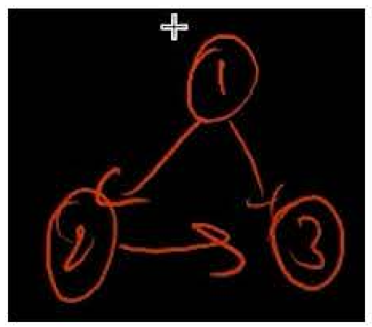
Different approaches to solve graph questions

1. Identify bfs/dfs
2. Apply appropriate strategy to solve the problem
   i. variation of **get connected components**
   ii. variation of minimum spanning tree **-Kruskals algo**
   iii. Variation of **topological sorting** (One has to come before other)
   iv. Which uses priority queue (Variation of **Dijkstra**) -> When each step has to choose min or max among all adjascent edges
   v. Variation of articulation point - Removal of an edge/node will bread the system into n parts , find the edge/node
   vi. Using **disjoint sets -** Questions in which we dynamically add edges to graph while solving the problem OR questions that has to check if edge or vertex belongs to a component OR Anything that involves **grouping /merging**
   vii. Variation of hamiltoninan or eulers path

Detection of a cycle using Disjoinnt Set for directed graph is not possible BUT if we know that the graph has a cycle, we can detect the edge which is causing the cycle



<u>Mother Vertex-</u>> vertex through which we can reach all the other vertices of the Graph .First element of the topological sort is the mother vertex

Graphs

```cpp
void DFS_Traversal(int node){
    visited[node] = 1;
    cout << node << " ";
    for(int i = 0; i < v; i++){
        if(adj[node][i] == 1 && visited[i] != 1)
            DFS_Traversal(i);
    }
}
```

DFS

```cpp
void BFS_Traversal(int src){
    vector <int> visited (v, 0);
    queue<int> q;
    //Starting with source node
    q.push(src);
    visited[src] = 1;
    while(!q.empty()){
        int node = q.front();
        q.pop();
        cout << node << " ";
        //Visiting all connected nodes and enqueue unvisited nodes for exploration
        for(int j = 0; j < v; j++){
            if(adj[node][j] == 1 && visited[j] != 1){
                visited[j] = 1;
                q.push(j);  //Enqueing unvisited node
            }
        }
    }
    cout << endl;
}
```

T: O(V+E)
S : O(V)

BFS

```
  0     3     4
  |     |     |
  |     |     |
  |     |     |
  1     2     5---6
Connected comp -> {[0,1], [2,3], [4,5,6]}
```

```cpp
void getConnectedComponent(int src, vector<bool> &visited, vector<int> &connectedComp){
    visited[src] = true;
    connectedComp.push_back(src);
    for(int &edge : adj[src]){
        if(visited[edge] == false){
            getConnectedComponent(edge, visited, connectedComp);
        }
    }
}
```

```cpp
vector<vector<int>> comps;    //To store all connected components
vector<bool> visited(7, false);
for(int i = 0; i < 7; i++){
    if(visited[i] == false){
        vector<int> comp;     //To store one single component
        g.getConnectedComponent(i, visited, comp);
        comps.push_back(comp);
    }
}
```

get Connected Components

```cpp
//If a graph is connected, all nodes must be visited in one traversal
vector<bool> visited(7, false);
vector<int> comp;
g.getConnectedComponent(0, visited, comp);
for(bool x : visited){
    if(x == false) cout << "false" << endl;
}
cout << "true" << endl;
```

Is graph connected

```cpp
void hamiltonmian(vector<bool> visited, int src, string psf){
    visited[src] = true;
    //Base case
    bool allVisited = true;
    for(bool x : visited){
        if(x == false){
            allVisited = false;
            break;
        }
    }
    if(allVisited){
        cout << psf + to_string(src) ;
        for(int node : adj[src]){
            if(node == psf[0] - '0'){
                cout << "*" << endl;
                return;
            }
        }
        cout << "." << endl;
        return;
    }
    //Code
    for(int node : adj[src]){
        if(visited[node] == false)
            hamiltonmian(visited, node, psf + to_string(src));
    }
    visited[src] = false;
}
```

```
0_____3_____4
|     |     |\
|     |     | \
|_____|_____|__\
1     2     5   6

0123456.
0123465.
0125643*
0346521*
```

jayte hue visit kare
vapas ate hue unvisit kare
base case-All nodes visited

hamiltonian path

A hamiltonian path is such which visits all vertices without visiting any twice. A hamiltonian path becomes a cycle if there is an edge between first and last vertex.

```cpp
bool hasCycleDFS(int node, vector<vector<int>>& adjList, vector<bool>& visited, int parent) {
    visited[node] = true;

    for (int neighbor : adjList[node]) {
        if (!visited[neighbor]) {
            if (hasCycleDFS(neighbor, adjList, visited, node))
                return true;
        }//* If we are visiting an already visited node which is not its parent => Cyclic
        else if (neighbor != parent) {
            return true;
        }
    }

    return false;
}
```

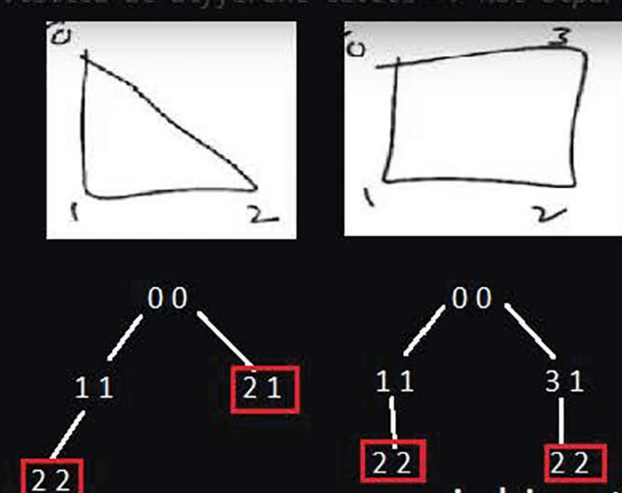
Cycle detection using DFS in undirected graph

The approach is the same for BFS
we create a queue {node, parent}
Whenever we are adding an adj node, we check if the adj node is visited, if the node if visited and is not parent then the graph is cyclic

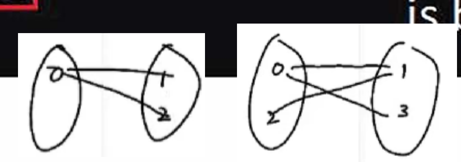
```cpp
bool Graph :: isBiparitie(int src, vector<int> &visited){
    // <node, level at which it is visited>
    queue<pair<int, int>> q;
    q.push({src, 0});
    while(!q.empty()){
        pair<int, int> node = q.front();
        q.pop();
        //if node is already visited
        if(visited[node.first] != -1){
            //visited-> contains level at which node was visited
            //node.second -> current level
            //If they both dont match => the node is being visited at different levels => Not bipartite
            if(node.second != visited[node.first]){
                return false;
            }
        }
        //If unvisited, mark 'visited at current level'
        visited[node.first] = node.second;
        for(int i : adj[node.first]){
            if(visited[i] == -1){
                q.push({i, node.second + 1});
            }
        }
    }
    return true;
}
```

This is true when the graph is cyclic, we further check if graph contains odd no of nodes or even no of node

If the node is visited at two different levels, then it has odd no of nodes
Else it has even no of nodes



is bipartite



A graph is said to be bipartite if its vertices can be devided into two mutually exclusive and exhaistive sets such tha all the edges are present accross the sets

All non cyclic graphs --> Bipartite
If cyclic-> Even no of Nodes -> Bipartite
Odd no of nodes -> Not Bipartite

If there are dirferent components in a graph --> All the commponents must be bipartite, for graph to be bipartite,

```cpp
void Graph :: topologicalSort(int src, vector<bool> &visited, stack<int> &s){
    visited[src] = true;
    for(int node : adj[src]){
        if(visited[node] == false)
            topologicalSort(node, visited, s);
    }
    s.push(src);
}
```

topoological sort

```cpp
vector<bool> visited(v, false);
stack<int> s;
for(int i = 0; i < v; i++){
    if(visited[i] == false){
        g.topologicalSort(i, visited, s);
    }
}
while(!s.empty()){
    cout << s.top() << endl;
    s.pop();
}
```

<u>Topological sort--></u> It is a permutation of all vertices for a directed graph in called a topological sort if for all directed edges uv, u appears before v in the graph.

