960. Delete Columns to Make Sorted III

String **Dynamic Programming** Hard

Problem Description

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

The problem provides an array of n strings, strs, with each string being of the same length. The goal is to determine the fewest

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.

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:

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

We initialize a dynamic programming array dp where dp[i] represents the length of the longest ordered subsequence ending

The maximum value in the dp array represents the length of the longest possible ordered subsequence that we can obtain

with the ith character (inclusive). We iterate over every pair i and j, where i > j. If the j th character is less than or equal to the ith character for all strings,

maximum length of an "ordered" subsequence that doesn't require deletion.

removes the rest, minimizing the number of deletions required while achieving the goal.

- 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.
- 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

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 ith column. Let's

Initialize the dp array of size n with all elements set to 1. Each dp[i] represents the length of the longest non-decreasing

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

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.

subsequence of columns when considering columns 0 to 1. Initially, we set them all to 1 since each column by itself can be a subsequence.

(max(dp)).

walk through the approach step by step:

For each pair of i and j, check if the jth column is less than or equal to the ith 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. 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

used to find the length of the longest non-decreasing subsequence ending at each column i.

• **<u>Dynamic Programming</u>**: Through the **dp** array to store intermediate results and avoid redundant computations.

- the ordered subsequence ending at j by one to include i. After filling the dp array, find the length of the longest non-decreasing subsequence by finding the maximum value in dp
- 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.

Since we need to find the minimum number of deletions, subtract the length of the longest non-decreasing subsequence

• **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"].

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

Compare every string's character at index j with index i.

Move to i = 2, again, compare with j = 0 and then j = 1.

■ For "cba", "daf", and "ghi", we compare pairs ("c", "b"), ("d", "a"), ("g", "h").

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

■ Since "c" > "b", "d" > "a", "g" > "h", the condition isn't met and we don't update dp[1].

We start the nested loop iteration.

 \circ For i = 1, we compare with j = 0.

Python

Java

C++

#include <vector>

#include <string>

#include <algorithm>

class Solution {

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

The principle algorithms and patterns used in this solution include:

■ Comparing index Ø 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

At this point, our dp array is still [1, 1, 1] as no increasing subsequences were found that include more than one column.

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

with ["a", "f", "i"], which are in non-decreasing order for each string.

def minDeletionSize(self, strs: List[str]) -> int:

str_length = len(strs[0])

for i in range(1. str length):

return str_length - max(dp)

public int minDeletionSize(String[] strs) {

// Iterate over all pairs of columns.

);

for (String str : strs) {

 $dp = [1] * str_length$

the condition, dp [2] remains unchanged.

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).

exhaustively checking every combination. This example illustrates the steps and logic of the solution approach clearly.

• 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

By following the dynamic programming approach, we efficiently found the minimum number of columns to delete without

- Solution Implementation
- from typing import List class Solution:
 - 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[i] <= s[i] for s in strs):</pre> # Update the DP array by considering the length of the subsequence

Initialize the dynamic programming (DP) array where dp[i] represents the

length of the longest subsequence that ends with the i-th character.

Compare the current character with all characters before it

dp[i] = max(dp[i], dp[j] + 1)

Calculate the length of the strings (assuming all strings have the same length)

Iterate over each character in the strings starting from the second character

ending at the j-th character plus the current character

Calculate the minimum number of columns to delete by subtracting the length

int maxSequenceLength = 1; // Initialize the maximum subsequence length to 1.

longestIncreasingSubsequence[currentIndex] = Math.max(

for (int previousIndex = 0; previousIndex < currentIndex; ++previousIndex) {</pre>

longestIncreasingSubsequence[currentIndex],

if (isNonDecreasingAcrossStrings(previousIndex, currentIndex, strs)) {

longestIncreasingSubsequence[previousIndex] + 1

private boolean isNonDecreasingAcrossStrings(int prevIndex, int currentIndex, String[] strs) {

return false; // If any string has a decreasing pair, return false.

return true; // All strings have non-decreasing order for this column index pair.

for (int currentIndex = 1; currentIndex < numColumns; ++currentIndex) {</pre>

of the longest increasing subsequence from the total number of columns

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.

Arrays.fill(longestIncreasingSubsequence, 1); // Initially, each sequence is of length 1 (the character itself).

// If the current column can follow the previous column in the increasing subsequence...

maxSequenceLength = Math.max(maxSequenceLength, longestIncreasingSubsequence[currentIndex]);

// Helper method to check if all strings have a non-decreasing order from column 'prevIndex' to 'currentIndex'.

int[] longestIncreasingSubsequence = new int[numColumns]; // DP array to store the length of longest increasing subsequences

// Update the longest subsequence for the current column if it's greater after adding the current column.

int numColumns = strs[0].length(); // Length of the strings, representing the number of columns.

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

// Update maximum subsequence length found so far.

if (str.charAt(currentIndex) < str.charAt(prevIndex)) {</pre>

for (int current = 1; current < columnCount; ++current) {</pre>

// Update the maximum length

if (s[current] < s[previous])</pre>

return columnCount - maxLength;

// Iterate over each row

for (string& s : strs) {

maxLength = max(maxLength, dp[current]);

for (int previous = 0; previous < current; ++previous) {</pre>

if (isNonDecreasing(current, previous, strs)) {

bool isNonDecreasing(int current, int previous, vector<string>& strs) {

dp[current] = max(dp[current], dp[previous] + 1);

```
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
```

// If the current column is in non-decreasing order compared to the previous column

// Update dp[current] with the maximum length sequence found thus far

// The number of columns to delete is the total number minus the length of the longest sequence

// This helper method checks if column 'current' is in non-decreasing order compared to column 'previous'

```
return false;
        // Return true if all pairs are in non-decreasing order
        return true;
};
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) {</pre>
        for (let previous = 0; previous < current; ++previous) {</pre>
            // 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

// If any corresponding pair of characters in current and previous columns is in decreasing order, return false

// 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 {

Calculate the length of the strings (assuming all strings have the same length)

Check if the current character is greater than or equal to all

Update the DP array by considering the length of the subsequence

corresponding characters in previous columns

if all(s[i] <= s[i] for s in strs):</pre>

Initialize the dynamic programming (DP) array where dp[i] represents the

// If any corresponding pair of characters in current and previous columns is in decreasing order, return false

```
# 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
```

str_length = len(strs[0])

for j in range(i):

return columnCount - maxLength;

if (s[current] < s[previous]) {</pre>

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

def minDeletionSize(self, strs: List[str]) -> int:

// Iterate over each row

return false;

for (let s of strs) {

return true;

class Solution:

Time Complexity

Space Complexity

from typing import List

```
# 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.
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:
```

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.

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).