2457. Minimum Addition to Make Integer Beautiful

Medium Greedy Math

Leetcode Link

Problem Description

In this problem, we are given two positive integers: `n`, which represents the number we're starting with, and `target`, a threshold value for the sum of digits of a number. A number is considered **beautiful** if the sum of its individual digits is less than or equal to the given `target`. The task is to find the minimum non-negative integer `x` that, when added to `n`, results in a beautiful number. It's guaranteed that there is always a way to turn `n` into a beautiful by adding such an `x`.

Intuition

The intuitive approach to solve this problem involves incrementing `n` gradually until we find a number that meets the condition of being beautiful. However, incrementing `n` by 1 and checking each time is inefficient. A more effective strategy should skip over likely candidates that clearly would not meet the condition.

Observing that every time you reach a digit `9` in the process of incrementing `n`, the next increment will carry over and turn that `9` into a `0`, increasing the next digit to the left by one, which is significant. Rather than dealing with individual increments, we want to make more substantial jumps to avoid unnecessary checks.

For instance, if `n` ends in a series of 9s (like 299), we know the next beautiful number will be at least 300. The solution finds the least significant non-9 digit, increments that digit by 1, turns all the digits to its right into 0, and computes the difference from the original number `n`. That difference is our `x`, which when added to `n` will bypass the unnecessary candidates and get us closer to a beautiful number.

This effective jump method ensures that we do not perform redundant checks, and that we will reach the solution in far fewer steps than incrementing by 1 each time. The function `f(x)` in the provided solution calculates the sum of the digits of `x`, which is used for determining if the current number `n + x` is beautiful or if we need to make another jump.

Solution Approach

The implementation of the solution can be broken down into the following parts:

1. **Sum of Digits Function (`f(x)`):** This function calculates the sum of the digits of a given integer `x`. It does so iteratively by taking `x % 10` to get the least significant digit, adding it to the sum `y`, and then using `x //= 10` to remove the least significant digit from `x`. This loop continues until `x` is reduced to zero.

2. **Main Logic:** In the `makeIntegerBeautiful` method of the `Solution` class, `x` is initialized to `0`. This `x` is the additional amount we will need to add to `n` to make it beautiful. The `while` loop checks whether `f(n + x) > target`. As long as this condition is true, the current `n + x` is not beautiful, and we need to increase `x`.

   a. **Finding the Next Candidate:** Within the loop, `y = n + x` is set to the current number we're evaluating. We are looking for the least significant non-zero digit in `y`. During this, the code progressively divides `y` by 10 if the current least significant digit is `0`. Simultaneously, `p` is multiplied by 10, effectively keeping track of the place value of the first non-zero digit from the right.

   b. **Jumping to the Next Significant Candidate:** Once a non-zero least significant digit is found, the code calculates `(y // 10 + 1) * p - n`. This operation turns the least significant non-zero digit into `8` and increments the next digit to the left by `1`, effectively skipping all the intermediary numbers that would also not qualify as beautiful.

3. **Loop Exit and Return Value:** Eventually, the `while` loop will exit when `f(n + x) <= target`, which means `n + x` is now beautiful. At this point, the function returns `x` which is the minimum increment needed to convert `n` into a beautiful number.

This algorithm effectively uses a greedy approach to jump through potential candidates for beautiful numbers. By identifying the digit increments that lead to the most significant change, the algorithm minimizes the number of operations needed to find the correct value of `x`. It is a more optimal solution than incrementing by 1 and checking each number individually.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have `n = 28` and `target = 10`. We're tasked with finding the minimum non-negative integer `x` that can be added to `n` to result in a number where the sum of its digits is less than or equal to `target`.

1. First, apply the **Sum of Digits Function** `f(x)` to `n` to see if it's already beautiful. The sum of digits of `28` is `2 + 8 = 10`, which equals the target. Since we need the sum to be *less than or equal to* the target, `n` is already beautiful, so `x = 0`.

   In this case, we get lucky on the first try, but let's consider a slightly altered example to show the rest of the process: Let's say we have `n = 29` and the same target of `10`. The sum of digits of `29` is `2 + 9 = 11`, which is greater than the target.

2. Now, commence the **Main Logic**. Initialize `x` to `0`, and check if `f(n + x) > target`. Indeed, `f(29 + 0) = f(29) = 11`, which is greater than `10`.

