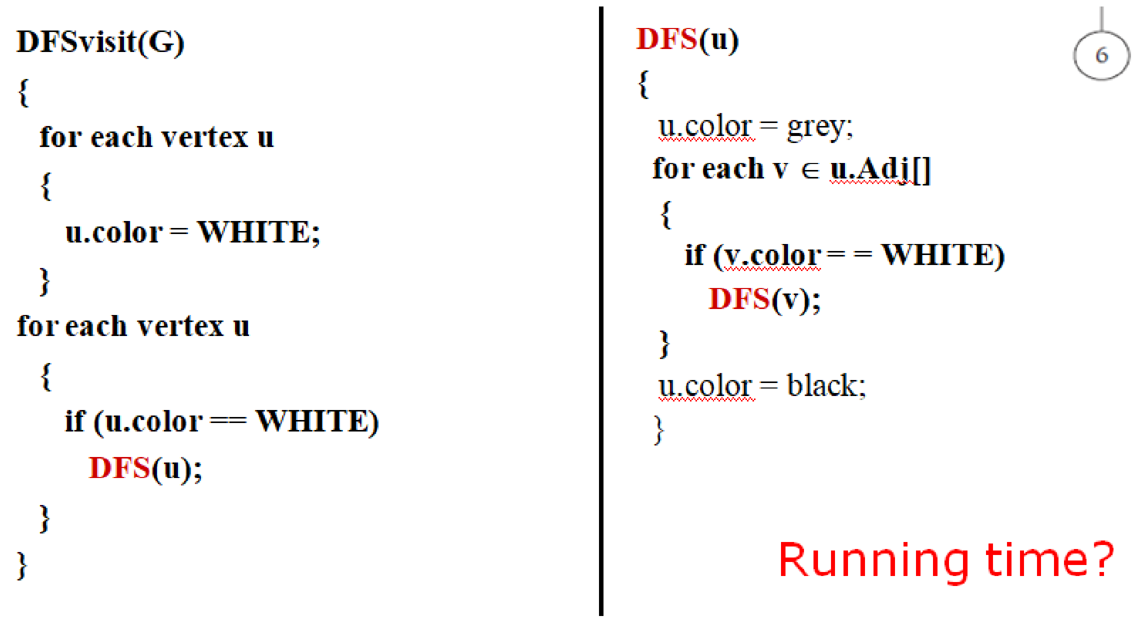
Algorithm 0x03

49.234.77.58/index.php/2019/12/15/algorithm-0x03

XZLang

2019年12月15日

DFS

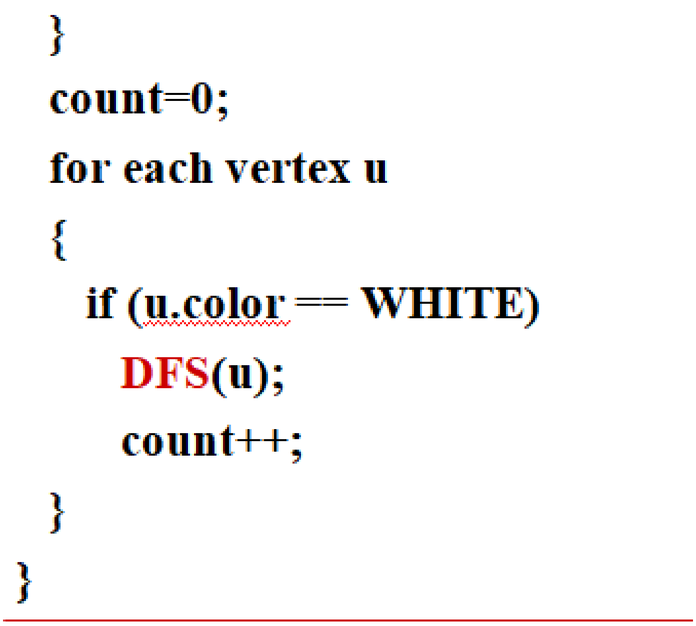


```
DFSvisit(G)
{
  for each vertex u
  {
    u.color = WHITE;
  }
for each vertex u
  {
    if (u.color == WHITE)
      DFS(u);
  }
}
```

```
DFS(u)
{
  u.color = grey;
 for each v ∈ u.Adj[]
  {
    if (v.color = = WHITE)
      DFS(v);
  }
  u.color = black;
 }
```

