940. Distinct Subsequences II

Hard String Dynamic Programming

Problem Description

Given a string `s`, the task is to calculate the total number of distinct non-empty subsequences that can be formed from the characters of the string. Note that a subsequence maintains the original order of characters but does not necessarily include all characters. Importantly, if the solution is a very large number, it should be returned modulo `10^9 + 7` to keep the number within manageable bounds. This modular operation ensures that we deal with smaller numbers that are more practical for computation and comparison.

Intuition

The intuition for solving this problem lies in dynamic programming - a method used for solving complex problems by breaking them down into simpler subproblems. The key insight is to build up the number of distinct subsequences as we iterate through each character of the given string.

We maintain an array `dp` of size 26 (to account for each letter of the alphabet) which keeps track of the contribution of each character towards the number of distinct subsequences so far. The variable `ans` stores the total count of distinct subsequences at any given point.

As we traverse the string:

- We calculate the index `i` corresponding to the character `c` (by finding the difference between the ASCII value of `c` and that of `a`).
- The variable `add` is set to the difference between the current total number of subsequences `ans` and the old count of subsequences ending with the character `c` (`dp[i]`). We add 1 to this because a new subsequence consisting of just the character `c` can also be formed.
- The `ans` is updated to include the new subsequences introduced by including the character `c`. The new `ans` is also taken modulo `10^9 + 7` to handle the large numbers.
- The count of subsequences ending with the character `c` (`dp[i]`) is incremented by `add` to reflect the updated state.

This approach ensures that each character's contribution is accounted for exactly once, and by the end of the iteration, `ans` contains the count of all possible distinct subsequences modulo `10^9 + 7`.

Solution Approach

The implementation of the solution is fairly straightforward once we've understood the intuition behind it. This problem employs dynamic programming and a single array to keep track of the contributing counts. Here's the breakdown of the approach:

1. **Initialize the Modulus and DP Array**:
   - We use a modulus `mod` set to `10**9 + 7` to ensure we manage the large numbers effectively by returning the number of subsequences modulo this value.
   - The data structure `dp` is an array initialized to size 26 (representing the English alphabet) with all zeroes. This array will store the count of subsequences that end with a particular character.
2. **Iterate Over the String**:
   - For each character `c` in the string `s`, we do the following steps:
3. **Determine the Index for `c`**:
   - We calculate an index `i` by taking the ASCII value of the character `c`, subtracting the ASCII value of 'a' from it (`ord(c) - ord('a')`). This gives us a unique index from 0 to 25 for each lowercase letter of the alphabet.
4. **Calculate the Additive Contribution**:
   - Compute `add`, which will determine how much the current character `c` will contribute to the new subsequences count. It is determined by the current total of distinct subsequences `ans` minus the previous count stored in `dp[i]` for that character `c`. We add one to `add` to account for the subsequence consisting solely of the new character `c`.
5. **Update the Total Count of Subsequences**:
   - Update `ans` by adding the new contribution from the character `c`. Applying modulo `mod` here ensures we handle overflow and large numbers.
6. **Update the DP Array**:
   - Increase the count in `dp[i]` by the value of `add`. This means that any subsequences that now end with the current character `c` include the old subsequences plus any new subsequences formed due to adding `c`.

Using this method, we build up the solution incrementally, considering the influence of each character on the subsequences. The reason we have to subtract the old count of subsequences for the character `c` before adding it again (after increasing with `add`) is to ensure that we do not double-count subsequences already accounted for by previous appearances of `c`.

The complexity of this solution is O(n), where n is the length of the string since we go through every character of the string once, performing O(1) operations for each.

Example Walkthrough

To illustrate the solution approach, let's go through a small example using the string `"aba"`.

