supersequence, which is different from nums but still meets the criteria, exists.

unique shortest supersequence, hence we return true.

that follow it directly in the permutation).

edges coming into that node.

Problem Description

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

The problem presents us with an interesting challenge. We're given two things: an array nums which is a permutation of integers from

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

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

1. 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. 2. 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.

and only one specific order.

Here's how we approach it:

- 3. 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.
- 4. 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.
- return false. 6. Final check - If we never encounter a queue with more than one element and all nodes have been processed, then nums is the

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

- 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.
- 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

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

The solution provided is a direct implementation of the topological sort algorithm used on a directed graph. The algorithm checks

• 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:

g = defaultdict(list)

Solution Approach

indeq = [0] * len(nums)for seq in sequences: for a, b in pairwise(seq): g[a - 1].append(b - 1)indeg[b - 1] += 1

■ 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).

```
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.
   1 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.

• 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.

Example Walkthrough

Now let's walk through the solution:

1 for j in g[i]:

1 while q:

if len(q) > 1:

i = q.popleft()

indeg[j] = 1

if indeg[j] == 0:

q.append(j)

4. Check if the sequence is reconstructable:

throughout the entire process, leading to a True result.

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

1. Constructing the directed graph and calculating indegrees:

subsequences of nums, which is [[1, 2], [1, 3]].

return False

processed next), and therefore it returns True. 1 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

• 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

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.

• 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

• 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

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

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

sequence to be unique, at this point, the algorithm would detect that there is more than one node with indegree 0 and return False.

Python Solution

from collections import defaultdict, deque

if len(queue) > 1:

return False

node = queue.popleft()

Go through all the adjacent nodes

if in_degree[adjacent] == 0:

queue.append(adjacent)

return all(degree == 0 for degree in in_degree)

in_degree[adjacent] -= 1

for (List<Integer> seq : sequences) {

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

while (!queue.isEmpty()) {

if (inDegrees[i] == 0) {

queue.offer(i);

if (queue.size() > 1) {

for (int neighbor : graph[node]) {

queue.offer(neighbor);

return false;

int n = originalSeq.size();

vector<vector<int>> graph(n, vector<int>());

12 // Initialize the inDegree array and adjacency list graph from sequences

function buildGraph(seqList: number[][], nodeCount: number): void {

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

for adjacent in graph[node]:

from itertools import pairwise

False.

12

13

14

15

16

17

18

19

20

21

24

25

26

27

28

29

30

31

32

33

34

35

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

3. Processing the nodes:

class Solution: def sequenceReconstruction(self, originalSequence: List[int], sequences: List[List[int]]) -> bool: # Create a graph and an in-degree list initialized with zeros for all elements graph = defaultdict(list) in_degree = [0] * len(originalSequence)

Decrement in-degree and if it becomes zero, add to the queue

Return True if the original sequence can be uniquely reconstructed

- # Build the graph and in-degree list using input sequences for sequence in sequences: for prev, current in pairwise(sequence): graph[prev - 1].append(current - 1) # Nodes are 0-based in the graph in_degree[current - 1] += 1 # Increment in-degree for the current node # Queue for nodes with zero in-degree queue = deque(node for node, degree in enumerate(in_degree) if degree == 0) # Process the graph using the queue while queue: # If there's more than one node with in-degree zero, the original sequence is not unique
- Java Solution 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 6 // Convert sequences into a directed graph

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

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

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

// Function to determine if a sequence can be reconstructed to match the original sequence

bool sequenceReconstruction(vector<int>& originalSeq, vector<vector<int>>& sequences) {

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

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

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

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

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

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

// Iterate over all the neighbors of the current node

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

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

C++ Solution

1 class Solution {

2 public:

return true;

