Description | Editorial | Solutions (1.7K) | Submissions

802. Find Eventual Safe States

Medium | 4.5K | 396

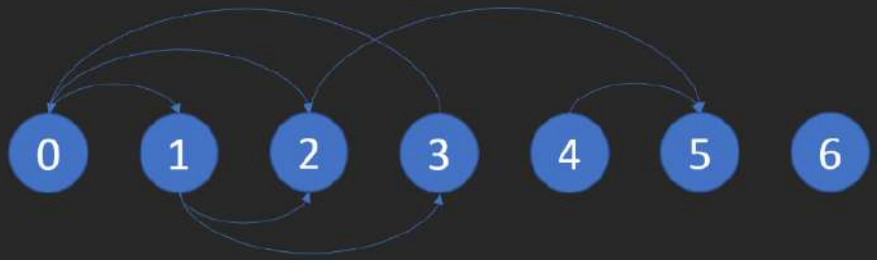
Companies

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:

Java | Auto

```java
class Solution {
    public boolean DFS(int s, List<List<Integer>> graph, boolean[] visited, boolean[] dfsVisited, boolean[] checkCycle) {
        visited[s] = dfsVisited[s] = true;
        for (int it : graph.get(s)) {
            if (!visited[it]) {
                if (DFS(it, graph, visited, dfsVisited, checkCycle))
                    return checkCycle[s] = true;
            } else if (dfsVisited[it]) {
                return checkCycle[s] = true;
            }
        }
        dfsVisited[s] = false;
        return false;
    }

    public List<Integer> eventualSafeNodes(int[][] graph) {
        int v = graph.length;
        List<List<Integer>> adjList = new ArrayList<>();
        for (int i = 0; i < v; i++) {
```

Testcase | Result

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

graph =

```
[[1,2],[2,3],[5],[0],[5],[],[]]
```

Output

```
[2,4,5,6]
```

Expected

Console | Run | Submit

Description | Editorial | Solutions (11.4K) | Submissions

242. Valid Anagram

Easy | 9.7K | 309

Companies

Given two strings `s` and `t`, return `true` *if* `t` *is an anagram of* `s`, *and* `false` *otherwise.*

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

```
Input: s = "anagram", t = "nagaram"
Output: true
```

Example 2:

```
Input: s = "rat", t = "car"
Output: false
```

Constraints:

- `1 <= s.length, t.length <= 5 * 10^4`
- `s` and `t` consist of lowercase English letters.

Follow up: What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

Accepted 2.3M | Submissions 3.7M | Acceptance Rate 63.1%

Java | Auto

```java
import java.util.Arrays;

class Solution {
    public boolean isAnagram(String s, String t) {
        char[] sChars = s.toCharArray();
        char[] tChars = t.toCharArray();

        Arrays.sort(sChars);
        Arrays.sort(tChars);

        return Arrays.equals(sChars, tChars);
    }
}
```

Testcase | Result

Accepted Runtime: 0 ms

Case 1 | Case 2

Input

s =
"anagram"

t =
"nagaram"

Output

Console | Run | Submit