2533. Number of Good Binary Strings

Medium Dynamic Programming Leetcode Link

Problem Description

This problem presents a condition-based combinatorial challenge where we need to count the number of binary strings that adhere to specific grouping rules. The binary strings must:

- Have a length within the inclusive range specified by `minLength` and `maxLength`.
- Contain blocks of consecutive `1`s where each block's size is a multiple of the given `oneGroup`.
- Contain blocks of consecutive `0`s where each block's size is a multiple of the given `zeroGroup`.

Furthermore, we are asked to return the count of such "good" binary strings modulo `10^9 + 7` to handle large numbers that could result from the computation.

Intuition

Understanding the pattern of good binary strings can lead us to a dynamic programming approach. Given the conditions, we can incrementally build a binary string of length `i` by adding a block of `1`s or `0`s at the end of a string, ensuring the following:

- Blocks must have sizes that are multiples of `oneGroup` or `zeroGroup`.
- We only add blocks if they produce strings within the desired length range.

The intuition behind the solution is to use a dynamic programming table to store the number of ways to form a good binary string of length `i`, which we denote as `f[i]`. Starting with the base case, `f[0]` is set to 1, representing the empty string.

For each length `i`, we look back `oneGroup` and `zeroGroup` lengths from `i` and add up the ways because:

- If we are at length `i` and we add a block of `1`s of `oneGroup` length, we can use all combinations of length `i−oneGroup` to form new combinations.
- Similarly, we can do the same with blocks of `0`s of `zeroGroup` length.
- We take care to only add blocks if the resulting length does not exceed our maximum length constraint.

Finally, since we are only interested in strings with lengths between `minLength` and `maxLength`, we sum the values stored in our dynamic programming table from `f[minLength]` to `f[maxLength]`, ensuring we only count the strings of acceptable lengths.

Implementing this approach in a loop that iterates through the possible lengths of binary strings, we can compute the answer. It's important to take the modulo at each step to prevent integer overflow given the constraint on the output.

Solution Approach

The solution to the problem uses a dynamic programming approach, a common pattern in solving combinatorial problems where we build solutions to sub-problems and use those solutions to construct answers to larger problems. Here, the sub-problems involve finding the number of good binary strings of a smaller length, and these are combined to find the number of strings of larger lengths.

The implementation details are as follows:

- **Data Structure**: We use a list `f` with length `maxLength + 1`. The index `i` in the list represents the number of good binary strings of length `i`. It is initialized with `f[0] = 1` (since the empty string is considered a good string) and `0` for all other lengths.

- **Algorithm**: We iterate from `1` to `maxLength` to fill up the dynamic programming table `f`. For each length `i`, we can either add a block of `1`s or a block of `0`s at the end of an existing good string which is shorter by `oneGroup` or `zeroGroup` respectively:

    - If `i − oneGroup >= 0`, we can take all good strings of length `i − oneGroup` and add a block of `1`s to them. Thus, we update `f[i]` to include these new combinations by adding `f[i − oneGroup]`.

    - If `i − zeroGroup >= 0`, similarly, we can take all good strings of length `i − zeroGroup` and add a block of `0`s to them, updating `f[i]` to include these combinations by adding `f[i − zeroGroup]`.

    - We use the modulo operator at each update to ensure that we do not encounter integer overflow, since we are interested in the count modulo `10^9 + 7`.

- **Final Count**: Once the dynamic programming table is complete, the last step is to obtain the sum of all `f[i]` values for `i` between `minLength` and `maxLength` inclusive, and take the modulo again to get the final count of good binary strings within the given length range.

Here is the core part of the solution, with comments added for clarification:

```
mod = 10**9 + 7
f = [1] + [0] * maxLength
for i in range(1, len(f)):
    if i - oneGroup >= 0:
        f[i] += f[i - oneGroup]
    if i - zeroGroup >= 0:
        f[i] += f[i - zeroGroup]
    f[i] %= mod
return sum(f[minLength:]) % mod
```

From the solution, we can see that the algorithm is a bottom-up dynamic programming solution since it builds up the answer iteratively. The space complexity of the algorithm is O(maxLength) because of the array `f`, and the time complexity is O(maxLength) as well, because we iterate through the array once.

