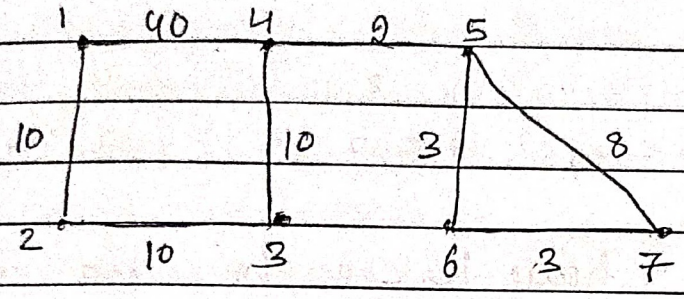
Graphs - Level 1

Graphs
↳ graph algorithms
↳ graph applications



① Nodes / vertices
② Edges
③ Undirected / directed
④ Weighted / Unweighted
⑤ Incoming edge / Outgoing edge

Applications:

① Google Maps
② Social Media (connections and friends)
③ Airline Management

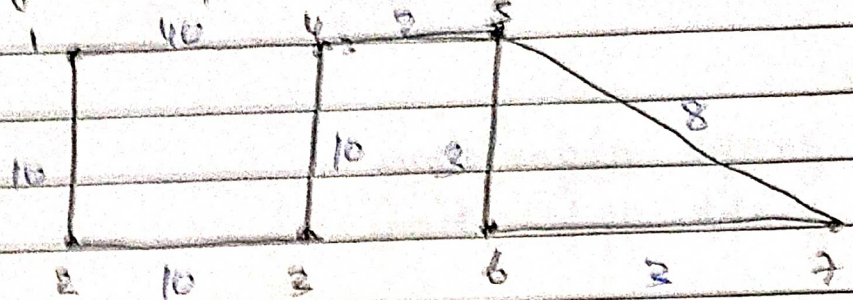
Implementation

① Edge - list {{1,2,10} {2,3,10} {1,4,40} {3,4,10} {4,5,2} .. {6,7,3}}
② Adjacency matrix
③ Adjacency list

⇒ Edge list is not a very good method because of the high complexity. In LC, we will mostly get the edge list only as the input. So, convert it to Adj Matrix or Adj. List.

Page No.
Date / / /

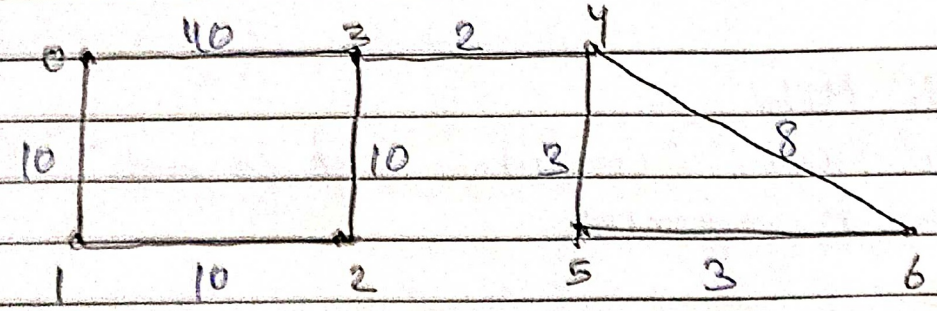
Edge List for the Graphs



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

Let we may I have to see the cities connecting to the city (vertex) 3. So, I will have to search in this list the vertex 3 including edge first. Hence high TC.

Adjacency Matrix {Rows = cols = Vertex}



	0	1	2	3	4	5	6
0	-1	10	-1	40	-1	-1	-1
1	10	-1	10	-1	-1	-1	-1
2	-1	10	-1	10	-1	-1	-1
3	40	-1	10	-1	2	-1	-1
4	-1	-1	-1	2	-1	3	8
5	-1	-1	-1	-1	3	-1	3
6	-1	-1	-1	-1	8	8	-1

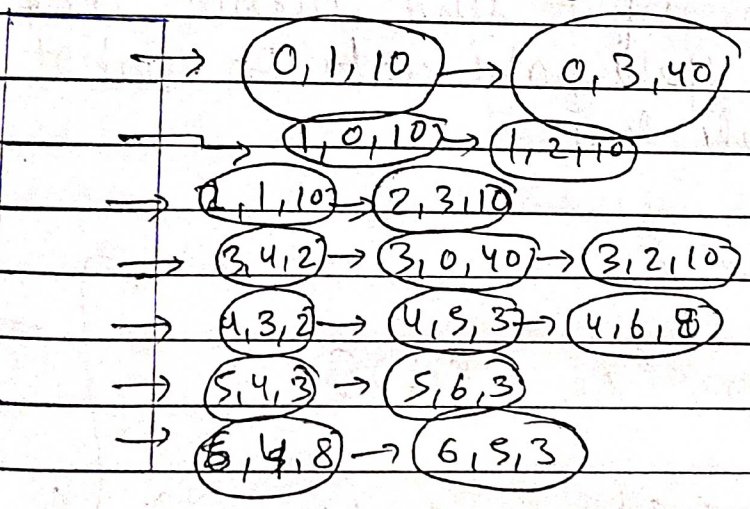
Vertices → n
edges (max) → $^nC_2 = \frac{n(n-1)}{2}$ } for undirected graph

So, if vertices is O(N) then edges is $O(N^2)$

Adjacency matrix also has a problem. Say I want edges from the vertex 3. I'll have to traverse the entire row with row index = 3.
However, we want ki jaise hum graph ko dekh kar bata pa rahe hai ki 0, 4 and 2 connected hai 3 se, vaise he hum store bhi karle.

Adjacency List

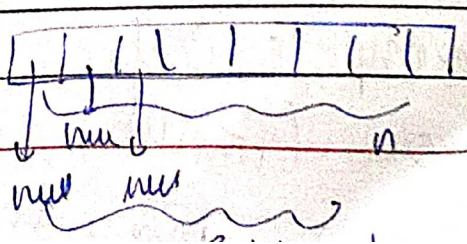
Make a bucket for every node (vertex)



** Very similar to HashMap.

class Edge {
int src;
int nbr;
int weight;
}

ArrayList<Edge>[] list
list = new ArrayList[V];



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

int edges = scn.nextInt();

while(edges-- > 0)
{
    int v1 = scn
    int v2 = scn
    int wt = scn
    graph[v1].add(new Edge(v1, v2, wt));
    graph[v2].add(new Edge(v2, v1, wt));
}

// Graph input lena bahot imp hai
// Agar yahi pahi aata to bahot problem
hone vali hai

display(){
    for(i=0; i<v; i++){
        for(Edge e: graph[i]){
            Print Edge
        }
    }
}
```

TC is O(E+V) Not O(V*E) because V vertices and total E edges are printed.

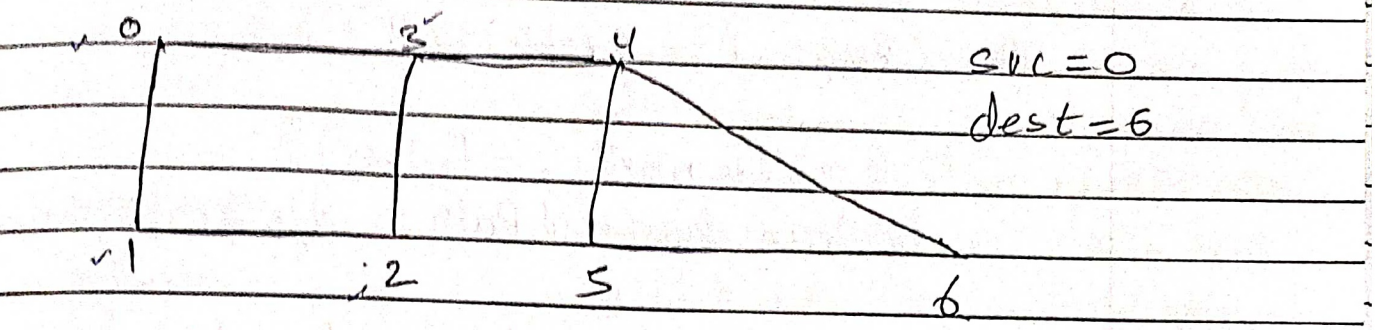
** Complete Code on GitHub

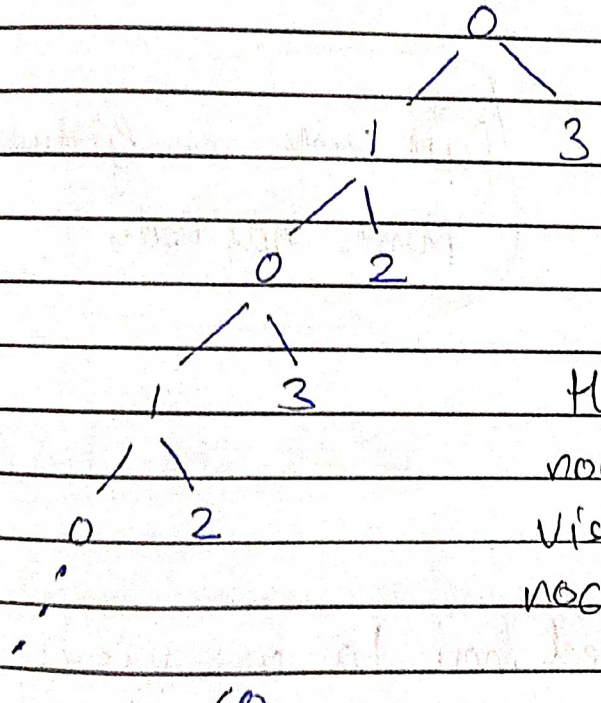
Next: Introduction to graphs

Classroom
Page No.
Date / / /

Ques Has Path

Source Vertex & dest vertex are given.
We have to check whether there is any path b/w source & destination.

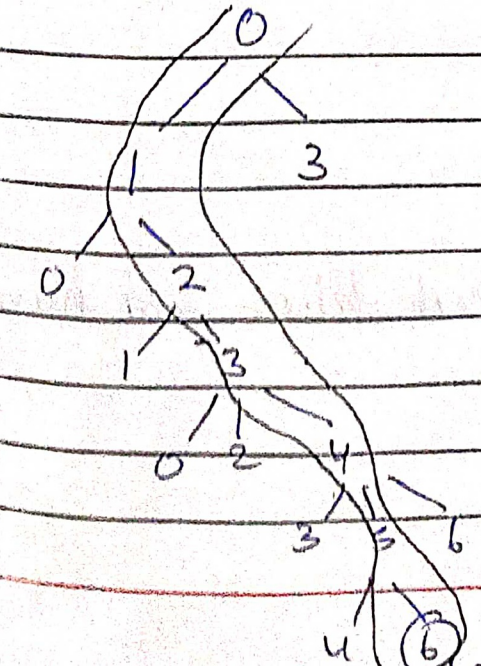




Infinite Recursion.

Hence, we will mark the node visited & will not visit the already visited node again.

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



dfs (graph, src, dest, vis)

```
dfs(graph, src, dest, vis) {
if (src == dest) return true;
vis[src] = true; //mark visited
for (Edge e : graph[src]) {
    if (vis[e.nbr] == false) {
        boolean hasChildPath = dfs(graph, e.nbr, de
                                   vis);
        if (hasChildPath == true) return true;
    }
}
return false;
}
```

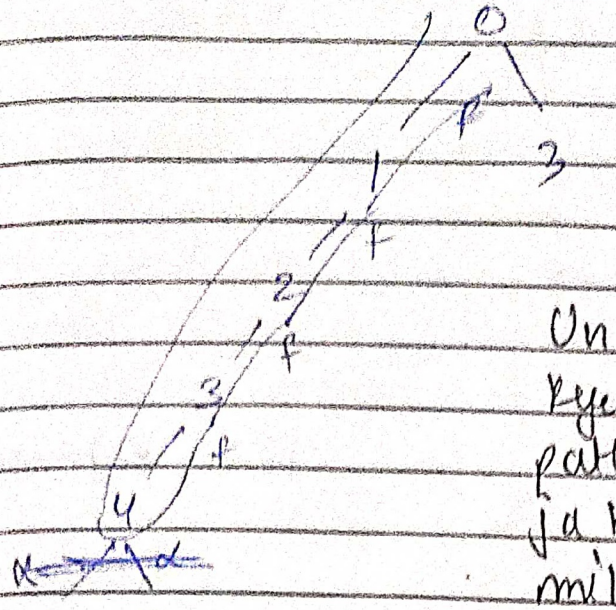
full code on Github
Name: has Path

⇒ Main baat

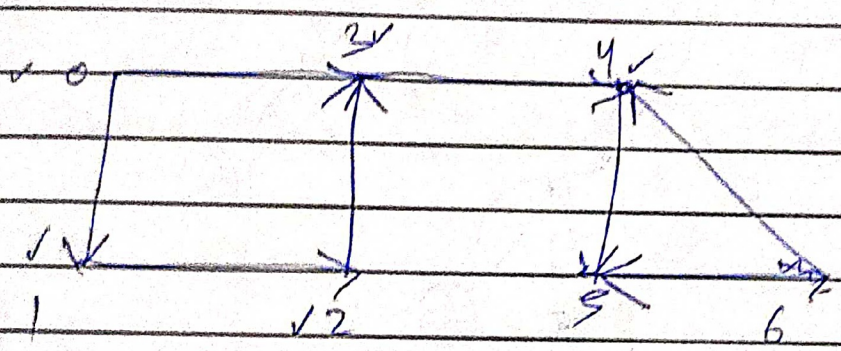
Dfs lagao.
Agar node visited hai to use visit nahi karna.

Doubt

Kya yaha backtrack kiya hai humne?
Nahi but why?



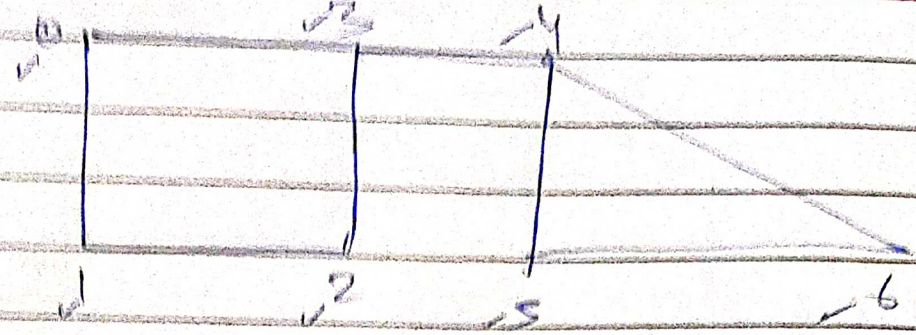
Unvisit isliye nahi kiya kyuki mmaan lo ek baar 3 se path nahi mila to dobara 3 pe jaa kya karoge, path nahi milega. Print all paths me backtrack karna padega.

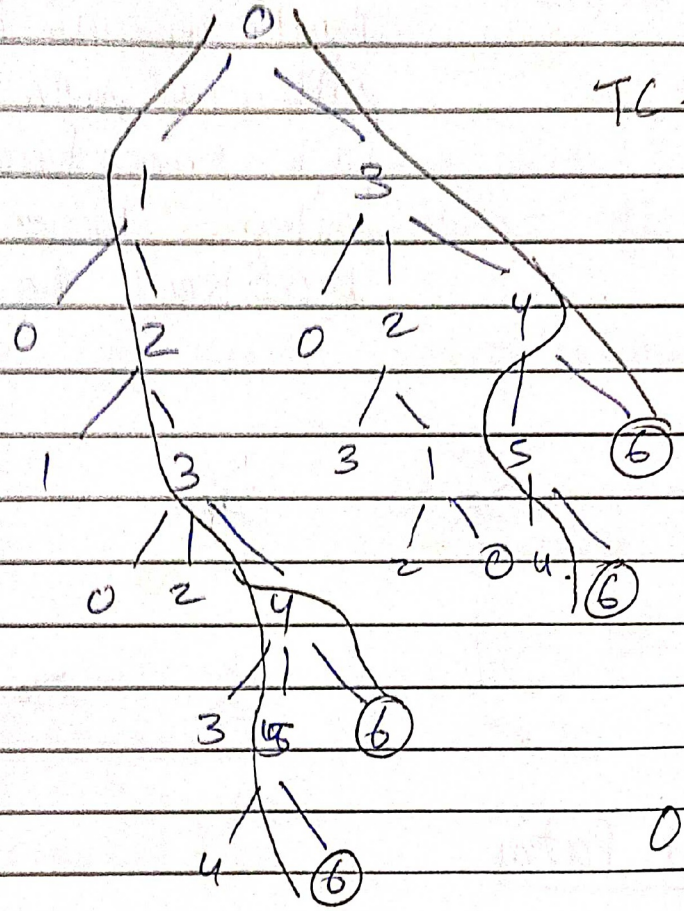


Print All Paths

Same algo with backtracking.

