115. Distinct Subsequences

Hard String Dynamic Programming Leetcode Link

Problem Description

Given two strings `s` and `t`, the task is to calculate the number of distinct subsequences of `s` that are equal to `t`. A subsequence of a string is defined as a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. For example, `"ace"` is a subsequence of `"abcde"` but `"aec"` is not. It's important to note that the problem specifies distinct subsequences, which means that each subsequence counts only once even if it can be formed in multiple ways.

The constraints of the problem ensure that any given answer fits within a 32-bit signed integer, thus eliminating the need to handle extremely large outputs that could otherwise result in overflow errors.

Intuition

The intuition behind the solution to this problem is to approach it using dynamic programming, which involves solving complex problems by breaking them down into simpler subproblems. The solution constructs a table that keeps track of the number of ways to form subsequences of varying lengths, progressively building up to the length of `t`.

To get to the solution, let's define `f[i][j]` as the number of distinct subsequences in `s[0...i]` that equals `t[0...j]`. However, to optimize space, we can use a one-dimensional array `f[j]` which holds the counts for the current iteration. The value of `f[j]` will be updated by considering two cases:

1. If the current character in `s` (`s[i]`) does not match the current character in `t` (`t[j-1]`), the number of distinct subsequences is unchanged.
2. If `s[i]` matches `t[j-1]`, the number of distinct subsequences is the sum of the subsequences without including `s[i]` (which is `f[j]` before update) and the subsequences including `s[i]`, which is the same as the number of ways to form `t[0...(j-1)]` from `s[0...(i-1)]`, which is stored in `f[j-1]`.

The initial state `f[0]` is 1 as there is exactly one subsequence of any string `s` that equals an empty string `t`: the empty subsequence. We start populating the array `f` from the end to the beginning to correctly use previously calculated values for the current state. After processing all characters of `s`, the last element of this array, `f[n]` (where `n` is the length of `t`), will contain the number of distinct subsequences of `s` which equal `t`, which is the final answer.

The provided solution code implements this dynamic programming approach efficiently in both time and space. The time complexity of this solution is O(length of `s` * length of `t`), and the space complexity is O(length of `t`).

Solution Approach

The implementation begins by defining a one-dimensional array `f`, where `f[i]` will represent the number of distinct subsequences of `s` that match the first `i` characters of `t`. The size of this array is `n + 1`, where `n` is the length of string `t`. This allows us to store the results for all prefixes of `t`, including an empty string as our base case which has exactly one matching subsequence (the empty subsequence itself).

Here are the steps of the implementation:

1. Initialize the array `f` with its first element `f[0]` equal to 1 and the rest set to 0. This corresponds to the fact that there is always one way to form the empty subsequence `t` from any string `s`.

2. Iterate over the characters of the string `s` using a variable `a`.

3. For each `a` in `s`, iterate over the string `t` backward from `n` down to 1 (inclusive). The reason for iterating backward is that we want to use the values from the previous step without them being overwritten, as we only keep one row in memory.

4. Check if the current character `a` matches the character in `t` at the current index `j - 1`.

5. If there is a match, update `f[j]` by adding `f[j - 1]` to it. This represents that for the current character match, the new number of distinct subsequences can be found by adding the subsequences found without including the current character `a` from `s` (`f[j]` before the update) to the number of subsequences that were found for the previous character of `t` (`f[j - 1]`).

After the outer loop completes, `f[n]` contains the total number of distinct subsequences of `s` that match string `t`. The code returns this value as the final answer.

This algorithm uses dynamic programming by storing intermediate results in an array and reusing them, and only requires `O(n)` space where `n` is the length of `t` due to the one-dimensional array, which is a significant optimization over a naive two-dimensional approach.

The data structure used here is a simple array, and the pattern is dynamic programming with memoization to avoid redundant computations. The algorithm's time complexity is `O(length of s * length of t)` and space complexity is `O(length of t)`.

Example Walkthrough

Let's consider `s = "babbbit"` and `t = "bit"`. We want to find the number of distinct subsequences of `s` that are equal to `t`.

We will follow these steps:

Step 1: Initialize array `f` of size `length of t + 1` which is `4` in this case (`t` has 3 characters, plus 1 for the base case). Set `f[0] = 1` because there is one subsequence of any string that equals an empty string—namely, the empty subsequence itself. So `f = [1, 0, 0, 0]`.

Step 2: Start iterating over characters in `s`. For each character, iterate over `t` backward:

- **When `a = b`:** Iterate through `t` in reverse:
  - `t[2] = 't'` doesn't match `a`, so `f[3]` remains `0`.
  - `t[1] = 'i'` doesn't match `a`, so `f[2]` remains `0`.
  - `t[0] = 'b'` matches `a`, so `f[1]` is updated to `f[1] + f[0]`, which becomes `1`. The array is now `[1, 1, 0, 0]`.
- **When `a = a`:**
  - No matches, no updates, because there is no 'a' in `t`. The array remains `[1, 1, 0, 0]`.
- **When `a = b` again:**
  - `t[2]` doesn't match, no update.
  - `t[1]` doesn't match, no update.
  - `t[0]` matches, so `f[1]` becomes `f[1] + f[0]`, now `2`, array is `[1, 2, 0, 0]`.
- **When `a = b` again:**
  - `t[2]` doesn't match, no update.
  - `t[1]` doesn't match, no update.
  - `t[0]` matches, `f[1]` becomes `f[1] + f[0]`, now `3`, array is `[1, 3, 0, 0]`.
- **When `a = b` again:** Still no changes for `t[1]` and `t[2]`, but `f[1]` becomes `4` because `f[1]` updates to `f[1] + f[0]`.
- **When `a = i`:**
  - `t[2]` doesn't match, no update.
  - `t[1] = 'i'` matches, so `f[2]` becomes `f[2] + f[1]`, which is `4`. Array is now `[1, 4, 4, 0]`.
- **When `a = t`:**
  - `t[2] = 't'` matches, so `f[3]` becomes `f[3] + f[2]`, now `4`. The array is `[1, 4, 4, 4]`.

