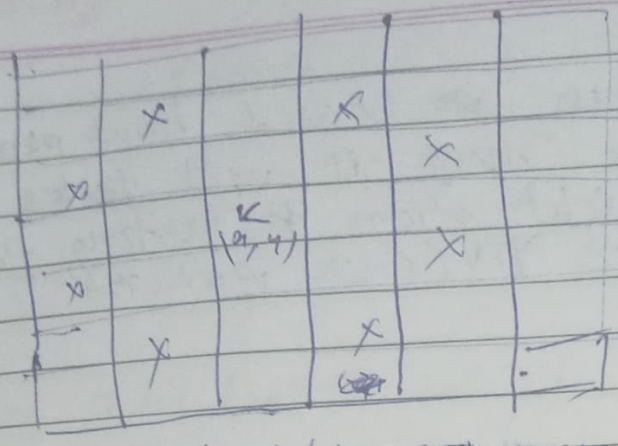
2) common side and corner ~~as~~ as edges.

```
void BFS (int srcX, int srcY)
{
    q.push({srcX, srcY});
    dist[srcX][srcY] = 0;
    vis[srcX][srcY] = 1;
```

```
while (!q.empty())
{
int currX = q.front().first;
int currY = q.front().second;
q.pop();
for(int i=0; i<4; i++)
if(isValid(currX + dx[i], currY + dy[i]))
{
int newX = currX + dx[i];
int newY = currY + dy[i];
dist[newX][newY] = dist[currX][currY]+1;
vis[newX][newY] = 1;
q.push({newX, newY});
}
}
}
```

Minimum moves to reach target by knight

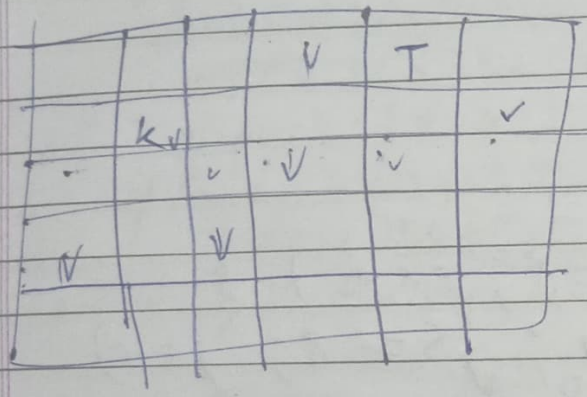
You are given a maze of size N X M there is a knight on point (a, b) and a target cell (a, b), find minimum number of steps to reach the target.

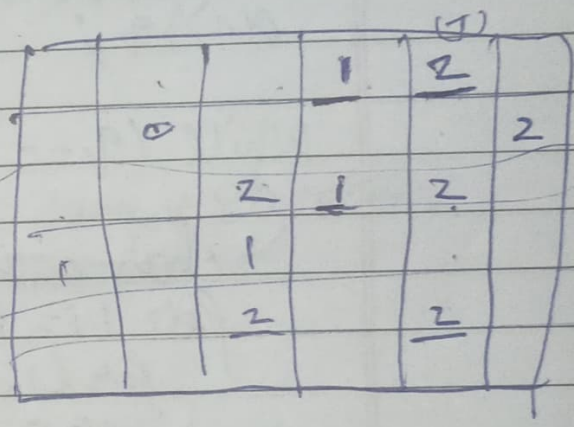


dx[] = {-2, -1, 1, 2, 2, 1, -1, -2}

dy[] = {1, 2, 2, 1, -1, -2, -2, -1}

Knight move





curr = (2, 2)

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

Queue.

curr (1, 4)

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

curr (3, 4)

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

(5, 3)