1. **Initialize the Modulus and DP Array**:
   We set `mod` to `10^9 + 7`. Our `dp` array, which has 26 elements for each letter of the English alphabet, is initialized to zeroes.
2. **Iterate Over the String**:
   We begin iterating through the string `"aba"` character by character.
3. **First Character - 'a'**:
   - **Determine the Index for 'a'**: We calculate the index for 'a' as `0` (since `'a' - 'a' = 0`).
   - **Calculate the Additive Contribution**: Since this is the first character, and there are no previous subsequences, we calculate `add = ans (initially 0) - dp[0] + 1 = 0 - 0 + 1 = 1`.
   - **Update the Total Count of Subsequences**: We set `ans = ans + add = 0 + 1 = 1` (the subsequences are now `""` and `"a"`).
   - **Update the DP Array**: We update `dp[0]` to `dp[0] + add = 0 + 1 = 1`.
4. **Second Character - 'b'**:
   - **Determine the Index for 'b'**: The index for 'b' is `1` (`'b' - 'a' = 1`).
   - **Calculate the Additive Contribution**: The current total count of distinct subsequences, `ans`, is 1. `add = ans - dp[1] + 1 = 1 - 0 + 1 = 2` (which corresponds to new subsequences `"b"` and `"ab"`).
   - **Update the Total Count of Subsequences**: Now `ans = ans + add = 1 + 2 = 3` (the subsequences are `""`, `"a"`, `"b"`, and `"ab"`).
   - **Update the DP Array**: We update `dp[1]` to `dp[1] + add = 0 + 2 = 2`.
5. **Third Character - 'a' (again)**:
   - **Determine the Index for 'a'**: The index is still `0`.
   - **Calculate the Additive Contribution**: We know `ans` is 3, and since we've encountered 'a' before, we subtract its previous contribution from `ans` and add 1: `add = ans - dp[0] + 1 = 3 - 1 + 1 = 3` (these are new subsequences: `"a"`, `"ba"`, and `"aba"`).
   - **Update the Total Count of Subsequences**: The new `ans` will be `ans + add = 3 + 3 = 6` (the subsequences now are `""`, `"a"`, `"b"`, `"ab"`, `"ba"`, `"aa"`, and `"aba"`; note that we don't count subsequences like `"aa"` as distinct since the order must be maintained).
   - **Update the DP Array**: We set `dp[0]` to `dp[0] + add = 1 + 3 = 4`.

At the end of this process, the final answer for the total count of distinct non-empty subsequences is `6`. However, if `ans` were larger, we would apply the modulo operation to ensure we obtain a result within the bounded range.

This walkthrough demonstrates how the algorithm incrementally computes the count of distinct subsequences using dynamic programming and efficient arithmetic operations, handling each character's contribution exactly once.

Solution Implementation

Python

```python
class Solution:
    def distinctSubseqII(self, s: str) -> int:
        # Define the modulo value to handle large numbers
        MOD = 10**9 + 7

        # Initialize an array to keep track of the last count of subsequences
        # ending with each letter of the alphabet
        last_count = [0] * 26

        # Initialize the total count of distinct subsequences
        total_count = 0

        # Iterate over each character in the string
        for char in s:
            # Get the index of the current character in the alphabet (0-25)
            index = ord(char) - ord('a')

            # Calculate how many new subsequences are added by this character:
            # It is total count (all previous subsequences) minus last count[index]
            # (which we have already counted with the current character) plus 1 for the character itself
            added_subseq = total_count - last_count[index] + 1

            # Update the total count of distinct subsequences
            total_count = (total_count + added_subseq) % MOD

            # Update the last count of subsequences for the current character
            last_count[index] = (last_count[index] + added_subseq) % MOD

        # Return the total count of distinct subsequences
        return total_count
```

Java

```java
class Solution {
    private static final int MOD = (int) 1e9 + 7; // Modulus value for handling large numbers

    public int distinctSubseqII(String s) {
        int[] lastOccurrenceCount = new int[26]; // Array to store the last occurrence count of each character
        int totalDistinctSubsequences = 0; // Variable to store the total count of distinct subsequences

        // Iterate through each character in the string
        for (int i = 0; i < s.length(); ++i) {
            // Determine the alphabet index of current character
            int alphabetIndex = s.charAt(i) - 'a';

            // Calculate the number to add. This number represents the new subsequences that will be formed by adding the new charact
            // Subtract the last occurrence count of this character to avoid counting subsequences formed by prior occurrences of thi
            // And add 1 for the subsequence consisting of the character itself.
            int newSubsequences = (totalDistinctSubsequences - lastOccurrenceCount[alphabetIndex] + 1 + MOD) % MOD;

            // Update the totalDistinctSubsequences by adding newSubsequences
            totalDistinctSubsequences = (totalDistinctSubsequences + newSubsequences) % MOD;

            // Update the last occurrence count for this character in the lastOccurrenceCount array
            lastOccurrenceCount[alphabetIndex] = (lastOccurrenceCount[alphabetIndex] + newSubsequences) % MOD;
        }

        // Since the result can be negative due to the subtraction during the loop,
        // we add MOD and then take the modulus to ensure a non-negative result
        return (totalDistinctSubsequences + MOD) % MOD;
    }
}
```

