Striver SDE Sheet

Part - 3

By Alok kumar

spyder._12

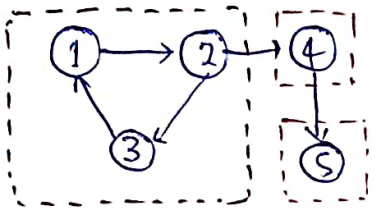
@insta

Graph Part-2

Strongly Connected Component (Kosaraju's Algorithm)

works on Directed graph

spyder.-12 @insta



G1

In strongly connect components → if we start from any node so we can visit all the node in that graph using every node.

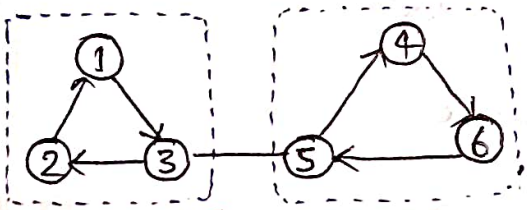
We can see we can visit all the node if we take any node from rectangle box so its a strongly connected component.

for 4 it can go back to 4 so its a strongly connected comp. himself alone.

1 2 3 → can be in any order

4

5



G2

1 2 3

4 5 6

basic intution is to apply DFS so we can visit nodes but in that case we will unable to config. the stongly connect compo-

for this we apply DFS in reverse order.

start from 5

↓ can go to any node.

goto 4

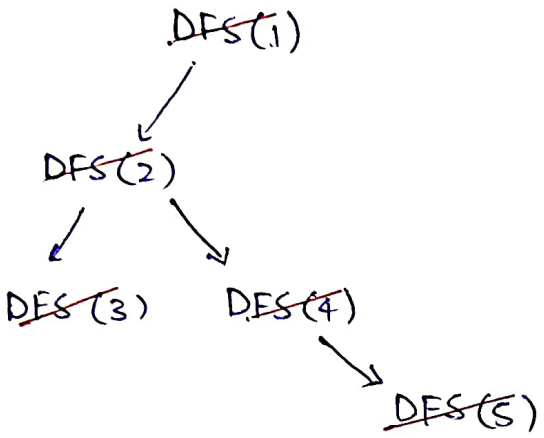
↓ can go to 5 but it is already vis.

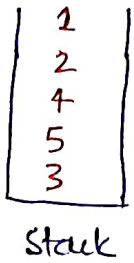
then 3, 2, 1

So start traverse from the back DFS calls.

① Sort all the nodes in order of finishing time. (Topo sort)

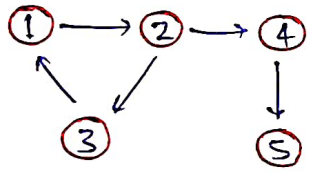
②

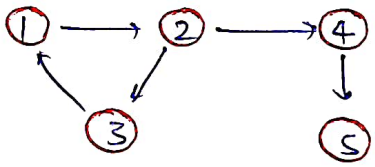




2)

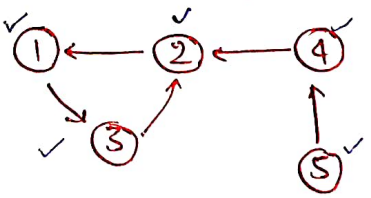
Now transpose the graph. (Means reverse the edge)





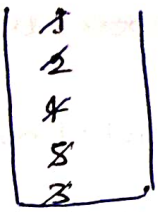
↓ reverse.

Now if we start traversing from the 1 so will goto 2 and 3 but no to 4.



so we never go apart from scc.

3) Do the DFS according to finishing time.



DFS(1)

DFS(3) DFS(4)

DFS(2) DFS(5)

print — 1 3 2
4
5

Toposort taken O(N+E)

transposing graph O(N+E)

≈ O(N+E)

Space complexity
= O(N+E) + O(N) + O(N)

```
void kosaRaju(adj, n) {
    int vis[];
    Stack<int> st;
    for(i=0; i<n; i++) {
        if(vis[i]==0) {
            dfs(i, st, adj, vis);
        }
    }
    // Now here code for transpose.
}
private void dfs(int node, st, adj, vis) {
    vis[node] = 1;
    for(Integer it : adj.get(node)) {
        if(vis[it]==0) {
            dfs(it, st, adj, vis);
        }
    }
    st.push(node);
}
```

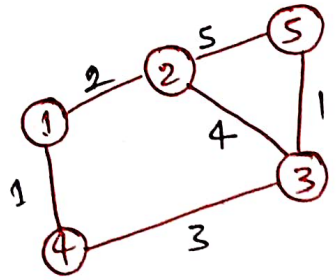
// Transpose code

```
ArrayList<ArrayList<int>> trans;
for(i=0; i<n; i++) {
        trans.add(new ArrayList<int>); }
for(i=0; i<n; i++) {
      vis[i] = 0;
      for (Integer it: adj.get(i)) {
            transpose.get(it).add(i);
                  }
            }

while(st.size()>0) {
        int node = st.peek();
        st.pop();
        if (vis[node] == 0) {
              revDFS(node, trans, vis);
              System.out.println();
                  }
            }
```

// revDFS

```
void revDFS(node, trans, vis) {
      vis[node] = 1;
      System.out.prt(node);
      for (Integer it: trans.get(node)) {
            if (vis[it]==0) {
            revDFS(it, trans, vis);
                  }
            }
      }
```

Print Shortest Path - Dijkstra's Algorithm



source = 1 , dest = 5

we have to return the list of paths.

So for reaching 1 to 5

we can go - $1 \rightarrow 2 \rightarrow 5$ total sum = 7

$1 \rightarrow 2 \rightarrow 3 \rightarrow 5$ total sum = 7

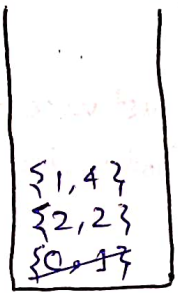
$[1 \rightarrow 4 \rightarrow 3 \rightarrow 5]$ total sum = (5)

parent -

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

dist [ ] -

~~0~~	~~2~~	I	~~1~~	I
1	2	3	4	5



{dist, node}

PQ

Now distance is 0 for reaching 1 means its a initial node, take this out of PQ

from 1 we can to {2,4}

So if we decide to go node 2 the distance taken is 2 which better than infinite.

and because we can coming on 2 from parent 1 so update the parent array

Now go to 4 and the moment go to 4 we take dist. $1 < \infty$ so update it.

put this in PQ also update the parent array

Now the Iteration for 1 has been completed.

So take (1,4) out from PQ

node = 4

from 4 we can go (1,3)

{7,5}
{4,3}
~~{1,4}~~
{2,2}

(dist, node)

PQ - minHeap.

So if we check the weight

1 → 4 its 1

d = 2 from N = 1

but

d = 2 which is greater than old distance

Now we go to 3, distance taken = 3 and prev distance = 1

Total dis = 4

it arrives from 4

parent -

1	1	~~4~~ 3	~~1~~ 4	~~∞~~ 2

dist -

0	2	~~∞~~ 4	1	~~∞~~ 7
1	2	3	4	5

all iteration for 4 has been completed.

Now take {2,2}

from 2 we can go to divide. {1, 3, 5}

Now for going back to 1 from 2 it take total d = 4 which is greater than prev. stored distance, not considerd.