TOPOLOGICAL SORT IS POSIBLE ONLY FOR DAG

Cycle Detection

1. For undirected graph

```cpp
bool hasCycleDFS(int node, vector<vector<int>>& adjList, vector<bool>& visited, int parent) {
    visited[node] = true;

    for (int neighbor : adjList[node]) {
        if (!visited[neighbor]) {
            if (hasCycleDFS(neighbor, adjList, visited, node))
                return true;
        }//* If we are visiting an already visited node which is not its parent => Cyclic
        else if (neighbor != parent) {
            return true;
        }
    }

    return false;
}
```

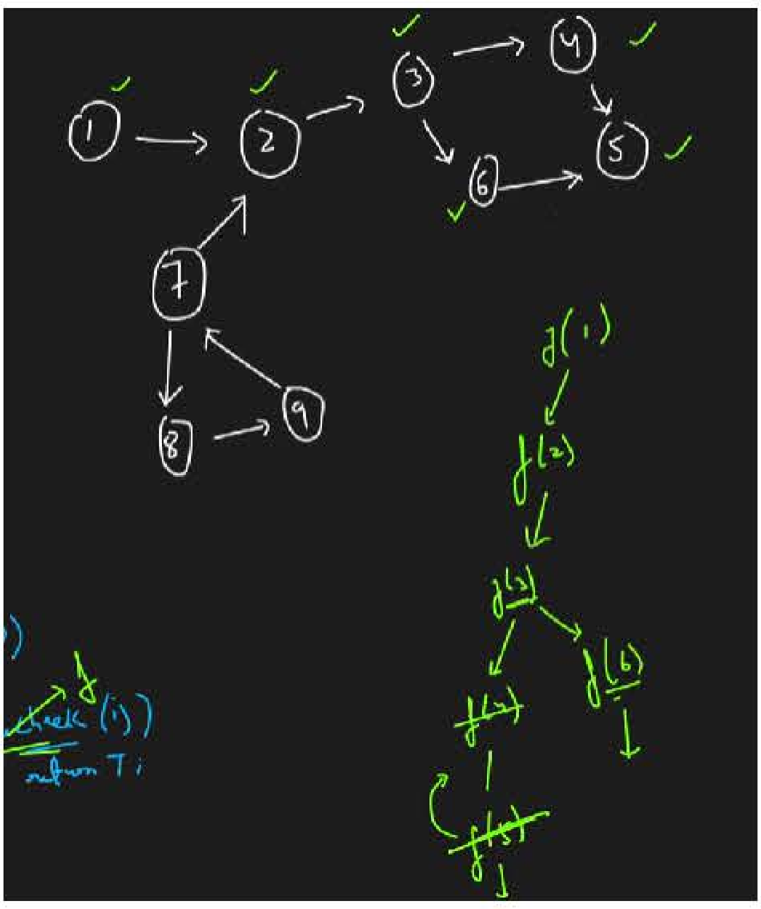
using DFS

```cpp
bool hasCycleBFS(int node, vector<vector<int>>& adjList, vector<bool>& visited) {
    queue<pair<int, int>> q;
    q.push({node, -1}); // {node, parent}

    visited[node] = true;
    while (!q.empty()) {
        int currNode = q.front().first;
        int parent = q.front().second;
        q.pop();

        for (int neighbor : adjList[currNode]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push({neighbor, currNode});
            } else if (neighbor != parent) {
                return true;
            }
        }
    }

    return false;
}
```

using bfs

2.For directed graph

The above algorithm will not work.



Consider this example.
When we call DFS continiously we go in the path
1->2->3->4->5
Since there are no adjj nodes for 5, we return
At 3 we have another adj node 6 so we go to 6
AT 6 we find that 5 is an adj node which is already visited => CYCLIC
BUT ITS NOT TRUE
So we need another dfsVis array which will keep track of all nodes visited while dfs traversal

```cpp
bool hasCycleDFS(int node, vector<vector<int>>& adjList, vector<bool>& visited, vector<bool>& dfsVisited) {
    visited[node] = true;
    dfsVisited[node] = true;

    for (int neighbor : adjList[node]) {
        if (!visited[neighbor]) {
            if (hasCycleDFS(neighbor, adjList, visited, dfsVisited))
                return true;
        } else if (dfsVisited[neighbor]) {
            return true;
        }
    }

    dfsVisited[node] = false;
    return false;
}
```

using dfs

Using BFS -> We use Khans algorithm for topological sorting. WKT topo sort is possible for only DAG. So we use the reverse approact here. If we are not able to topologically sort the nodes => CYCLIC
Approach -> Code is same as Khans algo but here we maintain a counter.
incriment counter for every node popped from the queue
if cnt != n then the graph is cyclic

Single Source Shortest Path Algos

```cpp
void addEdge(vector<pair<int, int> > adj[], int u, int v, int wt){
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}

void shortestPath(vector<pair<int, int> > adj[], int V, int src){
    priority_queue<iPair, vector<iPair>, greater<iPair>> pq; //Min heap

    // Create a vector for distances and initialize all distances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert source itself in priority queue and initialize its distance as 0.
    pq.push(make_pair(0, src));
    dist[src] = 0;

    // Looping till all distances are not finalized
    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        // Get all adjacent of u.
        for (auto x : adj[u]) {
            int v = x.first;
            int weight = x.second;

            // If there is shorted path to v through u.
            if (dist[v] > dist[u] + weight) {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
```

T : O(E + V logV)
S : O(V)

Single source shortest path to all other nodes

Dijstras Algo

Drawback -> Dijkstras ALgorithm may not work for negative weighted edges

Replacing queue with priority queue in BFS -> Dijskstras Algo

Time Complexity:

The time complexity of Dijkstra's algorithm will be O(E + V logV). This is because, we are iterating over all the edges once during the entire run of the algorithm In each iteration, we are popping one node and pushing the unvisited neighbour nodes. Since the priority queue can contain all the vertices, the push or pop operation will be O(log V). Hence the total time complexity will be O(E) + O(V) * O(log V) = O(E + V log V).

Note: You can argue that we might be having multiple Pairs having the same node's value. So, the maximum size of the priority queue will be not O(N) but O(E). But, even if you replace log V with Log E (cost of one push/pop operation), then there will be no difference in the time complexity as: O(E + VlogE) = O(E + Vlog(V^2)) = O(E + 2V logV) = O(E + V logV) only.

Space Complexity:

We are taking a priority queue of Pair nodes. Hence, the space complexity will be O(N) where N = maximum Pair nodes in the queue, which is equivalent to O(V).

```cpp
class edge{
public:
    int u, v, wt;
    edge(int u, int v, int wt){
        this->u = u;
        this->v = v;
        this->wt = wt;
    }
};

void shortestPath(vector<edge> graph, int V, int src){
    // Create a vector for distances and initialize all distances as infinite (INF)
    vector<int> dist(V, INF);

    //Relaxing all the edges V-1 times
    for(int i = 0; i < V-1; i++){
        for(edge e : graph){
            if(dist[e.u] + e.wt < dist[e.v]){
                dist[e.v] = dist[e.u] + e.wt;
            }
        }
    }

    //Relaxing one mmore time to find if negative cycle is present
    bool flag = false;
    for(edge e : graph){
        if(dist[e.u] + e.wt < dist[e.v]){
            flag = true;
            cout << "Has negative weighted cycle" << endl;
            return;
        }
    }
    // Print shortest distances stored in dist[]
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}
```

T : O(V*E)
S : O(V)

Bellmanford Algorithm

This can be used for negative weighted graphs

DRAWBACK ->

Doesnot work for negative weighted cycle

Since this algotithm doesnt work for -ve weighted cycle, so this wont work for undirected grph with -ve weights coz for an edge u-v in undirected graph it is same as u->v and u<-v. So if there is negative weight this becomes -ve weighted cycle

In the Bellman-Ford algorithm, we perform the relaxation step V-1 times where V is the number of vertices in the graph. The reason why we need to perform V-1 relaxations is that, in the worst-case scenario, the shortest path from a source vertex to a destination vertex can pass through at most V-1 edges.

