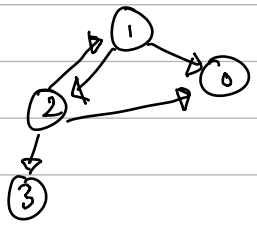
Graph

- Traversal In Graph
  - BFS
  - DFS
- Cycle Detection in Graph
  - UnDirected Graph
    - BFS
    - DFS
  - Directed Graph
    - BFS
    - DFS
- Topological Sorting
  - BFS (KAHN's Algorithm)
  - DFS
- Disjoint Set Union
  - Optimization using Path Compression & Rank.
- Bipartite Graphs
- Shortest Distances
- Minimum Spanning Tree.
- Strongly Connected Components
  - Kosarajus's Algorithm.
- Eulerian Geometry.

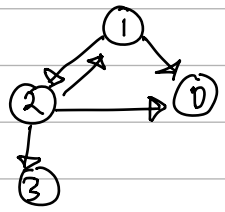
BFS & DFS

✓ For graph having nodes like 0, 1 .... n-1 we can track visited as `vector<bool> visited(n, false);`

DFS



BFS



Best suited for minimum Distance finding.

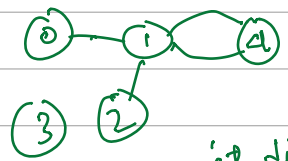
Cycle Detection

Undirected :-

in BFS;

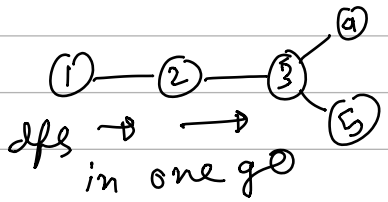
```
q.push({node, -1})                    // Parent
while(!q.empty()){
    node, prev = q.pop();
    // visited[node] = true;          // This is not necessary *
    for(int next : node){
        if(next == prev) continue
        if(visited[next]) CYCLE
        q.push({next, node});
        visited[next] = true;          // This is important *
    }
}
```

* For a testcase like

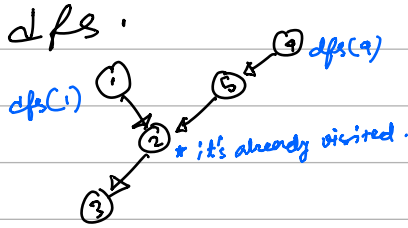


in gFg it didn't pass with...

Directed :- Difference b/w directed vs undirected with dfs.



1 visited array needed



2 visited array needed.
① global visited array (global visited needed otherwise TLE) gfg
② in dfs visited array

https://www.geeksforgeeks.org/

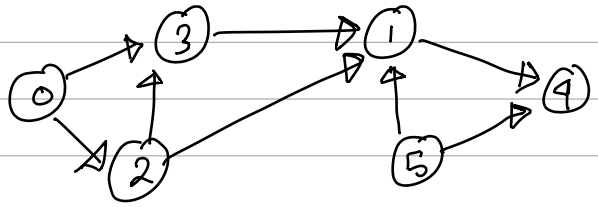
Topological Sorting

if (u) → (v) then u must occur before v.

Topological Sorting is only possible for DAGs. (Directed, No cycle).

Multiple Topological sorting is possible.

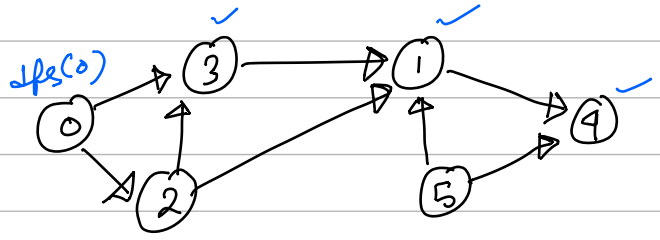
E.g.

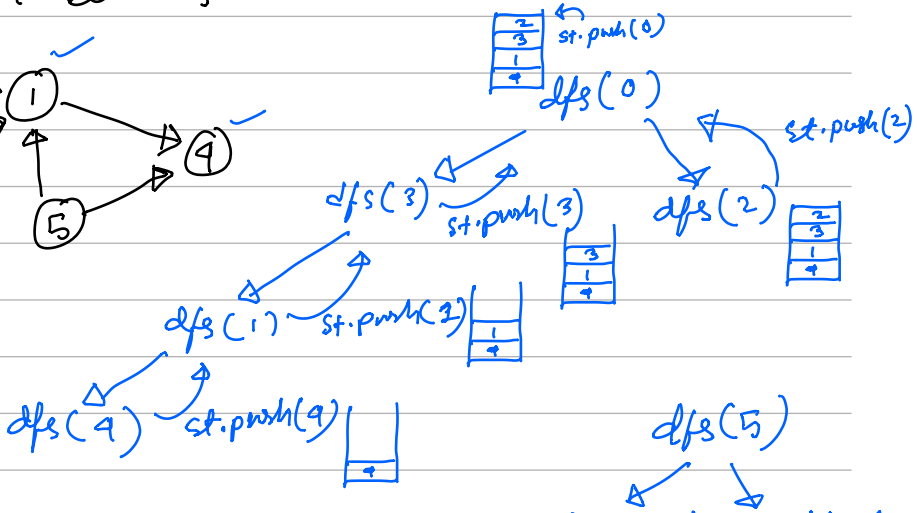


adj
0 → 3, 2
1 → 4
2 → 3, 1
3 → 1
4 →
5 → 1, 4

Topological Sorting

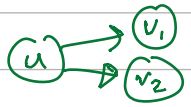
Using DFS :- DFS is topological sorting coz elements occur in their respective order.





At then end of dfs, i.e. returning from node push node value.

First push children then push yourself.