2 → 3 so from going 2 → 3 it takes total d = 6 which is greater so not consider and for 5 it take 7 which is less than ∞

final parent array :

1	1	4	1	3

distance :

0	2	4	1	5

Spyder_12 @insta

Now destination = 5

→ shortest distance

So 5 arrived from 3

(3) → distance = 5

Now to reach 3 we are taking distance 4 Now in parent Now where 3 came from the parent is 4.

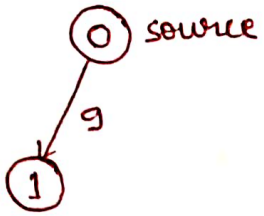
(1) → (4) → (3) → dist = 5

```
Static List<Int> shortestPath (n, m, int edges[][]) {
 ArrayList< ArrayList<Pair>> adj ;
 for (i→n) {
            adj.add (new ArrayList<>());
           }
 for(i→n) {
          adj.get(edges[i][0]).add (new Pair (edges[i][1] , edges[i][2]));
          adj.get (edges[i][1]). add (new Pair ( edges[i][0] , edges[i][2]));
        }

PriorityQueue <Pair> pq = new PriorityQueue <Pair> ((x, y) → x.first - y.first);
int[] dist = new int[n+1];
int[] parent = new int[n+1];
for (i= 1 , i<=n ; i++) {
          dis[i] = (int) (1e9);
          parent [i] = i ;
                    }
```

```
dist[1] = 0;
pq.add(new Pair(0,1));
while(pq.size()!=0){
    Pair it = pq.peek();
    int node = it.second;
    int dis = it.first;
    pq.remove();
    for(Pair iter: adj.get(node)){
        int adjNode = iter.first;
        int edW = iter.second;
        if(dis+edW < dist[adjNode]){
            dist[adjNode] = dis+edW;
            pq.add(new Pair(dis+edW, adjNode));
            parent[adjNode]=node;
        }
    }
}

List<Integer> path;
if(dist[n]==1e9){
    path.add(-1);
    return path;}

int node = n;
while(parent[node]!=node){
    path.add(node);
    node = parent[node];
}

path.add(1);
Collections.reverse(path);
return path;
}
```

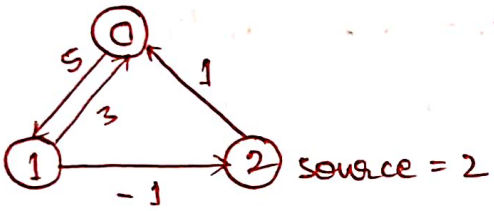
Bellman-ford · Help us to find detect negative cycle.

① Input - [[0,1,9]], src = 0



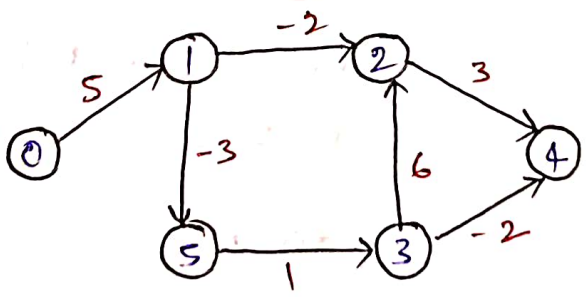
find the shortest distance of all the vertexes from source index.

② We can used the dijkstra algo over here but why we are using the Bellman-ford, because Bellman-ford can apply on Negative edges also.



Dijkstra fails due to negative edges.
also if there will be a negative cycle so dijkstra will keep running and gives the TLE.

source = 0



adj. [u, v, wt)

[3,2,6]
[5,3,1]
[0,1,5]
[1,5,-3]
[1,2,-2]
[3,4,-2]
[2,4,3]

Relax all the edges N-1 times sequentially

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

Now, we already know if we are going to find distance then take a distance array initializing ∞

distance

0	5	∞	∞	∞	∞
0	1	2	3	4	5

source will be 0

So we have to perform relaxation on every edges one by one

① [3, 2, 6]

∞

dis[3] + 6 < dis[2]

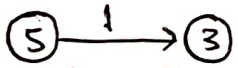
Not updated.

formula.

dis[u] + wt < dis[v]

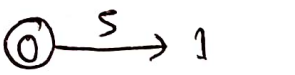
dis[v] = dis[u] + wt

② [5, 3, 1]



dis[5] + 1 < dis[3]

③ [0, 1, 5]



dis[0] + 5 < dis[1]

0 + 5 < ∞ (yes) update the val

dis[1] = 5

and update the distance in array.

So we have to perform this step n-1 times

after all iterations the final distance arr.

distance.

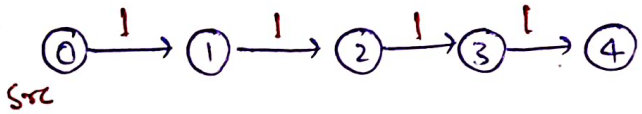
0	5	3	3	1	2
0	1	2	3	4	5

There are few questions.

Q1. why we are applying N-1 iterations

Q2. How to detect negative cycle

why N-1 ?



edges can be in any order

[u, v, wt]

[3, 4, 1]

[2, 3, 1]

[1, 2, 1]

[0, 1, 1]

dist

0	1	∞	∞	∞
0	1	2	3	4

[3, 4, 1]

③ $\xrightarrow{1}$ ④

dist[3] + 1 < dis[4] ✗

① $\xrightarrow{1}$ ②

dis[1]+1 < dis[2] ✗

② $\xrightarrow{1}$ ③

dis[2] + 1 < dis[3] ✗

⓪ $\xrightarrow{1}$ ①

dis[0] + 1 < dis[1] ✓

0 + 1 < ∞

So in the first iteration in the end the arr[1] gets updated.

in the next iteration arr[2] gets updated.

In the next iteration arr[3] and arr[4] gets updated.

So in the worst case the edges on the bottom can give you the ans as V-1

Need to traverse V-1 edges to reach from first to last.

for detecting Negative cycles:

On Vth iteration, the relaxation will be done, and if the dist[] arrays gets updated.

```
main (edges, V, s) {
    int dist[V];
for (i=0; i<V; i++) { dist[i] = (int)(1e8); }
dist[s] = 0;
for (i=0; i<V; ++i) {
        for (ArrayList<Integer> it; edges) {
            int u = it.get(0);
            int v = it.get(1);
            int wt = it.get(2);

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

// Nth relaxation to check negative cycle.

