2652. Sum Multiples

Easy Math

Problem Description

Given a positive integer `n`, the task is to calculate the sum of all integers within the range from `1` to `n` (both inclusive) which are divisible by either `3`, `5`, or `7`. Intuitively, this involves finding all the multiples of `3`, multiples of `5`, and multiples of `7` within that range, summing them up, and returning the result. A multiple of a number such as `3`, `5`, or `7` is any number that can be expressed as the product of that number and an integer. For example, `6` (`2 * 3`) and `15` (`3 * 5`) are multiples of both `3` and `5`.

Intuition

The problem can be approached intuitively in two primary ways:

Intuition behind Solution 1: Enumeration (Brute Force)

The most straightforward approach is to enumerate, i.e., iterate through every number in the range from `1` to `n` and check which numbers are divisible by `3`, `5`, or `7`. If a number is divisible by any of these, it gets added to a running sum. This method is not very efficient for large values of `n` because it requires checking every single number up to `n`, which means `O(n)` time complexity—linear time in relation to the size of the range.

Intuition behind Solution 2: Mathematics (Inclusion-Exclusion Principle)

A more mathematically elegant and efficient solution involves using the inclusion-exclusion principle to avoid enumerating through all numbers. This principle is a way to calculate the cardinality of the union of several sets—here, those sets are multiples of `3`, `5`, and `7`. The principle accounts for the fact that some numbers are common multiples and thus would be counted more than once if we simply summed all multiples of `3`, `5`, and `7`. To correct for this, we subtract the sums of pairs of multiples (the intersections of two sets) and then add back in the sum of numbers which are multiples of all three (the intersection of all three sets). By doing this, each multiple will be counted exactly once. This method reduces the problem to evaluating several arithmetic series, leading to a solution that runs in constant time, `O(1)`, regardless of the size of `n`.

Solution Approach

Solution 1: Enumeration

The provided code snippet demonstrates the enumeration approach through a list comprehension in Python. The `range(1, n + 1)` function generates a sequence of numbers from `1` through `n` inclusive. The generator expression inside the `sum` function iteratively checks for each `x` in this range whether `x % 3 == 0`, `x % 5 == 0`, or `x % 7 == 0`, which are conditions for divisibility by `3`, `5`, and `7`, respectively. If any of these conditions are true, `x` is included in the sum. This is essentially a brute force method that iterates through the entire range, checking each number individually.

The time complexity here is `O(n)`, since each number up through `n` is looked at once. The space complexity is `O(1)` because only a single sum is being maintained and no additional space is required.

Solution 2: Mathematics (Inclusion-Exclusion Principle)

This solution deploys the inclusion-exclusion principle to find the required sum without having to iterate over the entire range. To implement this approach mathematically, we first define a function `f(x)` that can give us the sum of all multiples of `x` within the range `[1, n]`. To find the sum of these multiples, we note that they form an arithmetic sequence with `x` as the first term, the largest multiple of `x` less than or equal to `n` as the last term, and the number of terms being `m = floor(n / x)`. Thus, the sum of an arithmetic series formula can be used to find `f(x)`, which is `f(x) = (x + mx) * m / 2`.

To find the final sum of numbers divisible by `3`, `5`, or `7`, we add up `f(3)`, `f(5)`, and `f(7)`, and then subtract the sums of the numbers that are common multiples—which would have been counted multiple times—using `f(3 * 5)`, `f(3 * 7)`, and `f(5 * 7)`. Lastly, we add back the sum of numbers that are divisible by `3`, `5`, and `7` (`f(3 * 5 * 7)`) to ensure they are counted once in the total sum, as the inclusion-exclusion principle dictates.

The entire sum can be expressed succinctly as:

```
f(3) + f(5) + f(7) - f(3 * 5) - f(3 * 7) - f(5 * 7) + f(3 * 5 * 7)
```

Here, the time complexity is `O(1)` because we are only performing a fixed number of arithmetic operations, and the space complexity remains `O(1)` as we are only calculating a sum with fixed terms and not storing multiple values.

Example Walkthrough

Let's walk through a small example to illustrate Solution 2, as it's the more mathematically involved approach. Consider the positive integer `n = 10`. We want to find the sum of all integers within the range from `1` to `10` (inclusive) that are divisible by `3`, `5`, or `7`.

Firstly, we need to determine the arithmetic series for multiples of `3`, `5`, and `7` within our range. Given that the integer `10` is our limit:

- The multiples of `3` are `3, 6, 9`. The sum of these is `3 + 6 + 9 = 18`.
- The multiples of `5` are `5, 10`. The sum of these is `5 + 10 = 15`.
- The multiples of `7` are `7`, as `14` is beyond our range. The sum is `7`.

Next, we account for the numbers that are multiples of both `3` and `5`, both `3` and `7`, and both `5` and `7` to avoid double-counting. Our range, however, is too small to have any common multiples for the latter two pairs, but for the multiples of both `3` and `5`, we have:

- The common multiples of `3` and `5` are simply `15` (as `30` is beyond our range), so this number is considered only once in the sum.

Finally, there are no numbers less than or equal to `10` that are multiples of `3`, `5`, and `7`.

