1940. Longest Common Subsequence Between Sorted Arrays

Medium Array Hash Table Counting

Problem Description

The problem provides an array of integer arrays, where each inner array is already sorted in strictly increasing order. The task is to find the longest common subsequence present in all these arrays.

A subsequence is a sequence that can be obtained from another sequence by deleting zero or more elements without changing the order of the remaining elements. Since we are interested in the common subsequence among all the arrays, our goal is to identify numbers that are present in every single array provided.

The output should be an integer array listing the elements of the longest common subsequence in any order.

Intuition

The intuition behind the solution centers on frequency analysis of each element across all the arrays. Since all the arrays are sorted in strictly increasing order, each number will appear in a subsequent at most once in every array. To find common elements, we need to count the occurrences of each number and check if the count matches the number of arrays. If a particular number appears as many times as there are arrays, we can conclude that this number is a part of the common subsequence for all arrays.

Here is the thought process for arriving at the solution:

1. Create a frequency map that holds the count of each element across all arrays.
2. Iterate over each array and for each element, increase its count in the map by one.
3. After processing all arrays, iterate through the map entries to check which elements have a count equal to the total number of arrays.
4. If an element's count equals the number of arrays, it is a common element and thus a part of the longest common subsequence.
5. Collect these common elements into a result list and return it.

Solution Approach

The implementation of the solution uses the following steps and components:

1. **HashMap for Frequency Counting**: A `HashMap<Integer, Integer>` named `counter` is used to keep track of the frequency of each integer. The keys in the map are the integers from the arrays, and the values are the counts representing how many arrays that integer appears in.
2. **Iterate Over Arrays**: The first `for` loop iterates over the given array of arrays. Each inner array is accessed in sequence.
3. **Increment Counts**: The nested `for` loop iterates over each integer in the current inner array. For every integer `e`, the code uses the `counter` map to increment the count associated with `e`. If the element is not already in the map, it is added with a default count of 0 using `getOrDefault()` before being incremented.

```
for (int e : array) {
    counter.put(e, counter.getOrDefault(e, 0) + 1);
}
```

4. **Check for Common Integers**:
   - The variable `n` holds the number of arrays, which is essential to determine if an integer is common to all arrays.
   - After counting the occurrences, the code iterates over the entries in the `counter` map.
5. **Collect Common Integers**: If an entry in `counter` has a value equal to `n` (number of arrays), it means the key (integer) is present in all arrays. Such keys are added to the `res` list.

```
for (Map.Entry<Integer, Integer> entry : counter.entrySet()) {
    if (entry.getValue() == n) {
        res.add(entry.getKey());
    }
}
```

6. **Return the Result**: Finally, the list `res` is returned, which contains all the integers that form the longest common subsequence between all arrays.

This approach harnesses the efficiency of hash maps to quickly access and update counts, avoiding quadratic runtime complexity that might arise from comparing each element of every array with each other. By leveraging the uniqueness of elements within each sorted array and the simple condition of an element must appear in each array exactly once to be included in the common subsequence, this algorithm achieves the desired results in an optimal manner.

Example Walkthrough

Consider the following three arrays representing the input, with each inner array sorted in strictly increasing order:

```
array1: [1, 3, 4]
array2: [1, 2, 3, 4, 5]
array3: [1, 3, 5, 7, 9]
```

The goal is to find the longest common subsequence among these arrays. Following the solution approach:

1. **HashMap for Frequency Counting**: We initialize an empty `HashMap<Integer, Integer>` named `counter`.
2. **Iterate Over Arrays**: We start by iterating over each given array. We will process `array1`, `array2`, and `array3` in turn.
3. **Increment Counts**:
   - When we process `array1`, the `counter` map will be updated as follows: `counter` = {1:1, 3:1, 4:1}
   - Next, processing `array2` updates the map to: `counter` = {1:2, 2:1, 3:2, 4:2, 5:1}
   - Finally, processing `array3` yields `counter` = {1:3, 2:1, 3:3, 4:2, 5:2, 7:1, 9:1}
4. **Check for Common Integers**: The `n` variable holds the number of arrays, which in this case is 3. We check each entry in the `counter` map to see if it has a value equal to 3.
5. **Collect Common Integers**: As we iterate through the map:
   - The integer 1 has a count of 3, so it is added to the `res` list.
   - The integer 3 has a count of 3, so it is also added to the `res` list.
   - No other integers meet the condition of being present in all arrays.
6. **Return the Result**: We return the list `res`, which now contains [1, 3]. These are the integers present in all the provided arrays, forming the longest common subsequence. The order of elements in the result is irrelevant.

