Graph.

Graph Data Structure is a Collection of nodes Connected by edges. It's used to represent relationship between different entities.

1. Graph data structure is a non-linear data structure.
2. It is used in fields such a network analysis recommendation system, and Compater network
3. In the field of sports datascience, graph data structure can be used to analize and understand the dynamics of team performance and player intractions on the field.

Components of Graph data Structure

1. Vertices
2. Edges

① Vertex -

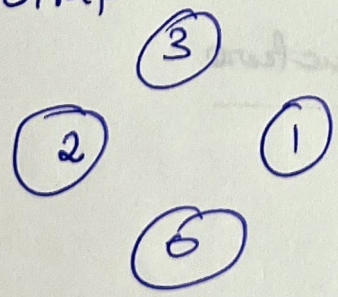
* Vertices are the fundamental units of the graph
* Vertices are also know as vertex or nodes
* Every node / vertex can be labeled or unlabelled

② Edges -

* Edges, are drawn or used to connect two nodes of the graph
* It can be ordered pair of nodes in a directed graph
* Edges are also known as arcs.

Types of Graph

① Null Graph



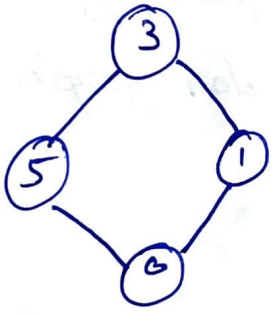
* A graph is known as null graph if there are no edges in the graph.

② Trivial Graph

Graph only have single vertex, it is also the smallest graph possible.

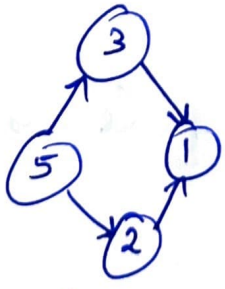
③ Undirected Graph.

Graph with edges dond have any directions.



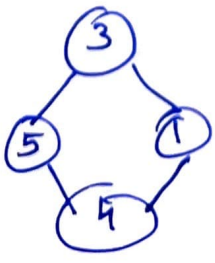
④ Directed Graph

A graph in which edge has a direction.



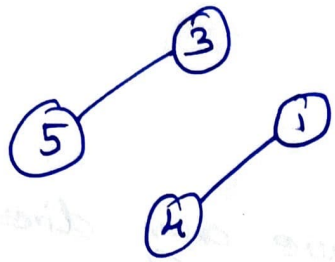
⑤ Connected graph

The graph in which from one node we can visit any other node in a graph is known as a Connected graph



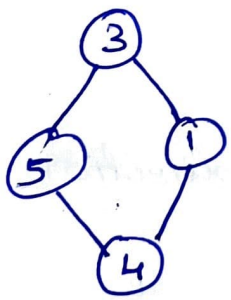
(6) Disconnected Graph:-

The graph in which one node is not reachable from a node is know as a disconnected graph.



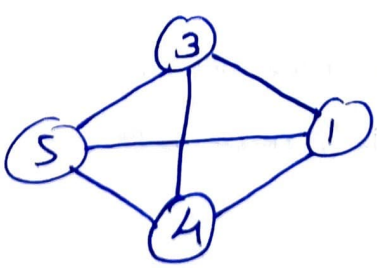
(7) Regular Graph:-

The graph in which the degree of every vertex is equal to k is called k regular graph.



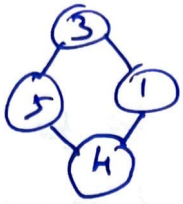
(8) Complete Graph

The graph is which each node is edge to each other node



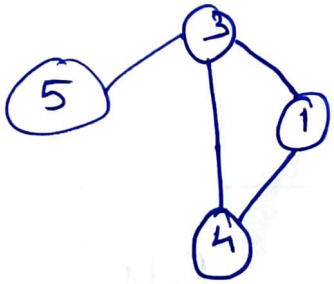
(9) Cycle graph

The graph in which the graph is a Cycle on itself, the minimum Value of degree of each Vertex is 2.



(10) Cyclic graph

A graph Containing at least one cycle is known as a Cyclic graph

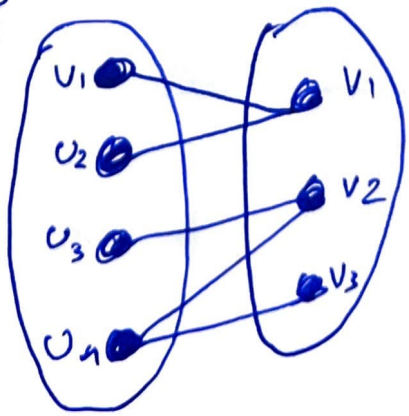


(11) Direct Acylic Graph: -

A directed graph that dose not Contain any Cycle

(12) Bipartite graph

A graph in which vertex Can be divided into two sets such that Vertex in each set dose not Contain any edge between them.



(13) weighted graph

* A graph in which the edges are already specified with suitable weight is known as a weighted graph

* weighted graph can be further classified as directed weighted graph and undirected graph.

Representation Of graph Data Structure :-

* Adjacency Matrix
* Adjacency List.

Dfs - Depth first Search. Adj List

step(1) - Convert edges to adjacency list.

step(2) - Create visited array to mark the visited nodes.

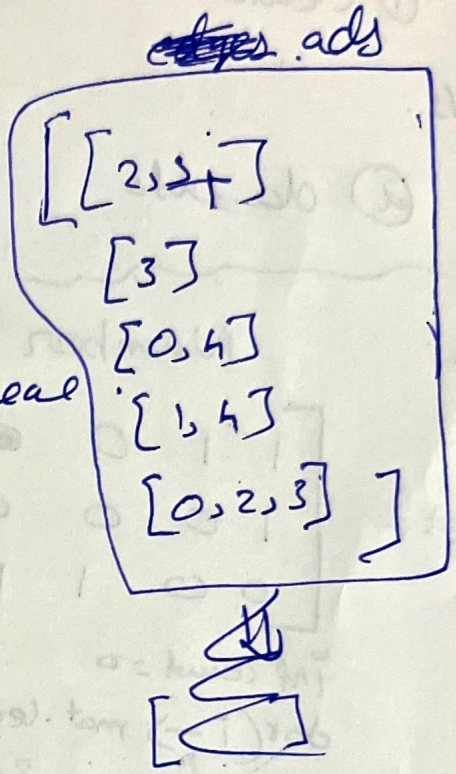
step(3) - Do dfs.

```
f.()
  v[curr] = 1;
 for (n -> adj(curr))
   {
     if(v[n] == 0)
      {
       dfs(n)
      }
   }
```

bfs

Adj List.

1. Convert edges to adj list.
2. Create visited array to mark visited nodes
3. do bfs.



bfs

- Create a queue
- add a first node to queue

```
Queue<> q;
q.offer(i);  // 0
v[c] = 1;
while (q.isEmpty)
  {
  int curr = q.poll();
    Print(curr);

  for (int n -> adj[curr])
     { if [visited[n] == 0]
       {
         qu.offer(n);
          visite[n] = 1;
       }
     }
  }
```

Adj matrix:

① Dfs

① Create a visited matrix to store Visited cells.

② do dfs

Ex: Number of Islands.

mat = $\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$

```
int count = 0
for(i -> mat.len)
  { for(j -> mat[i].len)
      { if(v[i][j] == 0){
        dfs(mat, v, i, j)
      } count ++
      }
  }
dfs(adj, v, row, col) {
  if(r < 0 || c < 0 || r >= adj.length || c >= adj[0].len) return;
  if(adj[r][c] == 0 || visited[r][c] == 1) return;
  visited[r][c] = 1;
  dfs(adj, v, r-1, c);
  dfs(adj, v, r, c+1);
  dfs(adj, v, r+1, c);
  dfs(adj, v, r, c-1);
}
```

Adj matrix

② Bfs

① Create a Visited matrix to mark the visited cell

② Do bfs

Exp: rotten tomatos:-

$$arr[][] = \begin{bmatrix} 2 & 1 & 0 & 2 & 1 \\ 1 & 0 & 1 & 2 & 1 \\ 1 & 0 & 0 & 2 & 1 \end{bmatrix}$$

① Create a cell datatype

```
Public class cell {
    int row;
    int col;
    cell (int row, int col){
        this.row = row;
        this.col = col;
    }
}
```

Create a queue.

② Put all ②'s in queue.

```
for (i -> arr.len)
{
    for (j -> arr.len)
    {
        if (arr[i][j] == 2)
        {
            queue.offer (new cell (i,j));
        }
    }
}
```

③ do bfs

```
count = 0;
while (queue.isEmpty()){
  n = queue.size()
  for(int i → n)
    {
      cell curr = queue.poll();
      visited [curr.row][curr.col] = 1;
// up
if (curr.row - 1 >= 0 && arr[r-1][c] == 1 && V[r-1][c])
  {
    visited[row][c] = 1;
    queue.offer(new cell (row col));
  }
//right
if ( col < arr[0].length && arr[r][c+1] == 1 && V[r-1][c])
  {
    visited [r][col] = 1;
    queue.offer (new cell [r, col]);
  }
// down
if (row+1 < arr.length && arr[r+1][c] == 1 && V[r+1][c])
  {
    visited [r+1][c] = 1;
    queue.offer (new cell (r+1, c));
  }
}
```

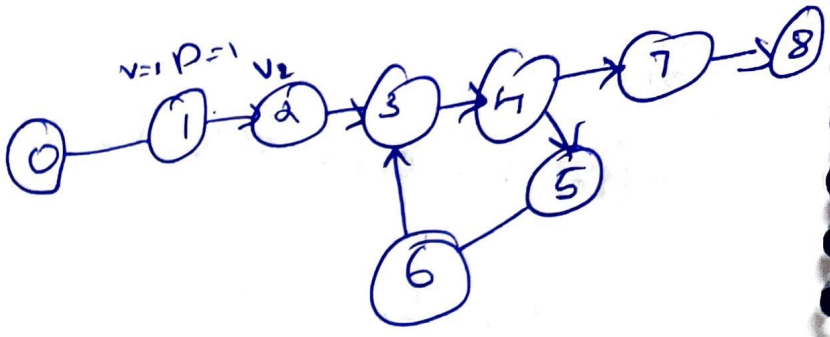
//left.

```
if (col-1 >=0 && arr[r][c-1]==1 && v[r][c-1]){
    {visited[r][c-1]=0;
     quen.offer(new cell(r,c-1));
    }
}
cout++;
}
```

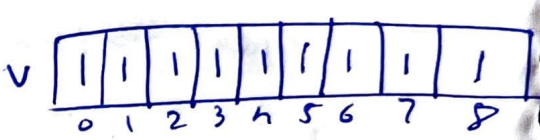
Cycle Detection Directed (DFS)

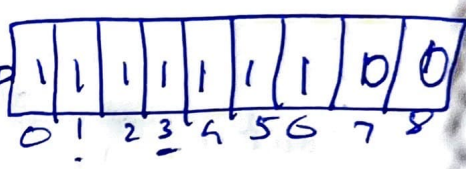
1. Convert edges to graph
2. Create a visited array and path visited array
3. do dfs.

