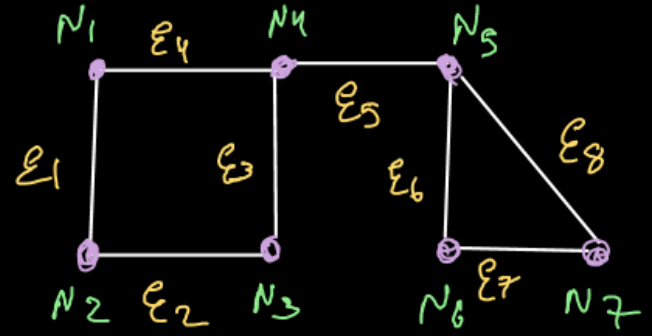
Graph Terminologies

① Node / Vertex
② Edges

} Graph is a collection of vertices & edges

$G = (V, E)$



③ Directed / Undirected

④ Unweighted / Weighted

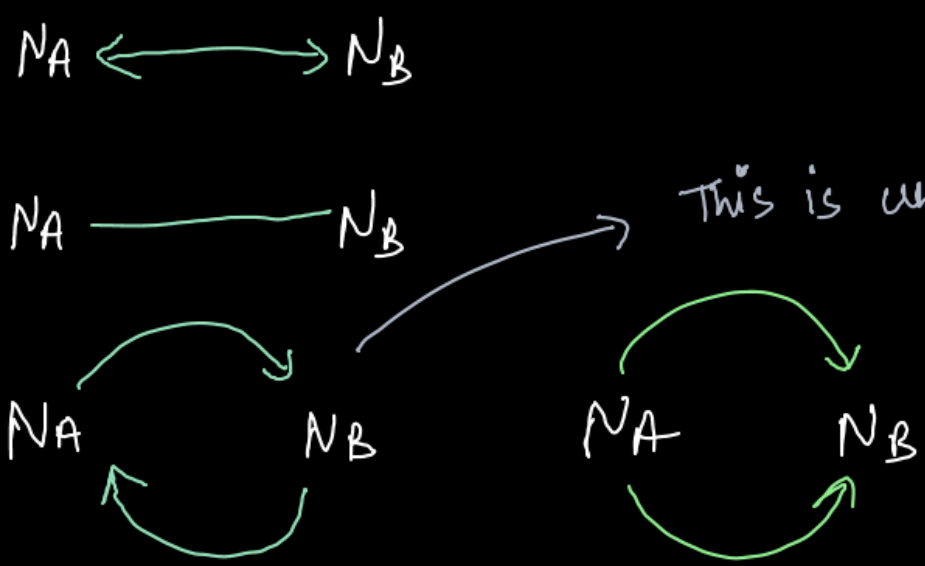
⑤ Acyclic / Cyclic

⑥ Incoming edge / Outcoming edge

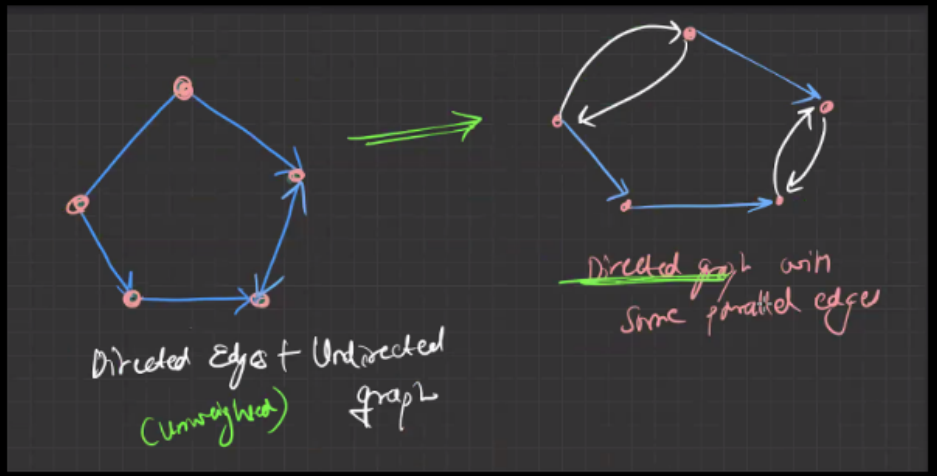
⑦ Parallel Edge

Undirected edge is both incoming as well as outgoing.

Parallel Edge

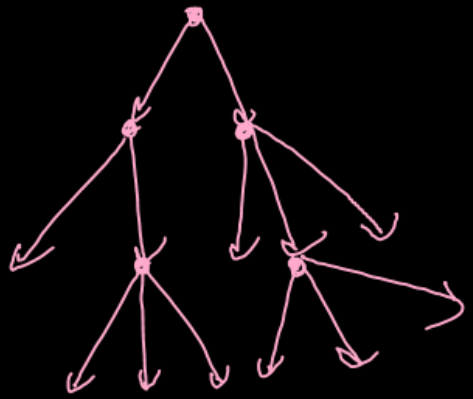


Parallel edges



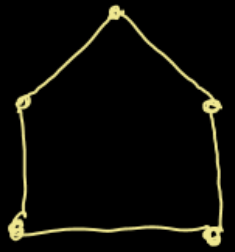
Cyclic and Acyclic Graph

Acyclic : Come back to the same node by travelling some edge

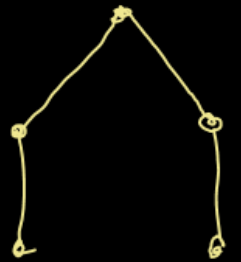


Acyclic

Cyclic



Cyclic



acyclic

self loop edge

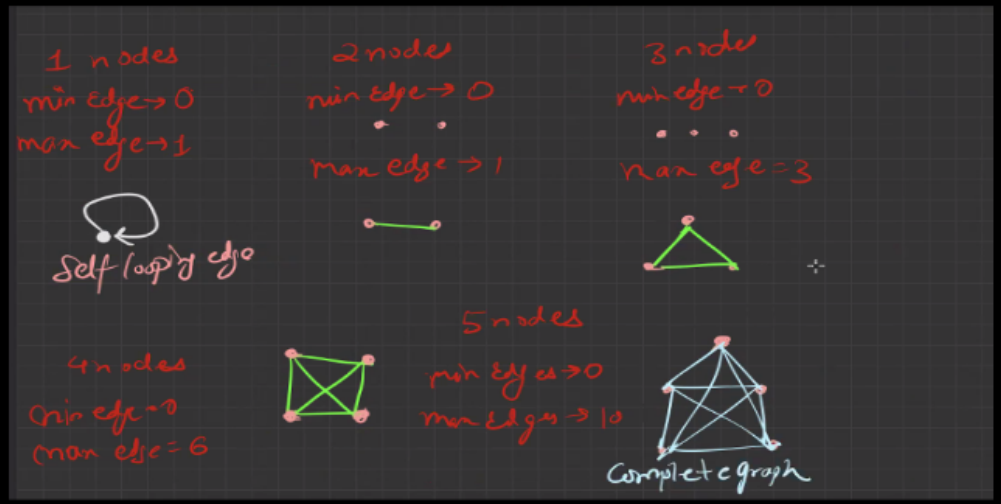
Cyclic

Generic tree is a directed (or rooted) and acyclic graph.

Every Acyclic graph is not a tree but every tree is an acyclic (rooted) graph

Graph can have minimum 0 edges. (whether directed or undirected)

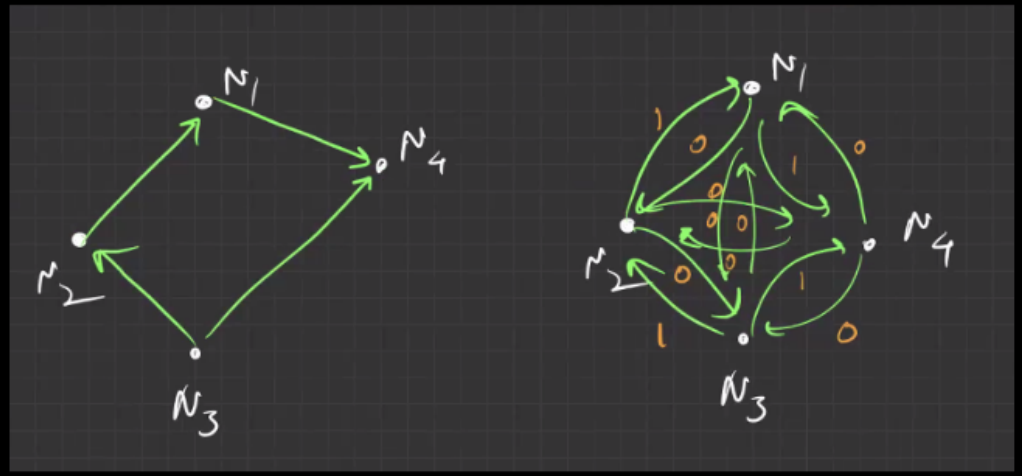
An undirected graph (without self loops & || edges) can have maximum of $^{N}C_{2}$ edges. ([N(N-1)]/2)



Complete Graph: A graph where every Node is directly connected to every other node. (edge b/w the nodes)

In directed graph, if parallel edges are allowed then max edges are N*(N-1) else it is [N*(N-1)]/2.

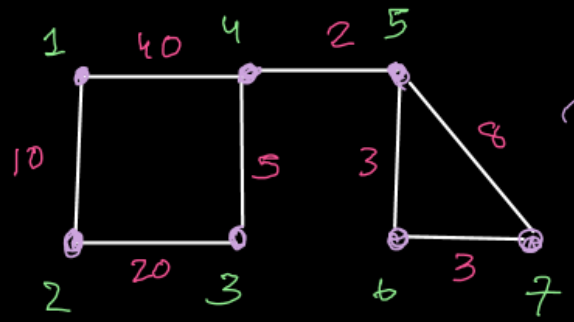
Unweighted Graph to Weighted Graph



To edge present hai, give them weight 1, else 0.

Ways of Implementing / Representing Graph

Edge List



N nodes,
M edges
Input: M pairs or triplets (weighted)

input space : O(E)

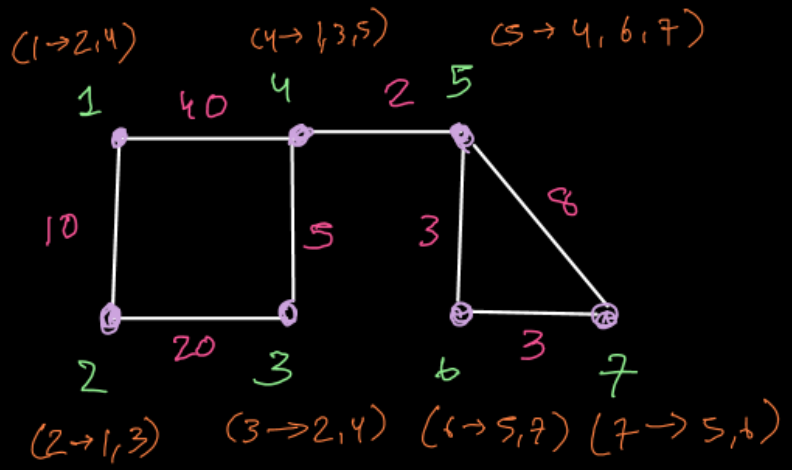
{ {1, 4, 40}
{1, 2, 10}
{2, 3, 20}
{3, 4, 40}
{4, 5, 2}
{5, 6, 3}
{5, 7, 8}
{6, 7, 3} }

To find neighbor of a node, traverse the entire Edge List.
O(E) → to find a neighbor

Hence edge list is not efficient.

Input se hume pata chalta hai ki graph weighted hai ya unweighted.
Directed / undirected will be mentioned in the question.