Minimum Spanning Tree ALgorithms

```cpp
class Pair{
    public :
        int v;  //Vertex
        int av; //Acquiring vertex
        int wt;  //Weight
        Pair(int wt, int v, int av){
            this->v = v;
            this->av = av;
            this->wt = wt;
        }
        Pair();
};
class Graph{
        int v;
        //vertex , weight
        list<pair<int, int>> *adj;
    public:
        Graph(int vertices);
        void addEdge(int src, int dest, int wt);
        void prims(int);
};
// Add edges to the graph
void Graph::addEdge(int src, int dest, int wt){
    adj[src].push_back({dest, wt});
    adj[dest].push_back({src, wt});
}

void Graph :: prims(int src){
    vector<bool> visited(v, false);

    // pair <weight, vertex, acquring vertex>
    priority_queue <Pair, vector<Pair>, myComparator> pq;
    pq.push(Pair(0, src, -1)); //-1-> Non existing vertex

    cout << "V AV WT" << endl << src << "  -  -" << endl;
    while(!pq.empty()){
        Pair p = pq.top();
        pq.pop();
        //If the node is already visited -> Continue (To avoid cycle)
        if(visited[p.v] == true) continue;

        //If unvisited, mark visited and print
        visited[p.v] = true;
        //Avoiding source node
        if(p.av != -1) cout << p.v << "  " << p.av << "  " << p.wt << endl;

        //Explore all unvisited adj nodes of current node
        for(pair<int, int> &neighbour : adj[p.v]){
            if(visited[neighbour.first] == false){
                pq.push(Pair(neighbour.second, neighbour.first, p.v));
            }
        }
    }
}
```

T: O((v+e)logv)

S: O(v+e)

T is V+E log v because Normal BFS has T = O(V+E) and here we are using PQ => Extra log v for each push and pop operation

Prims Algo

```cpp
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);  //Rank of all elements is initially zero
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;  //Initially, element is parent of itself => Size of each component is 1
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node]) return node;

        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u); //Ultiate parent of u
        int ulp_v = findUPar(v); //Ultiate parent of y

        if (ulp_u == ulp_v) return;
        //Whoever has a smaller rank is connected to bigger rank => Bigger rank is the parent after union
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        //If rank is same, make anyone as parent, but here we increase the rank of the parent by 1
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }
```

findUPar: O(1)

unionByRank: O(1)

```cpp
class Solution{
public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int kruskalsAlgo(int V, vector<vector<int>> adj[]){
        //Create a list of edges () vector<weight, u, v> ) from the given graph
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int adjNode = it[0];
                int wt = it[1];
                int node = i;

                edges.push_back({wt, {node, adjNode}});
            }
        }
        DisjointSet ds(V);

        //Sorting edges according to weight
        sort(edges.begin(), edges.end());
        int mstWt = 0;
        cout << "U V WT" << endl;
        for (auto it : edges) {
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            //If there is no cycle -> consider the edge
            if (ds.findUPar(u) != ds.findUPar(v)) { //If the two nodes belong to different components => No cycle
                mstWt += wt;
                ds.unionBySize(u, v);
                cout << u << " " << v << " " << wt << endl;
            }
        }

        return mstWt;
    }
};
```

T=O(E logv)

S=O(E +V)

Since for parent and rank array O(v) space is used. O(E) space is used for storing edges.

Kruskals Algo

Used to find the minnimum spanning tree

Spanning tree -> Suub graph of a graph having V vertices (same os original graph) and V-1 edges

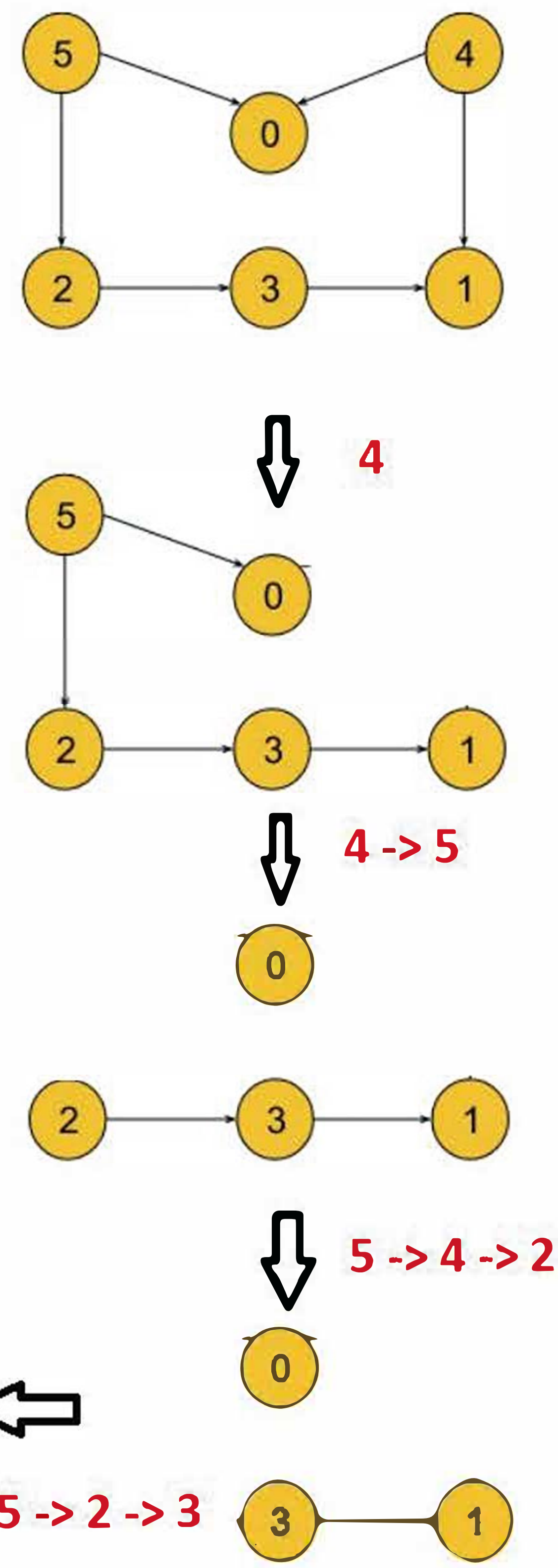
No of spanning trees = ${}^{E}C_{V-1}$ - no of cycles

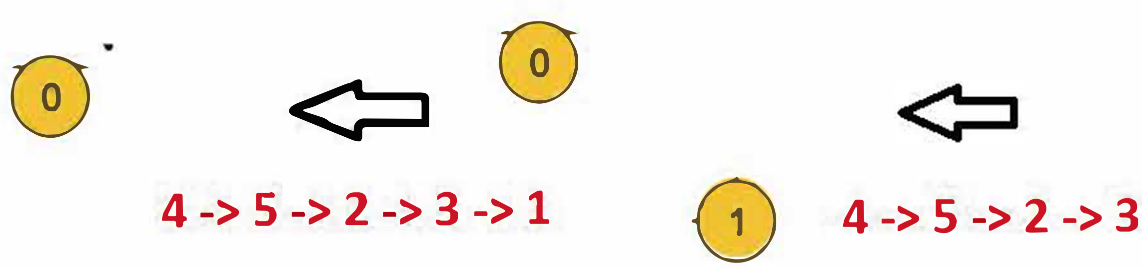
Khans Alfgorithm - Topological sorting using BFS

```cpp
vector<int> khansAlgo(vector<vector<int>> graph){
    //Calculating indegree of each vertes
    vector<int> indeg(graph.size());
    for(int i = 0; i < graph.size(); i++){
        for(int x : graph[i]){
            indeg[x]++;
        }
    }

    queue<int> q;
    //Pushing all nodes with indegree 0 to queue
    for(int i = 0; i < graph.size(); i++){
        if(indeg[i] == 0) q.push(i);
    }

    int idx = 0;
    vector<int> ans(graph.size());
    while(!q.empty()){
        int node = q.front();
        q.pop();

        ans[idx++] = node;
        for(int adj : graph[node]){
            indeg[adj]--;

            if(indeg[adj] == 0) q.push(adj);
        }
    }
    return idx == graph.size() ? ans : vector<int> {-1};
}
```

T ; O(V+E)

S : O(V)

