712. Minimum ASCII Delete Sum for Two Strings

Medium String Dynamic Programming Leetcode Link

Problem Description

This problem asks for the lowest ASCII sum of deleted characters to make two given strings, `s1` and `s2`, equal. In other words, we need to find out how many and which characters we should delete (from either or both strings) so that the remaining characters of `s1` and `s2` will be the same, and the sum of the ASCII values of these deleted characters is as small as possible. We only need the sum, not the specific characters to delete.

Intuition

The solution to this problem involves dynamic programming, a method for solving complex problems by breaking them down into simpler subproblems. We can visualize the problem as a grid where one string (`s1`) forms the rows and the other string (`s2`) forms the columns. Each cell in the grid represents the cost (lowest ASCII sum of deleted characters) to make the substrings equal up to that point.

Here's the intuition broken down into steps:

1. **Subproblem Definition**: Define `f[i][j]` as the minimum cost to make the first `i` characters of `s1` and the first `j` characters of `s2` equal.
2. **Initial Conditions**: Start by filling in the base cases. If one string is empty (`i` or `j` is 0), then `f[i][j]` is the sum of the ASCII values of all the characters in the other string up to that point.
3. **Iterative Solution**: For each character pair (from `s1[i]` and `s2[j]`), we check if the characters are equal. If they are, the cost does not increase and it's the same as the cost for `f[i-1][j-1]`. If they aren't, we choose the cheaper option between deleting a character from `s1` or `s2`. This means we add the ASCII value of the character we consider deleting to the corresponding cost, and we take the minimum of both options as our new cost for `f[i][j]`.
4. **Build Up Solution**: By filling out the grid following these rules, starting from `f[0][0]` and going to `f[m][n]` where `m` and `n` are the lengths of `s1` and `s2` respectively, we build up to the solution.
5. **Final Answer**: After the grid is entirely filled, the answer to the problem will be the value at `f[m][n]`, as it represents the cost of making both entire strings `s1` and `s2` equal.

By constructing this table, we systematically consider the cost of each possible operation, ensuring that we arrive at the minimum total cost for the entire strings.

Solution Approach

The implementation of this solution utilizes dynamic programming, which involves storing and reusing solutions to subproblems in order to build up to the solution for the whole problem. The specific data structure used here to implement dynamic programming is a 2-dimensional list (a list of lists), which acts as a table to store the minimum cost for each subproblem.

The algorithm proceeds as follows:

1. **Initialize the Table**: Create a 2D list `f` with dimensions `(m+1) x (n+1)`, where `m` and `n` are the lengths of `s1` and `s2`, respectively. This table will hold the solutions to subproblems, initialized with zeros.
2. **Set Up Base Cases**: Populate the first row and the first column of the table `f`. Each cell in the first row represents the minimum ASCII sum for making the first `i` characters of `s1` equal to an empty `s2` by deleting characters. Each cell in the first column represents the same but for `s2` to an empty `s1`.

   ```python
   for i in range(1, m + 1):
       f[i][0] = f[i - 1][0] + ord(s1[i - 1])
   for j in range(1, n + 1):
       f[0][j] = f[0][j - 1] + ord(s2[j - 1])
   ```

3. **Fill in the Table**: Loop through each cell of the table (excluding the first row and column which are already filled) and apply the following rules to calculate `f[i][j]`:
   - If `s1[i - 1] == s2[j - 1]`, set `f[i][j]` to `f[i - 1][j - 1]`, since there is no need to delete any character when they are the same.
   - If `s1[i - 1] != s2[j - 1]`, set `f[i][j]` to the minimum between `f[i - 1][j] + ord(s1[i - 1])` and `f[i][j - 1] + ord(s2[j - 1])`. This represents the cost of making strings equal either by deleting the current character from `s1` or `s2`.

   ```python
   for i in range(1, m + 1):
       for j in range(1, n + 1):
           if s1[i - 1] == s2[j - 1]:
               f[i][j] = f[i - 1][j - 1]
           else:
               f[i][j] = min(
                   f[i - 1][j] + ord(s1[i - 1]),
                   f[i][j - 1] + ord(s2[j - 1])
               )
   ```

4. **Extract the Final Answer**: After populating the entire table, read the value `f[m][n]`. This cell holds the minimum ASCII sum of deleted characters to make the two strings equal.