Adjacency Matrix



	N_1	N_2	N_3	N_4	N_5	N_6	N_7
N_1	$+\infty$	10	$+\infty$	40	$+\infty$	$+\infty$	$+\infty$
N_2	10	$+\infty$	20	$+\infty$	$+\infty$	$+\infty$	$+\infty$
N_3	$+\infty$	20	$+\infty$	5	$+\infty$	$+\infty$	$+\infty$
N_4	40	$+\infty$	5	$+\infty$	2	$+\infty$	$+\infty$
N_5	$+\infty$	$+\infty$	$+\infty$	2	$+\infty$	3	8
N_6	$+\infty$	$+\infty$	$+\infty$	$+\infty$	3	$+\infty$	3
N_7	$+\infty$	400	$+\infty$	$+\infty$	8	3	$+\infty$

Input Space : $O(N^2)$

Neighbor → $O(N)$

mat[r][c] = mat[c][r]

Edge List vs Adjacency Matrix

We know that accessing one neighbour in edge list takes $O(E)$ time.

Also, accessing one neighbour in adj. matrix takes $O(N)$ time.

Max no of edges for N nodes can be $= \frac{N(N-1)}{2} = O(N^2)$

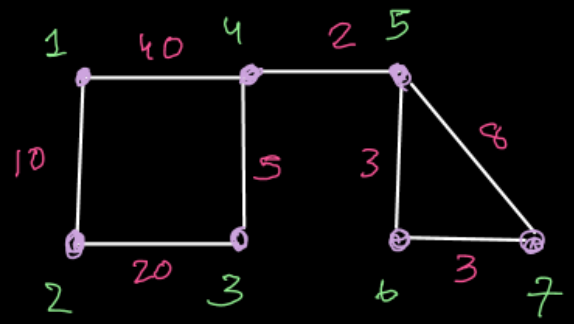
Hence, adjacency matrix in WC is $O(N)$ & Edge list is $O(N^2)$.

So, accessing a neighbour is easy in adjacency matrix as compared to Edge List.

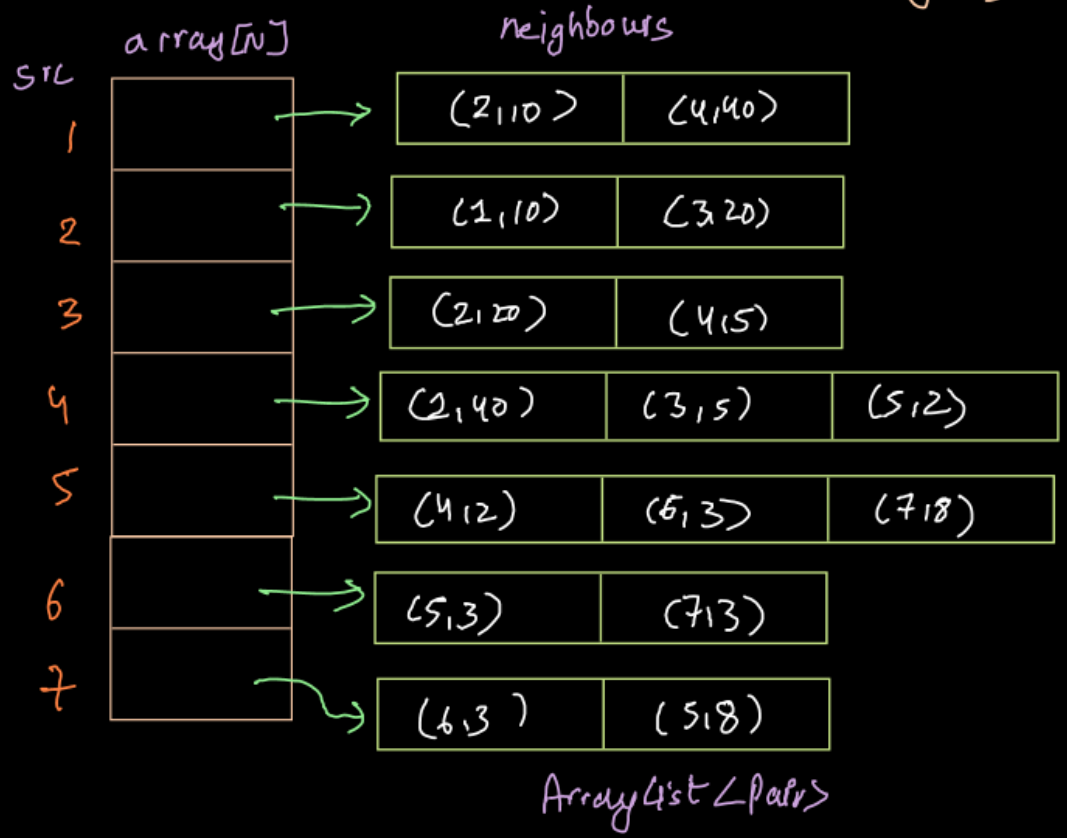
Adjacency List

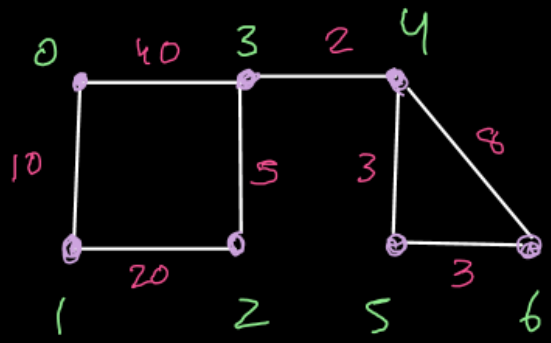
Neighbours finding TC → O(outgoing edges) = O(outdegree)

List (arrayList) of neighbours of each node.




```
ArrayList<Pair>[] adj =
        new ArrayList[N];

for(int i=0;i<N;i++){
   adj[i] = new ArrayList<>();
}
```






```
Finished in 157 ms
0 -> (3,40)     0 -> (1,10)
1 -> (0,10)     1 -> (2,20)
2 -> (1,20)     2 -> (3,5)
3 -> (0,40)     3 -> (2,5)      3 -> (4,2)
4 -> (3,2)      4 -> (5,3)      4 -> (6,8)
5 -> (4,3)      5 -> (6,3)
6 -> (5,3)      6 -> (4,8)
```

```java
class Pair {
    int nbr;
    int wt;

    Pair(int nbr, int wt) {
        this.nbr = nbr;
        this.wt = wt;
    }
}
```

```java
class Graph {
    ArrayList<Pair>[] adj;

    Graph(int n) {
        adj = new ArrayList[n];
        for(int i=0;i<n;i++) {
            adj[i] = new ArrayList<>();
        }
    }

    public void addEdge(int src, int dest, int wt) {
          adj[src].add(new Pair(dest,wt));
          adj[dest].add(new Pair(src,wt));
    }
```

```java
public static void main(String[] args) {
    Scanner scn = new Scanner(System.in);

    int vtces = scn.nextInt();
    Graph g = new Graph(vtces);

    int edges = scn.nextInt();

    for(int i=0;i<edges;i++) {
        int src = scn.nextInt();
        int dest = scn.nextInt();
        int wt = scn.nextInt();
        g.addEdge(src,dest,wt);
    }

    g.printAdjList();
}
```

```java
    public void printAdjList() {
        for(int i=0;i<adj.length;i++) {
            for(int j=0;j<adj[i].size();j++) {
                System.out.print(i + " -> (" + adj[i].get(j).nbr + "," + adj[i].get(j).wt +
")\t");
            }
            System.out.println();
        }
    }
}
```

Graph Applications

① Google Maps

→ minTime (src, dest)

→ minDist (src, dest)

→ connected cities in min roads

② Social Media (facebook/Meta)

→ friends are nodes & connections are edges

→ post sharing to direct friends & mutual friends

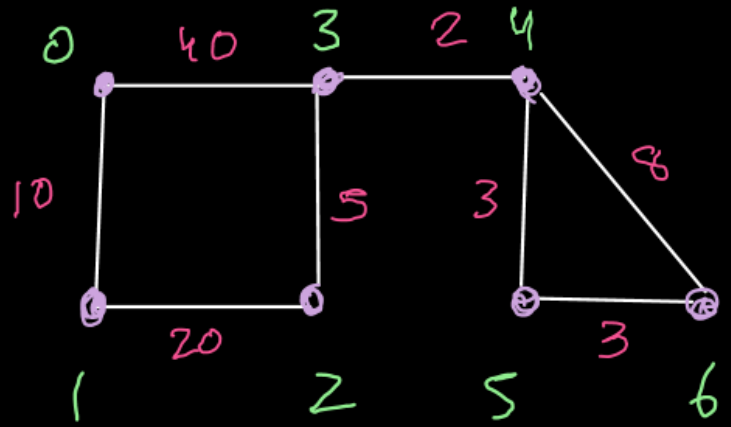
→ News feed.

③ OS → Deadlock detection + DBMS using Resource Allocation Graph (Bankers Algo)

④ Splitwise App → Money division.

Depth first Search (DFS) Traversal. {Has Path Question}

(Recursion + Backtracking)

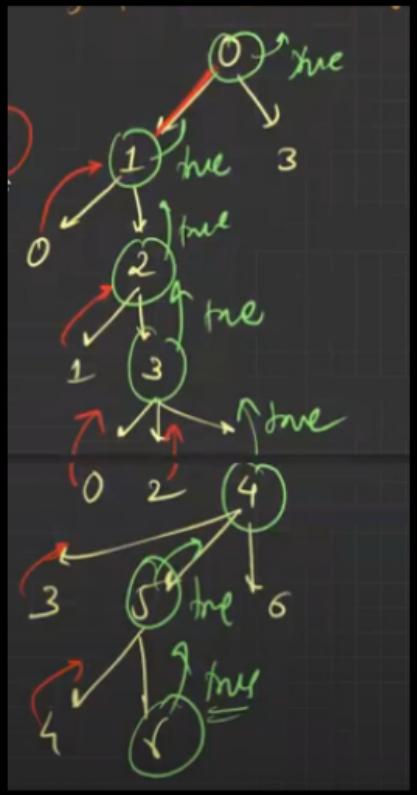


src = 0

dest = 6

0	1	2	3	4	5	6
f T	f T	f T	f T	f T	f T	f T

vis



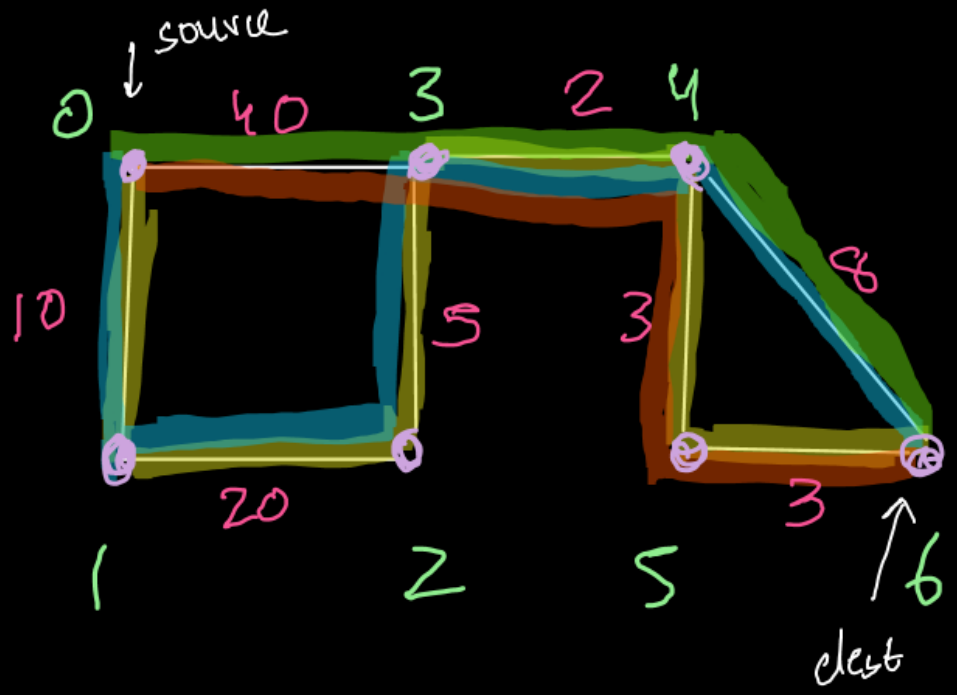
```java
static class Edge {
    int src;
    int nbr;

    Edge(int src, int nbr) {
        this.src = src;
        this.nbr = nbr;
    }
}
```

```java
public boolean DFS(ArrayList<Edge>[] graph, int src, int dest, boolean[] vis)
{

    if(src == dest) return true;

    vis[src] = true;

    for(Edge e : graph[src]) {
        if(vis[e.nbr] == false) {
            boolean hasNbrPath = DFS(graph,e.nbr,dest,vis);
            if(hasNbrPath == true) return true;
        }
    }

    return false;
}
```

```java
public boolean validPath(int n, int[][] edges, int source, int destination) {

    ArrayList<Edge>[] graph = new ArrayList[n];
    for(int i=0;i<n;i++) {
        graph[i] = new ArrayList<>();
    }

    for(int i=0;i<edges.length;i++) {
        int u = edges[i][0];
        int v = edges[i][1];

        graph[u].add(new Edge(u,v));
        graph[v].add(new Edge(v,u));
    }

    boolean[] vis = new boolean[n];
    return DFS(graph,source,destination,vis);
}
```