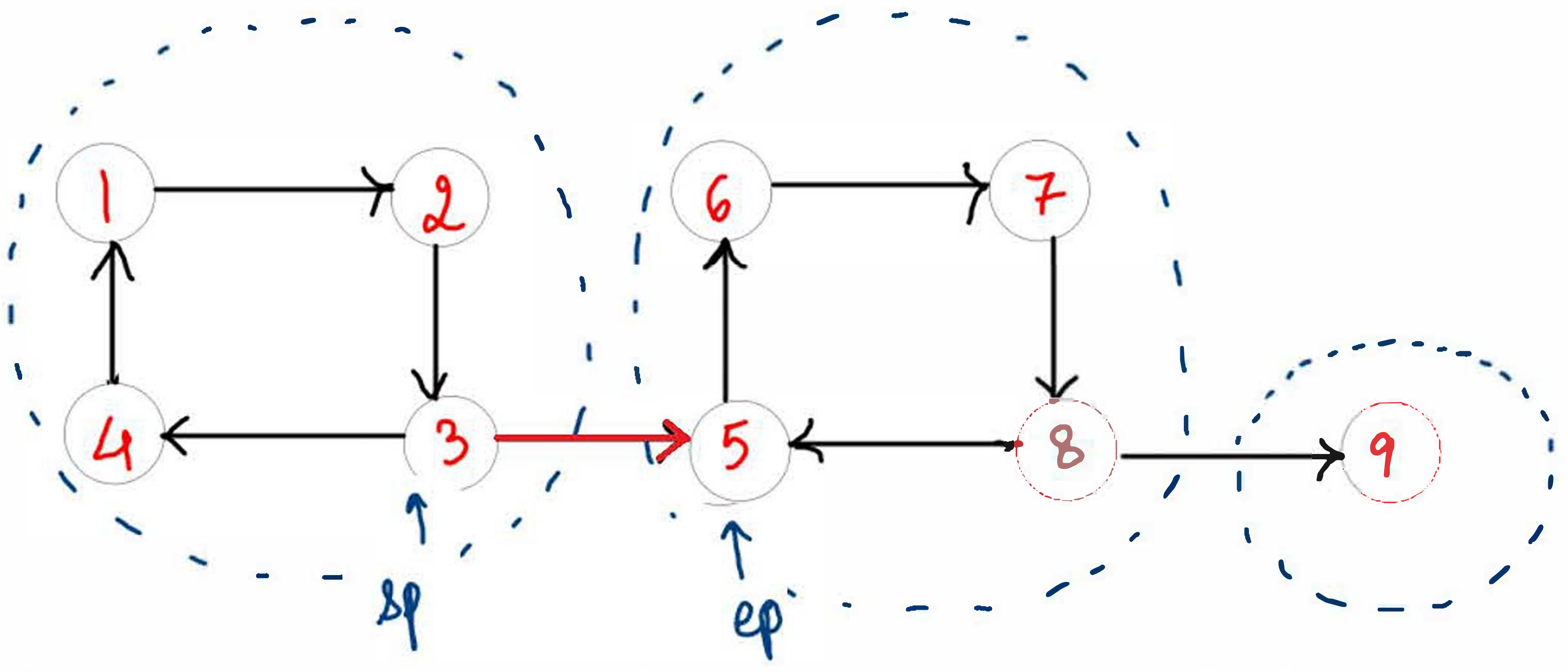




4 -> 5 -> 2 -> 3 -> 1 -> 0

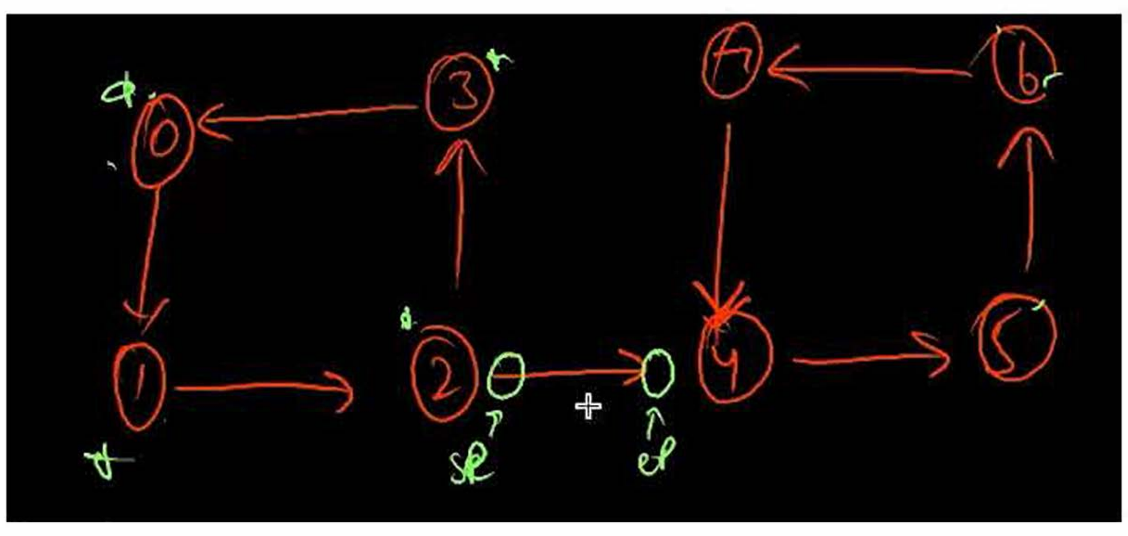
Kosaraju Algorithm

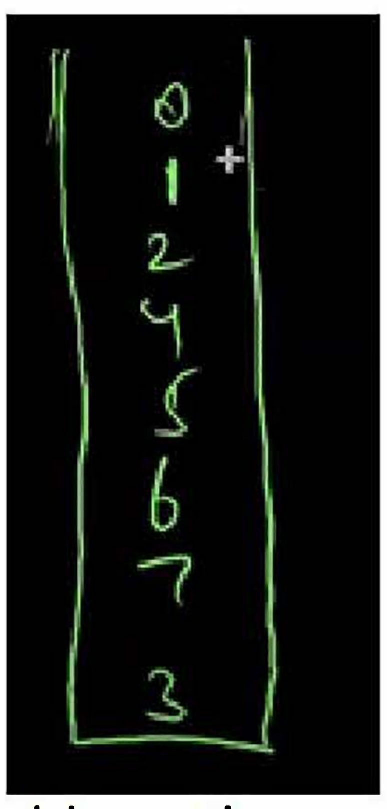
You will be given a graph with nodes and M directed edges. You have to find the number of Strongly Connected Components in the graph. First, understand what exactly is meant by Strongly Connected Components. A strongly connected component is a portion of the graph in which there is a path from each vertex to another vertex. It is only applicable on a directed graph. Also, all the strongly connected components of a graph are cyclic. But it does not mean that each cycle in a graph will be a strongly connected component. It is possible that multiple cycles can form one strongly connected component. Let's take a look at the graph given below:



If we apply normal DFS to this graph, it will give us the answer for the number of connected components as 1. This is because there is an edge between 3 and 5. But, since we can't go back from 5 to 3, it contradicts the statement for strongly connected components, and so, it should not be a part of the connected component, and form a separate strongly connected component. This is why we need a new algorithm to figure out the number of strongly connected components in a graph. So, through this, we figure out that the **edge between 3 and 5 is a problematic edge**

Our focus is now on the problamatic edge so let us consider the graph

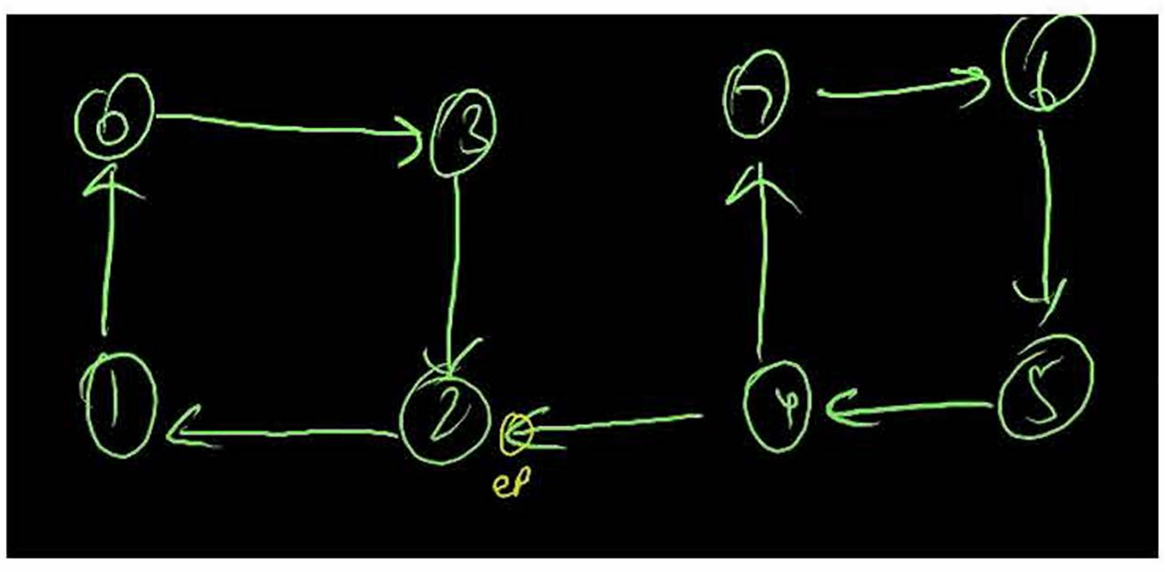




The problamatic edge is from 2 -> 4 here,   2 -> starting point , 4 -> End point of the problematic edge
The solution to the above problem is that we have to run DFS on the component at endpoint of the problematic edge.