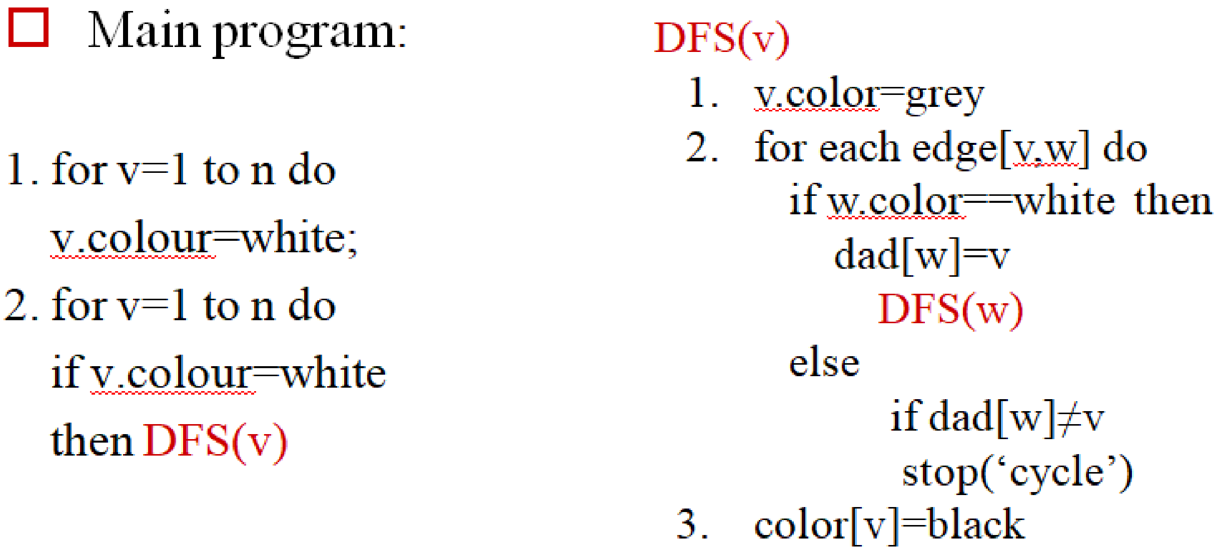
Running time?

Count the number of connected components in given graph G



```
}
count=0;
for each vertex u
{
  if (u.color == WHITE)
    DFS(u);
    count++;
}
}
```

Test whether graph G contains a cycle or not



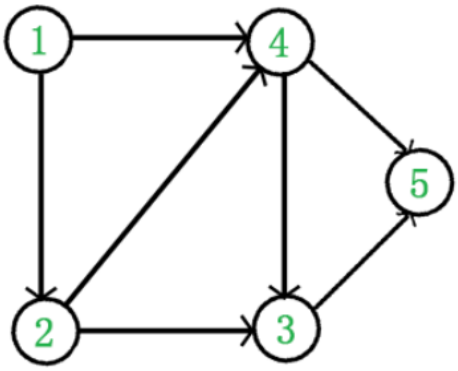
☐ Main program:

```
1. for v=1 to n do
       v.colour=white;
2. for v=1 to n do
       if v.colour=white
       then DFS(v)
```

```
DFS(v)
  1.  v.color=grey
  2.  for each edge[v,w] do
          if w.color==white  then
              dad[w]=v
                  DFS(w)
          else
                  if dad[w]≠v
                   stop('cycle')
  3.  color[v]=black
```