3. Start the `while` loop and set `y = n + x`, which is `29` currently. Then, search for the least significant non-9 digit. The least significant digit is `9`, and it's also the non-9 digit we're looking for as `n` has only two digits. The place value `p` corresponding to this digit is `10`.

4. Perform a jump as per the **Next Significant Candidate**: Calculate `(y // 10 + 1) * p - n`, which simplifies to `(2 + 1) * 10 - 29`. This equals `30 - 29 = 1`. So `x = 1`.

5. After the iteration, `n + x` is `29 + 1 = 30`. Using function `f(x)`, we find that `f(30) = 3 + 0 = 3`, which is less than `target`. Since `f(n + x) <= target`, the `while` loop exits.

6. Return `x`, which is `1`, as the minimum increment needed to make `29` beautiful with respect to the target `10`.

Putting all that through the given markdown template, it will look as below:

```
1. We are given `n = 29` and `target = 10`. The goal is to find the smallest `x` that can be added to `n` to make the sum of digits less
2.
3. **Sum of Digits Calculation**: The sum of digits for `n` is 11 (`2 + 9`), which is greater than the target. Hence, `n` is not beau
4.
5. **Main Logic**:
6.    - Initialize `x` to `0`.
7.    - We enter the while loop since `f(n + x) = f(29 + 0) = 11` is greater than the target.
8.
9. **Finding the Next Candidate**: With `y = 29` and `p = 10`, we find that the least significant digit is `9`.
10.
11. **Jumping to Next Significant Candidate**: Execute the jump calculation to get `x`. Now `x = (y // 10 + 1) * 10 - n` which equals `1`
12.
13. **Verification and Loop Exit**: The new value of `n + x` is `30`. Since `f(30) = 3`, it meets the condition `(f(n + x) <= target)`
14.
15. We return the value `x = 1` as the required number to add to `n` to make it beautiful.
```

Python Solution

```python
class Solution:
    def makeIntegerBeautiful(self, n: int, target: int) -> int:
        # Function to compute the sum of digits of an integer 'x'.
        def sum_of_digits(x: int) -> int:
            total_sum = 0
            while x > 0:
                total_sum += x % 10
                x //= 10
            return total_sum

        increments = 0  # Variable to keep track of how much we need to add to 'n' to make it beautiful.

        # Loop until the sum of digits of 'n + increments' is greater than 'target'.
        while sum_of_digits(n + increments) > target:
            temp_number = n + increments
            power_of_ten = 10

            # Find trailing zeros by reducing 'temp_number' until it's not divisible by 10.
            while temp_number % 10 == 0:
                temp_number //= 10
                power_of_ten *= 10  # Increase power of 10 correspondingly.

            # Compute increments such that 'n + increments' eliminates trailing zeros,
            # and reduces digit sum. Calculate the next temp number and adjust increments.
            increments = (temp_number // 10 + 1) * power_of_ten - n

        # Return the total amount required to add to 'n' to make it beautiful.
        return increments

# Example usage:
# sol = Solution()
# result = sol.makeIntegerBeautiful(123, 12)
# print(result)  # Output depends on the function logic and provided parameters
```

Java Solution

```java
class Solution {

    // Method to adjust a number n to make its digit sum equal to a specified target
    public long makeIntegerBeautiful(long n, int target) {
        long adjustment = 0; // Initialize an adjustment value as 0

        // Loop until the sum of digits of n + adjustment is greater than the target
        while (sumOfDigits(n + adjustment) > target) {
            long temp = n + adjustment; // Create a temporary variable to hold n + adjustment
            long multiplier = 10;       // Initialize a multiplier for finding the next number divisible by 10

            // Find the number of trailing zeros in the current number
            while (temp % 10 == 0) {
                temp /= 10;             // Remove the trailing zero
                multiplier *= 10;       // Increase the multiplier by a factor of 10
            }

            // Calculate the next adjustment value
            // It's like increment the number until it has no trailing zeros and then subtract n
            adjustment = (temp / 10 + 1) * multiplier - n;
        }

        return adjustment;  // Return the final adjustment value
    }

    // Helper method to calculate the sum of digits of a given number x
    private int sumOfDigits(long x) {
        int sum = 0;  // Initialize sum as 0
        while (x > 0) {
            sum += x % 10;  // Add the last digit to sum
            x /= 10;        // Remove the last digit
        }
        return sum;  // Return the final sum of digits
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to add a certain amount to a number 'n' to make the sum of its digits equal to 'target'
    long long makeIntegerBeautiful(long long n, int target) {
        // Define a lambda function to calculate the sum of digits of a given number.
        auto sumOfDigits = [](long long x) {
            int sum = 0;
            while (x) {
                sum += x % 10; // Add the last digit to 'sum'
                x /= 10;       // Remove the last digit from 'x'
            }
            return sum;
        };

        long long addition = 0; // The amount to add to 'n' to make it beautiful
        // Keep increasing 'n' until the sum of its digits is equal to or less than 'target'
        while (sumOfDigits(n + addition) > target) {
            long long y = n + addition;
            long long nextPowerOfTen = 10;
            // Skip over the trailing zeros in 'y'
            while (y % 10 == 0) {
                y /= 10;
                nextPowerOfTen *= 10;
            }
            addition = (y / 10 + 1) * nextPowerOfTen - n; // Calculate the next amount to add
        }
        return addition; // Return the amount that needs to be added to 'n' to make it beautiful
    }
};
```