But, the problem is, how will we figure out the problematic edge?
Therefore, Kosaraju algorithm was introduced. In this, we have used a simple trick.

1. First we impliment a random order dfs and add the nodes into a stack while backtracking.
   The element on the top of the stack will always be one of the elements from the side of the starting point of the problematic edge
2. Now, our next step will be to reverse all the edges of the graph



This will not cause any change in the strongly connected components as they will still be cyclic and remain connected to each other. The only major difference will be seen in the problematic edge. As now, the starting and ending points will be reversed.

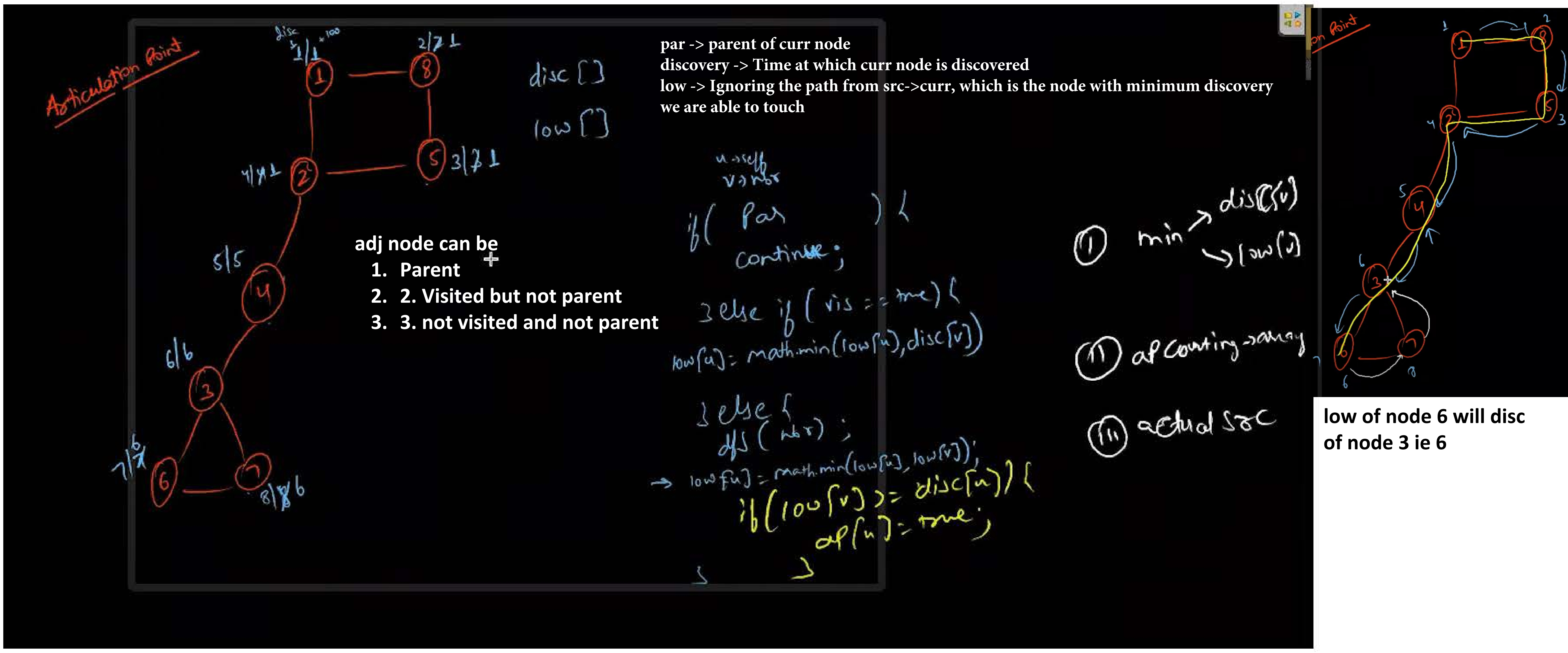
How will this help us ?
As we have seen earlier, by running random DFS we found an element from the side of the strting point of the problamatic edge. Now after reversing the edges, the element on the side of starting point has become an element on the ending point of problamatic edge which is what we needed

3. Now, apply DFS to the reversed graph in the order arranged in the stack

```
void dfs(int node, vector<vector<int>> &graph, vector<bool> &visited, stack<int> &s){
    visited[node] = true;
    for(int adj : graph[node]){
        if(!visited[adj]) dfs(adj, graph, visited, s);
    }
    s.push(node);
}

void revDfs(int node, vector<vector<int>> &graph, vector<bool> &visited){
    visited[node] = true;
    for(int adj : graph[node]){
        if(!visited[adj]) revDfs(adj, graph, visited);
    }
}
```

```
int kosarajusAlgp(vector<vector<int>> graph, int v){
    stack<int> s;
    vector<bool> visited(v, false);
    //1. Random DFS
    for(int i = 0; i < v; i++){
        if(!visited[i]){
            dfs(i, graph, visited, s);
        }
    }
    //2. Reversing all edges
    vector<vector<int>> revGraph(v);
    for(int i = 0; i < v; i++){
        visited[i] = false;
        for(int adj : graph[i]){
            revGraph[adj].push_back(i);
        }
    }
    // 3. DFS for reversed graph in the order of stack which we filled earlier
    int ans = 0;
    while(!s.empty()){
        int node = s.top();
        s.pop();

        if(!visited[node]){
            revDfs(node, revGraph, visited);
            ans++;
        }
    }
    return ans;
}
```

Attriculation Point



low of node 6 will disc of node 3 ie 6

A vertex is said to be an articulation point in a graph if removal of the vertex and associated edges disconnects the graph. So, the removal of articulation points increases the number of connected components in a graph.

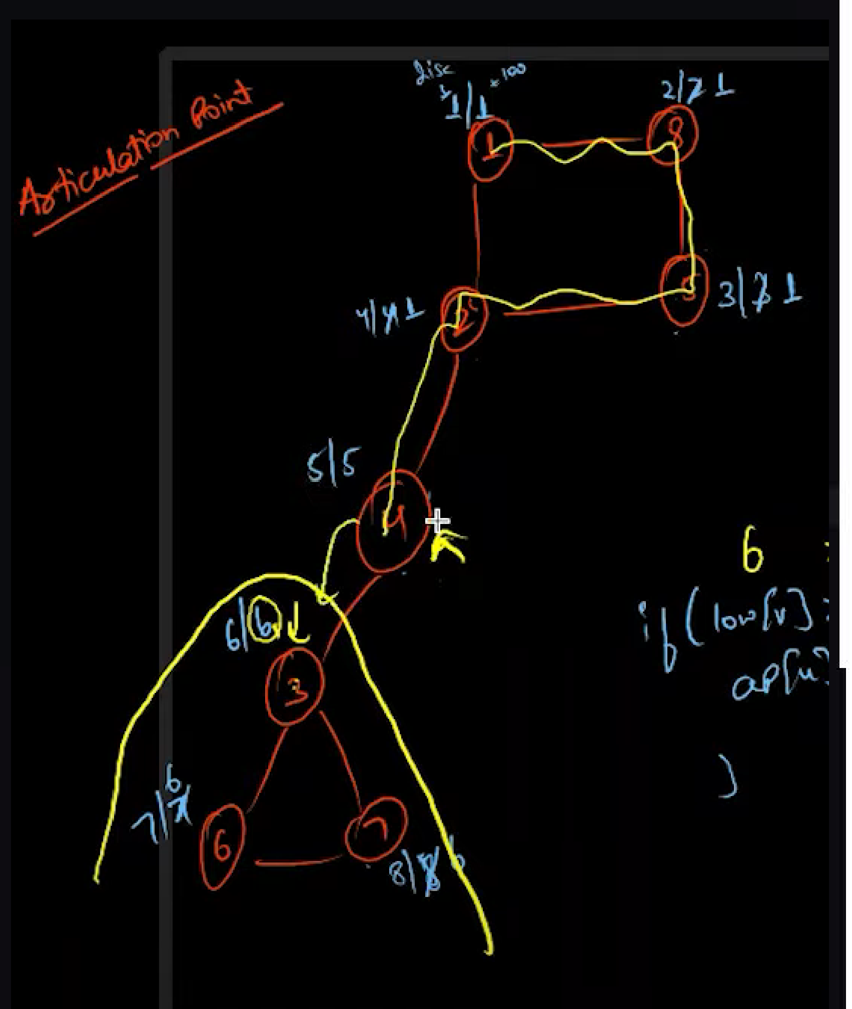
For an edge u->v
if low(v) >= disc(u)
then U is attriculation point