Therefore, by following this approach, we efficiently find `[1, 3]` as the longest common subsequence between the given arrays using a `HashMap` to account for the frequency and simple iteration techniques.

Solution Implementation

Python

```
def longest_common_subsequence(arrays):
    """
    Finds the longest common subsequence present in each of the sub-arrays
    of the provided 2D array.
    :param arrays: A 2D list where each sub-list is to be checked for common elements.
    :return: A list containing the longest common subsequence found in all sub-arrays.
    """
    # Initialize a dictionary to keep track of the frequency of each element across all sub-arrays.
    element_count_map = {}

    # Initialize a list to store the result (longest common subsequence).
    result_sequence = []

    # total arrays represents the total number of sub-arrays in the given 2D list.
    total_arrays = len(arrays)

    # Initially assume that all elements are possible candidates for the LCS (Longest Common Subsequence).
    potential_lcs = set(arrays[0])

    # Iterate over each sub-array.
    for sub_array in arrays[1:]:
        # Update potential lcs by intersecting with the set of elements in the current sub-array.
        # This keeps only those elements that are candidates for LCS from all sub-arrays seen so far.
        potential_lcs &= set(sub_array)

    # Go through each element in the potential LCS set.
    for element in potential_lcs:
        # Count how many times the element appears in each sub-array and compare against the total.
        count = sum(element in sub for sub in arrays)

        # If the count is equal to the total number of sub-arrays, add it to the result.
        if count == total_arrays:
            result_sequence.append(element)

    # Return the list containing the longest common subsequence across all sub-arrays.
    return result_sequence
```

Java

```
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

class Solution {
    // Finds the common elements that appear in all the subarrays
    public List<Integer> longestCommonSubsequence(int[][] arrays) {
        // Using a map to count occurrences of each integeer in all arrays
        Map<Integer, Integer> frequencyCounter = new HashMap<>();

        // Iterate through all the arrays
        for (int[] array : arrays) {
            // Iterate through each element of the current array
            for (int element : array) {
                // Increment the count for the element in the map
                frequencyCounter.put(element, frequencyCounter.getOrDefault(element, 0) + 1);
            }
        }

        // The length of the arrays or number of arrays
        int numberOfArrays = arrays.length;

        // List to store the common elements found in all the arrays
        List<Integer> result = new ArrayList<>();

        // Iterate through the map entries
        for (Map.Entry<Integer, Integer> entry : frequencyCounter.entrySet()) {
            // If the count of the element is equal to the number of arrays
            // then add it to the result list as it appears in all arrays
            if (entry.getValue() == numberOfArrays) {
                result.add(entry.getKey());
            }
        }

        // Return the list of common elements
        return result;
    }
}
```

C++

```
#include <vector>
#include <unordered_map>

class Solution {
public:
    // Function to find the longest common subsequence among all arrays.
    std::vector<int> longestCommonSubsequence(std::vector<std::vector<int>>& arrays) {
        // This map will count the frequency of each element across all arrays.
        std::unordered_map<int, int> elementFrequency;
        // This vector will store the results - the longest common subsequence
        std::vector<int> result;
        // Total number of arrays
        int numberOfArrays = arrays.size();

        // Iterate through each array in the collection of arrays
        for (auto &array : arrays) {
            // Using a std::unordered_set to ensure each element in the same array is counted once
            std::unordered_set<int> uniqueElements;

            // Iterate through each element of the current array
            for (int element : array) {
                // If this is the first occurrence of this element in this array
                if (uniqueElements.insert(element).second) {
                    // Increment the counter for this element by one
                    elementFrequency[element]++;
                }
            }

            // If the current element's count matches the number of arrays
            // it means this element is present in all arrays
            for (int element : uniqueElements) {
                if (elementFrequency[element] == numberOfArrays) {
                    // Add the element to the result vector
                    result.push_back(element);
                }
            }
        }

        // Before returning, ensure that the results vector is sorted
        // Since the order in the result does not matter by the prompt,
        // but a sorted array might be a common expectation.
        std::sort(result.begin(), result.end());

        // Return the final vector containing the longest common subsequence
        return result;
    }
};
```

