788. Rotated Digits

Medium · Math · Dynamic Programming · Leetcode Link

Problem Description

The problem states that an integer `x` is considered *good* if it can be turned into a valid number by rotating each digit by 180 degrees. The rotation should transform the number into a different one, meaning we can't just leave any digit unchanged.

A digit is valid after rotation if it's still a number. There are specific digits that transform into themselves or others:

- `0`, `1`, and `8` rotate to themselves.
- `2` and `5` can rotate to each other.
- `6` and `9` also rotate to each other.

Digits that are not in the above list become invalid after rotation.

Our task is to find the total number of *good* integers within the range from 1 up to and including `n`.

Intuition

To solve this problem, we approach it through recursive depth-first search (DFS) with caching (memoization) to improve performance. The idea is to iterate through each digit position, determining whether placing a certain digit there would contribute to the count of good numbers.

We define a recursive function `dfs` that considers the following arguments:

- `pos`: current position in the number we're inspecting.
- `ok`: a boolean indicating whether we've already placed a digit that becomes different after rotation (making the current number formation a good number).
- `limit`: a boolean that tells us if we are limited by the original number `n` in choosing our digits.

The base case of the recursion is when `pos <= 0`, which means we've checked all the digit positions. If `ok` is True at this point, we've formed a good number, and we can increment our count.

During each recursive call, we try placing each digit (0-9) in the current position and make further recursive calls to fill the remaining positions. There are conditions though:

- If the digit is 0, 1, or 8, we can continue the recursion without changing the `ok` status, as these digits do not alter the number's goodness by themselves.
- If the digit is 2, 5, 6, or 9, we make the recursive call with `ok` set to True, since these digits ensure the number's goodness.

To avoid unnecessary calculations, we memoize the results of the recursive calls with different parameter combinations.

We structure the number `n` into an array `a`, where each element represents a digit in its corresponding place. Then we walk through the number's digits from the most significant digit to the least significant digit (right to left), calling our recursive function to sum up the counts of good numbers according to the above-defined rules.

The solution's effectiveness comes from carefully analyzing the properties of good numbers and utilizing recursive calls with caching to systematically count the valid numbers without enumerating them all.

Solution Approach

The provided Python solution makes use of recursion, dynamic programming through memoization, and digit-by-digit analysis to solve the problem of finding good integers within a given range.

Firstly, a recursive helper function `dfs` is defined. This function uses the following parameters:

- `pos`: The current digit position we are filling in the potentially good integer.
- `ok`: A boolean flag indicating whether we've already included at least one digit in the number that makes it 'good' upon rotation, such as `2`, `5`, `6`, or `9`.
- `limit`: A flag that indicates whether the current digit being considered is constrained by the original integer `n`'s corresponding digit.

The `dfs` function utilizes memoization to cache and reuse the results of the recursive calls, which prevents recalculating the same scenarios—this is done using the `@cache` decorator, which caches the results of the expensive function calls.

The recursive calls made by the `dfs` function follow these rules:

- If `pos` is 0 or negative, it means we've looked at all digit positions, and we check the flag `ok`. If `ok` is true, we return 1 as we've found a good number; otherwise, we return 0 as the number does not meet the criterion.
- Otherwise, we iterate over all possible digits from 0 to 9 (or up to the digit in the original number `n` if the `limit` is true), placing each digit in the current position and making a recursive call to `dfs` to decide the next position.
- The current state of `ok` is passed on unless we're placing a digit that contributes to the goodness of the number (2, 5, 6, or 9), in which case `ok` is set to true for all deeper recursive calls.
- The `limit` is only maintained if the digit being placed is equivalent to the corresponding digit in `n`.

To kick off the recursion, the original number `n` is broken down into its constituent digits and stored in an array `a`. This allows the function to easily compare each possible combination against `n`'s corresponding digits to ensure validity.

Finally, the `dfs` function is called with the array length, `ok` set to 0 (as initially, we haven't placed any digits that would classify the integer as good), and `limit` set to True (because at the start, we are limited to `n`). The return value from this call gives the total number of good integers within the range specified.

