<https://www.lintcode.com/problem/605/>

**Description**

Check whether the original sequence org can be **uniquely** reconstructed from the sequences in seqs. The org sequence is a **permutation** of the integers from 1 to n, with 1≤*n*≤10^4.

**Reconstruction means building a shortest common super sequence of the sequences in seqs (i.e., a shortest sequence so that all sequences in seqs are subsequences of it).** Determine whether there is only one sequence that can be reconstructed from seqs and it is the org sequence.

**Example**

Example 1:

Input:org = [1,2,3], seqs = [[1,2],[1,3]]

Output: false

Explanation:

**[1,2,3] is not the only one sequence that can be reconstructed, because [1,3,2] is also a valid sequence that can be reconstructed.**

Example 2:

Input: org = [1,2,3], seqs = [[1,2]]

Output: false

Explanation:

The reconstructed sequence can only be [1,2], can't reconstruct the sequence [1,2,3].

Example 3:

Input: org = [1,2,3], seqs = [[1,2],[1,3],[2,3]]

Output: true

Explanation:

The sequences [1,2], [1,3], and [2,3] can uniquely reconstruct the original sequence [1,2,3].

Example 4:

Input:org = [4,1,5,2,6,3], seqs = [[5,2,6,3],[4,1,5,2]]

Output:true

**Attempt 1: 2022-11-19**

**Solution 1: Topological Sort (120min)**

public class Solution {

    /\*\*

    \* @param org: a permutation of the integers from 1 to n

    \* @param seqs: a list of sequences

    \* @return: true if it can be reconstructed only one or false

    \*/

    public boolean sequenceReconstruction(int[] org, int[][] seqs) {

        // Build graph

        Map<Integer, List<Integer>> graph = buildGraph(seqs);

        // Find initial indegree based on graph

        Map<Integer, Integer> indegree = getIndegree(graph);

        // Topological sort to find if unique path can be found from seqs as same as org

        List<Integer> tmp = new ArrayList<Integer>();

        Queue<Integer> q = new LinkedList<Integer>();

        for(int node : indegree.keySet()) {

            if(indegree.get(node) == 0) {

                q.offer(node);

            }

        }

        while(!q.isEmpty()) {

            int size = q.size();

            // candidate next node must be only 1 choice, because it require unique path,

            // if candidate next node(s) stored on queue more than 1, which means more

            // than one unique path, not satisfy requirement

            if(size > 1) {

                return false;

            }

            int node = q.poll();

            tmp.add(node);

            for(int neighbour : graph.get(node)) {

                indegree.put(neighbour, indegree.get(neighbour) - 1);

                if(indegree.get(neighbour) == 0) {

                    q.offer(neighbour);

                }

            }

        }

        if(tmp.size() != org.length) {

            return false;

        }

        for(int i = 0; i < org.length; i++) {

            if(org[i] != tmp.get(i)) {

                return false;

            }

        }

        return true;

    }

    private Map<Integer, List<Integer>> buildGraph(int[][] seqs) {

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int[] seq : seqs) {

            for(int node : seq) {

                graph.put(node, new ArrayList<Integer>());

            }

        }

        // Note: seq length may > 2, that's why use for loop

        // e.g Input:org = [4,1,5,2,6,3], seqs = [[5,2,6,3],[4,1,5,2]], Output:true

        for(int[] seq : seqs) {

            for(int i = 0; i < seq.length - 1; i++) {

                graph.get(seq[i]).add(seq[i + 1]);

            }

        }

        return graph;

    }

    private Map<Integer, Integer> getIndegree(Map<Integer, List<Integer>> graph) {

        Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

        for(int node : graph.keySet()) {

            indegree.put(node, 0);

        }

        for(int node : graph.keySet()) {

            for(int neighbour : graph.get(node)) {

                indegree.put(neighbour, indegree.get(neighbour) + 1);

            }

        }

        return indegree;

    }

}

Time Complexity: O(n+m)

The overall time complexity of this basic algorithm is O(n+m). The O(n) comes from the number of times

that the while loop (and initial for loop) is executed, and the O(m) from the nested for loop.

Although there is no way to calculate how many times the inner loop will be executed on any one iteration

of the outer loop, it will only be executed once for each successor of each member, which means that the

total number of times that it will be executed is the total number of successors of all the members

-- or the total number of relations.

