1716. Calculate Money in Leetcode Bank

Easy Math

Problem Description

Hercy has a consistent saving plan to buy his first car, where he deposits money into his Leetcode bank account every day following a specific pattern. The pattern works as follows:

- On the first Monday (first day), he deposits $1.
- From the second day (Tuesday) to the seventh day (Sunday), he increases the amount deposited by $1 each day.
- On every Monday after the first, he deposits $1 more than what he deposited on the previous Monday.

Given a number `n`, which represents the number of consecutive days Hercy has been making these deposits, the task is to find the total amount of money Hercy will have saved in the Leetcode bank at the end of the `n`th day.

Intuition

The problem can be thought of as finding the sum of an arithmetic series with a twist. The main idea is to break down the problem into smaller parts:

- Compute the total money deposited in full weeks. If `n` is a multiple of 7, this is straightforward; for partial weeks, we only need to account for the days that fall in the final, incomplete week.
- Remember that each full week starts with Hercy depositing $1 more than the previous week's Monday. This hints at a pattern similar to the sum of natural numbers, but with an offset due to the amount increasing each week.

For each full week:

- Hercy deposits an amount that starts at $1 on the first Monday and increases by 1$ each subsequent Monday.
- There is a base amount deposited every week: $1 (Monday) + 2$ (Tuesday) + $3 (Wednesday) + ... + 7$ (Sunday) = $28.
- Each full week has an additional amount on top of the base $28, determined by the count of weeks that have passed.

For the partial week (if `n` is not a multiple of 7), we need to calculate how many days we're into the new week (`b`) and the total amount he has deposited that week, given the extra dollar he adds at the start of each week.

Thus, the problem involves finding the sum of amounts deposited in full weeks and the additional amount in the last, possibly partial, week.

The solution code applies this logic using a mathematical formula:

- It uses the `divmod` function to get both the quotient `a` (full weeks completed) and the remainder `b` (additional days in the incomplete week).
- The term `28 + 28 + 7 * (a - 1)` calculates the base amount for the first and last full week, with the middle term adapting the arithmetic series formula to account for the increasing starting amount each Monday.
- The term `a * 2 + b + 1` calculates the amount for the partial week, considering the increase in the starting amount each Monday.

By neatly summing the contributions of full weeks and the partial week, the code computes the desired total amount.

Solution Approach

The solution to this problem involves using simple arithmetic and understanding of how to sum an arithmetic series. Here's how the provided implementation works, step by step:

- First, the use of the `divmod` function is crucial. This function divides `n` by `7` to separate the problem into full weeks (`a`) and the remaining days of a partial week (`b`). For instance, if Hercy has been depositing money for 10 days, `a` would be 1 (representing 1 full week), and `b` would be 3 (because there are 3 days in the second week so far).
- With these values, the solution calculates how much money was saved in the full weeks. This is where patterns in arithmetic series are used. The base amount without extra increments is known to be $28 per full week. However, each subsequent week, Hercy starts with 1$ more than he did the previous Monday. So for `a` full weeks, the first week has a base of $28, and the last full week would have a base of 28$ + 7 * (a - 1). The total sum for full weeks follows the natural numbers sum formula:

  ```
  Sum for full weeks = (base_first_week + base_last_week) * number_of_full_weeks / 2
  ```

  Which translates into code as:

  ```
  (28 + 28 + 7 * (a - 1)) * a // 2
  ```

- For the remaining days of the partial week (`b`), we need to account for the initial value of the week. By the formula, `a * 2 + b + 1` is the amount Hercy deposits on the last day of this partial week. The sum for these `b` days is again an arithmetic sequence starting from `a + 1` (since it's a new week, he starts with `$1` more than the last Monday) to `a + b`. This gives us:

  ```
  Sum for partial week = (initial_deposit + final_deposit) * number_of_days / 2
  ```

  And in code this is:

  ```
  (a * 2 + b + 1) * b // 2
  ```

- Finally, by adding the sum for full weeks to the sum for the partial week, we get the total amount of money saved by Hercy after `n` days. The solution wisely combines these calculations in a single return statement for efficiency and brevity.

The provided solution is effective and efficient because it reduces the problem to a series of arithmetic operations without the need for complex data structures or algorithms. The entire solution runs in constant time (O(1)), which means it requires the same amount of time to compute the answer regardless of the input size `n`.

