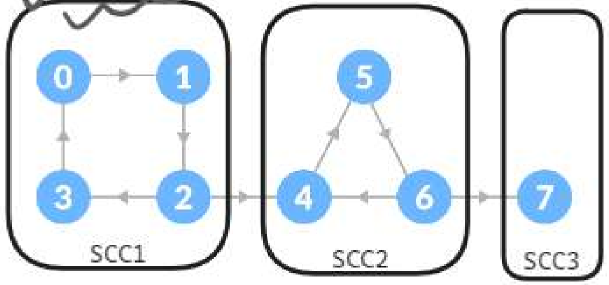
KOSARAJU ALGORITHM

SCC - STRONGLY CONNECTED COMPONENTS

GRAPH



linkedin@manojofficialmj

📂 4. KOSARAJU Algorithm

🚀 *Find strongly connected components*

What is and where to use Kosaraju Algorithm:

A strongly connected component is the component of a directed graph that has a path from every vertex to every other vertex in that component. It can only be used in a directed graph.

What is a strongly connected component:

😊 *Raju kisi bhi ek source node se kisi bhi ek destination node tak ja sakata hai usse hum strongly connected component kahate hai.*

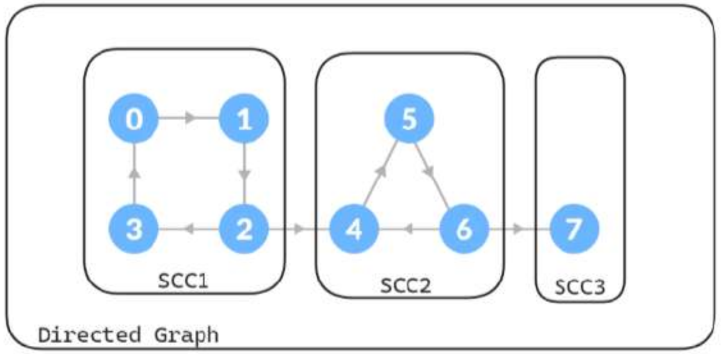
What is the flow of Kosaraju Algorithm:

✅***Step 1:*** *Find the order*
Yeh Order Topological Sort ki tarah hi hai but Topological Sort nahi hai kyunki T.S. only Acyclic Graph par work karta hai jabki ek strongly connected component hum tabhi find karte hai jab graph me cycle present hoti hai.

Why use of this order:
Iss order se hoga yeh ki Raju two components me move kar sakta hai
Point 1: *Raju current S.C.C. me move kar sakata hai yaa fir*
Point 2: *Raju visited S.C.C. me bhi move kar sakata hai*

But ***Point 2*** *ki wajah se hamara* ***extra time*** *kharab ho raha hai to iss time ko save karane ke liye hum ek* ***visited map*** *ka use kar lenge to track the visited node.*

✅***Step 2:*** *Reverse all edges to create the disconnected component and store in new adjList*
✅***Step 3:*** *Count the strongly connected components using DFS*



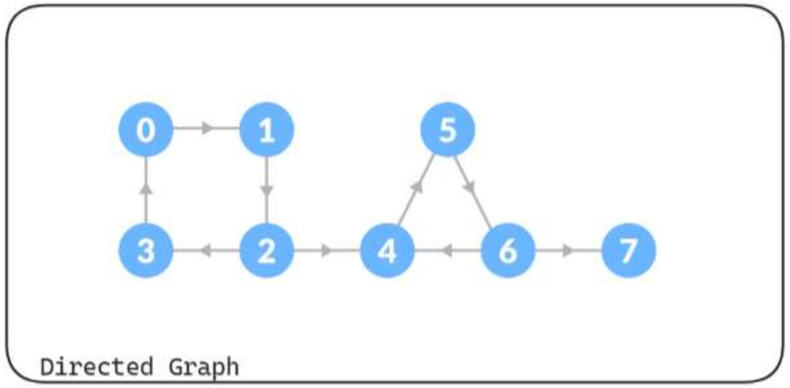
Cycle 1: SSC1
0->1->2->3
1->2->3->0
2->3->0->1
3->0->1->2