$O(N+E)$ → Time Complexity

All Paths (Single source - Single destination)

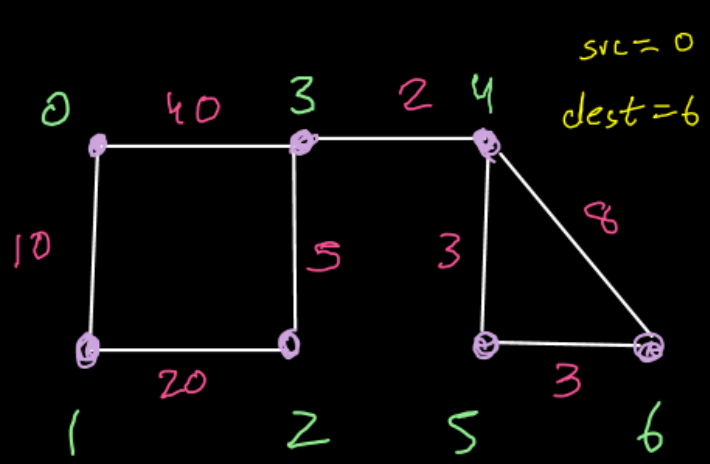


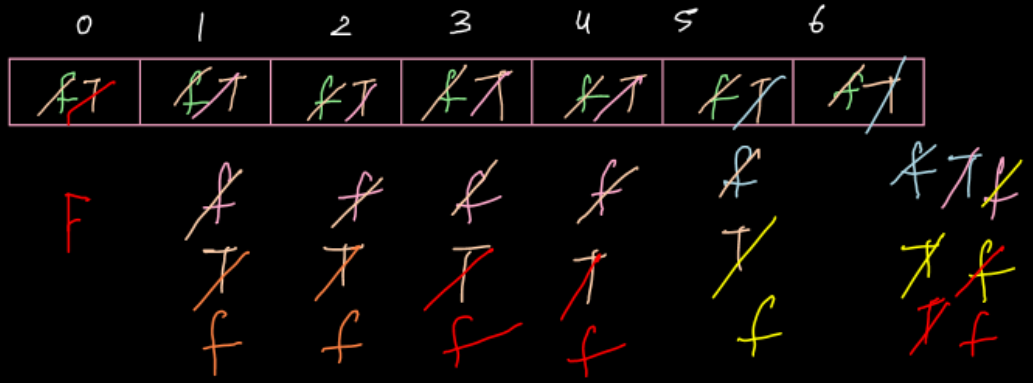
"0 1 2 3 4 5 6"

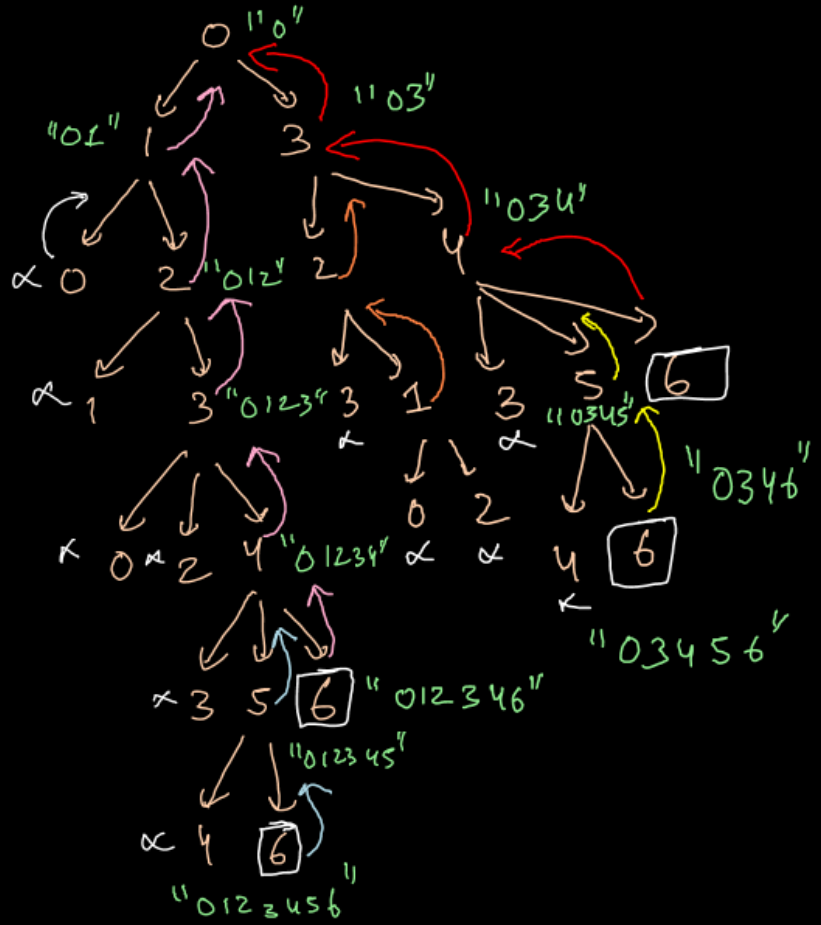
"0 1 2 3 4 6"

"0 3 4 5 6"

"0 3 4 6"


src = 0
dest = 6
0
40
3
2
4
10
5
3
8
20
3
1
2
5
6


0 1 2 3 4 5 6
F


"0"
"01"
"03"
"034"
"012"
"0123"
"01234"
"0345"
"0346"
"03456"
"012346"
"01234 5"
"012345 6"

```java
public void DFS(int src,int dest,int[][] graph,boolean[] vis,List<Integer>
psf,List<List<Integer>> res) {

    if(vis[src] == true) return;

    vis[src] = true;
    psf.add(src);

    if(src == dest) {
        res.add(new ArrayList<>(psf));
    }

    for(int nbr : graph[src]) {
        DFS(nbr,dest,graph,vis,psf,res);
    }


    vis[src] = false;
    psf.remove(psf.size() - 1);
}
```

loop detection

ArrayList is passed as a parameter.

TC: O(exponential)

```java
public List<List<Integer>> allPathsSourceTarget(int[][] graph) {
    List<Integer> psf = new ArrayList<>();
    List<List<Integer>> res = new ArrayList<>();
    boolean[] vis = new boolean[graph.length];

    DFS(0,graph.length-1,graph,vis,psf,res);
    return res;
}
```

```java
List<List<Integer>> res;

public void DFS(int src,int dest,int[][] graph,boolean[] vis,List<Integer> psf) {

    if(vis[src] == true) return;

    if(src == dest) {
        psf.add(dest);
        List<Integer> copy = new ArrayList<>(psf);
        res.add(copy);
        psf.remove(psf.size() -1);
        return;
    }

    vis[src] = true;
    psf.add(src);

    for(int nbr : graph[src]) {
        DFS(nbr,dest,graph,vis,psf);
    }

    vis[src] = false;
    psf.remove(psf.size() -1);
}
```

```java
public List<List<Integer>> allPathsSourceTarget(int[][] graph) {
    List<Integer> psf = new ArrayList<>();
    res = new ArrayList<>();
    boolean[] vis = new boolean[graph.length];

    DFS(0,graph.length-1,graph,vis,psf);
    return res;
}
```

Global ArrayList Method

TC: O(exponential)

$(outdegree)^N$

flood fill

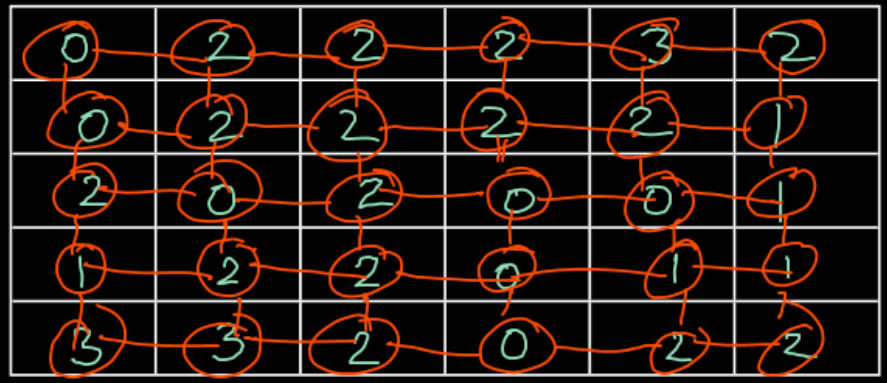
0	1 2	1 2	2	3	2
0	1 2	1 2	1 2	2	1
2	0	1 2	0	0	1
1	2	1 2	0	1	1
3	3	1 2	0	2	2

Marked cell should be colored 2

So, all the neighbors with same color(1) will also be painted 2

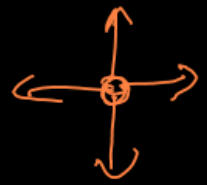
The matrix is not adj Matrix or adj list or edge list. Every ele is a node that has edges with its 4 directional neighbours.

This is graph representation.