```
void dfs( graph, src, dest, vis, psf)
    if (src == dest) Print psf;
    vis[src] = true;
    for(Edge e: graph[src]) {
        only for unvisited  dfs( graph, e.nbr, dest, vis, psf+e.nbr);
    }
    vis[src] = false; backtracking
}
```





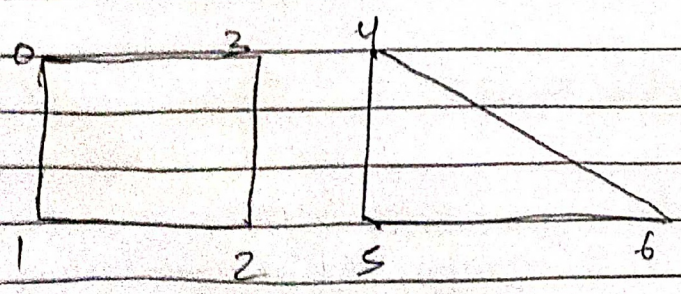
TC → Exponential

Kabhi bhi all paths polynomial time me nahi nikaal sakte (mostly)

$O(V^V)$

full code on Git hub Name: Print All Paths

Ques) Connected components (V Imp)

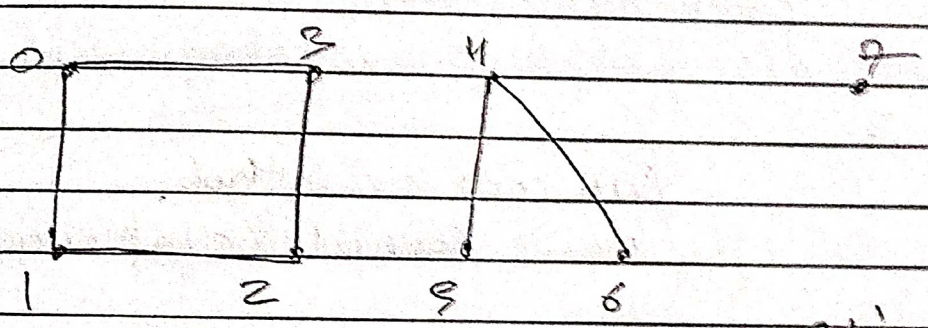


find no of connected components

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

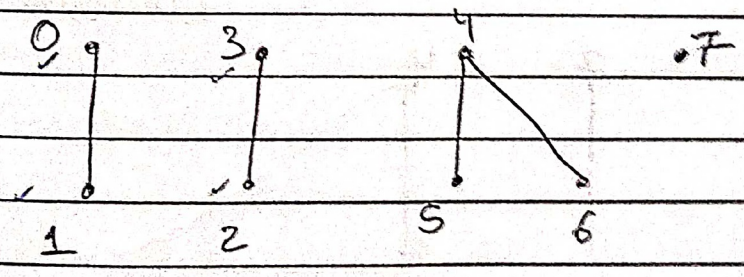
Connected Component Definition for Undirected graph.

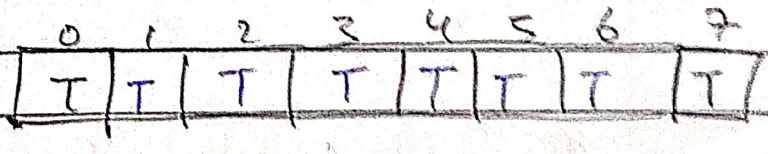
If from one vertex, we can reach every other vertex then all those vertices are part of one connected component.

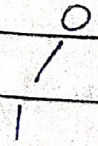


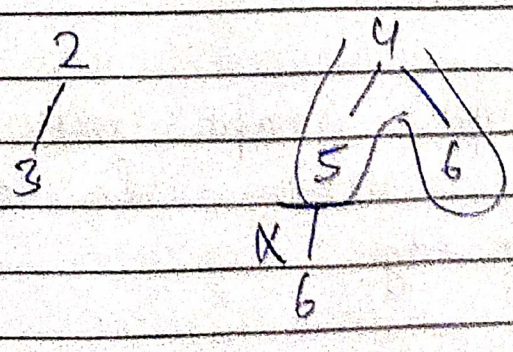
Still this is the case

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







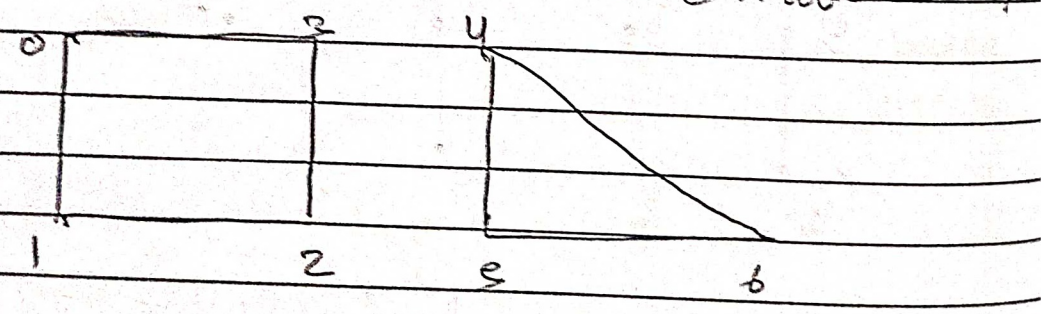


```
dfs(Graph, currentComp, src, vis) {
    // No base condition
    vis[src] = true;
    comp.add(e.nbr);
    for(Edge e : graph[src]) {
        if(vis[e.nbr] == false) {
            dfs(graph, currentComp, e.nbr, vis);
        }
    }
}
```

Applying this for every unvisited vertex in the vis array

Full code on Github
Name Connect Components

Ques. Is Graph Connected? (Variation of Connected Comp)



If no of component is greater than 1, it is not connected.

Minimum Graph: Graph with only one vertex. So, there will not be any edge.

Here, ans of no of comps is min 1 not 0.

No of comps → = 1 connected
→ > 1 disconnected.

So, just use previous algorithm and if the size of comps (list of list) is 1 then connected else not connected.

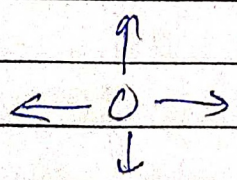
(full code on Github Name IS Graph Connected)

Ques Number of Islands (Leetcode: 200 Medium)
(and Pepcoding)
(Variation of Connected components)

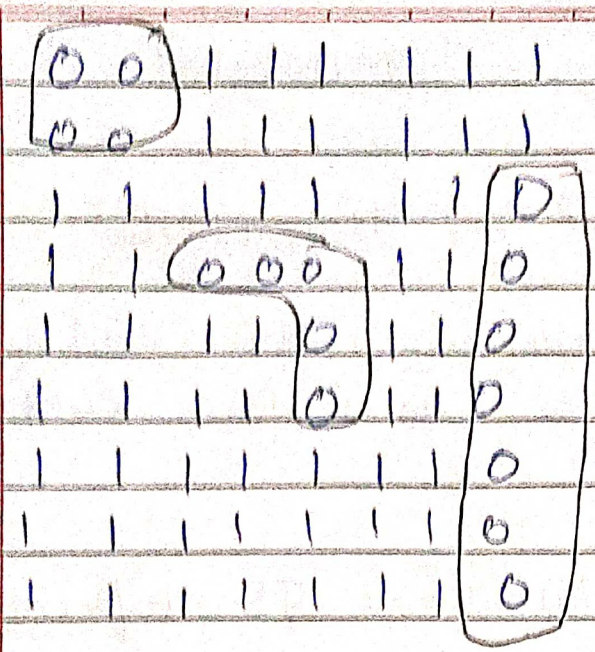
Sol Imp

Note: Matrix given to us is not an adjacency matrix.

An island is defined as '4 way connected zeroes



Again 8 way connection kaha ka diagonal se bhi connected kaha. Also, there is an assumption that outside the matrix, every where is water.



No of islands = 3

full code on GitHub No of Islands Repeat

```
dfs(arr, sr, sc) {

if (sr<0 || sr>row || sc<0 || sc>col) return;

// if already visited or water then also return
if (arr[sr][sc] == -1 || arr[sr][sc] == 0) return;

arr[sr][sc] = -1; (mark visited)

dfs(arr, sr-1, sc); up
dfs(arr, sr, sc+1); right
dfs(arr, sr+1, sc); down
dfs(arr, sr, sc-1); left

}
```

Main

```
for(i=0; i<n; i++) {
    for(j=0; j<m; j++) {
        if(arr[i][j] == -1 && arr[i][j] != 0)
        { dfs(arr, sr, sc);
          count++;
        }
    }
}
```

The question no of Islands on leetcode is reversed where, 0 means water, 1 means land & the array is char Array.

Full code on Github
No of Islands AC

No of edges = r * c * 4

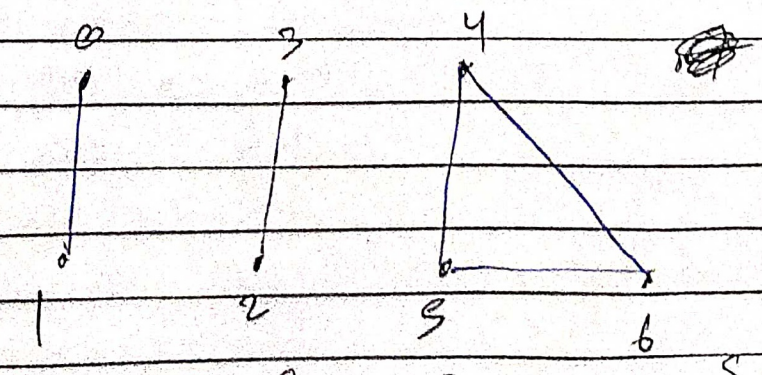
$TC = O(N+E)$
$= O(r*c + r*c*4)$
$= O(5rc)$
$\approx O(r*c)$

Ques) Perfect friends {Pepcoding} {Variation of connected components}

→ Given N (representing no of students)
→ Each student will have an id 0 to N-1
→ Given k (representing no of clubs)
→ Next k lines 2 nos sep by space, nos are ids of students belonging to same club.

Find no of ways to select a student such that both students are from different clubs.

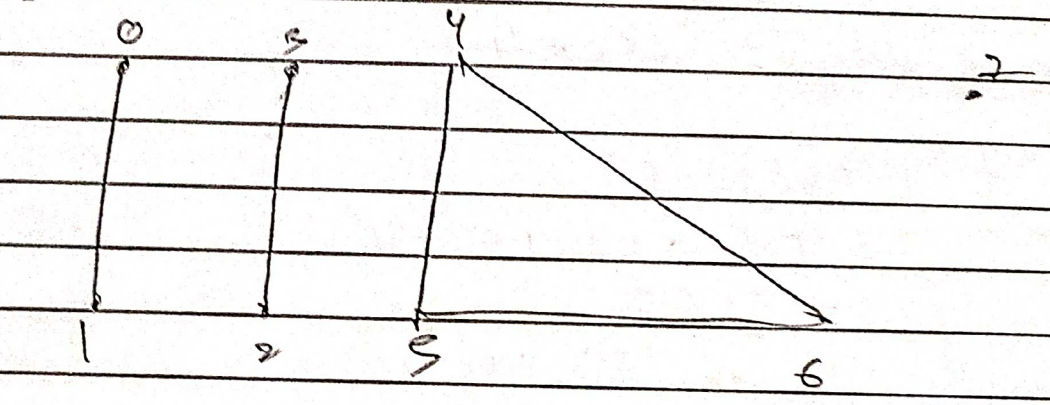
Ex.
7
5
0 1
2 3
4 5
5 6
4 6



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

The possible pairs are:

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



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

no of eles * no of eles

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

no of eles * no of eles

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

no of eles * no of eles

→ Use get connected comps method.

```
int res = 0;
for(int i=0; i<comps.size(); i++) {
    int remCompsSize = 0;
    for(int j=i+1; j<comps.size(); j++) {
        remCompsSize += comps.get(j).size();
    }
    res += comps.get(i).size * remCompsSize
}
return res;
```

(current component size)

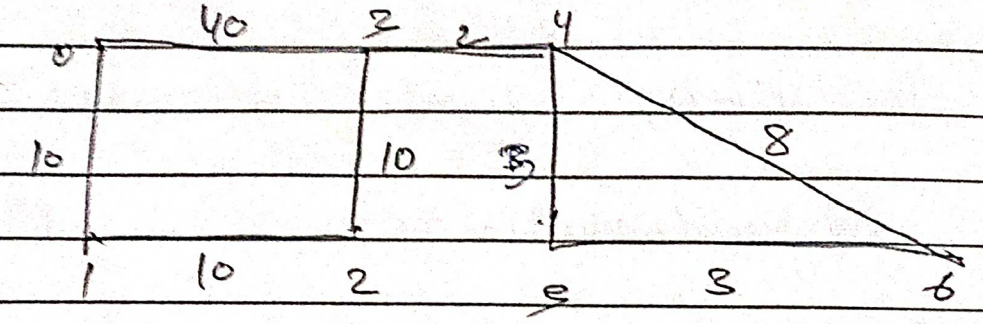
Full code on GitHub Name: Perfect Friends

1111) All Paths from Source to Target ✓ Leetcode: 797 Medium

Same Name GitHub

```
Return arraylist<arraylist<Integer>> paths.
Given graph in adj list (DAG)