Space Complexity: O(n+m)

Space complexity is also O(n+m). The O(n) component comes from the predecessor count information stored

for each member, and the maximum length of the auxiliary queue. The O(m) comes from storing the successors

for each member; once again, the total number of successors is the number of relations, so O(m).

**Refer to**

<https://www.cnblogs.com/grandyang/p/6032498.html>

**这道题给了我们一个序列org，又给我们了一些子序列seqs，问这些子序列能否唯一的重建出原序列。能唯一重建的意思就是任意两个数字的顺序必须是一致的，不能说在一个子序列中1在4的后面，但是在另一个子序列中1在4的前面，这样就不是唯一的了。或者没有严格限制两个节点相互的关系，比如对于[1,2,3]是否唯一由[1,2][1,3]构成就存在限制不够的问题，因为[1,2][1,3]可以构成[1,3,2]，在于缺少了[2,3]这个2必须在3之前的限制关系，这样也就不唯一了，还有一点就是，子序列seqs中不能出现其他的数字，就是说必须都是原序列中的数字。**

<https://www.lintcode.com/problem/605/solution/17390>

最后要是唯一的topological order，有三个条件：

a）Topological Sorting所输出的数组 == org

b）seq里面的node数目要等于org里面的node数目

c）顶点到末端距离 == org的长度

**Basic solution refer to**

<https://www.lintcode.com/problem/605/solution/18253>

public class Solution {

    /\*\*

    \* @param org a permutation of the integers from 1 to n

    \* @param seqs a list of sequences

    \* @return true if it can be reconstructed only one or false

    \*/

    public boolean sequenceReconstruction(int[] org, int[][] seqs) {

        // Write your code here

        // build graph

        Map<Integer, ArrayList<Integer>> graph = getGraph(seqs);

        /\*\*

        \* we need to build graph from seqs!! cause we are using seqs to try to build org

        \*

        \*/

        // build indegree

        Map<Integer, Integer> indegree = getIndegree(graph);

        // get zero indegree, offer in queue

        Queue<Integer> queue = new LinkedList<>();

        for (int node : graph.keySet()) {

            if (indegree.get(node) == 0) {

                queue.offer(node);

            }

        }

        // we shoud have a container to compare it to org

        List<Integer> sequence = new ArrayList<>();

        // bfs

        while (!queue.isEmpty()) {

            int size = queue.size();

            if (size > 1) {

                return false;

            }

            int node = queue.poll();

            sequence.add(node);

            for (int neib : graph.get(node)) {

                indegree.put(neib, indegree.get(neib) - 1);

                if (indegree.get(neib) == 0) {

                    queue.offer(neib);

                }

            }

        }

        if (sequence.size() != org.length) { // first compare length, to prevent egde cases.

            return false;

        }

        for (int i = 0; i < org.length; i++) {

            if (sequence.get(i) != org[i]) {

                return false;

            }

        }

        return true;

    }

    private Map<Integer, ArrayList<Integer>> getGraph(int[][] seqs) {

        Map<Integer, ArrayList<Integer>> graph = new HashMap<Integer, ArrayList<Integer>>();

        for (int[] path : seqs) {

            for (int node : path) {

                graph.put(node, new ArrayList<Integer>());

            }

        }

        for (int[] path : seqs) {

            for (int i = 0; i < path.length - 1; i++) {

                graph.get(path[i]).add(path[i+1]);

            }

        }

        return graph;

    }

    private Map<Integer, Integer> getIndegree(Map<Integer, ArrayList<Integer>> graph) {

        Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

        for (int node : graph.keySet()) {

            indegree.put(node, 0);

        }

        for (int node : graph.keySet()) {

            for (int neib : graph.get(node)) {

                indegree.put(neib, indegree.get(neib) + 1);

            }

        }

        return indegree;

    }

}

**Refer to**

<https://algo.monster/liteproblems/444>

**Problem Description**

The problem presents us with an interesting challenge. We're given two things: an array nums which is a permutation of integers from 1 to n, and a 2D array sequences which contains subsequences of nums. Our goal is to check if the array nums is the unique and shortest supersequence of all the subsequences in sequences. A supersequence is a sequence that contains all other subsequences as part of it without changing the order of elements in those subsequences. The twist is that we need to determine if nums is not just any supersequence, but the shortest possible one that exists uniquely for the given subsequences.