↓ next page




```
dfs (Curr)
{
  Visited [curr] = 1;
PathVisited [Curr] = 1;
```






```
  for (int n → adj [curr]) {
    if (visited [n] == 0) {
      if (dfs())
        { return true
        }
    }
    else if (PathVisited [n] == 1) {
      return true;
    }
  }
  PathVisted [curr] = 0;
  return false;
}
```

Detect Cycle in undirected Graph. (DFS)

① Convert edges to adj list.

② Create a visited array to mark visited nodes.

③ do dfs. | intution (Paret != neighbour) -Cycle det

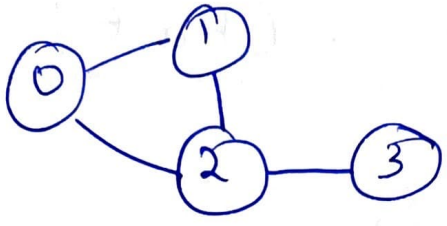
```
dfs(curr, p)
{ visited[curr] = 1;
  for (int n -> adj[curr])
  { if (visited[n] == 0)
    {
      if (dfs(ne, cu)) return true;
    }
    else if (Parent != neighbour){
      return true
    }
  }
}
return false;
}.
```

Detect a Cycle in undirected (BFS).

① Convert edges to adj list.

② Create a Visited array to ~~store~~ mark visited nodes

③ do Bfs.

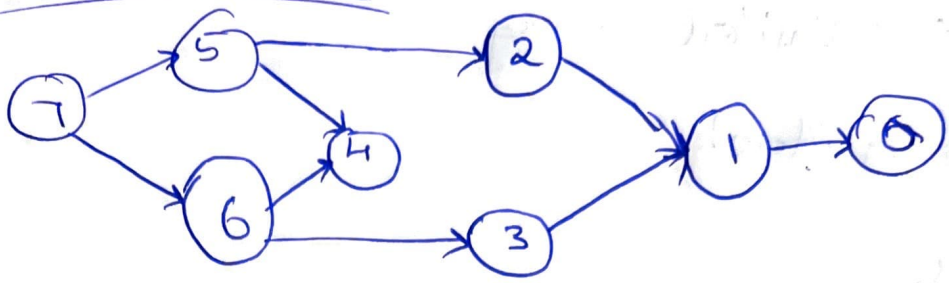


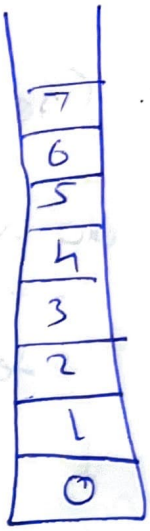
[[1,2]
[2]
[3]
[ ]
]


```
Queue <> q;
queue.offer(-1, start) → (0, -1)
while (!q.isEmpty()) {
    Visited [child] = true;
    for (int n → adj [curr])
    {
        if (visited [n] == 0) {
            que.offer (c, neig)
        }
    else if (parent != neighbours)
    {
    return true;
    }
    }
    return false;
}
```

1,0
p=0
if(n!=p)
{ retu
true
}

Topological Sort. - only work DAG (Directed Acyclic Graph)

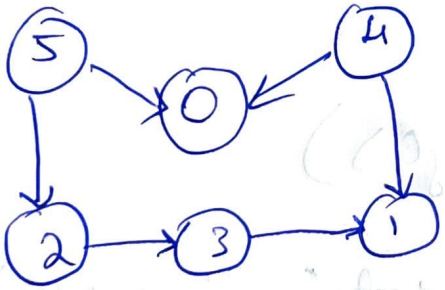




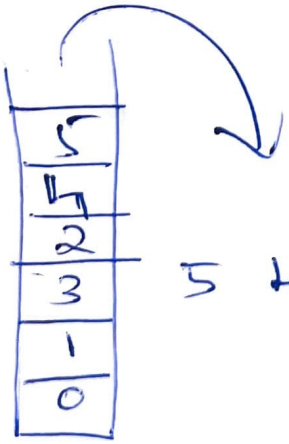
BFS

7 6 5 4 3 2 1 0

Topological sort DFS (using stack



Linear ordering of vertices such that if there is an edge between u and v u appears before v in the ordering.



5 4 2 3 1 0.

steps

① Create a visited array

② Crete a stack

③ do dfs.

```
for(int i = 0 → adj)
  {
    if (visited [i] == 0)
      {
        dfs (adj, visited, stack, i)
      }
  }
while(!stack.isEmpty())
  {
    System.out.print(stack.pop());
  }
```

```
dfs ( )
  { visited [curr] = 1;
    for(int n → adj[curr]){
      if (visited [n] == 0)
        {
          dfs( )
        }
      stack.push(curr);
  }
```

Topological sort (BFS) Kahn's Algorithm:-

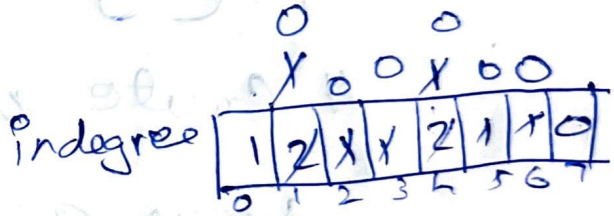
(1) Convert edges to adj list.

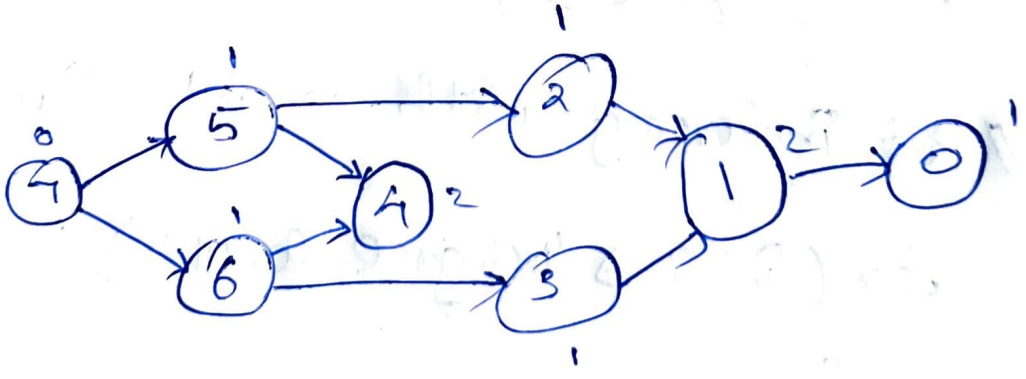
(2) calculate indegree.

(3) put node that has indegree 0 in to the queue

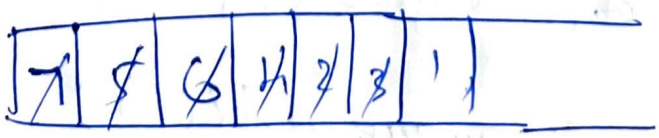
(4) Do Bfs

(5) while visiting neighbour node decrease current node ~~degree~~ indegree by one.





queue



result = 7 5 6 4 2 3 1

→ node.

```
// Calculat indegree.

int [] indegree = new int [adj.length];
    for(i -> adj.length)
    {  for (n -> adj[i])
            {  indegree[n]++;
            }
    }

bfs()
// Create queue
Queue q<>;
// Put indereeg with 0 into Queue.
    for (int i -> indegree.length)
    {  if (indegree[i] == 0)
            {  queue.add(i);
            }
    }
```

//

bfs

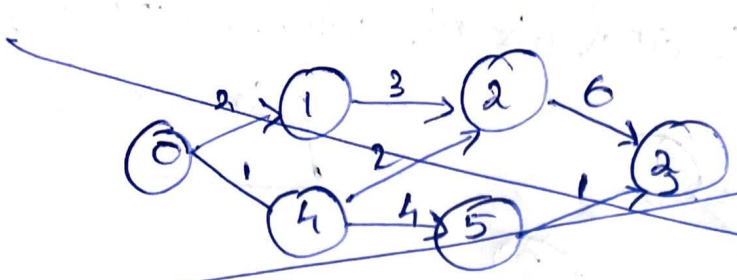
```
while (!q.isempty())
{
    int curr = queue.poll();
    res.add(curr);
    for (int n : adj[curr])
    {
        indegree[n]--;
        if (indegree[n] == 0)
        {
            q.offer(n);
        }
    }
}
```

Detect a Cycle in directed Graph.. (Topological)

1. Reverse the graph edges.
2. Calculate indegree.
3. Create queue.
4. put the node into queue with indegree 0
5. do Bfs - (Kahn's algorithm).