//base case me copy of arraylist (path.add(cpy))
//arraylist me backtracking { path.add(nbr)
                              dfs
                              path.remove(n-1)
//visited me backtracking
```

Ques) Multisolver - Smallest, Longest, Ceil, Floor, K^{th} Largest Path.



Src → 0
Dest → 6

Paths:		Cost
	0-1-2-3-4-5-6	(38)
	0-1-2-3-4-6	{40}
	0-3-4-5-6	{48}
	0-3-4-6	{50}

Smallest cost Path {0-1-2-3-4-5-6}
Longest cost path {0-3-4-6}

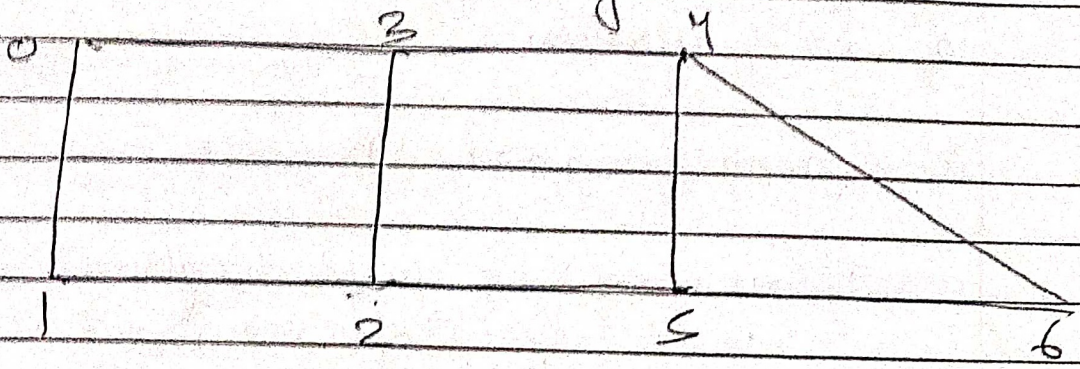
Given value say = 40
Ceil (40) Path: 0-3-4-5-6
Floor (40) Path - 0-1-2-3-4-5-6

K^{th} Largest Path → Use PQ

Code at Github Name Multisolver

⇒ No such concept except Travel & Arrange strategy. (Use of static variables)
⇒ Code is huge.

Ques Hamiltonian Path and Cycle



Hamiltonian Path: visits all the vertices in its path without repeating any vertex - Hence hamiltonian path exists only for a connected graph.

Hamiltonian Cycle: When there is an edge b/w starting & ending vertex of a hamiltonian path, it is a hamiltonian cycle

** finding even one hamiltonian path is also exponential. This is an NP-hard problem. bhi

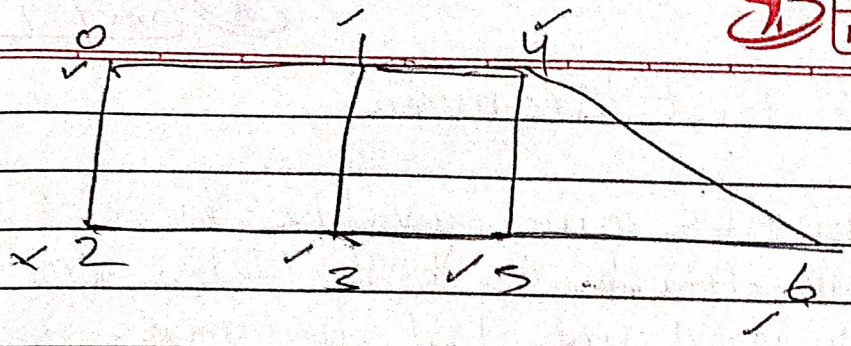
⇒ Kisi bhi node se start karke cycle me fark ~~aati~~ aayega ~~fahin~~ aur no of paths me bhi fark aa sakta hai. Dono change ho jaenge

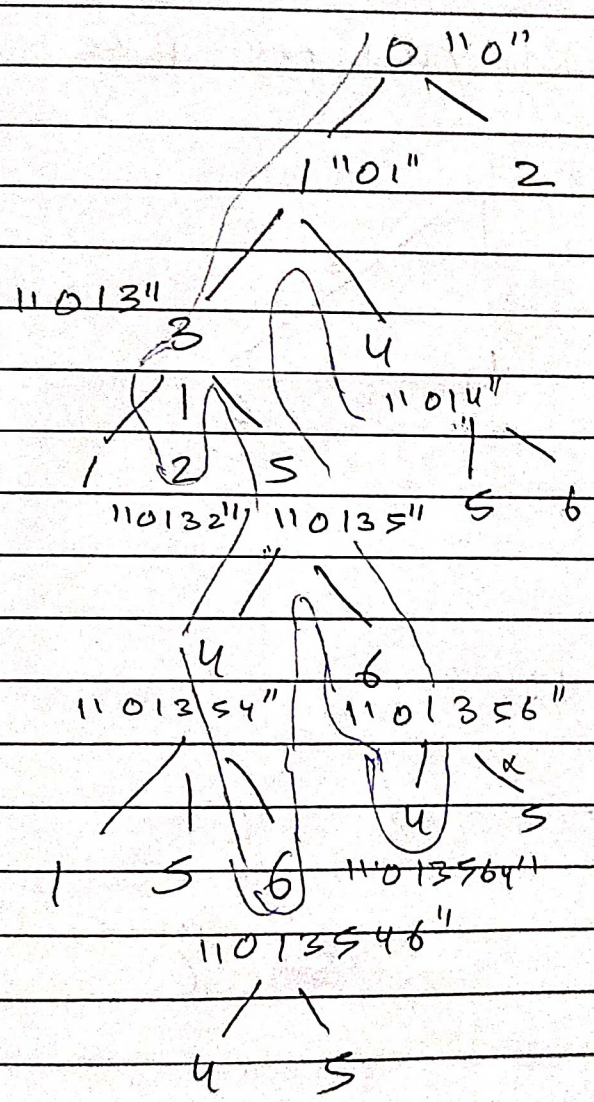
Paths: 0-1-2-3-4-5-6
0-1-2-3-4-6-5

Cycles: 0-1-2-5-6-4-3
0-3-4-6-5-2-1

Apply dfs normally maintaining the count of no of vertices.

```
dfs(graph, src, psf, vis, osrc) {          // This is a hint/hint

    if(vis.size() == graph.length-1) {
        if(isEdge(graph, osrc, src) == true) {
            Hamiltonian Cycle
        }
        else {
            Hamiltonian Path
        }
    }

    vis.add(src);

    for(Edge e: graph[src]) {
        if(vis.contains(e.nbr) == true) {
            dfs(graph, src, psf+e.nbr, vis, osrc);
        }
    }

    vis.remove(src);
}

isEdge(graph, osrc, src) {

    for(Edge e: graph[osrc]) {
        if(e.nbr == src) {
            true;
        }
    }
    false;
}
```





0

01

0154

01325

013546

01354

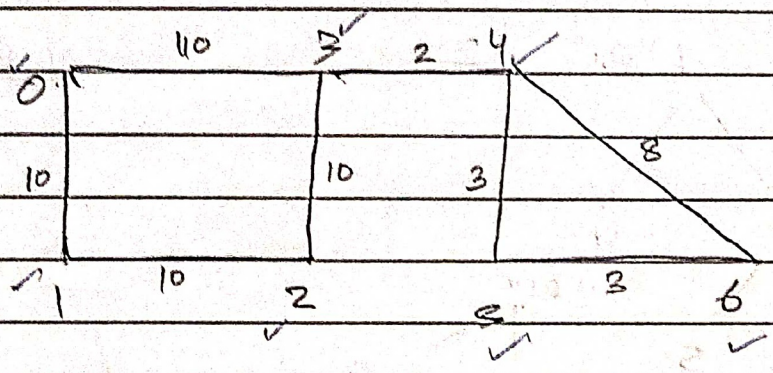
$T(n) = O(n!)$ or $O(n^n)$

Classroom
Page No.
Date

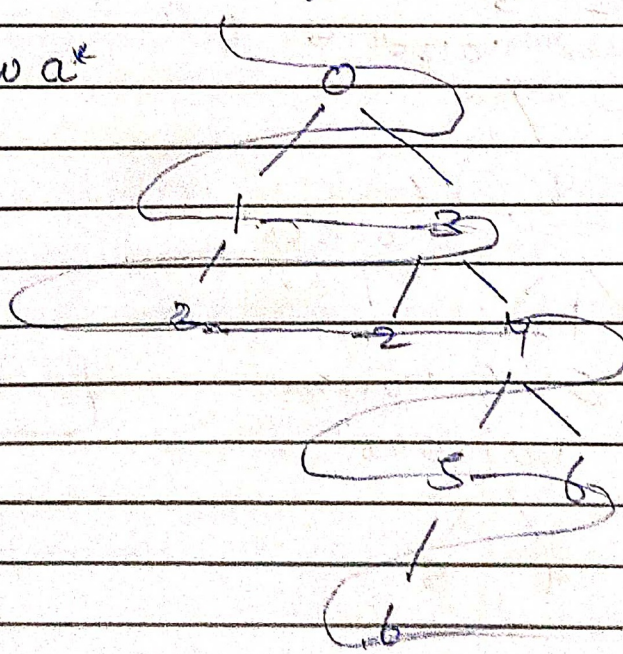
Ques Breadth first Traversal

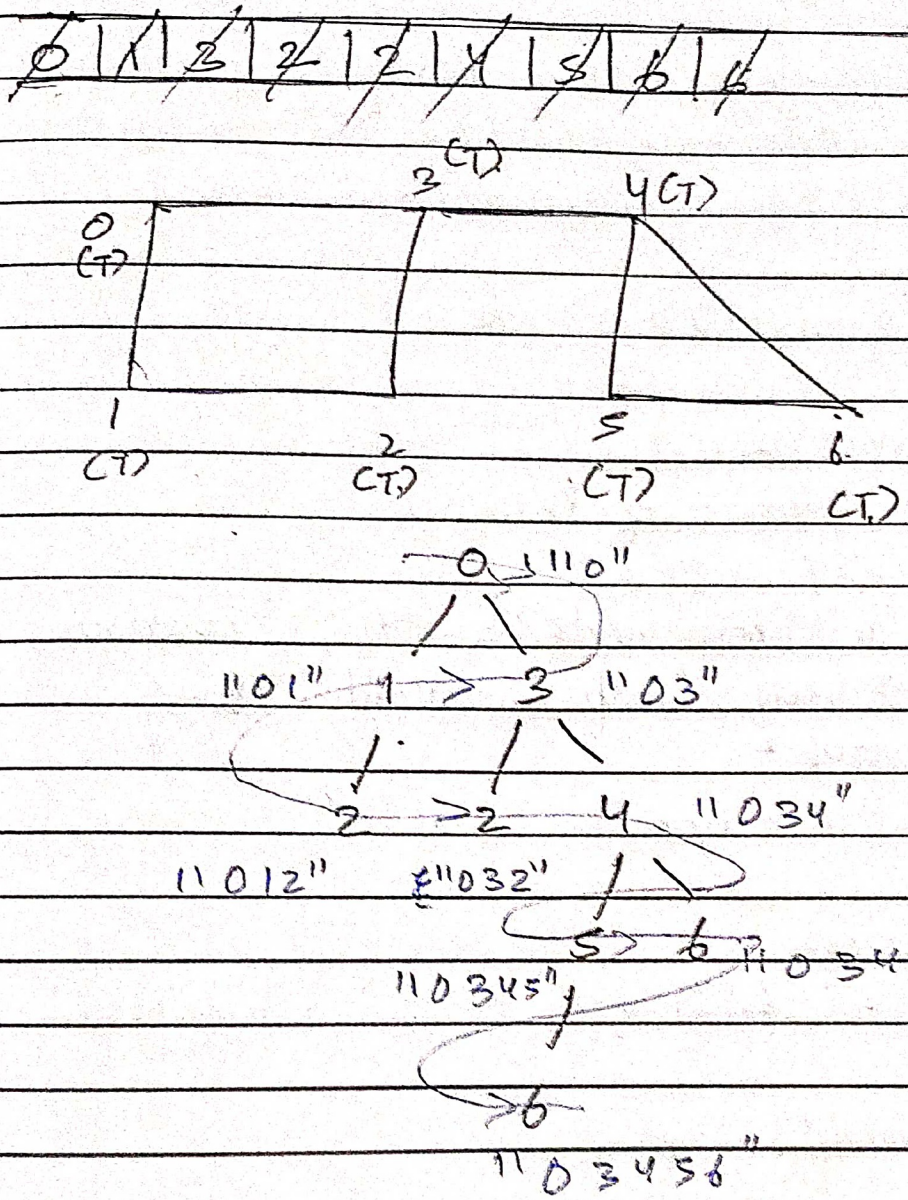
** Trees ki DFS aur graphs ki DFS me yahi farak tha ki trees ki DFS me v is already ki need nahi thi as vaha cycle nahi hoti.

Same will be the case for BFS of graph



Ram* wa*





TLE → Agar node already visited hui aur dobara is upar kaam kar rahe ho to TLE pakka aayega.

Pair {

int node;
String psf;

Constructor

}

```
//BFS
Queue<Pair> q = new ArrayDeque<>();
q.push_add(new Pair(src, "" + src));
Boolean[] vis = new boolean[v+1];

while(q.size() > 0) {

// remove
Pair rem = q.remove();

if(vis[rem.node] == true) continue;
  → very imp to avoid TLE
// mark *
vis[rem.node] = true

// work
sysout(rem.node + "@" + rem.psf);

