91. Decode Ways

Medium String Dynamic Programming

Problem Description

The problem involves decoding a message that consists exclusively of numeric digits, which represents encoded letters 'A' to 'Z' where 'A' corresponds to '1', and 'Z' to '26'. The objective is to determine the total number of valid ways the given numeric message can be decoded back into letters. A valid decoding is one where all the digits are grouped into subparts that represent numbers from '1' to '26'. For example, the sequence "123" can be decoded as 'ABC' (1, 2, 3), 'AW' (1, 23), or 'LC' (12, 3). However, some groupings may be invalid, such as "06" because '0' does not correspond to any letter, and the leading zero makes it an invalid encoding for 'F', which is '6'.

Intuition

The solution to the problem is grounded in dynamic programming, which is a method used to solve problems by breaking them down into simpler subproblems and solving each subproblem just once, storing its solution. The concept is to start with the base of the string and compute the number of ways we can decode it, then work our way up, adding the number of ways the rest of the string can be decoded.

For each character in the string `s`, we have one of three cases:

1. The current character is '0'. We cannot decode '0' on its own because there's no corresponding letter. So, we set the current number of decodings (`h`) to 0.
2. The current character is not '0'. This means we can decode it as a single digit. We set the current number of decodings (`h`) to the previous number of decodings (`g`), which represents the number of ways we can decode the string up to this point ignoring the previous digit.
3. The current character, along with the previous character, forms a valid two-digit number less than or equal to 26 (ignoring leading zeros). If this is the case, we add the number of decodings two characters back (`f`) to our current number of decodings (`h`). This represents considering the two-digit number as a single letter and counting the ways to decode the string up to this new point.

We iteratively update `f` and `g`, where `f` represents the number of ways to decode the string up to the previous character, and `g` represents the number of ways to decode the string up to the current character. By the end of the iteration, `g` will give us the total number of ways the entire string can be decoded.

Solution Approach

The solution implements a dynamic programming approach, leveraging the tabulation technique to store and utilize previously computed values for the current computation. The core of this solution relies on solving subproblems of increasing sizes while adhering to the problem's constraints.

Here's a walkthrough of the code provided, explaining its components:

- We start by defining two variables, `f` and `g`. Variable `g` acts as a running total for the number of ways to decode the current substring of `s`. Initially, it's set to 1, indicating that there's at least one way to decode an empty string (by doing nothing). `f` represents the number of ways to decode the substring that ends one character before the current one.
- The `for` loop iterates over the string `s`, where `i` is the index (1-based for convenience) and `c` is the character at the current position. Within this loop, we're interested in updating the count of decode ways for the current position.
- For the current character `c`, we initialize `h` to 0, but if `c` is not "0", it means it can be a valid digit on its own, so we set `h` to the current value of `g`, which is the number of ways to decode up to the previous character.
- We then check if there's a valid two-character combination that can be formed with the current and the previous character. To do this, we check if the previous character is not "0" and if converting the two characters into a number results in a value that's 26 or less. If so, we add `f` to `h`, because `f` represents the number of ways the string could have been decoded before these two characters were taken as a pair.
- After processing the current character, we update `f` to hold the previous value of `g`, and `g` gets updated to the value of `h` which now represents the number of ways to decode the string including the current character.
- Finally, once the loop is done, `g` holds the total number of ways to decode the entire string, which is the answer we return from the function `numDecodings`.

Here's the key idea of the algorithm: Each position in the input string is a pivot point that either continues the ways from the previous pivot or combines with the previous character to create additional ways to decode. This results in a cumulative way to count all possible decodings. By keeping track of these two states (`f` and `g`), we can efficiently compute the result without the need for a full array to store the ways to decode every substring, thus optimizing the space complexity of the algorithm.

This approach is often used in scenarios that require counting combinations or permutations, especially when the current state can be calculated from a fixed number of previous states.

Example Walkthrough

Let's use a small example to illustrate the solution approach with the string `s = "121"`. We want to determine the number of valid decodings for this message.

