1414. Find the Minimum Number of Fibonacci Numbers Whose Sum Is K

Medium Greedy Math

Problem Description

The problem presents a scenario where you are provided an integer `k`, and your task is to find the minimum number of Fibonacci numbers that can be combined to equal `k`. Importantly, it's allowed to use the same Fibonacci number more than once in the sum.

The Fibonacci sequence mentioned here starts with `F1 = 1` and `F2 = 1`, and each subsequent number is the sum of the previous two (`Fn = Fn-1 + Fn-2` for `n > 2`). The question guarantees that it's always possible to find a combination of Fibonacci numbers that sum up to the given `k`.

To put it simply, the challenge is to break down the number `k` into a sum consisting of the least possible number of elements from the Fibonacci sequence.

Intuition

The intuition behind approaching this problem is to utilize a greedy algorithm. Since we need to minimize the number of Fibonacci numbers, we should always try to fit the largest possible Fibonacci number into the sum without exceeding `k`. Each time we find such a number, we subtract it from `k` and continue with the process until `k` is reduced to 0. This approach ensures that at each step, the largest possible chunk is taken out of `k`, thereby ensuring the minimum number of steps.

To implement this method, we'll need to generate Fibonacci numbers on the fly and keep track of the two most recent ones, as any Fibonacci number can be obtained by summing up the previous two. As soon as we calculate a Fibonacci number that is larger than `k`, we'll set aside the last number computed as the largest Fibonacci number to be used and subtract it from `k`. This step is recursively repeated until `k` becomes 0, at which point the total count of Fibonacci numbers used gives us the answer.

Solution Approach

The implementation of the solution follows a recursive depth-first search (DFS) strategy with a greedy algorithm to find the minimum number of Fibonacci numbers that sum up to `k`.

Here is a step-by-step walkthrough of the solution code provided:

1. Define a recursive function `dfs` that takes the current value of `k` as its argument:
   - If `k` is less than 2, we can directly return `k` because if it's either 0 or 1, it's already a Fibonacci number and no further decomposition is needed.
   - Initialize two variables `a` and `b`, both with the value `1`, corresponding to the first two Fibonacci numbers (`F1` and `F2`).
   - Use a `while` loop to find the largest Fibonacci number that is less than or equal to `k`. This is done by continuously updating `a` and `b` with the next Fibonacci numbers in the sequence (where `b` takes the value of `a + b` and `a` takes the previous value of `b`). When `b` becomes larger than `k`, the loop breaks.
   - The function then calls itself with the reduced value `k - a`, which is `k` minus the largest Fibonacci number less than or equal to `k`. This step represents subtracting the largest Fibonacci chunk from the current number. It also adds `1` to the result, counting the used Fibonacci number.
2. The outer function `findMinFibonacciNumbers` defined in the `Solution` class invokes the `dfs` function with the value `k` and returns its result. The recursion will eventually reach `k` equal to 0, at which point the recursion unwinds and the total count of the Fibonacci numbers used is obtained.

The algorithm uses a recursive DFS approach to explore all possibilities in reducing `k` with Fibonacci chunks and the greedy strategy ensures that the solution is both effective and optimal for the given problem constraints. There is no need for additional data structures beyond the variables holding the two most recent Fibonacci numbers and the recursive stack.

Example Walkthrough

Let's consider an example with `k = 10`. Our goal is to find the minimum number of Fibonacci numbers that, when summed, equal `k`.

We start with the first two Fibonacci numbers being `1` (F1 and F2). As per the steps mentioned:

1. Since `k` is greater than 2, we proceed to find the largest Fibonacci number less than or equal to `k`. We keep generating Fibonacci numbers until we find that `8` is the largest Fibonacci number less than or equal to `10`. Here's the sequence: `1, 1, 2, 3, 5, 8`.
2. We now subtract this largest Fibonacci number from `k`: `10 - 8 = 2`. This means we've used `1` Fibonacci number (`8`) so far.
3. We now set `k` to the result from the subtraction above, which is `2`. We begin the process again to find the largest Fibonacci number less than or equal to `2`. In this case, `2` itself is a Fibonacci number.
4. We subtract `2` from `k`, resulting in `0`: `2 - 2 = 0`. We've used one more Fibonacci number (`2`).