// add *

for(Edge e : graph[rem.node]) {

  if(vis[e.nbr] == false) {
    q.add(new Pair(e.nbr, rem.psf + e.nbr
  }

}
```

Github Name BFS

Classroom
Page No.
Date / / /

Que) Is Graph Cyclic

Cycle tab banegi jab ek node bfs queue me 2 baar aaya aur alag alag path se. Check cycle for each component also.

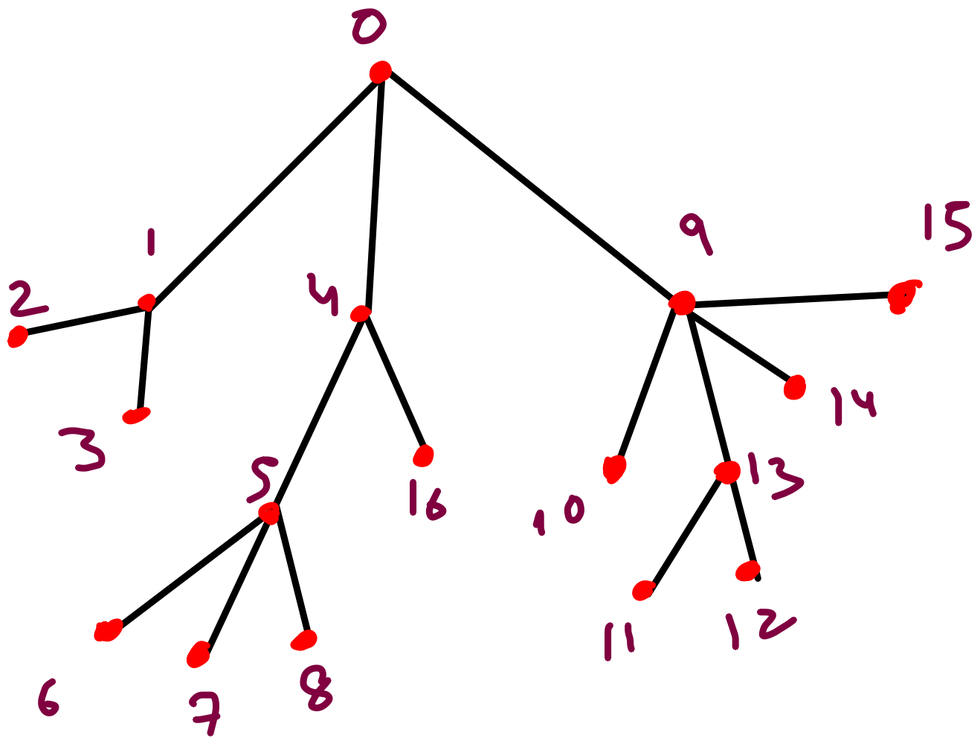
```
for (i=0; i<V; i++)
{
    if (vis[i] == false)
    {
        boolean isCyclic = bfs(graph, i);
        if(isCyclic) return true
    }
}
return false
```

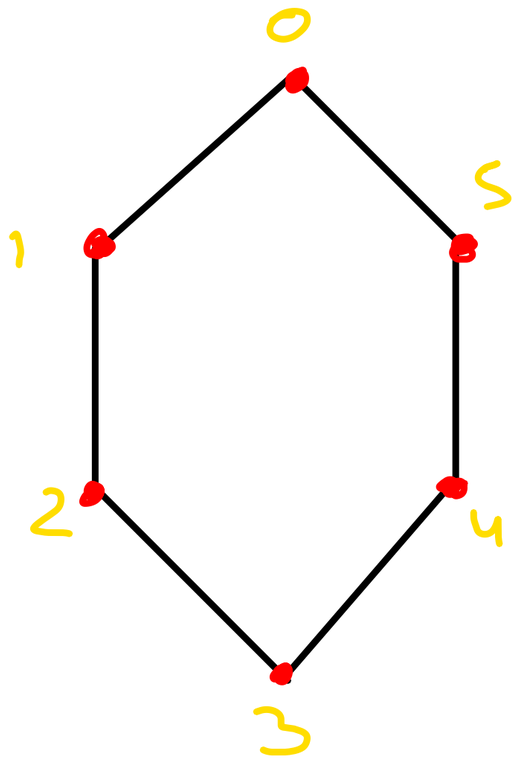
→ if (vis[rem.node] == true) return true.
↓ Here is the change in bfs

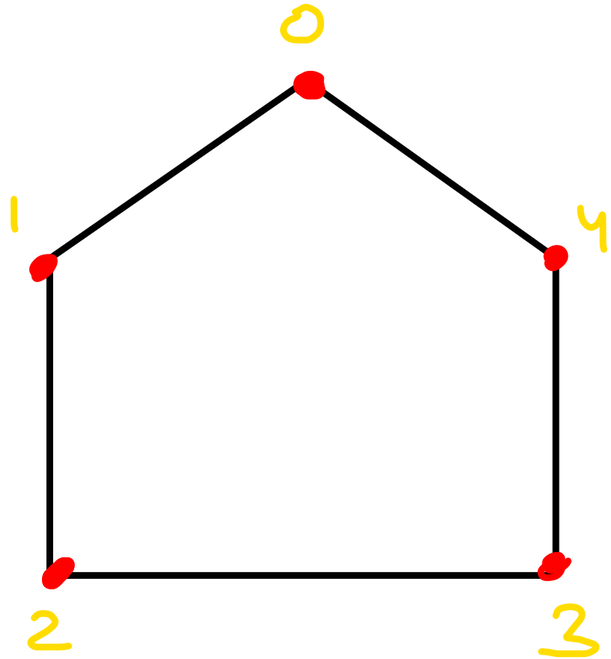
Github Name Is Graph Cyclic

TC = O(N+E)

Bipartite Graph { Is Graph Bipartite }







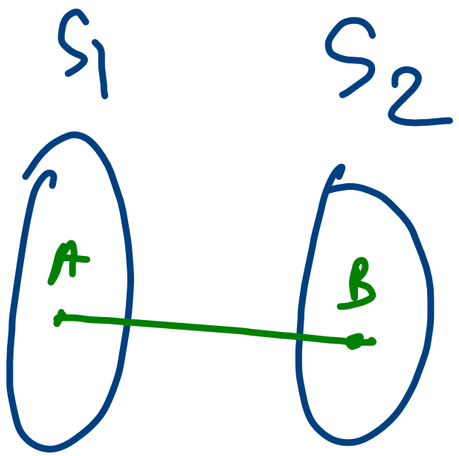
if we are able to divide all the vertices of a graph in two mutually exclusive & exhaustive sets and there is no edge within the sets.

① All vertices should be in Set 1 or 2.

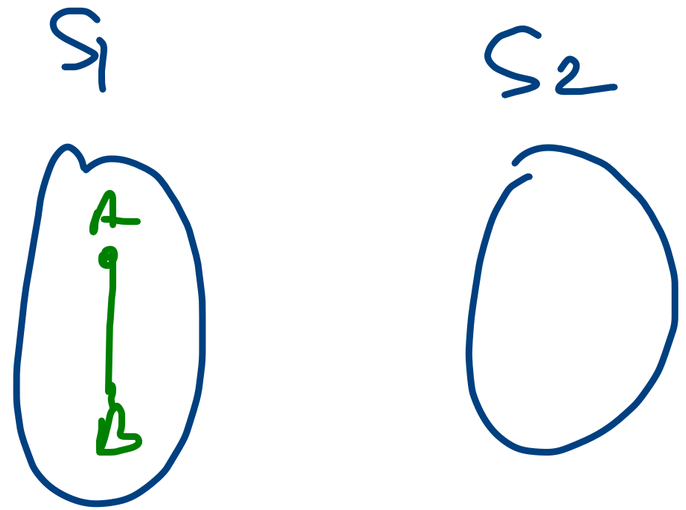
② No vertex should be in both the sets.

③ Let us say that AB is an edge of the graph.

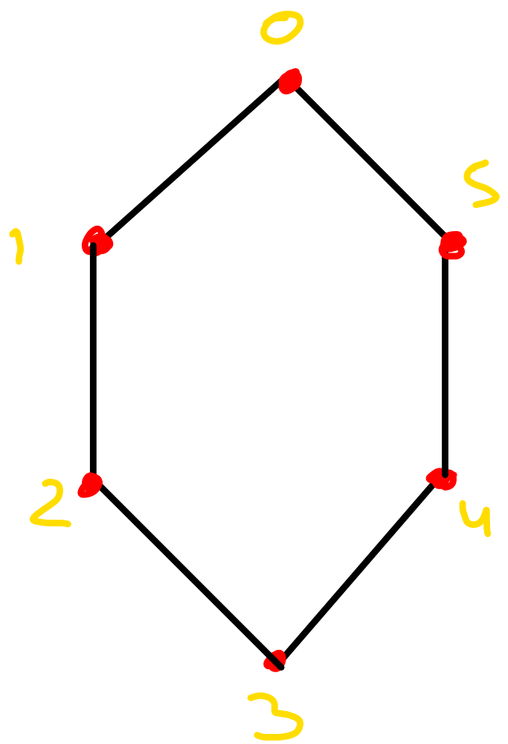
So,

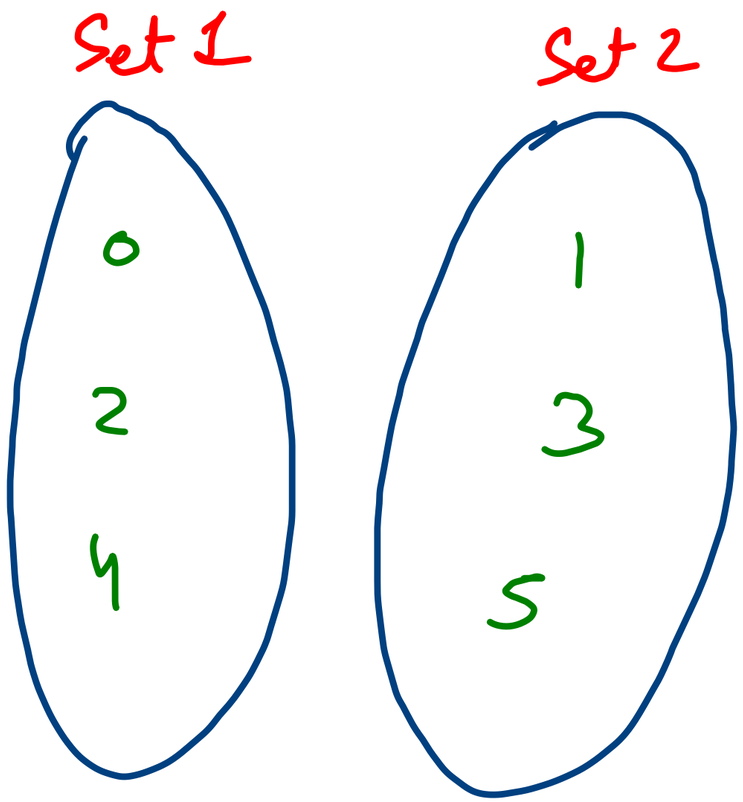


Allowed



Not allowed

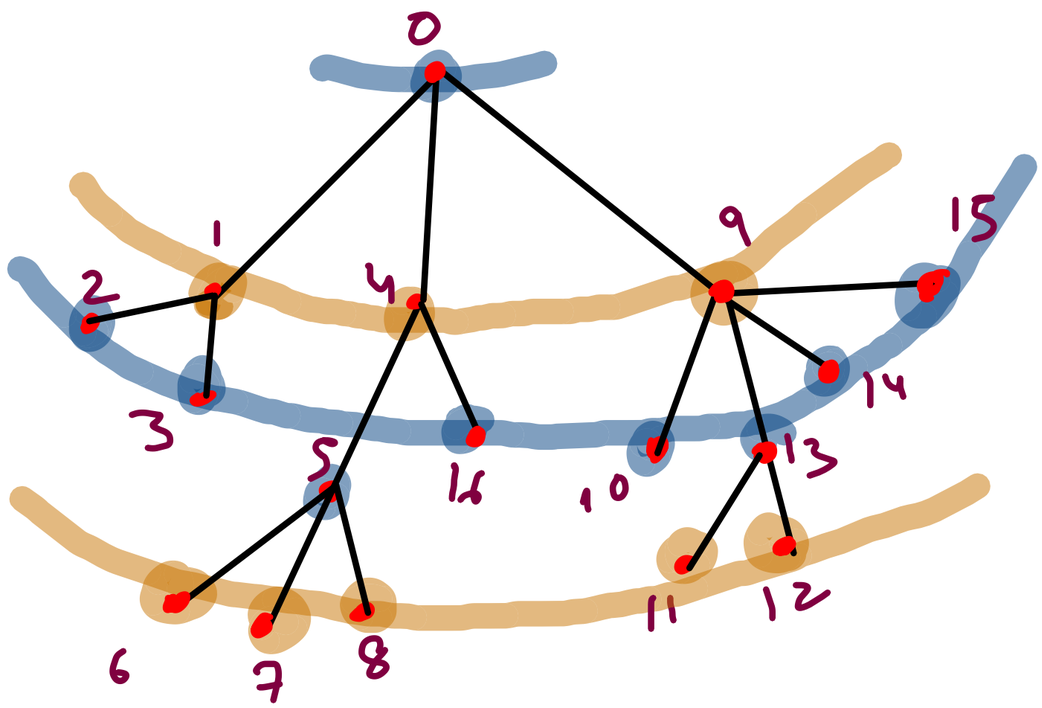




Bipartite Graph

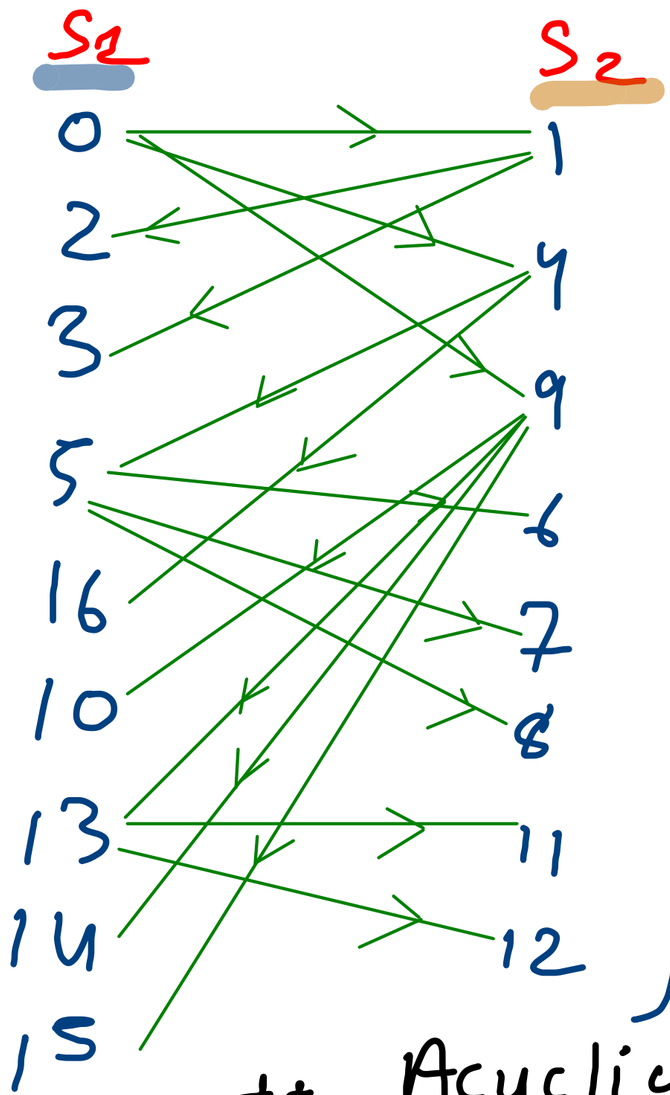
If all cycles are of even length, graph will be bipartite

Main Point is that edge should not be within same set. Across set ho ya na ho us se fark nahi padhta.



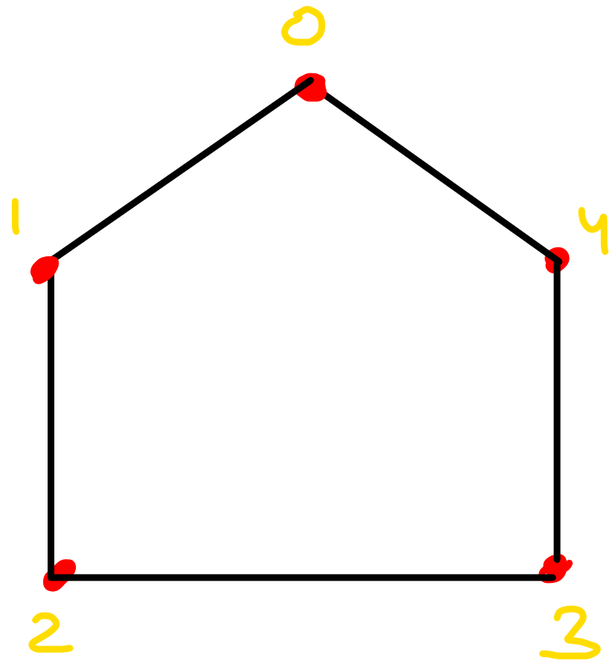
Level wise nodes in alternate sets.

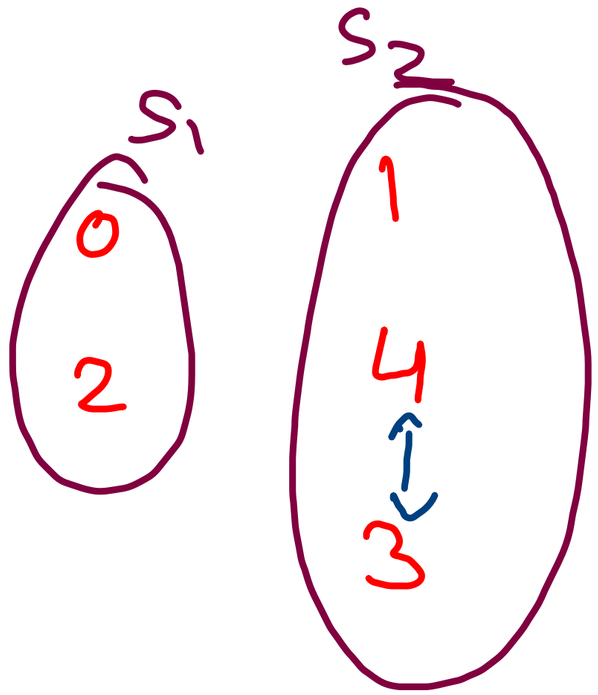
Also, a bipartite graph.



No cycle in any component.

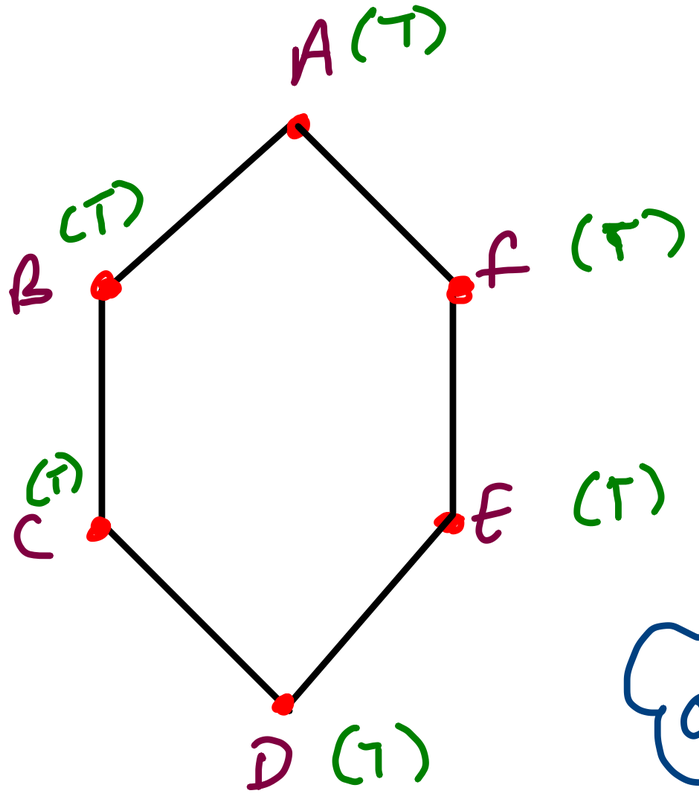
** Acyclic graph is always bipartite.





Not a bipartite graph.

Graph with at least one cycle of odd length is always non-bipartite.



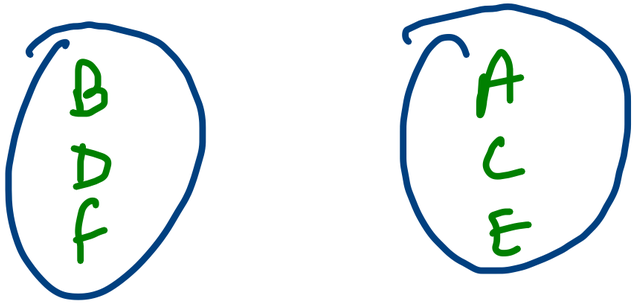
odd teven = vtces

HashSet <Integer> odd, even

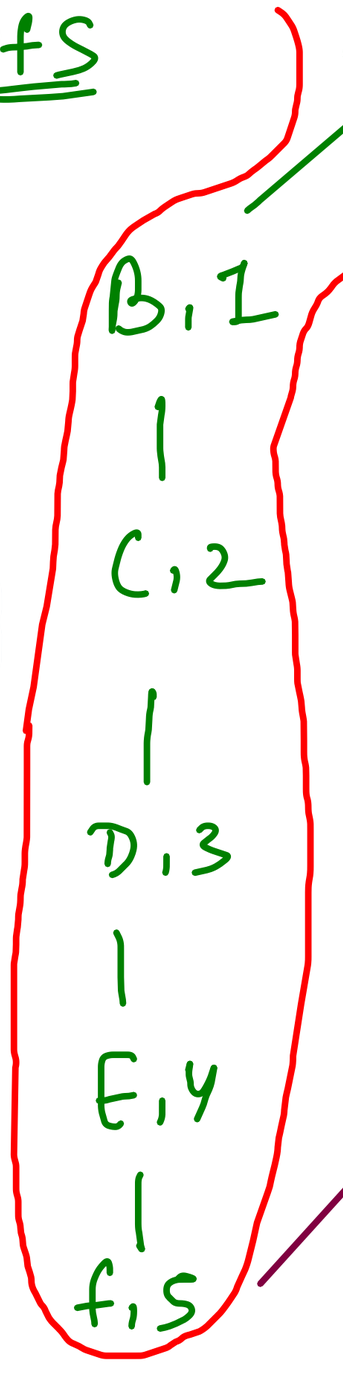
odd

even

level

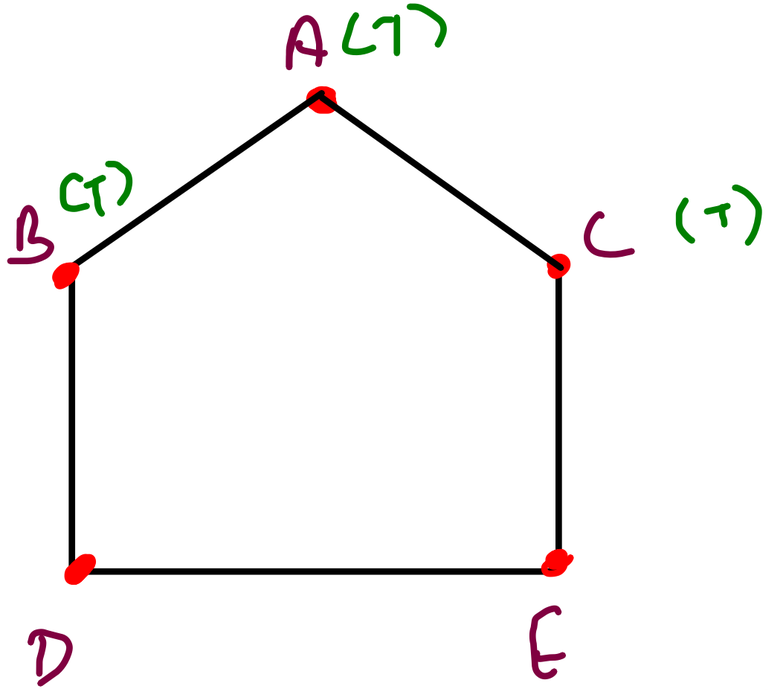


Dfs

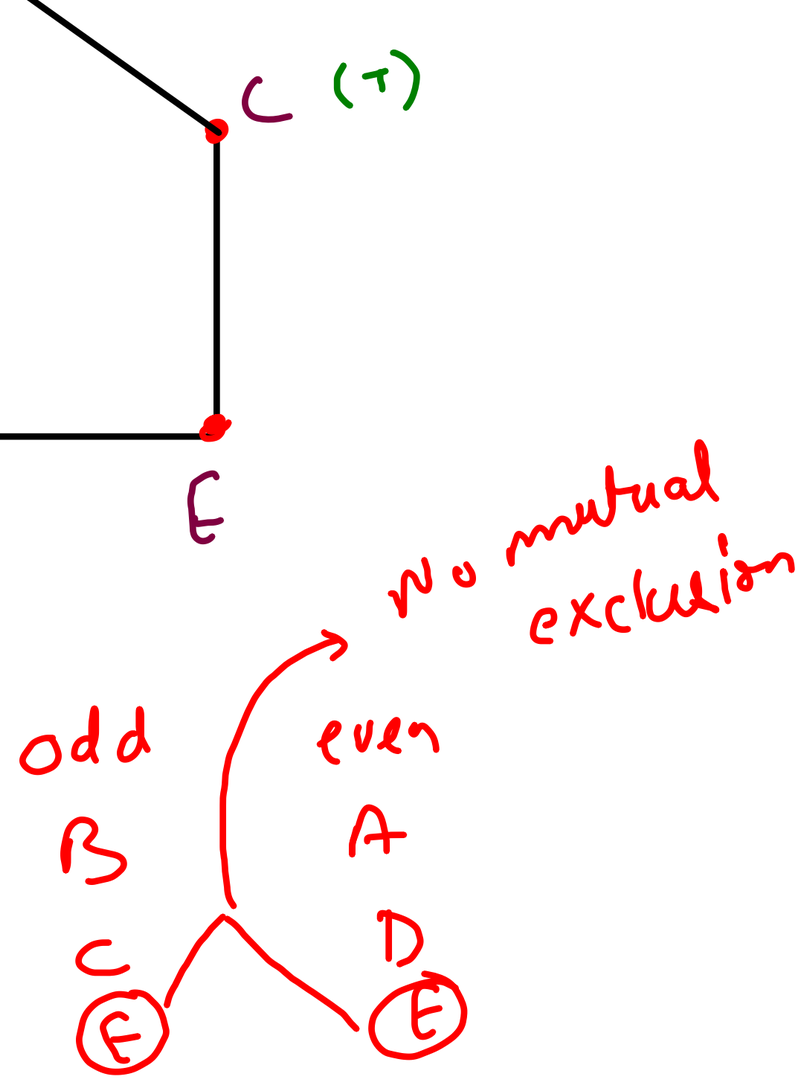


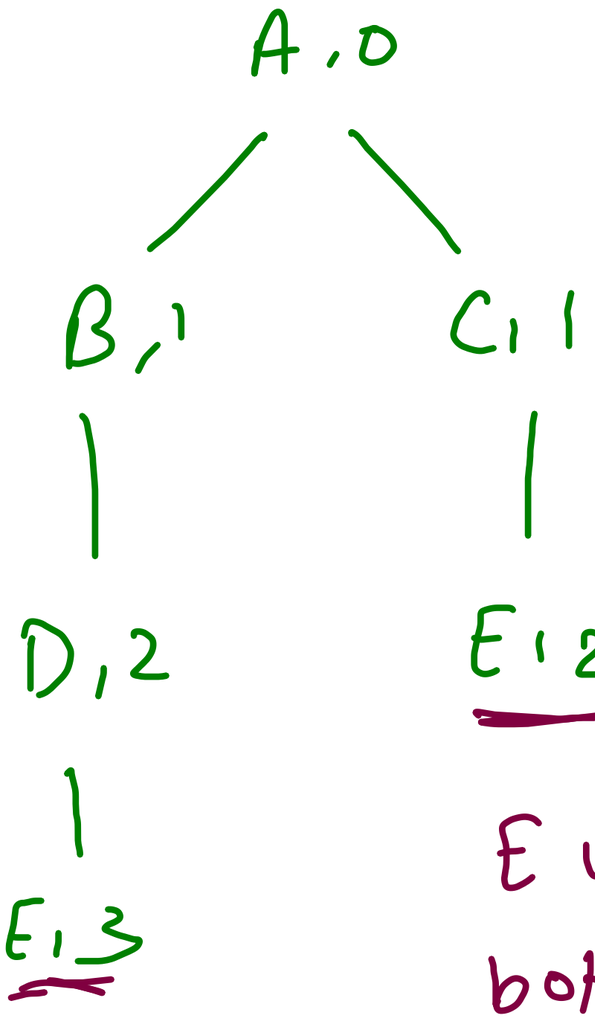
f pehle bhi odd set me tha aur abhi bhi vahi jana chahta hai.

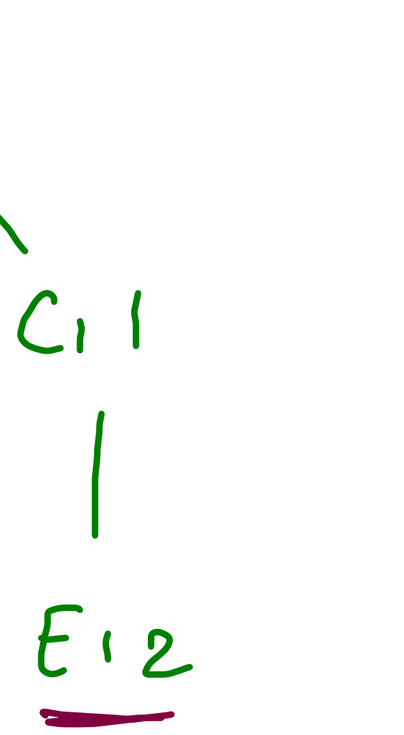
Hence Bipartite



A_0	B_1	C_2	D_2	E_2







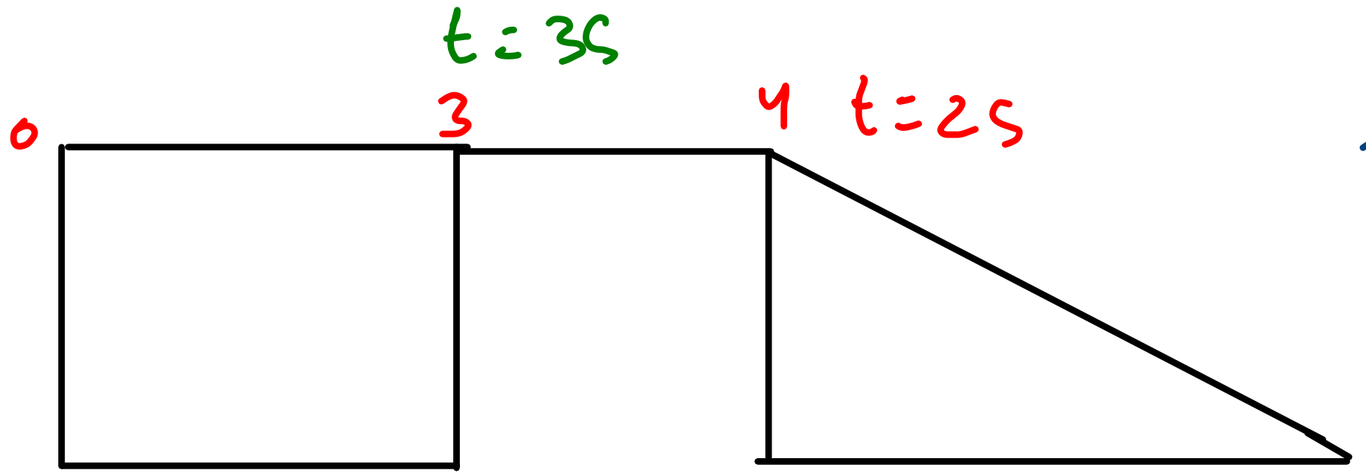
E wants to go in both sets. Hence not bipartite