1. Initialize two variables, `f = 0` and `g = 1`. `f` will keep track of the ways to decode the string up to the character before the current one, and `g` keeps track of the ways to decode up to the current character.
2. Start iterating over the string `s`:
   - **Iteration 1**: `i = 1`, `c = '1'`
     - `h` is initialized to 0. Since `c` is not '0', set `h` to `g`, which is 1.
     - There's no character before the first one, so we don't check the two-character combination here.
     - Update `f` to be the previous value of `g`, which is 1, and `g` now becomes the value of `h`, also 1.
   - **Iteration 2**: `i = 2`, `c = '2'`
     - `h` initialized to 0. `c` is not '0', so `h` is set to the current `g` value, which is 1.
     - The previous character '1' and current '2' can form '12'. Since '12' is ≤ 26, we add the value of `f` (1) to `h`.
     - Now, `h` is 1 (`g`) + 1 (`f`) = 2, which means there are two ways to decode "12".
     - Update `f` to `g` (1), and update `g` to the current `h` (2).
   - **Iteration 3**: `i = 3`, `c = '1'`
     - `h` initialized to 0. `c` is not '0', so `h` is set to the current `g` value, which is 2.
     - The previous character '2' and the current '1' can form '21'. Since '21' is ≤ 26, we add the value of `f` (1) to `h`.
     - Now, `h` is 2 (`g`) + 1 (`f`) = 3, which means there are three ways to decode "121".
     - As this is the last iteration, `i` is updated to `g` (2), and `g` is updated to `h` (3).
3. At the end of iteration, `g` now holds the value 3, which is the total number of ways to decode the entire string "121".

The final answer for the total number of valid ways to decode string `s = "121"` is `3`. This is because we can decode "121" as follows:

- 'ABA' (decoded as 1, 2, 1)
- 'AU' (decoded as 1, 21)
- 'LA' (decoded as 12, 1)

This demonstrates the dynamic programming approach to decoding the message, efficiently calculating the result iteratively without redundant recalculations.

Solution Implementation

Python

```python
class Solution:
    def num_decodings(self, s: str) -> int:
        # Initialize a previous count 'prev_count' of decodings and a current count 'curr_count'.
        prev_count, curr_count = 0, 1

        # Iterate over the string with index to keep track of positions.
        for i in range(len(s)):
            # The next count of decodings 'next_count' should start at 0.
            next_count = 0

            # If current character is not '0', we can use it for a valid decoding.
            if s[i] != '0':
                next_count = curr_count

            # If the current and the previous digit make a number ≤ 26, it can be decoded as well.
            # We need to ensure that we're at the second or later character and that
            # the previous character isn't '0'.
            if i > 0 and s[i-1] != '0' and (s[i-1] == '1' or (s[i-1] == '2' and s[i] < '7')):
                # If the condition is true, add the previous count of decodings.
                next_count += prev_count

            # Update 'prev_count' and 'curr_count' for the next iteration.
            prev_count, curr_count = curr_count, next_count

        # The final 'curr_count' is the total number of decodings for the string.
        return curr_count
```

Java

```java
class Solution {

    // Method to calculate the number of ways to decode a message
    public int numDecodings(String s) {
        // String length
        int length = s.length();
        // Variables to hold previous and current number of decodings
        int prevCount = 0, currentCount = 1;

        // Loop through each character in the string
        for (int i = 1; i <= length; ++i) {
            // Initialize the next count as 0
            int nextCount = 0;

            // If the current character is not '0', it can stand alone, so add current count to next count
            if (s.charAt(i - 1) != '0') {
                nextCount = currentCount;
            }

            // If there are more than one characters and the substring of two characters can represent a valid alphabet
            if (i > 1 && s.charAt(i - 2) != '0' && Integer.valueOf(s.substring(i - 2, i)) <= 26) {
                // Add the previous count to next count
                nextCount += prevCount;
            }

            // Update prevCount and currentCount for the next iteration
            prevCount = currentCount;
            currentCount = nextCount;
        }

        // Return the total count of decodings for the entire string
        return currentCount;
    }
}
```

