2417. Closest Fair Integer

Medium Math Enumeration

Leetcode Link

Problem Description

The problem asks for the smallest integer greater than or equal to a given positive integer `n` that is considered "fair." An integer is defined as "fair" if it has an equal number of even and odd digits. For example, if `n` is `23` which has one even and one odd digit, it is a fair integer. However, if `n` is `123`, we should find the next integer where the count of even and odd digits is the same.

Intuition

The intuition behind solving this problem involves determining whether the given `n` itself is a fair number or not. If `n` is not fair, we need to find the next fair number after `n`.

Here's the thinking process for the solution:

1. We first need to check if the given number of digits `k` in `n` is odd or even. If `k` is odd, there cannot be an equal number of even and odd digits in any number with `k` digits. In this case, we need to look for the smallest fair number with `(k+1)` digits (because the next fair number must have an even number of digits). This can be done by appending a '1' to the right half of the number made up of `k` number of '1's (to ensure the count of even and odd digits are the same) and padding the rest with zeroes.
2. If `k` is even, we need to count the even and odd digits of `n`. If the count is already the same, `n` is the smallest fair number. If not, we need to increase `n` by 1 and re-evaluate until a fair number is found.

A recursive approach works here where we keep increasing `n` by 1 and checking if it is fair. If we come across an odd length of `n` while increasing it, we can directly return the smallest fair number with `k+1` digits.

This algorithm works since every time we increment `n`, we move closer to the next fair number, and by handling the odd digit length directly, we avoid unnecessary checks.

Solution Approach

The given solution uses a simple iterative and recursive approach to find the smallest fair integer greater than or equal to the given number `n`. Here's how the Python code implements this approach:

1. The `closestFair` method is a recursive function that takes an integer `n` as an argument.
2. It starts by initializing three variables: `a` and `b` to count the number of odd and even digits respectively, and `k` to count the total number of digits.
3. The method goes through each digit of `n` to calculate `a`, `b`, and `k`. This is done using a `while` loop which checks each digit by performing modulo and division operations: `t % 10` extracts the last digit of `t` and `t //= 10` reduces `t` by one digit.
4. If the last digit (`t % 10`) is odd (checked using bitwise AND with `1`), `a` is incremented; otherwise, `b` is incremented.
5. The variable `k` is incremented after each iteration to keep track of the number of digits processed.
6. After the loop, if `k` is found to be odd, this means we cannot have a fair number with the same number of digits as `n`. Therefore, we calculate the smallest fair number with `k+1` digits. This is done by creating a number with `k+1` digits where the left half of the digits (excluding the middle digit) are all `1`, ensuring equal even and odd digits, and the rest are `0`. The number `x` represents `10**k` (creating a new digit), and `-` represents a half of '1's. The expression `'1' * (k >> 1)` creates a string of ones half the length of `k`, and `or '0'` is used for cases when `k >> 1` is zero.
7. If `k` is even and `a` equals `b`, then `n` itself is a fair number so it is returned.
8. Else, if `n` is not fair and `k` is even, the function calls itself recursively with the incremented value `n + 1` and checks again.

Using recursion allows the algorithm to keep trying consecutive numbers after `n` until it discovers a fair number, and using the modulo and division operations enables it to easily count the even and odd digits in the integer. The direct calculation for the case when `k` is odd is a smart optimization that prevents unnecessary calculations by leveraging the fact that you can't have a fair number with an odd number of digits.

Example Walkthrough

Let's say we are given `n = 1234`, and we wish to find the smallest fair integer greater than or equal to `n`.

Starting with `n = 1234`, the function initializes `a` and `b` to `0`, and `k` would be the number of digits in `n`. We iterate through each digit to calculate the number of odd (`a`) and even (`b`) digits, as well as the total number of digits (`k`).

1. `1234` - Last digit is `4` (even), so `b` becomes `1`.
2. `123` - Last digit is `3` (odd), so `a` becomes `1`.
3. `12` - Last digit is `2` (even), so `b` becomes `2`.
4. `1` - Last digit is `1` (odd), so `a` becomes `2`.

After the loop, we can see that `k = 4` (even number of total digits), and `a = 2` and `b = 2`. This means the number of even and odd digits is the same (2 each), so `n = 1234` is a fair number by the definition given in the problem description. Therefore, `1234` itself is the smallest fair integer greater than or equal to `n`, and we would return `1234`.