This solution effectively parses the problem domain with a depth-first traversal, leveraging the constraints to prune the search space and using memoization to optimize the recursive exploration of possibilities, resulting in an efficient algorithm to find all good integers up to `n`.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we want to find all good integers up to `n = 23`. Using the definitions and rules from the solution approach, we would proceed as follows:

1. We start by setting up our `dfs` function and decomposing our target integer `n = 23` into an array `a = [2, 3]`.

2. We then call `dfs(pos=len(a), ok=False, limit=True)` to start the recursion, which implies we are starting from the digit in the tens place (as array indices are 0-based).

3. At the first level of recursion (`pos = 2`), we have not yet placed any digit, so `ok` is `False`, and `limit` is `True`. We iterate over possible digits from 0 to 2 for the tens place since 2 is the tens digit of `n` (due to `limit=True`).

   - When we place a `0` or `1`, we keep `ok` as `False` since these digits don't contribute to a number's goodness. We also set `limit` to `False` for the next digit since we are already below `n`.
   - When we place a `2` (same as in `n`), we keep `limit` as `True` for the next digit to make sure we don't go over `n`. Here, we set `ok` to `True` because `2` contributes to the number's goodness upon rotation (2 rotates to 5).
4. Now, for the second level of recursion (`pos = 1`), we are checking the digit in the ones place. The `limit` flag instructs us whether we can consider digits up to `3` or all digits 0-9.

   - If we have placed a `0` or `1` in the tens place, we can consider all digits here, and if we place a `2`, `5`, `6`, or `9`, we set `ok` to `True`.
   - If we had placed a `2` in the tens place, we can go up to `3` in the ones place.
5. Each time we reach `pos = 0` (the base case), we check if `ok` is `True`, meaning we have made the number good. If so, we return `1`; otherwise, we return `0`.

6. Finally, after considering all possibilities for each digit, the sum of all recursive call returns will give us the count of good integers. In our case, we would find the valid good numbers to be `2`, `5`, `6`, `8`, `9`, `20`, `21`, `22`, and `25`. Numbers like `11`, `12`, `15`, `18`, and `19` are not counted because they are beyond `n`.

Thus, we've walked through this approach to find there are 9 good integers between 1 and 23.

Python Solution

```python
from functools import lru_cache

class Solution:
    def rotatedDigits(self, N: int) -> int:
        # A helper function that uses depth-first search to determine
        # the valid numbers according to the problem constraints.
        @lru_cache(maxsize=None)
        def dfs(position, is_good, is_limit):
            # Base case: if position is zero, return 1 if 'is_good' is True,
            # because we've found a valid number, and 0 otherwise.
            if position == 0:
                return 1 if is_good else 0

            # If 'is_limit' is True, we can only go up to 'digits[position]'
            # otherwise, we can use digits 0-9.
            upper_bound = digits[position] if is_limit else 9

            # This will accumulate the number of valid numbers found.
            valid_numbers = 0

            # Iterate through the possible digits at this position.
            for digit in range(upper_bound + 1):
                # Digits 0, 1, and 8 do not change the number's validity
                if digit in (0, 1, 8):
                    valid_numbers += dfs(position - 1, is_good, is_limit and digit == upper_bound)
                # Digits 2, 5, 6, and 9 make the number valid if it wasn't already
                elif digit in (2, 5, 6, 9):
                    valid_numbers += dfs(position - 1, 1, is_limit and digit == upper_bound)

            return valid_numbers

        # Convert the number to a list of its digits in reversed order.
        # Allocate enough space for digits (here it's 6 for some reason,
        # but we could dynamically determine this based on the size of N).
        digits = [0] * 6
        length = 1
        while N:
            digits[length] = N % 10
            N //= 10
            length += 1

        # Call the helper function 'dfs' with the current length of the number,
        # the initial state of 'is_good' as False, and 'is_limit' as True,
        # because we are starting with the actual limit of N.
        return dfs(length, 0, True)