At this point, our sum of Fibonacci numbers equals the original `k` (`8 + 2 = 10`), and `k` has been reduced to `0`. We have used `2` Fibonacci numbers altogether (number `8` and number `2`), and since we cannot lower the count further, the minimum number of Fibonacci numbers that sum up to the given `k` is `2`.

Thus, for `k = 10`, the minimum number of Fibonacci numbers required is `2`. This illustrates the greedy approach that the algorithm uses by choosing the largest possible Fibonacci numbers starting from the biggest one less than or equal to `k` to minimize the total count.

Solution Implementation

Python

```python
class Solution:
    def findMinFibonacciNumbers(self, k: int) -> int:
        # Helper function to find the minimum number of Fibonacci numbers whose sum is equal to k.
        def find_min_fib_nums(k):
            # Base case: if k is less than 2, it's already a Fibonacci number (0 or 1).
            if k < 2:
                return k

            # Initialize two Fibonacci numbers, a and b, starting with 1.
            a, b = 1, 1

            # Iterate to find the largest Fibonacci number less than or equal to k.
            while b <= k:
                # Update the previous two Fibonacci numbers.
                a, b = b, a + b

            # Once b is larger than k, a is the largest Fibonacci number less than or equal to k.
            # Use recursive call to find the sum of Fibonacci numbers for the remainder (k - a).
            # The 1 added represents the count for the Fibonacci number 'a' used in the sum.
            return 1 + find_min_fib_nums(k - a)

        # Initiate the recursive process starting with the original input k.
        return find_min_fib_nums(k)
```

Java

```java
class Solution {
    // Method to find the minimum number of Fibonacci numbers whose sum is equal to k.
    public int findMinFibonacciNumbers(int k) {
        // Base case: for k < 2, the result would be k itself as it is the sum
        // of 0 and 1, or just 1 in the Fibonacci sequence.
        if (k < 2) {
            return k;
        }

        int first = 1; // Initialize the first Fibonacci number.
        int second = 1; // Initialize the second Fibonacci number.

        // Generate Fibonacci numbers until the current number exceeds or is equal to k.
        while (second <= k) {
            second = first + second; // Update the second number to the next Fibonacci number.
            first = second - first;  // Update the first number to the previous second number.
        }

        // Recursive call, find the remaining number (k - first) which is the nearest
        // Fibonacci number less than k, and add 1 to the count.
        return 1 + findMinFibonacciNumbers(k - first);
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the minimum number of Fibonacci numbers whose sum is equal to k.
    int findMinFibonacciNumbers(int k) {
        // If k is less than 2, it's already a Fibonacci number (either 0 or 1).
        if (k < 2) return k;

        // Initialize two variables to represent the last two Fibonacci numbers.
        int prev = 1;     // Represents the second-to-last Fibonacci number.
        int curr = 1;     // Represents the last Fibonacci number.

        // Generate Fibonacci numbers until the current number is greater than k.
        while (curr <= k) {
            int temp = curr;
            curr = prev + curr; // Update curr to the next Fibonacci number.
            prev = temp;        // Update prev to the previous curr value.
        }

        // After finding the first Fibonacci number greater than k, subtract the
        // second-to-last Fibonacci number from k (prev is the largest Fibonacci number less than or equal to k).
        // Add 1 to the result since we include the prev Fibonacci number in the sum,
        // and recursively call the function to find the rest.
        return 1 + findMinFibonacciNumbers(k - prev);
    }
};
```