Example Walkthrough

Let's illustrate the solution approach using a small example: Assume Hercy has been depositing money for 10 days (`n = 10`).

First, we divide 10 by 7, which gives us a full week (`a = 1`) and a partial week of 3 days (`b = 3`).

For the full week:

- The base amount of money saved without additional increments is $28.
- Since there's only one full week (`a = 1`), the amount deposited on the first and last Monday of the full weeks would be the same, $1. So, there's no increase for the last full week as there's only one full week in this example.

For the partial week:

- Hercy starts the new week by depositing $2 on Monday, as it is 1$ more than what he deposited on the last Monday of the previous full week.
- He then deposits $3 on Tuesday and 4$ on Wednesday, following the increasing pattern of $1 more each day.

Now, calculating the sums:

- For the full week, using the arithmetic series formula, we get: `(28 + 28 + 7 * (a - 1)) * a / 2` which translates to `(28 + 28 + 7 * (1 - 1)) * 1 / 2 = 28`.
- For the partial week, the amount deposited on the last day is `$4`. Thus, using the arithmetic series formula again, we have: `(a * 2 + b + 1) * b / 2` which is `(1 * 2 + 3 + 1) * 3 / 2 = (2 + 3 + 1) * 3 / 2 = 9`.

Finally, we add the sums from the full and partial weeks to get the total amount saved after 10 days: `28 (from the full week) + 9 (from the partial week) = $37`.

So, Hercy will have saved a total of $37 in his Leetcode bank account at the end of the 10th day. This walkthrough demonstrates how the solution efficiently calculates the total amount deposited using arithmetic series properties.

Solution Implementation

Python

```python
class Solution:
    def totalMoney(self, n: int) -> int:
        # Calculate full weeks and remaining days
        full_weeks, remaining_days = divmod(n, 7)

        # Calculate total money after complete weeks
        # The sum of an arithmetic progression for a week is 28 (1+2+...+7),
        # then it increases by 7 for each subsequent week.
        week_start_sum = 28  # The sum for the first week
        week_end_sum = week_start_sum + 7 * (full_weeks - 1)  # Calculate the sum for the last week
        total_full_weeks_money = (week_start_sum + week_end_sum) * full_weeks // 2

        # Calculate total money for the remaining days
        # The money each day starts at full weeks + 1 and increases by 1 each day
        remaining_start_money = full_weeks + 1  # Starting money for the first remaining day
        remaining_end_money = remaining_start_money + remaining_days - 1 # Money for the last remaining day
        total_remaining_days_money = (remaining_start_money + remaining_end_money) * remaining_days // 2

        # Sum the money from full weeks and remaining days
        total_money = total_full_weeks_money + total_remaining_days_money
        return total_money
```

Java

```java
class Solution {

    /**
     * Calculates the total amount of money after n days, with the money amount increasing every day and resetting every week.
     *
     * @param n Total number of days.
     * @return Total amount of money accumulated over n days.
     */
    public int totalMoney(int n) {
        // Calculate the number of complete weeks.
        int completeWeeks = n / 7;

        // Calculate the remaining days after the complete weeks.
        int remainingDays = n % 7;

        // Calculate the total money saved during the complete weeks.
        // The first week, a person saves 1+2+...+7 = 28. The amount increases by 7 for every subsequent week.
        int totalCompleteWeeksMoney = (28 + (28 + 7 * (completeWeeks - 1))) * completeWeeks / 2;

        // Calculate the total money saved during the remaining days.
        // Starting day of the week determines the money saved on the first day of remaining days.
        int totalRemainingDaysMoney = ((completeWeeks * 2) + remainingDays + 1) * remainingDays / 2;

        // Return the sum of money saved during the complete weeks and the remaining days.
        return totalCompleteWeeksMoney + totalRemainingDaysMoney;
    }
}
```

