829. Consecutive Numbers Sum

Hard Math Enumeration

Problem Description

In this problem, we are given an integer `n` and are required to find the number of ways `n` can be expressed as the sum of consecutive positive integers. For example, the number 5 can be expressed as `2+3` or `5` alone, which are two ways of writing 5 as the sum of consecutive numbers.

Intuition

The solution to this problem is based on the idea of arithmetic progressions (AP). When you write `n` as a sum of consecutive numbers, you're essentially creating an arithmetic progression where the common difference is 1. For example, `5=2+3` represents the AP (2, 3), while `9=2+3+4` represents the AP (2, 3, 4), and so on.

We know that the sum of the first `k` terms of an AP is given by the formula `k * (first term + last term) / 2`. So, if `n` is the sum of `k` consecutive numbers starting from `x`, we have `n = k * (x + (x + k - 1)) / 2`. Simplifying this equation, we get `2n = k(2x + k - 1)`.

We look for all pairs `(k, x)` such that the equation is true. Since both `k` and `x` must be positive, `k` must be less than or equal to 2 times `n`. We only need to check for `k` satisfying this constraint. For every positive integer `k`, we verify whether `2n` is divisible by `k` and whether the resulting start term `x` is a positive integer by checking `(2n/k + 1 - k) % 2 == 0`. If these conditions are met, that means we found a valid group of consecutive numbers that sum up to `n`, and we increment our answer counter.

Each `k` that satisfies the condition adds to the number of ways `n` can be expressed as the sum of consecutive positive integers. The process is continued until `k * (k + 1)` becomes greater than `2 * n`, because larger `k` would result in non-positive `x`.

Solution Approach

The solution uses a single `while` loop, which continues until `k * (k + 1)` exceeds `2 * n`. Here's the breakdown of the implementation:

- Initially, the input `n` is multiplied by 2 for easier calculation (`n <<= 1`). This is because we derived `2n = k(2x + k - 1)` and we are going to use `2n` in our condition checks.
- We initialize `ans` to 0, which will hold the number of ways `n` can be expressed as the sum of consecutive positive numbers, and `k` to 1, which represents the length of consecutive numbers starting from `x`.
- The `while` loop runs as long as `k * (k + 1) <= 2 * n`. This is because, for a given `k`, if `k * (k + 1) > 2 * n`, `x` would not be positive, as derived from our equation. In other words, `k` represents the number of terms in the sequence. If the product of `k` and `k+1` (i.e., the sum of an AP where the first term is 1 and the last term is `k`) is greater than `2n`, then `x`, the starting term, would be less than 1, which is not allowed.
- Inside the loop, we check two conditions for a valid sequence:
  - `2n % k == 0`: This checks if `k` is a divisor of `2n`. If not, it's impossible to express `n` as the sum of `k` consecutive numbers.
  - `(2n // k + 1 - k) % 2 == 0`: After finding a `k` for which `2n` is divisible, this needs to be true for `x` to be a positive integer. It's derived from rearranging and simplifying the equation `2n = k(2x + k - 1)`; solving for `x` gives `x = (2n/k + 1 - k)/2`. This condition ensures that `x` is an integer.
- If both conditions are satisfied, we increment `ans` by one since we have found a valid grouping.
- After checking for a `k`, we increment `k` by 1 and proceed to check for the next possible sequence length.
- The loop ends when no more values of `k` satisfy the condition that `k * (k + 1) <= 2 * n`, at which point we return `ans` as the total number of ways `n` can be represented as the sum of consecutive positive integers.

This approach uses neither additional data structures nor complex algorithms but relies on mathematical properties of numbers and arithmetic progressions.

Example Walkthrough

Let's walk through an example using the number `n = 15` to illustrate the solution approach.

We want to find out how many different ways we can express `15` as the sum of consecutive positive integers. Let's apply the mentioned solution approach step by step:

- Begin by doubling `n`, which gives `2*n = 30`.
- Initialize `ans` to `0` to keep track of valid expressions, and `k` to `1` as the potential length of our consecutive numbers starting from some `x`.

