960. Delete Columns to Make Sorted III

Hard | Array | String | Dynamic Programming | Leetcode Link

Problem Description

The problem provides an array of `n` strings, `strs`, with each string being of the same length. The goal is to determine the fewest number of index deletions required so that, after the deletions, the remaining letters within each string form a non-decreasing sequence in lexicographic (alphabetical) order.

To visualize, imagine each string as a row in a grid, with the columns aligned vertically. Deleting an index corresponds to removing a column from this grid. The challenge is to delete the fewest columns necessary so that the letters in each row of the grid strictly increase or stay the same as you move left to right.

The result of the process is a set of indices that, if deleted, would satisfy the condition for every string in the array to be lexicographically ordered from the first to the last character. The task is to return the minimum size of such a set of deletion indices.

Intuition

The approach to this problem involves dynamic programming. Dynamic programming is a strategy for solving problems by breaking them down into simpler subproblems and storing the solutions to these subproblems to avoid redundant work.

Here's the intuition behind the solution:

1. The problem resembles the classic problem of finding the longest increasing subsequence (LIS), but inverted. Instead of finding the longest subsequence, we're trying to find the minimum number of deletions, which correlates to finding the maximum length of an "ordered" subsequence that doesn't require deletion.

2. We initialize a dynamic programming array `dp` where `dp[i]` represents the length of the longest ordered subsequence ending with the `i`th character (inclusive).

3. We iterate over every pair `i` and `j`, where `i > j`. If the `j`th character is less than or equal to the `i`th character for all strings, it means we can extend the ordered subsequence ending at `j` to include `i`. We update `dp[i]` if this subsequence is longer than the current one.

4. The maximum value in the `dp` array represents the length of the longest possible ordered subsequence that we can obtain without any deletions. Therefore, we subtract this value from the total number of columns `n` to get the minimum number of deletions required.

This solution essentially finds the longest subsequence of columns that are already in non-decreasing order for all rows and removes the rest, minimizing the number of deletions required while achieving the goal.

Solution Approach

The implementation of the solution utilizes dynamic programming, which is evident from the use of the `dp` array where each `dp[i]` keeps track of the length of the longest non-decreasing subsequence of characters ending with the `i`th column. Let's walk through the approach step by step:

1. Define `n` to be the length of the strings, which is also the number of columns if each string is considered a row in a grid.

2. Initialize the `dp` array of size `n` with all elements set to `1`. Each `dp[i]` represents the length of the longest non-decreasing subsequence of columns when considering columns `0` to `i`. Initially, we set them all to `1` since each column by itself can be a subsequence.

3. Use two nested loops with indices `i` and `j`, where `i` ranges from `1` to `n-1` and `j` ranges from `0` to `i-1`. These loops are used to find the length of the longest non-decreasing subsequence ending at each column `i`.

4. For each pair of `i` and `j`, check if the `j`th column is less than or equal to the `i`th column for all strings (`all(s[j] <= s[i] for s in strs)`). This ensures that including the character at column `i` after `j` maintains the non-decreasing order.

5. If the condition is true, update `dp[i]` to the maximum of its current value or `dp[j] + 1`. In other words, extend the length of the ordered subsequence ending at `j` by one to include `i`.

6. After filling the `dp` array, find the length of the longest non-decreasing subsequence by finding the maximum value in `dp` (`max(dp)`).

7. Since we need to find the minimum number of deletions, subtract the length of the longest non-decreasing subsequence from the total number of columns, `n - max(dp)`. This gives us the minimum columns that need to be deleted to ensure all strings are in lexicographic order.

The principle algorithms and patterns used in this solution include:

- **Dynamic Programming**: Through the `dp` array to store intermediate results and avoid redundant computations.
- **Nested Loops**: To compare every possible pair of columns.
- **Greedy Choice**: At each step, choosing the longest subsequence ending at `j` that can be extended by `i`.

By incorporating dynamic programming, the solution effectively leverages previously computed results to build upon and find the longest subsequence, reducing the complexity compared to naive approaches that might check every possible combination of deletions.

Example Walkthrough

Let's use a small example to illustrate the solution approach with the given strings array `strs = ["cba", "daf", "ghi"]`.

1. Since each string has three characters, we have `n = 3`.

2. Initialize the `dp` array with `n` elements to `[1, 1, 1]` because each character alone is a non-decreasing subsequence.