After completing this process, we have `f[n] = f[3] = 4`, so there are **4** distinct subsequences of `s` that are equal to `t`: "bbit", "bbit", "bbit", "bbit". Though they are formed from different positions in `s`, each represents the same subsequence, so the count is 4.

To clarify, these subsequences are derived from the following indices of `s`:

1. `s[0]s[5]s[7]` - "bbit"
2. `s[1]s[5]s[7]` - "bbit"
3. `s[2]s[5]s[7]` - "bbit"
4. `s[3]s[5]s[7]` - "bbit"

This example illustrates the dynamic programming approach where we break down the problem into smaller subproblems and use a table (in this case, a one-dimensional array) to store intermediate results and avoid redundant calculations.

Python Solution

```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        # Length of the string 't' to find
        target_length = len(t)
        # Initialize a DP array with zeros and set the first element to 1
        dp = [1] + [0] * target_length

        # Loop through each character in string 's'
        for char in s:
            # Iterate backwards through the target 't'
            for j in range(target_length, 0, -1):
                # When the characters match, update the DP array
                if char == t[j - 1]:
                    dp[j] += dp[j - 1]

        # Return the last element in the DP array, which holds the answer
        return dp[target_length]

```

Java Solution

```java
class Solution {
    public int numDistinct(String s, String t) {
        // Length of the target string 't'
        int targetLength = t.length();

        // dp array for storing the number of distinct subsequences
        int[] dp = new int[targetLength + 1];

        // Base case initialization: An empty string is a subsequence of any string
        dp[0] = 1;

        // Iterate through each character in the source string 's'
        for (char sourceChar : s.toCharArray()) {
            // Iterate backwards through the dp array
            // This is done to ensure that we are using the results from the previous iteration
            for (int j = targetLength; j > 0; --j) {
                // Get the jth character of the target string 't'
                char targetChar = t.charAt(j - 1);

                // If the current characters in 's' and 't' match,
                // we add the number of distinct subsequences up to the previous character
                if (sourceChar == targetChar) {
                    dp[j] += dp[j - 1];
                }
            }
        }

        // Return the total distinct subsequences of 't' in 's'
        return dp[targetLength];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int numDistinct(string source, string target) {
        int targetLength = target.size();                   // Get the length of the target string
        unsigned long long dp[targetLength + 1];            // Create a dynamic programming array to store intermediate results
        memset(dp, 0, sizeof(dp));                          // Initialize the array with zeroes
        dp[0] = 1;                                          // Base case: an empty target has one match in any source

        // Iterate over each character in the source string
        for (char& sourceChar : source) {
            // Iterate over the target string backwards, to avoid overwriting values we still need
            for (int i = targetLength; i > 0; --i) {
                char targetChar = target[i - 1];
                // If the current source character matches this character in target,
                // update the dp array to include new subsequence combinations
                if (sourceChar == targetChar) {
                    dp[i] += dp[i - 1];
                }
            }
        }
        // The answer to the problem (number of distinct subsequences) is now in dp[targetLength]
        return dp[targetLength];
    }
};

```

Typescript Solution

```typescript
function numDistinct(source: string, target: string): number {
    // Length of the target string
    const targetLength: number = target.length;

    // Initialize an array to keep track of the number of distinct subsequences
    const distinctSubseqCount: number[] = new Array(targetLength + 1).fill(0);

    // Base case: An empty target has exactly one subsequence in any source string
    distinctSubseqCount[0] = 1;

    // Iterate over the source string to find distinct subsequences matching the target
    for (const sourceChar of source) {
        // Work backwards through the target string
        // This prevents overwriting values that are still needed
        for (let idx = targetLength; idx > 0; --idx) {
            const targetChar = target[idx - 1];
            // If the characters match, update the count of distinct subsequences
            if (sourceChar === targetChar) {
                distinctSubseqCount[idx] += distinctSubseqCount[idx - 1];
            }
        }
    }
    // Return the total number of distinct subsequences that match the entire target string
    return distinctSubseqCount[targetLength];
}

```

Time and Space Complexity

The given Python code defines a function `numDistinct` that calculates the number of distinct subsequences of string `s` that equal string `t`. The time complexity and space complexity analysis are as follows:

- **Time Complexity:** The time complexity of the code is `O(n * m)`, where `n` is the length of string `t` and `m` is the length of string `s`. This is because there is a double loop structure, where the outer loop iterates over each character in `s` and the inner loop traverses the list `f` backwards from `n` to `1`. For each character in `s`, the inner loop compares it with the characters in `t` and updates `f[j]` accordingly.

- **Space Complexity:** The space complexity of the code is `O(n)`, where `n` is the length of string `t`. The list `f` has `n + 1` elements, corresponding to the number of characters in `t` plus one for the base case. No additional space is used that grows with the size of `s`, therefore, space complexity is linear with respect to the length of `t`.