Possible Topo order : u, v1, v2 or u, v2, v1

Topological Sort + Cycle Detect DFS

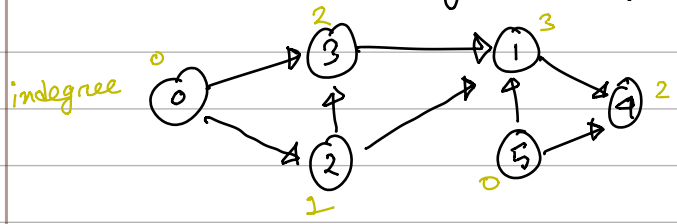
```
vector<int> topoOrder;
bool isCycle = False
vector<int> visited;

void DFSTraverse ( adj , node, unordered_map<int, bool> & seen )
{
    visited [node] = True
    seen [node] = True

    for (int next: adj[node])
    {   if (seen [next]) {
            isCycle = True
            return }
        if ( visited [next]) continue;
        else DFSTraverse ( adj, next , seen) }

    seen [node] = False
    topoOrder . pushback ( node)
}
```

→ Topological Sorting Using BFS (KHAN's Algorithm)



adj
0 → 3, 2
1 → 4
2 → 3, 1
3 → 1
4 →
5 → 1, 4

① push 0, 5 to queue
② when an element is processed, decrease indegree of its children.
③ if children's indegree == 0 push children to queue.

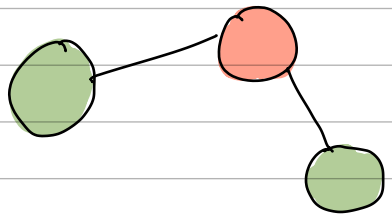
→ Detect Cycle using BFS :-

return length of topological sorted list != no. of nodes.

<< Course Schedule I & II >>

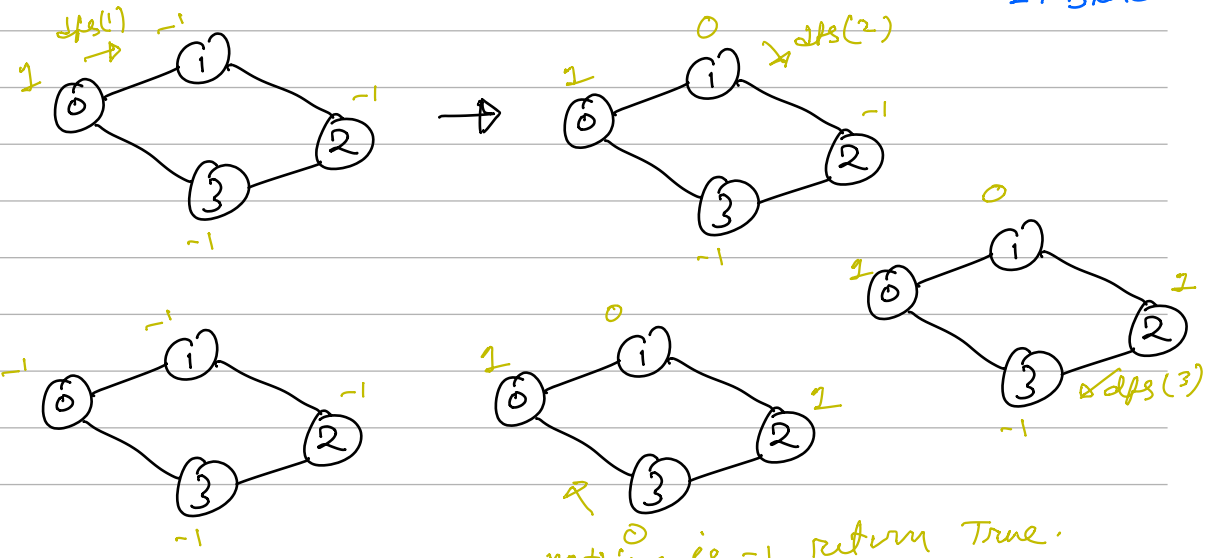
https://leetcode.com/problems/

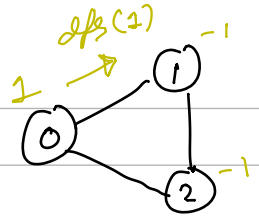
→ Bipartite Graph

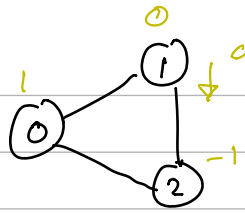


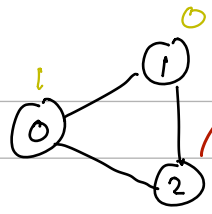
no 2 consequitive node have same colour.

→ Using DFS :- # Use array colours [-1, -1, -1, -1] -1 : unvisited, 0 : Red, 1 : Black

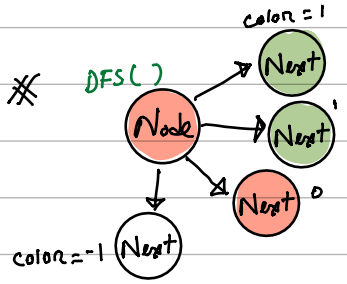








* colour[2] == (2)'s child's colour.



* in DFS of Node, while going through children of Node.

color[Next] == color[Node] Not Bipart!

color[Next] is opposite of Color[Node] No problem

color[Next] == -1 (unvisited) call dfs(Next)

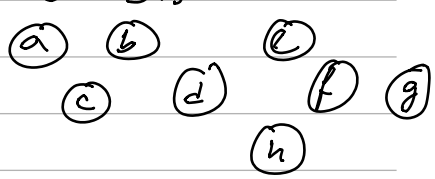
→ Similarly for BFS.

Disjoint Set Union / Union Find

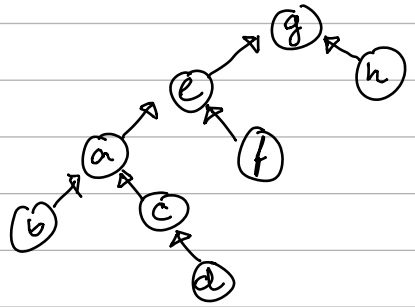
disjoint set → $S_1 \cap S_2 = \phi$

nodes : [a, b, c, d, e, f, g, h]

parent : [a, b, c, d, e, f, g, h]



after changing union and all;



nodes:	a	b	c	d	e	f	g	h
parent:	e	a	a	c	g	e	g	g

```
int findParent(int node, vector<int> & Parent){
    if( parent[node] == node) return node;
    return findParent( Parent[node] );
}
```