Now we run the loop as long as `k * (k + 1) <= 2 * n`. For `k = 1`:

- Check if `30 % 1 == 0` (Is `k` a divisor of `30`?). The answer is yes.
- Check if `(30 // 1 + 1 - 1) % 2 == 0` (Will `x` be a positive integer?). Simplified, `(30 + 1 - 1) % 2 == 0`, so yes.
- Both conditions are met, increment `ans` to `1`.

Increment `k` to `2`:

- Check if `30 % 2 == 0`. Yes.
- Check if `(30 // 2 + 1 - 2) % 2 == 0`. We have `(15 + 1 - 2) % 2 == 14 % 2 == 0`. It's true again.
- Increment `ans` to `2`.

Increment `k` to `3`:

- Check if `30 % 3 == 0`. Yes.
- Check if `(30 // 3 + 1 - 3) % 2 == 0`. We have `(10 + 1 - 3) % 2 == 8 % 2 == 0`. It's true.
- Increment `ans` to `3`.

Increment `k` to `4`:

- `30 % 4` is not `0`, so `k = 4` does not satisfy the condition.

Increment `k` to `5`:

- `30 % 5 == 0`. Yes.
- Check if `(30 // 5 + 1 - 5) % 2 == 0`. We have `(6 + 1 - 5) % 2 == 2 % 2 == 0`. True.
- Increment `ans` to `4`.

Proceeding like this:

- For `k = 6`, `30 % 6 == 0` but `(30 // 6 + 1 - 6) % 2 != 0`. It's invalid.
- For `k = 7`, `30 % 7` is not `0`.
- ...

We continue until `k * (k + 1)` is greater than `30`. At `k = 8`, `k * (k + 1)` becomes `64`, which is greater than `30`, so we break the loop.

The value stored in `ans` at the end of this process will be `4`, which means there are four ways to represent the number `15` as the sum of consecutive positive integers:

1. `15` alone.
2. `7 + 8`.
3. `4 + 5 + 6`.
4. `1 + 2 + 3 + 4 + 5`.

This walkthrough shows a practical application of the described solution approach and how it systematically finds all possible consecutive sequences that sum up to the provided `n`.

Solution Implementation

Python

```python
class Solution:
    def consecutiveNumbersSum(self, n: int) -> int:
        # Multiply n by 2 for simplifying the (x + x + k - 1) * k / 2 = n equation
        n <<= 1

        # Initialize the count of ways n can be written as a sum of consecutive numbers
        count = 0

        # Starting with the smallest possible sequence length k = 1
        k = 1

        # While the sum of the first k consecutive numbers is less than or equal to n
        while k * (k + 1) <= n:
            # Check if n is divisible by k (for a valid sequence)
            # and if the sequence starting number is a whole number
            if n % k == 0 and (n // k + 1 - k) % 2 == 0:
                # If conditions are met, increment the count of possible ways
                count += 1
            # Move to the next possible sequence length
            k += 1

        # Return the total count of ways n can be expressed as a sum of consecutive numbers
        return count
```

Java

```java
class Solution {
    public int consecutiveNumbersSum(int N) {
        // Multiply N by 2 to simplify the calculations below
        N <<= 1;

        int answer = 0; // Initialize the answer to count the number of ways

        // Iterate over possible values of k, where k is the number of consecutive integers
        for (int k = 1; k * (k + 1) <= N; ++k) {
            // Check if there is a sequence of k consecutive numbers adding up to N
            // To check that, we need to see if we can write N as k * m, where m is the median of the sequence.
            if (N % k == 0) {
                // Calculate the median of the sequence
                int medianTimesTwo = N / k + 1 - k; // The median times 2, to simplify even/odd check

                // Ensure the median of the sequence is a whole number
                if (medianTimesTwo % 2 == 0) {
                    // If we have a valid median, we have found one way to write N
                    // as a sum of consecutive integers
                    ++answer;
                }
            }
        }

        // Return the answer
        return answer;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the number of ways to express 'n' as a sum of
    // consecutive positive numbers.
    int consecutiveNumbersSum(int n) {
        // Multiplying n by 2 to simplify the (k * (k + 1) <= n) comparison later.
        n <<= 1;

        // Initialize the count of different ways to express 'n' to 0.
        int countOfWays = 0;

        // Iterate over all possible lengths 'k' of consecutive numbers.
        // The maximum length k can reach is when k * (k + 1) is equal to 2 * n.
        for (int k = 1; k * (k + 1) <= n; ++k) {
            // Check if 2n is divisible by k, which means it's possible to solve
            // the equation 2n = k * (k + 1) for some integer start value of the sequence.
            if (n % k == 0) {
                // Further check if the start value of the sequence (n/k + 1 - k) is even.
                // This means there exists a sequence of k consecutive numbers which sums to n,
                // since for the equation n = start + ... + (start + k - 1), you need `start`
                // to be an integer.
                if ((n / k + 1 - k) % 2 == 0) {
                    // If both conditions are met, increment the count of different ways.
                    ++countOfWays;
                }
            }
        }
        // Return the total count of different ways 'n' can be written as a sum of
        // consecutive positive numbers.
        return countOfWays;
    }
};
```

