1922. Count Good Numbers

Medium Recursion Math

Problem Description

The problem requires us to calculate the total number of `good` digit strings of a given length `n`. A `good` digit string is one where:

- The digits at even indices (0, 2, 4, ...) are even numbers (0, 2, 4, 6, 8).
- The digits at odd indices (1, 3, 5, ...) are prime numbers (2, 3, 5, 7).

The string is `0-indexed`, which means the indexing starts from 0. We are tasked with returning the count of such good strings modulo `10^9 + 7` to keep the number manageable since the count can be very large.

Intuition

The intuition behind the solution is to understand that since the digits at even and odd indices have separate and independent constraints, they can be handled separately. For digits at even positions, there are 5 choices (0, 2, 4, 6, 8), and for digits at odd positions, there are 4 choices (2, 3, 5, 7).

Given a string of length `n`, half of the digits will be at even indices and half at odd indices (for odd values of `n`, there will be one more digit at an even index since counting starts at 0). Hence, for strings of even length, we have a straightforward calculation:

- The number of ways to fill even indices = 5^(n/2).
- The number of ways to fill odd indices = 4^(n/2).

For odd lengths of `n`, there is one extra even index, hence:

- The number of ways to fill even indices = 5^((n+1)/2).
- The number of ways to fill odd indices = 4^(n/2).

To find the total combinations, we multiply the number of ways to fill even indices with the number of ways to fill odd indices.

Since the calculations can result in very large numbers, we use modulo arithmetic to avoid overflow and keep the results within integer limits. The modulo used is `10^9 + 7`, which is a large prime number, that helps manage big numbers in competitive programming problems.

In our Python code, the `myPow` function is a custom power function to calculate x^n under modulo. It uses the concept of binary exponentiation for efficient computation. It works by squaring the base `x` and dividing the power `n` by 2 at each iteration, and multiplying the result by `x` when `n` is odd. This process is repeated until `n` reaches 0.

Finally, the return statement calculates the total number of good digit strings by multiplying `5^((n+1)/2)` by `4^(n/2)` using our custom power function and applies the modulo to get the answer.

Solution Approach

The solution to this problem mainly involves number theory and the efficient calculation of large powers under a modular system. Here's a breakdown of how it's implemented:

- **Binary Exponentiation Algorithm for Modular Exponentiation:** The `myPow` function in the provided Python code is an implementation of the binary exponentiation algorithm, which calculates `x^n (mod m)` efficiently. The process involves iterating over the bits of `n`. For each bit that is set to 1 in the binary representation of `n`, we multiply the current result by `x` to the corresponding power of 2 and take the result modulo `m`. After each iteration, `x` is squared and `n` is shifted right by one bit (essentially divided by 2). This is done until `n` is reduced to 0.

  Pseudocode for the `myPow` function is as follows:

```
def myPow(x, n, mod):
    result = 1
    while n > 0:
        if n % 2 == 1:
            result = (result * x) % mod
        x = (x * x) % mod
        n = n // 2
    return result
```

- **Separation of Even and Odd Indices Constraints:** The elegance of the solution lies in recognizing that the constraints for even and odd indices are separate and can thus be dealt with independently. Since at even indices only even digits can be placed, and at odd indices only prime digits can be placed, the problem reduces to two separate combinations problems.
- **Multiplication under Modulo:** The solution multiplies the result of the powers for even and odd indices while always taking the modulo after each operation to prevent integer overflow. In Python, this is simply done with `% mod` after each multiplication.
- **Handling Odd Lengths:** The implementation accounts for odd lengths `n` by computing `5^((n+1)/2)` for even indices instead of `5^(n/2)`. This ensures that the extra even index in the case of an odd-length digit string is considered.
- **Final Computation:** The total number of good digit strings is derived by the formula:

```
total_good_strings = (myPow(5, (n + 1) // 2, mod) * myPow(4, n // 2, mod)) % mod
```

  This formula ensures that all digits at even indices have been accounted for with `5` possible values each and all digits at odd indices are accounted for with `4` possible values each.

The overall time complexity for this solution is `O(log n)` due to the binary exponentiation, while the space complexity is `O(1)` as it uses a constant amount of extra space.

Example Walkthrough

Let's consider a small example with `n = 3`, which refers to a good digit string of length 3.

For `n = 3`, an extra even index exists since we start from index 0. Therefore, we should have 1 odd index (index 1) and 2 even indices (indices 0 and 2).

The number of choices for the digit at even indices is 5 (0, 2, 4, 6, 8) and the number of choices for the digit at odd indices is 4 (2, 3, 5, 7).

Now, let's calculate the number of good digit strings:

1. For even indices, there will be 2 digits, so the number of ways to fill them is 5 choices raised to the power of the number of even indices: `5^2 = 25`.
2. For the odd index, there will be a single digit, so the number of ways to fill it is 4 choices raised to the power of the number of odd indices: `4^1 = 4`.

To get the total number of good strings, we multiply the number of ways to place digits at even indices with the number of ways to place digits at odd indices:

Total good strings for `n = 3` = 25 (even choices) * 4 (odd choice) = 100.

Now, since we must return this number modulo `10^9 + 7`, we do this simple calculation:

100 % (10^9 + 7) = 100.

So, the total number of good digit strings of length `n = 3` is 100.

Using the `myPow` function and the formula provided, we did the same computation programmatically:

- For the 2 even indices, `myPow(5, (3+1)//2, 10^9+7)` is called, which should return 25.
- For the 1 odd index, `myPow(4, 3//2, 10^9+7)` is called, which should return 4.

Finally, we would have `(25 * 4) % (10^9+7)`, which gives us the same result, 100, confirming that our solution approach is correct.