Neighbors are 4-way

neighbors



We will not make a seperate visited array.

```java
public void DFS(int[][] image,int sr, int sc,int newColor,int startColor) {
    if(sr < 0 || sc < 0 || sr >= image.length || sc >= image[0].length) return;

    if(image[sr][sc] != startColor) return;

    image[sr][sc] = newColor;

    DFS(image,sr -1,sc,newColor,startColor);
    DFS(image,sr,sc + 1,newColor,startColor);
    DFS(image,sr + 1,sc,newColor,startColor);
    DFS(image,sr,sc - 1,newColor,startColor);
}

public int[][] floodFill(int[][] image, int sr, int sc, int color) {
    if(image[sr][sc] == color) return image;
    DFS(image,sr,sc,color,image[sr][sc]);
    return image;
}
}
```

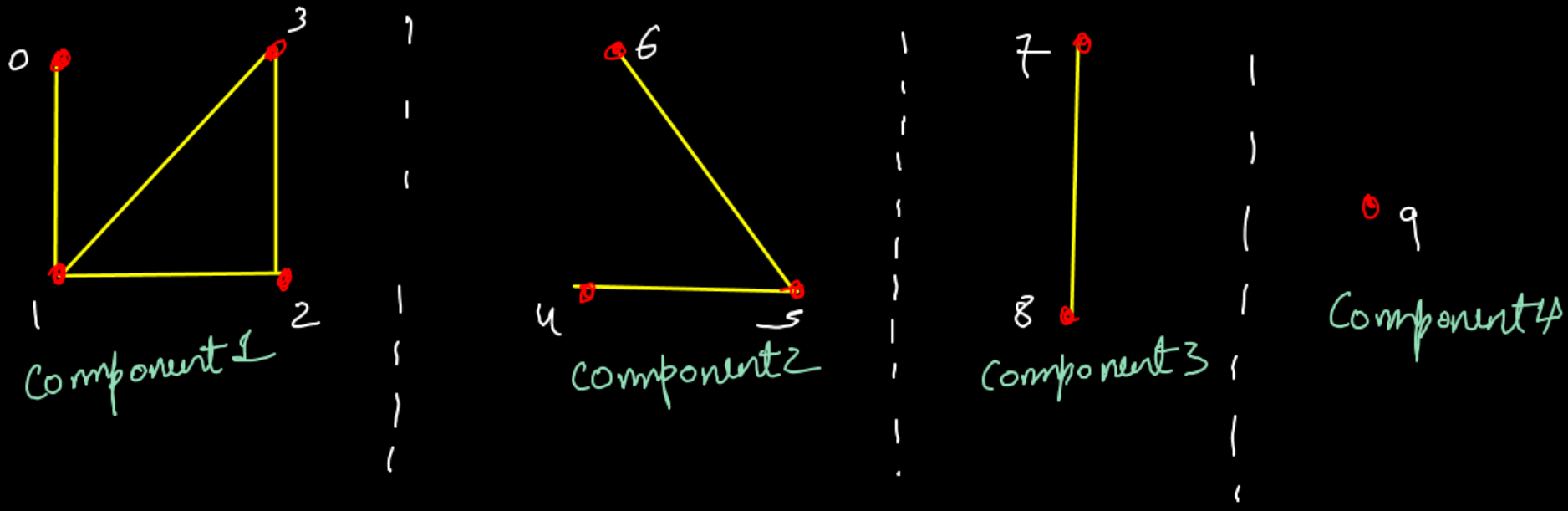
Time: O(N * M)

Space: O(1)

Recursion call Stack Space (worst case) : O(N * M)

Get Connected Components

Connected comp : only for directed graph

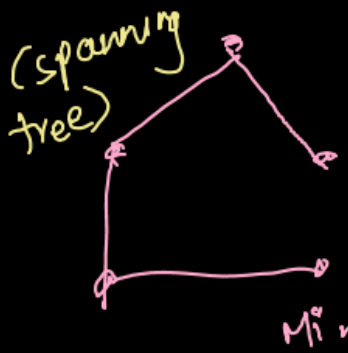


A component is a subgraph in which from every node, you can reach every other node (directly or indirectly)

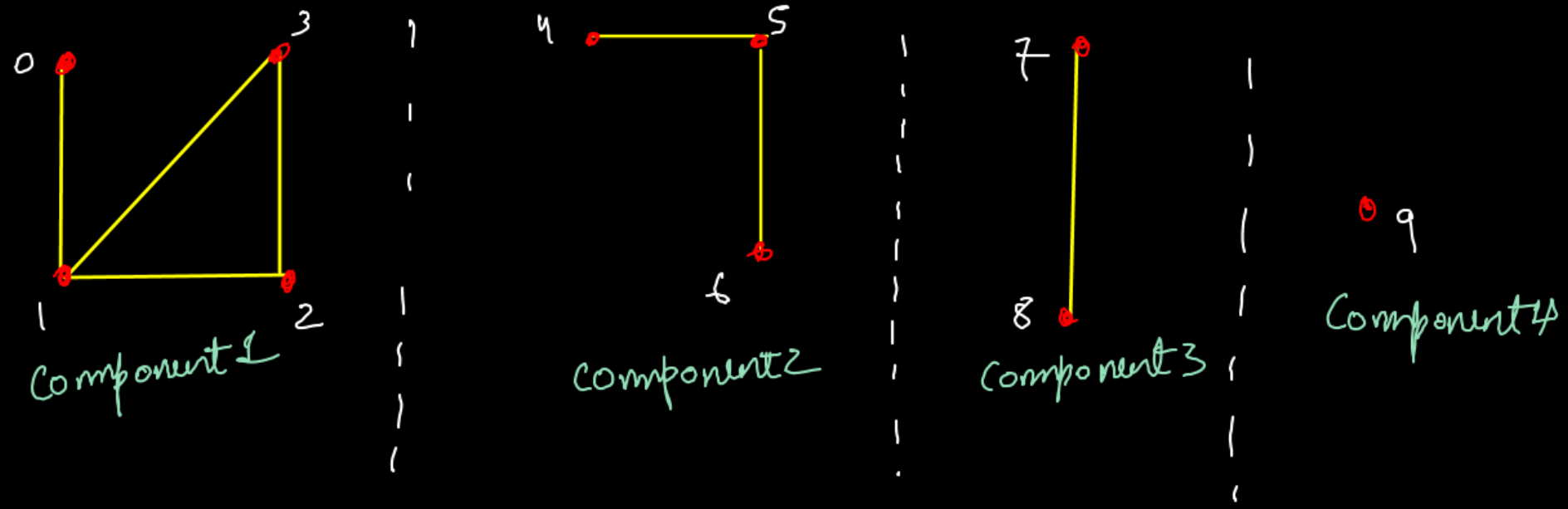
If a graph has N nodes. What are the min and max no of edges to make it connected?

N nodes → Min → $(N-1)$

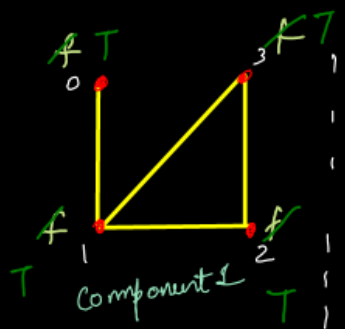
N nodes → Max → $\frac{N(N-1)}{2}$ { Make the graph a complete graph }

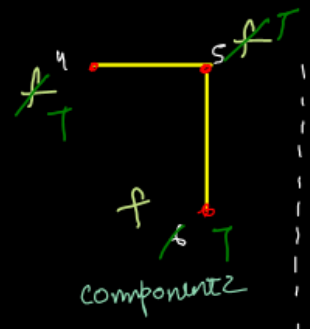


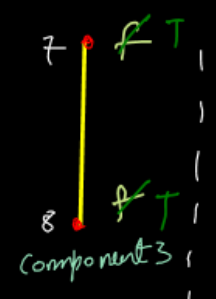
Max

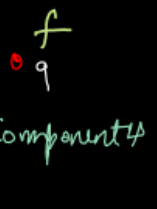


MultiSource – DFS → Apply dfs from all the nodes. Only the unvisited ones will be visited.

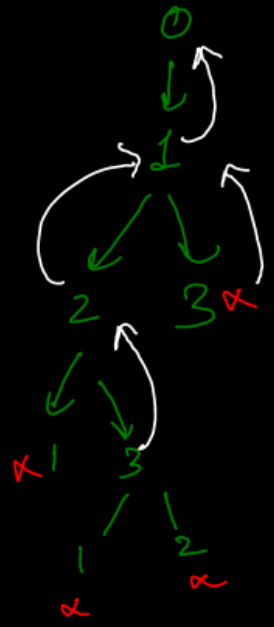

0
3
1
2
Component 1

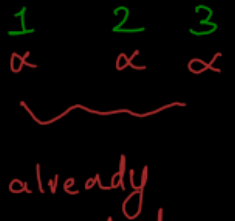

4
5
6
Component 2

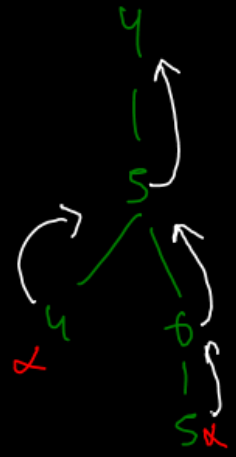

7
8
Component 3

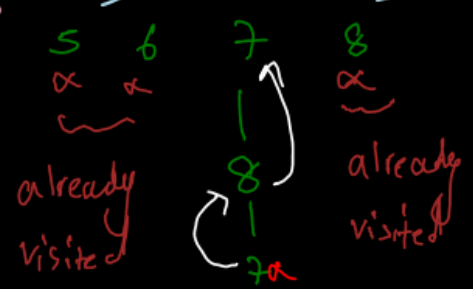

9
Component 4

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


0
1
2
3
1
2
3


1
2
3
already visited


4
5
4
6
5


5
6
7
8
9
already visited
already visited
8
7

```java
public static void DFS(ArrayList<Edge>[] graph, int src,boolean[] vis,List<Integer> path) {
    vis[src] = true;
    path.add(src);

    for(Edge e : graph[src]) {
        if(vis[e.nbr] == false) {
            DFS(graph,e.nbr,vis,path);
        }
    }
}

public static void connectedComponents(ArrayList<Edge>[] graph) {
    int n = graph.length;
    boolean[] vis = new boolean[n];

    List<List<Integer>> comps = new ArrayList<>();

    for(int i=0;i<n;i++) {
        if(vis[i] == false) {
            List<Integer> comp = new ArrayList<>();
            DFS(graph,i,vis,comp);
            comps.add(comp);
        }
    }

    System.out.println(comps);
}
```

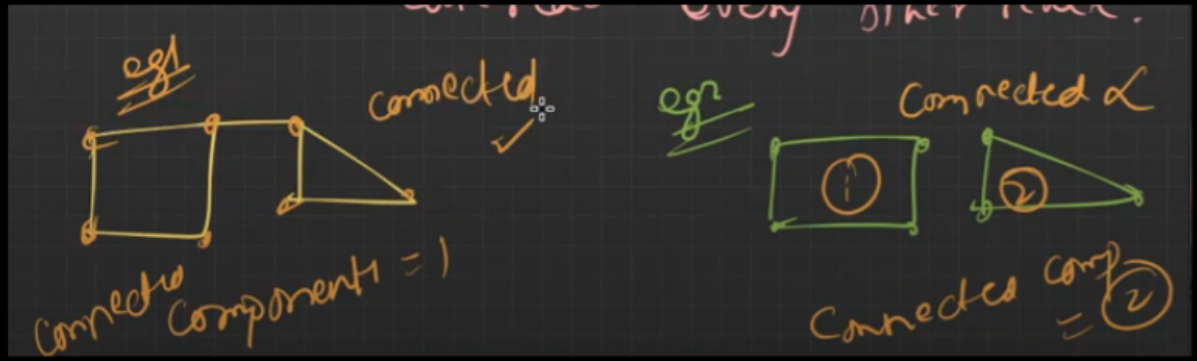
TC: O(N+E)

SC: O(1)

Is Graph Connected ?

→ A graph will be connected if the no of connected components in the graph is 1.

from every node of graph, we can reach every other node.



```java
public static void DFS(ArrayList<Edge>[] graph,int src,boolean[] vis,List<Integer> comp) {
    if(vis[src] == true) return;

    vis[src] = true;
    comp.add(src);

    for(Edge e : graph[src]) {
        DFS(graph,e.nbr,vis,comp);
    }
}

public static boolean isConnected(ArrayList<Edge>[] graph) {
    int n = graph.length;
    boolean[] vis = new boolean[n];
    List<List<Integer>> comps = new ArrayList<>();

    for(int i=0;i<n;i++) {
        if(vis[i] == false) {
            List<Integer> comp = new ArrayList<>();
            DFS(graph,i,vis,comp);
            comps.add(comp);
        }
    }

    if(comps.size() == 1) return true;
    return false;
}
```

Is graph connected code.

Another method can be to start DFS from any node. If after the DFS some nodes remain unvisited, this means that the graph is not connected.