By following this approach, the algorithm ensures that all possible deletions are considered, and the best choice is made at each step, leading to the optimal solution.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above. Suppose we have two strings `s1 = "sea"` and `s2 = "eat"`. We want to find the lowest ASCII sum of characters that need to be deleted to make `s1` and `s2` equal.

1. **Initialization**: We start by initializing a 2D list `f` with the dimensions `(m+1) x (n+1)`. For `s1 = "sea"` and `s2 = "eat"`, the grid will look like this with `m=3` and `n=3`:

   ```
   [ [0, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 0] ]
   ```

2. **Set Up Base Cases**: Next, we populate the first row and the first column based on the ASCII sum of the prefixes of `s1` and `s2`:

   ```
   [ [0, 'e', 'a', 't'],
     ['s', 0, 0, 0],
     ['e', 0, 0, 0],
     ['a', 0, 0, 0] ]
   ```

   After converting characters to their ASCII sums:

   ```
   [ [0, 101, 197, 314],
     [115, 0, 0, 0],
     [115 + 101, 0, 0, 0],
     [115 + 101 + 97, 0, 0, 0] ]
   ```

   The fully populated base cases are:

   ```
   [ [0, 101, 197, 314],
     [115, 0, 0, 0],
     [216, 0, 0, 0],
     [313, 0, 0, 0] ]
   ```

3. **Fill in the Table**: Now, we loop through the rest of the cells:
   - For `f[1][1]`: 's' vs 'e', they are different, so we take the min between `f[0][1] + ASCII('s') = 101 + 115` and `f[1][0] + ASCII('e') = 115 + 101`. We choose the minimum which is `216`, so `f[1][1] = 216`.
   - For `f[1][2]`: 's' vs 'a', they are different, so we take the min between `f[0][2] + ASCII('s') = 197 + 115` and `f[1][1] + ASCII('a') = 216 + 97`. We choose the minimum which is `216 + 97`, so `f[1][2] = 313`.
   - And so on, until we fill the entire table:

   ```
   [ [0, 101, 197, 314],
     [115, 216, 313, 330],
     [216, 115, 216, 313],
     [313, 216, 209, 216] ]
   ```

   The logic behind each calculation is to minimize the ASCII sum required to make the two substrings equal, considering whether we should delete a character from `s1` or `s2` at each step.

4. **Extract the Final Answer**: The value of `f[3][3]` represents the minimum ASCII sum of deleted characters to make `s1` and `s2` equal. From the table, this value is `216`. Therefore, the lowest ASCII sum of characters that need to be deleted from "sea" and "eat" to make them equal is `216`.

Python Solution

```python
class Solution:
    def minimumDeleteSum(self, s1: str, s2: str) -> int:
        # Calculate the length of both strings
        len_s1, len_s2 = len(s1), len(s2)

        # Initialize a 2D array to store the minimum delete sum for sub-problems
        dp = [[0] * (len_s2 + 1) for _ in range(len_s1 + 1)]

        # Pre-fill the first column with the delete sum of s1[i:]
        for i in range(1, len_s1 + 1):
            dp[i][0] = dp[i - 1][0] + ord(s1[i - 1])

        # Pre-fill the first row with the delete sum of s2[j:]
        for j in range(1, len_s2 + 1):
            dp[0][j] = dp[0][j - 1] + ord(s2[j - 1])

        # Start filling the DP table from (1,1) to (len_s1, len_s2)
        for i in range(1, len_s1 + 1):
            for j in range(1, len_s2 + 1):
                # If the current characters are equal, then no deletion is needed
                if s1[i - 1] == s2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                else:
                    # Find the minimum between deleting a character from s1 or s2
                    dp[i][j] = min(
                        dp[i - 1][j] + ord(s1[i - 1]),  # Delete from s1
                        dp[i][j - 1] + ord(s2[j - 1])   # Delete from s2
                    )

        # The value at dp[len_s1][len_s2] will be the minimum delete sum for s1 and s2
        return dp[len_s1][len_s2]