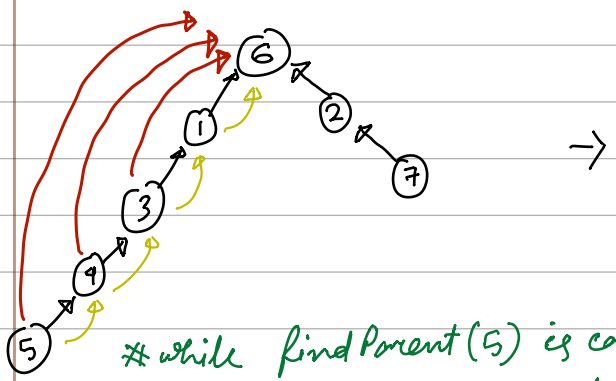
```
void union(int a, int b, vector<int>& parent){
    int parent of a = findParent(a, Parent)
    int parent of b = findParent(b, Parent)
    if (parent of a != parent of b){
        parent[parent of a] = parent of b } }
```

Time Complexity = O(V+E)

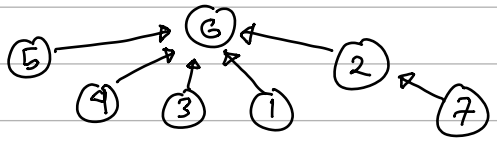
→ Optimization Using Path Compression and Rank :-

Path compression :-

```
// int findParent(int node, vector<int> & Parent){
    if( parent[node] == node) return node;
    return Parent[node] = findParent( Parent[node] );
}
```

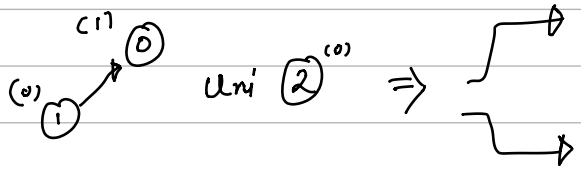


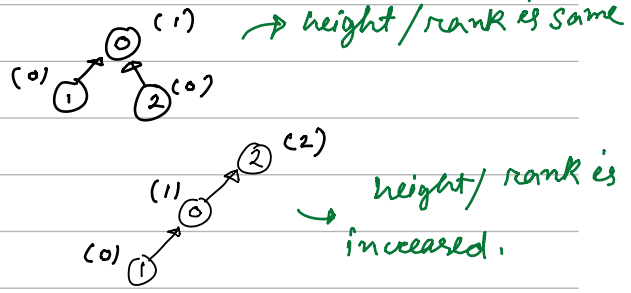
* while findParent(5) is called update 3, 4 and 5's parents to 6.



* findParent O(n) recursion time complexity to O(1) improvement.

* optimization with Rank :-





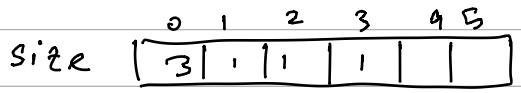
→ height/rank is same

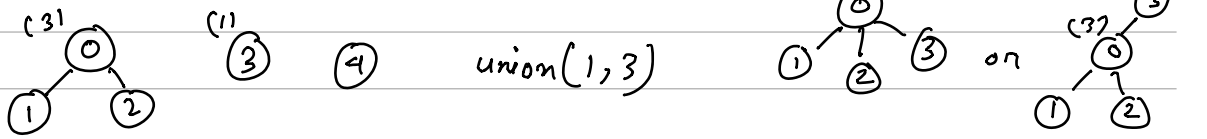
→ height/rank is increased.

```
// void Union(int A, int B, vector<int> & parent){
    int parentA = findParent(a, parent)
    int parentB = findParent( b, parent)
    if ( parentA == parentB) return;
    if ( rank[parentA] > rank[parentB] ){
        parent[parentB] = parentA
    }
else if (rank[parentA] < rank[parentB]){
        parent[parentA] = parentB
    }
else { parent[parentB] = parentA
        rank[parentA]++;
    }
}
```

→ Optimization using path compression & Size :-

Optimization using size :-

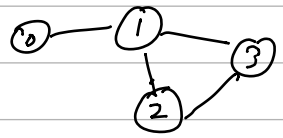





```
void makeUnion( int a, int b, vector<int> & parent )
{
    int pA = findParent(a, parent)
    int pB = findParent(b, parent)
    if (pA == pB) return

    if ( size[pA] > size[pB] ){
        parent[pB] = pA
        size[pA]++
    }
    else if ( size[pA] < size[pB] ) {
        parent[pA] = pB
        size[pB]++
    }
    else {
        parent[pB] = pA
        size[pA]++
    }
}
```

→ Detecting cycle Using DSU –



// For undirected graph, make all node in union if there exist an edge. (iterate node in increasing order).
if any any node reached again (parents are same) there's a cycle.

→ Number of operations to make network connected :–

1319. Number of Operations to Make Network Connected

Medium | Topics | Companies | Hint

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.

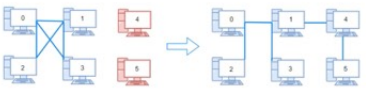
You are given an initial computer network connections. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected.

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.

Example 1:

Input: n = 4, connections = [[0,1],[0,2],[1,2]]
Output: 1
Explanation: Remove cable between computer 1 and 2 and place between computers 1 and 3.

Example 2:



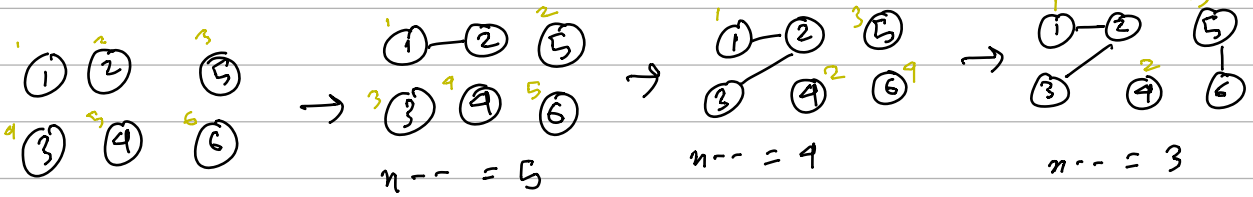
Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]
Output: 2

Example 3:

Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]
Output: -1
Explanation: There are not enough cables.

n = 6
edges = [{0,1}, {0,2}, {0,3}, {1,2}, {1,3}]
i →

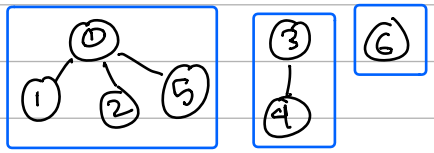
iterate over edges & count extra edges.



