2466. Count Ways To Build Good Strings

Medium   Dynamic Programming   Leetcode Link

Problem Description

In this problem, we are given four integers: `zero`, `one`, `low`, and `high`. Our goal is to count how many distinct strings can be constructed with the following rules:

1. We begin with an empty string.
2. At each step, we can either add the character '0' exactly `zero` times or add the character '1' exactly `one` times to the string.
3. This process can be repeated any number of times.

A "good" string is one that is built by these rules and has a length within the `low` and `high` range, inclusive. We are asked to return the number of distinct good strings that can be created subject to these rules. Because the answer could be very large, the final result should be returned modulo `10^9 + 7`.

Intuition

The intuition behind solving this problem lies in understanding that it is a question of combinatorial mathematics but modeled as a depth-first search (DFS) problem. Essentially, from any string of length `i`, it can extend to a string of length `i + zero` by appending `zero` number of '0's or to a string of length `i + one` by appending `one` number of '1's.

The key point is to explore all the possible paths of constructing strings from length `0` to up to `high`. However, if the length of a constructed string ever exceeds `high`, it can no longer be part of a valid solution, so that branch is terminated.

To solve this efficiently, we use a recursive approach with memoization (caching results), which is implemented using the `dfs` function in this case.

The `dfs` function works as follows:

- It takes the current length `i` as the input.
- It checks if the current length is greater than `high`, and if so, it ends the current branch (returns `0`).
- If the length `i` is within the range `[low, high]`, it is counted as a valid string and increments the `ans` by `1`.
- Then it calls itself twice recursively: once with the new length `i + zero`, and once with `i + one`, adding the results to the `ans`.
- The answer is kept within the limit of modulo `10^9 + 7` and returned.

The final result starts with a call to `dfs(0)` since we start with an empty string of length `0`. The function will then recursively compute the total number of distinct good strings. The use of the `@cache` decorator implies that Python will automatically remember the results of the `dfs` function calls with particular arguments, thus optimizing the number of computations by avoiding repeated calculation of the same function call.

Solution Approach

The solution for this problem utilizes a recursive depth-first search (DFS) approach with memoization. Here's a step-by-step implementation breakdown:

1. **Memoization Decorator (`@cache`)**: The function `dfs` is decorated with `@cache`, which is Python's built-in memoization decorator available from the `functools` module. This automatically remembers the results of the `dfs` function when it is called with specific arguments, so if the same length `i` is encountered again, it will not recompute the results. This reduces the complexity significantly as repeated function calls with the same arguments use the cached results.

2. **Recursive `dfs` Function**: This is the function that will be doing the heavy lifting. It receives an integer `i` which represents the current length of the string being constructed.

   - When the function is called, it first checks if `i` exceeds `high`, in which case it returns `0` as this branch cannot produce any good strings.

   - If `i` is within the `low` to `high` range, the function increments `ans` by `1` to account for the valid string of this particular length.

   - The function then makes a recursive call to itself for the next possible lengths, `i + zero` and `i + one`, and adds their results to `ans`. These recursive calls will branch out and cover all possible string lengths that can be created from this point.

3. **Modulo Operation (`% mod`)**: The result of every increment and addition is taken modulo `10^9 + 7` to ensure that the result never exceeds the specified limit. This is important because the number of strings can be quite large, and taking the result modulo `10^9 + 7` keeps the numbers within integer limits. It's also a common requirement in programming problems to prevent integer overflow and to simplify large number arithmetic.

4. **Initialization and Result**: The function `dfs` is initially called with `0` as it starts with an empty string. The result of this call gives us the total count of valid "good" strings that can be constructed within the given parameters.

Here's the key part of the implementation that accomplishes this:

```python
def countGoodStrings(self, low: int, high: int, zero: int, one: int) -> int:
    @cache
    def dfs(i):
        if i > high:
            return 0
        ans = 0
        if low <= i <= high:
            ans += 1
        ans += dfs(i + zero) + dfs(i + one)
        return ans % mod

    mod = 10**9 + 7
    return dfs(0)
```

Through these steps, the algorithm efficiently enumerates and counts all possible "good" strings that can be formed based on the input parameters. The use of memoization in this context optimizes the recursive exploration, making this a practical approach for potentially large input values.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following inputs:

- `low = 1`
- `high = 2`
- `zero = 1`
- `one = 2`

We want to count distinct strings of '0's and '1's that can be constructed using `zero` number of '0's and `one` number of '1's, and the length of the string must be between `low` and `high`.

