## Sequence Reconstruction

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 $1 \leq n \leq 10^4$. Reconstruction means building a shortest common supersequence 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 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].

### 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

## Solution 1

```
public class Solution {
    public boolean sequenceReconstruction(int[] org, int[][] seqs) {
        Map<Integer, Set<Integer>> map = new HashMap<>();
        Map<Integer, Integer> indegree = new HashMap<>();

        for(int[] seq: seqs) {
            if(seq.length == 1) {
                if(!map.containsKey(seq[0])) {
                    map.put(seq[0], new HashSet<>());
                    indegree.put(seq[0], 0);
                }
            } else {
                for(int i = 0; i < seq.length - 1; i++) {
                    if(!map.containsKey(seq[i])) {
                        map.put(seq[i], new HashSet<>());
                        indegree.put(seq[i], 0);
                    }

                    if(!map.containsKey(seq[i + 1])) {
                        map.put(seq[i + 1], new HashSet<>());
                        indegree.put(seq[i + 1], 0);
                    }

                    if(map.get(seq[i]).add(seq[i + 1])) {
                        indegree.put(seq[i + 1], indegree.get(seq[i + 1]) + 1);
                    }
                }
            }
        }

        Queue<Integer> queue = new LinkedList<>();
        for(Map.Entry<Integer, Integer> entry: indegree.entrySet()) {
            if(entry.getValue() == 0) queue.offer(entry.getKey());
        }

        int index = 0;
        while(!queue.isEmpty()) {
            int size = queue.size();
            if(size > 1) return false;
            int curr = queue.poll();
            if(index == org.length || curr != org[index++]) return false;
            for(int next: map.get(curr)) {
                indegree.put(next, indegree.get(next) - 1);
                if(indegree.get(next) == 0) queue.offer(next);
            }
        }
        return index == org.length && index == map.size();
    }
}
```

written by xietao0221 original link here

## Solution 2

For `org` to be uniquely reconstructible from `seqs` we need to satisfy 2 conditions:

1. Every sequence in `seqs` should be a subsequence in `org`. This part is obvious.
2. Every 2 consecutive elements in `org` should be consecutive elements in some sequence from `seqs`. Why is that? Well, suppose condition 1 is satisfied. Then for 2 any consecutive elements `x` and `y` in `org` we have 2 options.
    - We have both `x` and `y` in some sequence from `seqs`. Then (as condition 1 is satisfied) they must be consequtive elements in this sequence.
    - There is no sequence in `seqs` that contains both `x` and `y`. In this case we cannot uniquely reconstruct `org` from `seqs` as sequence with `x` and `y` switched would also be a valid original sequence for `seqs`.

So this are 2 necessary criterions. It is pretty easy to see that this are also sufficient criterions for `org` to be uniquely reconstructible (there is only 1 way to reconstruct sequence when we know that condition 2 is satisfied).

To implement this idea I have `idxs` hash that maps item to its index in `org` sequence to check condition 1. And I have `pairs` set that holds all consequitive element pairs for sequences from `seqs` to check condition 2 (I also consider first elements to be paired with previous `undefined` elements, it is necessary to check this).

```
var sequenceReconstruction = function(org, seqs) {
    const pairs = {};
    const idxs = {};

    for (let i = 0; i < org.length; i++)
        idxs[org[i]] = i;

    for (let j = 0; j < seqs.length; j++) {
        const s = seqs[j];
        for (let i = 0; i < s.length; i++) {
            if (idxs[s[i]] == null)
                return false;
            if (i > 0 && idxs[s[i - 1]] >= idxs[s[i]])
                return false;
            pairs[`${s[i - 1]}_${s[i]}`] = 1;
        }
    }

    for (let i = 0; i < org.length; i++)
        if (pairs[`${org[i - 1]}_${org[i]}`] == null)
            return false;

    return true;
};
```

written by dettier original link here

## Solution 3

The basic idea is to count how many numbers are smaller(self include) than the current number.
We then compare this count to the org.
It is pretty like the idea of count sort.

```
public class Solution {
    public boolean sequenceReconstruction(int[] org, int[][] seqs) {
        int len = org.length;
        int[] map = new int[len + 1];//map number to its index
        Arrays.fill(map, -1);
        int[] memo = new int[org.length];//count how many numbers are smaller(on
the right)
        for (int i = 0; i < len; i++) {
            map[org[i]] = i;
        }
        for (int[] seq : seqs) {
            if (seq.length == 0) continue;
            int prev = seq[0];
            if (prev <= 0 || prev > len || map[prev] == -1) return false;
            for (int i = 1; i < seq.length; i++) {
                int curr = seq[i];
                if (curr <= 0 || curr > len || map[curr] == -1) return false;
                memo[map[prev]] = Math.max(memo[map[prev]], len - map[curr] + 1);
                prev = curr;
            }
            memo[map[prev]] = Math.max(memo[map[prev]], 1);
        }
        for (int i = 0; i < memo.length; i++) {
            if (memo[i] != len - i) return false;
        }
        return true;
    }
}
```

written by jianhao2 original link here