Example

Given an a chess board find how many minimum move it will take to move knight from starting point to destination point

```
int main()
{
int x, y, q;
cin>>q;

while (q--)
{
  char a, b;
  for(int i=1; i<=8; i++){
    for(int j=1; j<=8; j++)
      visit[i][j]=0;
  }

cin>>a>>b;
x = getX(a);
y = b - '0';
cin>>a>>b
target X = getX(a);   // target X,
target Y = b - '0';   target Y
cout<<BFS(x,y)<<endl;  in
}                      global
                       space
```

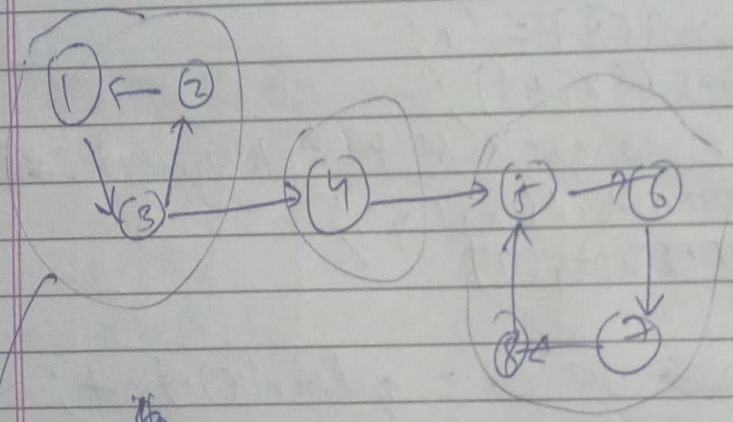
```
int getX(char a)
{
return a - 'a' + 1;
}

int BFS(int x, int y)
{
  queue<pair<int,int>> q;
   dist[x][y] = 0;
   vis[x][y] = 1;
   q.push({x, y});
  if(x == target X && y == target Y) return
    return dist[x][y];
while(!q.empty())
 {
  int curr_x = q.front().first;
  int curr_y = q.front().second;
  q.pop();
  for(int i = 0; i<8; i++)
   if(isValid(curr_x + dx[i], curr_y + dy[i]))
    {
     x = curr_x + dx[i];
     y = curr_y + dy[i];
     dist[x][y] = dist[curr_x][curr_y] + 1;
     vis[x][y] = 1;
     q.push({x, y});
     if(x == target X && y == target Y)
      return dist[x][y];
    }
 }
}
```

Kosaraju's Algo

Strongly connected component

It is a subset C of vertices such that for any 2 vertices in C there exists a path between them in the given graph

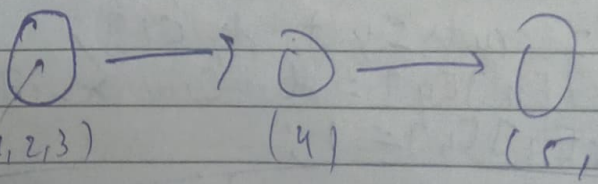
if $u, v \in C$
Then $u \to v$, $v \to u$



