2843. Count Symmetric Integers

Easy Math Enumeration Leetcode Link

Problem Description

The problem provides us with two positive integers, `low` and `high`. We need to determine the count of symmetric integers within this inclusive range. An integer `x` is considered symmetric if it has an even number of digits (`2 * n`), and the sum of the first half of its digits (`n` digits) is equal to the sum of the second half (`n` digits). For instance, the number `123321` is symmetric because the sum of `123` is equal to the sum of `321`. Integers with odd numbers of digits cannot be symmetric by definition provided. Our goal is to calculate how many symmetric numbers exist between `low` and `high`, inclusive.

Intuition

The problem at hand is an excellent example of brute force technique where we iterate over each number within the range provided and check if it is symmetric.

The intuition behind the provided solution starts with defining what a symmetric number is and understanding that symmetry is only possible with an even number of digits. Hence, numbers with an odd number of digits can be immediately discarded as they can never be symmetric.

Given the definition of symmetry, the next logical step is to split the number into two equal parts and compare their digit sums. To achieve this, for each number within the given range:

1. Convert the integer to a string, so it's easier to split and work with individual digits.
2. Check if the length of the string (representing the number) is odd. If it's odd, this number cannot be symmetric, and the function `f` should return `False`.
3. If the length is even, find the midpoint (`n`) and then split the string into two halves.
4. Convert each digit from both halves back to integers and sum them up separately.
5. Compare the two sums. If they are equal, `f` will return `True`, indicating that the number is symmetric; otherwise, `False`.

Having the function `f` ready, the next step is to apply it to each number within the `low` to `high` range. The solution approach uses Python's `sum()` function in a generator expression to count how many times `f` returns `True` for numbers in the range. The count of `True` returns is the count of symmetric numbers, which is what the `countSymmetricIntegers` method finally returns.

Solution Approach

The solution for this problem uses a simple brute force algorithm to find all symmetric numbers between two integers, `low` and `high`. This is achieved using a helper function `f` to check each number's symmetry and a main method that iterates through the range, using a generator expression to count the symmetric numbers.

Let's walk through the key components of the implementation:

1. **Helper Function `f(x: int) -> bool`**: This function takes an integer `x` and returns a boolean indicating whether `x` is a symmetric number.
    - It first converts the integer `x` into a string `s`, making it easier to handle individual digits.
    - The function immediately checks if the length of `s` is odd. If so, it returns `False` because symmetric numbers must have an even number of digits.
    - It calculates `n`, which is half the length of the string `s`, to divide the number into its two parts.
    - It then takes the first `n` digits and the second `n` digits, converts each digit to an integer using `map`, and calculates their sums.
    - Lastly, `f` compares the two sums and returns `True` if they are equal (symmetric) or `False` if not.
2. **Main Method `countSymmetricIntegers(low: int, high: int) -> int`**: This is where the range of numbers is processed.
    - It uses a `for` loop within a generator expression `(f(x) for x in range(low, high + 1))` to apply the helper function `f` to each integer `x` in the range from `low` to `high` (inclusive).
    - The `for` loop is enclosed in the `sum()` function, which adds up all the `True` values (each `True` is equivalent to `1` in Python) returned by `f`, thus counting the number of symmetric numbers.
    - The sum, which represents the count of symmetric numbers in the given range, is then returned as the output.

This solution approach does not use any complex data structures as the problem is more focused on digit manipulation and comparison rather than data manipulation. By converting the numbers to strings, it takes advantage of Python's string indexing and slicing capabilities to easily and intuitively work with numerical halves. The concise design of the solution with a separate function for symmetry check keeps the code clean and readable.

Example Walkthrough

Let's consider a small range of numbers to illustrate the solution approach, specifically the range from `low = 1200` to `high = 1301`.

We need to check each number within this range to see if it is a symmetric integer. According to the problem statement, for an integer to be symmetric, it must have an even number of digits, and the sum of the first half of its digits must be equal to the sum of the second half.