```java
public static boolean dfs(ArrayList<Edge>[] graph, int src, int level, boolean[] vis,HashSet<Integer> even, HashSet<Integer> odd) {

    vis[src] = true;

    if(level % 2 == 0) {
        even.add(src);
    } else {
        odd.add(src);
    }

    for(Edge e : graph[src]) {

        if(vis[e.nbr] == false) {
            boolean res =  dfs(graph,e.nbr,level + 1,vis,even,odd);
            if(res == false) return false;
        } else {
            if(odd.contains(e.nbr) == true && (level + 1) % 2 == 0) return false;
            if(even.contains(e.nbr) == true && (level + 1) % 2 != 0) return false;
        }
    }

    return true;
}
```

```java
    // write your code here
    HashSet<Integer> odd = new HashSet<>();
    HashSet<Integer> even  = new HashSet<>();
    boolean[] vis = new boolean[vtces];

    for(int i=0;i<vtces;i++) {
        if(vis[i] == false) {
            boolean isComponentBipartite =  dfs(graph,i,0,vis,even,odd);
            if(isComponentBipartite == false) {
                System.out.println(false);
                return;
            }
        }
    }

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

} main()

Spread of Infection {Using Bfs} {If 1s variation is Rotten Oranges}



How many nodes will be infected at the end of 3s.

Hence, at the end of 3s, 4 nodes will be infected.

{Bfs} ✓ DFS ✗

```java
// write your code here
boolean[] vis = new boolean[vtces];
Queue<Pair> q = new ArrayDeque<>();
q.add(new Pair(src,1));
int count = 0;

while(q.size() > 0) {
    Pair rem = q.remove();

    if(vis[rem.node] == true) continue;

    vis[rem.node] = true;

    if(rem.time > t) break;
    count++;

    for(Edge e : graph[rem.node]) {
        if(vis[e.nbr] == false) {
            q.add(new Pair(e.nbr,rem.time + 1));
        }
    }
}

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

Bfs is the shortest path algorithm when we have to find the shortest path in terms of no of edges.

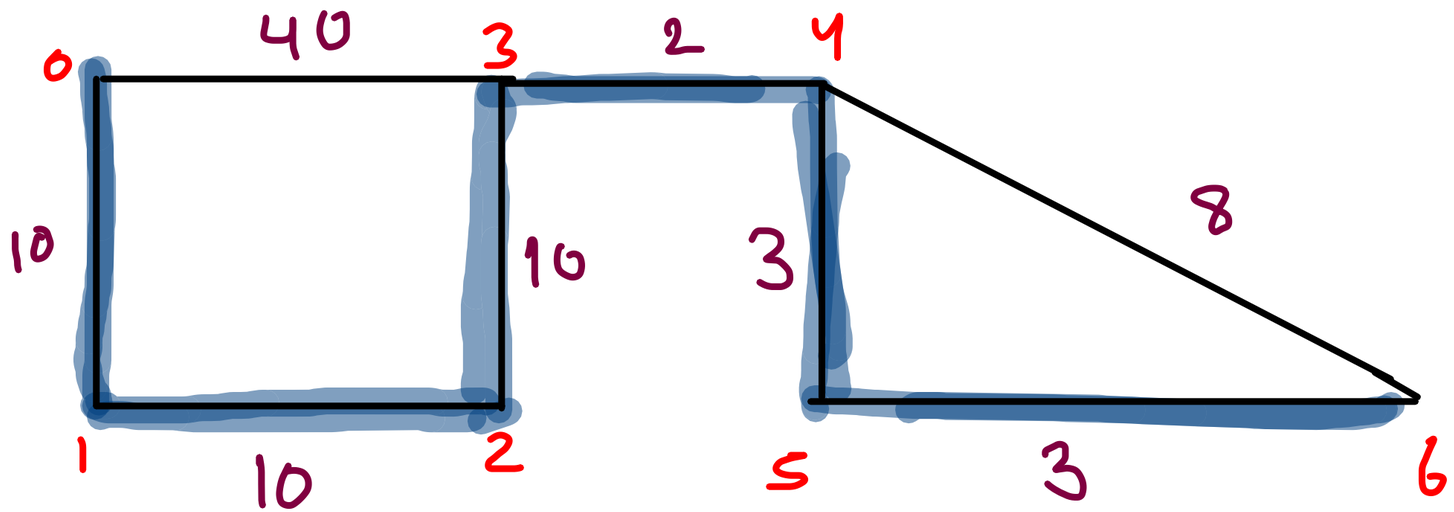
Since in unweighted graphs, shortest path is only calculated in terms of no of edges, we can also say that Bfs is the shortest path algo for undirected graph.

$TC = O(E+V)$ ⇒ Because we will not explore all the paths

Dijkstras' Algorithm

{Shortest Path in Weights}

{Single Source Shortest Path in Wts}



We will use a Priority Queue. Because we have to select the path on a custom criteria now i.e weight.