3. We start the nested loop iteration.

    - For `i = 1`, we compare with `j = 0`.
        - Compare every string's character at index `j` with index `i`.
        - For "cba", "daf", and "ghi", we compare pairs ("c", "b"), ("d", "a"), ("g", "h").
        - Since "c" > "b", "d" > "a", "g" > "h", the condition isn't met and we don't update `dp[1]`.
    - Move to `i = 2`, again, compare with `j = 0` and then `j = 1`.
        - Comparing index `0` with `2`, for "cba" ("c", "a"), "daf" ("d", "f"), and "ghi" ("g", "i"), it's not non-decreasing for all sequences.
        - Now compare index `1` with `2`, for "cba" ("b", "a"), "daf" ("a", "f"), and "ghi" ("h", "i"). Only in "ghi" is "h" <= "i". Since not all strings fulfill the condition, `dp[2]` remains unchanged.
4. At this point, our `dp` array is still `[1, 1, 1]` as no increasing subsequences were found that include more than one column.

5. The length of the longest non-decreasing subsequence is simply the max value in `dp`, which is `1`.

6. To find the minimum number of deletions, `n - max(dp)`, we subtract the longest subsequence's length (1) from the total number of columns (3).

Therefore, `3 - 1 = 2` is the minimum number of deletions required. For this example:

- Deleting the first and second columns ("c" from "cba", "d" from "daf", "g" from "ghi" and "b" from "cba", "a" from "daf", "h" from "ghi") leaves us with ["a", "f", "i"], which are in non-decreasing order for each string.

By following the dynamic programming approach, we efficiently found the minimum number of columns to delete without exhaustively checking every combination. This example illustrates the steps and logic of the solution approach clearly.

Python Solution

```python
from typing import List

class Solution:
    def minDeletionSize(self, strs: List[str]) -> int:
        # Calculate the length of the strings (assuming all strings have the same length)
        str_length = len(strs[0])

        # Initialize the dynamic programming (DP) array where dp[i] represents the
        # length of the longest subsequence that ends with the i-th character.
        dp = [1] * str_length

        # Iterate over each character in the strings starting from the second character
        for i in range(1, str_length):
            # Compare the current character with all characters before it
            for j in range(i):
                # Check if the current character is greater than or equal to all
                # corresponding characters in previous columns
                if all(s[j] <= s[i] for s in strs):
                    # Update the DP array by considering the length of the subsequence
                    # ending at the j-th character plus the current character
                    dp[i] = max(dp[i], dp[j] + 1)

        # Calculate the minimum number of columns to delete by subtracting the length
        # of the longest increasing subsequence from the total number of columns
        return str_length - max(dp)

# The above code finds the length of the longest subsequence of characters that is
# increasing across all strings. Then, it subtracts this length from the total number
# of columns (characters in a string) to find the minimum number of deletions required.