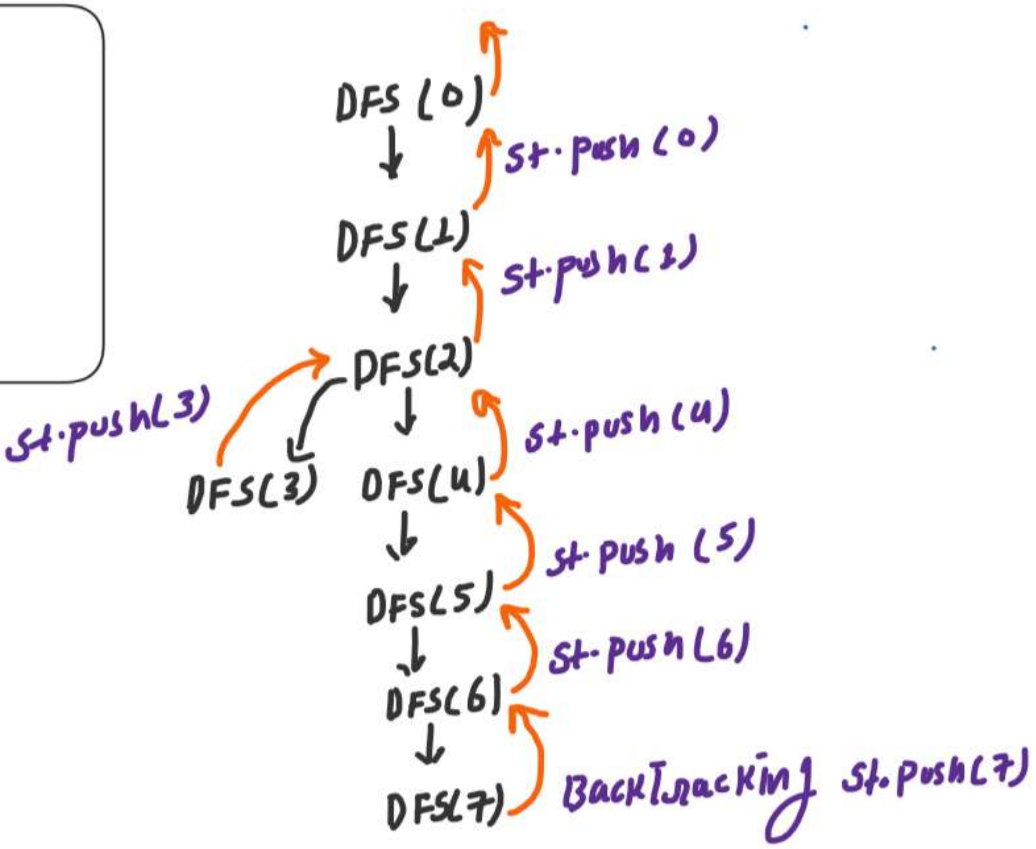
Cycle 2: SSC2
4->5->6
5->6->4
6->4->5

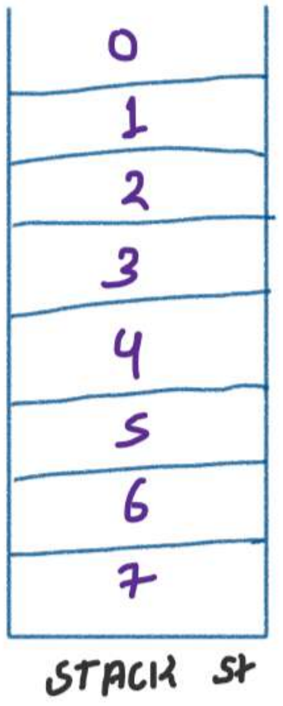
Cycle 3: SSC3
7

Total SCC: 3

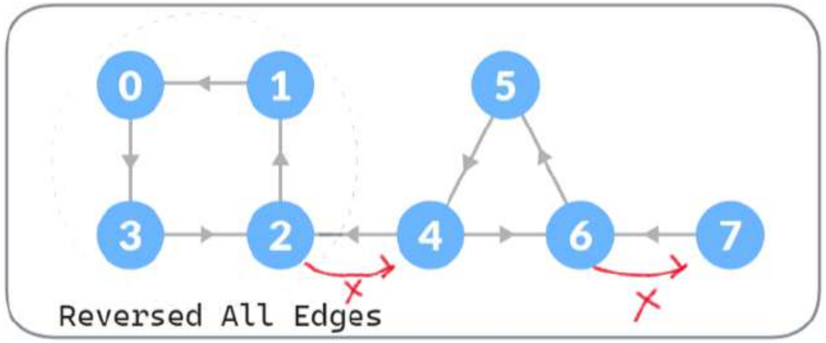
STEP 1 STORE ORDER IN THE STACK





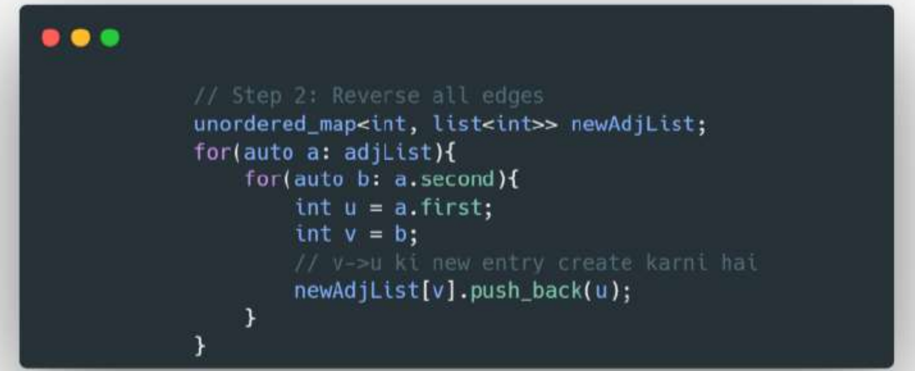


STEP2 Reverse All edges



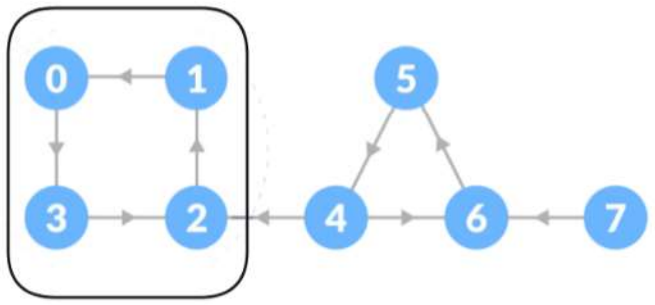
→ Now Raju is Not able to go from ② ~~→~~ TO ④ and from ⑥ ~~→~~ TO ⑦ it means

Humne GRAPH ko 3 component me Divide kar diya Hai



```
// Step 2: Reverse all edges
unordered_map<int, list<int>> newAdjList;
for(auto a: adjList){
    for(auto b: a.second){
        int u = a.first;
        int v = b;