```
static class Pair implements Comparable<Pair>{
    int node;
    int wsf;
    String psf;

    Pair(int node, int wsf, String psf) {
        this.node = node;
        this.wsf = wsf;
        this.psf = psf;
    }

    public int compareTo(Pair other) {
        return this.wsf - other.wsf;
    }
}
```

```java
int src = Integer.parseInt(br.readLine());
// write your code here
PriorityQueue<Pair> pq = new PriorityQueue<>();
pq.add(new Pair(src,0,"" + src));
boolean[] vis = new boolean[vtces];

while(pq.size() > 0) {
    //remove
    Pair rem = pq.remove();

    if(vis[rem.node] == true) continue;

    //mark*
    vis[rem.node] = true;

    //work
    System.out.println(rem.node + " via " + rem.psf + " @ " + rem.wsf);

    //add*
    for(Edge e : graph[rem.node]) {
        if(vis[e.nbr] == false) {
            pq.add(new Pair(e.nbr,rem.wsf + e.wt,rem.psf + e.nbr));
        }
    }
}
```

$O(E+V\log V)$ ~

$O(E\log V + V\log V)$ ~

$O(V+E\log V)$

↓

because of

push

pop being

$O(\log_2(n))$

operation

① Dijkstra will fail in case of Negative weight cycle graph.

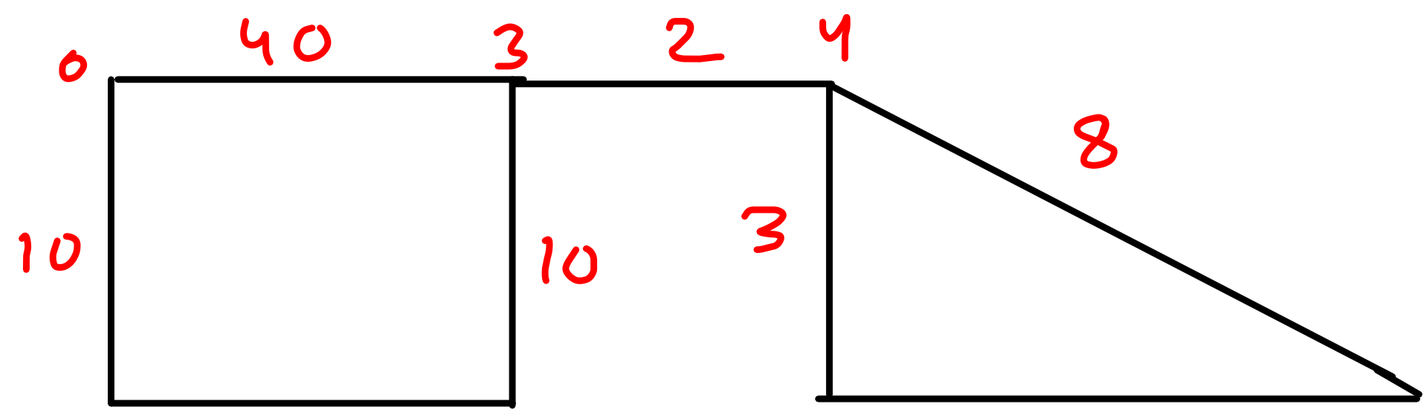
⇒ Learn using Bellman ford

② Also, this is not a multisource algo. So, if we want dijkstra for multisource, complexity will be $V * (V + E \log V)$ (E ↓ V^2) $\approx O(V^3 \log V)$ .max

Minimum Spanning Tree {Prims Algorithm}

↳ Subgraph {all vertices of Graph}

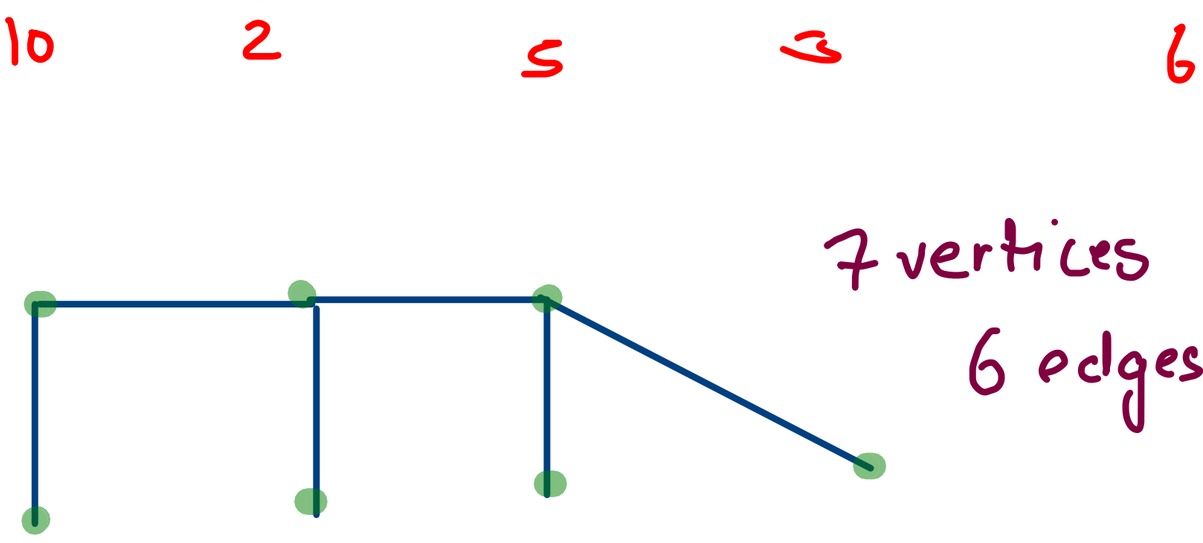
↳ Tree



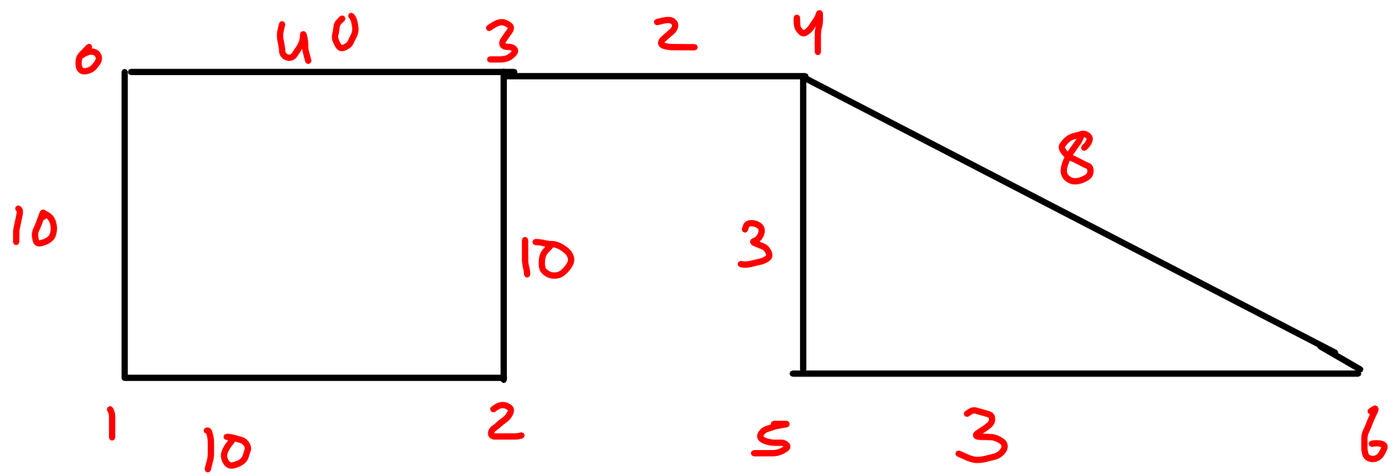
↳ min edges (spanning)

↳ Acyclic

↳ Connected

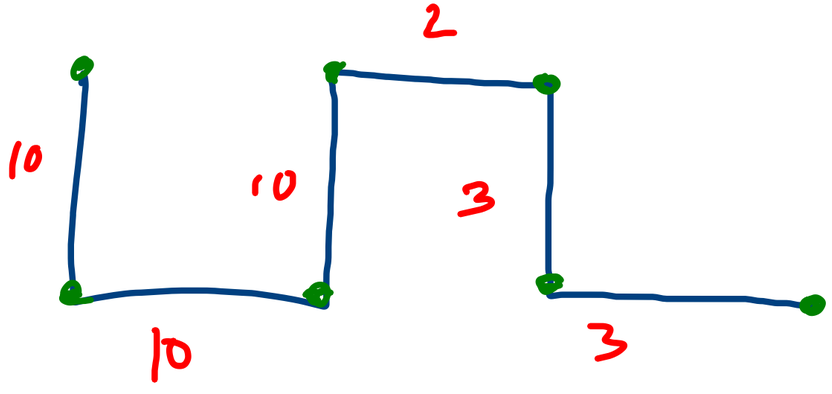


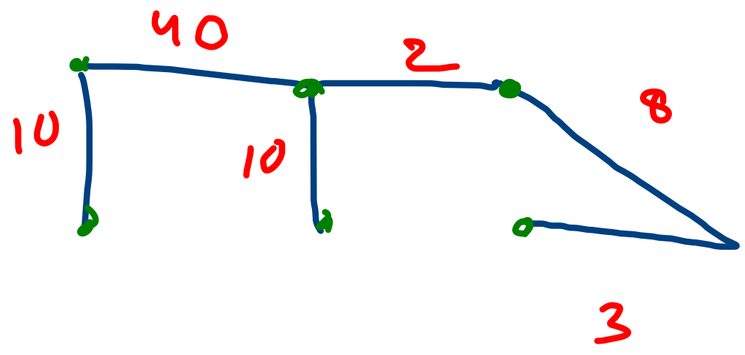
N vertices → N-1 edges

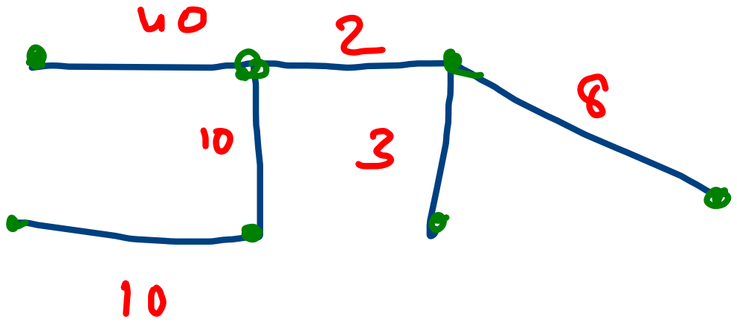


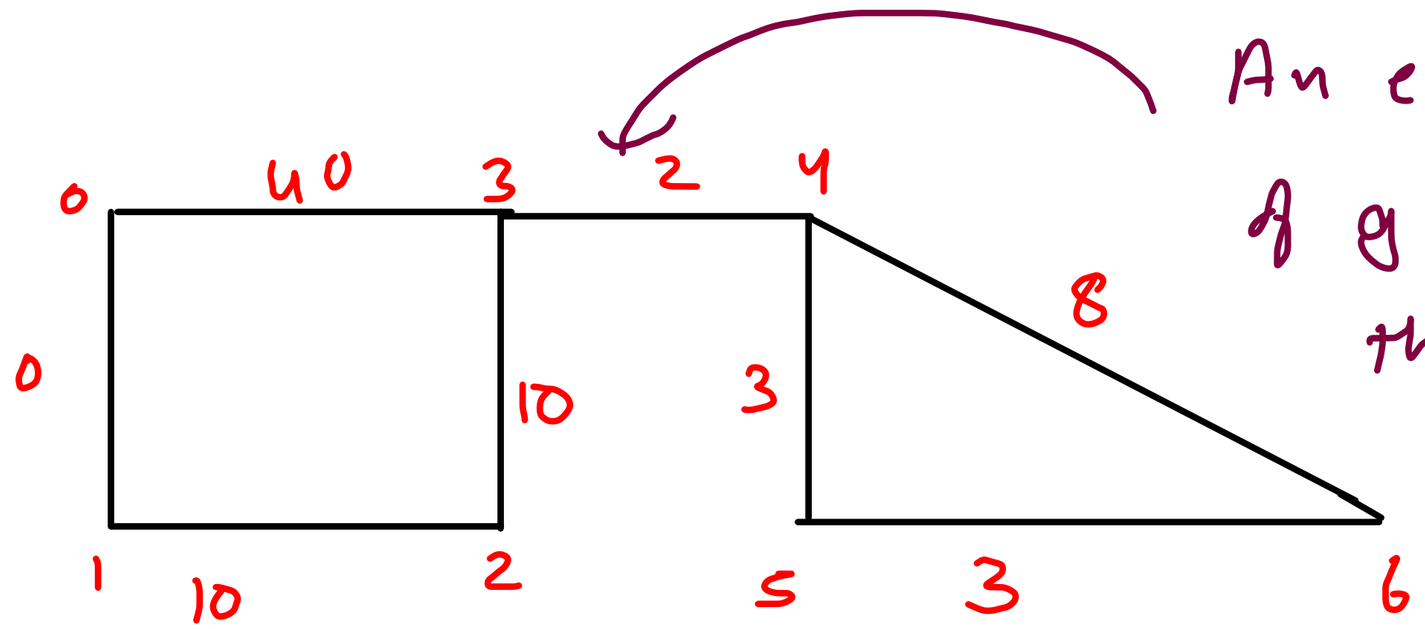
Multiple Spanning Trees are possible

Min spanning tree is one with the least cost









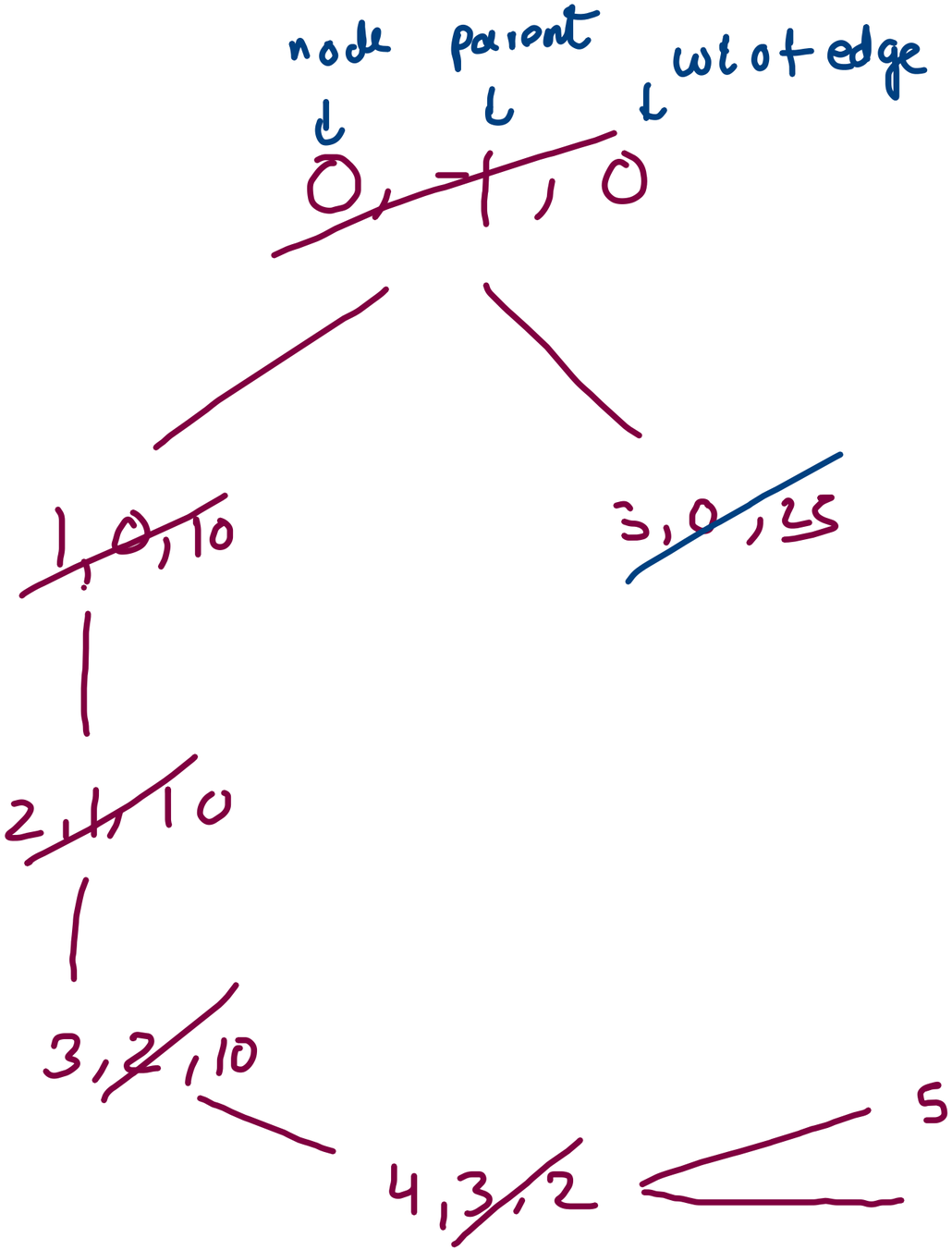
An edge that connects 2 comps of graph & on its removal, the graph will be divided in 2 components is called a bridge.

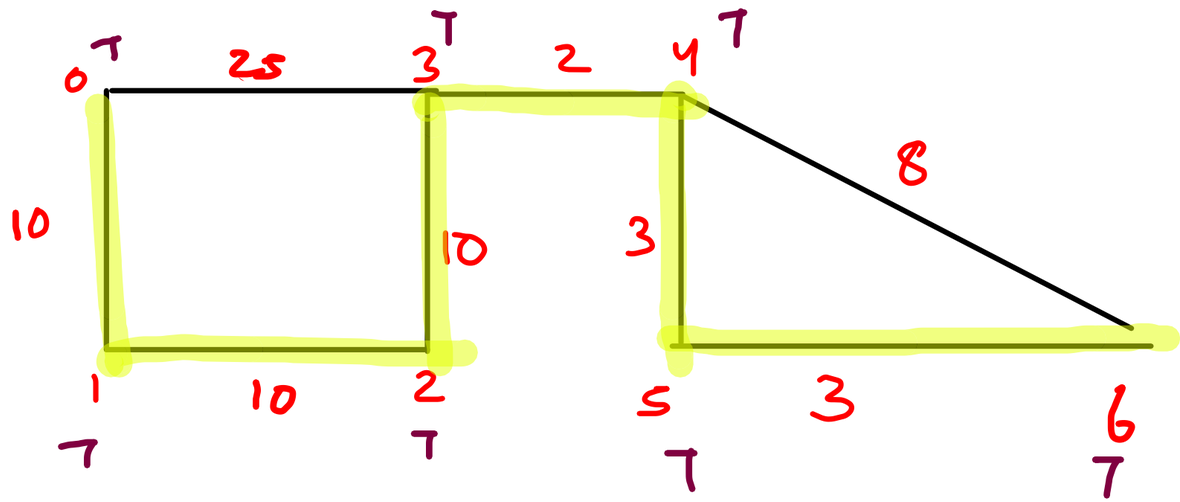
So, in Spanning tree formation, we cannot remove bridge as Then it will not be a tree.

Method for creating a Min cost Spanning Tree is easy. Just keep all the edges with min cost.

Process is very similar to Dijkstra Algorithm. The key difference is that in Dijkstra, we are considering wsf however in Prims' Algo we will only consider the weight of an edge.

Start Prims' Algo from any node of your choice.

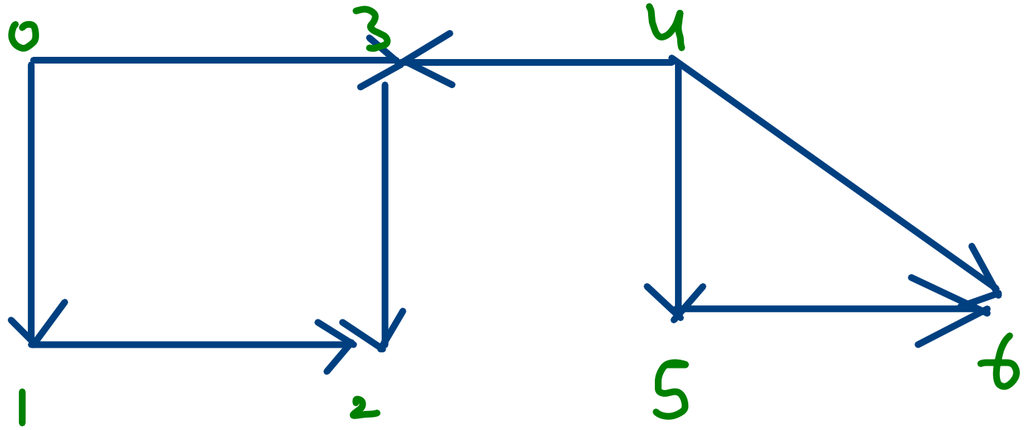

node
parent
wt of edge
0 , -1 , 0
1 , 0 , 10
3 , 0 , 25
2 , 1 , 10
3 , 2 , 10
4 , 3 , 2
5 , 4 , 3
6 , 5 , 3
6 , 4 , 8


0
25
3
2
4
10
10
3
8
1
10
2
5
3
6

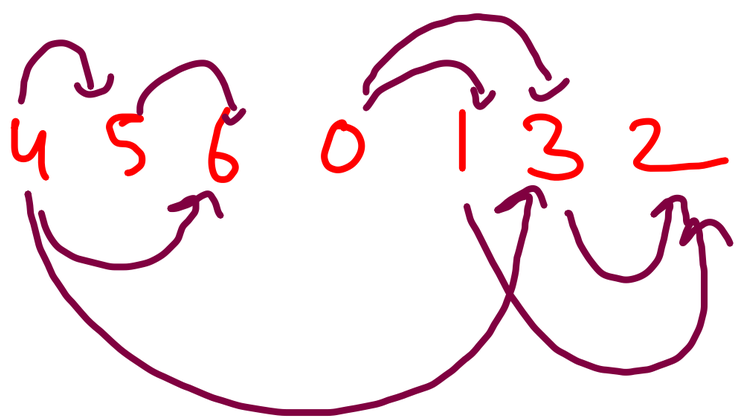
```java
//prims algo
PriorityQueue<Pair> pq = new PriorityQueue<>();
pq.add(new Pair(0,-1,0));
boolean[] vis = new boolean[vtces];