Shortest Path DAG:- (Topological sort)

1. Only works in Directed Acyclic graph.
2. first do dfs and get the result in stack.
3. Create a dist array
4. fill dist array with infinity
5. mark last node to 0.
6. ~~Do Dfs~~ Take a node out of the stack and check the neighbour nodes and update the dist array.

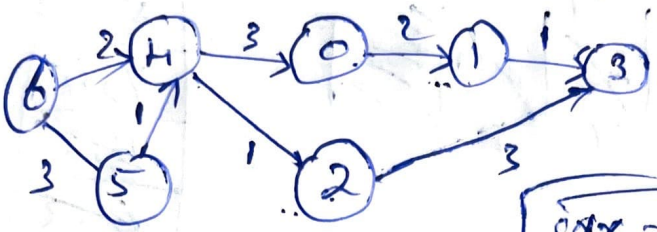


adjList

0 → {1, 2}

1 → {3, 1}

2 → {3, 3}



adjList

0 → {1, 2}

1 → {3, 1}

2 → {3, 3}

3 →

4 → {0, 3} {2, 1}

5 → {4, 1}

6 → {4, 2} {5, 3}

src = 6

node = 6, 0

+3 +2

node 4

node 5

+1

node 4

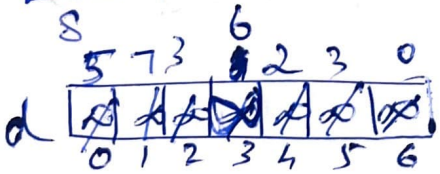
2

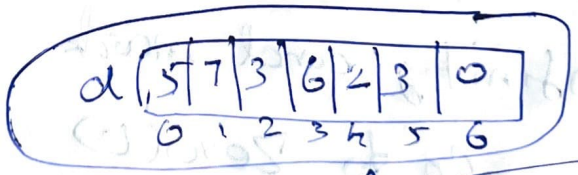
+3 +1

0 2

2 → 3

8





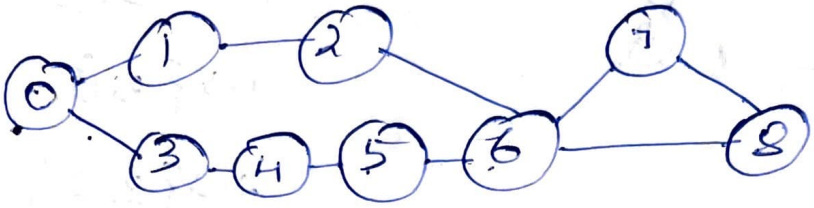
step(1). → Do toposort on a graph

mark src node to zero

step(2) → Take a nodes out of a stack and check the neighbours nods and update the dist array.

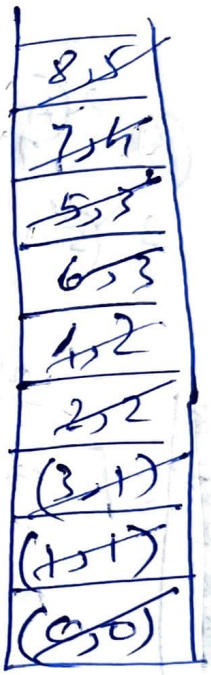
Shortest path in undirected Graph with unite weight

Src = 0



dist

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



Pair (node, wt)

(1) Convert edges to adj list

(2) Create a dist array and fill infinity and mark source node to zero(0)

(2) Do bfs.

important (2) (1)

* if the weigh is unite weight

Use Queue

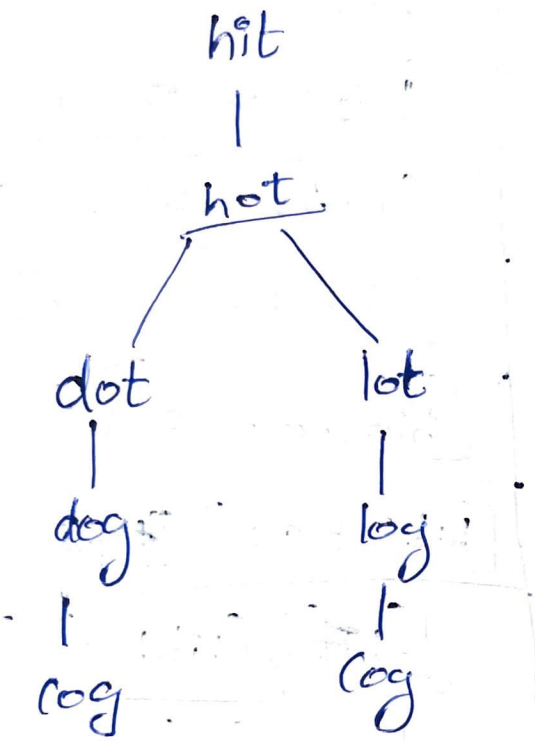
* if the weight is varying weight

Use priorityQueue (Dijkstra's algorithm

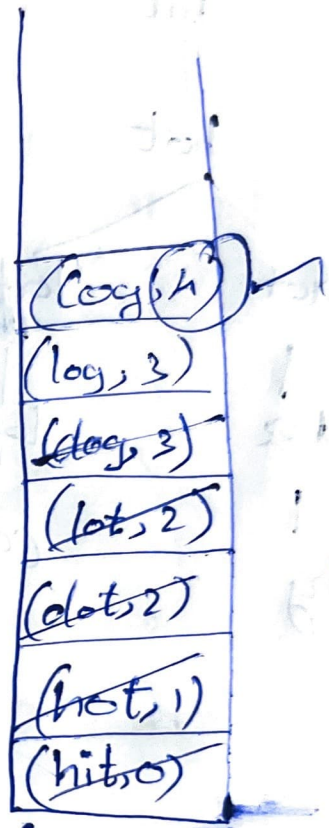
word Ladder - I

begin word = "hit", endword = "cog"

wordlist = ["hot", "dot", "dog", "lot", "log", "cog"]



(4)



(word, dist)

(1) Create a Set and put all word into the Set.

(2) Create a Queue and put begin word into queue (hit, 0) and length 0.

(3) Do bfs.

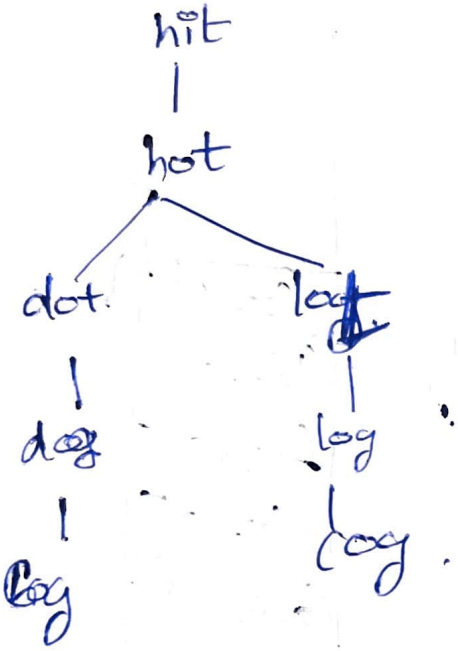
Take each word and put all combinations and check it is in set, if present put int into queue and delete word in set.

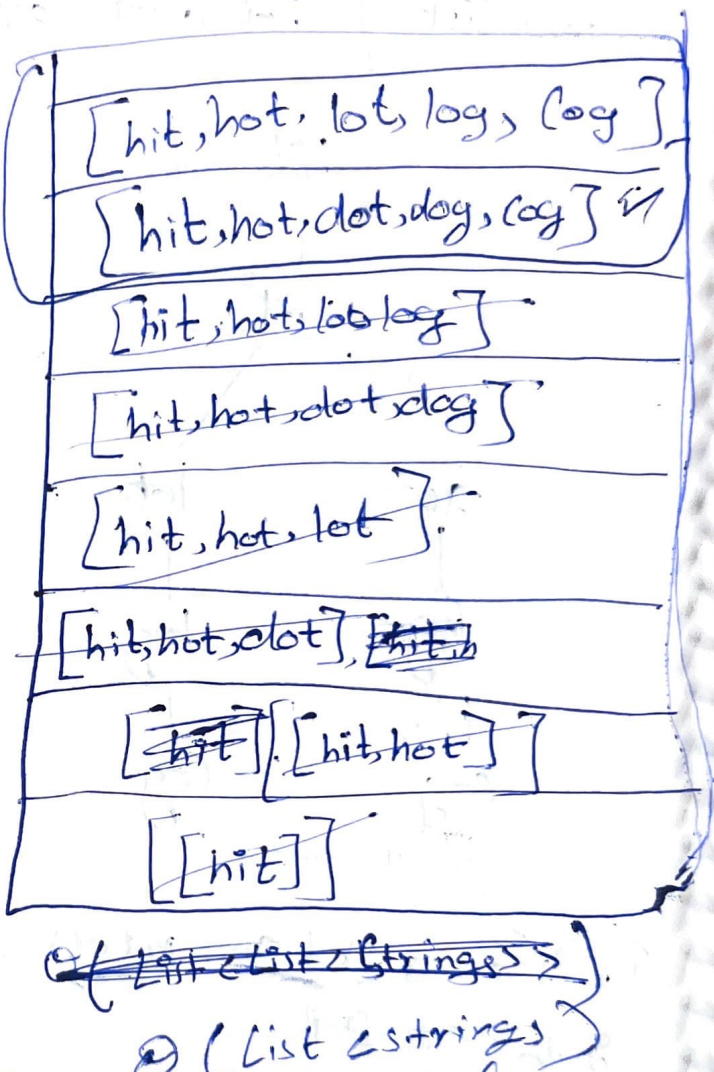
Do this until you get the end word.

word ladder 2

beginword = "hit", endword = "cog"

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





① Create a set and put all words into a set

② Creat a Queue and put begin word in List and put the list to Queue

③ do bfs

take the last word in List and put all combination if its valid add to Level Visited node and put the word into list and put the list back to Queue. do until getting result.

