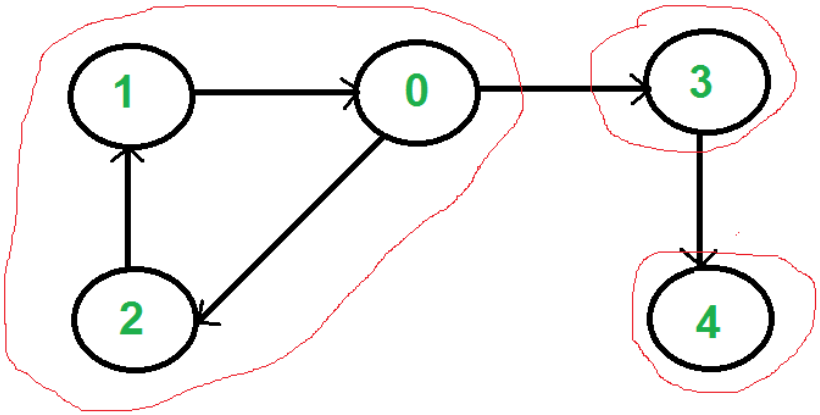
Kosaraju's Algorithm | Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



We can find all strongly connected components in O(V+E) time using

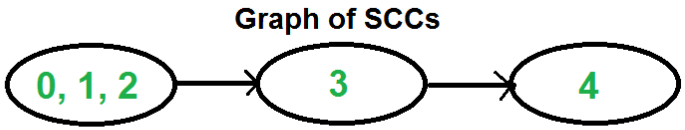
Kosaraju's algorithm
. Following is detailed Kosaraju's algorithm.

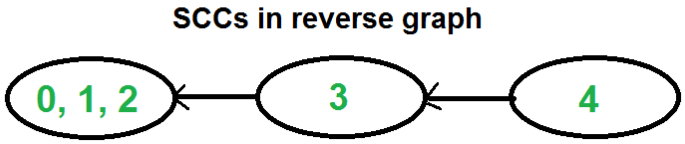
1. Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in the stack as 1, 2, 4, 3, 0.
2. Reverse directions of all arcs to obtain the transpose graph.
3. One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

How does this work?

The above algorithm is DFS based. It 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 at 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 sunk 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 of 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 than its own SCC), will always be greater than finish time of vertices in the other SCC (See this for proof). For example, in DFS of above example graph, the finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And the 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 the 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 the stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in the stack, we process vertices from the sink to the source (in the reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.





Dash
All
Articles
Videos
Problems
Quiz
Contest

Below is the implementation of Kosaraju's algorithm:

C++ Java

```
// Java implementation of Kosaraju's algorithm to print all SCCs

import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
```

```
        adj[i] = new LinkedList();
}

//Function to add an edge into the graph
void addEdge(int v, int w)  { adj[v].add(w); }

// A recursive function to print DFS starting from v
void DFSUtil(int v,boolean visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    System.out.print(v + " ");

    int n;

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> i =adj[v].iterator();
    while (i.hasNext())
    {
        n = i.next();
        if (!visited[n])
            DFSUtil(n,visited);
    }
}

// Function that returns reverse (or transpose) of this graph
Graph getTranspose()
{
    Graph g = new Graph(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i =adj[v].listIterator();
        while(i.hasNext())
            g.adj[i.next()].add(v);
    }
    return g;
}

void fillOrder(int v, boolean visited[], Stack stack)
{
```

```
        // Mark the current node as visited and print it
        visited[v] = true;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].iterator();
        while (i.hasNext())
        {
            int n = i.next();
            if(!visited[n])
                fillOrder(n, visited, stack);
        }

        // All vertices reachable from v are processed by now,
        // push v to Stack
        stack.push(new Integer(v));
    }

    // The main function that finds and prints all strongly
    // connected components
    void printSCCs()
    {
        Stack stack = new Stack();

        // Mark all the vertices as not visited (For first DFS)
        boolean visited[] = new boolean[V];
        for(int i = 0; i < V; i++)
            visited[i] = false;

        // Fill vertices in stack according to their finishing
        // times
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                fillOrder(i, visited, stack);

        // Create a reversed graph
        Graph gr = getTranspose();

        // Mark all the vertices as not visited (For second DFS)
        for (int i = 0; i < V; i++)
            visited[i] = false;
```

```
        // Now process all vertices in order defined by Stack
        while (stack.empty() == false)
        {
            // Pop a vertex from stack
            int v = (int)stack.pop();

            // Print Strongly connected component of the popped vertex
            if (visited[v] == false)
            {
                gr.DFSUtil(v, visited);
                System.out.println();
            }
        }
    }
```



```
    {
        // Create a graph given in the above diagram
        Graph g = new Graph(5);
        g.addEdge(1, 0);
        g.addEdge(0, 2);
        g.addEdge(2, 1);
        g.addEdge(0, 3);
        g.addEdge(3, 4);

        System.out.println("Following are strongly connected components "+
                           "in given graph ");
        g.printSCCs();
    }
}
// This code is contributed by Aakash Hasija
```

Output:

```
Following are strongly connected components in given graph
0 1 2
3
4
```

Time Complexity:
The above algorithm calls DFS, fins reverse of the graph and again calls DFS. DFS takes O(V+E) for a graph represented using adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simply traverse all adjacency lists.

Auxiliary Space: O(V+E)

Applications:

SCC algorithms can be used as the first step in many graph algorithms that work only on a strongly connected graph. In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked commonly liked a page or played a game.