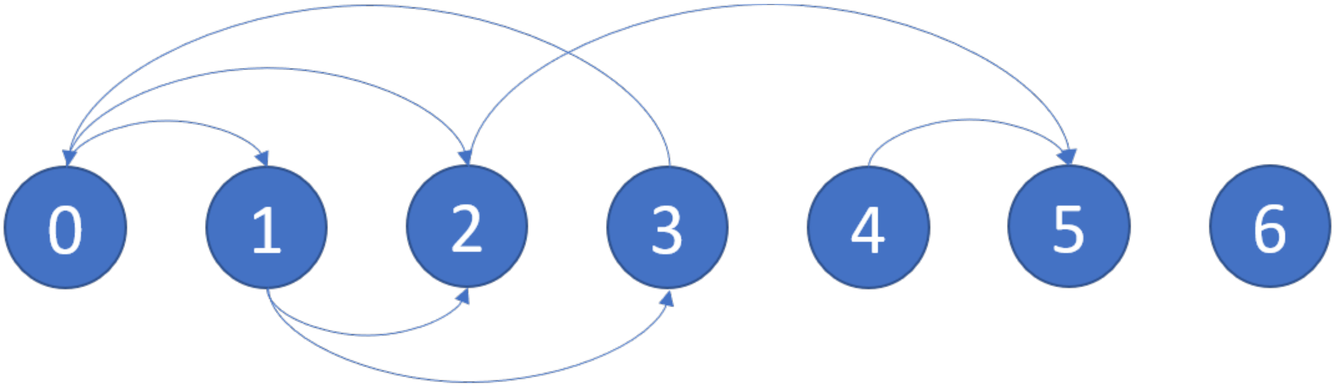
802. Find Eventual Safe States

There is a directed graph of `n` nodes with each node labeled from `0` to `n - 1`. The graph is represented by a **0-indexed** 2D integer array `graph` where `graph[i]` is an integer array of nodes adjacent to node `i`, meaning there is an edge from node `i` to each node in `graph[i]`.

A node is a **terminal node** if there are no outgoing edges. A node is a **safe node** if every possible path starting from that node leads to a **terminal node** (or another safe node).

Return *an array containing all the* ***safe nodes*** *of the graph*. The answer should be sorted in **ascending** order.

Example 1:



```
Input: graph = [[1,2],[2,3],[5],[0],[5],[],[]]
Output: [2,4,5,6]
Explanation: The given graph is shown above.
Nodes 5 and 6 are terminal nodes as there are no outgoing edges from either of
them.
Every path starting at nodes 2, 4, 5, and 6 all lead to either node 5 or 6.
```

Example 2:

```
Input: graph = [[1,2,3,4],[1,2],[3,4],[0,4],[]]
Output: [4]
Explanation:
Only node 4 is a terminal node, and every path starting at node 4 leads to node 4.
```

Constraints:

- `n == graph.length`
- `1 <= n <= 104`
- `0 <= graph[i].length <= n`
- `0 <= graph[i][j] <= n - 1`
- `graph[i]` is sorted in a strictly increasing order.
- The graph may contain self-loops.
- The number of edges in the graph will be in the range `[1, 4 * 104]`.

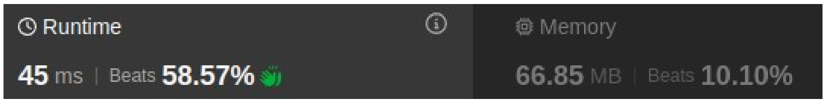
Overview

We are given a directed graph of `n` nodes with each node labeled from `0` to `n - 1`. The graph is represented by a 2D integer array `graph` where `graph[i]` is an integer array of nodes that have an incoming edge from node `i`.

The problem states that a node is a **terminal node** if there are no outgoing edges. A node is a **safe node** if every possible path starting from that node leads to a terminal node (or another safe node).

Our task is to return a sorted array of all the safe nodes of the graph.

802. Find Eventual Safe States



Runtime

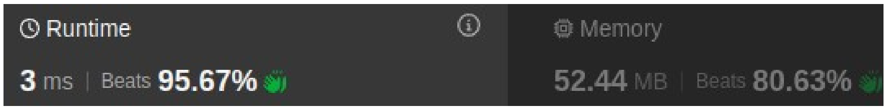
45 ms | Beats 58.57%

Memory

66.85 MB | Beats 10.10%

```
/*
    BFS-Topological sort (Kahn's algorithm)
    Time complexity: O(n+m)
    Space complexity: O(n+m)
    n: #nodes
    m: #edges

*/
typedef std::vector<int> vi;
typedef std::vector<std::vector<int>> vvi;

class Solution {
    public:
        vi eventualSafeNodes(vvi& graph){
            int n=graph.size();

            vvi g(n); // <O(n),O(n+m)>
            std::vector<int> indegree(n,0); // <O(n),O(n)>
            // <O(m),O(1)>
            for(int v=0;v<n;++v){
                for(auto& u: graph[v]){
                    g[u].push_back(v);
                    indegree[v]++;
                }
            }

            // <O(n),O(n)>
            std::queue<int> q;
            for(int node=0;node<n;++node){
                if(indegree[node]==0) q.push(node);
            }
            vi is_safe(n,0); // <O(n),O(n)>
            // <O(m),O(1)>
            while(!q.empty()){
                int node=q.front();
                q.pop();

                is_safe[node]=1;

                for(auto& u: g[node]){
                    indegree[u]--;
                    if(indegree[u]==0) q.push(u);
                }
            }

            // <O(n),O(n)>
            vi ans;
            for(int node=0;node<n;++node){
                if(is_safe[node]) ans.push_back(node);
            }

            return ans;
        }
};
```

802. Find Eventual Safe States


Runtime

3 ms | Beats 95.67%

Memory

52.44 MB | Beats 80.63%

```
/*
   DFS traversal
   Time complexity: O(n+m)
   Space complexity: O(3n)
   n: #nodes
   m: #edges
*/
typedef std::vector<int> vi;
typedef std::vector<std::vector<int>> vvi;
class Solution {
   public:
      vi eventualSafeNodes(vvi& graph){
         int n=graph.size();
         vi visited(n,0),has_back_edge(n,0);
         // Detect cycles
         // Return true, if a node u belongs to a cycle.
         // False, otherwise
         auto dfs=[&](int u,auto& self)->bool{
            visited[u]=1;
            has_back_edge[u]=1;
            for(auto& v: graph[u]){
               // Cycle found or a back edge to v exists
               if(!visited[v] && self(v,self) || has_back_edge[v]) return true;
            }
            // The ingoing edge to the current node u is not a back edge
            has_back_edge[u]=0;

            // current node u does not lead to a cycle
            return false;
         };
         for(int node=0;node<n;++node){
            if(visited[node]) continue;
            dfs(node,dfs);
         }
         vi ans;
         for(int node=0;node<n;++node){
            if(!has_back_edge[node]) ans.push_back(node);
         }return ans;}};
```