Strongly Connected Components - GeeksforGeeks

Source: https://www.geeksforgeeks.org/strongly-connected-components/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Strongly Connected Components Last Updated : 30 Jan, 2026 In a directed graph, a Strongly Connected Component is a subset of vertices where every vertex in the subset is reachable from every other vertex in the same subset by traversing the directed edges. Finding the SCCs of a graph can provide important insights into the structure and connectivity of the graph, with applications in various fields such as social network analysis, web crawling, and network routing . The below graph has two strongly connected components {1,2,3,4} and {5,6,7} since there is path from each vertex to every other vertex in the same strongly connected component. Strongly Connected Components Difference Between Connected and Strongly Connected Components (SCCs) : The term Connected is for undirected graphs and Strongly Connected is for directed graphs. A subgraph of a directed graph is considered to be an Strongly Connected Components(SCC) if and only if for every pair of vertices A and B, there exists a path from A to B and a path from B to A. Connecting Two Strongly Connected Component by a Unidirectional Edge : Two different connected components becomes a single component if a edge is added between a vertex from one component to a vertex of other component. But this is not the case in strongly connected components. Two strongly connected components d not become a single strongly connected component if there is only a unidirectional edge from one SCC to other SCC. Please refer the above graph for example. We have edges from 4 to 5 and 3 to 4, despite these edges, two components are different. Example: Input: adj[][] = [[], [3, 4], [1], [2], [5], []] Output: [[1, 2, 3], [4], [5]] Explanation: There are 3 different Strongly Connected Components. They are {1, 2, 3} and {4}, {5}. Try it on GfG Practice [Naive Approach] Brute Force Approach The main idea will be for each vertex i, find the vertices which will be the part of strongly connected component containing vertex i. Two vertex i and j will be in the same strongly connected component if they there is a directed path from vertex i to vertex j and vice-versa. Let's understand the approach with the help of following example: Brute Force Approach Starting with vertex 1. There is path from vertex 1 to vertex 2 and vice-versa. Similarly there is a path from vertex 1 to vertex 3 and vice versa. So, vertex 2 and 3 will be in the same Strongly Connected Component as vertex 1. Although there is directed path form vertex 1 to vertex 4 and vertex 5. But there is no directed path from vertex 4,5 to vertex 1 so vertex 4 and 5 won't be in the same Strongly Connected Component as vertex 1. Thus Vertex 1,2 and 3 forms a Strongly Connected Component. For Vertex 2 and 3, there Strongly Connected Component has already been determined. For Vertex 4, there is a path from vertex 4 to vertex 5 but there is no path from vertex 5 to vertex 4. So vertex 4 and 5 won't be in the Same Strongly Connected Component. So both Vertex 4 and Vertex 5 will be part of Single Strongly Connected Component. Hence there will be 3 Strongly Connected Component {1,2,3}, {4} and {5}. Below is the implementation of above approach: C++ #include <iostream> #include <vector> #include <algorithm> using namespace std ; // dfs Function to reach destination bool dfs (int curr , int des , vector < vector < int > >& adj , vector < int >& vis) { // If curr node is destination return true if (curr == des) { return true ; } vis [curr] = 1 ; for (auto x : adj [curr]) { if (! vis [x]) { if (dfs (x , des , adj , vis)) { return true ; } } } return false ; } // To tell whether there is path from source to // destination bool isPath (int src , int des , vector < vector < int > >& adj) { vector < int > vis (adj . size () + 1 , 0); return dfs (src , des , adj , vis); } // Function to return all the strongly connected // component of a graph. vector < vector < int > > findSCC (int n , vector < vector < int > >& a) { // Stores all the strongly connected components. vector < vector < int > > ans ; // Stores whether a vertex is a part of any Strongly // Connected Component vector < int > is_scc (n + 1 , 0); vector <