        // v->u ki new entry create karni hai
        newAdjList[v].push_back(u);
    }
}
```

STEP3 Count S.C.C. from reversed edges graph



visited

key	val
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	F
5	F
6	F
7	F

$\boxed{\text{Count SCC} = 1}$

(A) Get TOP from stack and pop it $\boxed{\text{TOP} = 0}$

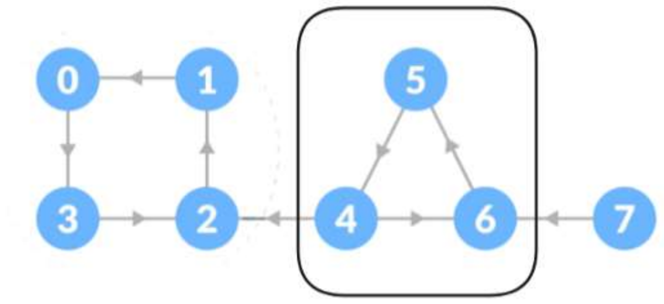
(B) DFS from TOP to All nbrs

$$\begin{tikzcd}
\text{DFS}(0) \arrow[r, ""] \arrow[d] & T \\
\text{DFS}(3) \arrow[r] \arrow[d] \arrow[u, bend left] & T \\
\text{DFS}(2) \arrow[r] \arrow[d] \arrow[u, bend left] & T \\
\text{DFS}(1) \arrow[r] \arrow[u, bend left] & T
\end{tikzcd}$$

