955. Delete Columns to Make Sorted II

Medium Greedy Array String [Leetcode Link]

Problem Description

In this problem, we are given an array `strs` containing `n` strings, where each string is of the same length. Our task involves deleting characters at certain indices across all the strings with the aim of making the resulting array of strings sorted in lexicographic (alphabetical) order. Lexicographic order means that the strings should appear as if they were sorted in a dictionary, i.e., `strs[0] <= strs[1] <= strs[2] <= ... <= strs[n - 1]`.

We can select any number of indices to delete, and these deletions will occur on every string in the array. Our goal is to find the minimum number of indices that we need to delete to achieve the lexicographically sorted array. The problem asks us to return the smallest possible size of the set of deletion indices.

Intuition

To solve this problem, the intuition is to iteratively check each character column (index) across all strings and decide whether that column should be deleted or kept to maintain the lexicographic order. If a column is found where a string appears before another string lexicographically but has a greater character at the current index, then this column breaks the required order and must be deleted. Otherwise, if the current column does not break the order, we may keep it.

However, simply looking at the current column in isolation is not sufficient. We must also remember if any prior columns have already established a strict lexicographic order between any two adjacent strings. If that's the case, these strings do not impact the decision for the current column because they are already ordered properly due to previous columns. We keep track of these decisions using the `cut` boolean array, which marks pairs of strings that are already sorted and do not need to be considered again.

The solution follows these steps:

1. Iterate over each column (index) of the strings.
2. Check if the character at the current column for each string maintains the lexicographic order with the next string.
3. If the order is violated, increase the deletion count and skip to the next column.
4. If the order is maintained, mark any string pairs that are now sorted due to this column.
5. Continue until all the indices have been processed.
6. Return the number of deletions that were necessary to sort the array lexicographically.

Solution Approach

The implementation of the solution uses a simple but effective approach to decide which columns (indices) need to be deleted to ensure the strings are in lexicographic order.

Here is how the approach is implemented:

1. **Initiation**: We begin by initializing the necessary variables:
   - `len` stores the length of the array `A`.
   - `wordLen` stores the length of each string within the array.
   - `res` initialized to 0 will hold the count of indices required to be deleted.
   - `cut` is a boolean array, initialized to false, that keeps track of which pairs of strings do not need further comparisons because they are already in correct order according to prior columns.
2. **Column-Wise Check**: We inspect each column using the outer loop which iterates over `j`, the index for the character position within the strings.
3. **Row-Wise Comparison**: For each column, a nested loop goes through each string and compares it with the string that comes after it (`i` and `i+1`). The main conditions checked in this loop are as follows:
   - If the `cut[i]` is false (meaning the strings at index `i` and `i+1` have not been marked as already sorted) and the character at the column `j` for string `i` is greater than the character at the same column for string `i+1`, it implies that the current column violates the lexicographic order and thus we must "cut" this column by incrementing `res` by 1 and continue to the next column.
   - If no lexicographic violation is detected, no increment to `res` will happen.
4. **Mark Sorted Pairs**: If a column does not lead to an increment of `res`, then for each string, excluding the last one, we check if the character at the current column is less than the character in the same column of the next string. If it is, this means these two strings are already sorted with respect to each other for the current column, hence we set `cut[i]` to true.
5. **Result**: After inspecting all columns, the value `res`, which is the count of the indices that needed to be deleted to sort the strings lexicographically, is returned as the final answer.
6. **Algorithm Complexity**: This approach would have a time complexity of O(N * W) where `N` is the number of strings and `W` is the width or length of each string. Space complexity is O(N) due to the additional array `cut` used to keep track of sorted pairs.

This algorithm uses a greedy approach, attempting at each step to make a local optimal decision (deleting a column if necessary), which leads to a globally optimal solution—the minimum number of columns deleted to achieve the lexicographically sorted array of strings.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following array of strings, where `n = 4`.

```
strs = ["cba", "daf", "ghi"]
```

Here our goal is to ensure this array is sorted lexicographically with the minimum number of column deletions.

Step 1: We initialize our variables. `len = 3`, `wordLen = 3`, `res = 0`, and `cut` is an array of false values of size `len - 1` which is 2 in this case, so `cut = [false, false]`.

Step 2 & 3: We start checking each column, starting with the first column (index 0).

- Comparing the first characters `c`, `d`, and `g` of each string, we see that they are already in the lexicographic order, so we don't increment `res`. Additionally, we don't mark any pairs as sorted because `c < d` and `d < g`, indicating that the order cannot be affected by subsequent characters.

Step 4: Move to the second column (index 1).

- Comparing the second characters `b`, `a`, `h` of each string, we notice the lexicographic order is violated (`a` should not come after `b`). Since `cut[0]` is `false` and 'a' < 'b', we need to delete this column to maintain the order. We increment `res` by 1 and we do not need to check further values in this column, so we continue to the next column.

Step 5: Inspect the third column (index 2).

- Comparing the third characters `a`, `f`, `i` of each string, they are in correct lexicographic order, and no further action is needed.

Step 6: Having inspected all columns, we get `res = 1` (since we had to delete the second column). This is the minimum number of deletions required to sort the strings lexicographically.

Thus, our function would return 1 for this example.