To further understand the problem, consider that a supersequence should accommodate all the ordering requirements specified by each subsequence. If sequences contains [1,2] and [1,3] as subsequences, the supersequence should start with 1 and then have 2 and 3 in any order because 1 comes before both 2 and 3 in both subsequences. However, if we have a subsequence [1,2,3], it now dictates the order that 2 must come before 3, which gives us only one possibility for the supersequence: [1,2,3].

Understanding this, the challenge becomes checking if nums is this strictly defined shortest supersequence and that no other supersequence, which is different from nums but still meets the criteria, exists.

**Intuition**

To tackle this problem, we can draw from concepts used in graphs, specifically topological sorting. We can create a directed graph where each number in sequences represents a node, and the relationship between the numbers represents the directed edges. The crucial insight is that if nums is indeed the unique shortest supersequence, then the topological sort of this graph should yield one and only one specific order.

Here's how we approach it:

**Build a graph** - We create a directed graph where each edge from a to b shows that a precedes b in at least one of the subsequences.

**Calculate indegrees** - The indegree of a node is the number of edges coming into it. For the purpose of topological sorting, nodes with an indegree of 0 can be considered as starting points.

**Queue for processing** - Nodes with 0 indegree are added to a queue to process them one by one. At each step, we should only have one node in our queue if nums is to be the unique shortest supersequence. Having more than one node means we have multiple ways to arrange the supersequence, and hence nums wouldn't be the unique supersequence.

**Reduce indegrees** - As we process a node, we reduce the indegree of its directed neighbors by 1. If any neighbor's indegree reaches 0, it means that all its prerequisites are processed, and it can be added to the queue.

**Check queue size** - If at any point we have more than one element in the queue, it implies multiple possible sequences, and we return false.

**Final check** - If we never encounter a queue with more than one element and all nodes have been processed, then nums is the unique shortest supersequence, hence we return true.

This method allows us to determine not only if nums is a supersequence but also if it is the unique and shortest one. It's a clever application of graph theory to sequence reconstruction.

**Solution Approach**

The solution provided is a direct implementation of the topological sort algorithm used on a directed graph. The algorithm checks whether there is a unique way to reconstruct the permutation sequence given the array of subsequences. Here is a step-by-step explanation of the algorithm, mapping them to the respective parts of the code:

**1.Constructing the directed graph and calculating indegrees:**

A directed graph is built using a dictionary g where each key-value pair is a node and its list of neighboring nodes (the nodes that follow it directly in the permutation).

The indegrees are stored in indeg, an array where each index corresponds to a node and its value represents the number of edges coming into that node.

The graph and the indegrees are generated by iterating over each pair of adjacent elements (a, b) in every subsequence using pairwise(seq) and doing the following:

Append b - 1 to the adjacency list of node a - 1 in the graph (g[a - 1].append(b - 1)).

Increase the indegree of node b - 1 by 1 (indeg[b - 1] += 1).

g = defaultdict(list)

indeg = [0] \* len(nums)

for seq in sequences:

for a, b in pairwise(seq):

g[a - 1].append(b - 1)

indeg[b - 1] += 1

**2.Initializing the queue:**

A deque q is created to hold nodes with an indegree of 0, i.e., nodes that are not preceded by another node in any subsequence and hence can start the permutation.

q = deque(i for i, v in enumerate(indeg) if v == 0)

**3.Processing the nodes:**

The nodes are processed one by one from the queue. The uniqueness check for the supersequence is done here - if there's ever more than one node with indegree 0, we return False. This condition indicates that there is more than one valid sequence, which violates the problem's constraint requiring a unique shortest supersequence.

while q:

if len(q) > 1:

return False

i = q.popleft()

...

For each node taken from the queue, the algorithm decrements the indegree of all its neighbors because the current node has been processed and therefore is no longer a prerequisite for its neighbors.

for j in g[i]:

indeg[j] -= 1

if indeg[j] == 0:

q.append(j)

**4.Check if the sequence is reconstructable:**

After processing all nodes, if the sequence is reconstructable as a unique shortest supersequence, the program would have successfully popped all nodes from the queue (without finding a condition where there's more than one node that can be processed next), and therefore it returns True.

return True

This algorithm ensures that if there is more than one way to order subsequences, it will be caught during the processing stage, leading to a False result. Conversely, if nums is the unique shortest supersequence, it will satisfy this single-path condition throughout the entire process, leading to a True result.