Let's look at another example with `n = 123`.

1. `123` - Last digit is `3` (odd), so `a` becomes `1`.
2. `12` - Last digit is `2` (even), so `b` becomes `1`.
3. `1` - Last digit is `1` (odd), so `a` becomes `2`.

Here, the loop finalizes with `k = 3` (an odd number of total digits). Since we can't have a fair number with an odd number of digits, we need to find the smallest fair number with `k+1` digits, which is `4`.

Now, to get the smallest number with `4` digits, we take `k = 3`, and we need `k + 1 = 4` digits in total. This is achieved by creating a number that has half of the digits as `1` and half as `0`, ensuring the fair count of even and odd digits. The smallest such number is `1001`, which has two '1's (odd digits) and two '0's (even digits).

In conclusion, if `n = 1234`, the smallest fair integer is `1234`, and if `n = 123`, the smallest fair integer is `1001`.

Python Solution

```python
class Solution:
    def closestFair(self, num: int) -> int:
        # Initialize digit counts for odds (odd_count) and evens (even_count)
        # Also initialize the number of digits (num_digits)
        odd_count = even_count = num_digits = 0
        temp = num

        # Count the number of even and odd digits and the total digits in the number
        while temp > 0:
            if temp % 10 % 2 == 1:
                odd_count += 1
            else:
                even_count += 1
            temp //= 10
            num_digits += 1

        # If the number of digits is odd, we find the smallest fair integer with more digits
        if num_digits % 2 == 1:
            base = 10 ** num_digits
            half_odd_digits = '1' * (num_digits // 2)
            if half_odd_digits == '':
                half_odd_digits = '0'
            return base + int(half_odd_digits)

        # If the number of odd and even digits is already the same, return the number
        if odd_count == even_count:
            return num

        # Otherwise recursively find the next closest fair number by incrementing the number
        return self.closestFair(num + 1)

```

Java Solution

```java
class Solution {
    // Method to find the closest 'fair' integer
    public int closestFair(int n) {
        // Initialize the count of odd and even digits
        int oddCount = 0, countEven = 0;
        // Variable to store the number of digits in n
        int numDigits = 0;
        // Copy of the original number n (temporary variable)
        int temp = n;

        // Count the number of odd and even digits in n
        while(temp > 0){
            // If the rightmost digit is even
            if ((temp % 10) % 2 == 1) {
                countOdd++;
            } else { // If the rightmost digit is even
                countEven++;
            }
            temp /= 10; // Remove the rightmost digit of temp
            numDigits++; // Increment the digit count
        }

        // If the number of digits is odd, it's impossible to have an equal number of even and odd digits
        if (numDigits % 2 == 1) {
            // Find the smallest even digit number with one more digit than n
            // This is calculated as: 1 with (Math.pow(10, numDigits));
            // Initialize the number which contains only '1' and half the length of the original number
            int halfOnes = 0;
            for (int i = 0; i < numDigits / 2; ++i) {
                halfOnes = halfOnes * 10 + 1;
            }
            // Return the next 'fair' number which is fair (guaranteed by the next power of ten)
            // and has more digits than n
            return nextPowerOfTen + halfOnes;
        }

        // If n has an equal number of odd and even digits, it is fair so return n
        if (countOdd == countEven) {
            return n;
        }

        // If n is not fair, recursively call the method with n + 1 to find the closest fair integer
        return closestFair(n + 1);
    }
}
```