```
vector<int> inDegree(n, 0);
  8
             // Construct the graph and count incoming degrees
  9
             for (const auto& seq : sequences) {
 10
                 for (int i = 1; i < seq.size(); ++i) {</pre>
 11
 12
                     int from = seq[i - 1] - 1; // Convert to 0-based index
                     int to = seq[i] - 1; // Convert to 0-based index
 13
 14
                     graph[from].push_back(to);
 15
                     ++inDegree[to];
 16
 17
 18
 19
             // Initialize a queue with all nodes that have an in-degree of 0
 20
             queue<int> nodesWithNoIncomingEdges;
 21
             for (int i = 0; i < n; ++i) {
 22
                 if (inDegree[i] == 0) {
 23
                     nodesWithNoIncomingEdges.push(i);
 24
 25
 26
 27
             // Process the nodes
 28
             while (!nodesWithNoIncomingEdges.empty()) {
                 // If there is more than one node with no incoming edges, return false
 29
 30
                 // Because we should be able to determine the order unambiguously
 31
                 if (nodesWithNoIncomingEdges.size() > 1) return false;
 32
 33
                 int currentNode = nodesWithNoIncomingEdges.front();
 34
                 nodesWithNoIncomingEdges.pop();
 35
 36
                 // Decrease the in-degree of neighboring nodes and add to queue if in-degree becomes 0
 37
                 for (int neighbor : graph[currentNode]) {
 38
                     if (--inDegree[neighbor] == 0) {
 39
                         nodesWithNoIncomingEdges.push(neighbor);
 40
 41
 42
 43
 44
             // Return true if we can reconstruct the sequence uniquely
 45
             return true;
 46
 47
    };
 48
Typescript Solution
  1 // TypeScript does not directly support typing for a 2D array with specific inner array lengths, so we use number[][] to represent
  2 // Represents the original sequence to be reconstructed
    let originalSeq: number[] = [];
    // Represents the sequences of numbers which are subsequences of the original sequence
    let sequences: number[][] = [];
    // Construct a directed graph and in-degree array from the sequences
    let graph: number[][];
    let inDegree: number[];
 11
```

33 34 35 36

graph = new Array(nodeCount);

for (const seq of seqList) {

let from = seq[i - 1] - 1;

graph[i] = [];

inDegree = new Array(nodeCount).fill(0);

for (let i = 1; i < seq.length; i++) {</pre>

for (let i = 0; i < nodeCount; i++) {</pre>

14

15

16

18

19

20

21

22

23

```
let to = seq[i] - 1;
 24
           graph[from].push(to);
 25
           inDegree[to]++;
 26
 27
 28
 29
 30
     // Function to determine if a sequence can be reconstructed to match the original sequence
     function sequenceReconstruction(originalSeq: number[], sequences: number[][]): boolean {
       const n: number = originalSeq.length;
       buildGraph(sequences, n); // Build the graph and inDegree arrays for the given sequences
       const nodesWithNoIncomingEdges: number[] = [];
 37
 38
       // Find all nodes with no incoming edges
 39
       for (let i = 0; i < n; i++) {
        if (inDegree[i] === 0) {
 40
           nodesWithNoIncomingEdges.push(i);
 41
 42
 43
 44
 45
       // Process the nodes while there are nodes with no incoming edges
 46
       let index = 0;
 47
       while (nodesWithNoIncomingEdges.length > 0) {
        // If there is more than one node with no incoming edges,
 48
 49
        // the sequence cannot be uniquely reconstructed
 50
         if (nodesWithNoIncomingEdges.length > 1) return false;
 51
 52
         // The next node should match the next element in the original sequence
 53
         let currentNode = nodesWithNoIncomingEdges.shift()!;
 54
         if (originalSeq[index++] !== currentNode + 1) return false;
 55
         // Decrease the in-degree of neighboring nodes
         // If the in-degree becomes 0, add to the queue
         graph[currentNode].forEach(neighbor => {
           if (--inDegree[neighbor] === 0) {
             nodesWithNoIncomingEdges.push(neighbor);
         });
      // The sequence can be reconstructed if all the nodes in the original sequence are processed
      return index === n;
    // Example usage:
     // originalSeq = [1, 2, 3, 4];
     // sequences = [[1, 2], [2, 3], [3, 4]];
    // console.log(sequenceReconstruction(originalSeq, sequences)); // Should output true
 73
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
```

57 58 59 60

56 61 62 63 64 68

sequences, and then using BFS to traverse through the constructed graph only once. The space complexity is also 0(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).

edges (order relationships) in sequences. This is because building the graph (adjacency list) requires traversing each pair in