```

Java Solution

```java
class Solution {
    public int minDeletionSize(String[] strs) {
        int numColumns = strs[0].length(); // Length of the strings, representing the number of columns.
        int[] longestIncreasingSubsequence = new int[numColumns]; // DP array to store the length of longest increasing subsequences
        Arrays.fill(longestIncreasingSubsequence, 1); // Initially, each sequence is of length 1 (the character itself).
        int maxSequenceLength = 1; // Initialize the maximum subsequence length to 1.

        // Iterate over all pairs of columns.
        for (int currentIndex = 1; currentIndex < numColumns; ++currentIndex) {
            for (int previousIndex = 0; previousIndex < currentIndex; ++previousIndex) {
                // If the current column can follow the previous column in the increasing subsequence...
                if (isNonDecreasingAcrossStrings(previousIndex, currentIndex, strs)) {
                    // Update the longest subsequence for the current column if it's greater after adding the current column.
                    longestIncreasingSubsequence[currentIndex] = Math.max(
                            longestIncreasingSubsequence[currentIndex],
                            longestIncreasingSubsequence[previousIndex] + 1
                    );
                }
            }
            // Update maximum subsequence length found so far.
            maxSequenceLength = Math.max(maxSequenceLength, longestIncreasingSubsequence[currentIndex]);
        }

        // The minimum number of deletions is the total columns minus the length of the longest increasing subsequence.
        return numColumns - maxSequenceLength;
    }

    // Helper method to check if all strings have a non-decreasing order from column 'prevIndex' to 'currentIndex'.
    private boolean isNonDecreasingAcrossStrings(int prevIndex, int currentIndex, String[] strs) {
        for (String str : strs) {
            if (str.charAt(currentIndex) < str.charAt(prevIndex)) {
                return false; // If any string has a decreasing pair, return false.
            }
        }
        return true; // All strings have non-decreasing order for this column index pair.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

class Solution {
public:
    // This method returns the minimum number of columns that need to be deleted
    // so that the remaining columns are in non-decreasing sorted order
    int minDeletionSize(vector<string>& strs) {
        // Get the number of columns in the strings
        int columnCount = strs[0].size();

        // Create a dynamic programming table where dp[i] represents the length
        // of the longest non-decreasing subsequence that ends with column i
        vector<int> dp(columnCount, 1);

        // Store the maximum length of non-decreasing subsequence found
        int maxLength = 1;

        // Iterate over each column to find the longest non-decreasing subsequence
        for (int current = 1; current < columnCount; ++current) {
            for (int previous = 0; previous < current; ++previous) {

                // If the current column is in non-decreasing order compared to the previous column
                if (isNonDecreasing(current, previous, strs)) {
                    // Update dp[current] with the maximum length sequence found thus far
                    dp[current] = max(dp[current], dp[previous] + 1);
                }
            }

            // Update the maximum length
            maxLength = max(maxLength, dp[current]);
        }

        // The number of columns to delete is the total number minus the length of the longest sequence
        return columnCount - maxLength;
    }

    // This helper method checks if column 'current' is in non-decreasing order compared to column 'previous'
    bool isNonDecreasing(int current, int previous, vector<string>& strs) {
        // Iterate over each row
        for (string& s : strs) {
            // If any corresponding pair of characters in current and previous columns is in decreasing order, return false
            if (s[current] < s[previous])
                return false;
        }

        // Return true if all pairs are in non-decreasing order
        return true;
    }
};

```

Typescript Solution

```typescript
function minDeletionSize(strs: string[]): number {
    // Get the number of columns in the strings
    const columnCount = strs[0].length;

    // Create a dynamic programming array where dp[i] represents the length
    // of the longest non-decreasing subsequence that ends with column i
    const dp: number[] = new Array(columnCount).fill(1);

    // Store the maximum length of non-decreasing subsequence found
    let maxLength = 1;

    // Iterate over each column to find the longest non-decreasing subsequence
    for (let current = 1; current < columnCount; ++current) {
        for (let previous = 0; previous < current; ++previous) {

            // If the current column is in non-decreasing order compared to the previous column
            if (isNonDecreasing(current, previous, strs)) {
                // Update dp[current] with the maximum length sequence found thus far
                dp[current] = Math.max(dp[current], dp[previous] + 1);
            }
        }

        // Update the maximum length
        maxLength = Math.max(maxLength, dp[current]);
    }

    // The number of columns to delete is the total number of columns minus the length of the longest sequence
    return columnCount - maxLength;
}

// This helper function checks if column 'current' is in non-decreasing order compared to column 'previous'
function isNonDecreasing(current: number, previous: number, strs: string[]): boolean {
    // Iterate over each row
    for (let s of strs) {
        // If any corresponding pair of characters in current and previous columns is in decreasing order, return false
        if (s[current] < s[previous]) {
            return false;
        }
    }

    // Return true if all pairs are in non-decreasing order
    return true;
}

```

Time and Space Complexity

The provided code performs a dynamic programming approach to find the minimum number of columns that need to be deleted from a list of equal-length strings to ensure that the remaining columns are lexicographically sorted. Here's an analysis of its complexities:

Time Complexity

The time complexity of the code is O(`n^2 * m`), where `n` is the length of each string in `strs` and `m` is the number of strings. This is because the code uses a nested loop structure where the outer loop runs `n` times (the number of columns), and the inner loop also runs up to `n` times for each iteration of the outer loop. Inside the inner loop, there is a comparison that runs `m` times for checking if every string maintains the lexicographic order for the pair of columns being considered. Multiplying all these together gives O(`n^2 * m`) as the time complexity.

Space Complexity

The space complexity of the function is O(`n`), which is due to the dynamic programming array `dp` of size `n`, where each element represents the length of the longest subsequence of sorted columns including the current column as the last sorted column. Additional space usage is constant and doesn't scale with input size (`n` or `m`), hence the overall space complexity is O(`n`).