TypeScript

```typescript
// Array of Fibonacci numbers in descending order.
const fibonacciNumbers = [
    1836311903, 1134903170, 701408733, 433494437, 267914296, 165580141, 102334155, 63245986,
    39088169, 24157817, 14930352, 9227465, 5702887, 3524578, 2178309, 1346269, 832040, 514229,
    317811, 196418, 121393, 75025, 46368, 28657, 17711, 10946, 6765, 4181, 2584, 1597, 987, 610,
    377, 233, 144, 89, 55, 34, 21, 13, 8, 5, 3, 2, 1,
];

/**
 * Finds the minimum number of Fibonacci numbers whose sum is equal to a given integer k.
 * @param {number} k - The target number to represent as a sum of Fibonacci numbers.
 * @returns {number} - The minimum count of Fibonacci numbers that sum up to k.
 */
function findMinFibonacciNumbers(k: number): number {
    let resultCount = 0; // Initialize the count of Fibonacci numbers used.

    // Iterate over the array of Fibonacci numbers.
    for (const number of fibonacciNumbers) {
        // If the current Fibonacci number can be subtracted from k, use it.
        if (k >= number) {
            k -= number; // Subtract the current number from k.
            resultCount++; // Increment the count as we have used one Fibonacci number.

            // If k becomes zero, we found a complete sum, so break the loop.
            if (k === 0) {
                break;
            }
        }
    }

    // Return the total count of Fibonacci numbers used to represent k.
    return resultCount;
}
```


```python
class Solution:
    def findMinFibonacciNumbers(self, k: int) -> int:
        # Helper function to find the minimum number of Fibonacci numbers whose sum is equal to k.
        def find_min_fib_nums(k):
            # Base case: if k is less than 2, it's already a Fibonacci number (0 or 1).
            if k < 2:
                return k

            # Initialize two Fibonacci numbers, a and b, starting with 1.
            a, b = 1, 1

            # Iterate to find the largest Fibonacci number less than or equal to k.
            while b <= k:
                # Update the previous two Fibonacci numbers.
                a, b = b, a + b

            # Once b is larger than k, a is the largest Fibonacci number less than or equal to k.
            # Use recursive call to find the sum of Fibonacci numbers for the remainder (k - a).
            # The 1 added represents the count for the Fibonacci number 'a' used in the sum.
            return 1 + find_min_fib_nums(k - a)

        # Initiate the recursive process starting with the original input k.
        return find_min_fib_nums(k)
```


Time and Space Complexity

Time Complexity

The time complexity of the given solution largely depends on how many recursive calls we make, which in turn is based on the value of `k` and the Fibonacci numbers generated.

For each call to `dfs(k)`, we perform a loop that generates Fibonacci numbers until the largest Fibonacci number less than or equal to `k` is found. The number of iterations of this loop is bounded by the index `n` of the Fibonacci number that is closest to `k` in value, where `F_n ≈ φ^n / sqrt(5)` and φ (phi) is the golden ratio `(1 + sqrt(5)) / 2` ≈ `1.618`.

The time complexity of generating each Fibonacci number is `O(1)`, but finding the correct Fibonacci number for subtraction requires looping over the Fibonacci sequence, which has a time complexity of `O(log(k))` since the Fibonacci numbers grow exponentially.

Recursively, we subtract the found Fibonacci number from `k` and repeat the process. In the worst case, we might have to go as far as the first Fibonacci numbers for each recursive call if `k` is a large Fibonacci number itself, leading to a time complexity of `O(log(k)) * O(log(k))` = `O(log^2(k))`.

Space Complexity

The space complexity of the solution is determined by the maximum depth of the recursion stack. Since we subtract the largest Fibonacci number less than or equal to `k` at every step, the maximum depth of the recursion is also `O(log(k))`. Every call to `dfs(k)` uses `O(1)` space, except for the space used in the recursive call stack.

Thus, the total space complexity is `O(log(k))`.