while(pq.size() > 0) {
    Pair rem = pq.remove();

    if(vis[rem.node] == true) continue;

    //mark*
    vis[rem.node] = true;

    //work
    if(rem.parent != -1) {
        System.out.println("[" + rem.node + "-" + rem.parent + "@" + rem.wt + "]");
    }

    //add*
    for(Edge e : graph[rem.node]) {
        if(vis[e.nbr] == false) {
            pq.add(new Pair(e.nbr,rem.node,e.wt));
        }
    }
}
```

Topological Sort { Order of Compilation }

↳ only for Directed Acyclic Graph (DAG)

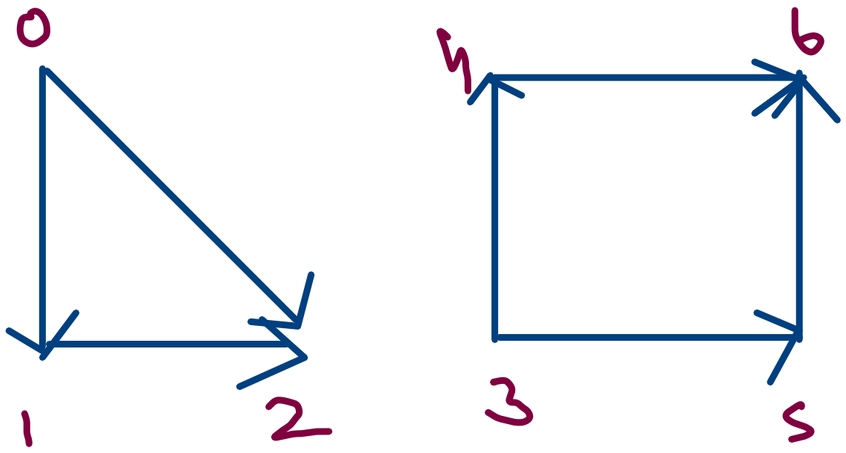


Ex:



Permutation of vertices of graph such that if there is an edge from $u \rightarrow v$, then u must occur before v in the permutation.

can be more than 1 topo sort.



① 0 1 2 3 4 5 6

② 0 1 2 3 5 4 6

③ 3 4 5 6 0 1 2

④ 3 5 4 6 0 1 2

} Multiple topological sorts.

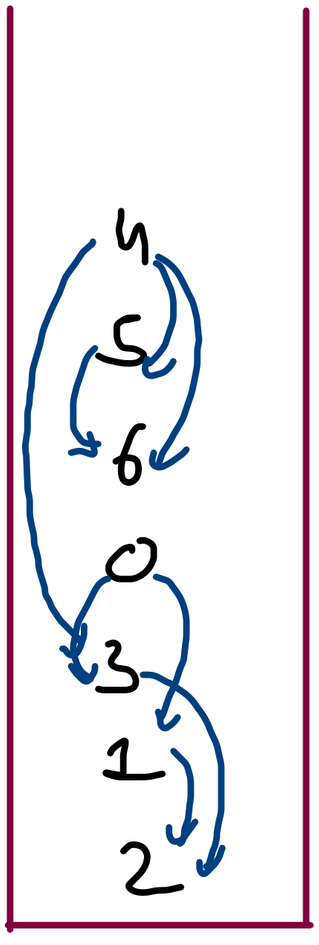
Algorithm (Dfs, Stack)

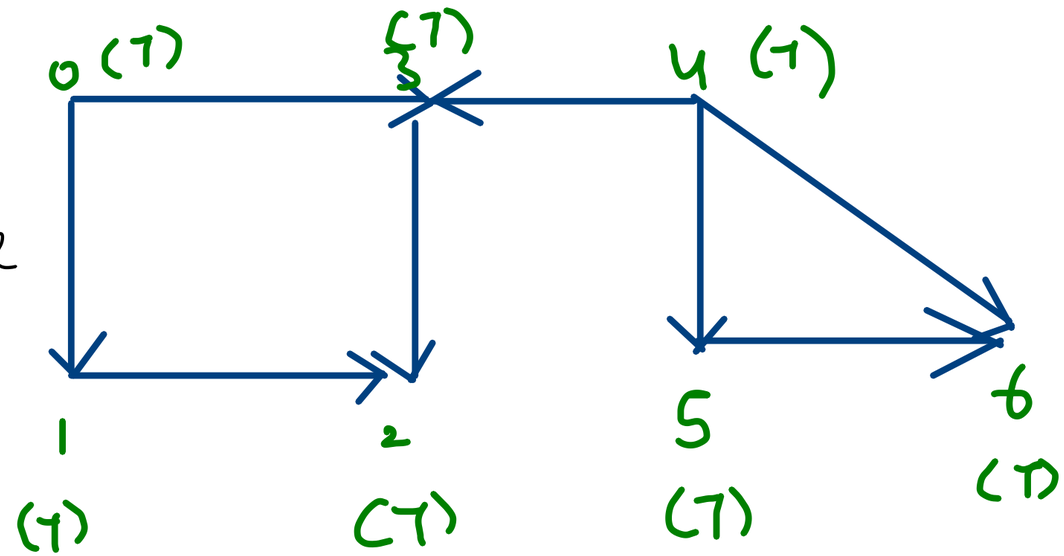
```
// write your code here
boolean[] vis = new boolean[vtces];
Stack<Integer> topoSort = new Stack<>();

for(int i=0;i<vtces;i++) {
    if(vis[i] == false) {
        dfs(graph,i,vis,topoSort);
    }
}
```

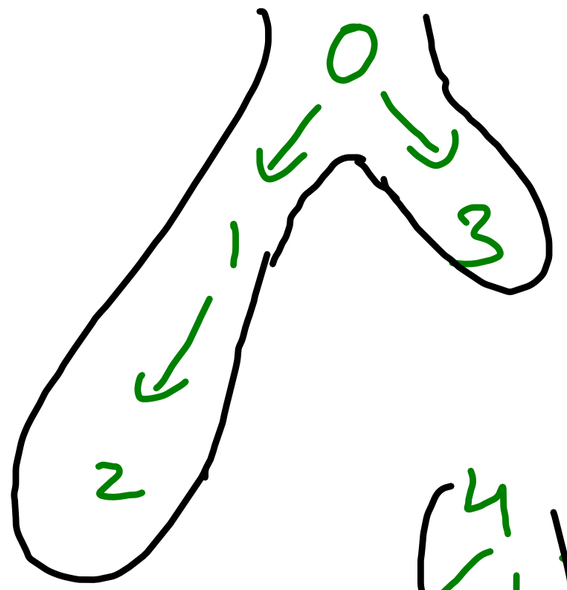
Kisi DAG par dfs lagane k liye visited array ki need nahi hai. Here, we are taking a vis array so that we can know whether to include the vertex (push it) into the stack or not.

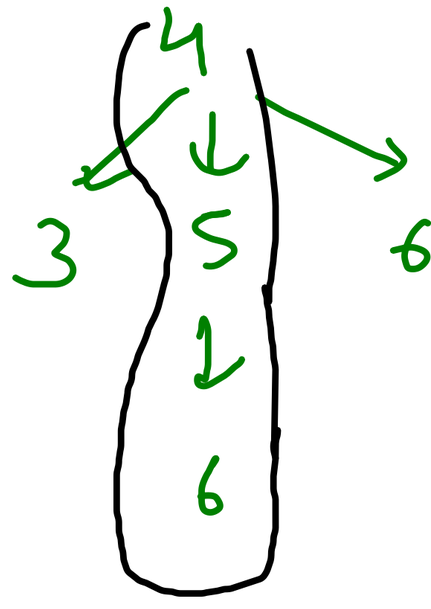
```
public static void dfs(ArrayList<Edge>[] graph, int src,boolean[] vis, Stack<Integer> topoSort) {
    vis[src] = true;