Shortest path Algoriths:

* Dijkstra Algorithm. - (shortest path in weighted graph) greedy
* Bellman - Ford Algorithm
* Floyd - warshall.

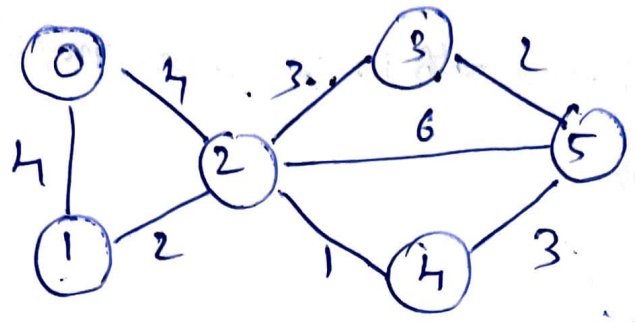
Dijkstra's Algorithm: (Priority Queue or set)

* Dijkstra's Agorithm is a graph search algorithm it is used to find the shortest path between nodes in a weighted graph
* It was developed by Edsger W. Dijkstra in 1956

Purpose:

* Find the shortest path from a starting node to all other node in the graph
* work only in non-negative edge weights (Positive)
* It is a greedy algorithm - It always selects the smallest known distance that hasn't been visited yet.

* Output = The shortest path from the source to each node



Adj list.

0 → {1,4} {2,4}

1 → {0,4} {2,2}

2 → {0,4} {1,2} {3,3} {4,1} {5,6}

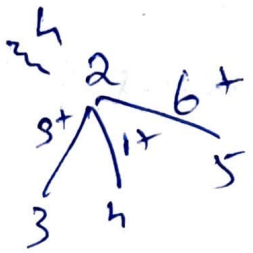
3 → {2,3} {5,2}

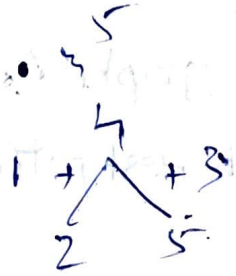
4 → {2,1} {5,5}

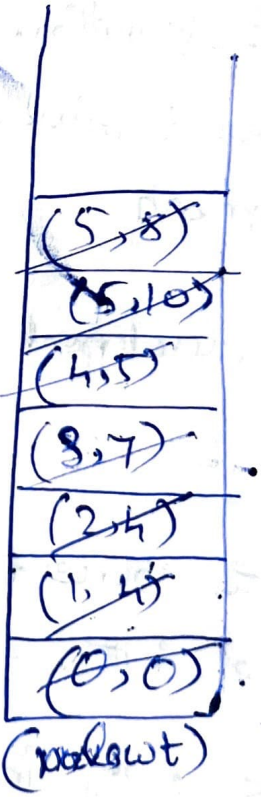
5 → {2,6} {3,2} {4,3}

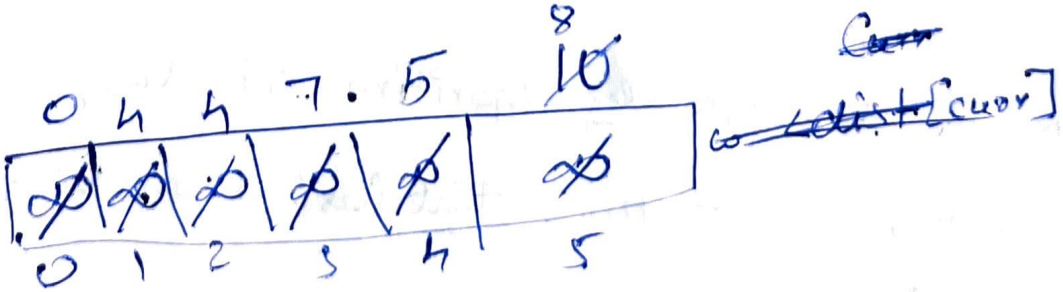
Ans 2

```
if (Current+Adjwt < dist[adjnode])
{
dist[adjnode] = Current+Adjwt;
pq.add()
}
```









step (1) - Convert edges to adjlist

step (2) - Create a priorityQueue (min heap) and put a source to pq

step (3) - Create a dist array to store the result

step (4) - fill dist array with infinity and mark src to 0

step (5) - do bfs

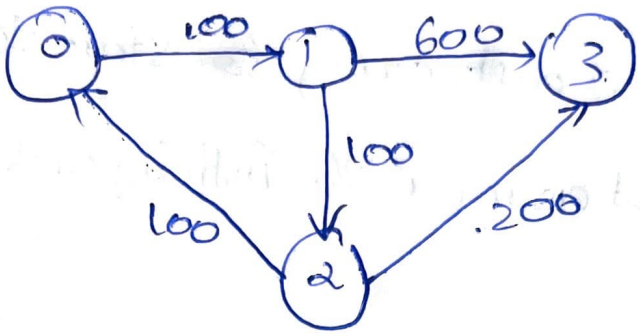
bfs

```
while (!pq.isEmpty())
{ Pair curr = pq.poll();
  int currNode = curr.node;
  int currwt = curr.wt;
  for (int[] node : adj.get(currNode)) {
    int adjnode = node[0];
    int adjwt = node[1];

    if (currwt+adjwt < dist[adjnode])
    {
      dist[adjnode] = currwt + adjwt;
      pq.offer(new pair(adjnode, dist[adj]))
    }
  }
}
```

Cheapest flight

s = 0
k = 2
d = 3



(X) here our first priority is stops.

0	100	200	700
0	1	2	3

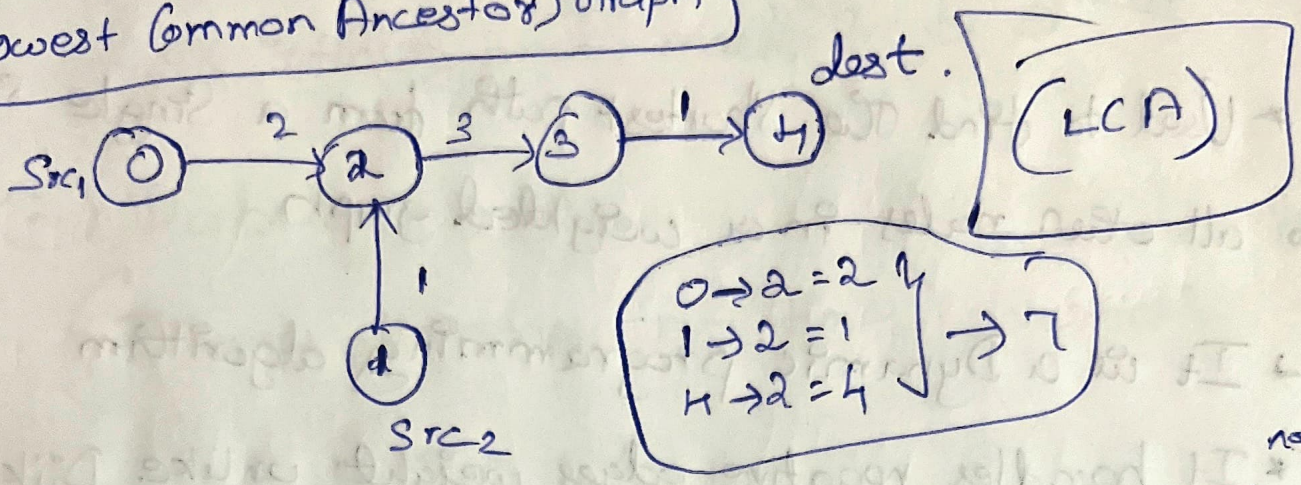
(2, 3, 400)

2, 2, 200
2, 3, 700
(1, 1, 100)
(0, 0, 0)

```
if (currdist+adjdist < dist[adjNode] && stop <= k)
{
    dist[adjNode] = currdist+adjdist;   (stops, node, dist)
    queue.offer(newpair(adjnode, currdis(,stop+1)
}
```

Minimum weighted Subgraph with the required path.

(Lowest Common Ancestor) Graph



node

(1) Convert edges to adj list - to calculate src to all

(2) Convert edges to reverse adj list - to calculate dest to all other nodes.

(3) Do dijkstra and get the shortest path

Src = 0 dist 1 | 0 | ∞ | 2 | 5 | 6 |

Src = 1 dist 2 | ∞ | 0 | 1 | 4 | 5 |

dest = 4 dist 3 | 7 | 8 | 4 | 1 | 0 |

LCA = 2

2+1+4 = 7 | 5+4+1 = 10 | 6+5+0 = 11

```
res = Inf;
for(int i → n)
{
  if(dist1[i] == Inf || dist2[i] == Inf || dist3[i] == Inf) continue;
  res = Math.min(dist1[i]+dist2[i]+dist3[i], res);
}
return res;
```

Bellman ford Algorithm:

* Used to find the shortest path from a single source to all other nodes in a weighted graph

→ It is a Dynamic programming algorithm

* It handles negative edges weight unlike Dijkstra

* It can also detect negative weight cycle in the graph

* Suitable when a Graph contains negative weight and only need path from single source

Algo:

(1) The algorithm works by "relaxing all edges $n-1$ times", where n is number of vertices

(2) If any Distance can still relaxed after $n-1$ iteration, a negative cycle exists.

