2177. Find Three Consecutive Integers That Sum to a Given Number

Medium Math Simulation

Problem Description

The problem requires writing a function that takes an integer `num` as an input and outputs three consecutive integers whose sum equals `num`. If no such three consecutive numbers exist, the function should return an empty array. Consecutive integers are numbers that follow each other without any gaps; for example, 4, 5, 6 are consecutive integers.

Intuition

To solve this problem, we exploit the property that the sum of any three consecutive integers is always divisible by 3. To understand why, consider any three consecutive integers `n - 1`, `n`, and `n + 1`. Their sum is `(n - 1) + n + (n + 1)`, which simplifies to `3n`. Because the sum is simply `3n`, it's clear that any three consecutive integers must sum up to a multiple of 3.

Knowing this, we can check whether our given integer `num` is divisible by 3, to determine if it's possible to find such three numbers. We do this by using the `divmod` function which takes two numbers and returns a pair (tuple) consisting of their quotient and remainder.

If `num` is divisible by 3 (remainder is 0), we know that `num` can be expressed as the sum of three consecutive integers. We can find the middle integer by dividing `num` by 3. Let's denote this middle integer as `x`. Then, the three consecutive numbers would be `x - 1`, `x`, and `x + 1`.

If `num` is not divisible by 3 (remainder is not 0), we return an empty array, since there is no set of three consecutive integers that sum up to `num`.

The provided Python solution uses the approach described above. It declares a class `Solution` with a method `sumOfThree` that accepts an integer `num` and returns a list of integers. The method calculates the quotient `x` and the remainder `mod` when `num` is divided by 3. It then checks if the remainder `mod` is zero; if so, it returns a list containing `x - 1`, `x`, and `x + 1`. If the remainder is not zero, it means that `num` cannot be expressed as the sum of three consecutive numbers, and the method returns an empty list.

Solution Approach

The solution to this problem uses a straightforward algorithm that incorporates basic arithmetic operations and conditional logic. It's a single pass approach without the need for any complex data structures. Here's an in-depth walk-through of the solution's implementation:

1. **Division and Modulus Operations**: The solution employs the `divmod` function which takes two arguments and returns a tuple containing the quotient and remainder of the division. In this case, `divmod(num, 3)` gives us the quotient `x` and the remainder `mod`. For example, if `num` is 6, calling `divmod(6, 3)` would return `(2, 0)` since 6 divided by 3 is 2 with a remainder of 0.
2. **Conditional Check**: After getting the quotient and the remainder, we check if the `mod` is zero. This is a crucial step because if the remainder is zero, `num` is divisible by 3 and we can express it as a sum of three consecutive integers according to the property mentioned earlier. Otherwise, it cannot be expressed as such and we return an empty list.
3. **Calculating the Consecutive Integers**: If the remainder is zero, we find the middle integer `x` by simply using the quotient. The three consecutive integers will be `(x - 1, x, x + 1)`. For instance, if `num` is 27, `divmod(27, 3)` returns `(9, 0)`. The consecutive integers would then be `(8, 9, 10)`.
4. **Returning the Result**: In the end, the solution either returns the list of the three consecutive integers if the remainder is zero or an empty list if it's not. This is done using a simple ternary conditional expression in Python: `[] if mod else [x - 1, x, x + 1]`. This line checks if `mod` is True (which in Python means any number other than 0). If it's True (non-zero remainder), it returns `[]`. If `mod` is False (zero remainder), it returns `[x - 1, x, x + 1]`.

The solution is efficient and runs in constant time `O(1)` because it always performs a fixed number of operations regardless of the input size. The space complexity is also `O(1)` since it only stores a few variables and potentially returns an array with three integers at most.

Example Walkthrough

Let's work through a small example to illustrate the solution approach. Imagine the function `sumOfThree` is called with an input number `num = 15`. Here's a step-by-step breakdown of how the function will process this input:

1. **Division and Modulus Operations**: The function starts by using the `divmod` function to divide `num` by 3. We call `divmod(15, 3)` which returns `(5, 0)` because 15 divided by 3 is exactly 5 with no remainder.
2. **Conditional Check**: The function then checks whether the remainder `mod` is zero. In this case, since `mod` is 0 (`divmod` returned `(5, 0)`), our input number is divisible by 3.
3. **Calculating the Consecutive Integers**: Knowing that 15 is divisible by 3, the function determines that it can be expressed as the sum of three consecutive integers. It uses the quotient (`x = 5`) to find these numbers. They are the integers `(x - 1)`, `x`, and `(x + 1)`, thus `(4, 5, 6)`.
4. **Returning the Result**: Since the remainder was zero, the function returns the list `[4, 5, 6]`. These numbers are indeed consecutive, and their sum is 15, meeting the original problem's requirements.

Hence, for `num = 15`, the `sumOfThree` function yields `[4, 5, 6]`.

Let us consider a scenario where `sumOfThree` is called with `num = 16`:

1. **Division and Modulus Operations**: By calling `divmod(16, 3)`, the function gets `(5, 1)`, meaning the quotient is 5 and the remainder is 1.
2. **Conditional Check**: Since the remainder `mod` is not zero, we conclude that 16 cannot be divided evenly by 3.
3. **Calculating the Consecutive Integers**: There is no need for this step because we've already determined that the number cannot be divided into three consecutive integers.
4. **Returning the Result**: The function returns an empty list `[]`, indicating that there is no sequence of three consecutive numbers summing up to 16.