C++

```cpp
class Solution {
public:
    int numDecodings(string s) {
        int n = s.size(); // Length of the input string

        // Use dynamic programming to solve the problem,
        // where prevTwo represents f[i-2] and prevOne represents f[i-1]
        int prevTwo = 0, prevOne = 1;

        for (int i = 1; i <= n; ++i) {
            // Calculate current ways to decode (current)
            int current = s[i - 1] != '0' ? prevOne : 0; // If current character isn't '0', it can be used as a single digit

            // Check if the previous character and current character
            // can form a valid two-digit number (10 to 26)
            if (i > 1 && (s[i - 2] == '1' || (s[i - 2] == '2' && s[i - 1] <= '6'))) {
                current += prevTwo; // Append valid two-digit decoding ways to current
            }

            prevTwo = prevOne; // Update prevTwo to be the previous value of prevOne
            prevOne = current; // Update prevOne to the current value (which will be prevOne for the next iteration)
        }

        // Return the total number of ways to decode the string
        return prevOne;
    }
};
```

TypeScript

```typescript
function numDecodings(s: string): number {
    const stringLength = s.length;
    let previousCount = 0;  // This will hold the count of decodings ending with the previous character.
    let currentCount = 1;   // This will hold the count of decodings up to the current character.

    // Iterate through the string to determine the number of ways to decode it.
    for (let i = 1; i <= stringLength; ++i) {
        // If current character is not '0', take the count from the last character.
        let newCount = s[i - 1] !== '0' ? currentCount : 0;

        // Check if the last two characters form a valid encoding ("10"-"26").
        // If they do, add the decoding count that ended before the previous character.
        if (i > 1 && (s[i - 2] === '1' || (s[i - 2] === '2' && s[i - 1] <= '6'))) {
            newCount += previousCount;
        }

        // Update the previous and current counts for the next iteration.
        [previousCount, currentCount] = [currentCount, newCount];
    }

    // Return the total number of ways to decode the entire string.
    return currentCount;
}
```

```python
class Solution:
    def num_decodings(self, s: str) -> int:
        # Initialize a previous count 'prev_count' of decodings and a current count 'curr_count'.
        prev_count, curr_count = 0, 1

        # Iterate over the string with index to keep track of positions.
        for i in range(len(s)):
            # The next count of decodings 'next_count' should start at 0.
            next_count = 0

            # If current character is not '0', we can use it for a valid decoding.
            if s[i] != '0':
                next_count = curr_count

            # If the current and the previous digit make a number ≤ 26, it can be decoded as well.
            # We need to ensure that we're at the second or later character and that
            # the previous character isn't '0'.
            if i > 0 and s[i-1] != '0' and (s[i-1] == '1' or (s[i-1] == '2' and s[i] < '7')):
                # If the condition is true, add the previous count of decodings.
                next_count += prev_count

            # Update 'prev_count' and 'curr_count' for the next iteration.
            prev_count, curr_count = curr_count, next_count

        # The final 'curr_count' is the total number of decodings for the string.
        return curr_count
```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the single loop that iterates over the input string `s`. Inside the loop, all operations (condition checking, integer conversion, and arithmetic operations) are executed with constant time complexity `O(1)`. Since the loop runs for each character in the input string, the time complexity is directly proportional to the length of the string. Hence, the time complexity of the code can be given as `O(n)`, where `n` is the length of the input string `s`.

Space Complexity

The space complexity of the code relates to the amount of memory used in relation to the input size. In this code, we use only a fixed number of variables (`f`, `g`, and `h`), which are independent of the input size. This leads to a constant space usage, regardless of the length of the input string. Consequently, the space complexity of the algorithm is `O(1)`, indicating that it requires constant space.