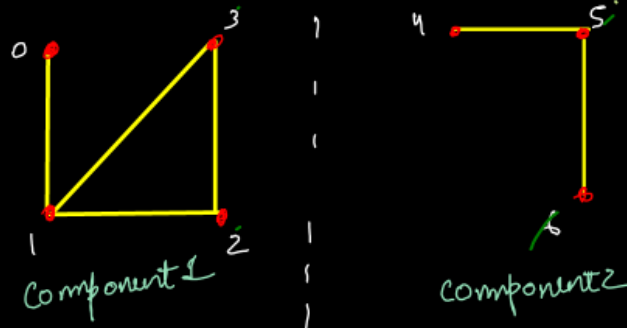
2316. Count Unreachable Pairs of Nodes in an Undirected Graph

Medium 280 6 Add to List Share

You are given an integer `n`. There is an **undirected** graph with `n` nodes, numbered from `0` to `n - 1`. You are given a 2D integer array `edges` where `edges[i] = [a_i, b_i]` denotes that there exists an **undirected** edge connecting nodes `a_i` and `b_i`.

Return *the **number of pairs** of different nodes that are **unreachable** from each other.*

This question is same as friends pairing qus.



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

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

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

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

2 ⇝ to reject (b, a) pairs when we have (a, b)

// $O(C^2)$ where C is the no of components.

$$\frac{4*(3+2+1) + 3*(4+2+1) + 2*(4+3+1) + 1*(4+3+2)}{2}$$

(+4−4, +3−3, +2−2, +1−1)

$$\Rightarrow \frac{4*(10-4) + 3*(10-3) + 2*(10-2) + 1*(10-1)}{2}$$

$$\Rightarrow \frac{\sum \text{component} * (N - \text{component})}{2}$$

} O(N) optimized

238. Product of Array Except Self

Medium 13114 767 Add to List Share

Given an integer array `nums`, return *an array* `answer` *such that* `answer[i]` *is equal to the product of all the elements of* `nums` *except* `nums[i]`.

The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in `O(n)` time and without using the division operation.

This is the question upon which prev optimization is based.

```java
class Graph {
    public ArrayList<Integer>[] adj;

    Graph(int n) {
        adj = new ArrayList[n];
        for(int i=0;i<n;i++) {
            adj[i] = new ArrayList<>();
        }

    }

    public void addEdge(int src, int dest) {
        adj[src].add(dest);
        adj[dest].add(src);
    }
}
```

```java
public long countPairs(int n, int[][] edges) {
    Graph g = new Graph(n);
    for(int[] edge : edges) {
        g.addEdge(edge[0],edge[1]);
    }

    boolean[] vis = new boolean[n];
    List<List<Integer>> comps = new ArrayList<>();

    long pairs = 0;
    for(int i=0;i<n;i++) {
        if(vis[i] == false) {
            List<Integer> comp = new ArrayList<>();
            DFS(i,g,vis,comp);
            pairs = pairs + ((comp.size()) * ((n * 11) - comp.size()));
        }
    }

    return pairs/21;

}
```

```java
public static void DFS(int src,Graph g,boolean[] vis, List<Integer> comp) {
    if(vis[src] == true) return;

    vis[src] = true;
    comp.add(src);

    for(int nbr : g.adj[src]) {
        DFS(nbr,g,vis,comp);
    }
}
```

Time → $O(N+E)$, Space → $O(N)+O(N)$
DFS
Visited Arr

Count of Islands

Islands { DFS / Connected Components }

1. Leetcode 200: No of Islands (No1-1)
2. Leetcode 547: No of provinces (No1-11)
3. Leetcode 1020: No of Enclaves (No1-111)
4. Hintcode: No of Distinct Islands

LC 463

6. Island Perimeter
6. Island Area

LC 695

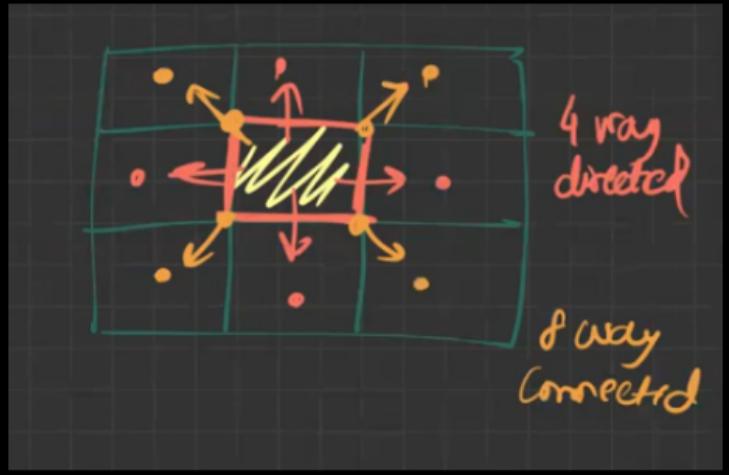
} Island questions variations

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

Land → 1s

Water → 0s

4 directional connection



6 comps / islands

Approach is to apply DFS as soon as we see a land. Visited will be marked in matrix itself.

```
    public void DFS(int r, int c, char[][] grid) {
        if(r < 0 || c < 0 || r >= grid.length || c >= grid[0].length || grid[r][c] == '0' || grid[r]
[c] == '2' ) return;

        grid[r][c] = '2';

        DFS(r -1,c,grid);
        DFS(r,c + 1,grid);
        DFS(r + 1,c,grid);
        DFS(r,c -1,grid);
    }
```

```
  public int numIslands(char[][] grid) {
      int res = 0;

      for(int i=0;i<grid.length;i++) {
          for(int j=0;j<grid[0].length;j++) {
              if(grid[i][j] == '1') {
                  DFS(i,j,grid);
                  res++;
              }
          }
      }

      return res;
  }
```

Tc: O(N*M)

Number of Provinces {Connected comps}

547. Number of Provinces

Medium 5477 245 Add to List Share

There are `n` cities. Some of them are connected, while some are not. If city `a` is connected directly with city `b`, and city `b` is connected directly with city `c`, then city `a` is connected indirectly with city `c`.

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an `n x n` matrix `isConnected` where `isConnected[i][j] = 1` if the i^{th} city and the j^{th} city are directly connected, and `isConnected[i][j] = 0` otherwise.

Return *the total number of* ***provinces***.

Here, the input is in the form of adj matrix. So, directly solve karne pe TC high hogi. We will first convert it into adj list & then solve.

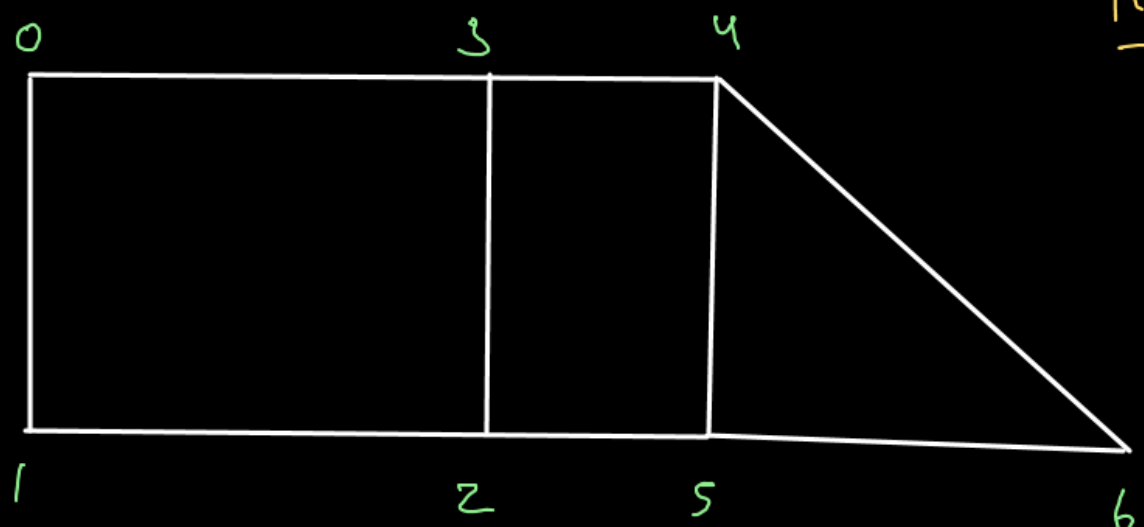
```
class Graph {
    public ArrayList<Integer>[] adj;

    Graph(int n) {
        adj = new ArrayList[n];
        for(int i=0;i<n;i++) {
            adj[i] = new ArrayList<>();
        }

    }

    public void addEdge(int src, int dest) {
        adj[src].add(dest);
        adj[dest].add(src);
    }
}
```

TC: O(N+E + N^2)

```
public int findCircleNum(int[][] isConnected) {
    int n = isConnected.length;

    Graph g = new Graph(n);

    for(int i=0;i<n;i++) {
        for(int j=0;j<n;j++) {
            if(i != j && isConnected[i][j] == 1) {
                g.addEdge(i,j);
            }
        }
    }

    boolean[] vis = new boolean[n];
    List<List<Integer>> comps = new ArrayList<>();

    for(int i=0;i<n;i++) {
        if(vis[i] == false) {
            List<Integer> comp = new ArrayList<>();
            DFS(i,g,vis,comp);
            comps.add(comp);
        }
    }

    return comps.size();
}
```

```
public static void DFS(int src,Graph g,boolean[] vis, List<Integer> comp) {
    if(vis[src] == true) return;

    vis[src] = true;
    comp.add(src);

    for(int nbr : g.adj[src]) {
        DFS(nbr,g,vis,comp);
    }
}
```

Hamiltonian Path and Cycle



Hamiltonian Path: All nodes are visited exactly one time.

Hamiltonian Cycle: If in a hamiltonian path, there is an edge b/w src & dest (last node in the path), then it is a cycle.

Source = 0

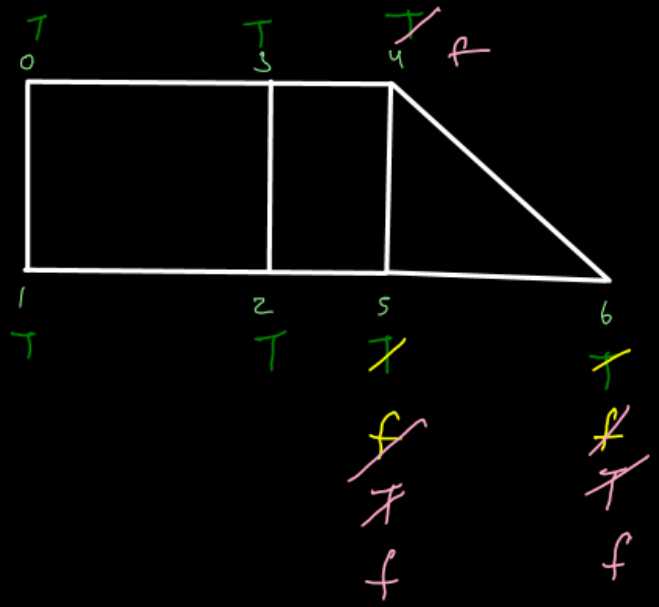
0 1 2 3 4 5 6 Path (.)

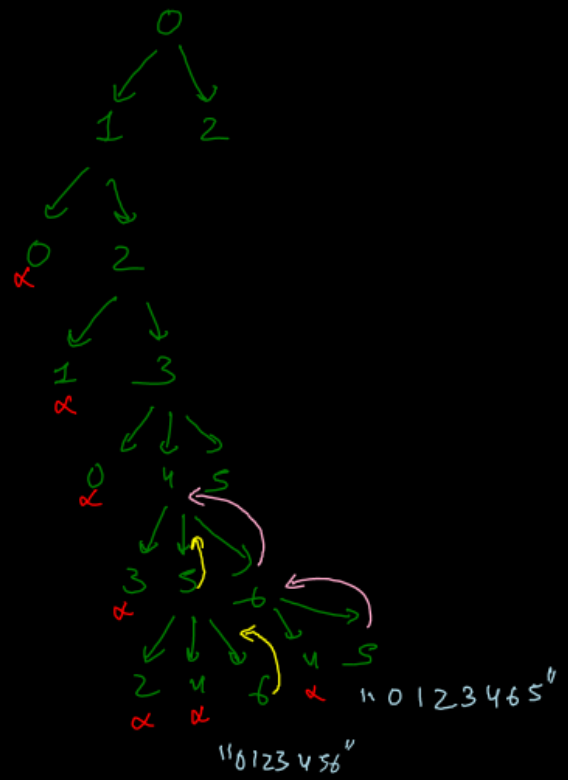
0 1 2 3 4 6 5 Path (.)