~~0~~	← TOP
1	
2	
3	
4	
5	
6	
7	

STACK st

STEP3



Ⓐ Get TOP from stack and pop it $\boxed{\text{TOP} = 4}$

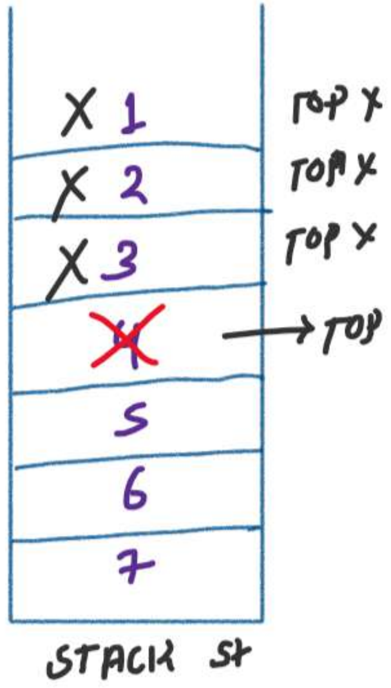
Ⓑ DFS from TOP to All Nbrs

$$\begin{tikzcd}
\text{DFS}(4) \xrightarrow{} T \arrow[d] \\
\text{DFS}(6) \xrightarrow{} T \arrow[d] \arrow[u, bend left] \\
\text{DFS}(5) \xrightarrow{} T \arrow[u, bend left]
\end{tikzcd}$$

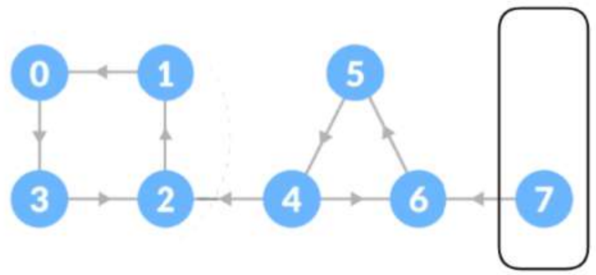
Visited

Key	val
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T
6	~~F~~ T
7	F

$\boxed{\text{Count SCC} = 2}$



STEP3



Visited

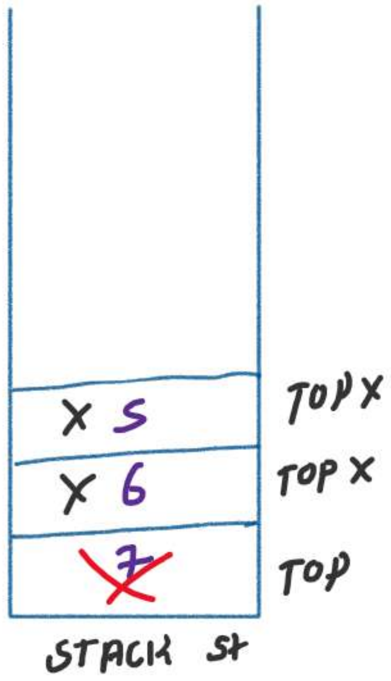
key	val
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T
5	~~F~~ T
6	~~F~~ T
7	~~F~~ T

$\boxed{\text{Count SCC} = 3}$

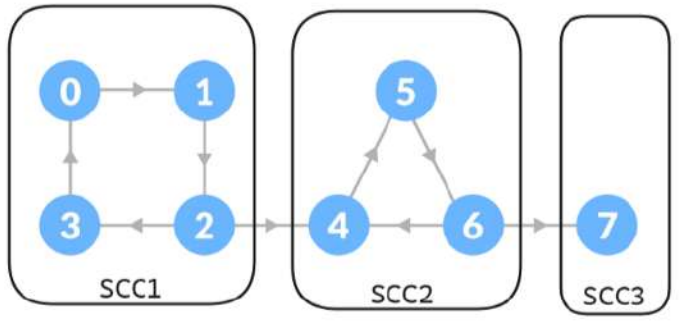
(A) Get TOP from stack and pop it $\boxed{\text{TOP} = 7}$