**Example Walkthrough**

Let's illustrate the solution approach using a small example:

Consider an array nums that is a permutation of integers from 1 to 3, which is [1, 2, 3], and a 2D array sequences that contains subsequences of nums, which is [[1, 2], [1, 3]].

Now let's walk through the solution:

**1.Constructing the directed graph and calculating indegrees:**

We initialize a graph g and array indeg to store the indegrees. From the subsequences, we can see that 1 precedes both 2 and 3. In terms of indegrees, 1 has an indegree of 0, and 2 and 3 both have indegrees of 1 since they are each preceded once by 1.

The graph g will have edges from 1 to 2 and from 1 to 3.

**2.Initializing the queue:**

We initialize a queue q and add the node 1 to it since it's the only number with an indegree of 0.

**3.Processing the nodes:**

We pop 1 from the queue and decrement the indegrees of its neighbors, which are 2 and 3. After decrementing, both nodes have indegrees of 0 and are added to the queue. However, because we can only have one node in the queue for the sequence to be unique, at this point, the algorithm would detect that there is more than one node with indegree 0 and return False.

Using the above example, it's clear that the permutation nums = [1, 2, 3] is not the unique shortest supersequence for the given sequences = [[1, 2], [1, 3]] since we ended up with two nodes in the queue at the same time, indicating that there are multiple possible sequences. The function would thus conclude the permutation sequence cannot be uniquely reconstructed and return False.

**Solution Implementation**

class Solution {

public boolean sequenceReconstruction(int[] nums, List<List<Integer>> sequences) {

int n = nums.length; // number of elements in the original sequence

int[] inDegrees = new int[n]; // array to record the number of incoming edges for each vertex in the graph

List<Integer>[] graph = new List[n]; // adjacency list to represent the graph

Arrays.setAll(graph, k -> new ArrayList<>()); // initialize the adjacency list

// Convert sequences into a directed graph

for (List<Integer> seq : sequences) {

for (int i = 1; i < seq.size(); ++i) {

int from = seq.get(i - 1) - 1; // convert 1-based index to 0-based index

int to = seq.get(i) - 1; // convert 1-based index to 0-based index

graph[from].add(to); // add an edge from 'from' to 'to'

inDegrees[to]++; // increase the in-degree of the 'to' node

}

}

Deque<Integer> queue = new ArrayDeque<>(); // queue to perform topological sorting

// Find all nodes with no incoming edges and add them to the queue

for (int i = 0; i < n; ++i) {

if (inDegrees[i] == 0) {

queue.offer(i);

}

}

// Perform topological sort and check whether the original sequence is uniquely reconstructible

while (!queue.isEmpty()) {

if (queue.size() > 1) {

// More than one node with no incoming edges means there can be more than one sequence

return false;

}

int node = queue.poll(); // remove the next node from the queue

// Iterate over all the neighbors of the current node

for (int neighbor : graph[node]) {

if (--inDegrees[neighbor] == 0) { // if the in-degree becomes 0, add the neighbor to the queue

queue.offer(neighbor);

}

}

}

// If the graph has been fully traversed and a unique sequence is determined, return true

return true;

}

}

**Time and Space Complexity**

The time complexity of the given code is O(V + E), where V is the number of vertices (numbers) in nums and E is the total number of edges (order relationships) in sequences. This is because building the graph (adjacency list) requires traversing each pair in sequences, and then using BFS to traverse through the constructed graph only once.

The space complexity is also O(V + E), which stems from the space required to store the graph g and indegree array indeg. The adjacency list g stores at most E edges, whereas the indegree array indeg has space for V vertices. The queue q in the BFS procedure will also require at most V space in the worst case (when all vertices have zero indegree at the same time).

**Solution 2:  Topological Sort with comparison on the fly (10min)**

public class Solution {

    /\*\*

    \* @param org: a permutation of the integers from 1 to n

    \* @param seqs: a list of sequences

    \* @return: true if it can be reconstructed only one or false

    \*/

    public boolean sequenceReconstruction(int[] org, int[][] seqs) {

        // Build graph

        Map<Integer, List<Integer>> graph = buildGraph(seqs);

        // Find initial indegree based on graph

        Map<Integer, Integer> indegree = getIndegree(graph);

        // Topological sort to find if unique path can be found from seqs as same as org

        List<Integer> tmp = new ArrayList<Integer>();

        Queue<Integer> q = new LinkedList<Integer>();

        for(int node : indegree.keySet()) {

            if(indegree.get(node) == 0) {

                q.offer(node);