Let's start with the first number in the range, `1200`:

1. Convert it to a string: "1200"
2. The length of the string is 4, which is even, so it's possible for the number to be symmetric.
3. Calculate the midpoint: `n = len("1200") / 2 = 2`
4. Split the string into two parts: "12" and "00"
5. Convert the digits and calculate the sums: sum of "12" = 1 + 2 = 3, sum of "00" = 0 + 0 = 0
6. Compare the sums: since 3 is not equal to 0, `f(1200)` returns `False`. Hence, 1200 is not a symmetric number.

Moving on to the number `1301`:

1. Convert it to a string: "1301"
2. The length of the string is 4, which is even, so it's possible for the number to be symmetric.
3. Calculate the midpoint: `n = len("1301") / 2 = 2`
4. Split the string into two parts: "13" and "01"
5. Convert the digits and calculate the sums: sum of "13" = 1 + 3 = 4, sum of "01" = 0 + 1 = 1
6. Compare the sums: since 4 is not equal to 1, `f(1301)` returns `False`. Hence, 1301 is not a symmetric number.

None of the numbers from `1200` to `1301` are symmetric since we won't find any number in that range satisfying the symmetric condition. Thus, when the `countSymmetricIntegers(1200, 1301)` method is called, it will iterate through the range using the helper function `f` and the sum of the number of symmetric integers will be zero as none of the numbers will return `True`.

If we carefully observe, the smallest number that could potentially be symmetric in this range is `1210` because its digit sum for the first half "12" is 1 + 2 = 3, which is equal to the digit sum for the second half "10", which is 1 + 0 = 1, but since 3 is not equal to 1, it is also not symmetric. Any number in this range that has a zero cannot be symmetric as any other digit's sum will be more than zero making it impossible to be symmetric. So, it saves us the computational effort of checking each number by understanding the inherent properties of the numbers within the given range.

Python Solution

```python
class Solution:
    def count_symmetric_integers(self, low: int, high: int) -> int:
        # Helper function to check if an integer is symmetric
        # An integer is symmetric if sum of the first half of its digits equals sum of the second half,
        # and only the even-length numbers can be symmetric by this definition.
        def is_symmetric(num: int) -> bool:
            str_num = str(num)
            # Check for even length
            if len(str_num) % 2 == 1:
                return False
            half_length = len(str_num) // 2
            # Calculate sum of the first half and the second half of the digits
            first_half_sum = sum(map(int, str_num[:half_length]))
            second_half_sum = sum(map(int, str_num[half_length:]))
            # Check if both halves have equal sum
            return first_half_sum == second_half_sum

        # Calculate the number of symmetric integers within the given range
        symmetric_count = sum(is_symmetric(num) for num in range(low, high + 1))
        return symmetric_count

```

Java Solution

```java
class Solution {
    // Method to count symmetric integers within a given range
    public int countSymmetricIntegers(int low, int high) {
        int count = 0; // Initialize count to keep track of symmetric integers
        // Iterate through the range from low to high
        for (int num = low; num <= high; ++num) {
            // Add the result of the isSymmetric helper function to the count
            count += isSymmetric(num);
        }
        return count; // Return the total count of symmetric integers
    }

    // Helper method to determine if an integer is symmetric
    private int isSymmetric(int num) {
        String numStr = Integer.toString(num); // Convert the integer to a string representation
        int length = numStr.length(); // Calculate the length of the string
        if (length % 2 == 1) { // If the length of the number is odd, it is not symmetric
            return 0;
        }
        int firstHalfSum = 0, secondHalfSum = 0; // Initialize sums for both halves
        // Sum the digits in the first half of the string
        for (int i = 0; i < length / 2; ++i) {
            firstHalfSum += numStr.charAt(i) - '0'; // Convert char to int and add to the sum
        }
        // Sum the digits in the second half of the string
        for (int i = length / 2; i < length; ++i) {
            secondHalfSum += numStr.charAt(i) - '0'; // Convert char to int and add to the sum
        }
        // If the sums of both halves match, the number is symmetric
        return firstHalfSum == secondHalfSum ? 1 : 0;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the symmetric integers between `low` and `high`.
    int countSymmetricIntegers(int low, int high) {
        // Initialize the answer counter.
        int count = 0;
        // Define the lambda function to check if an integer is symmetric.
        auto isSymmetric = [](int num) {
            // Convert the number to a string.
            string numStr = to_string(num);
            // Get the number of digits in the string.
            int length = numStr.size();
            // Check if the number of digits is odd. If it is, return 0 immediately.
            if (length % 2 == 1) {
                return 0;
            }
            // Initialize the sums of the first half and second half of the digits.
            int sumFirstHalf = 0, sumSecondHalf = 0;
            // Iterate over the first half and second half digits and sum them up.
            for (int i = 0; i < length / 2; ++i) {
                sumFirstHalf += numStr[i] - '0';
                sumSecondHalf += numStr[length / 2 + i] - '0';
            }
            // Return 1 if the sums are equal, else return 0.
            return sumFirstHalf == sumSecondHalf ? 1 : 0;
        };
        // Iterate over the range of numbers from `low` to `high`.
        for (int num = low; num <= high; ++num) {
            // Increase the counter if the current number is symmetric.
            count += isSymmetric(num);
        }
        // Return the final count of symmetric integers.
        return count;
    }
};