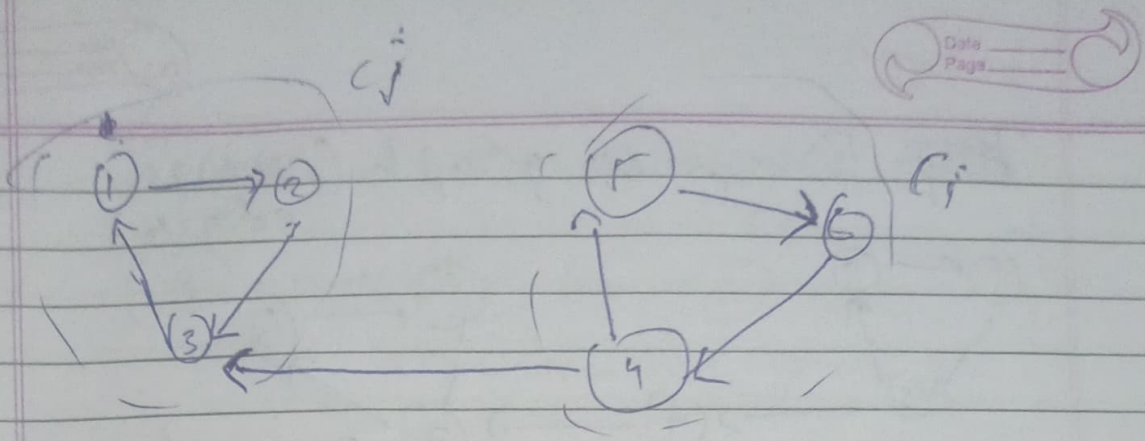
3 strongly connected component

condensation graph

Each Stro conn. comp. of original graph acts as a vertex in condensation gr



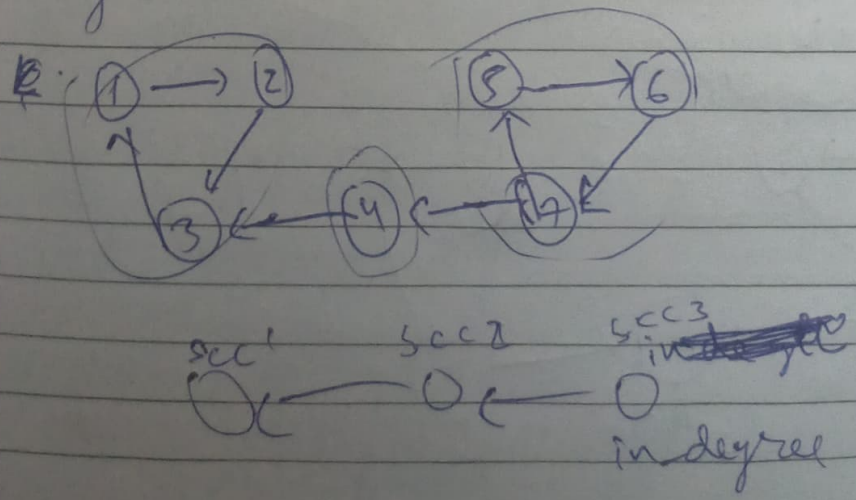
condensation graph can't have cycle.



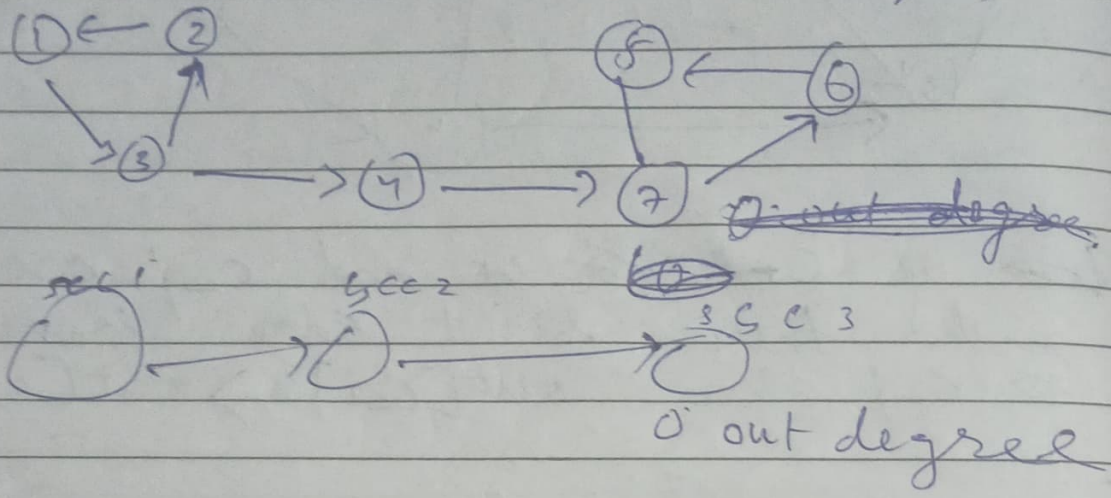
if c_i and c_j are scc and there exist edge from c_i to c_j then out[c_i] > out[c_j]

Important observation

1→ Graph and transposed Graph have the same strongly connected components.
2- Condensation graph.
3 - Acyclic property of condensation graph.
4- out[c_i] > out[c_j] if there is an edge from c_i to c_j in condensation graph.
5- In directed Acyclic graph there is at least 1 node with in-degree 0.