(3) Time Complexity: $O(V \times E)$

V - Vertices

E - edges.

Steps ① - Create a dist array to store distance fill infinity and mark src to 0

② Do bellman ford. (Traverses all edges n-1 times) n-number of vertices.

```
for(int i → n-1)
{
    for(int j → edges.length)
    {
        int u = edges[j][0]
        int v = edges[j][1]
        int wt = edges[j][2]
        if(dist[u]+wt < dist[v])
        {
            dist[v] = dist[u]+wt;
        }
    }
}
```

③ Detect Negative Cycle.

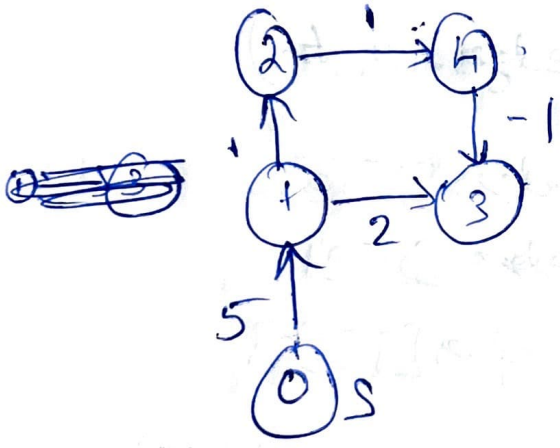
```
for(int i → edges.length)
{
    u = ed[j](0)
    v = -''
    w = ''
    {
        if(dist[u] != (int)(ieq) && dist[u]+wt < dist[v])
        {
            return true
        }
```

if result improves negative cycle else exist.

edges = [u v wt
[1, 3, 2]
[4, 3, -1]
[2, 4, 1]
[1, 2, 1]
[0, 1, 5]]

∞+2 0+5
5+2
∞

```
if dist[u]+wt < dist[v]{
    dist[v] = dist[u]+wt;
}
```



dist

0	5	6	7	7
0	1	2	3	4

Floydwarshall Algorithm:-

* It is used to find the shortest path between all pairs of Vertices in a weighted graph

* It works in both directed and undirected.

* It is a Dynamic programming algorithm

* It uses 2D matrix to store the shortest distance between each pairs of nodes

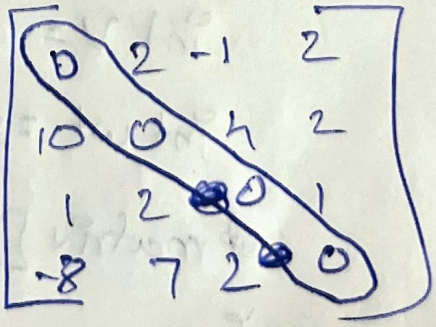
* It works with negative edges as long as there is no negative cycle.

*

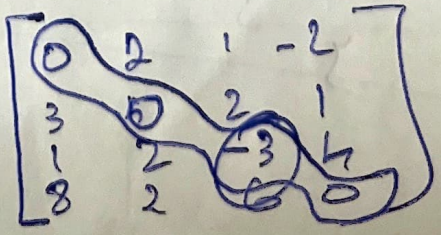
It can detect the size of negative cycle

if (dist[i][i] < 0), return true.

Neg Cycle exist



no Cycl



Cycle exist.

Time Complixity $O(V^3)$ where V is the number of vertices

Space Complixity $O(V^2)$ due to distance matrix

```
dist[i][j] = math.min(dist[i][j], dist[i][k] + dist[k][j])
```

steps (1) - Convert the edges to adj matrix

fill infinity except diagonal matrix

step (2) - Set edges to matrix

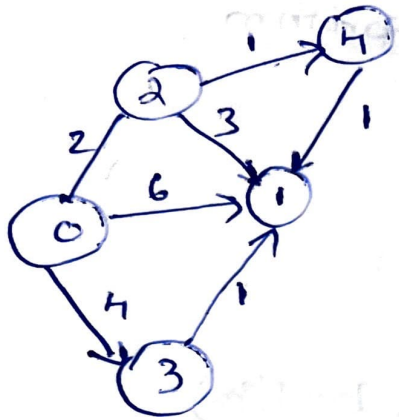
```
for(i -> 0 -> edges.length)
  {
  for(j -> edges[i].length)
    {
      int u = edges[i][0];
      int v = edges[i][1];
      int wt = edges[i][2];
      matrix[u][v] = wt;
    }
}
```

Step (3) - Do Floyd Warshall algorithm.

```
for ( via → n)
    {
      for (i → matrix.length)
        {
          for (j → matrix[i].length)
            {
dist[i][j] = Math.min (dist[i][j], dist[i][via]+dist[via][j])
            }
        }
    }
```

Step (4) Detect Negative Cycle.

```
for (i → matrix.length)
  { if (matrix[i][i] < 0)
      {
        return true;
      }
  }
```



* (Multisource Shortest path Algorithm)
* help you detect negative cycles as well

Note: Go via every vertex/node.

$d[0][1] = 0 \to 1$: 6

Via ② $d[0][1] = d[0][2] + d[2][1] \to 5$

$0 \to 2 \to 1$

Via ③ $d[0][1] = d[0][3] + d[3][1] = 5$

Via ④ $d[0][1] = d[0][4] + d[4][1] = 4$

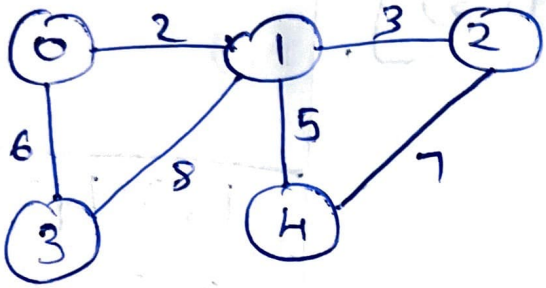
do the same for all the edges

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

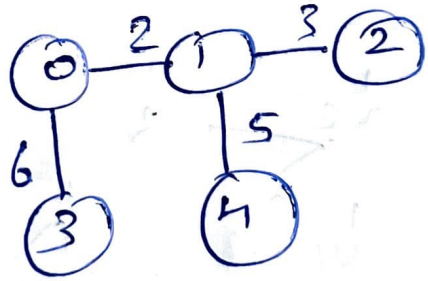
Prims Algorithm.

Minimum Spanning Tree

- Prims Algorithm
- Kruskal's Algorithm

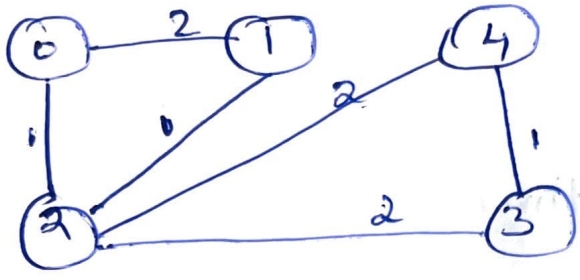


— MST.



sum = 16

Prims Algorithm

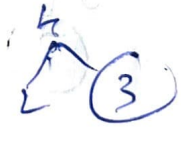


Sum = ~~0 1 2 4~~ 5

MST ⟹ [(0, 2), (2, 1), (4, 2), (4, 3)].

Visited

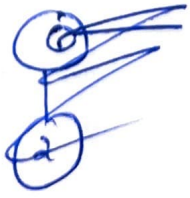
1	1	1	1	1
0	1	2	3	4

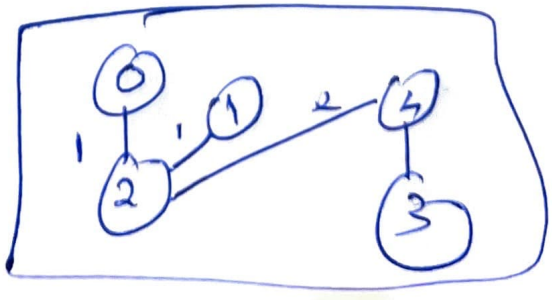


(wt, node, parent)
~~1, 3, 4~~
2, 3, 2
~~2, 4, 2~~
~~1, 1, 2~~
~~1, 2, 0~~
~~2, 1, 0~~
~~(0, 0, -1)~~

minheap

if (parent == -1)
dont add edge to MST or ~~Sum~~ wt to Sum





Step (1) .. Convert edges to adjlist

Step (2) - Create a visited array.

step (3) - Creat mst array to store list of mst edges

step (4) - Create a sum variable to Calculate the weight

step (5) . Create a priority Queue (min-heap)

step (6) ~

(1) start from any node

(2) Maintain a minheap (priority Queue) to Pick the edge with the smallest weight that Connects the new vertex.

(3) Grows the tree mst one vertex at time

(4) Time Complixity $O(E \log V)$

It is a greedy Algorithms

Convert edges to adj

```
List<int[]> mst = new ArrayList<>();
int visited = new int[n];
int sum = 0;

PQ<>
PQ.offer(n, wt, p)
(0, 0, -1)
while (!PQ.isEmpty())
{
    if (visited[currnod] == 1) continue
    visite[currentnod] = 1
    if (currparent != -1) {
        mst.add(new int[] {currnode, currpar
        sum = sum + currwt;
    }
    for (int[] node : adj.get(currnode)) {
        int adjn = nod[0];
        int adjw = nod[0];
        { if (visited[adjnode] == 0) {
            PQ.offer(new pair(adjnod, cu, adj
        }
```

Disjoint Set (Undirected graph)

only works in undirected graph

(find Union)

Solve graph Probs in O(1)

not actually for intution

~~Properties~~

methods.

→ find parent ()

→ Union () < by rank / by size

A disjoint set Union (DSU) or Union find datastructure keeps track of a partition of elements into disjoint (non-overlaping subset.