1. We begin with an empty string `s = ""` and the current length `i = 0`.

2. We call `dfs(i)` with `i = 0`. Since `i` is not greater than `high`, we do not return `0`.

3. Now, `i` is not within `low` and `high`, so we do not increment `ans`.

4. We will call `dfs(i + zero)` which is `dfs(1)` and `dfs(i + one)` which is `dfs(2)`.

5. For `dfs(1)`:

   - `i = 1` is within the range `[1, 2]`, so we increment `ans` by `1`.
   - We make recursive calls `dfs(1 + 1)` and `dfs(1 + 2)` which are `dfs(2)` and `dfs(3)`.
   - `dfs(2)` we already will calculate separately, so let's look at `dfs(3)`:
     - `i = 3` is greater than `high`, so this call returns `0` and does not contribute to `ans`.
6. For `dfs(2)` (from the initial `dfs(0)` call):

   - `i = 2` is within the range `[1, 2]`, so we increment `ans` by `1`.
   - We make recursive calls `dfs(2 + 1)` and `dfs(2 + 2)` which are `dfs(3)` and `dfs(4)`.
   - Both `dfs(3)` and `dfs(4)` are greater than `high` and will return `0`, contributing nothing to `ans`.
7. At this point, we have `ans = 1` for `dfs(1)` and `ans = 1` for `dfs(2)`. Our `dfs(0)` call will thus return `ans = 1 + 1`.

8. The memoization ensures that during our recursive calls, the result of `dfs(2)` was calculated only once, saving computational time.

9. With the modulo operation, the final result would simply be `2 % (10^9 + 7) = 2`.

10. Hence, the function countGoodStrings would return `2` as there are `2` distinct strings ("01", "11") that can be constructed within the given parameters and rules.

Note that in the actual implementation, the function will not recompute `dfs(2)` from both `dfs(0)` and `dfs(1)` due to the memoization decorator, as the result of `dfs(2)` would have already been cached and retrieved from the memo on the second call.

Python Solution

```python
from functools import lru_cache  # Import lru_cache for memoization

class Solution:
    def count_good_strings(self, low: int, high: int, zero: int, one: int) -> int:
        # The mod value to ensure the results stay within the bounds of 10^9 + 7
        mod = 10**9 + 7

        # Use lru_cache to memoize results of recursive calls
        @lru_cache(None)
        def dfs(current_value):
            """
            Depth-first search to calculate number of good strings using recursion.

            :param current_value: The current integer value being built
            :return: The number of good strings from the current_value to high
            """
            # Base case: if current_value exceeds the 'high' value, no further strings can be considered
            if current_value > high:
                return 0

            # Initialize answer for this state
            ans = 0

            # If the current value is within the range [low, high], count it as a good string
            if low <= current_value <= high:
                ans += 1

            # Recursively count good strings adding zero and one to the current value
            ans += dfs(current_value + zero)
            ans += dfs(current_value + one)

            # Return the total count modulo 10^9 + 7
            return ans % mod

        # Begin recursion with 0 as the starting value
        return dfs(0)

# Example usage:
# If you have bounds and increment values, you can create an instance of Solution and call count_good_strings
# sol = Solution()
# result = sol.count_good_strings(low, high, zero, one)