vector < int > > adj (n + 1); for (int i = 0 ; i < a . size (); i ++) { adj [a [i][0]]. push_back (a [i][1]); } // Traversing all the vertices for (int i = 1 ; i <= n ; i ++) { if (! is_scc [i]) { // If a vertex i is not a part of any SCC // insert it into a new SCC list and check // for other vertices whether they can be // thr part of thidl ist. vector < int > scc ; scc . push_back (i); for (int j = i + 1 ; j <= n ; j ++) { // If there is a path from vertex i to // vertex j and vice versa put vertex j // into the current SCC list. if (! is_scc [j] && isPath (i , j , adj) && isPath (j , i , adj)) { is_scc [j] = 1 ; scc . push_back (j); } } // Insert the SCC containing vertex i into // the final list. ans . push_back (scc); } } return ans ; } int main () { int V = 5 ; vector < vector < int > > edges { { 1 , 3 }, { 1 , 4 }, { 2 , 1 }, { 3 , 2 }, { 4 , 5 } }; vector < vector < int > > ans = findSCC (V , edges); cout << "Strongly Connected Components are: \n " ; for (int i = 0 ; i < ans . size (); i ++) { for (int j = 0 ; j < ans [i]. size (); j ++) { cout << ans [i][j] << " " ; } cout << " \n " ; } } Java class GfG { // dfs Function to reach destination boolean dfs (int curr , int des , int [][] adj , int [] vis , int n) { if (curr == des) return true ; vis [curr] = 1 ; for (int i = 1 ; i <= n ; i ++) { if (adj [curr][i] == 1 && vis [i] == 0) { if (dfs (i , des , adj , vis , n)) return true ; } } return false ; } // To tell whether there is path from source to destination boolean isPath (int src , int des , int [][] adj , int n) { int [] vis = new int [n + 1] ; return dfs (src , des , adj , vis , n); } // Function to return all the strongly connected component of a graph. int [][] findSCC (int n , int [][] a , int m) { // Stores whether a vertex is part of any SCC int [] is_scc = new int [n + 1] ; // Adjacency matrix representation of graph int [][] adj = new int [n + 1][n + 1] ; for (int i = 0 ; i < m ; i ++) { int u = a [i][0] ; int v = a [i][1] ; adj [u][v] = 1 ; } // Result array to store all SCCs (assume worst-case n SCCs of size n) int [][] result = new int [n][n + 1] ; // +1 to store count at index 0 int sccCount = 0 ; // Traversing all the vertices for (int i = 1 ; i <= n ; i ++) { if (is_scc [i] == 0) { int [] scc = new int [n + 1] ; int size = 0 ; scc [++ size] = i ; for (int j = i + 1 ; j <= n ; j ++) { if (is_scc [j] == 0 && isPath (i , j , adj , n) && isPath (j , i , adj , n)) { is_scc [j] = 1 ; scc [++ size] = j ; } } // Mark current node as visited in SCC is_scc [i] = 1 ; // Store size at index 0 and then elements scc [0] = size ; result [sccCount ++] = scc ; } } // Trim result to valid SCCs only int [][] finalResult = new int [sccCount][] ; for (int i = 0 ; i < sccCount ; i ++) { int size = result [i][0] ; finalResult [i] = new int [size] ; for (int j = 0 ; j < size ; j ++) { finalResult [i][j] = result [i][j + 1] ; } } return finalResult ; } } public class Main { public static void main (String [] args) { GfG obj = new GfG (); int V = 5 ; int [][] edges = { { 1 , 3 }, { 1 , 4 }, { 2 , 1 }, { 3 , 2 }, { 4 , 5 } }; int M = edges . length ; int [][] ans = obj . findSCC (V , edges , M); System . out . println ("Strongly Connected Components are:"); for (int [] scc : ans) { for (int x : scc) { System . out . print (x + " "); } System . out . println (); } } } Python class GfG : # dfs Function to reach destination def dfs (self , curr , des , adj , vis): # If current node is the destination, return True if curr == des : return True vis [curr] = 1 for x in adj [curr]: if not vis [x]: if self . dfs (x , des , adj , vis): return True return False # To tell whether there is a path from source to destination def isPath (self , src , des , adj): vis = [0] * (len (adj) + 1) return self . dfs (src , des , adj , vis) # Function to return all the strongly connected components of a graph. def findSCC (self , n , a): # Stores all the strongly connected components. ans = [] # Stores whether a vertex is a part of any Strongly Connected Component is_scc = [0] * (n + 1) adj = [[] for _ in range (n + 1)] for i in range (len (a)): adj [a [i][0]] . append (a [i][1]) # Traversing all the vertices for i in range (1 , n + 1): if not is_scc [i]: # If a vertex i is not a part of any SCC, insert it into a new SCC list # and check for other vertices whether they can be part of this list. scc = [i] for j in range (i + 1 , n + 1): # If there is a path from vertex i to vertex j and vice versa, # put vertex j into the current SCC list. if not is_scc [j] and self . isPath (i , j , adj) and self . isPath (j , i , adj): is_scc [j] = 1 scc . append (j) # Insert the SCC containing vertex i into the final list. ans . append (scc) return ans if __name__ == "__main__" : obj = GfG () V = 5 edges = [[1 , 3], [1 , 4], [2 , 1], [3 , 2], [4 , 5]] ans = obj . findSCC (V , edges) print ("Strongly Connected Components are:") for x in ans : for y in x : print (y , end = " ") print () C# using System ; class GfG { // DFS function to reach destination public bool dfs (int curr , int des , int [,] adj , int [] vis , int n) { if (curr == des) return true ; vis [curr] = 1 ; for (int i = 1 ; i <= n ; i ++) { if (adj [curr , i] == 1 && vis [i] == 0) { if (dfs (i , des , adj , vis , n)) return true ; } } return false ; } // To tell whether there is a path from source to destination public bool isPath (int src , int des , int [,] adj , int n) { int [] vis = new int [n + 1]; return dfs (src , des , adj , vis , n); } // Function to return all the strongly connected components of a graph public int [][] findSCC (int n , int [,] edges , int m) { int [,] adj = new int [n + 1 , n + 1]; for (int i = 0 ; i < m ; i ++) { int u = edges [i , 0]; int v = edges [i , 1]; adj [u , v] = 1 ; } int [] isScc = new int [n + 1]; int [][] result = new int [n][]; int sccIndex = 0 ; for (int i = 1 ; i <= n ; i ++) { if (isScc [i] == 0) { int [] temp = new int [n + 1]; // [0] stores size int size = 0 ; temp [++ size] = i ; for (int j = i + 1 ; j <= n ; j ++) { if (isScc [j] == 0 && isPath (i , j , adj , n) && isPath (j , i , adj , n)) { isScc [j] = 1 ; temp [++ size] = j ; } } isScc [i] = 1 ; temp [0] = size ; int [] scc = new int [size]; for (int k = 0 ; k < size ; k ++) { scc [k] = temp [k + 1]; } result [sccIndex ++] = scc ; } } // Trim result to actual number of SCCs

int [][] finalResult = new int [sccIndex][]; for (int i = 0 ; i < sccIndex ; i ++) { finalResult [i] = result [i]; } return finalResult ; } } class Program { static void Main () { GfG obj = new GfG (); int V = 5 ; int [,] edges = { { 1 , 3 }, { 1 , 4 }, { 2 , 1 }, { 3 , 2 }, { 4 , 5 } }; int M = edges . GetLength (0); int [][] ans = obj . findSCC (V , edges , M); Console . WriteLine ("Strongly Connected Components are:"); foreach (int [] scc in ans) { foreach (int node in scc) { Console . Write (node + " "); } Console . WriteLine (); } } } JavaScript class GfG { // Function to reach the destination using DFS dfs (curr , des , adj , vis) { // If the current node is the destination, return true if (curr === des) { return true ; } vis [curr] = 1 ; for (let x of adj [curr]) { if (! vis [x]) { if (this . dfs (x , des , adj , vis)) { return true ; } } } return false ; } // Check whether there is a path from source to destination isPath (src , des , adj) { const vis = new Array (adj . length + 1). fill (0); return this . dfs (src , des , adj , vis); } // Function to find all strongly connected components of a graph findSCC (n , a) { // Stores all strongly connected components const ans = []; // Stores whether a vertex is part of any Strongly Connected Component const is_scc = new Array (n + 1). fill (0); const adj = new Array (n + 1). fill (). map (() => []); for (let i = 0 ; i < a . length ; i ++) { adj [a [i][0]]. push (a [i][1]); } // Traversing all the vertices for (let i = 1 ; i <= n ; i ++) { if (! is_scc [i]) { // If a vertex i is not part of any SCC, // insert it into a new SCC list and check // for other vertices that can be part of this list. const scc = [i]; for (let j = i + 1 ; j <= n ; j ++) { // If there is a path