#⭐

at each moment whenever there an union Total no. of universal disconnect dicreases; even if union is getting in a differen set. like in (5)–(6) set aside 1 2 3 set.

→ Count Unreachable Pairs



unordered map 0 : 4
3 : 2
6 : 1

how to calculate no. of pairs
count += size x remaining
e.g :– 4 x 3 + 2 x 1 + 1 x 0 = 19
or
2 x 5 + 9 x 1 = 19
or
1 x 6 + 2 x 9 = 19

#⭐, #Revision, #Trick

https://leetco...

https://leetcode.com/proble...

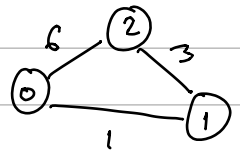
DSU Question << ✓satisfyability of equations
✓no. of operation to make network connected
✓count unreachable pairs

https://leetcode.com/problems/count-...

Dijkstras

* usage of min heap

priority_queue<P, vector<P>, greater<P>>



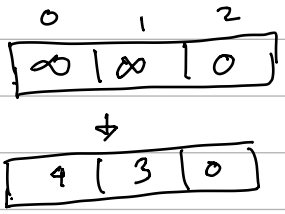
adj

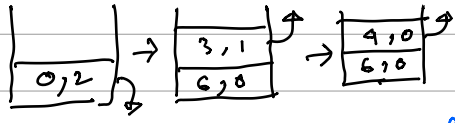
0 - {1,1}, {2,6}

1 - {0,1}, {2,3}

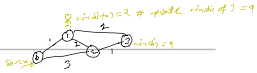
2 - {0,6}, {1,3}

dist from 2 :

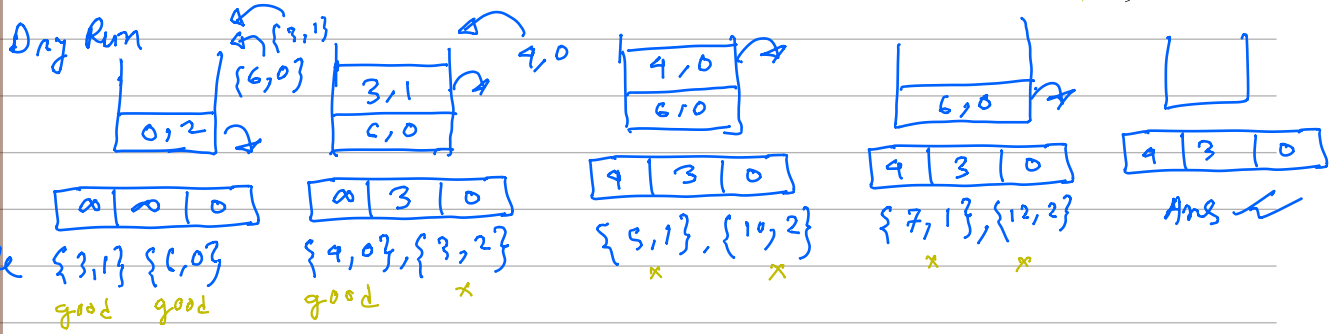




update in result & push in pq.



Dry Run



possible {3,1} {6,0} good good

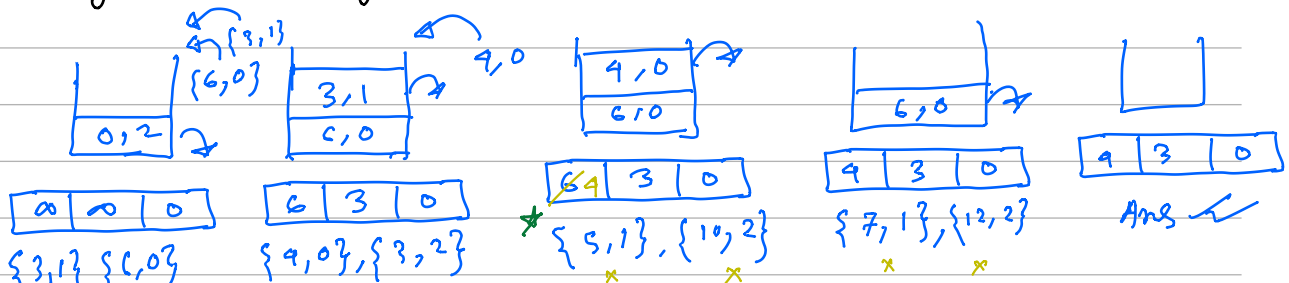
{4,0}, {3,2} good ×

{5,1}, {10,2} × ×

{7,1}, {12,2} × ×

Ans ✓

→ Dijkstra using set :- (ordered set)



possible {3,1} {6,0} good good

{4,0}, {3,2} good ×

{5,1}, {10,2} × ×

{7,1}, {12,2} × ×

Ans ✓

In set at this point

We can erase (6,0) from set (optimization)

* set.erase({random thing not present in set}) → doesn't give error

✓ Why Queue is not used ?

⇒ Priority Queue presents the smallest possible answer till that point. Queue may iterate over all possible min distances before the smallest possible.

→ Time complexity :-

while (pq != empty) → v times

pq.pop → log V

for (next : node) → E

pq.push → log V

Total

$v(\log v + E \log v)$

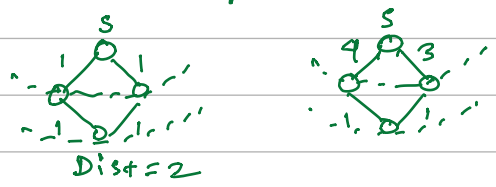
$= E v \log(v)$

worst case $E = v-1$

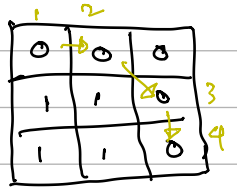
$\simeq v(v-1)\log(v)$

$\simeq v^2 \log(v)$

BFS Doesn't work for shortest path whenever there's weight in path



→ Shortest path in Binary Matrix



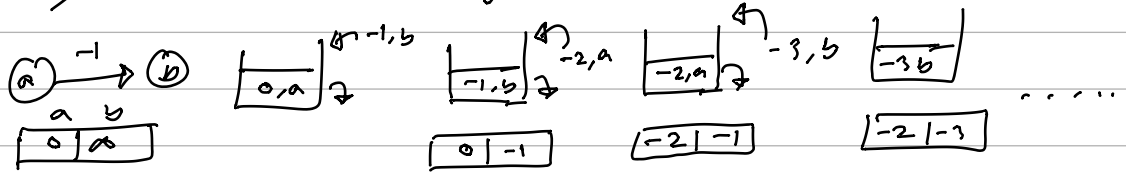
ans = 4

BFS ✓ Dijkstra ✓ { priority queue ✓ & queue ✓

Dijkstra Questions << Network Delay time
Path with minimum effort
shortest path in Binary matrix >>

→ Problems of Dijkstra :—

1) can't work for -ve edges.
2) can't detect -ve cycle.