TypeScript

```
/**
 * Finds the longest common subsequence present in each of the sub-arrays of the provided 2D array.
 * @param {number[][]} arrays - A 2D array where each sub-array is to be checked for common elements.
 * @return {number[]} - An array containing the longest common subsequence found in all sub-arrays.
 */
function longestCommonSubsequence(arrays: number[][]): number[] {
    // Initialize a map to keep track of the frequency of each element across all sub-arrays.
    const elementCountMap = new Map<number, number>();

    // Initialize an array to store the result (longest common subsequence).
    const resultSequence: number[] = [];

    // totalArrays represents the total number of sub-arrays in the given 2D array.
    const totalArrays: number = arrays.length;

    // Iterate over each sub-array.
    for (let i = 0; i < totalArrays; i++) {
        // Iterate over each element in the sub-array.
        for (let j = 0; j < arrays[i].length; j++) {
            // Get the current element.
            const element = arrays[i][j];

            // Update the frequency count of the current element in the map.
            elementCountMap.set(element, (elementCountMap.get(element) || 0) + 1);

            // If the current element's frequency matches the total number of sub-arrays,
            // include it in the resultSequence as it's common in all sub-arrays.
            if (elementCountMap.get(element) === totalArrays) {
                resultSequence.push(element);
            }
        }
    }

    // Return the array that contains the longest common subsequence across all sub-arrays.
    return resultSequence;
}
```

```
def longest_common_subsequence(arrays):
    """
    Finds the longest common subsequence present in each of the sub-arrays
    of the provided 2D array.
    :param arrays: A 2D list where each sub-list is to be checked for common elements.
    :return: A list containing the longest common subsequence found in all sub-arrays.
    """
    # Initialize a dictionary to keep track of the frequency of each element across all sub-arrays.
    element_count_map = {}

    # Initialize a list to store the result (longest common subsequence).
    result_sequence = []

    # total arrays represents the total number of sub-arrays in the given 2D list.
    total_arrays = len(arrays)

    # Initially assume that all elements are possible candidates for the LCS (Longest Common Subsequence).
    potential_lcs = set(arrays[0])

    # Iterate over each sub-array.
    for sub_array in arrays[1:]:
        # Update potential lcs by intersecting with the set of elements in the current sub-array.
        # This keeps only those elements that are candidates for LCS from all sub-arrays seen so far.
        potential_lcs &= set(sub_array)

    # Go through each element in the potential LCS set.
    for element in potential_lcs:
        # Count how many times the element appears in each sub-array and compare against the total.
        count = sum(element in sub for sub in arrays)

        # If the count is equal to the total number of sub-arrays, add it to the result.
        if count == total_arrays:
            result_sequence.append(element)

    # Return the list containing the longest common subsequence across all sub-arrays.
    return result_sequence
```

Time and Space Complexity

Time Complexity

The time complexity of the code mainly consists of two parts:

1. The first loop, which iterates over each array in `arrays` and then over each element `e` in these arrays, resulting in a time complexity of `O(N * M)`, where `N` is the number of arrays, and `M` is the average length of each array.
2. The second loop, which iterates over entries in the `counter` hashmap. The size of the hashmap is at most equal to the number of unique elements across all arrays. Let's denote this as `U`. This results in `O(U)` time complexity for the second loop.

Overall, assuming `U <= N * M`, the time complexity is `O(N * M)`, as this is the dominating factor.

Space Complexity

The space complexity is determined by the space required to store the `counter` hashmap, which holds as many as `U` unique elements and their counts across all arrays. Thus, the space complexity is `O(U)`. In the worst case where all elements are unique across all arrays, `U` would be equal to `N * M`, leading to a worst-case space complexity of `O(N * M)`.

Additionally, space is used for the resultant `res` ArrayList, but since the size of `res` is at most `U`, it does not exceed the space complexity determined by `counter`.