            }

        }

        // Add variable 'level'(used in 'org' as index) to indicate if unique path

        // is identical to 'org' during build process or not

        int level = 0;

        while(!q.isEmpty()) {

            if(level >= org.length) {

                return false;

            }

            int size = q.size();

            // candidate next node must be only 1 choice, because it require unique path,

            // if candidate next node(s) stored on queue more than 1, which means more

            // than one unique path, not satisfy requirement

            if(size > 1) {

                return false;

            }

            int node = q.poll();

            // If current node on unique path is different than org[level] (level equal to

            // index start with 0 usage), no path avaiale

            if(node != org[level]) {

                return false;

            }

            tmp.add(node);

            level++;

            for(int neighbour : graph.get(node)) {

                indegree.put(neighbour, indegree.get(neighbour) - 1);

                if(indegree.get(neighbour) == 0) {

                    q.offer(neighbour);

                }

            }

        }

        // After all the unique path length(indicate by 'level') have to equal to 'org',

        // otherwise even all nodes on unique path are identical to 'org', it may miss

        // later section in 'org'

        if(level != org.length) {

            return false;

        }

        return true;

    }

    private Map<Integer, List<Integer>> buildGraph(int[][] seqs) {

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int[] seq : seqs) {

            for(int node : seq) {

                graph.put(node, new ArrayList<Integer>());

            }

        }

        // Note: seq length may > 2, that's why use for loop

        // e.g Input:org = [4,1,5,2,6,3], seqs = [[5,2,6,3],[4,1,5,2]], Output:true

        for(int[] seq : seqs) {

            for(int i = 0; i < seq.length - 1; i++) {

                graph.get(seq[i]).add(seq[i + 1]);

            }

        }

        return graph;

    }

    private Map<Integer, Integer> getIndegree(Map<Integer, List<Integer>> graph) {

        Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

        for(int node : graph.keySet()) {

            indegree.put(node, 0);

        }

        for(int node : graph.keySet()) {

            for(int neighbour : graph.get(node)) {

                indegree.put(neighbour, indegree.get(neighbour) + 1);

            }

        }

        return indegree;

    }

}

Time Complexity: O(n+m)

The overall time complexity of this basic algorithm is O(n+m). The O(n) comes from the number of times that

the while loop (and initial for loop) is executed, and the O(m) from the nested for loop.

Although there is no way to calculate how many times the inner loop will be executed on any one iteration

of the outer loop, it will only be executed once for each successor of each member, which means that the

total number of times that it will be executed is the total number of successors of all the members

-- or the total number of relations.

Space Complexity: O(n+m)

Space complexity is also O(n+m). The O(n) component comes from the predecessor count information stored

for each member, and the maximum length of the auxiliary queue. The O(m) comes from storing the successors

for each member; once again, the total number of successors is the number of relations, so O(m).

**More efficient solution refer to**

<https://www.lintcode.com/problem/605/solution/16613>

**和九章答案有一些细节的不同，一边build map一边检查，貌似更高效一些**

public class Solution {

    public boolean sequenceReconstruction(int[] org, int[][] seqs) {

        // write your code here

        Map<Integer, List<Integer>> graph = new HashMap<>();

        Map<Integer, Integer> indegree = new HashMap<>();

        // build up the map

        for (int[] seq : seqs) {

            for (int i = 0; i < seq.length; i++) {

                graph.putIfAbsent(seq[i], new ArrayList<Integer>());

                indegree.putIfAbsent(seq[i], 0);

                if (i > 0) {

                    graph.get(seq[i - 1]).add(seq[i]);

                    indegree.put(seq[i], indegree.get(seq[i]) + 1);

                }

            }

        }

        if (org.length != indegree.size()) {

            return false;

        }

        // classical BFS

        Queue<Integer> queue = new LinkedList<>();

        for (int key : indegree.keySet()) {

            if (indegree.get(key) == 0) {

                queue.add(key);

            }

        }

        int index = 0;

        while (queue.size() == 1) {

            int cur = queue.poll();

            if (org[index++] != cur) {

                return false;

            }

            for (int neighbor : graph.get(cur)) {

                indegree.put(neighbor, indegree.get(neighbor) - 1);

                if (indegree.get(neighbor) == 0) {

                    queue.add(neighbor);

                }

            }

        }

        return index == org.length;

    }

}