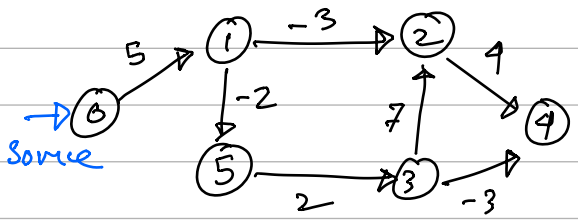



```
vector<int> distance;
priority_queue<P, vector<P>, greater<P>> pq / set<P>

while(!pq.empty())
    node = pq.top().second
    dist = pq.top().first        pq.pop()
    for( ele : adj[node])
        next = ele[0]
        wt = ele[1]
        new_dist = dist + wt
        if(new_dist < distance[next]){ // st.erase({distance[next], next})
            distance[next] = new_dist
            pq.push({new_dist, next})}
```

Bellman Ford Algo :-

① works only on directed graph.



edges

(3, 2, 7)
(5, 3, 2)
(0, 1, 5)
(1, 5, -3)
(1, 2, -2)
(3, 4, -2)
(2, 4, 3)

{ iterate over edges & update distance array

↳ Do this not more than $v-1$ time (no. of vertices)

distance = | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
(indices: 0 1 2 3 4 5)

```
for (int i=0 ; i< v-1 ; i++){
    for ( e : edges){
        u = e[0]
        v = e[1]
        wt = e[2]
        if ( distance[u] != ∞
             && distance[u] + wt < distance[v])
            {distance[v] = distance[u] + wt} }}
```

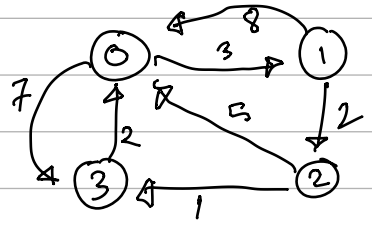
* edges array can be in any order ; but every $v-1$ iteration must follow the same order of edges.

* if done more than $v-1$ time ; -ve cycle will come at play

Time complexity Dijkstra : $E \log V$
Bellman Ford : EV.

Floyd Warshall Algo :-

: min distance from $u \to v$ check u to v going through all possible vertices w. $u \to w \to v$.



	0	1	2	3
0	0	3	∞	7
1	8	0	2	∞
2	5	∞	0	1
3	2	∞	∞	0

grid

update	via 0	via 1	via 2	via 3
grid[0][1]	0→0→1 3	0→1→1 3	0→2→1 ∞	0→3→1 11
grid[0][2]	0→0→2 ∞	0→1→2 5	0→2→2 5	0→3→2 13
[0][3]	0→0→3 7	0→1→3 15	0→2→3 6	0→3→3 6
[1][0]	1→0→0 8	1→1→0 8	1→2→0 7	1→3→0 5
[1][2]	1→0→2 ∞	1→1→2 2	1→2→2 2	1→3→2 10
[1][3]	1→0→3 15	1→1→3 15	1→2→3 3	1→3→3 3
[2][0]	2→0→0 5	2→1→0 ∞	2→2→0 5	2→3→0 3
[2][1]	2→0→1 ∞	2→1→1 ∞	2→2→1 ∞	2→3→1 6
[2][3]	2→0→3 12	2→1→3 ∞	2→2→3 1	2→3→3 1
[3][0]	3→0→0 2	3→1→0 13	3→2→0 12	3→3→0 2
[3][1]	3→0→1 5	3→1→1 5	3→2→1 ∞	3→3→1 5
[3][2]	3→0→2 ∞	3→1→2 7	3→2→2 7	3→3→2 7

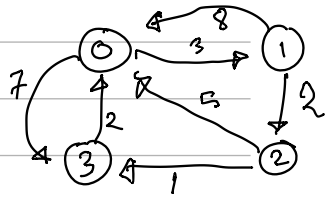
	0	1	2	3
0	0	3	∞	7
1	8	0	2	∞
2	5	∞	0	1
3	2	∞	∞	0

	0	1	2	3
0	0	3	5	7
1	8	0	2	15
2	5	∞	0	1
3	2	5	7	0

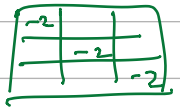
	0	1	2	3
0	0	3	5	6
1	7	0	2	3
2	5	∞	0	1
3	2	5	7	0

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0

```
for(int via=0; via<V; via++)
    for(int i=0; i<V; i++)
        for(int j=0; j<V; j++)
            grid[i][j] =
            min(grid[i][j], grid[i][via] + grid[via][j]);
```



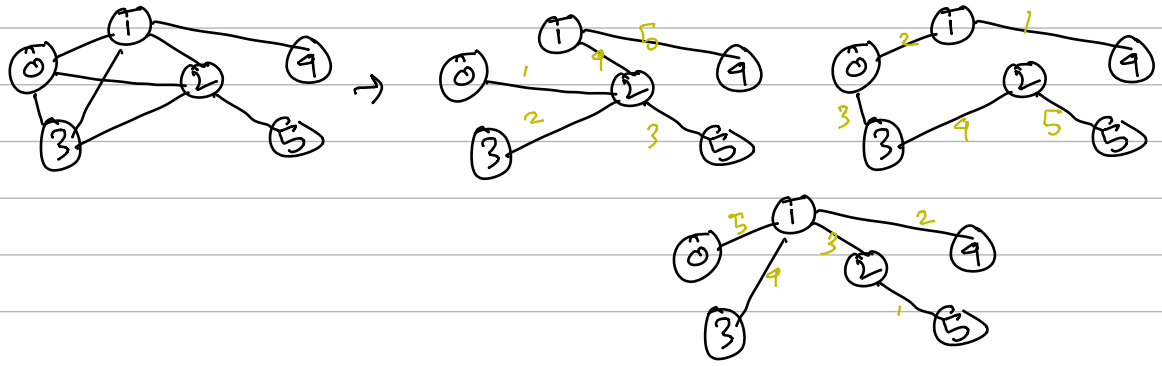
* For checking -ve cycle check diagonal elements. if any one is -ve, that means -ve cycle present.