It supports two main operations

find () → find the leader or root of a set containing x

Union (x, y) → Merge the set Containing x and y

Efficient implementation uses

* Path Compression (in find)
* Union by Rank / Size (in union).

Time Complexity: $O(\alpha(n))$ per operation, where $\alpha(n)$ is the inverse Ackermann function - (extremly fast).

Use Cases

1. Kruskal's Algorithm (minimum Spanning Tree)
2. Cycle Detection in Undirected Graph
   If you try to union two vertices already connected (same root), a cycle exist.
3. Connected Components
   * "How many distinct components are there?"

Just Count how many unique parent exists after Unioing Connected nodes.

(4). Dynamic Connectivity.

* you can dynamically merge Component and check weather two nods are in the same Component - useful in realtime graph processing.

(5). Network Connectivity

used in problems like

→ Are all computers in network Connected.

→ After adding/removing Cables, is the network still Connected.

(6) Percolation Theory / Maze Generation

→ used in Simulations to track Connected open space in grid or graph.

④ Game Logic

Used in :

→ Region merging

→ Area Claiming

→ Team or alliance Tracking.

U V

(1, 2)

(2, 3)

(4, 5)

(6, 7)

(5, 6)

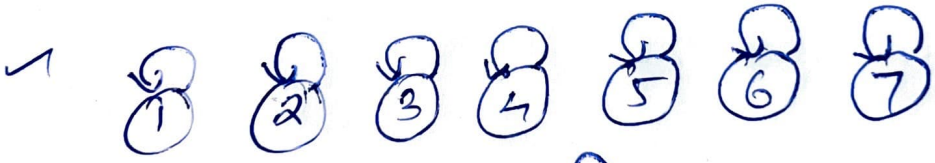
(3, 7)

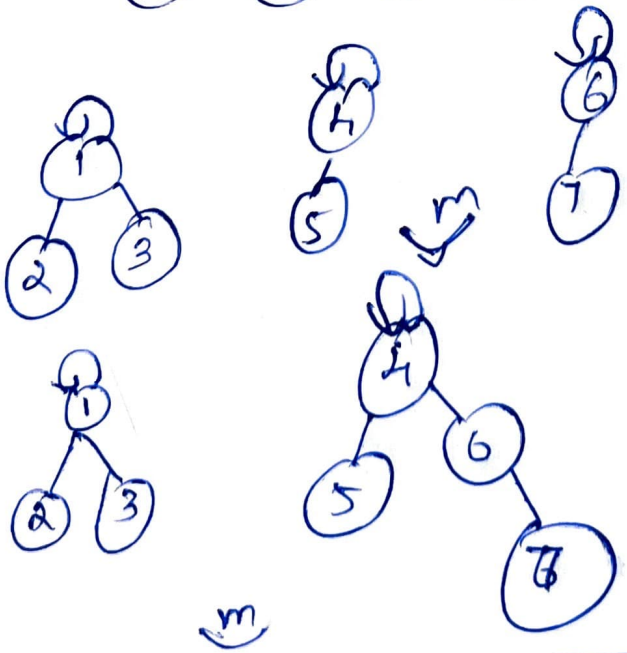
rank

1	2	3	4	5	6	7
∅	0	0	∅	0	∅	0
1			2		1	

Parent

1	2	3	4	5	6	7
1	~~1~~	~~1~~	4	~~4~~	∅	~~6~~ ∅
					4	4





Union (U, V)

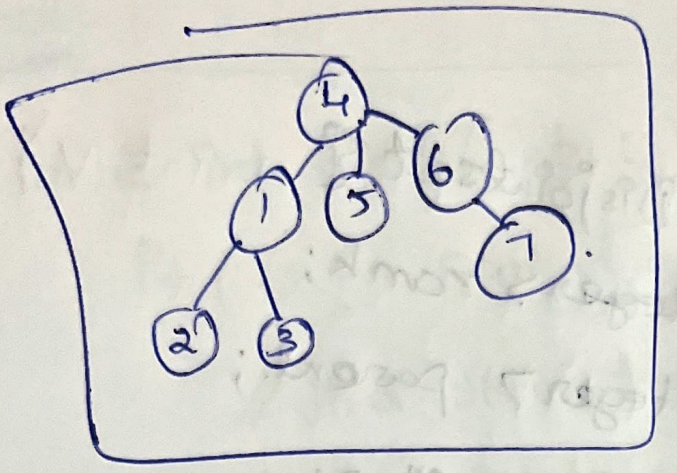
① find the Ultimate Parent of U & V

PU, PV

② find rank PU & PV

③ Connect smaller rank to larger rank

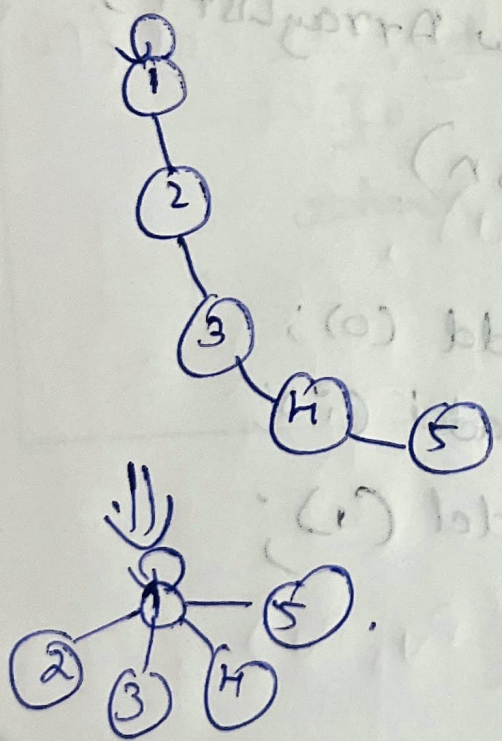
while merging parent with same rank increase rank



find ()

Path Compression

find (5).



Path Compression.

```
function (u)
{ if(u != parent.get(u))
  {
    // find ultimate parent
int root = find (parent.get(u))
    // path compression
  parent.set (u, root)
  return root;
  }
return parent.get (u);
}
```

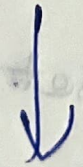
Disjoinset

```
Public static class Disjointset {
    List<Integer> rank;
    List<Integer> parent;
    List<Integer> size;

    Djointset (int n) {
        this.rank = new ArrayList<>();
        this.parent = new ArrayList();
        this.size = new ArrayList();
        for (int i → n)
        {
            rank.add (0);
            Parent.add (i);
            size.add (1);
        }
    }
```

// find

```
Public int find (int u){
    if (u != parent.get(u)) {
        // find ultimate parent
        int root = find(parent.get(u));
        // path compression
        parent.set(u, root);
        // return root
        return root;
    }
    return parent.get(u);
}
```



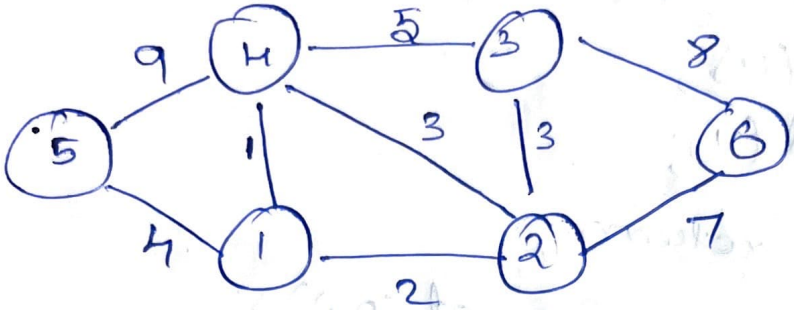
Union by rank

```
Pubic Void UnionbyRank (int u, int v) {
    int pu = find (u);
    int pv = find (v);
    if (pu == pv) return ;
    if (rank.get (pu) < rank.get (pv)) {
      Parent.set (pu, pv);
    }
    else if (rank.get (pv) < rank.get (pu)) {
      Parent.set (pv, pu);
    }
    else {
        Parent.set (pv, pu);
        rank.set (pu, rank.get (pu) + 1);
    }
}
```

Union by Size.

```
Public Void UnionbySize (int u, int v) {
    int pu = find(u);
    int pv = find(v);
    if (pu == pv) return;
    if (size.get(pu) < size.get(pv)) {
        parent.set(pu, pv);
        size.set(pv, size.get(pv) + size.get(pu));
    }
    else if (size.get(pv) < size.get(pu)) {
        {
        parent.set(pv, pu);
        size.set(pu, size.get(pu) + size.get(pv));
    }
    else {
        parent.set(pv, pu);
        size.set(pu, size.get(pu) + size.get(pv));
    }
}
```

Kruskal's Algorithm → find MST

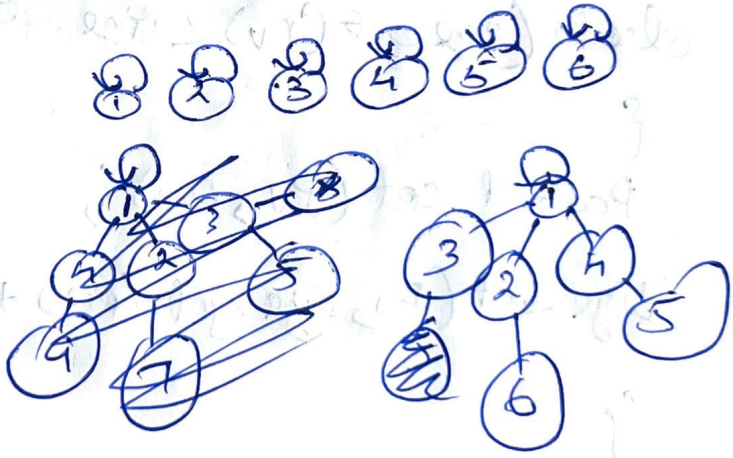


Step (1) - Sort all the edges according to weight.