```
for (ArrayList<Integer> it : edges) {
        int u = it.get(0);
        int v = it.get(1);
        int wt = it.get(2);
    if (dist[u] != 1e8 && dis[u] + wt < dist[v]) {
        int temp[] = new int[1];
        temp[0] = -1;
        return temp;
    }
}
return dist;
    }
}
```

Time - O(V X E)

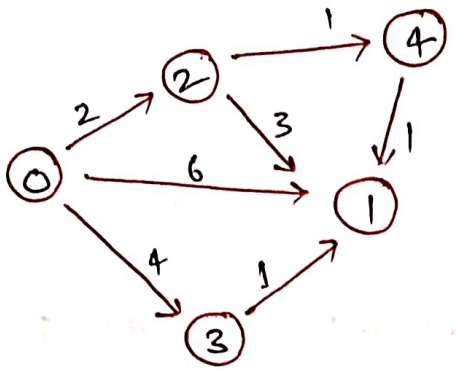
Exeternal space - O(V)

Take more time than dijekstra.

spyder.-12 @insta

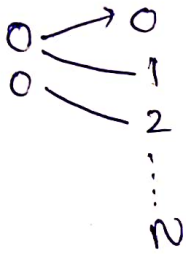
Floyd Warshal Algorithm. Multisource shortest path.

detect negative cycle.

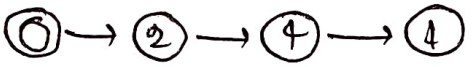


in the dijkstra and bellman it was working for single src

floyd warshal can be used for getting the distance to every Node like



for reaching 1 from 0 the shortest path



Note : go via every vertex / Node.

dis[0][1] ⟶ (0 → 1)

d[0][1] ⟶ (0 → 1) cost me 6

(0 → 2) + (2 → 1) cost me 5

(0 → 3) + (3 → 1) cost me 5

(0 → 4) + (4 → 1) cost me 4

minimal among all

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

but we have lineup undisturbed, implement this on a wider scale

0 ⟶ 4 there is not edge (Directly) went via (2)

means (0-2) has to be computed

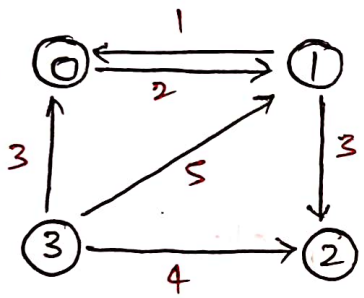
[0-2] [2-4]

means dp

we can use something that has been already computed,

↓

relatable.



understand if we are on 0 and for reaching to 0 what the cost will be - 0

Similarly we are on 1 so for reaching 1 what the cost will be means 0

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

cost.

In this case DG is here.

If undirected graph so put the weight two times in matrix with same value.

assume we are moving via vertex 0.

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

cost

via → 0

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

moving via k

i j

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

$0 + 2$

$0 \rightarrow 2 \Rightarrow [0][0] + [0][2]$

$[1 \rightarrow 2] \Rightarrow [1][0] + [0][2]$

$\Rightarrow 1 + \infty$

$\Rightarrow \infty$, Not a better option

$[1 \rightarrow 3] \Rightarrow [1][0] + [0][3] \Rightarrow 1 + \infty$, No update

$[2 \rightarrow 1] \Rightarrow [2][0] + [0][1]$

∞

$[2 \rightarrow 2] \Rightarrow [2][0]$

∞

$[2 \rightarrow 3] = [2][0]$

∞

$[3 \rightarrow 1] \Rightarrow [3][0] + [0][1]$

$\Rightarrow 3 + 2 = 5$

$[3 \rightarrow 2] \Rightarrow [3][0] + [0][2]$

$= 3 + \infty = \infty$

The same thing will happen for via 1, 2, and 3.

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

Now via one

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

means he is going to himself.

because starting from 1 going via 1 is no sense.

Copy the val's.

→ via 1

	0	1	2	3
0	0	2	5	
1	1	0	3	∞
2		∞	0	
3		5		0

and at same it doesn't make sense for going via one and reaching 1

$[0][2] \Rightarrow [0][1] + [1][2]$

$\Rightarrow 2 + 3 \Rightarrow 5$ (better)

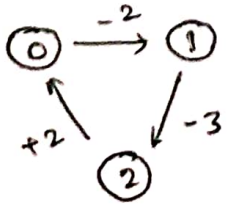
Similar done for others

via → 1 mat will be

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

How to detect a cycle

distance of node to himself should be zero.



Node 0 has to be 0. but it is mins so loop / cycle

```
main (mat[][]) {
    int n = mat.length;
    for(i → n) {
        for (j → n) {
        if (mat[i][j] == -1) {
                mat[i][j] = (int)(1e9);
                }
            if (i == j) mat[i][j] = 0;
                }
            }
        }

for (k = 0 ; k < n ; k++) {
        for(i → n) {
            for (j → n) {
                mat[i][j] = Math.min([mat[i][j] , mat[i][k] +
                                                mat[k][j]);
                }
            }
        }

for(i → n) {
    for(j → n) {
        if (mat[i][j] == (int) (1e9)) {
                mat[i][j] = -1;
                }
            }
        }
    }
```

Minimum spanning Tree : Prim's Algorithm

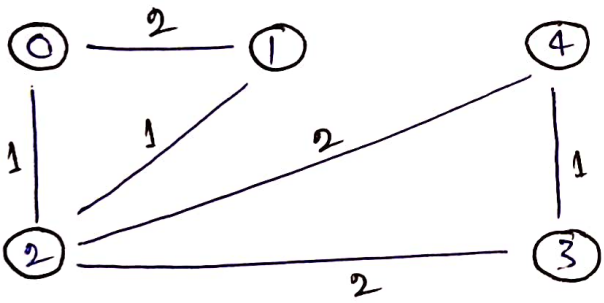
A graph will be given to you :

minimum Spanning Tree - No cycle in the Tree

↓

take those edge whose weight is minimum

n-1 edge are there



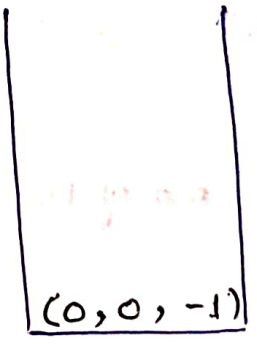
sum = 0

vis[] =

0	1	2	3	4
0	0	0	0	0

can start from any node -

starting from 0.



(wt, node, parent)

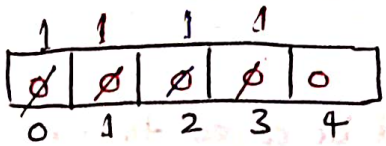
min Heap.

node = (0, 0, -1)

if parent == -1 mean first node so don't added to MST.

mark vis[0] = 1

vis =



adjacent of 0. are (1) and (2)
put in the min Heap only if unvisited.

MST →
[(0, 2)
(1, 2)
(2, 3)

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

minHeap

Now least weight will be on Top - property of min Heap.

Node = (1, 2, 0) out of PQ

Sum = 0 + 1 = 1

mark vis[2] as visited.

Now we are standing at 2 look at 0 but already vis.
1 is unvisited.
put that in PQ

4 is unvisited. put that in PQ

Similar for 3.

All adjacent of 2 has been completed.

Take (1, 1, 2) out of PQ
↓
unvisited. → yes.

Sum = 1 + 1 = 2

Mark 1 as visited

Now on 1 its adjacents are 0 and 2 but are allready vis.

Take Node = (2, 1, 0)

But 1 is already vis donot add in adj list.

Take Node = (2, 3, 2)

sum = 2 + 2 = 4

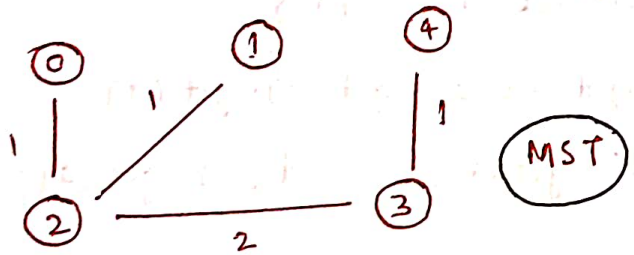
adjacent of 3 is (4) and 2 but 2 is already vis

↓ added in PQ

after PQ is empty.

sum will be 5

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



Need to create a pair

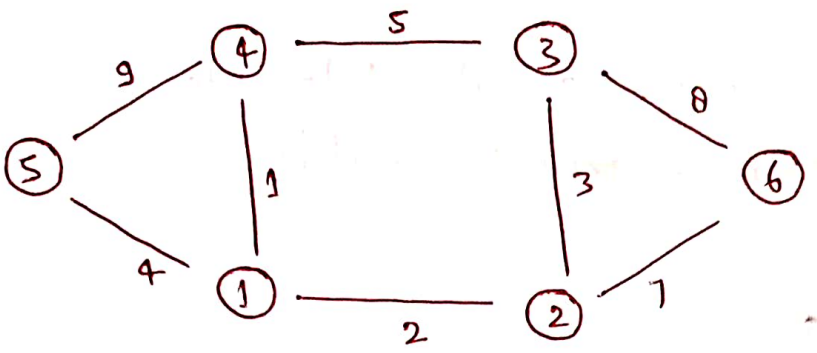
```
class Pair {
    int node;
    int dist;
    public Pair(int node, int dist) {
        this.node = node;
        this.dist = dist;
    }
}
```