Minimum Spanning Tree :-

$G(V, E) \longrightarrow$ connected subgraph where -

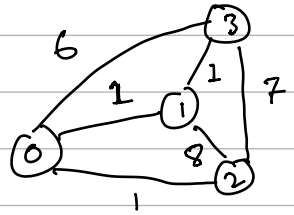
1) All the vertices of G are connected.
2) It has $v-1$ number of edges.



Prim's Algorithm :-

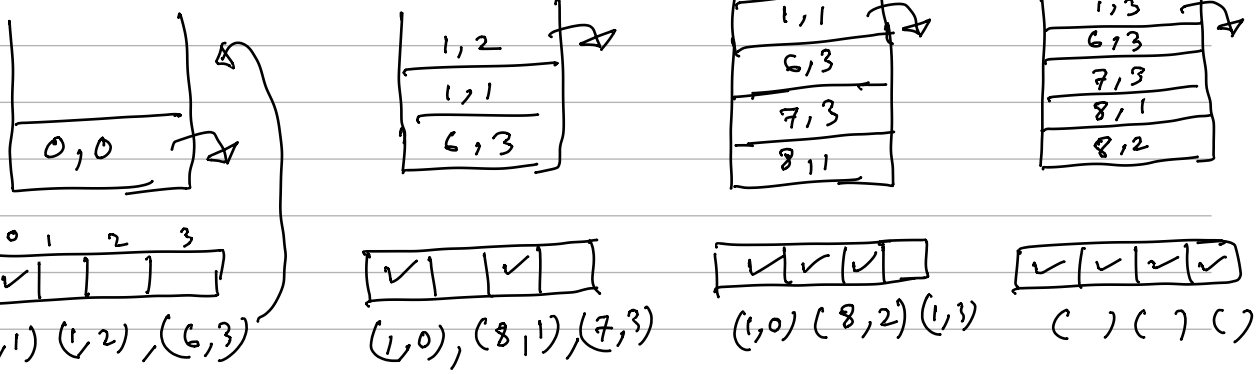
// at every point take the minimum edge and add it to mst

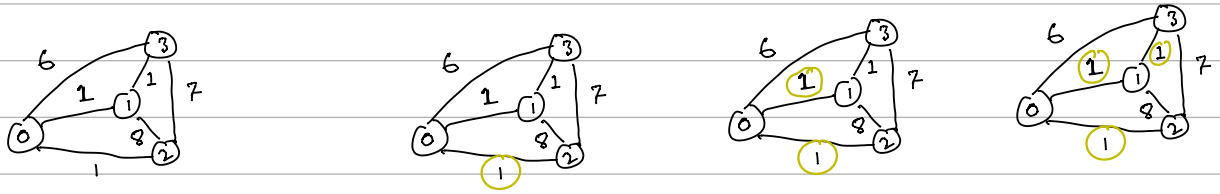
```
priority_queue< P, vector<P>, greater<P>> pq;
pq.push({0, 0})
```