(B) DFS From TOP TO All Nbrs

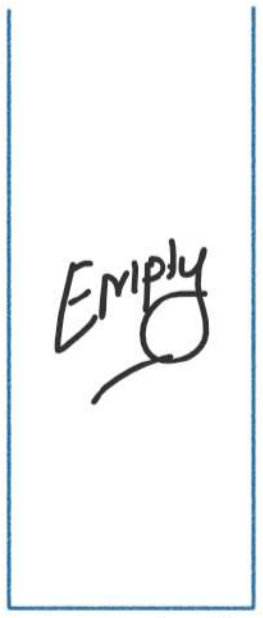
DFS(7) $\to$ T



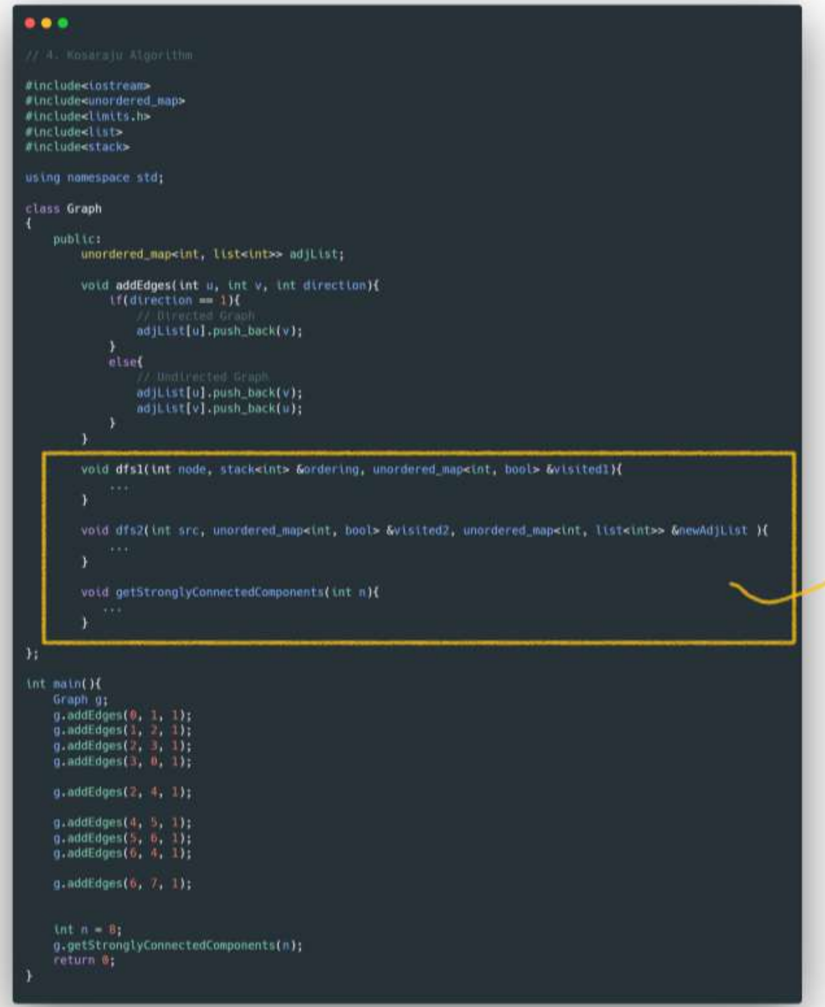
STEP3



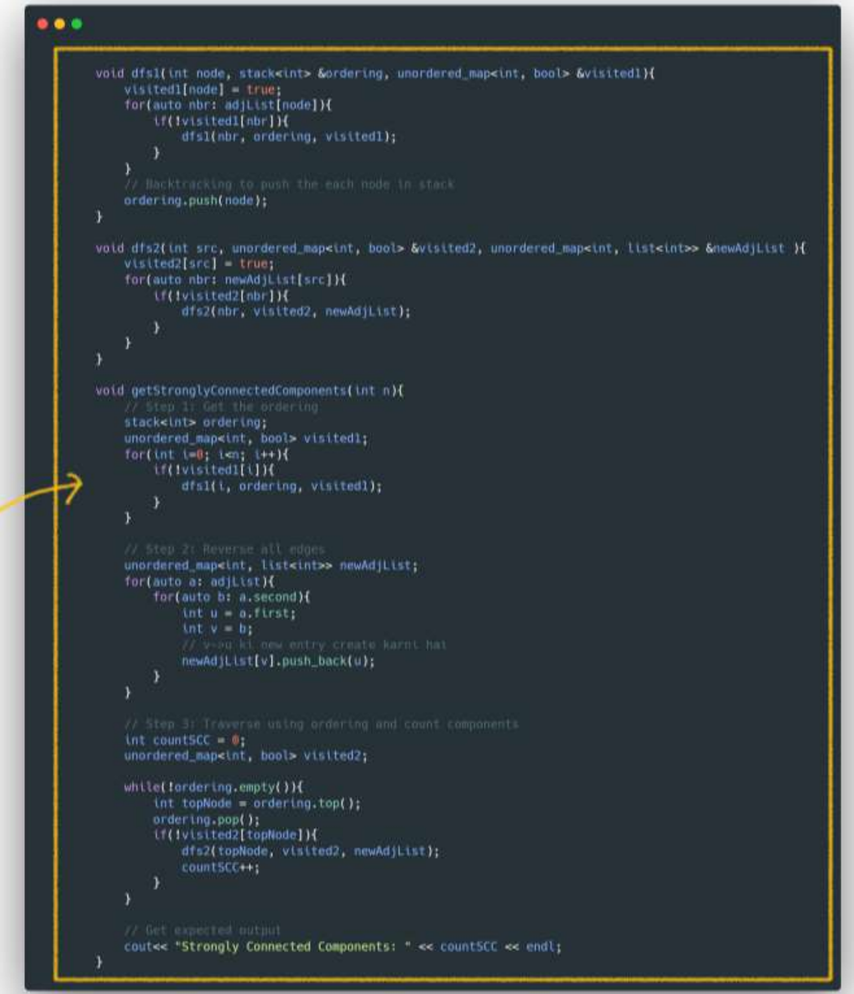
$\boxed{\text{Count SCC} = 3}$ This is our output