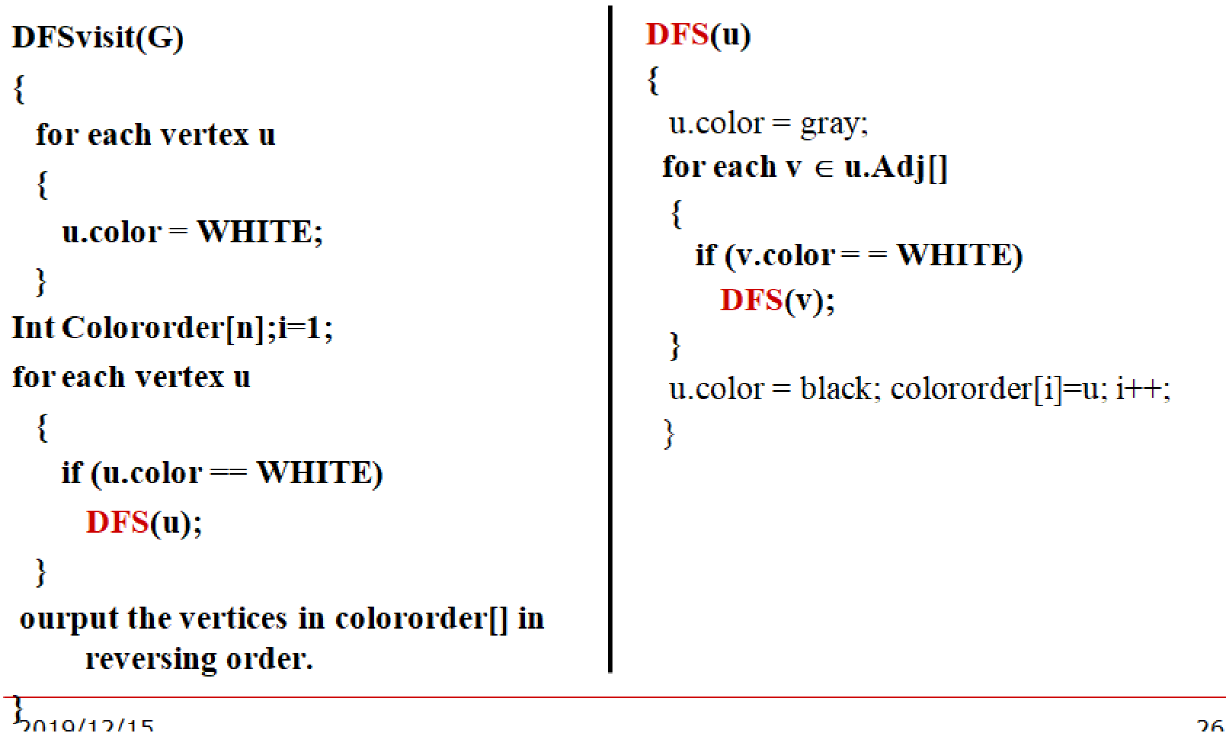
在图论中，拓扑排序（Topological Sorting）是一个有向无环图（DAG, Directed Acyclic Graph）的所有顶点的线性序列。且该序列必须满足下面两个条件：

1. 每个顶点出现且只出现一次。
2. 若存在一条从顶点 A 到顶点 B 的路径，那么在序列中顶点 A 出现在顶点 B 的前面。
3. 有向无环图（DAG）才有拓扑排序，非DAG图没有拓扑排序一说。



它是一个 DAG 图，那么如何写出它的拓扑排序呢？这里说一种比较常用的方法：

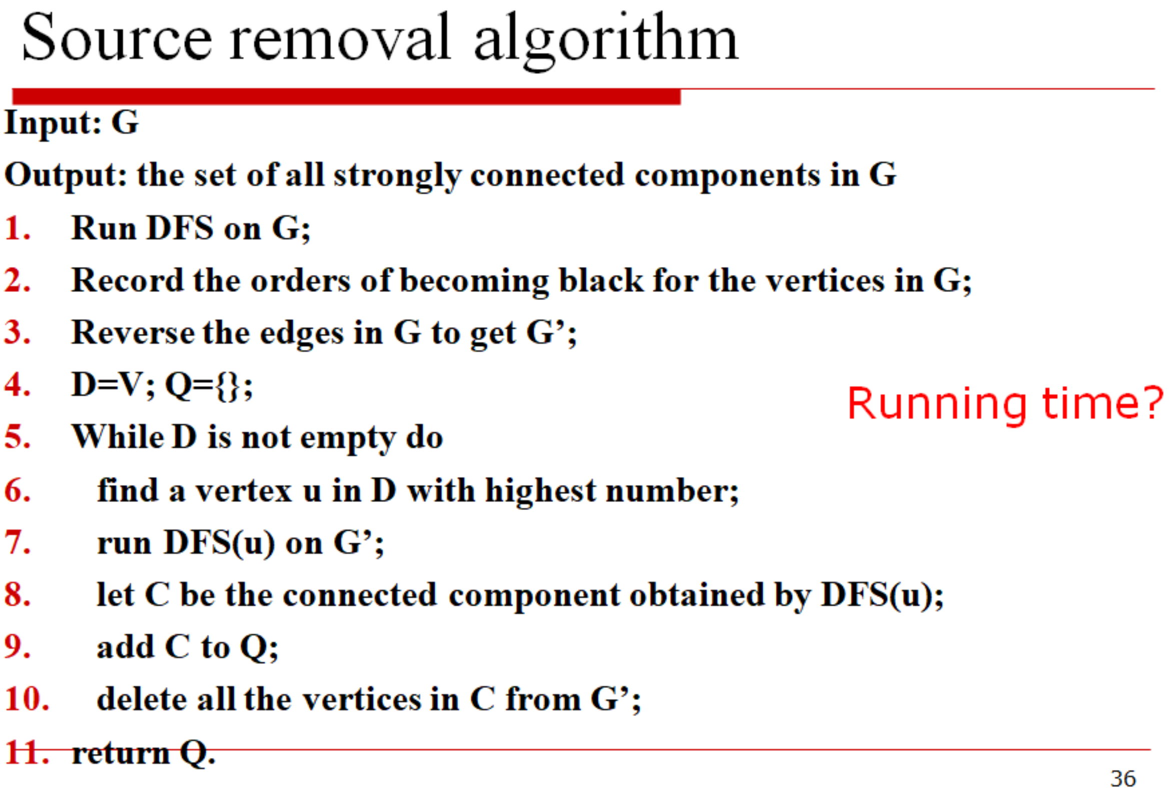
1. 从 DAG 图中选择一个 没有前驱（即入度为0）的顶点并输出。
2. 从图中删除该顶点和所有以它为起点的有向边。
3. 重复 1 和 2 直到当前的 DAG 图为空或**当前图中不存在无前驱的顶点为止**。后一种情况说明有向图中必然存在环。



```
DFSvisit(G)
{
  for each vertex u
  {
    u.color = WHITE;
  }
Int Colororder[n];i=1;
for each vertex u
  {
    if (u.color == WHITE)
      DFS(u);
  }
 ourput the vertices in colororder[] in
      reversing order.
}
```

```
DFS(u)
{
  u.color = gray;
 for each v ∈ u.Adj[]
  {
    if (v.color = = WHITE)
      DFS(v);
  }
  u.color = black; colororder[i]=u; i++;
 }
```

