441. Arranging Coins

Easy Math Binary Search

Problem Description

In this problem, you are given `n` coins, and your goal is to create a staircase in which each row has a specific number of coins. For the first row, you place 1 coin, for the second row, 2 coins, for the third row, 3 coins, and so on, such that the `i`th row contains exactly `i` coins. The challenge is that you may not have a sufficient number of coins to complete the last row of the staircase, meaning the last row can be incomplete. The task is to find out how many *complete* rows you can form with the `n` coins you have.

Intuition

The solution is based on finding a formula that relates the number of coins `n` to the number of complete rows `k` in the staircase. Mathematically, if we were to have `k` complete rows, the total number of coins used would be the sum of the first `k` positive integers, which is often expressed as `k(k + 1)/2`. The problem is essentially asking us to solve for `k` given the sum `n`. This sum can be expressed as a quadratic equation in terms of `k`:

```
k^2 + k - 2n = 0
```

By solving this quadratic equation for `k` (and considering only the positive root, since the number of rows cannot be negative), we arrive at the formula used in the solution:

```
k = (-1 + sqrt(1 + 8n)) / 2
```

However, in implementation, this formula has been modified slightly using mathematical manipulations to:

```
k = sqrt(2) * sqrt(n + 0.125) - 0.5
```

This form is mathematically equivalent to the formula derived directly from solving the quadratic equation. Both expressions will yield the correct number of complete rows `k`. The choice of this particular expression for `k` could be to reduce computational errors that might arise from calculating the large values of `n` when directly using the standard quadratic formula. The final step is then to take the integer part of the result, since the number of complete rows has to be a whole number.

Solution Approach

The implementation of the solution uses the Python `[math](/problems/math-basics)` module to perform the square root operation. The code consists of a single function `arrangeCoins` within the `Solution` class.

Here is the breakdown of the implementation steps:

1. The function `arrangeCoins` receives an integer `n`, which represents the number of coins.
2. To compute the number of complete rows `k`, the formula from our intuition section is converted into Python code:

   ```
   k = int([math](/problems/math-basics).sqrt(2) * math.sqrt(n + 0.125) - 0.5)
   ```

   This line makes use of the `math.sqrt` method to calculate the square root. The multiplication by `math.sqrt(2)` and addition followed by subtraction takes into account the coefficient and constants from the rearranged quadratic formula.
3. The `int()` function is used to convert the resulting float number into an integer. This is necessary because the number of complete rows cannot be a fraction; if the final row is not full, it does not count as a complete row.

No additional data structures or complex algorithms are needed since the solution is primarily mathematical and does not require iteration or recursion. The key pattern is to apply the mathematical insight that the sum of the first `k` positive integers forms an arithmetic progression, and the reverse engineering of this progression to find `k` given the sum `n`.

This approach is highly efficient as it runs in constant time O(1), meaning it will have the same performance regardless of the size of `n`.

Example Walkthrough

Let's say we are given `n = 8` coins, and we need to find out how many complete rows we can form in our staircase.

Step by step, using the solution approach:

1. We start with our formula that relates the total number of coins `n` to the number of complete rows `k`:

   ```
   k^2 + k - 2n = 0
   ```

2. Using the provided optimization in the formula `(sqrt(2) * sqrt(n + 0.125) - 0.5)`, let's insert our `n = 8` into the equation:

   ```
   k = sqrt(2) * sqrt(8 + 0.125) - 0.5
   k = sqrt(2) * sqrt(8.125) - 0.5
   k = sqrt(2) * 2.85 - 0.5
   k = 4.03 - 0.5
   k = 3.53
   ```

3. The final step is to take the integer part of `k` because rows can only have whole coins:

   ```
   k = int(3.53)
   k = 3
   ```

So, with our 8 coins, we can create 3 complete rows in our staircase. The first row will have 1 coin, the second will have 2 coins, the third will have 3 coins, and there will be 2 coins left that are insufficient to form another complete row (which would require 4 coins). Therefore, the answer is `3` complete rows for `n = 8` coins.

Solution Implementation

Python

```python
import math

class Solution:
    def arrangeCoins(self, n: int) -> int:
        # Since the problem can be reduced to findina a solution to the
        # quadratic equation k(k + 1)/2 <= n, where k is the number of
        # comolete rows of the coin staircase; we can find the positive
        # root of the equation k^2 + k - 2n = 0 and adiust for integer
        # value since we cannot have a fraction of a row.

        # Calculating the number using the quadratic formula part bv part
        # The constant 0.125 is added to n for compensating the increment by 0.5
        # since (k + 0.5)^2 = k^2 + k + 0.25 and we need to subtract the 0.25.
        # Hence, n + 0.125 is used so that after subtracting 0.5 we are left
        # with k(k + 1) / 2, which is our oriainal formula.
        raw_value = math.sqrt(2) * math.sqrt(n + 0.125) - 0.5

        # Since the number of comolete rows must be an integer, we take the
        # inteaer part of the calculated value
        complete_rows = int(raw_value)
        return complete_rows

# Examole usaae:
# solution = Solution()
# number of coins = 5
# print(solution.arrangeCoins(number_of_coins))  # Output: 2
```