C++

```cpp
class Solution {
public:
    int totalMoney(int n) {
        // Calculate the number of complete weeks
        int numWeeks = n / 7;
        // Calculate the remaining days after complete weeks
        int remainingDays = n % 7;

        // Calculate the total money for the complete weeks
        // The total for each full week forms an arithmetic progression starting from 28
        // First week total: 28 => 1+2+3+4+5+6+7
        // Second week total: 28 + 7 => (1+1)+(2+1)+(3+1)+(4+1)+(5+1)+(6+1)+(7+1), and so on.
        // General formula for the sum of an arithmetic series: n/2 * (first term + last_term)
        // Sum for weeks: (first week total + last week total) / 2 * numWeeks
        int totalMoneyWeeks = (28 + 28 + 7 * (numWeeks - 1)) * numWeeks / 2;

        // Calculate the money for the remaining days
        // This also forms an arithmetic sequence starting from some number depending on the number of passed weeks
        // Starting with the next day's money after the last complete week
        int startDayMoney = numWeeks + 1;
        // Last day's money of the additional days
        int endDayMoney = numWeeks + remainingDays;
        // Sum for remaining days: (startDayMoney + endDayMoney) * remainingDays / 2
        int totalMoneyRemaining = (startDayMoney + endDayMoney) * remainingDays / 2;

        // Total money is the sum of money saved in complete weeks and remaining days
        return totalMoneyWeeks + totalMoneyRemaining;
    }
};
```

TypeScript

```typescript
// Calculates the total amount of money saved over 'n' days
function totalMoney(n: number): number {
    // Calculate the number of complete weeks
    const numWeeks: number = Math.floor(n / 7);
    // Calculate the remaining days after complete weeks
    const remainingDays: number = n % 7;

    // Calculate the total money saved during the complete weeks
    // The total for each complete week forms an arithmetic progression starting from 28
    // Sum for weeks is calculated using the formula for the sum of the first n terms of an arithmetic series
    const firstWeekTotal: number = 28;
    const lastWeekTotal: number = 28 + 7 * (numWeeks - 1);
    const totalMoneyWeeks: number = (firstWeekTotal + lastWeekTotal) * numWeeks / 2;

    // Calculate the money saved during the remaining days, which is part of an arithmetic progression
    // Starting from the money that would be saved on the next day if there was another complete week
    const startDayMoney: number = numWeeks + 1;
    // Ending on the last remaining day's money
    const endDayMoney: number = numWeeks + remainingDays;
    // Sum for the remaining days is again calculated using the formula for the sum of arithmetic series
    const totalMoneyRemaining: number = (startDayMoney + endDayMoney) * remainingDays / 2;

    // The total money saved is the sum of the money saved during complete weeks and the remaining days
    return totalMoneyWeeks + totalMoneyRemaining;
}
```

```python
class Solution:
    def totalMoney(self, n: int) -> int:
        # Calculate full weeks and remaining days
        full_weeks, remaining_days = divmod(n, 7)

        # Calculate total money after complete weeks
        # The sum of an arithmetic progression for a week is 28 (1+2+...+7),
        # then it increases by 7 for each subsequent week.
        week_start_sum = 28  # The sum for the first week
        week_end_sum = week_start_sum + 7 * (full_weeks - 1)  # Calculate the sum for the last week
        total_full_weeks_money = (week_start_sum + week_end_sum) * full_weeks // 2

        # Calculate total money for the remaining days
        # The money each day starts at full weeks + 1 and increases by 1 each day
        remaining_start_money = full_weeks + 1  # Starting money for the first remaining day
        remaining_end_money = remaining_start_money + remaining_days - 1 # Money for the last remaining day
        total_remaining_days_money = (remaining_start_money + remaining_end_money) * remaining_days // 2

        # Sum the money from full weeks and remaining days
        total_money = total_full_weeks_money + total_remaining_days_money
        return total_money
```

Time and Space Complexity

The given Python function `totalMoney` calculates the total amount of money based on a weekly saving pattern. Here is the analysis of its time complexity and space complexity:

Time Complexity

The time complexity of the function is `O(1)`. This is because the calculation involves a constant number of arithmetic operations (addition, multiplication, modulus, and integer division), regardless of the value of `n`. These operations do not depend on iteration or recursion that would otherwise affect the time complexity based on the size of `n`. Hence, the calculation time is constant.

Space Complexity

The space complexity of the function is also `O(1)`. It uses a fixed amount of additional space: two variables `a` and `b` to store the quotient and remainder from the division of `n` by 7, and a few auxiliary variables for intermediate calculations. As the space used does not scale with the input size `n`, it is constant space complexity.