Typescript Solution

```typescript
// This function computes the minimal positive integer that should be added to 'n' to make the sum of its digits less than or equal t
function makeIntegerBeautiful(n: number, target: number): number {
    // Function to calculate the sum of the digits of an integer 'x'.
    const calculateDigitSum = (x: number): number => {
        let digitSum = 0; // Initialize the digit sum to zero.
        // Loop to add each digit to the sum.
        while (x > 0) {
            digitSum += x % 10; // Add the rightmost digit to the digit sum.
            x = Math.floor(x / 10); // Remove the rightmost digit from 'x'.
        }
        return digitSum; // Return the calculated sum of the digits.
    };

    let addition = 0; // Initialize the addition to zero.

    // Loop until the sum of the digits of (n + addition) is not greater than 'target'.
    while (calculateDigitSum(n + addition) > target) {
        let modifiedN = n + addition; // The modified value of 'n' after the addition.
        let multiplier = 10; // A multiplier used to find the next significant digit to increment.

        // Loop to disregard trailing zeros in the modified 'n' value.
        while (modifiedN % 10 === 0) {
            modifiedN = Math.floor(modifiedN / 10); // Divide by 10 to remove the trailing zero.
            multiplier += 10; // Update the multiplier for the next significant digit.
        }
        // Calculate the value to be added so that the next addition results in a digit sum less than or equal to 'target'.
        addition = (Math.floor(modifiedN / 10) + 1) * multiplier - n;
    }
    return addition; // Return the least number to add to 'n' to make it beautiful.
}

// Example usage:
// let result = makeIntegerBeautiful(1234, 10);
// console.log(result); // Shows the minimal positive integer addition to make '1234' have a digit sum not greater than '10'.
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by two nested loops: the `while` loop that iterates until `f(n + x)` is no longer greater than `target`, and the inner `while` loop that determines the next value of `x`.

The outer `while` loop can be executed potentially many times depending on the initial value of `n` and the `target` value. In the worst-case scenario, the digits of `n` are large, and it takes several increments of `n` to reduce the sum of its digits to `target`. For each increment, the inner loop runs until a non-zero digit is found (dividing by 10 each time). The cost of an iteration of the inner `while` loop is O(log(n)) as it is iterating over the digits of `y`, which is at most O(log(n)) digits long.

However, the inner loop execution time is offset by the fact that on each iteration, the value of `x` is increased significantly due to the multiplication by `p` (which is increased by a factor of 10 for each trailing zero of `y`). This effectively reduces the number of times the outer loop executes because the next `n + x` will have fewer trailing zeroes and thus will return a smaller digit sum `f(n + x)` more quickly. It could lead to the outer loop having less than linear time complexity in `n` on average. This complexity is hard to define precisely without specifics about the distribution of `n` and `target`.

Given this, we can cautiously estimate the worst-case time complexity to be O(log(n)^2) but the actual time complexity may vary greatly and could be much lower on average, depending on the input values.

Space Complexity

The space complexity is O(1), which means it requires constant additional space regardless of the input size. This is because the space used does not increase with the size of `n` or the `target`. Only a fixed number of variables (`x`, `y`, `p`) are used to store intermediate results and they do not depend on the size of the input.