```
main (V, ArrayList <ArrayList <ArrayList <Int>>> adj) {

PriorityQueue <Pair> pq = new PriorityQue (Pair>((x,y) →
                                    x.dist - y.dist);
int vis [] = new int [V];
pq.add (new Pair (0,0));
int sum = 0;
while (pq.size ()>0) {
        int wt = pq.peek().dist;
        int node = pq.peek().node;
        pq.remove ();
        if (vis [node] == 1) continue;
        vis [node] = 1;
        sum += wt;

    for (i=0; i< adj.get(node).size (); i++) {
            int edW = adj.get (node).get (i).get (1);
            int adjNode = adj.get (node).get (i).get (0);
            if (vis [adjNode] == 0) {
                pq.add (new Pair (edW, adjNode));
                }
            }

return sum;
    }

// if need MST so take parent as well in pair class
```

TC- O(E log E)

SC- O(E)

MST - Kruskal's Algorithm → disjoint set ds + MST



sort all the edges accuroding to their weight.

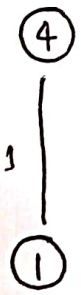
take the edges which has the minmam wt.

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)

node = (1, 1, 4)

Need to check ① — ④ is belonging to the same component or not

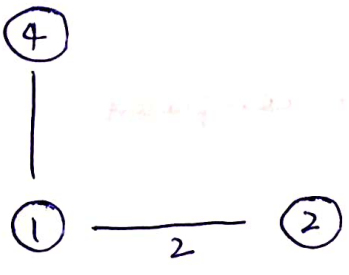
Not belong.



node =[2,1,2]

so does ① — ② belong to same compo → ✗

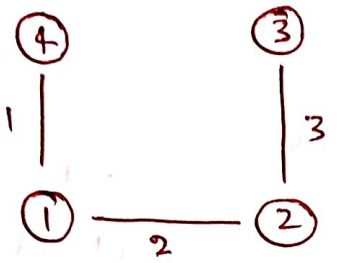
make an edge can connect Because parent of 1 and 2 are diff



node = (3, 2, 3)

② ⟶ ③ check 2 and 3 belong to same compo → ✗

So connect it to 2 because parent are diff.



node = (3, 2, 4)

② ⟶ ④

2 and 4 belong to same component

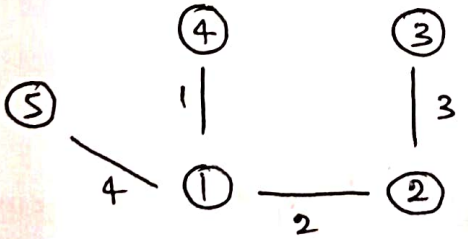
both parent is 1.

Not picked.

node = (4, 1, 5)

① —— ⑤

1 and 5 belong to same compo → ✗



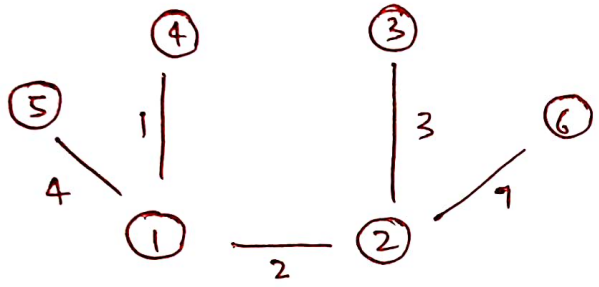
node = (4, 1, 5) has been completed

node = (5, 3, 4)

(3,4) belong to same compo. ✓ because form a cycle if create a edge

node = (7, 2, 6)

(2) — (6) does not belong to same compont ∴ create edge.



Now (8,3,6) and (9,4,5) are belonging to same compo.

TC- $O(M \log M)$ M is no. of edge to sort

$+$

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

$\approx O(M \log M)$.

SC = $O(M)$

spyder. - 12 @insta

Kth Largest Element in a Stream

Problem statement - implement a class where for a given k maintain stream. and continiously returns kth hightest test score after new no added.

lets say : [3, [4,5,8,2]] → initial array.

↓
keep track 3rd kth largest val.

if call add function to add element in arr

but if kth largest called,

So for maintaining the order need to take minHeap.

mean in sorted order - kth will be our answer

mean after add function call we need to tell kth largest.

"kthLargest", "add", "add", "add", "add", "add"

[[3, [4,5,8,2]] [3] [5] [10] [9] [4]]

↓
add three into arr and sort

[2, 3, 4, 5, 8]

add 5 in arr and sort.

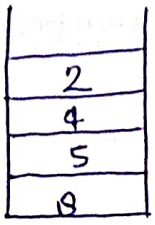
[2, 3, 4, 5, 8]
↓
deleted

we have to maintain arr size 3 and take peek

element

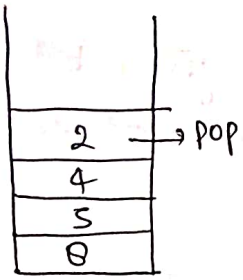
{4,5,8,2}

k=3

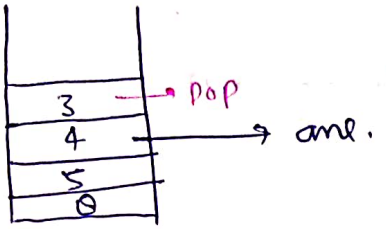


minHeap

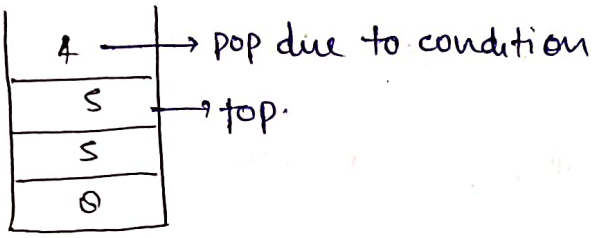
if minHeap.size() > k then pop



add → 3



add → 5



Bruteforce - if adding a no then after adding new no sort all the array so numbers can be set in ascending order.

nlogn → if sort

Optimal Approach:

```
private PriorityQueue <int> pq;
private int k;
public kth largest (int k, int [] nums) {
    this.k = k;
    for (int i : nums) {
        minHeap.add (i);
        if (minHeap.size ()>k) {
            minHeap.poll ();
        }
    }
}
```

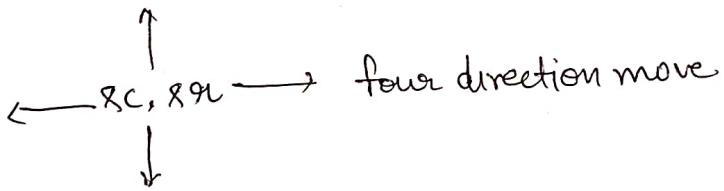
```
public int add(int val) {
    minHeap.add(val);
    if (minHeap.size() > k) {
        minHeap.pop();
    }
    return minHeap.peek();
}
```

Time complexity - O(N)
Space complexity - O(N)

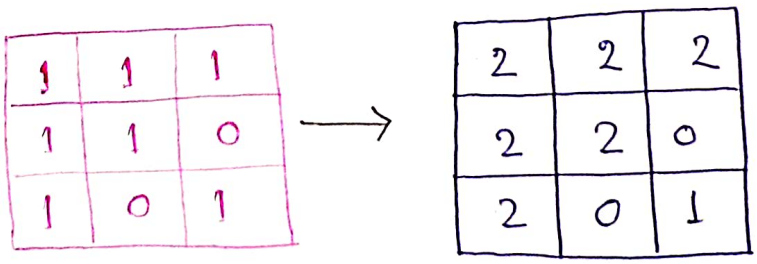
kth largest in array will same and based on similar concept.

flood fill

m x n → Matrix
sc, sr → index of a matrix.
col → color given



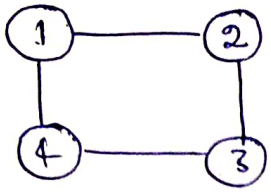
we have to keep track old color because where sc, sr exist there we have to apply new color




```
int[][] floodfill(int[][] image, int sr, int sc, int newcolor)
{
    if (image[sr][sc] == newcolor) return image;
    dfs(image, sr, sc, newcolor, image[sr][sc]);
    return image;
}
```

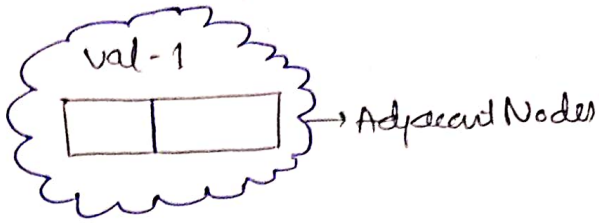
```
void dfs( int [][] image, int row, int col, int new_col, int old_col)
if (row >= image.length || row < 0 || col >= image[0].length || col < 0 ||
  image [row] [col] ] = oldcol) {
                        return;
                            }
 Image [row] [col] = newcolor;
 dfs (image, row+1, col, new col., oldcolor);
 dfs (image, row - 1, col, new col, old color);
 dfs (image, row, col+1, new col, oldcolor);
 dfs (image, row, col-1, new col, od color);
}
```

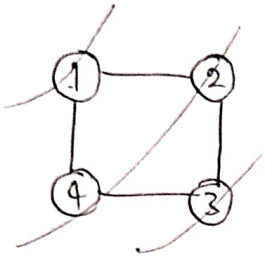
Clone Graph → "find the node who is already vis and should not be the parent of current node."



→ class

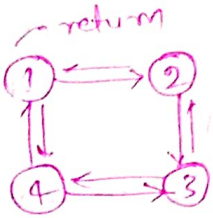
Node contains:



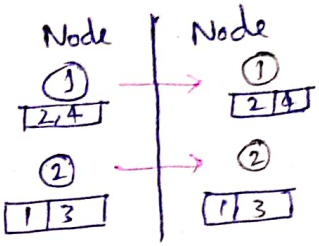


BFS

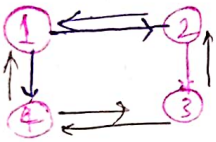
DFS



Hashmap

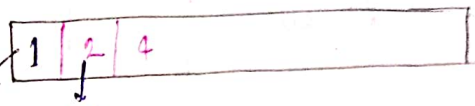


Try using BFS.



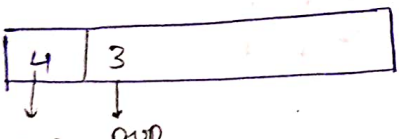
because 1 is neighbour of 2 so fill Node ① val → 2

take a queue

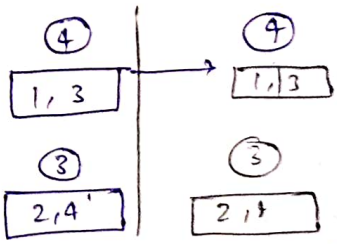


pop pop

take 2 from que and check in Hashmap its avail or not.

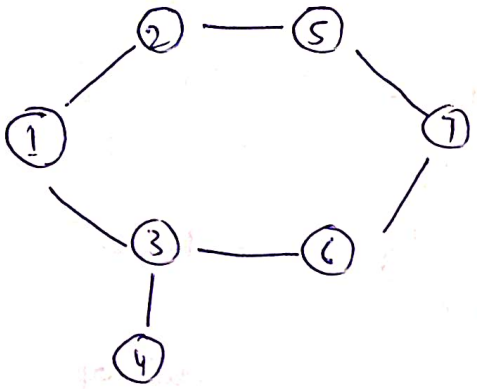


pop pop




```
if (node == null) return null;
Map<Node, Node> map;
Queue<Node> q;
q.add(node);
map.put(node, new Node(node.val, new ArrayList<>()));
while (!q.isEmpty()){
    Node h = q.poll();                    → construct val
    for (Node neighbour : h.neighbour) {
        if (!map.containsKey(neighbour)) {
            map.put(neighbour, new Node(
                neighbour.val, new Array<>());
            Que.add(neighbour);
        map.get(h).neighbour.add(map.get(neighbour));}
return map.get(node)
```

Detect cycle in an undirected graph → BFS.



adj list-

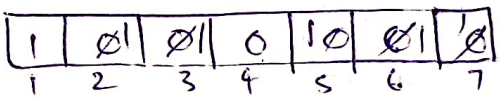
1 → {2, 3}
2 → {1, 5}
3 → {1, 4, 6}
4 → {3}
5 → {2, 7}
6 → {3, 7}
7 → {5, 6}

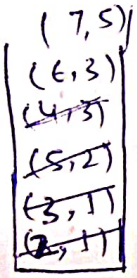
started in two diff path and get the same node (colliding in same portion) because there is a circle

→ it is -1 mean it came from no where.

BFS → Queue

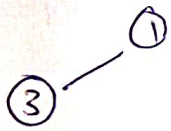
1 can go to 2 and 3.



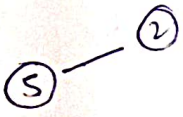


parent is 1 so don't need to go back

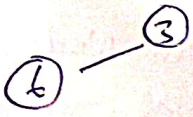
So go to 5



we can go (1, 4, 6) but 1 is parent no need



{2, 7} 2 is parent no need to go



Now (6 can go to 3, 7 so we can go to 7 but is already vis means some has

vis this before this means cycle.

