944. Delete Columns to Make Sorted

Easy Array String

Leetcode Link

Problem Description

The problem presents us with an array `strs` that contains `n` strings, and all strings have the same length. We can think of these strings as rows in a grid where each column is made up of characters from the same position in each string.

For example:

```
abc
bce
cae
```

If we visualize the strings in this way, our task is to find and remove the columns that are not sorted in ascending lexicographical order. A column is considered sorted if each character is the same or comes before the character in the row below it.

In the given example, the second column contains "b", "c", and "a". Because "b" does not come before "a", this column is not sorted, so it needs to be deleted. The problem asks us to return the count of such columns that need to be removed.

Intuition

The intuition behind the solution is to simulate the column-wise traversal mentioned in the problem description. We need to iterate over each column and check if its elements are in lexicographical order. This can be done by comparing each character in the column with the character above it, starting with the second row and continuing until the bottom of the column.

If, at any point, we find a character that is less than the character in the row above, we can conclude that this column is not sorted. Each such instance implies that we must delete the column to meet the requirement of having all columns sorted.

The challenge is to count the number of columns that violate this sorted order. We use a variable `ans` to keep track of this count. By iterating through each element in the grid column by column, and checking the required condition with the elements in the rows, we can increment the `ans` variable when an unsorted instance is detected. Finally, returning `ans` gives us the total number of columns that would be deleted.

Solution Approach

The implementation of the solution follows a straightforward approach that aligns closely with the intuition previously described. The main algorithmic pattern used here is a nested loop to iterate through the grid columns and rows.

Here's a step-by-step breakdown of the code:

1. First, determine the dimensions of the grid. The variable `m` is set to the length of each string, which represents the number of columns. The variable `n` is the number of strings, which corresponds to the number of rows.
2. Initialize a counter `ans` to zero. This counter will keep track of the number of columns to delete.
3. Set up a loop to iterate column by column over the grid. The outer loop variable `j` goes from `0` to `m-1`, representing the index of the current column.
4. For each column, use another loop to compare each character with the character above it (row by row). The inner loop variable `i` goes from `1` to `n-1`, as we start from the second string and compare it with the first one above.
5. Within the inner loop, compare the character in the current row and column (`strs[i][j]`) with the character in the previous row and same column (`strs[i - 1][j]`). If `strs[i][j]` is found to be smaller, it means the column is not in lexicographical order.
6. In case an unsorted column is detected, increment the `ans` by one. Since the column is not sorted, it's marked for deletion, and there's no need to check other characters in the same column. Hence, a `break` statement terminates the inner loop early.
7. After both loops complete, return `ans`, which now contains the count of columns to be deleted.

It's worth noting that no additional data structures are required for this solution, and it operates in-place, using the given `strs` array as the basis for the comparison. The complexity of this solution is O(m*n), where `m` is the number of columns, and `n` is the number of rows since every character in the grid is visited only once during these nested iterations.

Example Walkthrough

Let's use the example given in the problem description to illustrate each step of the solution approach:

Given the strings in array `strs`:

```
abc
bce
cae
```

We analyze the columns to find the ones that are not sorted lexicographically.

1. We initialize `m = 3` (since each string has 3 characters, indicating 3 columns) and `n = 3` (since there are 3 strings, indicating 3 rows).
2. We set `ans = 0` as our counter for unsorted columns.
3. We start the outer loop with `j = 0`, to begin evaluating the first column ("abb").
4. The inner loop starts with `i = 1`, comparing the first character of the second string 'b' with the first character of the first string 'a'. Since 'b' comes after 'a', we continue to `i = 2`, the third string.
5. We compare the first character of the third string 'c' with the first character of the second string 'b'. Still in correct order, we complete the inner loop without incrementing `ans`.
6. We move to the second column with `j = 1`. We now check the characters "b", "c", and "a".
7. Comparing 'c' (second string) with 'b' (first string) is fine. However, when comparing 'a' (third string) with 'c' (second string), we find 'a' comes before 'c' lexicographically, which is not sorted.
8. As this column is unsorted, we increment `ans` to 1 and break the inner loop early since there's no need to check further.
9. We then move to the third column with `j = 2`, where we have the characters "c","e","e".
10. As we progress through the column, we find that each character is in the correct order or the same as the one above it ("c" < "e", "e" = "e").
11. With no increase in `ans`, we've completed the checks for all columns.

After finishing the loop for all columns, we find that only one column was unsorted. Hence, `ans = 1`.

Returning `ans` gives us the total number of columns that would be deleted: 1.