What does the above condition mean -> If condition is true if low (child) is less thann discovery (parent) this means that the whole graph below it is not able to reach parent or any of the nodes above parent

THIS CONDITION WILL NOT WORK FOR ROOT IE ACTUAL SOURCE

FOR ROOT OR SOURCE NODE -> IF THERE ARE MORE THAN ONE DFS CALLS FOR SRC NODE THEN SRC NODE IS ALSO AN ATTRICULATION POINT

```
class Pair{
public:
    int par;
    int disc;
    int low;
    bool isAP;
    Pair(){
        par = 0;
        disc = 0;
        low = 0;
        isAP = false;
    }
};

void dfs(vector<vector<int>> &graph, int u, vector<bool> &visited, vector<Pair> &ap, int disc){
    ap[u].disc = ap[u].low = disc;
    int cnt = 0; //Thsi variable is used to keep the count of no of dfs calls we make. This is useful

    visited[u] = true;
    for(int v : graph[u]){
        //If nbr is parent => nbr already is present in the path from src to current node
        if(v == ap[u].par) continue;
        //If nbr is not parent but is visited then the edge curr->nbe is not along the path from src to curr node  => low can be updated
        else if(visited[v]){
            ap[u].low = min(ap[u].low, ap[v].disc);
        }
        //else it is an unvisited nod => DFS
        else{
            ap[v].par = u;
            dfs(graph, v, visited, ap, disc+1);
            cnt++;

            ap[u].low = min(ap[u].low, ap[v].low);
            //Parent[u] = -1 => it is actual source.
            if(ap[u].par == -1 && cnt >= 2){
                ap[u].isAP = true;
            }
            else if(ap[v].low >= ap[u].disc){
                ap[u].isAP = true;
            }
        }
    }
}
void articulationPoint(vector<vector<int>> &graph, int v){
    vector<bool> visited(v, false);
    vector<Pair> ap(v);
    ap[0].par = -1;

    dfs(graph, 0, visited, ap, 1);
    for(int i = 0; i < v; i++){
        if(ap[i].isAP) cout << i+1 << " ";
    }

}
```



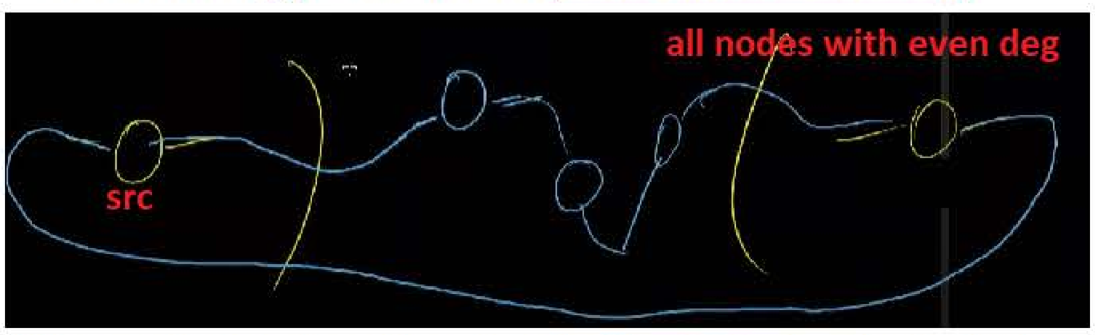
for edge 4->3
disc(4) = 5
low(3) = 6
=> low(child) >= disc(parent)
Iska matlab ye hain ki disc=5 ie node 4 ko ye pata chalgaya ki mere badh ek aisa banda hain jiski reach mere se pehle valo tak nahi he
=> node 4 is an attriculation point

EULRIAN PATH

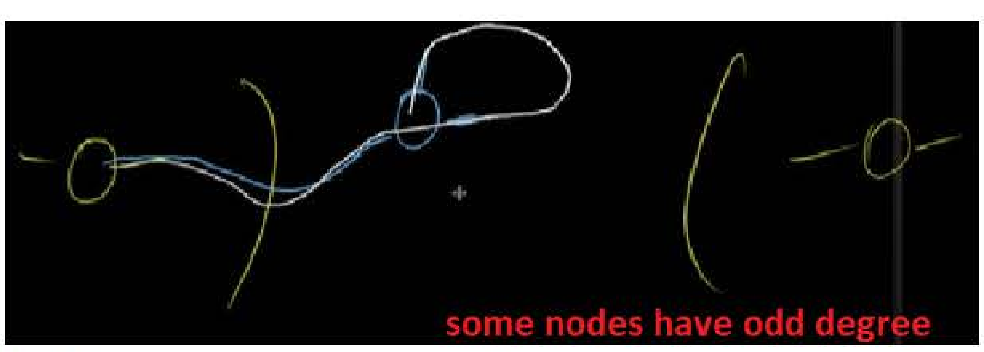
An Eulerian path is a path in a graph that visits every edge exactly once, allowing for the possibility of revisiting vertices. In other words, it is a sequence of edges that traverses the entire graph without lifting the pen from the paper (or without removing the finger from the touchscreen).
It is same as hamiltonian path but here it is edges instead of vertices

1. For Undirectred Graph

An undirected grap with all nodes having **even degree** has a eularian circuit



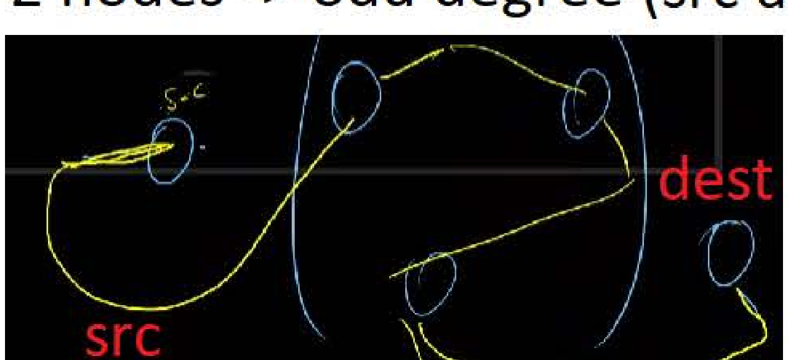
We can visualise the graph like this. Each node has an even degree => Each node has an entry point and exit point of an edge => We can easily make a eularian circuit



Node with even degree has an entry point but no exit point => we will be struck inside

For eularian path

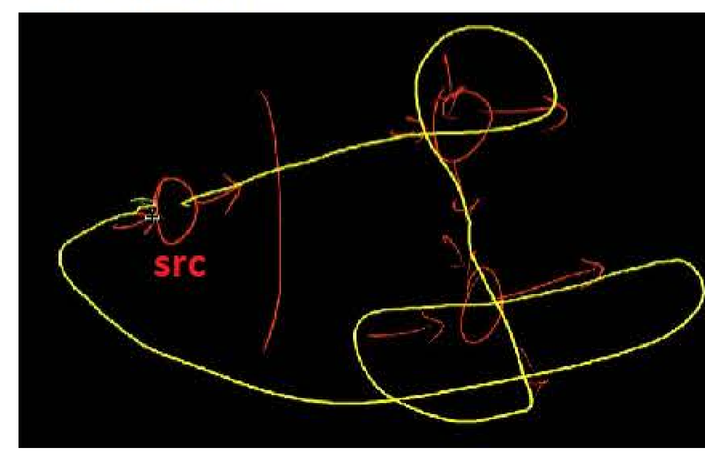
(n-2) nodes ->should have even degree
2 nodes -> odd degree (src and dest)