Give a graph G, find all the strongly connected components in G



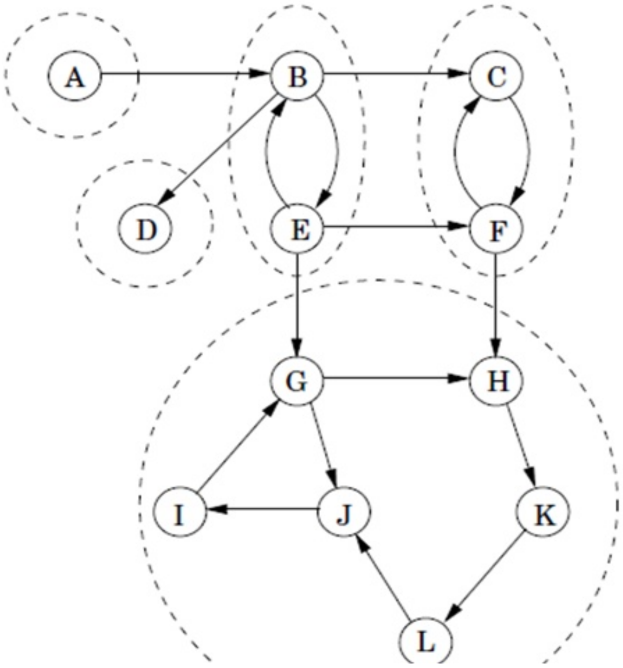
Source removal algorithm

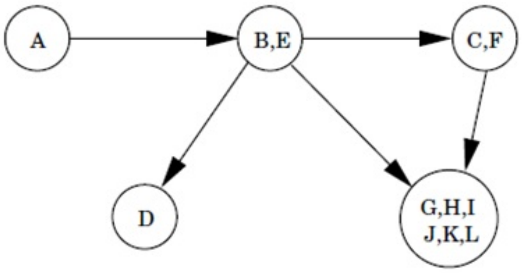
Input: G

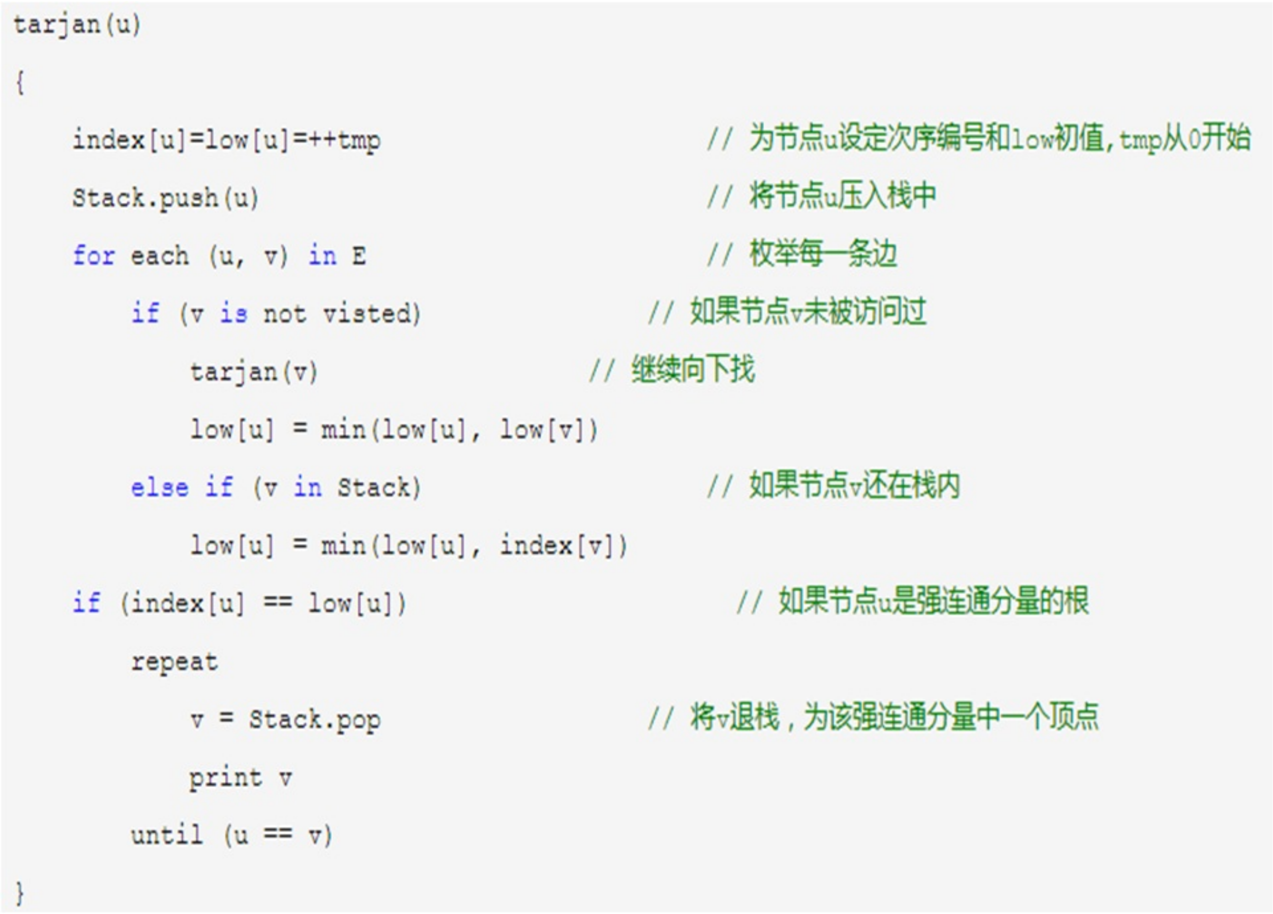
Output: the set of all strongly connected components in G

```
Run DFS on G;
Record the orders of becoming black for the vertices in G;
Reverse the edges in G to get G';
D=V; Q={};
While D is not empty do
  find a vertex u in D with highest number;
  run DFS(u) on G';
  let C be the connected component obtained by DFS(u);
  add C to Q;
  delete all the vertices in C from G';
return Q.
```

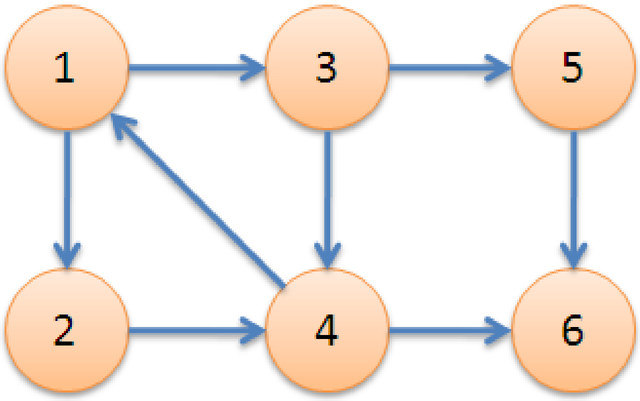
Running time?







```
tarjan(u)
{
    index[u]=low[u]=++tmp                    // 为节点u设定次序编号和low初值,tmp从0开始
    Stack.push(u)                            // 将节点u压入栈中
    for each (u, v) in E                     // 枚举每一条边
        if (v is not visted)             // 如果节点v未被访问过
            tarjan(v)                // 继续向下找
            low[u] = min(low[u], low[v])
        else if (v in Stack)                 // 如果节点v还在栈内
            low[u] = min(low[u], index[v])
    if (index[u] == low[u])                    // 如果节点u是强连通分量的根
        repeat
            v = Stack.pop                // 将v退栈，为该强连通分量中一个顶点
            print v
        until (u == v)
}
```



