Floyd's Cycle Finding Algorithm

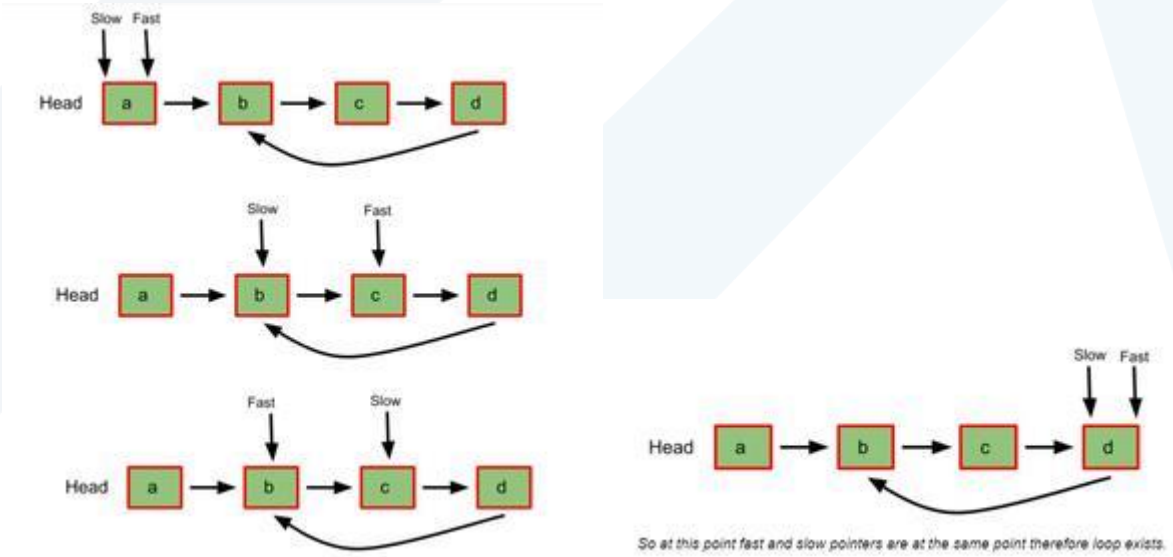
Floyd's cycle finding algorithm or Hare-Tortoise algorithm is a pointer algorithm that uses only two pointers, moving through the sequence at different speeds. This algorithm is used to find a loop in a linked list. It uses two pointers one moving twice as fast as the other one. The faster one is called the faster pointer and the other one is called the slow pointer.

How Does Floyd's Cycle Finding Algorithm Works?

While traversing the linked list one of these things will occur-

- The Fast pointer may reach the end (NULL) this shows that there is no loop n the linked list.
- The Fast pointer again catches the slow pointer at some time therefore a loop exists in the linked list.

Example:



Pseudocode:

- Initialize two-pointers and start traversing the linked list.
- Move the slow pointer by one position.
- Move the fast pointer by two positions.
- If both pointers meet at some point then a loop exists and if the fast pointer meets the end position then no loop exists.

Algorithm to find whether there is a cycle or not :

1. Declare 2 nodes say **`slowPointer`** and **`fastPointer`** pointing to the linked list head.
2. Move **`slowPointer`** by one node and **`fastPointer`** by 2 nodes till either of one reaches nil.

3. If at any point in the above traversal, **`slowPointer`** and **`fastPointer`** are found to be pointing to the same node, which implies the list has a cycle

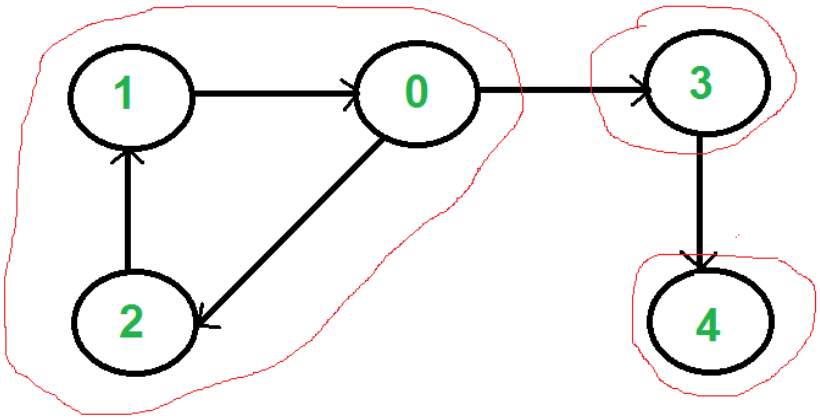
Algorithm to find the starting node of the cycle:

After figuring out whether there is a cycle or not perform the following steps

1. Reset the **`slowPointer`** to point to the head of the linked list and keep the **`fastPointer`** at the intersected position.
2. Move both the **`fastPointer`** and **`slowPointer`** pointers by one node.
3. The point at which they will intersect is the starting of the cycle.

Strongly Connected Components

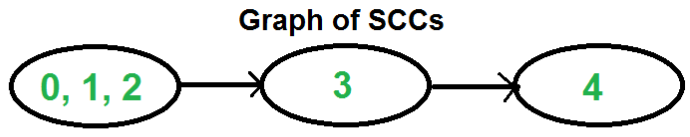
A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.

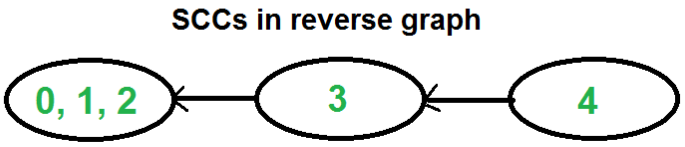


We can find all strongly connected components in O(V+E) time using Kosaraju's algorithm. Following is detailed Kosaraju's algorithm.
1) Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in 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 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.

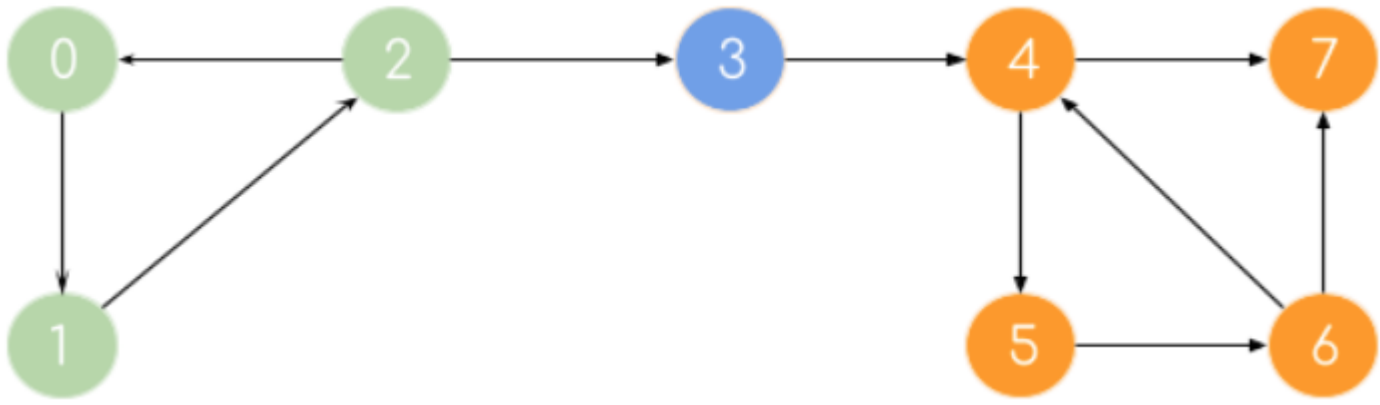




PSEUDOCODE:

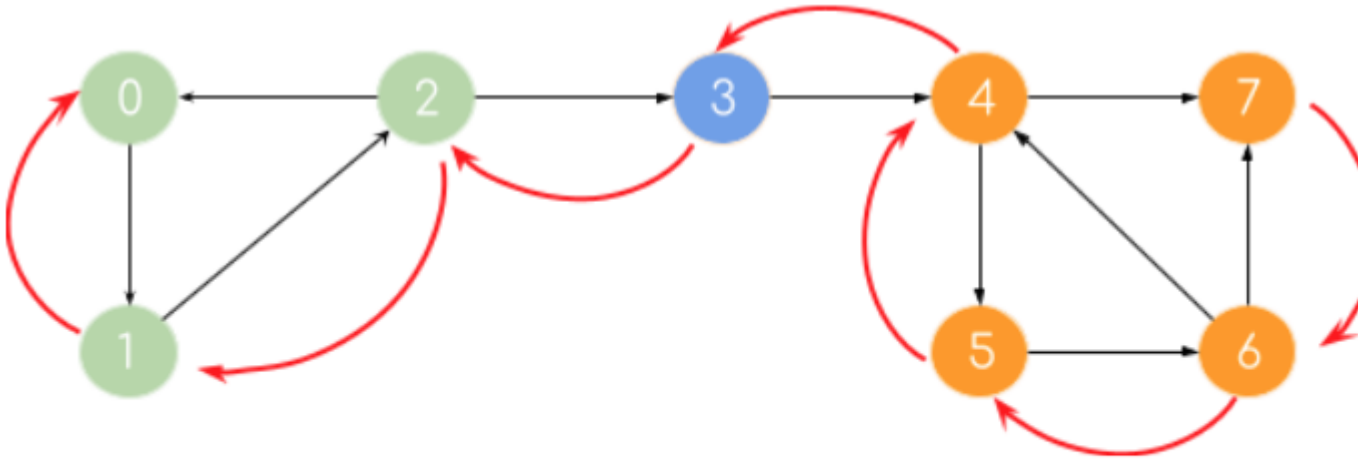
- Perform DFS traversal of the graph. Push node to stack before returning.
- Find the transpose graph by reversing the edges.
- Pop nodes one by one from the stack and again to DFS on the modified graph.

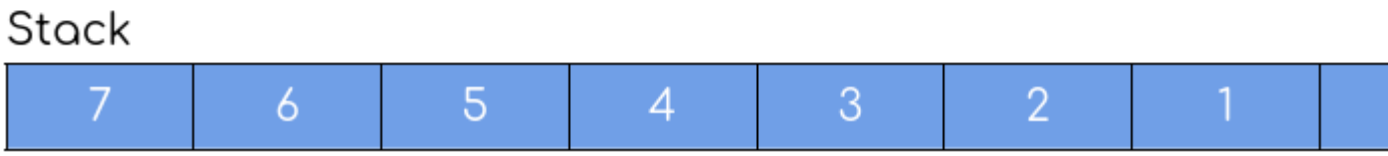
Let's show an example to support our algorithm:



We will begin DFS at 0 on the above graph and traverse every outgoing edge from each node that is unvisited. For visiting information we will make a visited array, and we should also

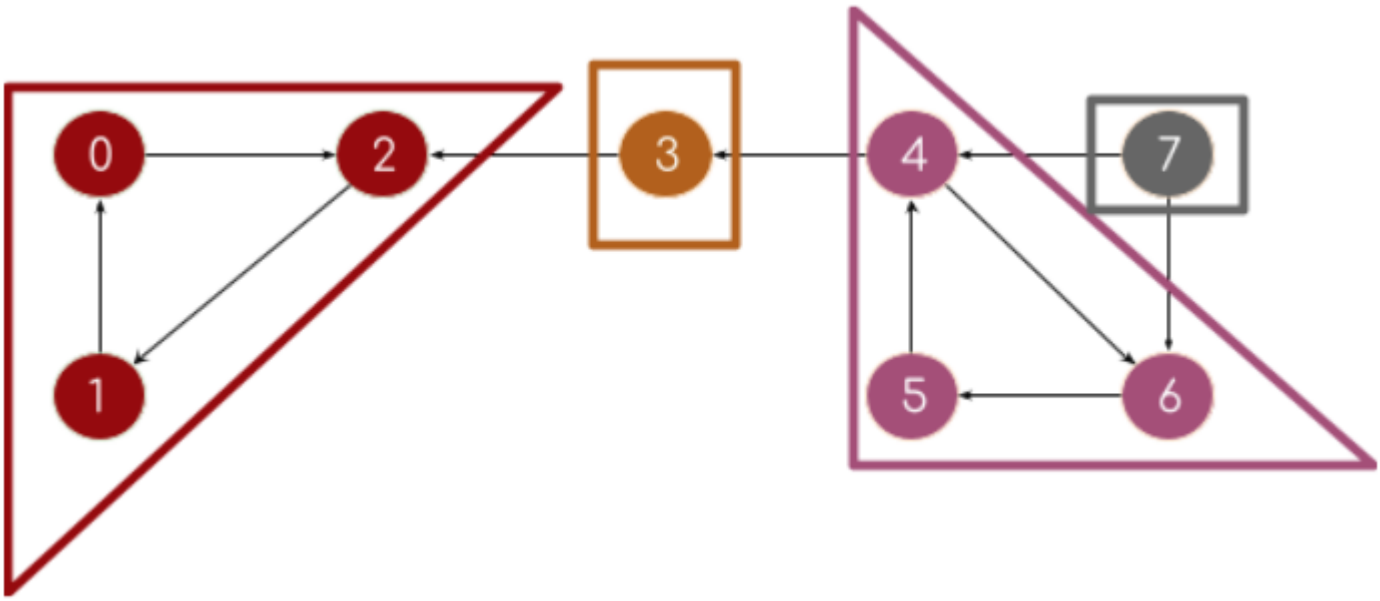
maintain a stack to store the nodes in the order of which we exhausted their outgoing edges. Now we will see the result:





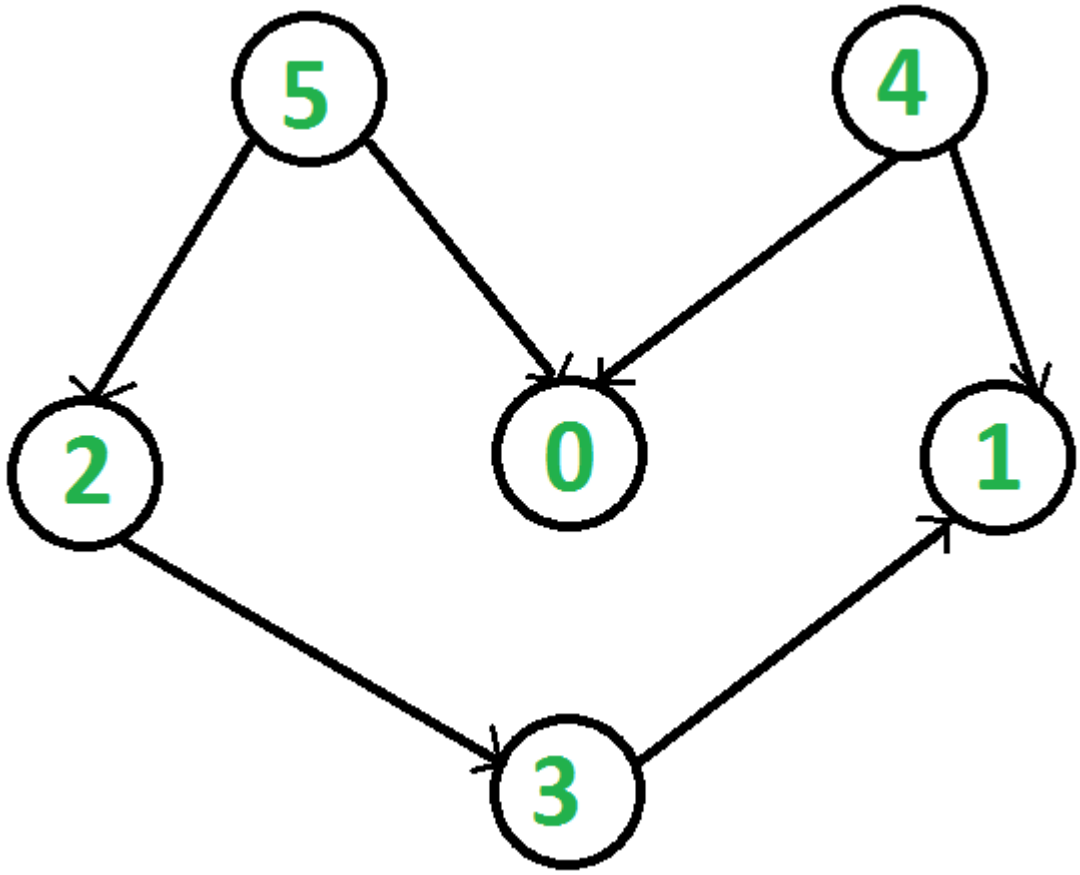
Now we pop the stack one by one and do DFS on the modified/traversed graph while popping nodes. At the end of each successful DFS we will have an SCC.