```

Java Solution

```java
class Solution {
    public int minimumDeleteSum(String s1, String s2) {
        // Lengths of strings s1 and s2
        int length1 = s1.length(), length2 = s2.length();
        // Initialize a 2D array to store the minimum delete sum
        int[][] dp = new int[length1 + 1][length2 + 1];

        // Fill in the first column with the ASCII values of characters of s1
        for (int i = 1; i <= length1; ++i) {
            dp[i][0] = dp[i - 1][0] + s1.charAt(i - 1);
        }

        // Fill in the first row with the ASCII values of characters of s2
        for (int j = 1; j <= length2; ++j) {
            dp[0][j] = dp[0][j - 1] + s2.charAt(j - 1);
        }

        // Calculate the minimum delete sum for each substring of s1 and s2
        for (int i = 1; i <= length1; ++i) {
            for (int j = 1; j <= length2; ++j) {
                // If the current characters are the same, no need to delete, so copy the value from dp[i - 1][j - 1]
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    // If characters are different, calculate the minimum delete sum
                    // by comparing the cost of deleting from s1 to the cost of deleting from s2
                    dp[i][j] = Math.min(dp[i - 1][j] + s1.charAt(i - 1), dp[i][j - 1] + s2.charAt(j - 1));
                }
            }
        }

        // The answer is in dp[length1][length2] which contains the minimum delete sum to make the strings equal
        return dp[length1][length2];
    }
}

```

C++ Solution

```cpp
#include <cstring> // for using memset
#include <algorithm> // for using min function
using namespace std;

class Solution {
public:
    int minimumDeleteSum(string s1, string s2) {
        int m = s1.size();
        int n = s2.size();
        // dp array to store the minimum ASCII delete sum for two substrings ending up to (i-1) in s1 and (j-1) in s2
        int dp[m + 1][n + 1];
        // Initializing the dp array with 0's
        memset(dp, 0, sizeof(dp));

        // Calculate the delete sum for s1 when s2 is empty
        for (int i = 1; i <= m; ++i) {
            dp[i][0] = dp[i - 1][0] + s1[i - 1];
        }

        // Calculate the delete sum for s2 when s1 is empty
        for (int j = 1; j <= n; ++j) {
            dp[0][j] = dp[0][j - 1] + s2[j - 1];
        }

        // Compute dp values in a bottom-up manner
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                // If characters are the same, no need to delete, carry over the sum from previous subproblem
                if (s1[i - 1] == s2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    // Otherwise, choose the minimum sum achieved by deleting character from either s1 or s2
                    dp[i][j] = min(dp[i - 1][j] + s1[i - 1], dp[i][j - 1] + s2[j - 1]);
                }
            }
        }

        // The final result is stored in dp[m][n], which involves the whole of s1 and s2
        return dp[m][n];
    }
};

```

Typescript Solution

```typescript
function minimumDeleteSum(s1: string, s2: string): number {
    const s1Length = s1.length;
    const s2Length = s2.length;

    // Initialize a 2D array to store the minimum delete sum dp values
    const dp = Array.from({ length: s1Length + 1 }, () => Array(s2Length + 1).fill(0));

    // Fill out the base cases for the first row, cumulative ASCII sum of s1's prefixes
    for (let i = 1; i <= s1Length; ++i) {
        dp[i][0] = dp[i - 1][0] + s1.charCodeAt(i - 1);
    }

    // Fill out the base cases for the first column, cumulative ASCII sum of s2's prefixes
    for (let j = 1; j <= s2Length; ++j) {
        dp[0][j] = dp[0][j - 1] + s2.charCodeAt(j - 1);
    }

    // Dynamic programming to fill out the rest of the dp table
    for (let i = 1; i <= s1Length; ++i) {
        for (let j = 1; j <= s2Length; ++j) {
            if (s1[i - 1] === s2[j - 1]) {
                // If characters match, take the diagonal value
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                // Otherwise, take the minimum of deleting from s1 or s2, and add the ASCII
                // value of the deleted character
                dp[i][j] = Math.min(
                    dp[i - 1][j] + s1.charCodeAt(i - 1),
                    dp[i][j - 1] + s2.charCodeAt(j - 1),
                );
            }
        }
    }

    // The last cell of the dp table will have the answer
    return dp[s1Length][s2Length];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the nested loops that iterate over the lengths of both strings `s1` and `s2`. Since we iterate over all `i` from 1 to `m` and all `j` from 1 to `n`, where `m` is the length of `s1` and `n` is the length of `s2`, the time complexity is `O(m * n)`.

Space Complexity

The space complexity of the code is primarily due to the 2D list `f`, which has dimensions `(m + 1) x (n + 1)`, requiring a total of `(m + 1) * (n + 1)` space. So, the space complexity is also `O(m * n)`.