In this case, for `num = 16`, the `sumOfThree` function correctly responds with `[]`, as there are no whole consecutive numbers that add up to 16.

Solution Implementation

Python

```python
from typing import List  # Import List type from typing module for type hints.

class Solution:
    def sumOfThree(self, num: int) -> List[int]:
        # Divide the number by 3 to find if there exists a sequence of 3 numbers that add up to 'num'.
        quotient, remainder = divmod(num, 3)

        # If the remainder is not zero, 'num' cannot be expressed as a sum of 3 consecutive numbers.
        # Hence, return an empty list.
        if remainder != 0:
            return []

        # If the remainder is zero, construct the list of 3 consecutive numbers.
        # Since 'quotient' is the middle number of the three consecutive numbers, we return
        # a list containing 'quotient' - 1, 'quotient', and 'quotient' + 1.
        return [quotient - 1, quotient, quotient + 1]

# The function sumOfThree can now be used to check for any number if it can be expressed
# as a sum of three consecutive integers. For example, calling sumOfThree(33) will return [10, 11, 12].
```

Java

```java
class Solution {
    // This method finds if the given number can be expressed as the sum of three consecutive integers.
    public long[] sumOfThree(long num) {
        // Check if the number is divisible by 3 to ensure it can be the sum of three consecutive integers.
        if (num % 3 != 0) {
            // If the number is not divisible by 3, return an empty array because it's not possible to find such three numbers.
            return new long[] {};
        }

        // Calculate the middle number of the three consecutive inteoers.
        // This works because the sum of three consecutive integers is always divisible by 3.
        long middleNumber = num / 3;

        // Return an array containing the three numbers that sum up to the given number.
        // middleNumber - 1, middleNumber, and middleNumber + 1 are three consecutive integers.
        return new long[] {middleNumber - 1, middleNumber, middleNumber + 1};
    }
}
```

C++

```cpp
#include <vector> // Include the header for using the vector container.

// Define the Solution class as before.
class Solution {
public:
    // Define the sumOfThree function which takes a long long integer 'num' and returns a vector of long long integers.
    std::vector<long long> sumOfThree(long long num) {
        // Check if 'num' is divisible by 3 since the sum of any three consecutive numbers is divisible by 3.
        if (num % 3 != 0) {
            // If 'num' is not divisible by 3, return an empty vector as it can't be the sum of three consecutive integers.
            return {};
        }

        // If it is divisible, calculate the middle number by dividing 'num' by 3.
        long long middle = num / 3;

        // The three consecutive numbers would then be (middle - 1), 'middle', and (middle + 1).
        // Return the vector containing these three numbers.
        return {middle - 1, middle, middle + 1};
    }
};
```

TypeScript

```typescript
/**
 * Finds if a number can be expressed as a sum of three consecutive numbers.
 * If possible, it returns the three consecutive numbers as an array, otherwise returns an empty array.
 *
 * @param {number} num - The number to be checked.
 * @returns {number[]} An array containing three consecutive numbers that sum up to 'num', or an empty array if not possible.
 */
function sumOfThree(num: number): number[] {
    // Check if 'num' is not divisible by 3. If so, there cannot be three consecutive numbers that add up to it.
    if (num % 3 !== 0) {
        return [];
    }

    // Calculate the middle number of the three consecutive numbers.
    const middleNumber = Math.floor(num / 3);

    // Return an array consisting of three consecutive numbers.
    return [middleNumber - 1, middleNumber, middleNumber + 1];
}
```

```python
from typing import List  # Import List type from typing module for type hints.

class Solution:
    def sumOfThree(self, num: int) -> List[int]:
        # Divide the number by 3 to find if there exists a sequence of 3 numbers that add up to 'num'.
        quotient, remainder = divmod(num, 3)

        # If the remainder is not zero, 'num' cannot be expressed as a sum of 3 consecutive numbers.
        # Hence, return an empty list.
        if remainder != 0:
            return []

        # If the remainder is zero, construct the list of 3 consecutive numbers.
        # Since 'quotient' is the middle number of the three consecutive numbers, we return
        # a list containing 'quotient' - 1, 'quotient', and 'quotient' + 1.
        return [quotient - 1, quotient, quotient + 1]

# The function sumOfThree can now be used to check for any number if it can be expressed
# as a sum of three consecutive integers. For example, calling sumOfThree(33) will return [10, 11, 12].
```

Time and Space Complexity

The given function `sumOfThree` aims to find if the input number `num` can be expressed as the sum of three consecutive integers.

Time Complexity:

The time complexity of the function is `O(1)`. This is because the function consists of only a few constant-time operations:

1. The `divmod` function is called once, which takes constant time.
2. Checking the result of `divmod` using an `if` condition which is again a constant-time operation.
3. Returning a list with three elements or an empty list, both of which are constant-time operations.

There are no loops or recursive calls that depend on the size of the input, so the running time of the algorithm does not scale with the size of `num`.

Space Complexity:

The space complexity of the function is also `O(1)`. This is because the function uses a fixed amount of space:

1. Storing the quotient `x` and the modulus `mod` uses a constant amount of space.
2. The output list has at most three integers, which is a constant size irrespective of the input size.

Therefore, no additional space that grows with the input size is required.