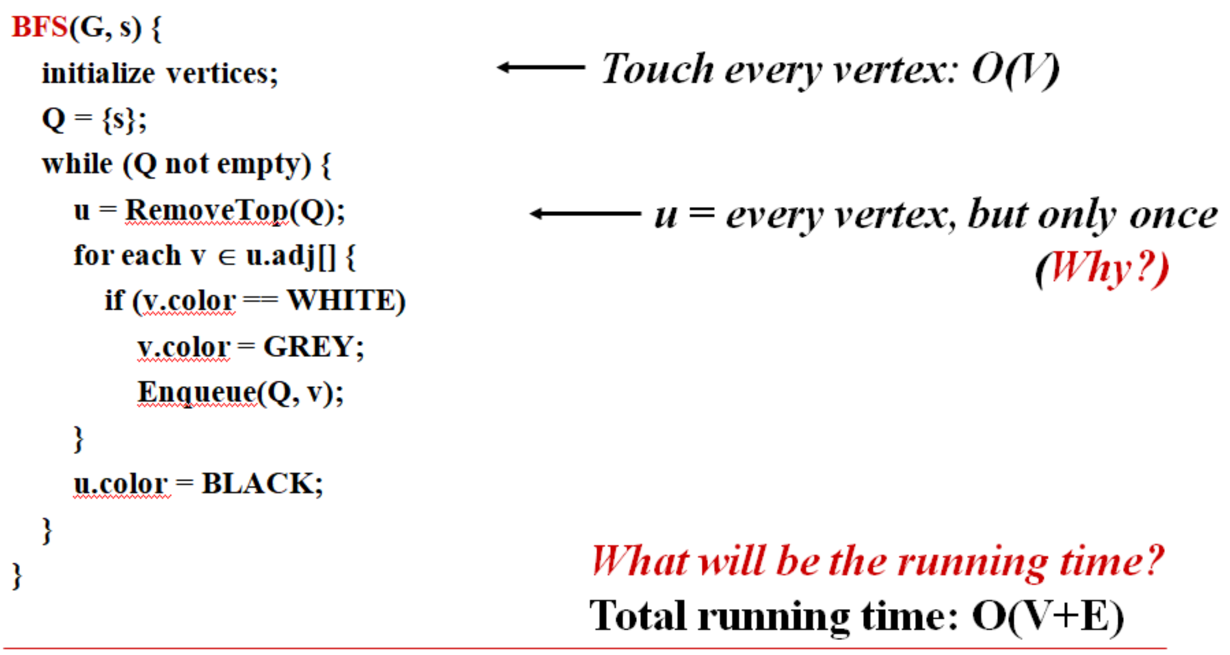
Tarjan算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一棵子树。搜索时，把当前搜索树中未处理的节点加入一个堆栈，回溯时可以判断栈顶到栈中的节点是否为一个强连通分量。

定义DFN(u)为节点u搜索的次序编号(时间戳)，Low(u)为u或u的子树能够追溯到的最早的栈中节点的次序号。由定义可以得出，

```
Low(u)=Min
{
    DFN(u),
    Low(v),(u,v)为树枝边，u为v的父节点
    DFN(v),(u,v)为指向栈中节点的后向边(非横叉边)
}
```

当DFN(u)=Low(u)时，以u为根的搜索子树上所有节点是一个强连通分量。

BFS



```
BFS(G, s) {
  initialize vertices;
  Q = {s};
  while (Q not empty) {
    u = RemoveTop(Q);
    for each v ∈ u.adj[] {
      if (v.color == WHITE)
        v.color = GREY;
        Enqueue(Q, v);
    }
    u.color = BLACK;
  }
}
```

← *Touch every vertex: O(V)*

← *u = every vertex, but only once*

(Why?)

What will be the running time?

Total running time: O(V+E)