Hence, we can calculate the sum using the inclusion-exclusion principle as follows:

- Add the sums of the multiples of `3`, `5`, and `7`: `18 + 15 + 7 = 40`.
- Subtract the sum of the multiples of `3` and `5` to correct for double-counting: `40 - 15 = 25`.
- Since we don't have any multiples in our range that are common to `3` and `7` or `5` and `7`, and there are no multiples of `3`, `5`, and `7`, we do not subtract anything further.

The final sum of all integers within the range from `1` to `10` that are divisible by `3`, `5`, or `7` is `25`.

This example demonstrates how even with a simple case, the inclusion-exclusion principle helps in accurately calculating the desired sum with minimal computational steps, adhering to an `O(1)` time complexity.

Solution Implementation

Python

```python
class Solution:
    def sum_of_multiples(self, n: int) -> int:
        """
        Calculate the sum of all multiples of 3, 5, or 7 up to and including a given number n.

        :param n: The upper limit of the range within which to look for multiples.
        :type n: int
        :return: The sum of all multiples of 3, 5, or 7 up to and including n.
        :rtype: int
        """
        # Use a generator expression within the sum function to iterate over all numbers from 1 to n (inclusive).
        # Use the modulo operator (%) to check if the current number is a multiple of 3, 5, or 7.
        # If the current number x is a multiple of any of those, it gets included in the sum.
        return sum(x for x in range(1, n + 1) if x % 3 == 0 or x % 5 == 0 or x % 7 == 0)

# Example usage:
# sol = Solution()
# result = sol.sum_of_multiples(10)
# print(result)  # Output will be the sum of multiples of 3, 5, or 7 up to 10
```

Java

```java
public class Solution {

    // Method to calculate the sum of all multiples of 3, 5, or 7 up to and including the number n
    public int sumOfMultiples(int n) {
        // Initialize the accumulator variable for the sum
        int sum = 0;
        // Loop from 1 through n (inclusive)
        for (int i = 1; i <= n; ++i) {
            // Check if the current number i is a multiple of 3, 5, or 7
            if (i % 3 == 0 || i % 5 == 0 || i % 7 == 0) {
                // Add the current number i to the sum if it is a multiple
                sum += i;
            }
        }
        // Return the resultant sum
        return sum;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the sum of multiples of 3, 5, or 7 up to a given number n.
    int sumOfMultiples(int n) {
        int sum = 0; // Initialize the sum to zero.

        // Loop from 1 to n (inclusive).
        for (int i = 1; i <= n; ++i) {
            // Check if the current number is a multiple of 3, 5, or 7.
            if (i % 3 == 0 || i % 5 == 0 || i % 7 == 0) {
                sum += i; // Add the number to the sum if it's a multiple.
            }
        }

        // Return the calculated sum of the multiples.
        return sum;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the sum of all multiples of 3, 5, or 7 up to and including number n.
 *
 * @param {number} n - The upper limit for checking the multiples.
 * @return {number} - The sum of the multiples of 3, 5, or 7 up to n.
 */
function sumOfMultiples(n: number): number {
    // Initialize the answer variable that will store the sum of the multiples.
    let sum = 0;

    // Iterate through all numbers from 1 to n (inclusive).
    for (let currentNumber = 1; currentNumber <= n; ++currentNumber) {
        // Check if the current number is a multiple of 3, 5, or 7.
        if (currentNumber % 3 === 0 || currentNumber % 5 === 0 || currentNumber % 7 === 0) {
            // If it is a multiple, add the current number to the sum.
            sum += currentNumber;
        }
    }

    // Return the final sum after completing the loop.
    return sum;
}

// Example usage:
// const result = sumOfMultiples(10);
// console.log(result); // This would output the sum of multiples of 3, 5, or 7 up to 10.
```

```python
class Solution:
    def sum_of_multiples(self, n: int) -> int:
        """
        Calculate the sum of all multiples of 3, 5, or 7 up to and including a given number n.

        :param n: The upper limit of the range within which to look for multiples.
        :type n: int
        :return: The sum of all multiples of 3, 5, or 7 up to and including n.
        :rtype: int
        """
        # Use a generator expression within the sum function to iterate over all numbers from 1 to n (inclusive).
        # Use the modulo operator (%) to check if the current number is a multiple of 3, 5, or 7.
        # If the current number x is a multiple of any of those, it gets included in the sum.
        return sum(x for x in range(1, n + 1) if x % 3 == 0 or x % 5 == 0 or x % 7 == 0)

# Example usage:
# sol = Solution()
# result = sol.sum_of_multiples(10)
# print(result)  # Output will be the sum of multiples of 3, 5, or 7 up to 10
```

Time and Space Complexity

The provided code snippet computes the sum of all multiples of 3, 5, or 7 up to and including `n`.

- The time complexity of this function is `O(n)`, where `n` is the input to the function. This is because the code iterates through a range from 1 to `n`, checking each number to see if it is a multiple of 3, 5, or 7.
- The space complexity is `O(1)` because no additional space is used that grows with the input size. The only space used is for the variable `x` during the iteration and temporary space for the sum function, which does not depend on the size of `n`.