Python Solution

```python
class Solution:
    def min_deletion_size(self, strings):
        # Check if the input list is None or has only one string; if so, no deletion is needed.
        if strings is None or len(strings) <= 1:
            return 0

        # Initialize the number of strings and the length of the first string.
        num_of_strings = len(strings)
        string_length = len(strings[0])
        deletions = 0

        # Initialize a list to keep track of which strings are already sorted.
        sorted_status = [False] * num_of_strings

        # Iterate over each column by index.
        for j in range(string_length):
            # Attempt to update sorted status for this column.
            for i in range(num_of_strings - 1):
                # If the current string is not sorted with the next, and the current character
                # is greater than the next string's character, we need to delete this column.
                if not sorted_status[i] and strings[i][j] > strings[i + 1][j]:
                    # Increment the deletion counter.
                    deletions += 1
                    break  # Skip to the next column without updating the sorted status.

            else: # This else belongs to the for, it is executed if the loop is not 'break'-ed.
                # Update sorted status if this column does not need to be deleted.
                for i in range(num_of_strings - 1):
                    # If the characters are in ascending order, mark as sorted.
                    if strings[i][j] < strings[i + 1][j]:
                        sorted_status[i] = True

        # Return the total number of columns that need to be deleted.
        return deletions

```

Java Solution

```java
class Solution {
    public int minDeletionSize(String[] strings) {
        // Check if input array is null or has only one string, if so no deletion needed.
        if (strings == null || strings.length <= 1) {
            return 0;
        }

        // Initialize the length variables and the result counter.
        int numOfStrings = strings.length;
        int stringLength = strings[0].length();
        int deletions = 0;

        // Boolean array to keep track of sorted strings.
        boolean[] sorted = new boolean[numOfStrings];

        // Iterate through each column.
        for (int j = 0; j < stringLength; j++) {
            // Inner loop to compare characters in the current column.
            for (int i = 0; i < numOfStrings - 1; i++) {
                // If the current and the next string are not sorted and the current character is greater
                // than the next, we need to delete this column.
                if (!sorted[i] && strings[i].charAt(j) > strings[i + 1].charAt(j)) {
                    deletions += 1;
                    continue; // Skip to the next column without updating the sorted array.
                }
            }

            // Update the sorted array for characters that are already sorted.
            for (int i = 0; i < numOfStrings - 1; i++) {
                if (strings[i].charAt(j) < strings[i + 1].charAt(j)) {
                    sorted[i] = true;
                }
            }
        }
        // Return the total number of deletions.
        return deletions;
    }
}

```

C++ Solution

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

class Solution {
public:
    int minDeletionSize(std::vector<std::string>& strings) {
        // Check if input vector is empty or has only one string, if so no deletion is needed.
        if (strings.empty() || strings.size() <= 1) {
            return 0;
        }

        // Initialize the length variables and the result counter.
        int numOfStrings = strings.size();
        int stringLength = strings[0].size();
        int deletions = 0;

        // Vector to keep track of sorted strings.
        std::vector<bool> sorted(numOfStrings, false);

        // Iterate through each column.
        for (int j = 0; j < stringLength; ++j) {
            // Inner loop to compare characters in the current column.
            for (int i = 0; i < numOfStrings - 1; ++i) {
                // If the current and the next string are not sorted and the current character is greater
                // than the character in the next string, we need to delete this column.
                if (!sorted[i] && strings[i][j] > strings[i + 1][j]) {
                    deletions++; // Increment the deletion count
                    break; // Skip to the next column without updating the sorted vector.
                }
            }

            // Update the sorted vector for strings that are already sorted in this column.
            for (int i = 0; i < numOfStrings - 1; ++i) {
                if (strings[i][j] < strings[i + 1][j]) {
                    sorted[i] = true; // Mark as sorted
                }
            }
        }
        // Return the total number of deletions.
        return deletions;
    }
};

```

Typescript Solution

```typescript
// Initialize a variable to hold the minimum number of deletions.
let minDeletions: number = 0;

// Function to calculate the minimum number of deletions required to make each column non-decreasing.
function minDeletionSize(strings: string[]): number {
    // Check if input array is null or has only one string; if so, no deletion needed.
    if (!strings || strings.length <= 1) {
        return 0;
    }

    // Initialize the length variables.
    const numOfStrings: number = strings.length;
    const stringLength: number = strings[0].length;

    // Boolean array to keep track of which strings are sorted.
    const sorted: boolean[] = new Array(numOfStrings).fill(false);

    // Iterate through each column.
    for (let j = 0; j < stringLength; j++) {
        // Reset the deletion flag for the current column.
        let needToDeleteColumn: boolean = false;

        // Compare characters in the current column.
        for (let i = 0; i < numOfStrings - 1; i++) {
            // If current and next strings are not sorted, and current char is greater than next,
            // mark the column for deletion.
            if (!sorted[i] && strings[i].charAt(j) > strings[i + 1].charAt(j)) {
                needToDeleteColumn = true;
                break; // Break out of the loop since we've decided to delete this column.
            }
        }

        // If we need to delete the column, increment the deletion count and continue to the next column.
        if (needToDeleteColumn) {
            minDeletions++;
            continue;
        }

        // Update the sorted array for rows that are sorted with the current column considered.
        for (let i = 0; i < numOfStrings - 1; i++) {
            if (strings[i].charAt(j) < strings[i + 1].charAt(j)) {
                // Mark this string as sorted up to the current column.
                sorted[i] = true;
            }
        }
    }

    // Return the total number of deletions required.
    return minDeletions;
}

```

Time and Space Complexity

The time complexity of the given code can be analyzed based on the nested for-loops. The outer loop runs for `j` from 0 to `wordLen`, and for each iteration of `j`, the inner loop runs for `i` from 0 to `len - 1`. There's also another nested loop with the same range for updating the `cut` array. Therefore, each character is visited once in the check and once in the update step for each inner loop, which leads to a total of `O(wordLen * len)` operations where `wordLen` is the length of the strings and `len` is the total number of strings.

The space complexity of the code is mostly dependent on the additional boolean array `cut` used to store the information on which strings do not need to be compared further. The `cut` array has a size equivalent to the number of strings `len`, so the space complexity is `O(len)`.

Overall, the time complexity of the algorithm is `O(wordLen * len)`, and the space complexity is `O(len)`.