Python Solution

```python
# This class contains a method to find the minimum number of columns that need to be deleted
# to ensure that each remaining column is in non-decreasing sorted order.
class Solution:
    def minDeletionSize(self, strs: List[str]) -> int:
        # Get the dimensions of the list of strings,
        # 'm' represents the length of the string (number of columns),
        # 'n' represents the number of strings (number of rows).
        num_columns = len(strs[0])
        num_rows = len(strs)

        # Initialize a counter for the number of columns to delete.
        deletions = 0

        # Iterate over each column.
        for col in range(num_columns):
            # Iterate over each row starting from the second one,
            # to compare with the previous row.
            for row in range(1, num_rows):
                # If the current element is smaller than the previous element in the same column,
                # this column is not sorted and needs to be deleted.
                if strs[row][col] < strs[row - 1][col]:
                    # Increment the deletion counter and exit the inner loop
                    # as we don't need to check further in this column.
                    deletions += 1
                    break
        # Return the total number of columns that need to be deleted.
        return deletions

# Note: The placeholder `List` needs to be imported from the typing module for type annotations,
# if it's not already present at the beginning of the code.
# Add this line at the beginning of the code if it's missing:
# from typing import List
```

Java Solution

```java
class Solution {
    // Method to find the minimum number of columns to be deleted so that each row is in non-decreasing order
    public int minDeletionSize(String[] strs) {
        // m represents the length of the first string, assuming all strings are the same length.
        int columnLength = strs[0].length();
        // n represents the number of strings in the array.
        int rowLength = strs.length;
        // Initialize the counter for the minimum number of columns to delete.
        int minDeletions = 0;

        // Iterate over each column
        for (int columnIndex = 0; columnIndex < columnLength; ++columnIndex) {
            // Check each row in the current column, starting from the second row
            for (int rowIndex = 1; rowIndex < rowLength; ++rowIndex) {
                // Compare the current character with the one in the previous row.
                if (strs[rowIndex].charAt(columnIndex) < strs[rowIndex - 1].charAt(columnIndex)) {
                    // If the current character is smaller, increment the count of columns to delete
                    ++minDeletions;
                    // No need to check further in this column; break out of the inner loop.
                    break;
                }
            }
        }

        // Return the count of columns to delete so that all rows are sorted non-decreasingly
        return minDeletions;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find minimum number of columns to delete to make each row lexicographically ordered
    int minDeletionSize(vector<string>& strs) {
        int rowCount = strs.size();     // num of rows in the input vector
        int colCount = strs[0].size();  // num of columns (length of first string)
        int deleteCount = 0;            // counter for the number of columns to delete

        // Loop through each column
        for (int col = 0; col < colCount; ++col) {
            // Compare elements in the same column of adjacent rows
            for (int row = 0; row < rowCount - 1; ++row) {
                // If the current character is greater than the one in the next row, column is unsorted
                if (strs[row][col] > strs[row + 1][col]) {
                    deleteCount++;          // Increment the delete counter as this column needs to be deleted
                    break;                  // No need to check further in this column, move to the next one
                }
            }
        }
        // Return the number of columns that needs to be deleted
        return deleteCount;
    }
};
```

Typescript Solution

```typescript
// Define the type to represent an array of strings
type StringsArray = string[];

// Function to find the minimum number of columns to delete to make each row lexicographically ordered
function minDeletionSize(strs: StringsArray): number {
    const rowCount: number = strs.length;       // Number of rows in the input array
    const colCount: number = strs[0].length;    // Number of columns (length of the first string)
    let deleteCount: number = 0;                // Counter for the number of columns to delete

    // Loop through each column
    for (let col = 0; col < colCount; ++col) {
        // Compare elements in the same column of adjacent rows
        for (let row = 0; row < rowCount - 1; ++row) {
            // If the current character is greater than the one in the next row, the column is unsorted
            if (strs[row].charAt(col) > strs[row + 1].charAt(col)) {
                deleteCount++;              // Increment the delete counter as this column needs to be deleted
                break;                      // No need to check further in this column, move to the next one
            }
        }
    }
    // Return the number of columns that need to be deleted
    return deleteCount;
}
```

Time and Space Complexity

The given Python code takes in a list of strings (`strs`) and determines the minimum number of columns that must be deleted such that each remaining column is in non-decreasing sorted order.

Time Complexity

The time complexity of the code is calculated by analyzing the nested loops. The outer loop iterates over each column, of which there are `m`, where `m` is the length of each string (the number of columns). The inner loop iterates over the rows for each column, of which there are `n-1` comparisons to make (since there are `n` strings), where `n` is the number of strings (the number of rows).

Therefore, in the worst case, the code will make `m * (n - 1)` comparisons, which simplifies to `O(m * n)`. This is the worst-case scenario because the inner loop may terminate early if the current column does not need to be deleted (i.e., it is already sorted).

Space Complexity

The space complexity of the code is `O(1)` because the algorithm uses a constant amount of extra space. The variables `ans`, `m`, and `n` are simple integer counters regardless of the input size. There is no additional data structure that grows with the input size, meaning that the space needed does not scale with the size of the input (`strs`).