Example Walkthrough

Let us assume `minLength` is 3, `maxLength` is 5, `oneGroup` is 2, and `zeroGroup` is 3. We need to count the number of binary strings that fit the criteria based on the given conditions.

We initialize our dynamic programming array `f` with the size of `maxLength + 1`, which is `f[0]` to `f[5]` for this example. We start by setting `f[0] = 1` since the empty string is a valid string and zero for all other lengths.

Now, we start populating `f` from `f[1]` to `f[5]`:

1. For `i = 1` and `i = 2`: We cannot add any blocks of `1`s or `0`s because `oneGroup` and `zeroGroup` are larger than `i`. Thus, `f[1]` and `f[2]` remain 0.

2. For `i = 3`: We can add a block of `0`s because `i − zeroGroup = 0` and `f[0] = 1`. Therefore, `f[3]` becomes 1 after considering the block of `0`s. We cannot add a block of `1`s since `oneGroup` is 2 and there's no way to form a `1` block within this length.

3. For `i = 4`: We can add a block of `1`s since `i − oneGroup = 2` and `f[2] = 0`. So, `f[4]` becomes 0 + 0 = 0. We do not have a way to add consecutive `0`s of length `zeroGroup` to a string of length 2 (since that would exceed the current length `i = 4`).

4. For `i = 5`: We can add a block of `1`s (`f[3] = 1` from prior step), so `f[5]` is updated to 1. We also check for zero blocks, but `5 − zeroGroup` is less than `0` and thus, cannot form a valid string.

To summarize the DP array: `f[0] = 1`, `f[1] = 0`, `f[2] = 0`, `f[3] = 1`, `f[4] = 0`, `f[5] = 1`.

Finally, to count the number of good strings of length between `minLength` and `maxLength`, we sum `f[3]`, `f[4]`, and `f[5]`. The result is `1 + 0 + 1 = 2`. Thus, there are two "good" binary strings that satisfy the conditions between the length of 3 and 5, given the groupings of `1`s and `0`s.

The final answer is 2, and we will return this count modulo `10^9 + 7`, which in this simple case remains 2.

Python Solution

```python
class Solution:
    def goodBinaryStrings(self, min_length: int, max_length: int, one_group: int, zero_group: int) -> int:
        MOD = 10**9 + 7  # define the modulo value

        # Initialize the dynamic programming table `f` with the base case f[0] = 1 and the rest 0s
        f = [1] + [0] * max_length

        # Calculate the number of good binary strings for each length up to maxLength
        for i in range(1, len(f)):
            # If adding a group of 1s is possible, update f[i]
            if i - one_group >= 0:
                f[i] += f[i - one_group]

            # If adding a group of 0s is possible, update f[i]
            if i - zero_group >= 0:
                f[i] += f[i - zero_group]

            # Modulo operation to keep the number within the bounds of MOD
            f[i] %= MOD

        # Sum up all combinations from minLength to maxLength, and take modulo
        return sum(f[min_length:]) % MOD