0 1 2 5 6 4 3 Cycle (*)

0 3 4 6 5 2 1 Cycle (*)

finding just any 1 hamiltonian path is also an exponential problem as we will have to necessarily backtrack to get a hamiltonian path. {N-P Problem}





```java
public static boolean isEdge(int src, int dest,ArrayList<Integer>[] graph) {
    for(Integer nbr : graph[src]) {
        if(nbr == dest) return true;
    }
    return false;
}

public static void DFS(int src, boolean[] vis, ArrayList<Integer>[] graph,int visCount,String path) {
    if(vis[src] == true) return;

    vis[src] = true;
    visCount++;

    for(int nbr : graph[src]) {
        DFS(nbr,vis,graph,visCount,path + nbr);
    }

    if(visCount == graph.length) {
        System.out.print(path);
        int a = path.charAt(0) - '0';
        if(isEdge(a,src,graph) == true) {
            System.out.println("*");
        } else {
            System.out.println(".");
        }
    }

    vis[src] = false;
}
```

* A euler graph is one in which all edges are visited exactly once.

mutable / heap

backtrack karvana hai.

Stack / Immutable

backtrack nahi karna

TC: Exponential

Space: O(N)

```java
    // write all your codes here
    boolean[] vis = new boolean[graph.length];
    DFS(src,vis,graph,0,src + "");
}
```

1376. Time Needed to Inform All Employees

Medium 1987 127 Add to List Share

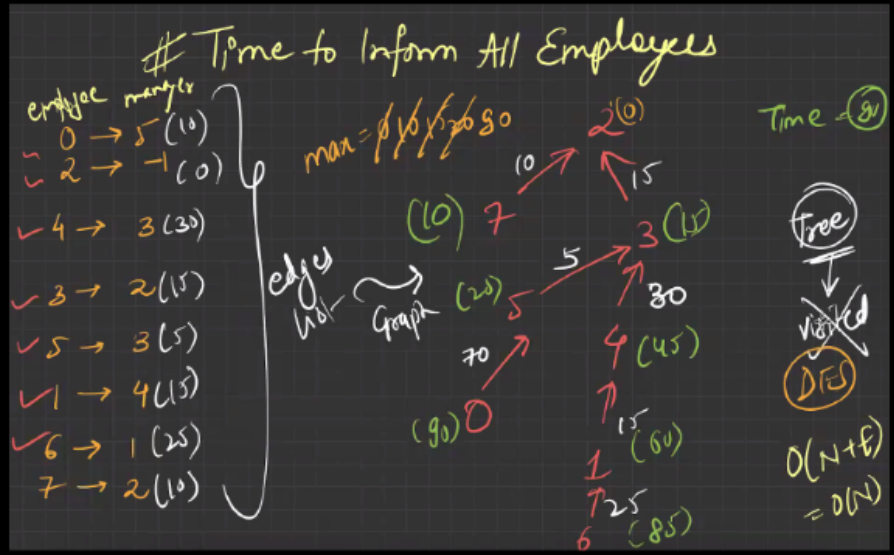
A company has `n` employees with a unique ID for each employee from `0` to `n - 1`. The head of the company is the one with `headID`.

Each employee has one direct manager given in the `manager` array where `manager[i]` is the direct manager of the `i-th` employee, `manager[headID] = -1`. Also, it is guaranteed that the subordination relationships have a tree structure.

The head of the company wants to inform all the company employees of an urgent piece of news. He will inform his direct subordinates, and they will inform their subordinates, and so on until all employees know about the urgent news.

The `i-th` employee needs `informTime[i]` minutes to inform all of his direct subordinates (i.e., After informTime[i] minutes, all his direct subordinates can start spreading the news).

Return *the number of minutes* needed to inform all the employees about the urgent news.



```java
class Pair {
    int nbr;
    int wt;

    Pair(int nbr, int wt) {
        this.nbr = nbr;
        this.wt = wt;
    }
}

class Graph {
    ArrayList<Pair>[] adj;

    Graph(int n) {
        adj = new ArrayList[n];
        for(int i=0;i<n;i++) {
            adj[i] = new ArrayList<>();
        }
    }

    public void addEdge(int src, int nbr,int wt) {
//          Directed and Weighted
        adj[src].add(new Pair(nbr,wt));
    }
}
```

```java
    public int DFS(int src,int time,Graph g) {
        int maxTime = time;
        for(Pair p : g.adj[src]) {
            maxTime = Math.max(maxTime,DFS(p.nbr,time + p.wt,g));
        }

        return maxTime;
    }

    public int numOfMinutes(int n, int headID, int[] manager, int[] informTime) {
        Graph g = new Graph(n);
        int time = 0;

        for(int i=0;i<n;i++) {
            //i-> child and array[i] is parent
            if(manager[i] == -1) {
                time = informTime[i];
            } else {
                g.addEdge(manager[i],i,informTime[i]);
            }
        }

        return DFS(headID,time,g);
    }
}
```

Time: O(N+E)

Space: O(N+E) → Graph

1319. Number of Operations to Make Network Connected

Medium 2323 34 Add to List Share

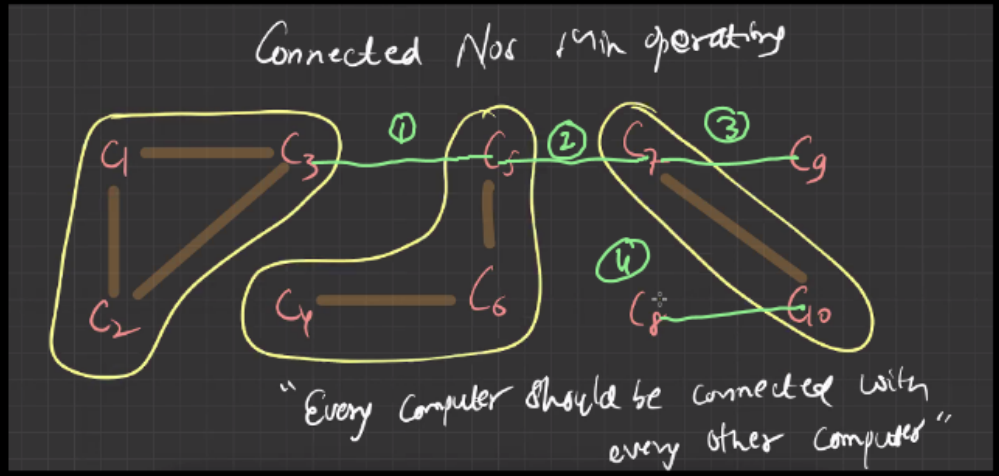
There are `n` computers numbered from `0` to `n - 1` connected by ethernet cables `connections` forming a network where `connections[i] = [a_i, b_i]` represents a connection between computers `a_i` and `b_i`. Any computer can reach any other computer directly or indirectly through the network.

You are given an initial computer network `connections`. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected.

Return *the minimum number of times you need to do this in order to make all the computers connected.* If it is not possible, return `-1`.

Minimum no of wires so that every computer must be connected to every other computer.

wires = connected components − 1



```java
class Graph {
    ArrayList<Integer>[] adj;

    Graph(int n) {
        adj = new ArrayList[n];

        for(int i=0;i<n;i++) {
            adj[i] = new ArrayList<>();
        }
    }

    public void addEdge(int src, int dest) {
        adj[src].add(dest);
        adj[dest].add(src);
    }
}
```

```java
public int countComps(Graph g) {
    int n = g.adj.length;

    boolean[] vis = new boolean[n];
    List<List<Integer>> comps = new ArrayList<>();
    int count = 0;
    for(int i=0;i<n;i++) {
        if(vis[i] == false) {
            List<Integer> comp = new ArrayList<>();
            DFS(g,i,vis,comp);
            comps.add(comp);
        }
    }

    System.out.println(comps);
    return comps.size();
}
```

```java
public void DFS(Graph g, int src, boolean[] vis,List<Integer> comp) {
    if(vis[src] == true) return;

    vis[src] = true;
    comp.add(src);

    for(int nbr : g.adj[src]) {
        DFS(g,nbr,vis,comp);
    }
}
```

```java
public int makeConnected(int n, int[][] connections) {
    Graph g = new Graph(n);

    //min n-1 edges should be available
    if(connections.length < n-1) return -1;

    for(int[] edge : connections) {
        g.addEdge(edge[0],edge[1]);
    }

    int cmps = countComps(g);
    return cmps -1;
}
```

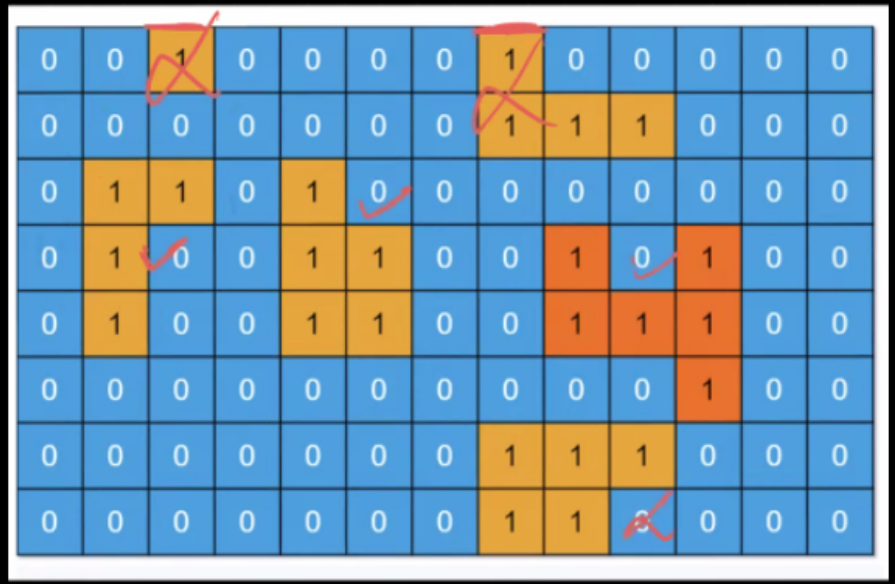
1020. Number of Enclaves

Medium 1399 35 Add to List Share

You are given an `m x n` binary matrix `grid`, where `0` represents a sea cell and `1` represents a land cell.

A **move** consists of walking from one land cell to another adjacent (**4-directionally**) land cell or walking off the boundary of the `grid`.

Return *the number of land cells in* `grid` *for which we cannot walk off the boundary of the grid in any number of* ***moves***.



Approach : Jitne islands boundary ko touch kar rahe hai unhe -1 bana do. Fir prev algo chala do.