from vertex i to // vertex j and vice versa, put vertex j // into the current SCC list. if (! is_scc [j] && this . isPath (i , j , adj) && this . isPath (j , i , adj)) { is_scc [j] = 1 ; scc . push (j); } } // Insert the SCC containing vertex i into the final list. ans . push (scc); } } return ans ; } } const obj = new GfG (); const V = 5 ; const edges = [[1 , 3], [1 , 4], [2 , 1], [3 , 2], [4 , 5]]; const ans = obj . findSCC (V , edges); console . log ("Strongly Connected Components are:"); for (let x of ans) { console . log (x . join (" ")); } Output Strongly Connected Components are: 1 2 3 4 5 Time complexity: O((n^2) * (n + m)), because for each pair of vertices we are checking whether a path exists between them. Auxiliary Space: O(N) [Expected Approach 1] Using Kasaraju's Algorithm Performing Depth-First Search (DFS) on the Original Graph : We first do a DFS on the original graph and record the finish times of nodes (i.e., the time at which the DFS finishes exploring a node completely). Performing DFS on the Transposed Graph : We then reverse the direction of all edges in the graph to create the transposed graph. Next, we perform a DFS on the transposed graph, considering nodes in decreasing order of their finish times recorded in the first phase. Each DFS traversal in this phase will give us one SCC. Let us take the following graph as an example. It has 3 strongly connected components. Connected Components How does this work? The above algorithm does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other that its own SCC), will always be greater than finish time of vertices in the other SCC. For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4. In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one. Graph of SCCs C++ #include <iostream> #include <vector> #include <stack> using namespace std ; class GfG { public : // Run a dfs on the original graph void DFS1 (int u , vector < vector < int >>& adj , vector < bool >& visited , stack < int >& st) { visited [u] = true ; for (int v : adj [u]) { if (! visited [v]) DFS1 (v , adj , visited , st); } st . push (u); } // DFS on reversed graph to collect SCC void DFS2 (int u , vector < vector < int >>& revAdj , vector < bool >& visited , vector < int >& scc) { visited [u] = true ; scc . push_back (u); for (int v : revAdj [u])

{ if (! visited [v]) DFS2 (v , revAdj , visited , scc); } } vector < vector < int >> kosaraju (int V , vector < vector < int >>& adj) { vector < bool > visited (V , false); stack < int > st ; // Fill stack with finish time order for (int i = 0 ; i < V ; i ++) { if (! visited [i]) DFS1 (i , adj , visited , st); } // Reverse the graph vector < vector < int >> revAdj (V); for (int u = 0 ; u < V ; u ++) { for (int v : adj [u]) { revAdj [v]. push_back (u); } } // Process reversed graph in order of stack fill (visited . begin (), visited . end (), false); vector < vector < int >> SCCs ; while (! st . empty ()) { int u = st . top (); st . pop (); if (! visited [u]) { vector < int > scc ; DFS2 (u , revAdj , visited , scc); SCCs . push_back (scc); } } return SCCs ; } }; int main () { GfG obj ; int V = 5 ; vector < vector < int > > edges { { 1 , 3 }, { 1 , 4 }, { 2 , 1 }, { 3 , 2 }, { 4 , 5 } }; vector < vector < int >> adj (V + 1); for (int i = 0 ; i < edges . size (); i ++) { int u = edges [i][0]; int v = edges [i][1]; adj [u]. push_back (v); } vector < vector < int >> SCCs = obj . kosaraju (V + 1 , adj); cout << "Strongly Connected Components: \n " ; for (int i = 0 ; i < SCCs . size () - 1 ; i ++) { for (int node : SCCs [i]) { cout << node << " " ; } cout << " \n " ; } return 0 ; } Java import java.util.Stack ; class GfG { // Run a dfs on the original graph void DFS1 (int u , int [][] adj , boolean [] visited , Stack < Integer > st) { visited [u] = true ; for (int v : adj [u]) { if (v == - 1) break ; // stop at -1 (our custom end marker) if (! visited [v]) DFS1 (v , adj , visited , st); } st . push (u); } // DFS on reversed graph to collect SCC void DFS2 (int u , int [][] revAdj , boolean [] visited , int [] scc , int [] idx) { visited [u] = true ; scc [idx [0]++] = u ; for (int v : revAdj [u]) { if (v == - 1) break ; if (! visited [v]) DFS2 (v , revAdj , visited , scc , idx); } } int [][] kosaraju (int V , int [][] adj) { boolean [] visited = new boolean [V] ; Stack < Integer > st = new Stack <> (); // Fill stack with finish time order for (int i = 0 ; i < V ; i ++) { if (! visited [i]) DFS1 (i , adj , visited , st); } // Reverse the graph int [][] revAdj = new int [V][V] ; for (int i = 0 ; i < V ; i ++) { for (int j = 0 ; j < V ; j ++) revAdj [i][j] = - 1 ; } int [] count = new int [V] ; for (int u = 0 ; u < V ; u ++) { for (int v : adj [u]) { if (v == - 1) break ; revAdj [v][count [v]++] = u ; } } // Process reversed graph in order of stack for (int i = 0 ; i < V ; i ++) visited [i] = false ; int [][] SCCs = new int [V][V] ; for (int i = 0 ; i < V ; i ++) { for (int j = 0 ; j < V ; j ++) SCCs [i][j] = - 1 ; } int sccCount = 0 ; while (! st . isEmpty ()) { int u = st . pop (); if (! visited [u]) { int [] scc = new int [V] ; for (int i = 0 ; i < V ; i ++) scc [i] = - 1 ; int [] idx = { 0 }; DFS2 (u , revAdj , visited , scc , idx); SCCs [sccCount ++] = scc ; } } int [][] result = new int [sccCount][] ; for (int i = 0 ; i < sccCount ; i ++) result [i] = SCCs [i] ; return result ; } static int [][] buildAdjMatrix (int [][] edges , int V) { int [][] adj = new int [V + 1][V + 1] ; for (int i = 0 ; i <= V ; i ++) { for (int j = 0 ; j <= V ; j ++) { adj [i][j] = - 1 ; } } int [] count = new int [V + 1] ; for (int [] edge : edges) { int u = edge [0] , v = edge [1] ; adj [u][count [u]++] = v ; } return adj ; } public static void main (String [] args) { GfG obj = new GfG (); int V = 5 ; int [][] edges = { { 1 , 3 }, { 1 , 4 }, { 2 , 1 }, { 3 , 2 }, { 4 , 5 } }; int [][] adj = buildAdjMatrix (edges , V); int [][] SCCs = obj . kosaraju (V + 1 , adj); System . out . println ("Strongly Connected Components:"); for (int i = 0 ; i < SCCs . length - 1 ; i ++) { for (int j = 0 ; j < SCCs [i] . length && SCCs [i][j] != - 1 ; j ++) { System . out . print (SCCs [i][j] + " "); } System . out . println (); } } } Python from collections import defaultdict class GfG : # Run a dfs on the original graph def DFS1 (self , u , adj , visited , st): visited [u] = True for v in adj [u]: if not visited [v]: self . DFS1 (v , adj , visited , st) st . append (u) # DFS on reversed graph to collect SCC def DFS2 (self , u , revAdj , visited , scc): visited [u] = True scc . append (u) for v in revAdj [u]: if not visited [v]: self . DFS2 (v , revAdj , visited , scc) def kosaraju (self , V , adj): visited = [False] * V st = [] # Fill stack with finish time order for i in range (V): if not visited [i]: self . DFS1 (i , adj , visited , st) # Reverse the graph revAdj = [[] for _ in range (V)] for u in range (V): for v in adj [u]: revAdj [v] . append (u) # Process reversed graph in order of stack visited = [False] * V SCCs = [] while st : u = st . pop () if not visited [u]: scc = [] self . DFS2 (u , revAdj , visited , scc) SCCs . append (scc) return SCCs if __name__ == "__main__" : obj = GfG () V = 5 edges = [[1 , 3], [1 , 4], [2 , 1], [3 , 2], [4 , 5]] adj = [[] for _ in range (V + 1)] for u , v in edges : adj [u] . append (v) SCCs = obj . kosaraju (V + 1 , adj) print ("Strongly Connected Components:") for i in range (len (SCCs) - 1): for node in SCCs [i]: print (node , end = " ") print () C# using System ; class GfG { // Run a dfs on the original graph void DFS1 (int u , int [][] adj , bool [] visited , int [] stack , ref int top) { visited [u] = true ; foreach (int v in adj [u]) { if (v == - 1) break ; if (! visited [v]) DFS1 (v , adj , visited , stack , ref top); } stack [++ top] = u ; } // DFS on reversed graph to collect SCC void DFS2 (int u , int [][] revAdj , bool [] visited , int [] scc , ref int idx) { visited [u] = true ; scc [idx ++] = u ; foreach (int v in revAdj [u]) { if (v == - 1) break ; if (! visited [v]) DFS2 (v , revAdj , visited , scc , ref idx); } } public int [][] kosaraju (int V , int [][] adj) { bool [] visited = new bool [V]; int [] stack = new int [V]; int top = - 1 ; // Fill stack with finish time order for (int i = 0 ; i < V ; i ++) { if (! visited [i]) DFS1 (i , adj , visited , stack , ref top); } // Reverse the graph int [][] revAdj = new int [V][]; int [] revCount = new int [V]; for (int i = 0 ; i < V ; i ++) { revAdj [i] = new int [V]; for (int j = 0 ; j < V ; j ++) revAdj [i][j] = - 1 ; } for (int u = 0 ; u < V ; u ++) { foreach (int v in adj [u]) { if (v == - 1) break ; revAdj [v][revCount [v] ++] = u ; } } // Process reversed graph in order of