```

Java Solution

```java
class Solution {
    public int goodBinaryStrings(int minLength, int maxLength, int oneGroup, int zeroGroup) {
        // Define the modulo constant as it is frequently used in the computation
        final int MODULO = (int) 1e9 + 7;

        // Initialize an array to store the number of good binary strings of length i
        int[] goodStringsCount = new int[maxLength + 1];
        // A binary string of length 0 is considered a good string (base case)
        goodStringsCount[0] = 1;

        // Start to fill the array with the number of good binary strings for each length
        for (int i = 1; i <= maxLength; ++i) {
            // If it's possible to have a group of 1's,
            // add the count of the previous length where a 1's group can be appended
            if (i - oneGroup >= 0) {
                goodStringsCount[i] = (goodStringsCount[i] + goodStringsCount[i - oneGroup]) % MODULO;
            }
            // Similarly, if it's possible to have a group of 0's,
            // add the count of the previous length where a 0's group can be appended
            if (i - zeroGroup >= 0) {
                goodStringsCount[i] = (goodStringsCount[i] + goodStringsCount[i - zeroGroup]) % MODULO;
            }
        }

        // Initialize the variable that will store the sum of all good strings
        // within the given range
        int sumGoodStrings = 0;

        // Accumulate the good strings count within the specified length range
        for (int i = minLength; i <= maxLength; ++i) {
            sumGoodStrings = (sumGoodStrings + goodStringsCount[i]) % MODULO;
        }

        // Return the total number of good binary strings
        return sumGoodStrings;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int goodBinaryStrings(int min_length, int max_length, int one_group, int zero_group) {
        const int MOD = 1e9 + 7; // Modulus value for avoiding integer overflow.
        int dp[max_length + 1];   // An array to store the dynamic programming state.
        memset(dp, 0, sizeof dp); // Initialize the dynamic programming array with zeros.
        dp[0] = 1;  // The base case: there's one way to form an empty string.

        // Calculate the number of good binary strings of each length
        for (int i = 1; i <= max_length; ++i) {
            if (i - one_group >= 0) {
                dp[i] = (dp[i] + dp[i - one_group]) % MOD; // Add valid strings ending in a '1' group.
            }
            if (i - zero_group >= 0) {
                dp[i] = (dp[i] + dp[i - zero_group]) % MOD; // Add valid strings ending in a '0' group.
            }
        }

        // Sum up the counts of good strings for all lengths within the given range
        int answer = 0;
        for (int i = min_length; i <= max_length; ++i) {
            answer = (answer + dp[i]) % MOD; // Aggregate the results using modular addition.
        }

        return answer; // Return the total count of good binary strings of valid lengths.
    }
};

```

Typescript Solution

```typescript
function goodBinaryStrings(
    minLength: number,
    maxLength: number,
    oneGroup: number,
    zeroGroup: number,
): number {
    // Modular constant to prevent overflows in calculations
    const MOD: number = 10 ** 9 + 7;

    // Initialize an array to hold counts for each length up to maxLength
    const goodStringCounts: number[] = Array(maxLength + 1).fill(0);

    // Base case: An empty string is considered a good string
    goodStringCounts[0] = 1;

    // Generate counts for good strings of each length up to maxLength
    for (let currentLength = 1; currentLength <= maxLength; ++currentLength) {
        // If the current length is at least as large as the minimum
        // required '1's group, we can add previous count;
        // this implies addition of a '1's group
        if (currentLength >= oneGroup) {
            goodStringCounts[currentLength] += goodStringCounts[currentLength - oneGroup];
        }

        // If the current length is at least as large as the minimum
        // required '0's group, we can add previous count;
        // this implies addition of a '0's group
        if (currentLength >= zeroGroup) {
            goodStringCounts[currentLength] += goodStringCounts[currentLength - zeroGroup];
        }

        // Apply modulus operation to prevent integer overflow
        goodStringCounts[currentLength] %= MOD;
    }

    // Sum all counts from minLength to maxLength (inclusive) to find
    // the total number of good strings within the given range.
    // The result is returned with a modulus operation to keep it within the int bounds.
    return goodStringCounts.slice(minLength).reduce((accumulator, currentCount) => accumulator + currentCount, 0) % MOD;
}

```

Time and Space Complexity

The given Python code defines a method within the `Solution` class that calculates the number of good binary strings with certain properties. The method `goodBinaryStrings` counts the number of binary strings with a specified minimum and maximum length, where each group of consecutive `1`'s is at least `oneGroup` in length and each group of consecutive `0`'s is at least `zeroGroup` in length.

Time Complexity

To analyze the time complexity of the method `goodBinaryStrings`, consider the following points:

- The list `f` is initialized to have a length of `maxLength + 1`.
- There is a single loop that iterates `maxLength` times.
- Within the loop, there are two constant-time conditional checks and arithmetic operations.
- The final summation using `sum(f[minLength:])` runs in `O(n)` time where `n` is `maxLength − minLength + 1`.

Combining these factors, the overall time complexity is as follows:

- Initialization of `f`: `O(maxLength)`
- Loop operations: `O(maxLength)`
- Final summation: `O(maxLength − minLength)`

Since `maxLength` dominates the runtime as it tends to be larger than `maxLength − minLength`, the total time complexity is `O(maxLength)`.

Space Complexity

For the space complexity of the method `goodBinaryStrings`, consider the following points:

- A list `f` is used to store intermediate results and has a size of `maxLength + 1`.

Therefore, the space complexity is `O(maxLength)`, as this is the most significant factor in determining the amount of space used by the algorithm.