STACK St



```cpp
// 4. Kosaraju Algorithm

#include<iostream>
#include<unordered_map>
#include<limits.h>
#include<list>
#include<stack>

using namespace std;

class Graph
{
    public:
        unordered_map<int, list<int>> adjList;

        void addEdges(int u, int v, int direction){
            if(direction == 1){
                // Directed Graph
                adjList[u].push_back(v);
            }
            else{
                // Undirected Graph
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }

        void dfs1(int node, stack<int> &ordering, unordered_map<int, bool> &visited1){
            ...
        }

        void dfs2(int src, unordered_map<int, bool> &visited2, unordered_map<int, list<int>> &newAdjList ){
            ...
        }

        void getStronglyConnectedComponents(int n){
           ...
        }
};

int main(){
    Graph g;
    g.addEdges(0, 1, 1);
    g.addEdges(1, 2, 1);
    g.addEdges(2, 3, 1);
    g.addEdges(3, 0, 1);

    g.addEdges(2, 4, 1);

    g.addEdges(4, 5, 1);
    g.addEdges(5, 6, 1);
    g.addEdges(6, 4, 1);

    g.addEdges(6, 7, 1);


    int n = 8;
    g.getStronglyConnectedComponents(n);
    return 0;
}
```



```cpp
void dfs1(int node, stack<int> &ordering, unordered_map<int, bool> &visited1){
    visited1[node] = true;
    for(auto nbr: adjList[node]){
        if(!visited1[nbr]){
            dfs1(nbr, ordering, visited1);
        }
    }
    // Backtracking to push the each node in stack
    ordering.push(node);
}

void dfs2(int src, unordered_map<int, bool> &visited2, unordered_map<int, list<int>> &newAdjList ){
    visited2[src] = true;
    for(auto nbr: newAdjList[src]){
        if(!visited2[nbr]){
            dfs2(nbr, visited2, newAdjList);
        }
    }
}

void getStronglyConnectedComponents(int n){
    // Step 1: Get the ordering
    stack<int> ordering;
    unordered_map<int, bool> visited1;
    for(int i=0; i<n; i++){
        if(!visited1[i]){
            dfs1(i, ordering, visited1);
        }
    }

    // Step 2: Reverse all edges
    unordered_map<int, list<int>> newAdjList;
    for(auto a: adjList){
        for(auto b: a.second){
            int u = a.first;
            int v = b;
            // v->u ki new entry create karni hai
            newAdjList[v].push_back(u);
        }
    }

    // Step 3: Traverse using ordering and count components
    int countSCC = 0;
    unordered_map<int, bool> visited2;

    while(!ordering.empty()){
        int topNode = ordering.top();
        ordering.pop();
        if(!visited2[topNode]){
            dfs2(topNode, visited2, newAdjList);
            countSCC++;
        }
    }

    // Get expected output
    cout<< "Strongly Connected Components: " << countSCC << endl;
}
```

T.C. = ?

S.C. = ?