So, there are 3 islands that are completely surrounded by water

```java
public void DFS(int[][] grid, int r, int c) {
    if(r < 0 || c < 0 || r >= grid.length || c >= grid[0].length || grid[r][c] == 0 || grid[r][c] == -1) return;

    grid[r][c] = -1;

    DFS(grid,r -1,c);
    DFS(grid,r,c + 1);
    DFS(grid,r + 1,c);
    DFS(grid,r,c -1);
}
```

```java
//for 0th row
for(int j=0;j<cols;j++) {
    if(grid[0][j] == 1) {
        DFS(grid,0,j);
    }
}

//for last col
for(int i=0;i<rows;i++) {
    if(grid[i][cols -1] == 1) {
        DFS(grid,i,cols-1);
    }
}

//for last row
for(int j=cols-1;j>=0;j--) {
    if(grid[rows-1][j] == 1) {
        DFS(grid,rows-1,j);
    }
}

//for first column
for(int i=rows-1;i>=0;i--) {
    if(grid[i][0] == 1) {
        DFS(grid,i,0);
    }
}
```

DFS on land touching the borders

```java
public int numEnclaves(int[][] grid) {

    int rows = grid.length;
    int cols = grid[0].length;

    int landCount = 0;

    for(int i=0;i<rows;i++) {
        for(int j=0;j<cols;j++) {
            if(grid[i][j] == 1) landCount++;
        }
    }
```

Count of 1s before DFS

```java
    int visCount = 0;
    for(int i=0;i<rows;i++) {
        for(int j=0;j<cols;j++) {
            if(grid[i][j] == -1) visCount++;
        }
    }

    return landCount - visCount;
```

initial no of 1s − no of (-1s & visited)

417. Pacific Atlantic Water Flow

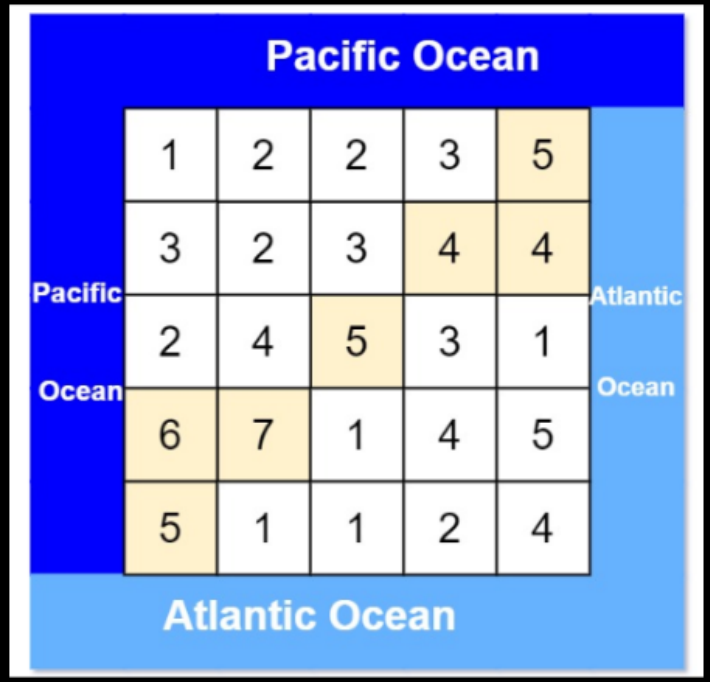
Medium 4041 894 Add to List Share

There is an `m x n` rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**. The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an `m x n` integer matrix `heights` where `heights[r][c]` represents the **height above sea level** of the cell at coordinate `(r, c)`.

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is **less than or equal to** the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return *a **2D list** of grid coordinates* `result` *where* `result[i] = [ri, ci]` *denotes that rain water can flow from cell* `(ri, ci)` *to **both** the Pacific and Atlantic oceans.*



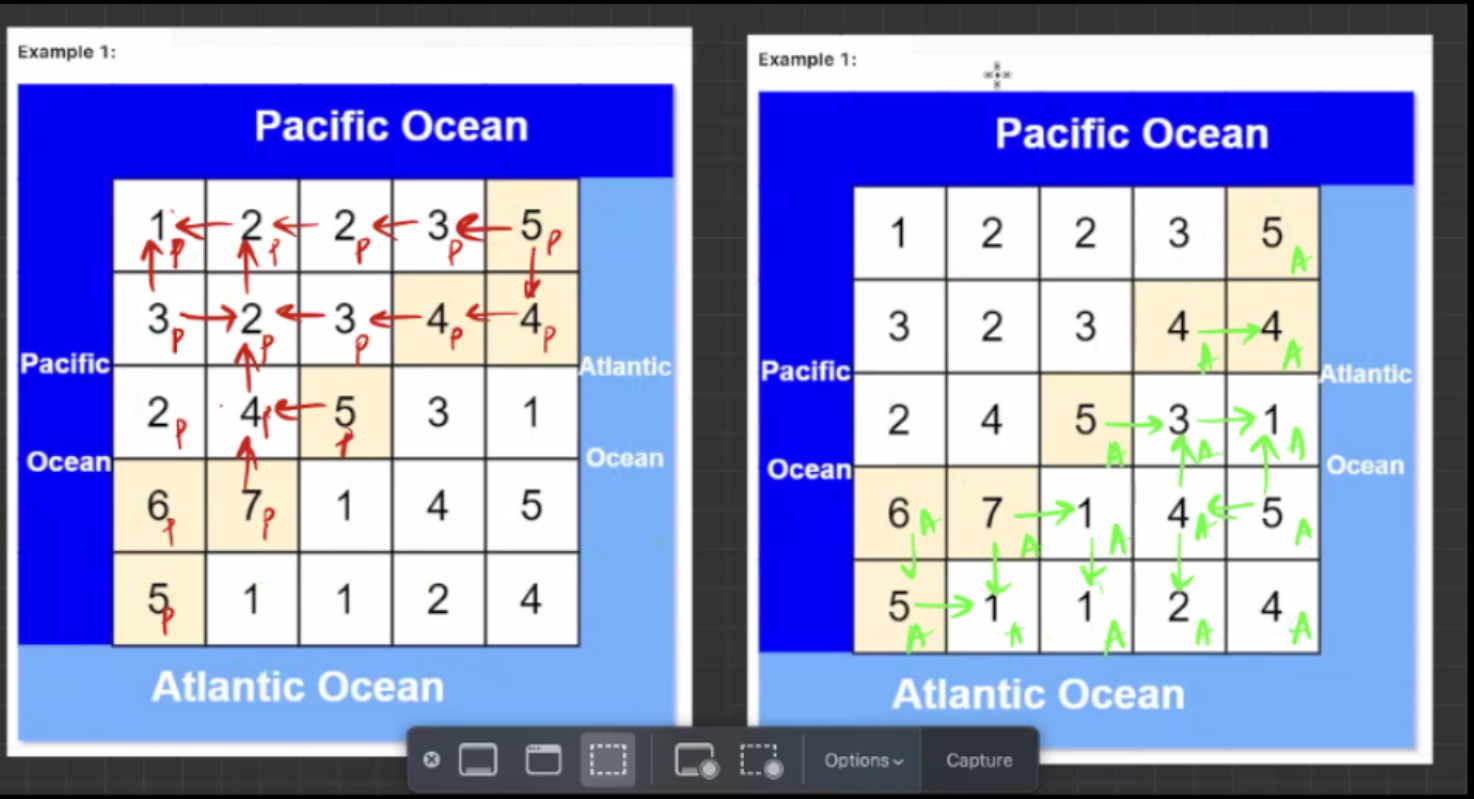
So, we have to return the coordinates of all cell that go to both pacific and atlantic.

The approach is simple. Here, we apply Dfs from dest to source. We will maintain 1 boolean array for pacific ocean & other for atlantic ocean. At last, the indices where there will be a true value for both the oceans will be our answer. {Dfs is applied on borders.}

```java
public void DFS(int[][] heights,int r, int c,boolean[][] vis) {
    if(r < 0 || c < 0 || r >= vis.length || c >= vis[0].length || vis[r][c] == true) return;

    vis[r][c] = true;
    //r -1
    if(r - 1 >= 0 && heights[r-1][c] >= heights[r][c]) {
        DFS(heights,r -1,c,vis);
    }
    //c + 1
    if(c + 1 < heights[0].length && heights[r][c + 1] >= heights[r][c]) {
        DFS(heights,r,c + 1,vis);
    }
    //r + 1
    if(r + 1 < heights.length && heights[r+1][c] >= heights[r][c]) {
        DFS(heights,r + 1,c,vis);
    }

    //c -1
    if(c - 1 >= 0 && heights[r][c - 1] >= heights[r][c]) {
        DFS(heights,r,c - 1,vis);
    }
}
```

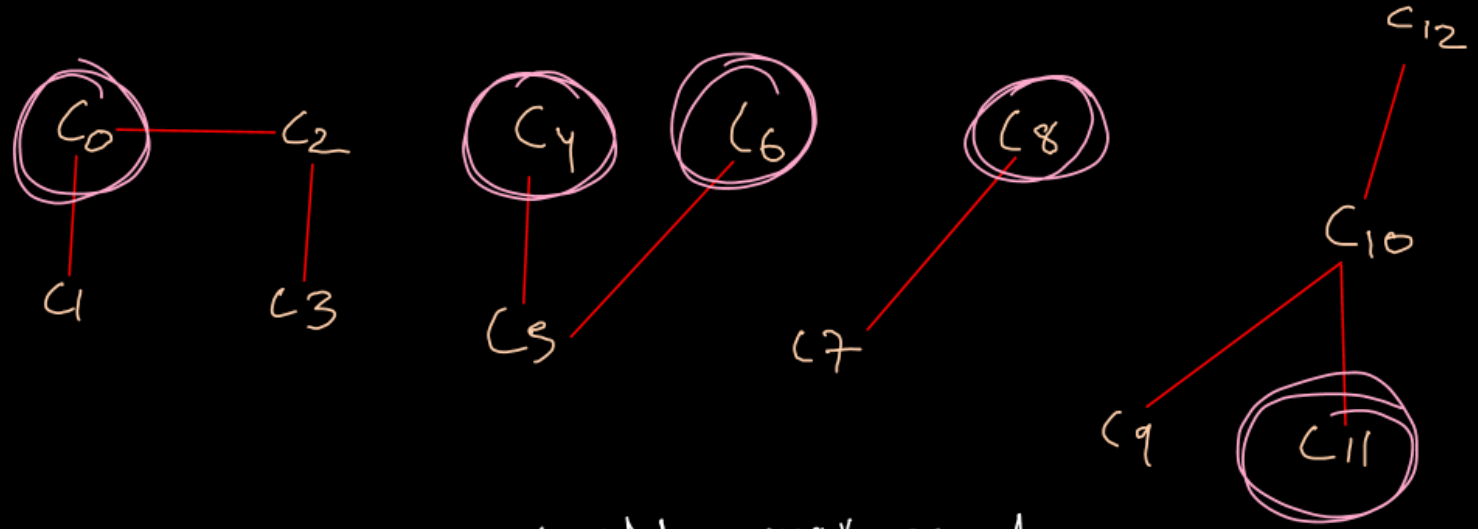
```java
public List<List<Integer>> pacificAtlantic(int[][] heights) {
    boolean[][] visPacific = new boolean[heights.length][heights[0].length];

    for(int j=0;j<heights[0].length;j++) {
        DFS(heights,0,j,visPacific);
    }

    for(int i=0;i<heights.length;i++) {
        DFS(heights,i,0,visPacific);
    }

    boolean[][] visAtlantic = new boolean[heights.length][heights[0].length];

    for(int j=0;j<heights[0].length;j++) {
        DFS(heights,heights.length-1,j,visAtlantic);
    }

    for(int i=0;i<heights.length;i++) {
        DFS(heights,i,heights[0].length-1,visAtlantic);
    }
```

```java
    List<List<Integer>> res = new ArrayList<>();
    for(int i=0;i<heights.length;i++) {
        for(int j=0;j<heights[0].length;j++) {
            if(visPacific[i][j] == true && visAtlantic[i][j] == true) {
                List<Integer> index = new ArrayList<>();
                index.add(i);index.add(j);
                res.add(index);
            }
        }
    }

    return res;
}
```



Time complexity is $O(N^2)$

Minimize Malware Spread



→ consider max count as answer.

Apply DFS on every infected node. Take its count only if no other node in that comp is infected.