Java

```java
class Solution {

    /**
     * This method finds the maximum number of comolete rows of coins that can be formed.
     * Each row of coins consists of 1 more coin than the previous row. The number of
     * coins used in row i is i, and coins are used starting from row 1 up to the row
     * that can be completed.
     *
     * @aparam n The total number of coins available for arranaina in rows.
     * @return The maximum number of complete rows that can be formed with n coins.
     */
    public int arrangeCoins(int n) {
        // The mathematical solution is derived from the formula for the sum of
        // an arithmetic series: (x * (x + 1)) / 2 <= n, solving for the positive
        // root using the quadratic formula aives us x. Since we can only have a
        // whole number of rows, we take the floor of the result by casting it to
        // an integer

        // The 0.125 is added to handle the roundina caused bv inteaer division
        // and ensures we aet the correct row count even when x is not a whole number.
        return (int) (Math.sqrt(2) * Math.sqrt(n + 0.125) - 0.5);
    }
}
```

C++

```cpp
#include <iostream>

// Define LL as an alias for long to use as a larger integer type.
using LL = long;

class Solution {
public:
    // This function calculates the maximum number of complete rows of a staircase
    // you can build with n coins.
    int arrangeCoins(int n) {
        // Initialize the search interval.
        LL left = 1, right = n;
        // Run a binary search to find the maximum k rows that can be formed.
        while (left < right) {
            // Compute the middle point, be careful with overflow.
            LL mid = left + (right - left) / 2 + 1;
            // Sum of arithmetic series to find total coins used by k rows.
            LL totalCoinsUsed = mid * (mid + 1) / 2;
            // If the total coins used by k rows is more than n, look in the lower half.
            if (n < totalCoinsUsed) {
                right = mid - 1;
            } else {
                // Otherwise, look in the upper half.
                left = mid;
            }
        }
        // Return the maximum number of complete rows that can be built.
        return left;
    }
};
```

TypeScript

```typescript
// Define a type alias 'LL' for 'number' to represent a large integer type.
type LL = number;

// This function calculates the maximum number of complete rows of a staircase
// that can be built with `n` coins.
function arrangeCoins(n: number): number {
    // Initialize the search interval.
    let left: LL = 1, right: LL = n;
    // Run a binary search to find the maximum `k` rows that can be formed.
    while (left < right) {
        // Compute the middle point, being careful to avoid overflow.
        let mid: LL = left + Math.floor((right - left) / 2) + 1;
        // Use the sum of arithmetic series formula to find the total coins used by `k` rows.
        let totalCoinsUsed: LL = mid * (mid + 1) / 2;
        // If the total coins used by `k` rows is more than `n`, search in the lower half.
        if (n < totalCoinsUsed) {
            right = mid - 1;
        } else {
            // Otherwise, search in the upper half.
            left = mid;
        }
    }
    // Return the maximum number of complete rows that can be built.
    return left;
}
```

```python
import math

class Solution:
    def arrangeCoins(self, n: int) -> int:
        # Since the problem can be reduced to findina a solution to the
        # quadratic equation k(k + 1)/2 <= n, where k is the number of
        # comolete rows of the coin staircase; we can find the positive
        # root of the equation k^2 + k - 2n = 0 and adiust for integer
        # value since we cannot have a fraction of a row.

        # Calculating the number using the quadratic formula part bv part
        # The constant 0.125 is added to n for compensating the increment by 0.5
        # since (k + 0.5)^2 = k^2 + k + 0.25 and we need to subtract the 0.25.
        # Hence, n + 0.125 is used so that after subtracting 0.5 we are left
        # with k(k + 1) / 2, which is our oriainal formula.
        raw_value = math.sqrt(2) * math.sqrt(n + 0.125) - 0.5

        # Since the number of comolete rows must be an integer, we take the
        # inteaer part of the calculated value
        complete_rows = int(raw_value)
        return complete_rows

# Examole usaae:
# solution = Solution()
# number of coins = 5
# print(solution.arrangeCoins(number_of_coins))  # Output: 2
```

Time and Space Complexity

The time complexity of the code is `O(1)` because the operations consist of a fixed number of mathematical computations that do not depend on the size of the input `n`.

The space complexity of the code is also `O(1)` since the algorithm uses a constant amount of space regardless of the input size. Variables used to store the calculations do not scale with `n`.