```
main (int V, adj) {
boolean vis[];
for (int i=0; i<V; i++) {
    vis[i] = false;
}
for (int i=0; i<V; i++) {
    if (vis[i] == false) {
        if (checkCycle (i, V, adj, vis)) return true;
    }
}
return false;
}
```

["if graph is broken into multiple components"]
↳ connected components problem

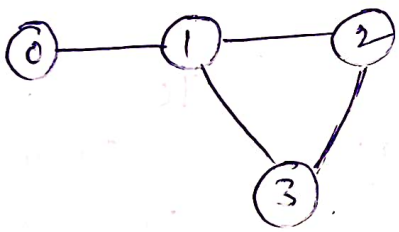
```
checkCycle (int src, adj, vis) {
vis[src] = true;
Queue <int, int> q = new LinkedList<>();
                  pair
q.add (new Pair (src, -1));
while (!q.isEmpty()) {
    int node = q.peek.first;
    int parent = q.peek.second;
    q.remove();
    for (int adjacent node : adj.get (node)) {
        if (vis[adjacentnode] == false) {
            vis[adjacentnode] = true;
            q.add (new Pair (adjacent node, node));
        }
        else if (parent != adjacentnode) {
            return true;
        }
    }
return false;
}
```

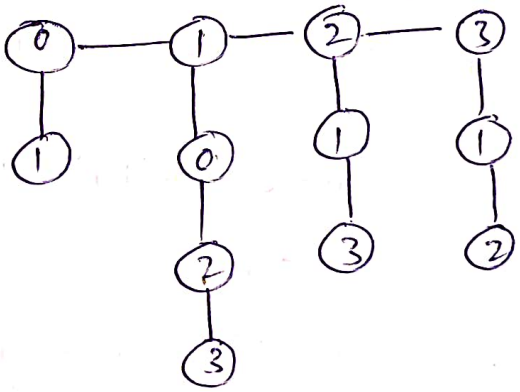
TC-
O(N+2E)
Sc → O(N)

DFS Approach -

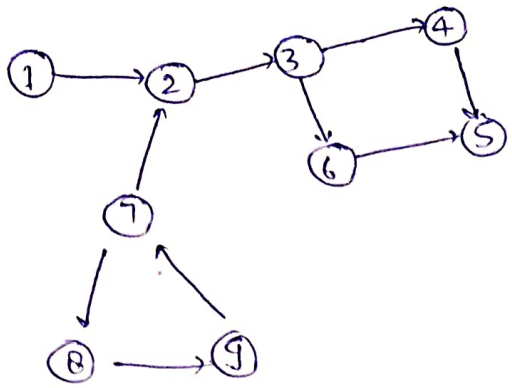
```
main(adj, V){
    boolean [vis];
    for (int i = 0; i<V; i++) {
        vis[i] = false;
    }
for (int i=0; i<V; i++){
if (!vis[i]) {
        if (DFS(vis, adj, i)){
            return true;
        }
    }
return false;
}
```

```
DFS(vis, adj, i){
vis[i] = true;
for (int v; adj.get(i)){
if (DFS(adj, v, vis, i))
        return true;
else if (v != parent)
        return
        true;
}
return false
```



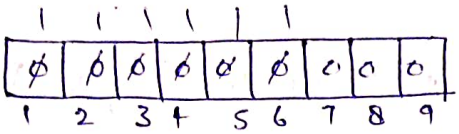


* Detect a cycle in a directed graph - DFS



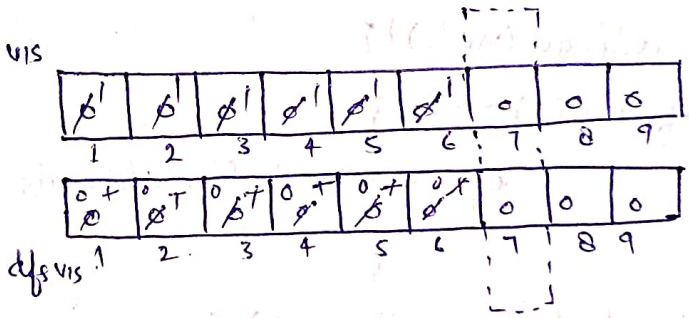
adjancy list

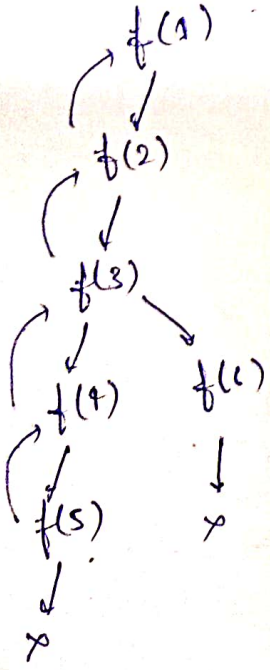
1 → 2
2 → 3
3 → 4, 6
4 → 5
5 →
6 → 5
7 → 2, 8
8 → 9
9 → 7



DFS(1)
↓
DFS(2)
↓
DFS(3) → DFS(6)
↓
DFS(4)
↓
DFS(5)
↓
×

for checking the cycle in directed graph We need two arrays to keep track visited.





imp - if any Node has no recursion calls so mark unvisited for that in dfs_visited arr.

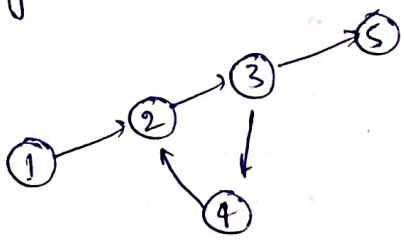
" if visited in both the arrays then only it can have a cycle."

```
f(i=1→9){
    if(!vis(i)){
        if(checkcycle(i)){
            return T;
        }
    }
}
```

TC → O(N+E)
SC → O(2N)

```
bool isCycle(N, adj){
    int vis[] = new vis[N];
    int dfs[];
    for(1 → N){
        if(vis[i]==0){
            if(checkCycle(i, adj, vis, dfsvis)==true) return true;
        }
    }
    return false;
}

bool checkCycle(node, adj, vis[], dfsvis[]){
    vis[node]=1;
    dfs[node]=1;
    for(i → adj.get(node)){
        if(vis[i]==0){
            if(checkCycle(i, adj, vis, dfsvis)==true){
                return true;
            }
        }
        else if(dfsVis[i]==1){
            return true;
        }
    }
    dfsVis[node] = 0;
    return false;
}
```

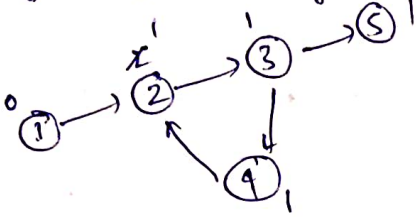
Using BFS

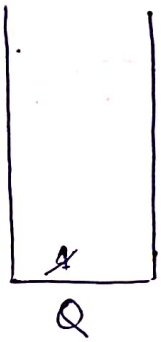


Kahn's algo

1 → 2
2 → 3
3 → 4, 5
4 → 2
5 → { }

going to find indegrees in the graph.:





Topsort = 1

The problem is simple → By this graph if you are able to generate a topo sort means it doesn't have a cycle otherwise it has.

↓

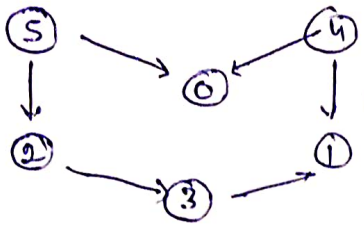
Because it is (DAG) → "Directed Acyclic graph"

Nature

