Algorithm ***DFS***(*G*)
// Implements a depth-first search traversal of a given graph
// Input: Graph *G* = (*V*, *E*)
// Output: Graph G with its vertices marked with consecutive integers in the order they
// are first encountered by the DFS traversal

mark each vertex in *V* with 0 as a mark of being "*unvisited*"
count ← 0
for each vertex *v* in *V* do
    if *v* is ("*unvisited*") marked with 0, *dfs*(*v*)

dfs(*v*)
// visits recursively all the unvisited vertices connected to a vertex *v* by a path
// and numbers them in the order they are encountered via global variable *count*

```
        count ← count + 1; mark v with count ("visited")     (1)
        for each vertex w in the adjacency list of v do      (2)
                if w is marked with 0                        (3)
                        dfs(w)                               (4)
```

/* end dfs */

Algorithm ***TopologicalSort*****(vertex** ***v*****)**
// prints the vertices accessible from v in reverse topological order
mark *v* with 1 ("visited")
for each vertex *w* in the adjacency list of *v* **do**
   if *w* is "unvisited" (marked with 0) *TopologicalSort*(*w*)
print *v*

When topological sort finishes searching all vertices adjacent to a given vertex *x*, it prints *x*. The effect of calling *topologicalSort*(*v*) is to print in a reverse topological order all vertices of a directed acyclic graph (dag) accessible from *v* by a path in a dag.

Algorithm to find the strongly connected components of a given digraph G:

(1) Perform a depth-first search of G and number the vertices in order of completion of the recursive calls; i.e., assign a number to vertex v after line (4).

(2) Construct a new directed graph G_r, by reversing the direction of every arc in G.

(3) Perform a depth-first search in G_r, starting the search from the highest-numbered vertex according to the numbering assigned at step (1). If the depth-first search does not reach all vertices, start the next depth-first search from the highest-numbered remaining vertex.

(4) Each tree in the resulting spanning forest is a strongly connected component of G.

Proof: We have claimed that the vertices of a strongly connected component correspond precisely to the vertices of a tree in the spanning forest of the second depth-first search. To see why, observe that if v and w are vertices in the same strongly connected component, then there are paths in G from v to w and from w to v. Thus there are also paths from v to w and from w to v in G_r.

Suppose that in the depth-first search of G_r, we begin a search at some root x and reach either v or w. Since v and w are reachable from each other, both v and w will end up in the spanning tree with root x.

Now suppose v and w are in the same spanning tree of the depth-first spanning forest of G_r. We must show that v and w are in the same strongly connected component. Let x be the root of the spanning tree containing v and w. Since v is a descendant of x, there exists a path in G_r from x to v. Thus there exists a path in G from v to x.

In the construction of the depth-first spanning forest of G_r, vertex v was still unvisited when the depth-first search at x was initiated. Thus x has a higher number than v, so in the depth first search of G, the recursive call at v terminated before the recursive call at x did. But in the depth-first search of G, the search at v could not have started before x, since the path in G from v to x would then imply that the search at x would start and end before the search at v ended.

We conclude that in the search of G, v is visited during the search of x and hence v is a descendant of x in the depth-first spanning forest for G. Thus there exists a path from x to v in G. Therefore x and v are in the same strongly connected component. An identical argument shows that x and w are in the same strongly connected component and hence v and w are in the same strongly connected component, as shown by the path from v to x and the path from w to x to v.