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

    topoSort.push(src);
}
```

faith: Hamjin pe depend hai vo pehle hi stack me honge









⇒ Topological Sort

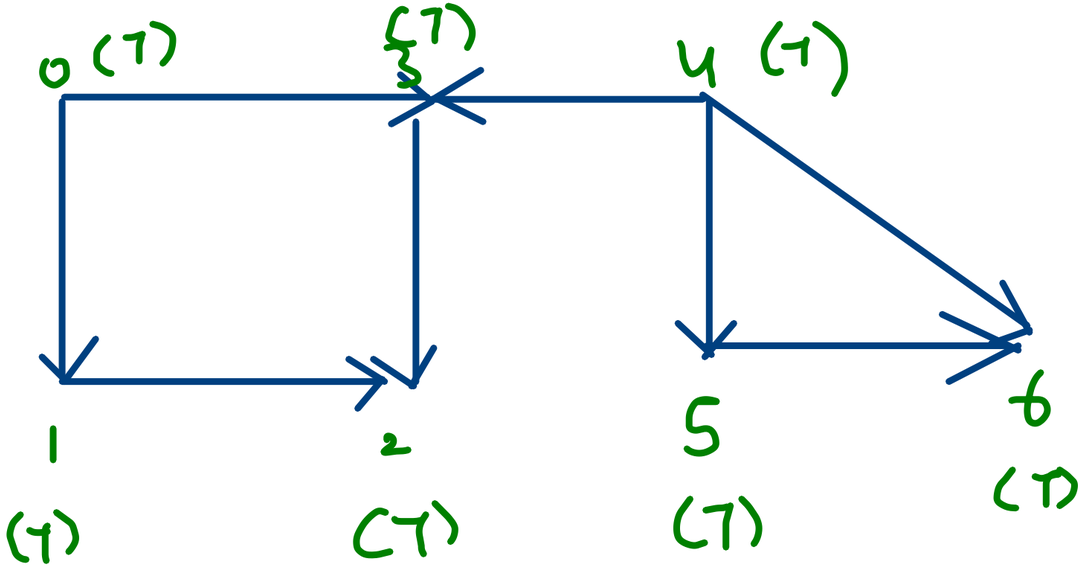
++ Even when the graph was connected we could not do dfs in go from one comp as it was not strongly connected.

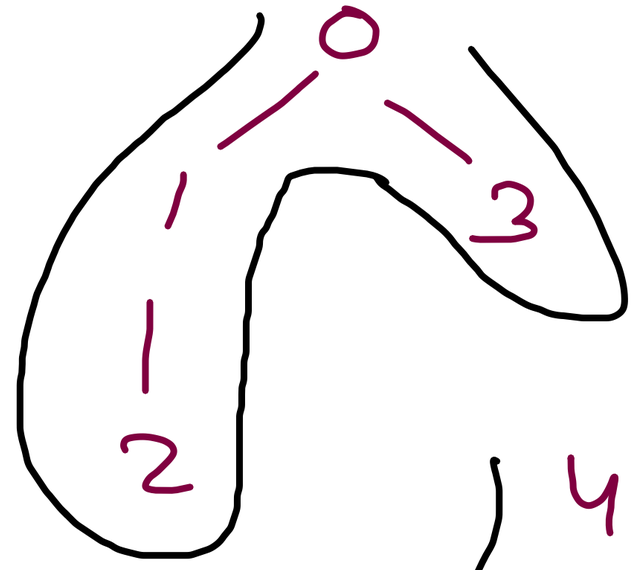
Applications of Topological Sorting

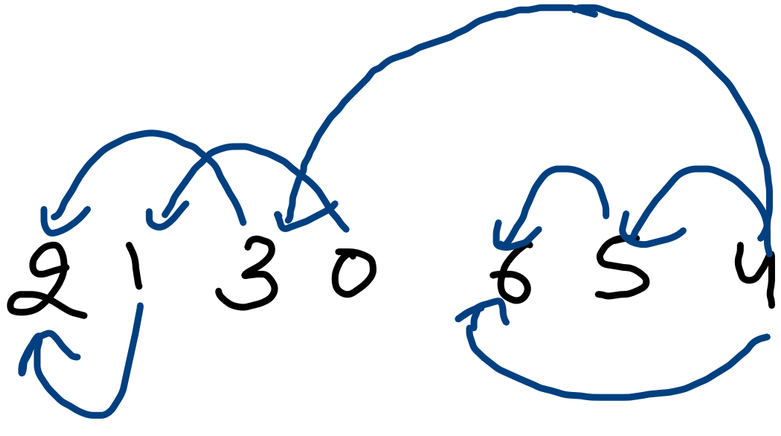
1. Prerequisites / Course Schedule
2. Alien Dictionary
3. Minimum Time Job Scheduling
4. Mother Vertex
5. Reconstruct Itenary

1. Finding cycle in a graph { Cycle Detection }
2. Operation System deadlock detection
3. Dependency resolution
4. Sentence Ordering
5. Critical Path Analysis
6. Course Schedule problem
7. Other applications like manufacturing workflows, data serialization and context-free grammar.

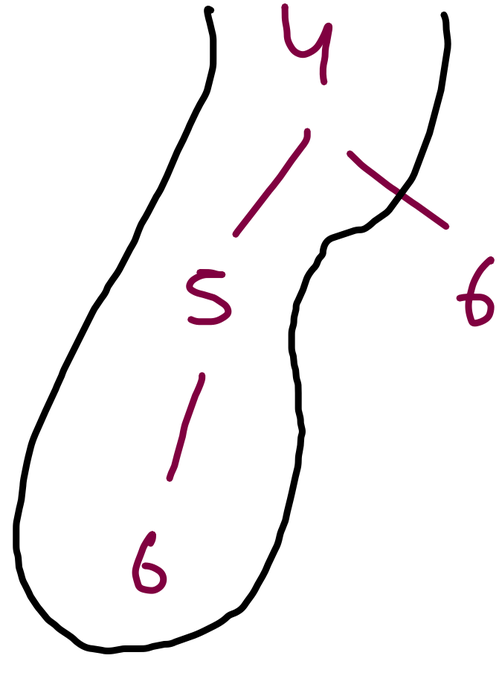
Why use stack? Why can't we directly print in Post order?



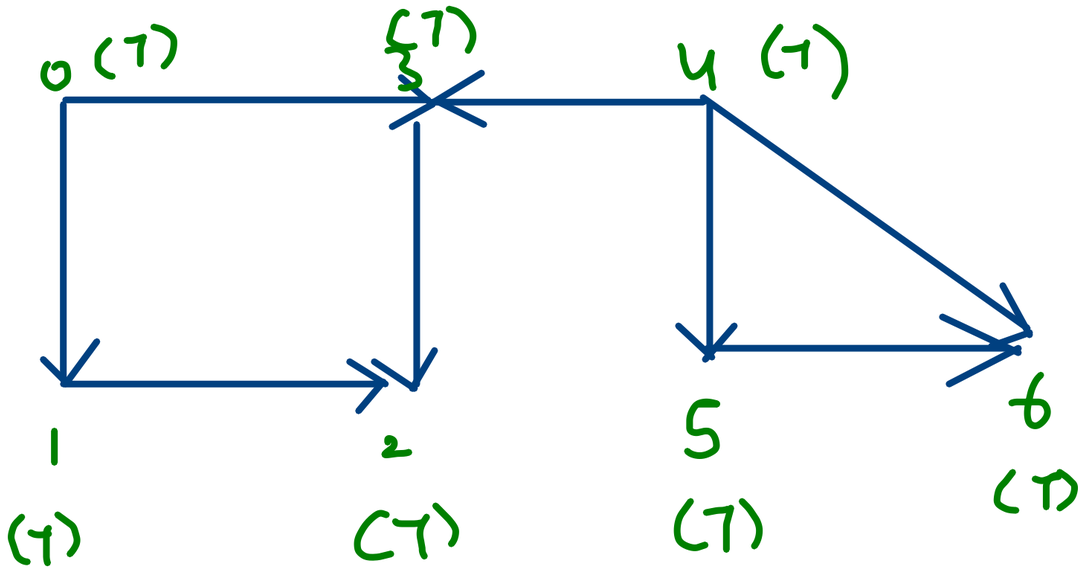


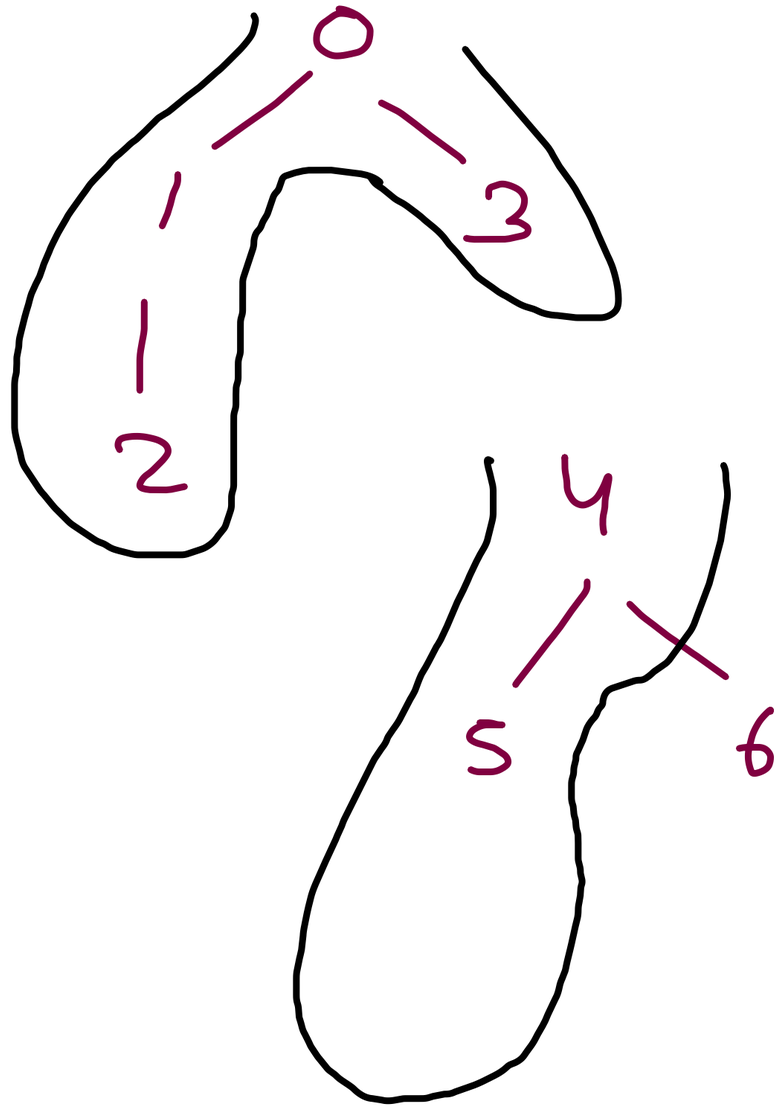


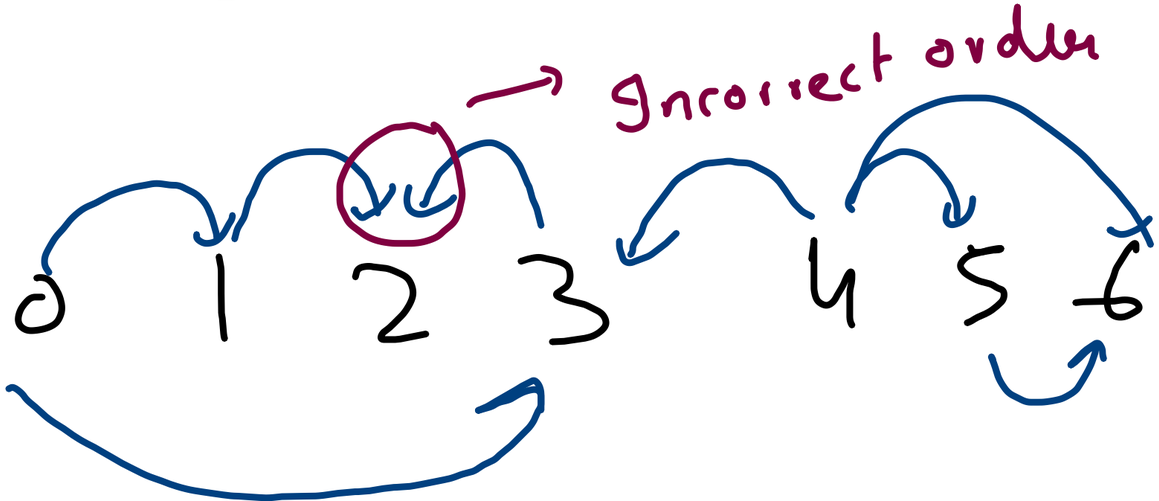
{Reverse Printed}



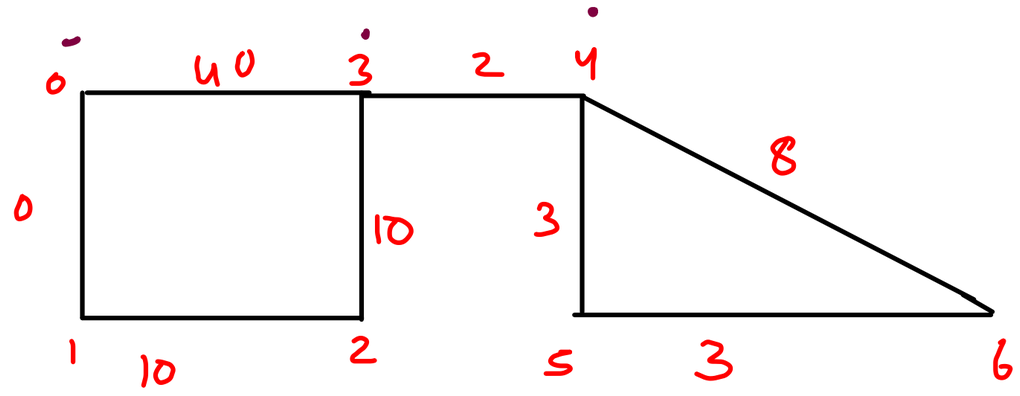
Why not use Preorder?





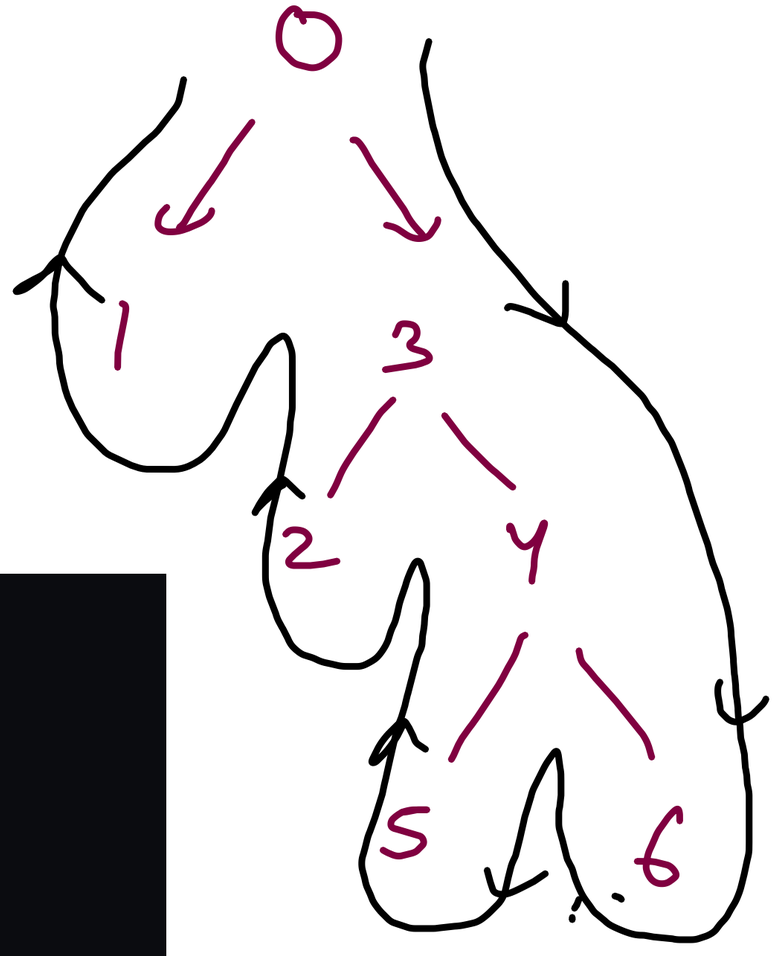


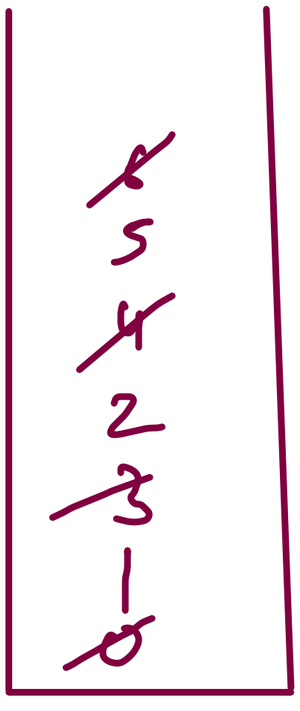
Iterative DfS



O(V+E)

Use this to avoid Stack overflow.






```java
while(st.size() > 0) {
  Pair rem = st.pop();

  if(vis[rem.node] == true) continue;

  vis[rem.node] = true;

  System.out.println(rem.node + "@" + rem.psf);

  for(Edge e : graph[rem.node]) {
      if(vis[e.nbr] == false) {
          st.push(new Pair(e.nbr,rem.psf + e.nbr));
      }
  }
}
```