```
main(V, adj){
int indegree[];
for (i→V){
      for(int i: adj.get(i)){
        indegree[i] ++;
          }
        }
Que <int>q;
for (i→V) {
        if (indegree[i]==0){
                q.add(i));
                                {
      }
```

```
int cnt = 0
while (!q.isEmpty()){
int node = q.peek();
q.remove();
cnt++;
for (int i: adj.get(node)){
   indegree[i]--;
  if (indegree[i]==0) q.add(i);
                    }
                }
if (cnt == V) return false;
return true;
```

* Kahiva Algorithm.



Topological sorting (BPS)

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

linear ordering vertices such that if there is an edge between u & v. u appears before v.

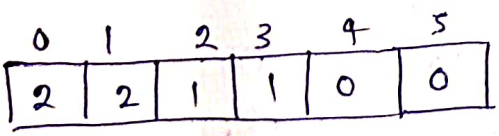
for example linear order of this graph is - 5 4 0 2 3 1

edges -

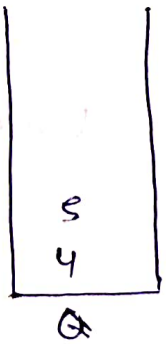
5 0 → 5 appears before zero
4 - 0 → 4 same "
5 2
4 1 ← edges.
2 3
3 1

this was that the linear ordering definit. states.

anyorder that follow linear ordering vertices also it has be a nature of DAG.



Indegree



if indegree is 0 so take it as start for linear ordering.

4 → out of Q.

Now 4 was into 0 and 1.

go into that array and reduce the val by -1.

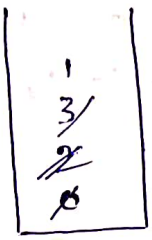
0	1	2	3	4	5
~~1~~	~~1~~	1	1	0	0

after that check the indegree one zero, no-

take out 5 out of q.

from 5 we can go 0 and 2 so now the arr will be.

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



take 0 and pop.

* from 0 we can't go.
* from 2 we can go 3.

Now take 3 and pop

↓

we can at 1.

Now q. empty,

4 5 0 2 3 1 ⟶ valid topo seq.

```
main(adj){
int v = adj.size();
int indegree[v];
for(i=0;i<v;i++){
    for(int it:adj.get(i)){
        indegree[it]++;
    }
}
```

```
Queue<int> q;
for(i=0;i<v;i++){
    if(indegree[i]==0){
        q.add(i);
    }
}
Arraylist<int> ans;
i=0;
while(!q.isEmpty()){
    int node = q.peek();
    q.remove;
    ans.add(node);
```

```
for (int it : adj.get(node)) {
        indegree[it]--;
        if (indegree[it] == 0) {  q.add(it);
                                  }
                               }
return ans;
}
```

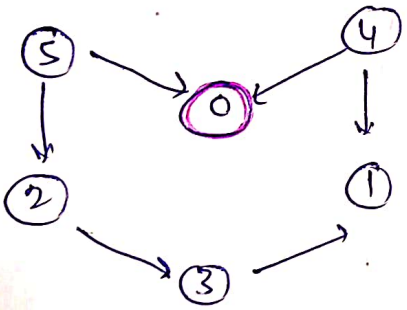
Topo sort using DFS -

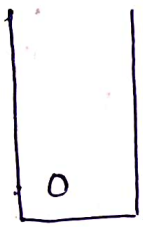
The approach will be same as Bfs means u has to appears before v and it has to be a linear.

↓

No Acyle

So same for storing the visited element take an array and for maintain the call take a stack.





stack.

adj.

0 → { }

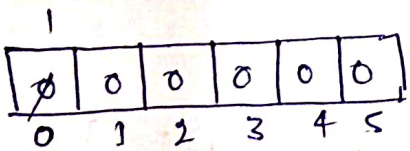
1 → { }

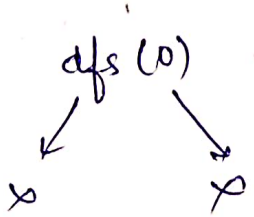
2 → {3}

3 → {1}

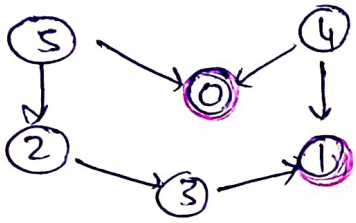
4 → {0,1}

5 → {0,2}

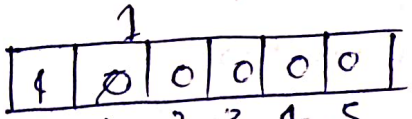


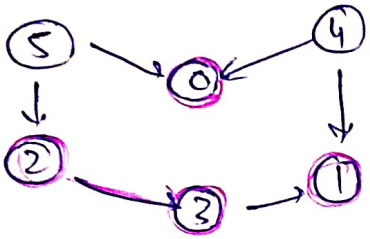


we can go any where from 0



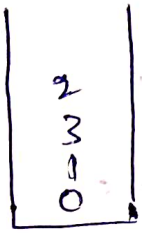
dfs(1)

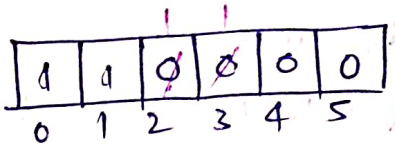




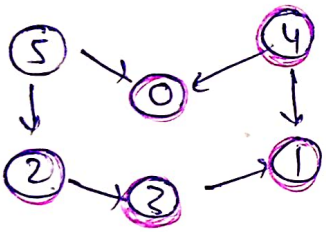
dfs(2)

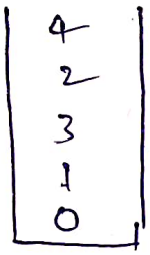
dfs(3)

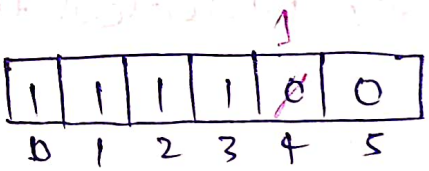




dfs(1) → but already visited in the arr.







dfs(4)

dfs(0) dfs(1) → both visited

same for dfs(5) so we will visit all the nodes.

So the intution is simple take our vis array. Start traverse from 0 node → 0 and check if not vis. then call the function and mark the node as vis in vis array.

```
topoSort(N, adj){
Stack<int> st;
int vis[N];
for(i=0;i<N;i++){
    if(vis[i]==0){
        Topological(i, vis, adj, st);
            }
        }
int topo = new[N];
int ind=0
while(!st.isEmpty()){
    topo[ind++] = st.pop;
        }
return topo)
}
```

/* calling the Topological function.