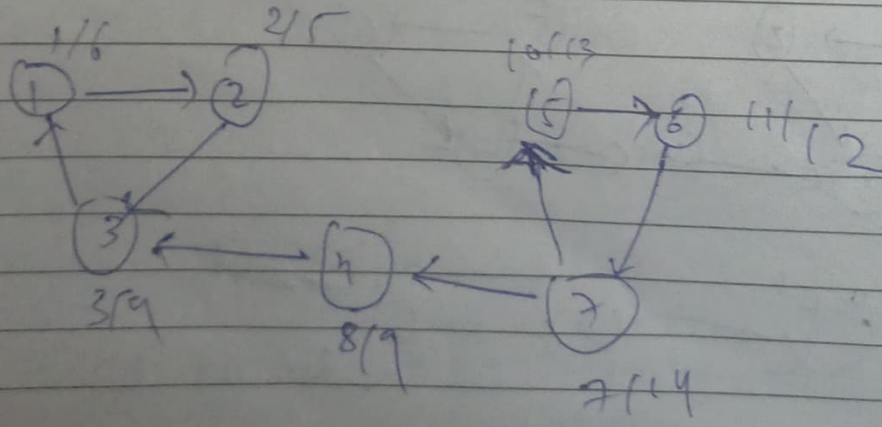


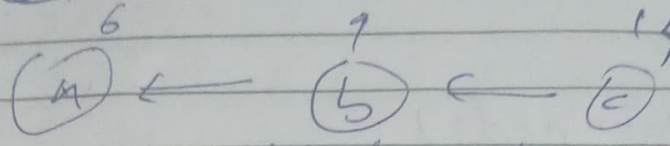
Transpose graph (reverse edges)



apply dfs on scc3 any node (5,6,7) it will stay in scc3 only then remove scc3 and go to scc2 (0 in degree in original graph after removal of scc3 or 0 out degree in transposed graph)
repeat till no element is left.

Run DFS on the graph and assign out time of each node, then sort the list by out times of nodes.





condensation graph.

Nodes by increasing out time.

node number.

3	2	1	4	5	7	6

```
#include <bits/stdc++.h>
#define REP(i,n) for(int i=1; i<=n; i++)
#define mod 1000000007
#define ff first.
#define ss second.
#define ii pair<int, int>
#define vi vector<int>
#define vii vector<ii>
#define lli long long int.

using namespace std;
vi ar[1001];
vi tr[1001];
vi order;
vi scc;
int vis[1001];
void dfs(int node)
{
  vis[node] = 1;
  for(int child : ar[node])
    if(vis[child] == 0)
      dfs(child);
```

```
    order.pb(node);
}
void dfs1(int node)
{
    vis[node] = 1;
    for (int child: tr[node])
        if (vis[child] == 0)
            dfs1(child);
    scc.pb(node);
}
int main()
{
int n, a, b, m, t;
cin>>t;
while(t--)
{
    cin>>n>>m;
REP(i, n) ar[i].clear(), tr[i].clear(), vis[i]=
order.clear();
REP(i, n)
{
cin>>a>>b;
ar[a].pb(a);
tr[b].pb(a);
}
REP(i,n)
if(vis[i]==0) dfs(i);
```