Step (2) - Do union for all edges.

(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 ✓



```
Public static Void kruskals(int[][] edges, int n)
{
    Disjointset ds = new DisjointSet(n);
    int sum = 0;
    List<int[]> mst = new ArrayList<>();
    Array.sort(edges, (a,b) -> a[2]-b[2]);
    for (int i=0; i<edges.length; i++)
    {
        int u = edges[i][0];
        int v = edges[i][1];
        int wt = edges[i][2];
        if(ds.find(u) != ds.find(v)) {
            ds.union(u,v);
            sum = sum + wt;
            mst.add(new int[][]{u,v});
        }
    }
    System.out.print(sum);
}
}
```

Merge Accounts:-

[["John","Johnsmith@gmail.com","John_newyork@gmail.com"],
["John","Johnsmith@gmail.com","Johnoo@gmail.com"],
["Mary","mary@gmail.com"],["Jhon","Johnnybravo@gmail.com"]]

↓

(1) create a map <String, Integer>

Eg: Johnsmith@gmail.com = 0
John_newyork@gmail.com = 0
Johnsmith@gmail.com = 1
Johnoo@gmail.com = 1
mary@gmail.com = 2
johnnybravo@gmail.com = 3.

↓

(2) Unionmap map<Integer, List<Strings>>

0 = [Johnsmith@gmail.com, John_newyork@gmail.com
Johnoo@gmail.com]

2 ⇒ marry@gmail.com

3 ⇒ Johnngbrave@gmail.com

4-1-1-1

Number of Islands - II

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

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

3: 1+1

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

```
get node (r, c)
{
    return i*n+j;
}
```

Making Large Island :-

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

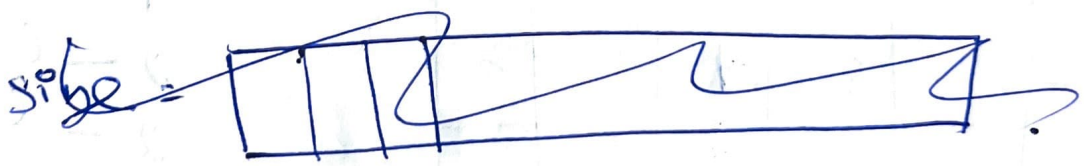
(0,0) → 10

$i * n + j$ (Col)

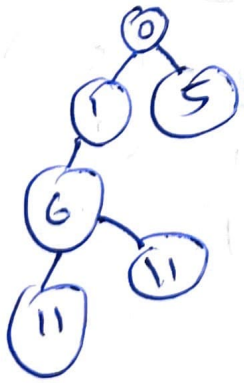
n = grid[0].length

	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
4	20	21	22	23	24
5	25	26	27	28	29

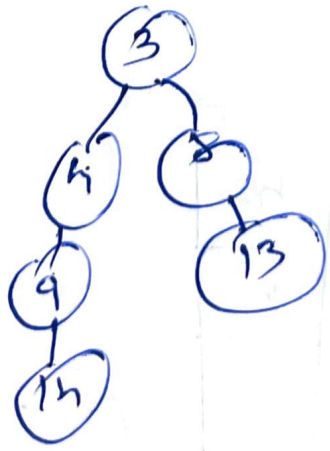
(1) first traverse the grid, and connect the components.



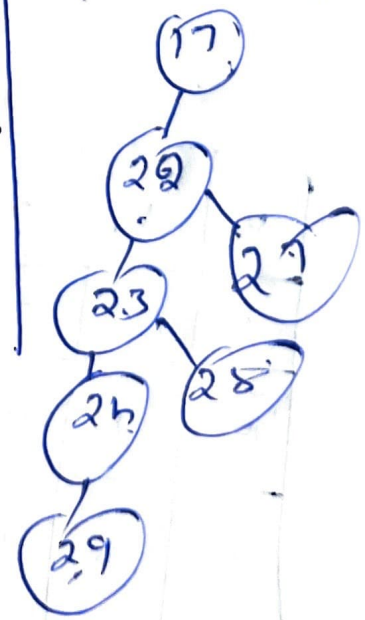
Size = 6



Size = 6



Size = 7



(2) do a traversal of grid[row][col] == 0

use Set to avoid duplication of adding size

for example

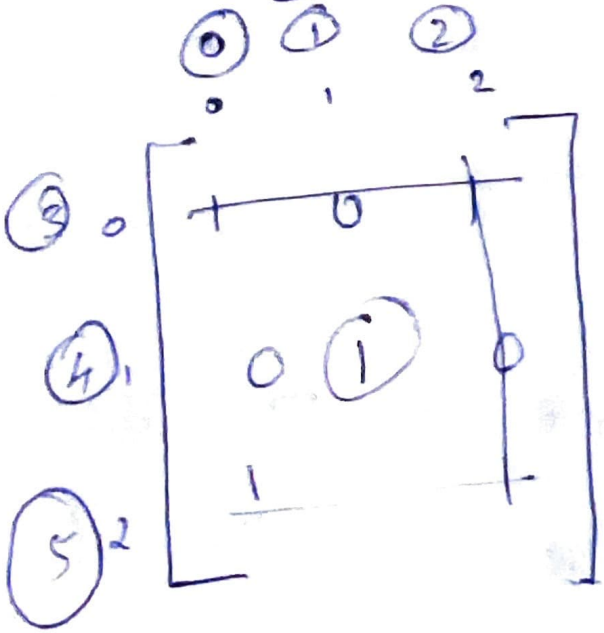
$6 - (12) - 6 = 19$

---

(1) most Stones removed with Same row or column.

Input =

Stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]



r = 0
1

col = row + n

n = naber of rows

formula

Total number of stones -

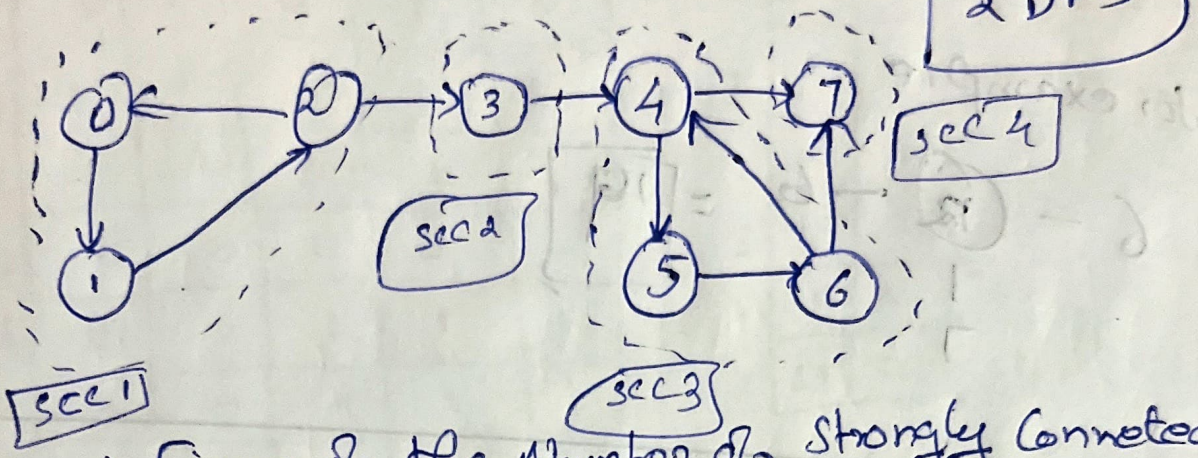
number of Components

Strongly Connected Connected Components:-

Kosaraju's Algorithm

~~2 DF~~

2 DFS



→ Figure of the Number of Strongly Connected Components.

→ Print the Strongly Connected Components.

Strongly Connected Components are only Valid for Directed graph

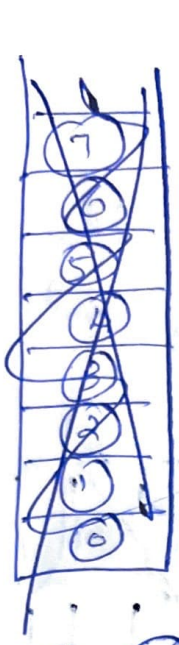
Implementation steps -

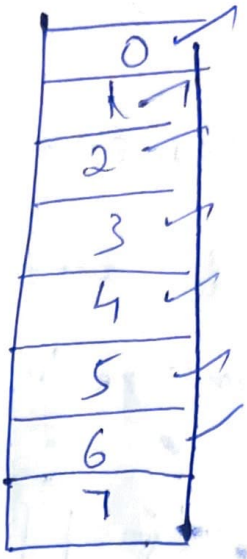
step ①

~~Do~~ Do dst and get the priority

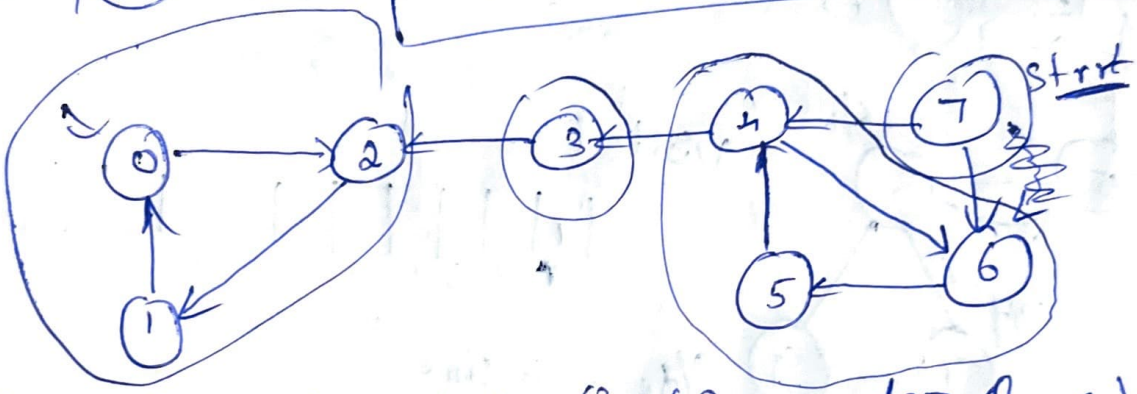
Sort all the ~~do~~ edges according to the finishing time