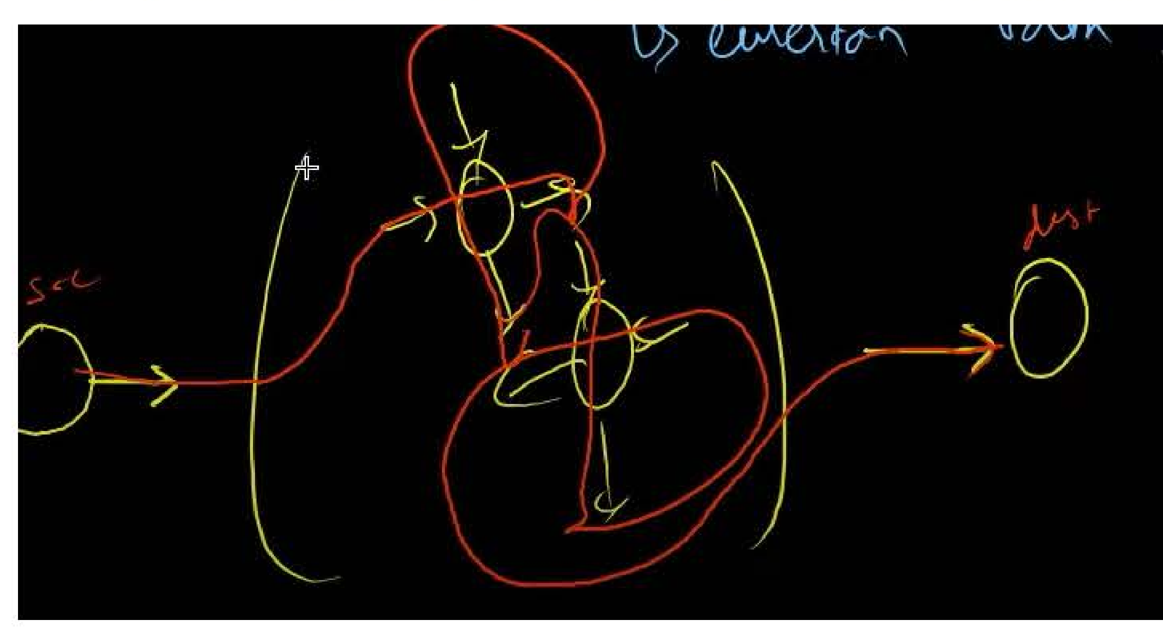
We want the inner nodes to have an even degree which will force it to have entry and exit point
Src-> odd deg => 1 exit point
dest -> 1 entry point

2. For directed graph

Eularian circuit-> Indegree=Outdegree for all nodes



Eularian path
(n-2) nodes -> indegree = outdegree
2 nodes -> special condition
src -> Outdeg should be 1 more than its indeg
dest-> Indeg should be 1 more than its outdeg



Implimentation -> Just run DFS until all the edges are visited
Node -> Addition of a node into path should be in post order

Eularian path -> AN edge can be visited exactly once and a vertex can be visited any no of times
Hamiltonian path-> An edge can be visited any no of times but a vertex can be visited exactly once

1. A common approach for variation of get connected components

```cpp
bool isvalid(int i,int j,int m,int n){
    return !(i == m || j == n || j < 0 || i < 0);
}

vector<pair<int, int>> dir = {{1,0},{0,1},{0,-1},{-1,0}}; //Direction
vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
    queue<pair<int,int>> q;
    int m = matrix.size();
    int n = matrix[0].size();
    //Creating a dist matrix and marking all 1's as unexplored
    vector<vector<int>> dis(m,vector<int>(n,-1));
    //Pushing all 0's to queue so that we can explore nearest 1 for each of them
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            if(matrix[i][j] == 0){
                q.push({i,j});
                dis[i][j] = 0;
            }
        }
    }
    while(!q.empty()){
        pair<int,int> curr = q.front();
        q.pop();

        int r = curr.first;
        int c = curr.second;
        for(auto &x : dir){
            int nr = r + x.first;  //next row
            int nc = c + x.second; //next col
            //If nr and nc are valid indices and unvisited
            if(isvalid(nr, nc, m, n) && dis[nr][nc] == -1){
                q.push({nr, nc});
                dis[nr][nc] = dis[r][c] + 1;
            }
        }
    }
    return dis;
}
```

nearest 0 to all 1's

1. Declare a direction array
2. Push all elements of a component onto queue
3. BFS :
   For all 4 directrions
   3.1. Calc nr and nc
   3.2. If nr and nc are valid
        do required WORK

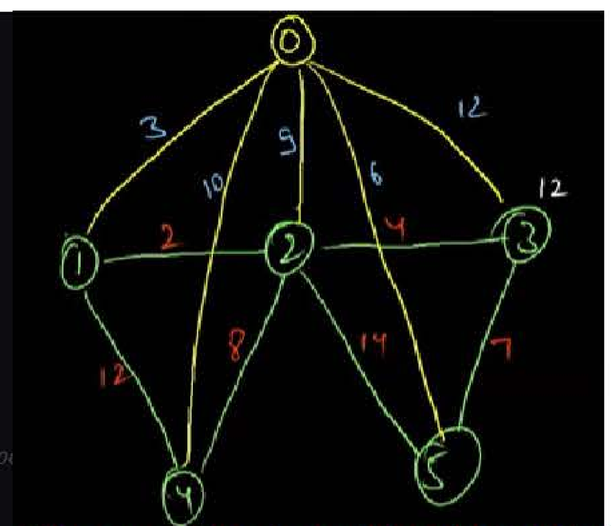
2. Application of topological sort

Pepcoding offers total of n courses labelled from 0 to n-1.
Some courses may have prerequisites. you have been given m pairs ai,bi. where 1 pair means you must take the course bi before the course ai.
Given the total number of courses numCourses and a list of the prerequisite pairs, return the ordering of courses you should take to finish all courses. If it is impossible to finish all courses print -1.

Other example -> Alien Dictionary

3. Application of MST

```cpp
int kruskalsAlgo(int V, vector<pair<int, pair<int, int>>> edges){
    DisjointSet ds(V);

    //Sorting edges according to weight
    sort(edges.begin(), edges.end());
    int mstWt = 0;
    int pipeCnt = 0;
    for (auto edge : edges) {
        int wt = edge.first;
        int u = edge.second.first;
        int v = edge.second.second;

        //If there is no cycle -> consider the edge
        if (ds.findUPar(u) != ds.findUPar(v)) { //If the two nodes belong to different compo
            mstWt += wt;
            ds.unionBySize(u, v);
        }
    }

    return mstWt;
}

int minCostToSupplyWater(int n, vector<int>&wells, vector<vector<int>>& pipes){
    //vector<wt, u, v>
    vector<pair<int, pair<int, int>>> edges;
    //We introduce a dummy node 0 which acts like well and digging well in ith polt = edge between 0 and i
    for(int i = 1; i <= n; i++){
        edges.push_back({wells[i-1], {0, i}});
    }
    for(vector<int> &edge : pipes){
        edges.push_back({ edge[2], {edge[0], edge[1]}});
    }

    return kruskalsAlgo(n+1, edges);
}
```



The quastion is nothing but MST where site acts as a vertex and pipe acts as an edge.
Since there must be atleast1 well, we create and extra dummy node 0 which acts like well and digging well in ith polt = edge between 0 and i

optimum WaterDistribution

There are n construction sites in a town. We want to supply water for all the construction sites by building wells and laying pipes.
For each site i, we can either build a well inside edge directly with cost wells[i-1], or pipe in water from another well to edge. The costs to lay pipes between sites are given by the 2d array cost, where each row of cost contains 3 numbers ai,bi and wi where wi is the cost to connect ai to bi. connections are bidirectional.
Return the minimum total cost to supply water to all the construction sites.

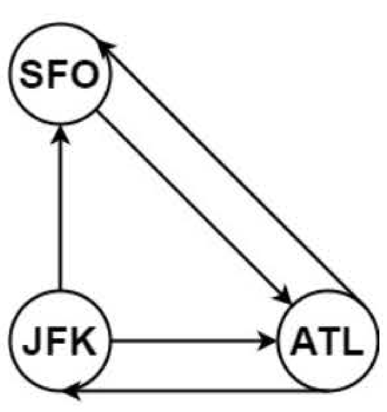
7. Eularian Path

You are given a list of airline `tickets` where `tickets[i] = [from_i, to_i]` represent the departure and the arrival airports of one flight. Reconstruct the itinerary in order and return it.

All of the tickets belong to a man who departs from `"JFK"`, thus, the itinerary must begin with `"JFK"`. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string.

- For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.