TypeScript

```typescript
// Function to calculate the number of ways to express 'n' as a sum of
// consecutive positive numbers.
function consecutiveNumbersSum(n: number): number {
    // Multiplying n by 2 to simplify the (k * (k + 1) <= n) comparison later.
    n *= 2;

    // Initialize the count of different ways to express 'n' to 0.
    let countOfWays: number = 0;

    // Iterate over all possible lengths 'k' of consecutive numbers.
    // The maximum length k can reach is when k * (k + 1) is equal to 2 * n.
    for (let k = 1; k * (k + 1) <= n; ++k) {
        // Check if 2n is divisible by k, which means it's possible to solve
        // the equation 2n = k * (k + 1) for some integer start value of the sequence.
        if (n % k === 0) {
            // Further check if the start value of the sequence (n/k + 1 - k) is even.
            // This means there exists a sequence of k consecutive numbers which sums to n,
            // since for the equation n = start + ... + (start + k - 1), you need `start`
            // to be an integer.
            if ((n / k + 1 - k) % 2 === 0) {
                // If both conditions are met, increment the count of different ways.
                ++countOfWays;
            }
        }
    }
    // Return the total count of different ways 'n' can be written as a sum of
    // consecutive positive numbers.
    return countOfWays;
}
```

```python
class Solution:
    def consecutiveNumbersSum(self, n: int) -> int:
        # Multiply n by 2 for simplifying the (x + x + k - 1) * k / 2 = n equation
        n <<= 1

        # Initialize the count of ways n can be written as a sum of consecutive numbers
        count = 0

        # Starting with the smallest possible sequence length k = 1
        k = 1

        # While the sum of the first k consecutive numbers is less than or equal to n
        while k * (k + 1) <= n:
            # Check if n is divisible by k (for a valid sequence)
            # and if the sequence starting number is a whole number
            if n % k == 0 and (n // k + 1 - k) % 2 == 0:
                # If conditions are met, increment the count of possible ways
                count += 1
            # Move to the next possible sequence length
            k += 1

        # Return the total count of ways n can be expressed as a sum of consecutive numbers
        return count
```

Time and Space Complexity

The given Python code is designed to find the number of ways to express a given positive integer `n` as a sum of consecutive positive integers.

Time Complexity

The time complexity of the code is determined by the while loop, which iterates until `k * (k + 1) <= n`. Since `n` is doubled at the beginning (`n <<= 1`), the actual breaking condition for the loop is `k * (k + 1) <= 2n`.

We need to find the maximum value of `k` at which the loop stops. This is when `k(k + 1) = 2n`, solving for `k` yields `k = sqrt(2n)`, which is the maximum number of iterations the loop can run. Therefore, the time complexity of the algorithm is `O(sqrt(n))`.

Space Complexity

The space complexity of the code is `O(1)`. The reason for this constant space complexity is that the algorithm only uses a fixed number of integer variables (`ans`, `k`, and `n`) which do not scale with the input size. No additional data structures that grow with the input size are used in this algorithm.