stack for (int i = 0 ; i < V ; i ++) visited [i] = false ; int [][] SCCs = new int [V][]; for (int i = 0 ; i < V ; i ++) { SCCs [i] = new int [V]; for (int j = 0 ; j < V ; j ++) SCCs [i][j] = - 1 ; } int sccCount = 0 ; while (top >= 0) { int u = stack [top --]; if (! visited [u]) { int [] scc = new int [V]; for (int i = 0 ; i < V ; i ++) scc [i] = - 1 ; int idx = 0 ; DFS2 (u , revAdj , visited , scc , ref idx); SCCs [sccCount ++] = scc ; } } int [][] result = new int [sccCount][]; for (int i = 0 ; i < sccCount ; i ++) result [i] = SCCs [i]; return result ; } static int [][] BuildAdjMatrix (int [][] edges , int V) { int [][] adj = new int [V + 1][]; int [] count = new int [V + 1]; for (int i = 0 ; i <= V ; i ++) { adj [i] = new int [V + 1]; for (int j = 0 ; j <= V ; j ++) adj [i][j] = - 1 ; } foreach (int [] edge in edges) { int u = edge [0], v = edge [1]; adj [u][count [u] ++] = v ; } return adj ; } public static void Main () { GfG obj = new GfG (); int V = 5 ; int [][] edges = new int [][] { new int [] { 1 , 3 }, new int [] { 1 , 4 }, new int [] { 2 , 1 }, new int [] { 3 , 2 }, new int [] { 4 , 5 } }; int [][] adj = BuildAdjMatrix (edges , V); int [][] SCCs = obj . kosaraju (V + 1 , adj); Console . WriteLine ("Strongly Connected Components:"); for (int i = 0 ; i < SCCs . Length - 1 ; i ++) { for (int j = 0 ; j < SCCs [i]. Length && SCCs [i][j] != - 1 ; j ++) { Console . Write (SCCs [i][j] + " "); } Console . WriteLine (); } } } JavaScript class GfG { // DFS on original graph to fill the stack with finishing times DFS1 (u , adj , visited , st) { visited [u] = true ; for (let v of adj [u]) { if (! visited [v]) this . DFS1 (v , adj , visited , st); } st . push (u); } // DFS on reversed graph to collect SCC DFS2 (u , revAdj , visited , scc) { visited [u] = true ; scc . push (u); for (let v of revAdj [u]) { if (! visited [v]) this . DFS2 (v , revAdj , visited , scc); } } kosaraju (V , adj) { let visited = Array (V). fill (false); let st = []; // Fill stack with finish time order for (let i = 0 ; i < V ; i ++) { if (! visited [i]) this . DFS1 (i , adj , visited , st); } // Reverse the graph let revAdj = Array . from ({ length : V }, () => []); for (let u = 0 ; u < V ; u ++) { for (let v of adj [u]) { revAdj [v]. push (u); } } // Process reversed graph in order of stack visited . fill (false); let SCCs = []; while (st . length > 0) { let u = st . pop (); if (! visited [u]) { let scc = []; this . DFS2 (u , revAdj , visited , scc); SCCs . push (scc); } } return SCCs ; } } (function () { const obj = new GfG (); const V = 5 ; const edges = [[1 , 3], [1 , 4], [2 , 1], [3 , 2], [4 , 5]]; const adj = Array . from ({ length : V + 1 }, () => []); for (let [u , v] of edges) { adj [u]. push (v); } const SCCs = obj . kosaraju (V + 1 , adj); console . log ("Strongly Connected Components:"); for (let i = 0 ; i < SCCs . length - 1 ; i ++) { console . log (SCCs [i]. join (" ")); } })(); Output Strongly Connected Components: 1 2 3 4 5 Time Complexity - O(V + E) Auxiliary Space - O(V + E) [Expected Approach 2] Using Tarjan's Algorithm Tarjan's Algorithm is more efficient because it finds SCCs in a single DFS pass using a stack and some additional bookkeeping: DFS Traversal : During the DFS, maintain an index for each node and the smallest index (low-link value) that can be reached from the node. Stack : Keep track of nodes currently in the recursion stack (part of the current SCC being explored). Identifying SCCs : When a node’s low-link value equals its index, it means we have found an SCC. Pop all nodes from the stack until we reach the current node. Please refer Tarjan's Algorithm for more details Comment Article Tags: Article Tags: Graph DSA Visa DFS graph-connectivity + 1 More