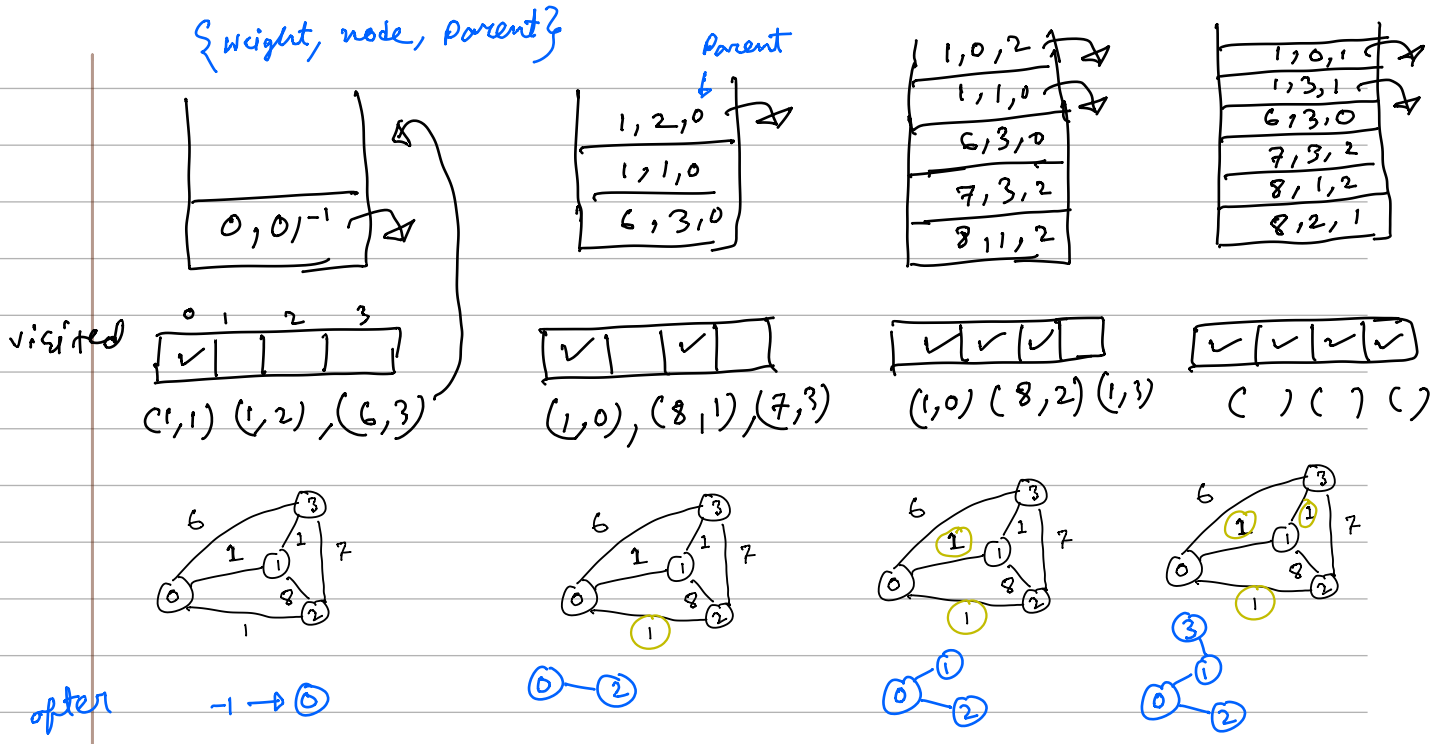


adj

0 : 1,1 | 2,1 | 3,6
1 : 0,1 | 2,8 | 3,1
2 : 0,1 | 1,8 | 3,7
3 : 0,6 | 1,1 | 2,7







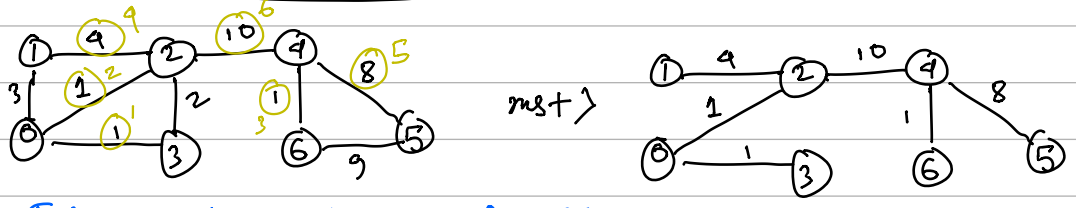
* when a not visited node gets visited it's done so with using the cheapest path possible (it starts from node 0)

```
vector<bool>  inMst(V, false);
vector<int>   parent(V, -1);
priority_queue<P, vector<P>, greater<P>>   → {wt, {node, parent}}
pq.push({0, 0, -1})

while(!pq.empty())                                  → E
    node = pq.top().second.first
    wt = pq.top().first ;   pq.pop();               → log E
    if(visited[node]) continue;
    mstSum += wt;   parent[node] = pq.top().second.second
    for(ele : adj[node])
        next = ele[0]   next_wt = ele[1]
        if(!visited[node]){
            pq.push({next_wt, {next, node}});       → log E
```

O(E log(E))

Kruskal's Algorithm :-



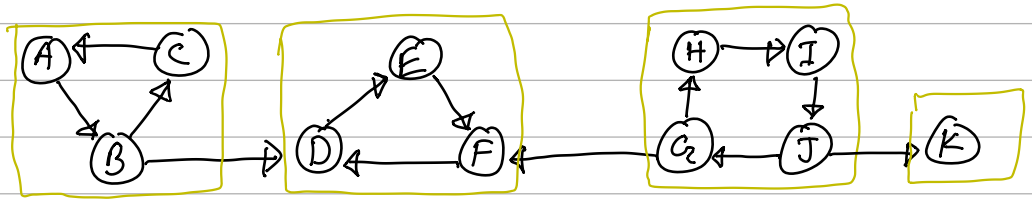
Take smallest edge & union it.
Don't take if already connected.

```
sort(adj.begin(), adj.end())        adj -> {wt, {u,v}}
for(ele : adj){                  -> E times
    wt = ele[0]
    u = ele[1]
    v = ele[2]
    if(findParent(u) != findParent(v))
    {   makeUnion(u, v);
        cost += wt;  }
```

4×α

cost = E × (4 × alpha)

Kosaraju's Algorithm / Strongly Connected Component

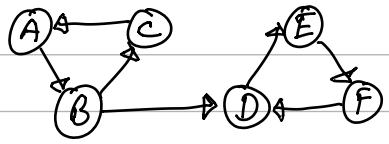


strongly connected components, {A, B, C} A → B & B → A both possible.

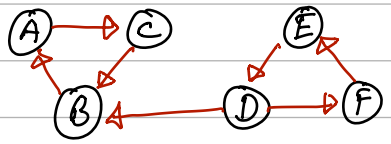
A, D are not strongly connected A → D possible but not D → A

✓ no. of strongly connected components = 4

✓✓ Intution :-

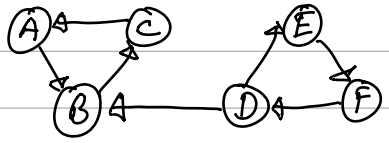


Reverse the edge

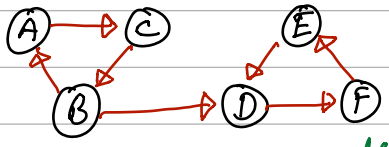


*DFS from A gives {A, C, B} SSC.

But what if the graph was,



reverse →



*DFS from A will go towards D, E, F.

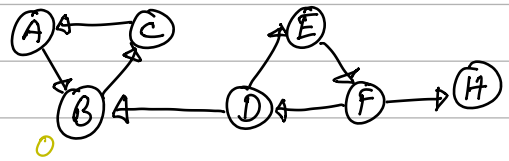
For this case we need to start DFS from D, E, F side not A, B.

*So we need a Topological order to start DFS from.

→ Kosarajus Algo :—

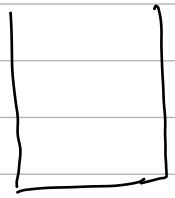
1. Generate topological order stack
2. Generate Reverse edge graph
3. DFS from stack top.

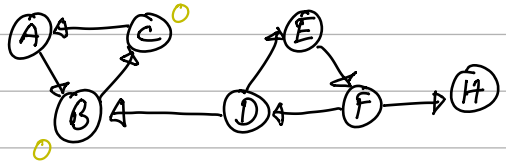
Dry Run,



B has children
don't push.

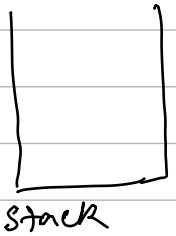
visited
B

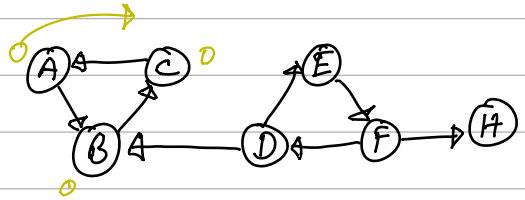




C has children.