```

Java Solution

```java
class Solution {

    // Member variable to hold digits of the number
    private int[] digits = new int[6];

    // DP cache to store intermediate results, initialized to -1 indicating uncomputed states
    private int[][] dpCache = new int[6][2];

    public int rotatedDigits(int n) {
        int length = 0; // Will keep track of the number of digits

        // Initialize the dpCache array with -1, except where it's been computed already
        for (int[] row : dpCache) {
            Arrays.fill(row, -1);
        }

        // Backfill the array 'digits' with individual digits of the number n
        while (n > 0) {
            digits[++length] = n % 10; // Store each digit
            n /= 10; // Reduce n by a factor of 10
        }

        // Start the depth-first search from most significant digit with 'ok' as false and limit as true
        return depthFirstSearch(length, 0, true);
    }

    private int depthFirstSearch(int position, int countValid, boolean limit) {
        if (position <= 0) {
            // Base case: when all positions are processed, check if we
            // have at least one valid digit (2, 5, 6, or 9)
            return countValid;
        }

        // Check the DP cache to avoid re-computation unless we're limited by the current value
        if (!limit && dpCache[position][countValid] != -1) {
            return dpCache[position][countValid];
        }

        // Calculate the upper bound for this digit. If we have a limit, we cannot exceed the given digit
        int upperBound = limit ? digits[position] : 9;
        int ans = 0; // To store the number of valid numbers

        for (int i = 0; i <= upperBound; ++i) {
            if (i == 0 || i == 1 || i == 8) {
                // Digits 0, 1, and 8 are valid but not counted towards 'countValid' flag
                ans += depthFirstSearch(position - 1, countValid, limit && i == upperBound);
            }
            if (i == 2 || i == 5 || i == 6 || i == 9) {
                // Digits 2, 5, 6, and 9 are valid and counted towards 'countValid' flag
                ans += depthFirstSearch(position - 1, 1, limit && i == upperBound);
            }
        }

        // Cache the computed value if there was no limit
        if (!limit) {
            dpCache[position][countValid] = ans;
        }
        return ans; // Return the calculated number of valid rotated digits
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int digits[6];        // Array to store the digits of the number n
    int memo[6][2];       // Memoization table for dynamic programming

    // Main function that initiates the process and returns the count of good numbers
    int rotatedDigits(int n) {
        memset(memo, -1, sizeof(memo)); // Initialize the memoization table with -1
        int length = 0;                 // Used to store the number of digits in n
        // Extract the digits of n and store them in reverse order in the array 'digits'
        while (n) {
            digits[++length] = n % 10;
            n /= 10;
        }
        // Start the DFS from the most significant digit with the condition that it has not
        // encountered a different good digit yet (ok = 0) and that it is the initial limit
        return depthFirstSearch(length, 0, true);
    }

    // Recursive function to count the good numbers
    int depthFirstSearch(int position, int hasDiffGoodDigit, bool isLimit) {
        // If we have no more digits to process and we have encountered a good digit different
        // from 0, 1, 8, return 1 as this is a good number
        if (position == 0) {
            return hasDiffGoodDigit;
        }
        // Use memoization to avoid recalculating states we have already seen
        if (!isLimit && memo[position][hasDiffGoodDigit] != -1) {
            return memo[position][hasDiffGoodDigit];
        }
        // Determine the upper bound for the digit that we can place at the current position
        int upperLimit = isLimit ? digits[position] : 9;
        int count = 0; // Initialize count of good numbers
        // Iterate through possible digits
        for (int i = 0; i <= upperLimit; ++i) {
            // If the digit is 0, 1, or 8, we continue the search without changing the state
            if (i == 0 || i == 1 || i == 8) {
                count += depthFirstSearch(position - 1, hasDiffGoodDigit, isLimit && i == upperLimit);
            }
            // If the digit is 2, 5, 6, or 9, we continue the search and mark the state as having
            // encountered a different good digit
            if (i == 2 || i == 5 || i == 6 || i == 9) {
                count += depthFirstSearch(position - 1, 1, isLimit && i == upperLimit);
            }
        }
        // If we are not at the limit, update the memoization table
        if (!isLimit) {
            memo[position][hasDiffGoodDigit] = count;
        }
        // Return the count of good numbers for the current digit position and state
        return count;
    }
};