C++

```cpp
class Solution {
public:
    const int MODULO = 1e9 + 7;

    // Method to calculate the number of distinct subsequences in the string `s`.
    int distinctSubseqII(string s) {
        vector<long> lastOccurrence(26, 0); // Array to store the last occurrence contribution for each character
        long totalCount = 0;  // Total count of distinct subsequences

        // Loop through each character in the string
        for (char& c : s) {
            int index = c - 'a';  // Map character 'a' to 'z' to index 0 to 25
            long additionalCount = (totalCount - lastOccurrence[index] + 1 + MODULO) % MODULO; // Calculate the additional count for

            totalCount = (totalCount + additionalCount) % MODULO; // Update the total count
            lastOccurrence[index] = (lastOccurrence[index] + additionalCount) % MODULO; // Update the last occurrence contribution fo
        }

        return (int)totalCount; // Return the total count of distinct subsequences as an integer
    }
};
```

TypeScript

```typescript
function distinctSubseqII(s: string): number {
    const MODULO: number = 1e9 + 7; // A constant for the modulo operation to prevent overflow
    const lastOccurrence = new Array<number>(26).fill(0); // Store the count of distinct subsequences ending with each letter

    // Iterate over each character in the string
    for (const char of s) {
        const charIndex: number = char.charCodeAt(0) - 'a'.charCodeAt(0); // Map 'a' to 0, 'b' to 1, etc.

        // Calculate the number of new distinct subsequences ending with the current character
        // It is the sum of all distinct subsequences seen so far plus 1 (for the char itself)
        lastOccurrence[charIndex] = lastOccurrence.reduce((runningTotal, currentValue) =>
            (runningTotal + currentValue) % MODULO, 0) + 1;
    }

    // Return the sum of all distinct subsequences modulo the defined constant to avoid overflow
    return lastOccurrence.reduce((runningTotal, currentValue) =>
        (runningTotal + currentValue) % MODULO, 0);
}
```


```python
class Solution:
    def distinctSubseqII(self, s: str) -> int:
        # Define the modulo value to handle large numbers
        MOD = 10**9 + 7

        # Initialize an array to keep track of the last count of subsequences
        # ending with each letter of the alphabet
        last_count = [0] * 26

        # Initialize the total count of distinct subsequences
        total_count = 0

        # Iterate over each character in the string
        for char in s:
            # Get the index of the current character in the alphabet (0-25)
            index = ord(char) - ord('a')

            # Calculate how many new subsequences are added by this character:
            # It is total count (all previous subsequences) minus last count[index]
            # (which we have already counted with the current character) plus 1 for the character itself
            added_subseq = total_count - last_count[index] + 1

            # Update the total count of distinct subsequences
            total_count = (total_count + added_subseq) % MOD

            # Update the last count of subsequences for the current character
            last_count[index] = (last_count[index] + added_subseq) % MOD

        # Return the total count of distinct subsequences
        return total_count
```


Time and Space Complexity

The provided code computes the count of distinct subsequences in a string using dynamic programming.

Time Complexity

The key operation is iterating over each character in the input string `s`. For each character, the algorithm performs a constant amount of work; it updates `ans` and modifies an element in the `dp` array, which is of fixed size 26 (corresponding to the lowercase English alphabet). The time complexity is therefore **O(N)**, where **N** is the length of the string `s`.

Space Complexity

The space complexity of the algorithm is defined by the space needed to store the `dp` array and the variables used. The `dp` array requires space for 26 integers, regardless of the length of the input string. Hence, the space complexity is **O(1)** since the required space does not scale with the input size but is a constant size due to the fixed alphabet size.