```
Topological(node, vis[], adj, stack){
    vis(node] = 1;
    for(Integer it. adj.get(node)){
        if(vis[node]==0){
            Topological(it, vis, adj, stack);
                }
            }
    st.push(node);
        }
```

Time complexity - O(N+E)

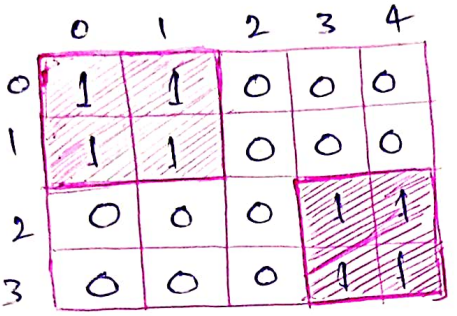
Space - O(N) + O(N) for stack and visited

Auxilary space → O(N) → for Recursion calls.

* Number of Islands *

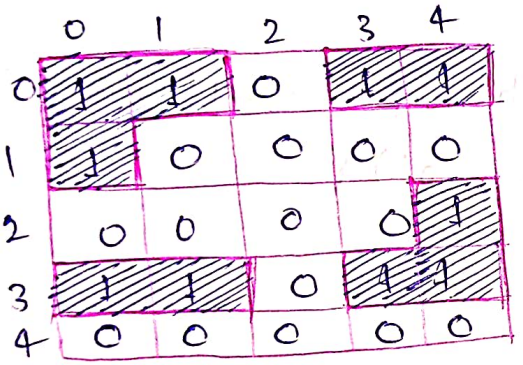
will be given a matrix where 1 → land, 0 → water we have to find the Number of iselands.

what is island → an area surrounded by water.

① 

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

mean there are 2 iselands

② 

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

Total four istands.

① create an array visited -2D

start from mat[0][0] and go in all four directions if island so mark. 0 as the index.

Suppose for any index all the recursion call's has been completed so cnt ++ so put that in a set.

the direction we can move.

(row-1, col)

(row, col-1) (row, col) → (row, col+1)

(row+1, col)

```
NumofIselands (char[][] grid) {
    int N = grid.length;
    int cnt = 0;
    for (i=0; i<N; i++) {
        for (j=0; j<grid[i].length; j++) {
            cnt += dfgrid (i, j, grid);
        }
    }
    return cnt;
}

// function call

dfgrid (i, j, grid) {
    if (i<0 || j<0 || i == grid.length || j == grid[0].length ||
                    grid[i][j] == '0') {
        return 0;
    }
    grid[i][j] = '0';
    dfgrid (i+1, j, grid);
    dfgrid (i-1, j, grid);
    dfgrid (i, j+1, grid);
    dfgrid (i, j-1, grid);
    return 1;
}
```

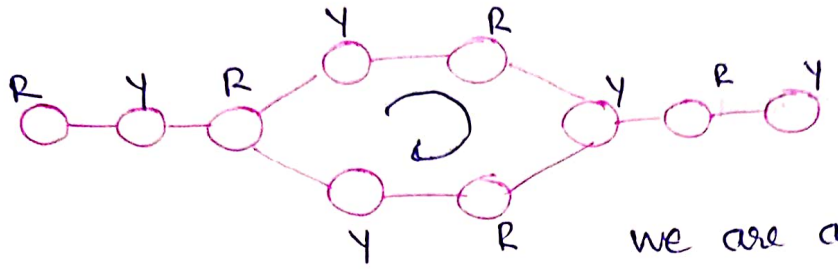
Time complexity - $O(N \times M \times \log(N \times M)) + O(N \times M \times 4) \sim O(N \times M)$

Space complexity - $O(N \times M)$, $O(N \times M)$

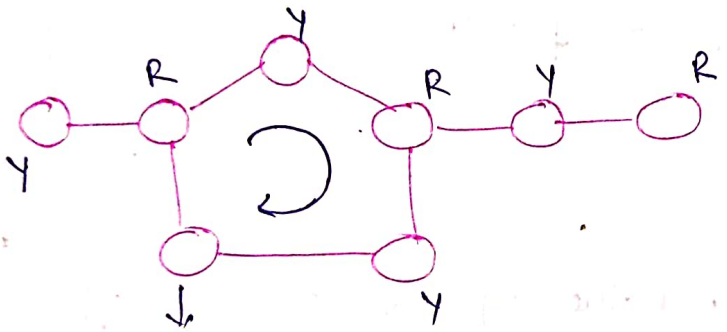
* Bipartite graph : BFS

↳ color the graph with two colors so no two adjacent nodes has same color.

color we are taking - [Red, Yello]



we are able to color this graph using 2 colors. → So its a Bipartite graph



over here

we can see that if we fill the red color so it connected to upper node and if we fill yello so its connected to right filled with yello.

we can say its not a Bipartite graph.

Intution - Any graph with odd length cycle will not be abipartite graph.

② for solving this problem using BFS take anqueu.

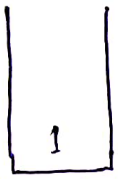
③ instedd of visited array create an array full with -1 so we can store the color vals at every node.

1	2	3	4	5	6	7	8	
-1	-1	-1	-1	-1	-1	-1	-1	→ color

↓

we will try to color with 0 and 1.

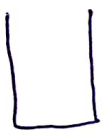
1 → {2}
2 → {1, 3, 6}
3 → {2, 4}
4 → {3, 5, 7}
5 → {4, 6}
6 → {2, 5}
7 → {4, 8}
8 → {7}



after putting the val 1 into the Q fill the color in array.

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

Now get out the top most element of queue and get the adjacent nodes of 1



node → 1

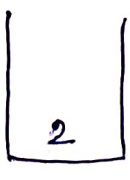
and the Neighbour node = 2

so we check if 2 colored or not so its not colored

So we have to color this with opposite color so it will be 1.

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

after that put the node into the queue.



1 has no other adjacent nodes.

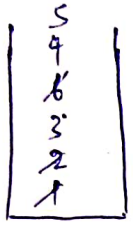
take 2 out of queue.

node → 2

adjacent node of 2 ⟶ {1, 3, 6}

So we start from 1 but already colored., we have {3,6} 2 nodes which not colored.

So now 2 has the color 1 we will give the opposite colors.



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

Now iteration for Node 2 has been done.

Now take 6 out of q and perform same step

Now node → 6 adjacent {2,5}

① Now 2 is colred and its with different color.

② We have node 5 so color it with 1

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

Now take 4 out of q and Nowed 3, 5, 7

So checked 3 filled corred in opposite color but 5 is not so its not a bipartite graph.

If at any moment any two nodes that same color means its not bipartite graph.

```
boolean isBipar (V, adj){
  int color[];
  for(i<v){ color[i] = -1; }
  for(i<v){
        if[color[i] == -1){
            if(check(i,v, adj, color) == false){
                return false;
            }
        }
  }
  return true;
}

boolean check(start, v, adj, color){
    Queue<int> q;
    q.add(start);
    color[start] = 0;
    while(!q.isEmpty){
        int node = q.poll();
        for(int it: adj.get(node)){
            if(color[it] == 1 - color[node];
            if(color[it] == -1){
                color[it] = 1 - color[node];
                q.add(it);
            }
            else if(color[it] == color[node]){
                return false;
            }
        }
    }
    return true;
}
```