C++ Solution

```cpp
#include <cmath>  // include cmath for the pow function

class Solution {
public:
    // Function to find the smallest integer greater than or equal to 'n'
    // such that the count of even and odd digits is the same (a "fair" number)
    int closestFair(int n) {
        int oddDigits = 0, evenDigits = 0;
        // 'numDigits' counts odd digits, 'evenDigits' counts even digits
        int oddDigits = 0, evenDigits = 0;
        // 'temp' will be used to process the input number 'n'
        int temp = n;
        // 'numDigits' counts the number of digits in 'n'
        int numDigits = 0;

        // Loop to count even and odd digits in 'n'
        while (temp > 0) {
            if ((temp % 10) & 1) { // Check if the last digit is odd
                ++oddDigits;
            } else { // The last digit is even
                ++evenDigits;
            }
            ++numDigits; // Increment the count of digits
            temp /= 10;  // Remove the last digit
        }

        // If the counts of even and odd digits are the same, return 'n'
        if (oddDigits == evenDigits) {
            return n;
        }

        // If the number of digits is odd, it's impossible to have the same
        // count of even and odd digits, so we calculate the next smallest "fair" number
        if (numDigits % 2 == 1) {
            // Calculate the smallest number with 'numDigits + 1' digits
            // 'pow(10, numDigits)' is equivalent to 10^numDigits
            int nextPower = pow(10, numDigits);
            // 'halfFair' will store the number with equal count of odd and even digits
            int halfFair = 0;
            // Calculate half of the required odd digits for a fair number
            for (int i = 0; i < numDigits / 2; ++i) {
                halfFair = halfFair * 10 + 1;
            }
            // Return the 'nextNumber' with 'halfFair' added to it (to make it fair)
            return nextNumber + halfFair;
        }

        // If the number of digits is even but 'n' is not "fair", try the next integer
        return closestFair(n + 1);
    }
};
```

Typescript Solution

```typescript
// Include the Math library for the Math.pow function
function countDigits(num: number): number {
    let count = 0;
    while (num > 0) {
        count++;
        num = Math.trunc(num / 10);
    }
    return count;
}

function isFair(num: number): boolean {
    let oddDigits = 0, evenDigits = 0;
    let temp = num;

    while (temp > 0) {
        let digit = temp % 10;
        if (digit % 2 == 0) {
            evenDigits++;
        } else {
            oddDigits++;
        }
        temp = Math.trunc(temp / 10);
    }

    return oddDigits === evenDigits;
}

function nextFairNumber(num: number): number {
    let digitCount = countDigits(num);

    if (isFair(num)) {
        return num;
    }

    if (digitCount % 2 === 1) {
        let nextNumber = Math.pow(10, digitCount);
        let halfFairNumber = 0;
        for (let i = 0; i < digitCount / 2; i++) {
            halfFairNumber = halfFairNumber * 10 + 1;
        }
        return nextNumber + halfFairNumber;
    }

    return nextFairNumber(num + 1);
}

function closestFair(n: number): number {
    return nextFairNumber(n);
}
```

Time and Space Complexity

The given Python code defines a method `closestFair` which finds the closest integer greater than or equal to `n` that contains an equal number of even and odd digits.

Time Complexity

Firstly, the method calculates the number of even (`b`) and odd (`a`) digits in the input integer `n` and counts the total number of digits (`k`). This involves a while loop that runs once for each digit in `n`, resulting in `O(d)` complexity, where `d` is the number of digits in `n`.

Next, for an odd number of digits (`k`), it constructs a number with a `1` repeated `k >> 1` times (which takes constant time) and adds it to `10**k` (also constant time). This part does not depend on the input size and is considered `O(1)`.

If the number of even and odd digits is equal for an even number of digits (`k`), it immediately returns the number, which is `O(1)`.

Lastly, if the number of even and odd digits is not equal, it recursively calls `closestFair` on the next integer `n + 1`. The worst-case scenario for this recursion is tricky to analyze, as it will depend on how the digits in `n` are distributed and how many increments are necessary to reach an equal number of even and odd digits. However, in the worst case, it could potentially increment through many numbers and hence have a worst-case time complexity affected by the magnitude of `n` and the mathematics of digit distribution, which is difficult to express in standard notation.

Space Complexity

The space complexity is primarily the space required to hold the variables and the recursion stack if the method keeps calling itself. Since the number of variables `a`, `b`, `k`, `t`, `n`, and `num` are fixed, the method utilizes `O(1)` space in terms of variables.

The recursion depth can be significant in the worst case (if the method keeps calling itself for each consecutive number until a fair number is found), but because Python's default recursion limit is typically a constant threshold (not dependent on the input size), the space complexity due to recursion is also `O(1)` under the Python recursion limit constraint.

In conclusion, the dominant time complexity is difficult to determine accurately without a more concrete analysis or pattern in the digit distribution of `n`. However, it can be bounded by `O(d)` for the initial digit-counting, and potentially worse, depending on subsequent recursive calls for an imbalanced `n`. The space complexity is `O(1)` under Python's recursion stack size constraint, as the recursion depth will hit a limit.