```

Typescript Solution

```typescript
function countSymmetricIntegers(low: number, high: number): number {
    let count = 0; // Initialize the counter for symmetric integers

    // Define a helper function to check if a number is symmetric
    // In this context, a symmetric number is defined as having an even number of digits,
    // with the sum of the first half of the digits equal to the sum of the second half
    const isSymmetric = (num: number): number => {
        const strNum = num.toString(); // Convert the number to a string
        const length = strNum.length; // Get the length of the string

        // If the number of digits is odd, it can't be symmetric
        if (length & 1) {
            return 0;
        }

        let firstHalfSum = 0; // Sum of the first half of digits
        let secondHalfSum = 0; // Sum of the second half of digits

        // Calculate the sums for the first and the second halves of the string
        for (let i = 0; i < length / 2; ++i) {
            firstHalfSum += Number(strNum[i]); // Add the digit to the first half's sum
            secondHalfSum += Number(strNum[length / 2 + i]); // Add the digit to the second half's sum
        }

        // If both halves have the same sum, return 1 indicating a symmetric number
        return firstHalfSum === secondHalfSum ? 1 : 0;
    };

    // Loop through the range from 'low' to 'high' to count symmetric numbers
    for (let num = low; num <= high; ++num) {
        count += isSymmetric(num); // Add the result of isSymmetric to the count
    }

    return count; // Return the total count of symmetric numbers
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be evaluated by looking at the number of operations it performs in relation to the input range `low` to `high`.

We perform a loop from `low` to `high`, inclusive. Therefore, the number of iterations is `(high - low + 1)`.

Within each iteration, we do the following:

- Convert the number `x` to a string, which takes `O(d)` time, where `d` is the number of digits in the number.
- Check the length of the string; this is a constant time operation, `O(1)`.
- Calculate the half-length of the string, which is also a constant time operation, `O(1)`.
- Split the string and sum the digits of both halves. Since each half of the string has `d/2` digits, the map and sum operations together take `O(d)` time for each half, totaling to `O(d)` for the entire string.

Since each digit in the input number can be processed independently, we can consider `d` to be `O(log10(x))`, where `x` is a number in the given range. This is because the number of digits in a number is proportional to the logarithm of the number itself.

Therefore, for each iteration, we spend `O(log10(x))` time, and since we do this for each x in the range from `low` to `high`, the overall time complexity is `O((high - low + 1) * log10(high))`, assuming `high` has the maximum number of digits within the range.

Space Complexity

Regarding space complexity, the code uses only a constant amount of extra space, which is independent of the input size. It includes:

- Storage for temporary variables such as `s`, `n`, and the return value of `f(x)`.
- The space needed for the integer-to-string conversion, which at most storage for a string representation of `high`.

As the storage does not grow with the size of the input range, the space complexity remains constant `O(1)`.