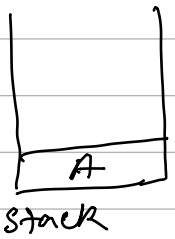
visited
B, C

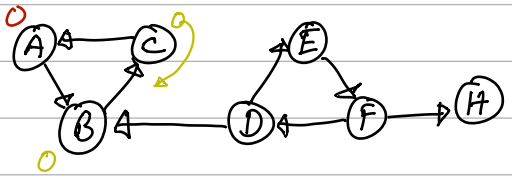




A has children,
but visited, pop

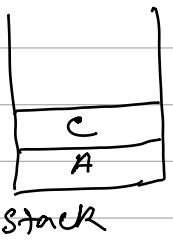
visited
B, C, A

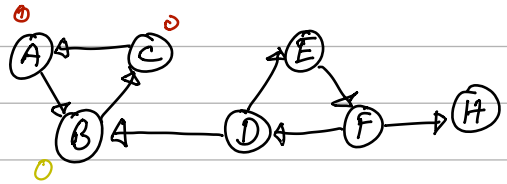




C has no children
push C

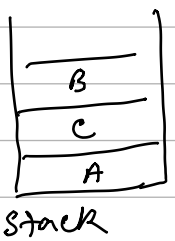
visited
B, C, A

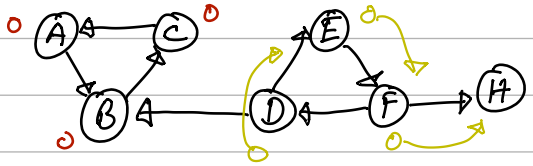




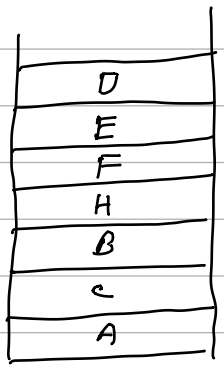
B has no children
push B

visited
B, C, A

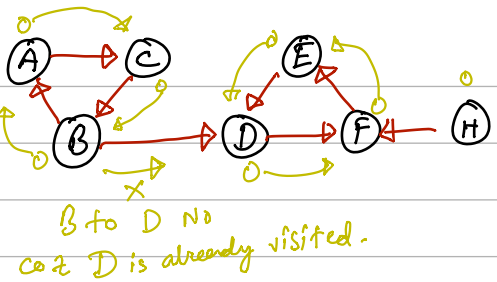




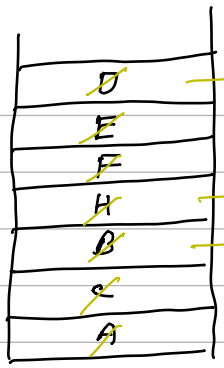
visited
B, C, A,
D, E, F,
H



Now Reverse edge & Run DFS From the top of Stack,



B to D No
coz D is already visited.



D → E → F { D, E, F }

H { H }

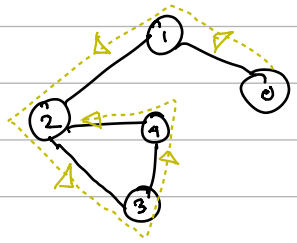
B → C → A { B, C, A }

SSC

Eulerian Geometry :-

Eulerian path :-

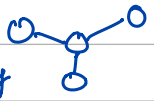
A path of edges that visits all the edges in a graph exactly once.



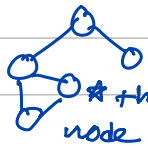
eulerian path : 0, 1, 2, 3, 4

not all graph have eulerian path.
not all node could give eulerian path starting from it.

e.g



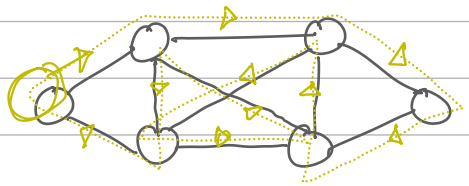
eg.



* this node can't

Eulerian Circuit :-

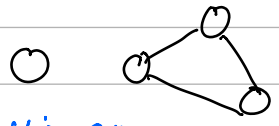
An eulerian path that starts and ends on the same node called Eulerian circuit



if there's Eulerian Circuit then every node can be the starting node.
if a graph does not have Eulerian Circuit. then :-

① Either you won't be able to come back to start node
② or you will not be able to visit all edges of graph.

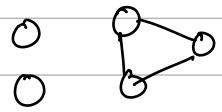
miscellenious : ①



all edges ✓
once ✓
start node == finish node ✓

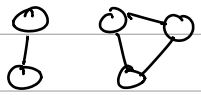
it's an Eulare path & circuit.

②



Eular path ✓
Eular circuit ✓

③



all edges ✗

it's not Eulare path or circuit

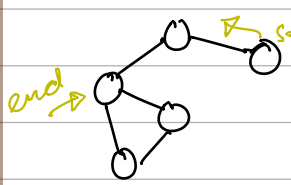
* all non zero degree nodes must connect to a single component.

Fact 1 :- all nodes must have even degree to have Eular Circuit

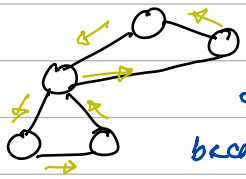
} 2 edge needed for incoming & outgoing.

* a graph which has Eular path but doesn't have Eular Circuit called Semi-Eularian Graph.
otherwise it's called Eularian Graph.

✓ semi Eularian : 1) all edges ✓
2) start == end ✗



semi eularian
start & end vtx
always odd indegree.



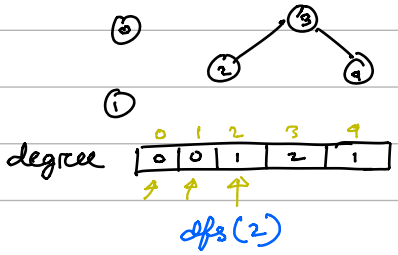
all even degree became Eularian

⑧ no of odd degree = 2 Eular Path ✓
Eular circuit ✗

no. of odd degree = 0 Eular circuit ✓

→ check for Non Eulerian, Semi Eulerian or Eulerian :-

① check all non zero degree nodes are connected.



after dfs(2)

visited :

F	F	T	T	T

all non zero degree must be visited.

② count no. of nodes where degree = odd.