Stack





Step (2) - ~~Reverse~~ Revers the graph edges



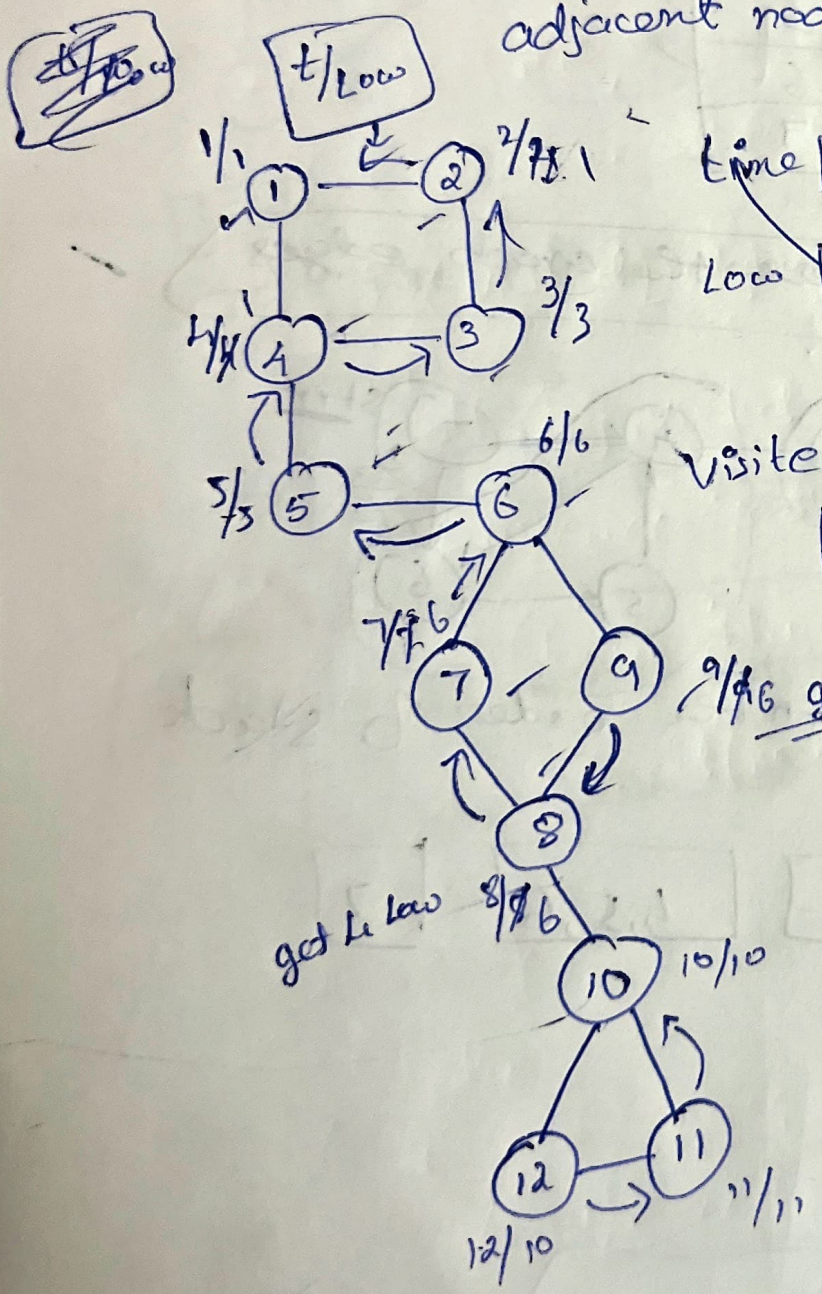
Step (3) - Do dst in the order of stack

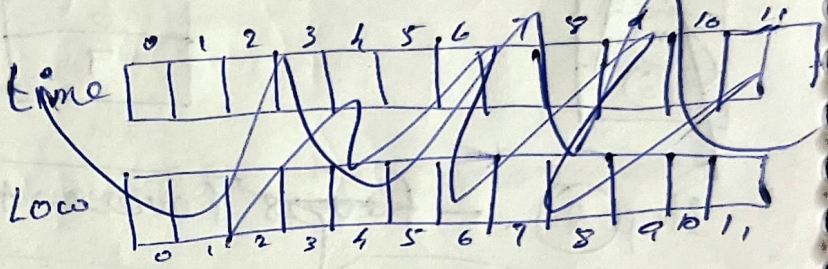
~~1~~	0, 1, 2	3	4, 5, 6	7

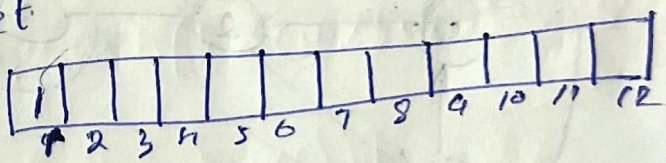
Tarjan's Algorithm

time[ ] - Dfs time Insertion

low[ ] - mn lowest time insertion of all adjacent nodes apart from parent



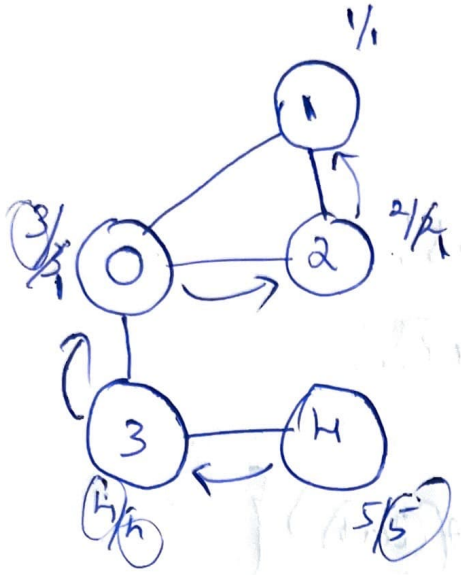




8,10
5,6
5,4

low[curr] > low[n]

?
(curr, n)
}




```
if (low[neighbour] > time[curr])
{
  res.add [curr, neighbour];
}
```

step(1) - Convert edges to adjacent list

step(2) - Create time, low, visited, res to store result.

step(3) - do dfs

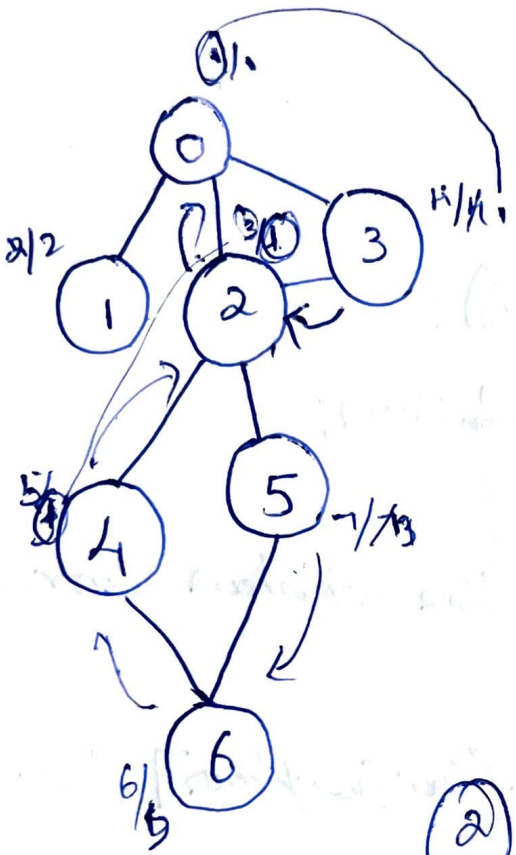
```
dfs() {
  time[curr] = timer[0];
  low[curr] = timer[0];
  timer[0]++;
  for (in
```

next page

```
dfs() {
    visited[curr] = 1
    timer[curr] = timer[0];
    low[curr] = time[curr];
    timer[0]++;
    for (int neighbour : adj.get(curr)) {
        if(visited[neighbour] == 0) {
        {
            dfs(adj, visited, neighbour, timer, low, res, curr)
            Low[curr] = math.min(low[curr], low[neighbour])
            if (low[neighbour] > time[curr]){
                res.add(new ArrayList<>(Arrays.asList
                                        (curr.neighbour))
            }
            else
            {
                Low[curr] = Math.min(time[neighbour],
                                     dow[curr]);
            }
        }
    }
}
```

Articulation point:

→ Nodes on whose removal in the graph brakes into multiple Components.



time[] - store the time of insertion on during Dfs

Low[] - mn of all adjcent nodes apert from Parent & visited nodes.

(2) (D)

if (Low[neighbour] >= time[curr] && parent != -1)

{

```
dfs()
{visited[curr]=+;
low[curr] = timer[0];
time[curr] = timer[0];
timer[0]++;
for(int neighbour : adj.get(curr)){
    if(neighbour ==parent) continue;
    if(visited[neighbour]==0){
        dfs(adj, visited, low, time, neighbour, curr, timer,
            res);
        low[curr] =Math.min(low[neighbour], low[curr])
        if(Low[neighbour] >= tim[curr] && parent != -1){
          res.add(curr);
          }
        child++;
    }
    else{
        low[curr] =Math.min (time[neighbour], low[cur
    }
}
if(child > 1 && parent == -1)
{ res.add(curr);
}}
```