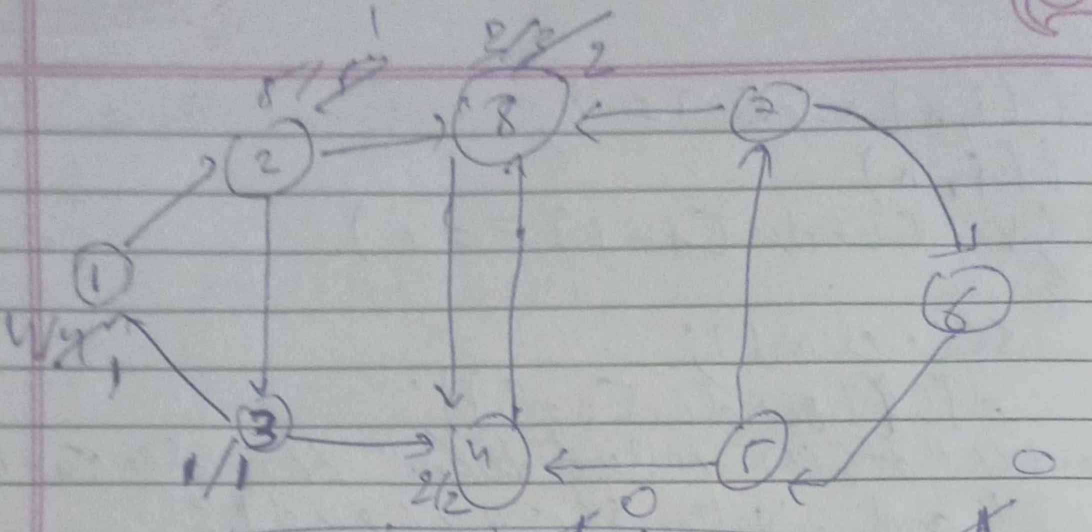
```
REP(i,n) vis[i] = 0;
REP(i,n)
 if(vis[corder[n-i]] == 0)
  {
    scc.clear();
    dfs(corder[n-i]);
cout<<"dfs() called from "<<corder[n-i]<<
    " and printing scc "<<endl;
for(int node : scc)
  {  cout<<node<<" ";
     cout<<endl;
```

Tarjan's Algo.

use low link value (low[]) and number of different low link value is number of scc in graph but can this method can fail on different order of dfs we have to apply extra restriction on it.

During execution of DFS and calculation of Low-link value, Tarjan's algo track of active nodes.

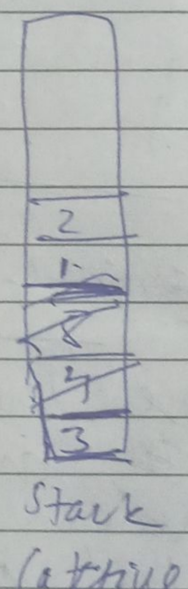
A node u can use node v to minimize its low-link if and only if v is in active node list.



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

on stack array

and so on




```
using namespace std;
vi g[1001];
bool vis[1001], onstack[1001];
int in[1001], low[1001];
stack <int> st;
int times = 1, scc = 0;
void dfs(int node)
{
  vis[node] = 1;
  in[node] = low[node]
   = times++;
  onstack[node] = true;
  st.push(node);
  for(int u : g[node])
  {
    if((vis[u] == true) && (onstack[u] == true))
      low[node] = min(low[node], in[u]);
```

```
else
if(vis[v]==false)
{
  dfs(v);
  if(onstack[v]==true)
    low[node]=min(low[node],low[v]);
}
}
if(in[node]==low[node])
{
  scc++;
  cout<<"scc#"<<scc<<endl;
  int v;
  while(1)
  {
    u=st.top()
    st.pop(), onstack[u]=false;
    cout<<u<<" ";
    if(u==node) break;
  }
  cout<<endl;
}
}
int main()
{ int n,m,a,b;
  cin>>n>>m;
    REP(i,m) cin>>a>>b, g[a].pn(b);
REP(i,n) vis[i]=onstack[i]=false;
REP(i,n)
  if(vis[i]==false) dfs(i); }
```