Solution Implementation

Python

```python
class Solution:
    def countGoodNumbers(self, n: int) -> int:
        MODULO = 10**9 + 7

        # Define the power function to calculate (x^n) % MODULO using binary exponentiation.
        def my_pow(base, exponent):
            result = 1
            while exponent > 0:
                # If exponent is odd, multiply the result with base.
                if exponent % 2 == 1:
                    result = (result * base) % MODULO
                # Square the base and reduce the exponent by half.
                base = (base * base) % MODULO
                exponent //= 2
            return result

        # Number of primes at odd positions is 5 (2,3,5,7). Hence, we use 5 as the base.
        # Number of evens at even positions is 4 (0,2,4,6,8). Hence, we use 4 as the base. Even positions are considered 0-indexed.
        # The +1 is needed when n is odd, to calculate 5 raised to the (n//2 + 1).
        return (my_pow(5, (n + 1) // 2) * my_pow(4, n // 2)) % MODULO
```

Java

```java
class Solution {
    // Define the modulus value as a constant for mod operations
    private static final int MOD = 1000000007;

    // This method calculates the count of good numbers
    public int countGoodNumbers(long n) {
        // Good numbers are created by even indices with 5 choices (0 to 4)
        // and odd indices with 4 choices (0, 2, 4, 6, 8).
        // We use exponentiation by squaring to compute the large powers efficiently
        // We need to use (n+1)/2 for the power of 5 if n is odd.
        // Similarly, we use n/2 for the power of 4, no matter if n is even or odd.
        return (int) (myPow(5, (n + 1) / 2) * myPow(4, n / 2) % MOD);
    }

    // Helper method for fast exponentiation
    private long myPow(long base, long exponent) {
        long result = 1; // Start from the identity value
        while (exponent != 0) {
            if ((exponent & 1) == 1) {
                // If the current exponent bit is 1, multiply the result by base
                result = (result * base) % MOD;
            }
            // Square base and move to the next bit of the exponent
            base = (base * base) % MOD;
            exponent >>= 1; // Right shift exponent by 1 (equivalent to dividing by 2)
        }
        return result;
    }
}
```

C++

```cpp
int kMod = 1000000007; // Define the modulo constant for large number arithmetic

class Solution {
public:
    // Function to count the number of 'good' numbers of length n
    int countGoodNumbers(long long n) {
        // Calculate the product of 5^(n/2 + 1) and 4^(n/2) modulo kMod
        // (n + 1) / 2 accounts for the case when n is odd and each part is good
        // n / 2 ensures we deal with even number of digits for 4s
        return static_cast<int>(myPow(5, (n + 1) / 2) * myPow(4, n / 2)) % kMod);
    }

private:
    // Helper function to compute x^n % kMod efficiently
    long long myPow(long long x, long long n) {
        long long res = 1; // Initialize result to 1
        while (n > 0) {    // Loop until n becomes 0
            if (n & 1) {   // If n is odd, multiply res with x
                res = res * x % kMod;
            }
            x = x * x % kMod; // Change x to x^2 for next iteration
            n >>= 1;          // Divide n by 2
        }
        return res; // Return the result of x^n
    }
};
```

TypeScript

```typescript
const kMod: number = 1000000007; // Define the modulo constant for large number arithmetic

// Function to count the number of 'good' numbers of length n
function countGoodNumbers(n: bigint): number {
    // Calculate the product of 5^(ceil(n/2)) and 4^(floor(n/2)) modulo kMod
    // Ceil is obtained by (n + 1n) / 2n which considers the case when n is odd
    // Floor is n / 2n which ensures we deal with an even number of digits for 4s
    return Number((myPow(BigInt(5), (n + 1n) / 2n) * myPow(BigInt(4), n / 2n)) % BigInt(kMod));
}

// Helper function to compute x^n % kMod efficiently
function myPow(x: bigint, n: bigint): bigint {
    let res: bigint = BigInt(1); // Initialize the result to 1
    while (n > 0n) { // Loop until n becomes 0
        if (n & 1n) { // If n is odd, multiply the result with x
            res = (res * x) % BigInt(kMod);
        }
        x = (x * x) % BigInt(kMod); // Change x to x^2 for the next iteration
        n >>= 1n; // Divide n by 2 by bitwise shift
    }
    return res; // Return the result of x^n
}
```

```python
class Solution:
    def countGoodNumbers(self, n: int) -> int:
        MODULO = 10**9 + 7

        # Define the power function to calculate (x^n) % MODULO using binary exponentiation.
        def my_pow(base, exponent):
            result = 1
            while exponent > 0:
                # If exponent is odd, multiply the result with base.
                if exponent % 2 == 1:
                    result = (result * base) % MODULO
                # Square the base and reduce the exponent by half.
                base = (base * base) % MODULO
                exponent //= 2
            return result

        # Number of primes at odd positions is 5 (2,3,5,7). Hence, we use 5 as the base.
        # Number of evens at even positions is 4 (0,2,4,6,8). Hence, we use 4 as the base. Even positions are considered 0-indexed.
        # The +1 is needed when n is odd, to calculate 5 raised to the (n//2 + 1).
        return (my_pow(5, (n + 1) // 2) * my_pow(4, n // 2)) % MODULO
```

Time and Space Complexity

The time complexity of the provided code is `O(log n)`. This is because the key operation in the function is `myPow`, which uses a binary exponentiation algorithm that computes `x^n` by continually squaring `x` and reducing `n` by half at each step. As such, the number of steps required is proportional to the number of bits in `n`, which is `O(log n)`.

The space complexity of the code is `O(1)`, meaning it is constant. The `myPow` function uses only a fixed number of integer variables (`res` and `x`) that do not depend on the input size, and the computation is done in place without the need for additional space that scales with `n`.