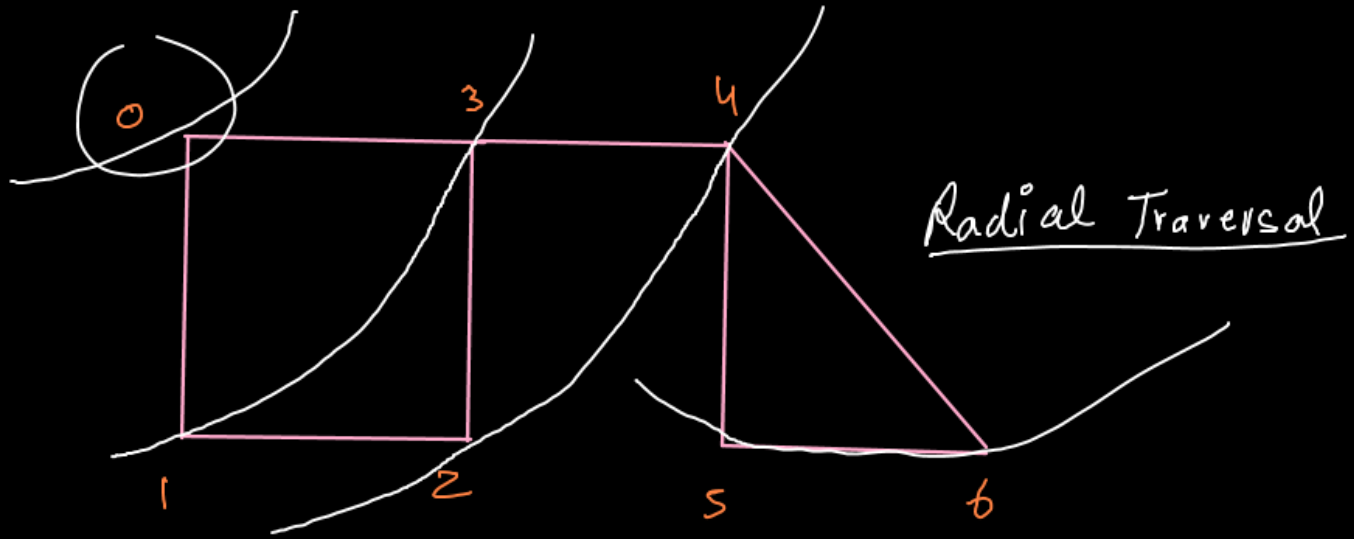
```java
static int size;
static int infected;

public void DFS(int[][] graph,int src, boolean[] vis,int[] initial) {
    if(vis[src] == true) return;

    vis[src] = true;
    size++;

    if(Arrays.binarySearch(initial,src) >= 0) {
        infected++;
    }

    for(int i=0;i<graph.length;i++) {
        if(graph[src][i] == 1) {
            DFS(graph,i,vis,initial);
        }
    }
}
```

```java
public int minMalwareSpread(int[][] graph, int[] initial) {
    Arrays.sort(initial);

    boolean[] vis = new boolean[graph.length];

    int maxSize = 0, node = initial[0];
    for(int src: initial) {
        size = 0;
        infected = 0;
        DFS(graph,src,vis,initial);

        if(size > maxSize && infected == 1) {
            node = src;
            maxSize = size;
        }
    }

    return node;
}
```

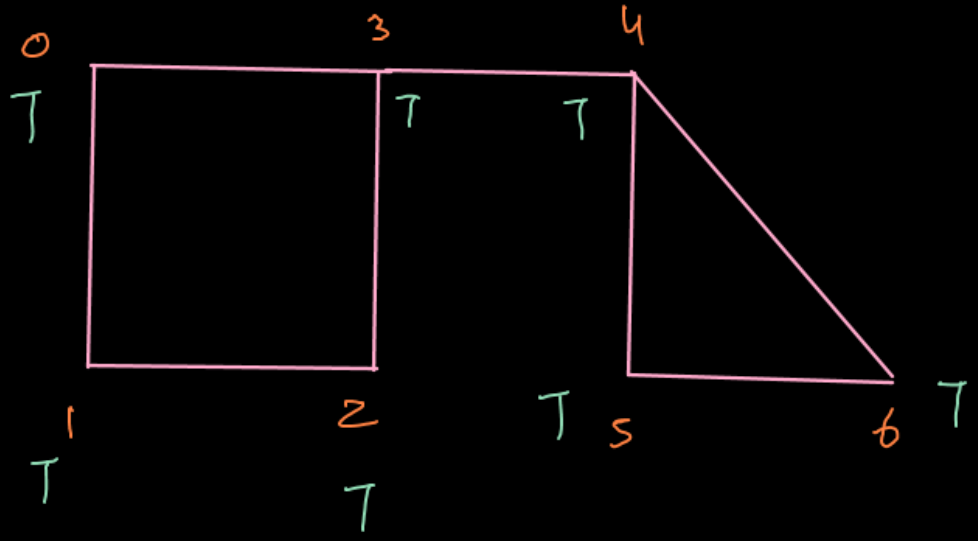
Space: O(N) Vis O(N) Recursion,

Time: $O(N+N^2)$

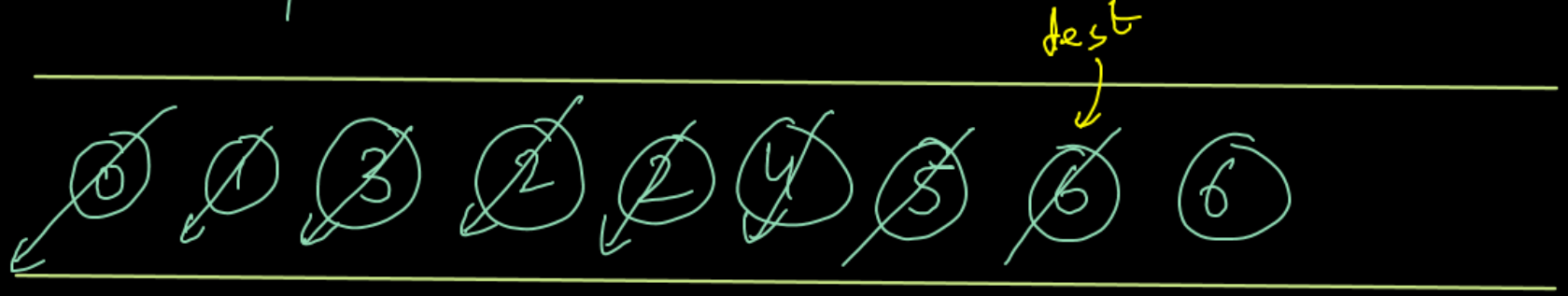
Breadth first Traversal



Radial Traversal



visited hum node ko pop karne k baad mark karenge.



```java
public boolean BFS(int src, int dest,ArrayList<Edge>[] graph) {
    Queue<Integer> q = new ArrayDeque<>();
    int n = graph.length;
    boolean[] vis = new boolean[n];

    q.add(src);

    while(q.size() > 0) {
        int front = q.remove();

        if(vis[front] == true) continue;
        if(front == dest) return true;
        vis[front] = true;

        for(Edge e : graph[front]) {
            if(vis[e.nbr] == false) {
                q.add(e.nbr);
            }
        }
    }

    return false;
}
```

O(N)

O(E)

Independent

So,

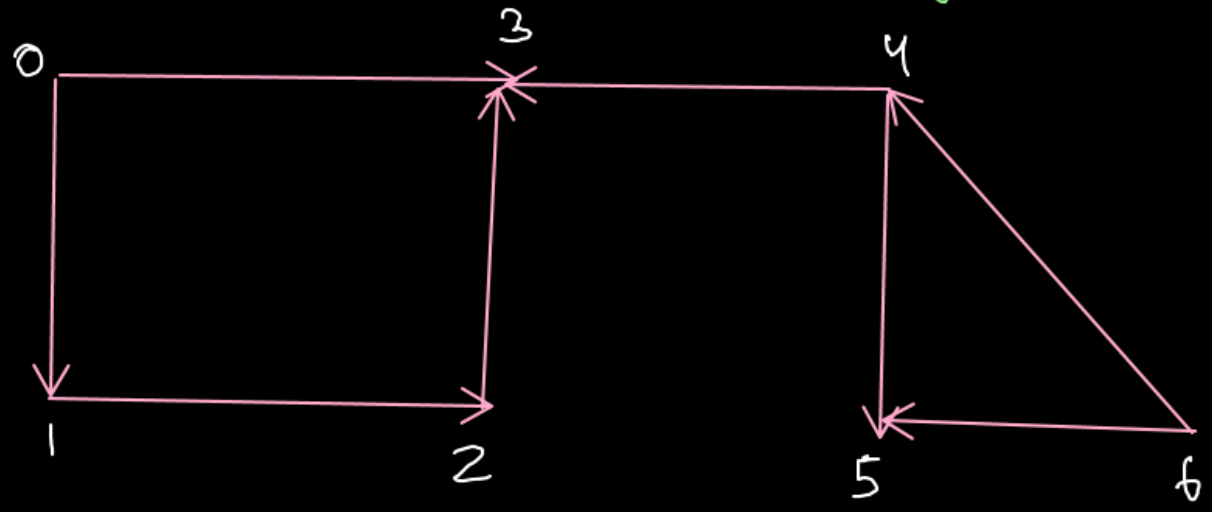
TC = O(N+E)

SC = O(N)

Topological Sort

- DFS (Recursive)
- BFS Kahn's Algorithm

Directed Acyclic Graph Only (Generic Tree w/o a root)



Example Topo Sorts

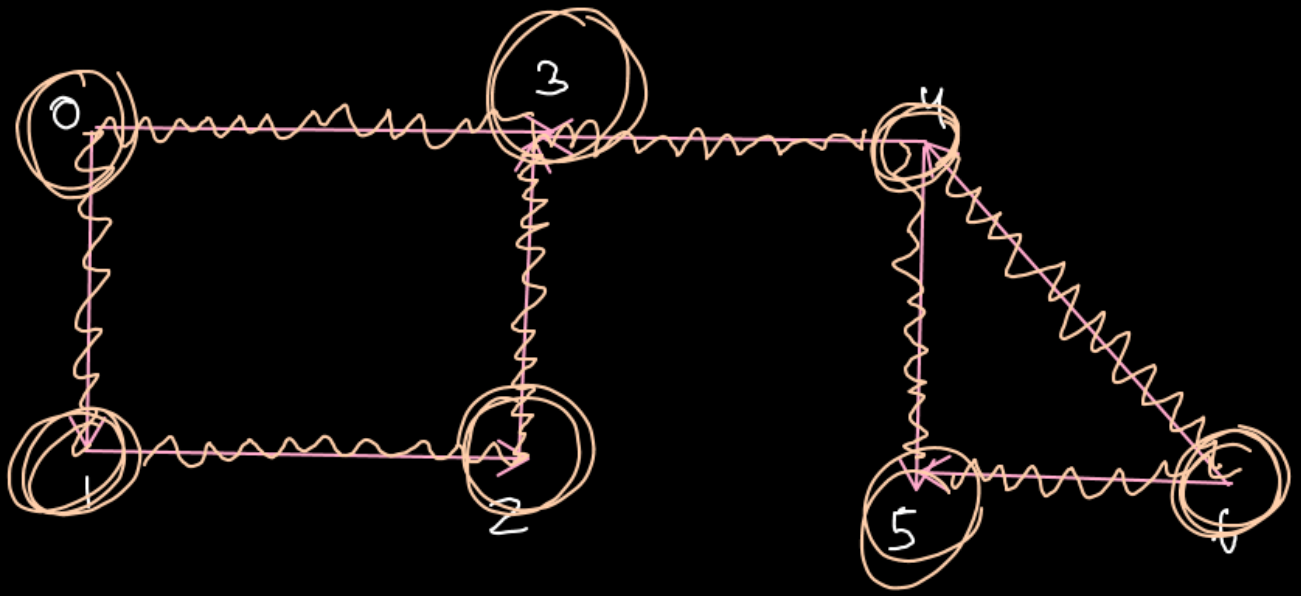
"6450123"

"0 1 2 6 4 3 5"

"0 6 1 2 4 3 5"

Topo Sort: Order of all the nodes such that for every edge (u, v), u must appear before v.

In topological sort, nodes are in increasing order of indegree.



"0 1 2 6 4 3 5"

Pick any node with 0 indegree. Then delete all its outgoing edges.

So, we have to take all nodes of indegree 0, then 1, then 2 & so on. So, we can apply BFS.