You may assume all tickets form at least one valid itinerary. You must use all the tickets once and only once.




```
Input: tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],
["ATL","SFO"]]
Output: ["JFK","ATL","JFK","SFO","ATL","SFO"]
Explanation: Another possible reconstruction is
["JFK","SFO","ATL","JFK","ATL","SFO"] but it is larger in lexical order.
```

Here src->dest is the edge and the cities are the nodes
We can use the ticket (edge) exactly once and visit all the cities(edges).
Since there is a constraint on edge => Vertices can be visited more than once => EULARIAN PATH

```cpp
//map <string, priorityQueue<string>>  -> PQ because we are asked to choose the cities in lexicoraphical order
unordered_map<string, multiset<string>> graph;
vector<string> ans;
void dfs(string vtex){
    multiset<string> &edges = graph[vtex];
    while (!edges.empty()){
        string to = *edges.begin();
        edges.erase(edges.begin()); //Marking the edge as visited. By doing so, we need not check if this edge is visited or not
        in further interation coz the edge doesnt exist now
        dfs(to);
    }
    ans.push_back(vtex);
}

vector<string> findItinerary(vector<vector<string>> tickets) {
    for (vector<string> &e : tickets) graph[e[0]].insert(e[1]);
    dfs("JFK");
    reverse(ans.begin(), ans.end());
    return ans;
}
```

4. Application of Dijkstra

```cpp
bool isValid(int i, int j, int n){
    return !(i < 0 || j < 0 || i == n || j == n);
}
int swimInWater(vector<vector<int>>& grid) {
    int n = grid.size();
    priority_queue<myPair, vector<myPair>, compare> pq;
    vector<vector<bool>> visited (n, vector<bool>(n, false));
    vector<pair<int, int>> dir = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    pq.push(myPair(0, 0, grid[0][0]));
    visited[0][0] = true;
    while(!pq.empty()){
        myPair node = pq.top();
        pq.pop();

        if(node.i == n-1 && node.j == n-1){
            return node.maxt;
        }
        for(auto p : dir){
            int ni = node.i + p.first;
            int nj = node.j + p.second;

            if(!isValid(ni, nj, n) || visited[ni][nj]) continue;

            pq.push(myPair(ni, nj, max(node.maxt, grid[ni][nj])));
            visited[ni][nj] = true;
        }
    }
    return -1;
}
```

This is an application of dijkstra algorithm, because at each step we have to choose a minimum between the adjascent nodes

0	1	2	3	4
24	23	22	21	5
12	13	14	15	16
11	17	18	19	20
10	9	8	7	6

op -> 16

You are given an `n x n` integer matrix `grid` where each value `grid[i][j]` represents the elevation at that point `(i, j)`.

The rain starts to fall. At time `t`, the depth of the water everywhere is `t`. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most `t`. You can swim infinite distances in zero time. Of course, you must stay within the boundaries of the grid during your swim.

Return *the least time until you can reach the bottom right square* `(n - 1, n - 1)` *if you start at the top left square* `(0, 0)`.

5. Articulation Point

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

A *critical connection* is a connection that, if removed, will make some servers unable to reach some other server.

Return all critical connections in the network in any order.

```cpp
void dfs(vector<vector<int>> &graph, int u, vector<bool> &visited, vector<Pair> &ap, int disc, vector<vector<int>> &ans){
    visited[u] = true;
    ap[u].disc = disc;
    ap[u].low = disc;
    int cnt = 0;

    for(int v : graph[u]){
        //If nbr is parent => nbr already is present in the path from src to current node
        if(v == ap[u].par) continue;
        //If nbr is not parent but is visited then the edge curr->nbe is not along the path from src to curr node  => low can
        be updated
        else if(visited[v]){
            ap[u].low = min(ap[u].low, ap[v].disc);
        }
        //else it is an unvisited nod => DFS
        else{
            ap[v].par = u;
            dfs(graph, v, visited, ap, disc+1, ans);

            ap[u].low = min(ap[u].low, ap[v].low);
            if(ap[u].par == -1){
                cnt++;
                if(cnt >= 2){
                    ans.push_back({u, v});
                }
            }
            else if(ap[v].low >= ap[u].disc){
                ans.push_back({u, v});
            }
        }
    }
}
```

This question is direct application of articulaion point but with a slight modification. Here ther have asked for edge instead of node.
So when we find an articulation point, we add an edge between u and v

Critical Connection

6. DSU

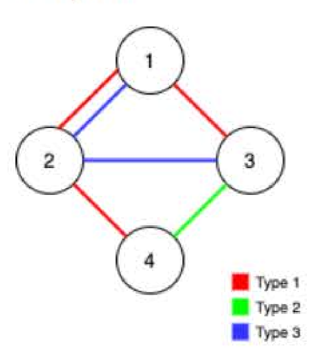
Alice and Bob have an undirected graph of `n` nodes and three types of edges:

- Type 1: Can be traversed by Alice only.
- Type 2: Can be traversed by Bob only.
- Type 3: Can be traversed by both Alice and Bob.

Given an array `edges` where `edges[i] = [type_i, u_i, v_i]` represents a bidirectional edge of type `type_i` between nodes `u_i` and `v_i`, find the maximum number of edges you can remove so that after removing the edges, the graph can still be fully traversed by both Alice and Bob. The graph is fully traversed by Alice and Bob if starting from any node, they can reach all other nodes.

Return *the maximum number of edges you can remove, or return* `-1` *if Alice and Bob cannot fully traverse the graph.*

Example 1:




```
Input: n = 4, edges = [[3,1,2],[3,2,3],[1,1,3],[1,2,4],[1,1,2],[2,3,4]]
Output: 2
Explanation: If we remove the 2 edges [1,1,2] and [1,1,3]. The graph will still
be fully traversable by Alice and Bob. Removing any additional edge will not
make it so. So the maximum number of edges we can remove is 2.
```

Approach is very simple, we keep two Disjoint sets for alice and bob, out main goal is to keep both alice and bob's graph connected.
1. Process the type 3 edges first- union in both alice and bob dsu
2. Run bfs algorithm on all the edges -> if the nodes of an edge are already connected ie thay both have the same ultimate parent then there is no need of that edge -> decriment no of edges

TO MAKE DSU OF A 2D MATRIX, WE MAKE USE OF CELL NO
CNO = R * N + C

HATKE questions

01 BFS

```
void addEdge(vector<vector<pair<int, int>>> &graph, int u, int v){
    graph[u].push_back({v, 0});
    graph[v].push_back({u, 1}); //Reverse edge with cost 1
}

// Approach-> Just add a reverse edge with cost 1 and apply path from
// src to dest using BFS and add cost at each step
int solution(vector<vector<pair<int, int>>> &graph, int src, int dest){
    list<pair<int, int>> q;
    q.push_back({src, 0});

    vector<bool> visited(graph.size(), false);
    while(q.size()){
        pair<int, int> node = q.front();
        q.pop_front();

        if(node.first == dest) return node.second;

        visited[node.first] = true;
        for(pair<int, int> &adj : graph[node.first]){
            if(visited[adj.first]) continue;

            if(adj.second == 0) q.push_front({adj.first, node.second});
            else q.push_back({adj.first, node.second + 1});
        }
    }
    return -1;
}
```

You are given a directed graph , src and destinationn
You have to find the minimum number of edges you have to reverse in order to have atleast one path from vertex 1 to N, where the vertices are numbered from src to dest

Bus routes

```
int numBusesToDestination(vector<vector<int>>& routes, int source, int target) {
    if(source == target) return 0;
    //map <stop, busses>
    unordered_map <int, vector<int>> mp;
    for(int i = 0; i < routes.size(); i++){
        for(int j = 0; j < routes[i].size(); j++){
            mp[routes[i][j]].push_back(i);
        }
    }
    unordered_set<int> bv;  //To check if bus is already boarded once -> BUS VISITED
    unordered_set<int> bsv; //To check if the bus stop is already visited -> BUS STOP VISITED
    queue <int> q;
    q.push(source);
    int count = 0;
    while(!q.empty()){
        int size = q.size();
        while(size--){
            int stop = q.front();
            q.pop();
            for(int bus : mp[stop]){
                if(bv.find(bus) != bv.end()) continue; //If bus is already boarded => continue
                for(int busStop : routes[bus]){
                    if(bsv.find(busStop) != bsv.end()) continue; //If bus stop is already visited => Continue
                    if(busStop == target) return count+1; //Because current bus is not counted
                    bsv.insert(busStop);

                    q.push(busStop);

                }
                bv.insert(bus);
            }
        }
        count++;
    }
    return -1;
}
```

leetcode hard- Bus routes

You are given an array `routes` representing bus routes where `routes[i]` is a bus route that the i^{th} bus repeats forever.

- For example, if `routes[0] = [1, 5, 7]`, this means that the 0^{th} bus travels in the sequence `1 -> 5 -> 7 -> 1 -> 5 -> 7 -> 1 -> ...` forever.

You will start at the bus stop `source` (You are not on any bus initially), and you want to go to the bus stop `target`. You can travel between bus stops by buses only.

Return *the least number of buses you must take to travel from* `source` *to* `target`. Return `-1` if it is not possible.

Example 1:

```
Input: routes = [[1,2,7],[3,6,7]], source = 1, target = 6
Output: 2
Explanation: The best strategy is take the first bus to the bus stop 7, then
take the second bus to the bus stop 6.
```