172. Factorial Trailing Zeroes

Medium Math

Problem Description

Given an integer `n`, the objective is to calculate how many trailing zeroes are found in the factorial of `n`, noted as `n!`. The factorial of a number `n` is the product of all positive integers from `1` to `n`, inclusive. Trailing zeroes in this context refer to the number of zeroes that appear at the end of the number `n!` before any other digit appears.

Intuition

Trailing zeroes are created by the product of the factors 2 and 5. Since the factorial of any number `n` will have more factors of 2 than factors of 5, the number of trailing zeroes is determined by the number of times the factor 5 is included in the factorial.

Therefore, to count the number of trailing zeroes, we need to find out how many times the number `5` can fit into the factors of all numbers from 1 to `n`. This is done by dividing `n` by 5 and adding the quotient to the count of trailing zeroes. However, numbers that are powers of 5 (such as 25, 125, etc.) are counted multiple times because they contribute more than one factor of 5. For example, 25 is 5 * 5, so it contributes two factors of 5.

The solution approach is to iteratively divide `n` by 5 and add the quotient to a cumulative sum until `n` is zero. On each iteration, `n` is updated by dividing it by 5. This way, we account for additional factors of 5 contributed by powers of 5. The final sum is the number of trailing zeroes in `n!`.

Solution Approach

The implementation of the solution leverages a simple iterative approach without the need for any complex data structures or algorithms. It makes use of basic arithmetic division and addition operations to determine the count of trailing zeroes.

Here's the step-by-step breakdown of the solution approach:

1. Initialize a variable `ans` to 0 to hold the accumulated count of trailing zeroes.
2. Enter a while loop that will continue to execute as long as `n` is not zero. Inside the loop:
   - Perform integer division of `n` by 5 and update `n` with the quotient. This is done using the `//` operator in Python which is the floor division operator. It returns the largest possible integer that is less than or equal to the division result.
     ```
     n //= 5
     ```
   - Add the updated value of `n` to `ans`. Since `n` was divided by 5, it now represents the number of additional factors of 5 that can be included from the updated range of numbers to `n`.
     ```
     ans += n
     ```
3. After the loop terminates (when `n` is reduced to 0), `ans` will contain the total count of trailing zeroes in the factorial of the original `n`.
4. Return `ans` as the final result.

By repeatedly dividing `n` by `5`, the solution effectively accounts for all the factors of `5` present in the factorial of `n`. These factors are the ones responsible for contributing trailing zeroes since there are always sufficient factors of `2` to pair with them. This process takes into account not just the single multiples of `5` but also the multiples of higher powers of `5`, which contribute more than one `5` to the factorial.

There are no additional patterns or complex algorithms used in this approach as the problem can be efficiently solved with this direct method.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we want to find the number of trailing zeroes in the factorial of `n=10`. The factorial of 10 is `10! = 10 x 9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1`.

Following the steps in the solution approach:

1. We initialize `ans` to 0. This will keep track of the number of trailing zeroes.
2. We check how many times `10` can be divided by `5`:
   - In the first iteration, `10 // 5 = 2`. So, we set `n` to `2` and add `2` to `ans`. Now `ans = 2`.
3. We continue the while loop with the updated value of `n`:
   - `n` is now `2`, which is not zero, so we continue.
   - Now, `2 // 5 = 0`. We set `n` to `0` (since `2` is less than `5`, and using floor division, we get `0`).
   - We add `0` to `ans`. This does not change `ans`, so it remains `2`.
4. The while loop ends because `n` is now `0`.
5. At the end, `ans = 2`, which means the factorial of `10` (`10!`) has 2 trailing zeroes.

This small example shows how the solution approach efficiently calculates the number of trailing zeroes of a factorial by considering the powers of `5` in `n!`. The process of dividing `n` by `5` and summing the quotients ensures we count each factor of `5` present in the factorial, leading to the correct number of trailing zeroes.

Solution Implementation

Python

```python
class Solution:
    def trailingZeroes(self, n: int) -> int:
        # Initialize zeros count to 0
        zeros_count = 0

        # Keep dividing n by 5 and update zeros count
        while n:
            n //= 5
            zeros_count += n

        # Return the count of trailing zeros in n!
        return zeros_count
```

Java

```java
class Solution {
    // This method calculates the number of trailing zeros in the factorial of a given number 'n'.
    public int trailingZeroes(int n) {
        int count = 0; // Initialize a counter to store the number of trailing zeros
        while (n > 0) {
            n /= 5; // Divide 'n' by 5, reducing it to the count of factors of 5
            count += n; // Increment the count by the number of 5's factors in the current 'n'
        }
        return count; // Return the total count of trailing zeros
    }
}
```

C++

```cpp
class Solution {
public:
    int trailingZeroes(int n) {
        int count = 0; // Initialize count of trailing zeroes in factorial

        // Loop to divide n by powers of 5 and add to the count
        while (n > 0) {
            n /= 5;      // Factor out the 5's, as 10 is the product of 2 and 5,
                         // and there are always more 2's than 5's in factorial.
            count += n;  // Accumulate the number of 5's in all factors of n
        }

        return count; // return the number of trailing zeroes in n!
    }
};
```

TypeScript

```typescript
// Calculates the number of trailing zeroes in the factorial of a given number
// by counting the number of multiples of 5 in the factors of the numbers that compose the factorial.
// This is because each pair of 2 and 5 contribute to a trailing zero, and there will always
// be more multiples of 2 than 5, so we only count the multiples of 5.

/**
 * Calculates the number of trailing zeroes in n! (n factorial).
 *
 * @param {number} n - The input number to calculate the factorial for.
 * @returns {number} The number of trailing zeroes in n!.
 */
function trailingZeroes(n: number): number {
    let zeroCount = 0; // Initialize the count of trailing zeros

    // Keep dividing n by 5 and update zeroCount to count
    // the factors of 5 in n! as each contributes to a trailing zero
    while (n > 0) {
        n = Math.floor(n / 5); // Divide n by 5
        zeroCount += n; // Increment zeroCount by the quotient
    }

    // Return the total count of trailing zeroes in n!
    return zeroCount;
}
```

```python
class Solution:
    def trailingZeroes(self, n: int) -> int:
        # Initialize zeros count to 0
        zeros_count = 0

        # Keep dividing n by 5 and update zeros count
        while n:
            n //= 5
            zeros_count += n

        # Return the count of trailing zeros in n!
        return zeros_count
```

Time and Space Complexity

The given Python code function `trailingZeroes` calculates the number of trailing zeros in the factorial of a given number `n`. The main idea behind the code is that trailing zeros are created by the pair of prime factors 2 and 5, and since there are more 2s than 5s in the factorial, the number of 5s will determine the number of trailing zeros.

The algorithm works by iteratively dividing `n` by 5 to count how many multiples of 5, 25, 125, etc., are there in `n!` (factorial of n), as each contributes to a trailing zero.

Time Complexity:

The time complexity is determined by the number of times `n` can be divided by 5. This loop runs approximately `log5(n)` times since we divide `n` by 5 in each iteration until `n` becomes 0. Each operation inside the loop takes constant time. Hence, the time complexity is `O(log5(n))`.

Space Complexity:

The space complexity of this algorithm is `O(1)`, since we only need a fixed amount of space for the variable `ans`, no matter the size of the input `n`.