```

Java Solution

```java
class Solution {
    // Constant for the modulus to be used for result to prevent overflow
    private static final int MOD = (int) 1e9 + 7;

    // Cache for already computed results to avoid repeated calculations
    private int[] memoization;

    // Variables to store the lower and upper bounds of the string length
    private int lowerBound;
    private int upperBound;

    // Variables to store the value to be added when encountering a '0' or '1' in the string
    private int valueZero;
    private int valueOne;

    public int countGoodStrings(int low, int high, int zero, int one) {
        // Initialize the cache array with a size of 'high + 1' and fill it with -1
        // indicating that no calculations have been made for any index
        memoization = new int[high + 1];
        Arrays.fill(memoization, -1);

        // Assign the provided bounds and values to the respective class fields
        lowerBound = low;
        upperBound = high;
        valueZero = zero;
        valueOne = one;

        // Begin depth-first search from the starting point '0'
        return dfs(0);
    }

    // Helper method that employs depth-first search to compute the good strings
    private int dfs(int index) {
        // If the current index is beyond the upper bound, return 0 as it cannot form a valid string
        if (index > upperBound) {
            return 0;
        }
        // If a result for the current index has already been computed (memoized), return it
        if (memoization[index] != -1) {
            return memoization[index];
        }

        // Initialize the answer for this index
        long ans = 0;

        // If the current index is within the bounds, count it as one valid string
        if (index >= lowerBound && index <= upperBound) {
            ans++;
        }

        // Use recurrence to count additional good strings by adding valueZero and valueOne to index recursively
        ans += dfs(index + valueZero) + dfs(index + valueOne);

        // Take modulus to prevent overflow
        ans %= MOD;

        // Cache the computed result for the current index before returning
        memoization[index] = (int) ans;
        return memoization[index];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Declare the modulus as a constant expression since it does not change.
    static constexpr int MOD = 1e9 + 7;

    // Count the good strings with given constraints using memoization.
    int CountGoodStrings(int low, int high, int zero, int one) {
        // Use a vector for memoization, initialized with -1.
        vector<int> memo(high + 1, -1);

        // Define a lambda function for DFS (Depth-First Search) with memoization.
        function<int(int)> DFS = [&](int i) -> int {
            // Base case: if the current value is greater than 'high', no good strings can be formed.
            if (i > high) return 0;

            // If the current number of good strings has been calculated before, return the stored result.
            if (memo[i] != -1) return memo[i];

            // Initialize count starting with 1 if 'i' is within range [low, high].
            long count = (i >= low && i <= high) ? 1 : 0;

            // Recursively count the good strings by adding 'zero' and 'one' to the current value 'i'.
            count += DFS(i + zero) + DFS(i + one);

            // Ensure the count does not exceed the MOD.
            count %= MOD;

            // Store the result in the memoization vector and return it.
            memo[i] = count;
            return count;
        };

        // Start the DFS from 0.
        return DFS(0);
    }
};

```

Typescript Solution

```typescript
// Define the modulus as a constant since it does not change.
const MOD = 1e9 + 7;

// Memoization table, using a Map to associate indices with their counts
let memo: Map<number, number> = new Map();

// Count the good strings with given constraints using memoization
function countGoodStrings(low: number, high: number, zero: number, one: number): number {
    // Clear the memoization table before a new computation
    memo.clear();

    // Helper function for DFS (Depth-First Search) with memoization
    function dfs(current: number): number {
        // Base case: if the current value is greater than 'high', no good strings can be formed
        if (current > high) return 0;

        // If the current number of good strings has been calculated before, return the stored result
        if (memo.has(current)) return memo.get(current)!;

        // Initialize count starting with 1 if 'current' is within the range [low, high]
        let count: number = (current >= low && current <= high) ? 1 : 0;

        // Recursively count the good strings by adding 'zero' and 'one' to the current value 'current'
        count = (count + dfs(current + zero) + dfs(current + one)) % MOD;

        // Store the result in the memoization table
        memo.set(current, count);

        return count;
    }

    // Start the DFS from 0 and return the total count
    return dfs(0);
}

```

Time and Space Complexity

The time complexity and space complexity analysis of the given code are as follows:

Time Complexity

The time complexity of the `dfs` function primarily depends on the number of unique states it will visit during its execution. In this scenario, each state is represented by a value of `i` that is checked against `low` and `high`. Since we increment `i` by either `zero` or `one`, and we use memoization (`@cache`), each state is computed only once.

The maximum number of unique states that can be visited can be roughly estimated as `(high - low) / min(zero, one)`, because we're incrementally increasing `i` starting from `0` to `high` in steps of `zero` or `one`. However, we do perform extra checks when `i` exceeds `high`, so some overhead is present.

Therefore, the time complexity can be considered as `O((high - low) / min(zero, one))`.

Space Complexity

The space complexity involves the stack space used by the recursive DFS calls and the space used by the memoization cache. Since we cache every unique state and each state calls `dfs` twice (once with `i + zero` and once with `i + one`), the depth of the recursion tree can go up to `high / min(zero, one)` in the worst case.

Therefore, the space complexity for the recursive stack is `O(high / min(zero, one))`. Additionally, the cache will store each unique state, contributing a space complexity that is also in the order of `O(high / min(zero, one))`.

Considering both the stack and the caching, the total space complexity can also be approximated as `O(high / min(zero, one))`.