```

Typescript Solution

```typescript
// TypeScript does not require specifying the size of array beforehand like C++
let digits: number[] = [];
// Initialize memoization array with undefined values since TypeScript doesn't have memset
let memo: number[][] = Array.from({ length: 6 }, () => Array(2).fill(undefined));

// Function to initialize the process and return the count of good numbers
function rotatedDigits(n: number): number {
    // Reset memoization array
    memo.forEach(row => row.fill(undefined));
    let length = 0;
    // Extract the digits of n and store them in reverse order in 'digits' array
    while (n > 0) {
        digits[length++] = n % 10;
        n = Math.floor(n / 10);
    }
    // Start the DFS from the most significant digit
    return depthFirstSearch(length, 0, true);
}

// Recursive function to count the good numbers
function depthFirstSearch(position: number, hasDiffGoodDigit: number, isLimit: boolean): number {
    // Base case: if no digits left and a different good digit has been found
    if (position === 0) {
        return hasDiffGoodDigit;
    }
    // Check memo table to possibly use previously calculated result
    if (!isLimit && memo[position][hasDiffGoodDigit] !== undefined) {
        return memo[position][hasDiffGoodDigit];
    }
    let upperLimit = isLimit ? digits[position - 1] : 9; // fixing off-by-one error from original code
    let count = 0;
    // Iterate through all possible digits for the current position
    for (let i = 0; i <= upperLimit; i++) {
        // Good digits that are the same when rotated
        if (i === 0 || i === 1 || i === 8) {
            count += depthFirstSearch(position - 1, hasDiffGoodDigit, isLimit && i === upperLimit);
        }
        // Digits that become different good digits when rotated
        else if (i === 2 || i === 5 || i === 6 || i === 9) {
            count += depthFirstSearch(position - 1, 1, isLimit && i === upperLimit);
        }
    }
    // Memoize result if not at a limit
    if (!isLimit) {
        memo[position][hasDiffGoodDigit] = count;
    }
    return count;
}

```

Time and Space Complexity

The given code defines a function `rotatedDigits` which takes an integer `n` and returns the count of numbers less than or equal to `n` where the digits 2, 5, 6, or 9 appear at least once (making the number 'good'), and the digits 3, 4, or 7 do not appear at all (since they cannot be rotated to a valid number). It does not include numbers that remain the same after being rotated.

Time Complexity

The time complexity of the modified code is `O(logN * 4^logN)` which simplifies to `O(N^2)` where `N` is the number of digits in the input number `n`. This is because there are `logN` digits in `n`, and for each digit position, we iterate over up to 4 possible 'good' digits. There is a factor of `logN` for each digit position since there are that many recursive calls at each level, considering that limit is set to `True` only when `i == up` which means the next function call will honour the limit created by the previous digit.

Using memoization with `@cache`, we avoid repetitive calculations for each unique combination of the position of the digit, the flag whether we have encountered a 'good' digit, and whether we are limited by the original number's digit at this position (`up`) which leads to pruning the search space significantly.

Space Complexity

The space complexity of the code is `O(logN * 2 * logN)` which simplifies to `O((logN)^2)`.

This includes space for:

- The memoization cache which could potentially store all states of the function arguments (`pos`, `ok`, `limit`).
- The array `a` of length `6`, indicating the input number is decomposed into up to `6` digits, accounting for integers up to a maximum of `999999`. However, this `6` is a constant and does not scale with the input, so it does not affect the time complexity.

Therefore, the space used by the recursion stack and the memoization dictionary depends on the number of different states the `dfs` function can be in, which is influenced by the number of digits `logN` (representing different positions) and the boolean flags `ok` and `limit`, leading to the complexity of `O((logN)^2)`.