Now go ahead and perform this. For the sake of the article I won't be illustrating it (might take too many pages), I will just show the final result. Here you'll see each SCC colored differently. We will do the steps the same as the previous one, but the next node to traverse will be given by stack. Ignore those stack elements which have already been considered in previous SCCs.


0
2
3
4
7
1
5
6

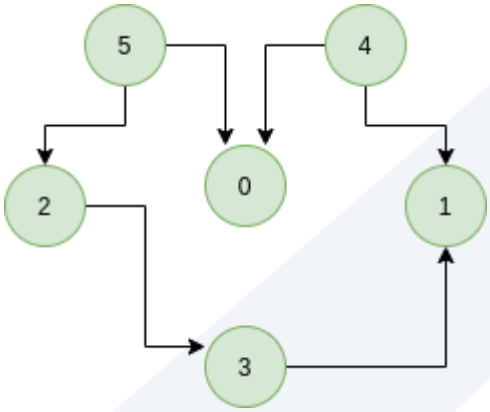
Kahn’s algorithm for Topological Sorting

Topological sorting for **D**irected **A**cyclic **G**raph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.
For example, a topological sorting of the following graph is “5 4 2 3 1 0?. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 0 3 1″. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

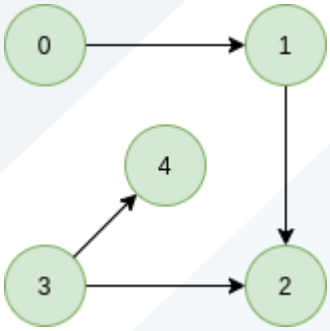


Let’s look at few examples with proper explanation,
Example:

Input:



Output: 5 4 2 3 1 0
Explanation: The topological sorting of a DAG is done in a order such that for every directed edge uv, vertex u comes before v in the ordering. 5 has no incoming edge. 4 has no incoming edge, 2 and 0 have incoming edge from 4 and 5 and 1 is placed at last.
Input:



Output: 0 3 4 1 2
Explanation: 0 and 3 have no incoming edge, 4 and 1 has incoming edge from 0 and 3. 2 is placed at last.
Solution: In this article we will see another way to find the linear ordering of vertices in a directed acyclic graph (DAG). The approach is based on the below fact:
A DAG G has at least one vertex with in-degree 0 and one vertex with out-degree 0.
Proof: There’s a simple proof to the above fact is that a DAG does not contain a cycle which means that all paths will be of finite length. Now let S be the longest path from u(source) to v(destination). Since S is the longest path there can be no incoming edge to u and no outgoing edge from v, if this situation had occurred then S would not have been the longest path
=> indegree(u) = 0 and outdegree(v) = 0
Algorithm: Steps involved in finding the topological ordering of a DAG:
Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)
Step-3: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighbouring nodes.
3. If in-degree of a neighbouring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.
Step 5: If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph.
How to find in-degree of each node?
There are 2 ways to calculate in-degree of every vertex:
Take an in-degree array which will keep track of
Traverse the array of edges and simply increase the counter of the destination node by 1.

```
for each node in Nodes
    indegree[node] = 0;
for each edge(src, dest) in Edges
    indegree[dest]++
```

1. Time Complexity: O(V+E)
2. Traverse the list for every node and then increment the in-degree of all the nodes connected to it by 1.

```
for each node in Nodes
    If (list[node].size()!=0) then
    for each dest in list
        indegree[dest]++;
```

1. Time Complexity: The outer for loop will be executed V number of times and the inner for loop will be executed E number of times, Thus overall time complexity is O(V+E).
   The overall time complexity of the algorithm is O(V+E)

Kahn's Algorithm

Steps involved in finding the topological ordering of a DAG:

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If the in-degree of a neighboring node is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.

Step 5: If the count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

In this article, we will explore how we can implement **Topological sorting** using **Kahn's algorithm**.