

509. Fibonacci Number

Easy

Recursion

Memoization

Math

Dynamic Programming

Problem Description

The problem is to find the n th number in the Fibonacci sequence. The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. The sequence starts with 0 and 1. Mathematically, it can be defined as:

$$F(0) = 0, F(1) = 1$$
$$F(n) = F(n - 1) + F(n - 2), \text{ for } n > 1.$$

The task is to write a function that, given a non-negative integer n , returns the n th Fibonacci number.

Intuition

To solve this problem, we think about the properties of the Fibonacci sequence. Since $F(n)$ depends only on the two previous numbers, $F(n - 1)$ and $F(n - 2)$, we can compute it using an iterative approach. We start with the first two numbers, 0 and 1, and repeat the process of adding the last two numbers to get the next number until we reach the n th term. This removes the need for [recursion](#), which can be inefficient and possibly lead to a stack overflow for large values of n .

The intuition behind the iterative solution is based on the observation that we don't need to retain all the previous numbers computed - just the last two numbers to calculate the next number in the sequence. This leads to a time and space efficient algorithm.

Solution Approach

The solution implements an iterative approach to calculate Fibonacci numbers. The main algorithm used here doesn't require complex data structures or patterns - it simply relies on variable swapping and updating values in each iteration.

Here's a step-by-step explanation of the code:

- We initialize two variables `a` and `b` to the first two Fibonacci numbers, 0 and 1, respectively. These two variables will keep track of the last two numbers in the sequence at each step.
- We then enter a loop that will iterate `n` times. On each iteration, we simulate the progression of the sequence.
- Inside the loop, we have a single line of code that performs the update:

```
a, b = b, a + b
```

This line is a tuple unpacking feature in Python, which allows for the simultaneous update of `a` and `b`. Here's the breakdown:

- `b` is assigned to `a`, which moves the sequence one step forward.
- `a + b` is the sum of the current last two numbers, producing the next number in the sequence, and it is assigned to `b`.

This operation is repeated until we have looped `n` times, by which point `a` will contain the n th Fibonacci number, and the function returns `a`.

One important note is that the looping starts from 0 and goes to `n-1`, thus iterating `n` times. The reason for this is that we start counting from 0 in the Fibonacci sequence, so after `n` iterations, we've already achieved the n th term.

There are no recursive calls, which makes this algorithm run in $O(n)$ time complexity, which is the number of iterations equal to `n`, and $O(1)$ space complexity, because we're only ever storing two values, regardless of the size of `n`.

Example Walkthrough

Let's illustrate the solution approach with an example where we want to find the 5th number in the Fibonacci sequence.

- We start by initializing two variables `a` and `b` with the first two Fibonacci numbers, 0 and 1, respectively. So `a = 0` and `b = 1`.
- We need to loop from 0 to `n-1`, where `n` is 5 in this example, since we want to find the 5th number in the sequence.
- The loop starts, and at each iteration, we will perform the following operation:

```
a, b = b, a + b
```

- Let's see how the values of `a` and `b` change with each iteration:
 - Iteration 1: `a = 1, b = 1` (`0 + 1`)
 - Iteration 2: `a = 1, b = 2` (`1 + 1`)
 - Iteration 3: `a = 2, b = 3` (`1 + 2`)
 - Iteration 4: `a = 3, b = 5` (`2 + 3`)

After 4 iterations (which is `n-1` for `n=5`), we can stop since `a` now holds the value of the 5th Fibonacci number.

- The function will then return `a`, which is `3`. So, the 5th number in the Fibonacci sequence is 3.

Now implementing this approach with actual Python code would look like this:

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

# Example usage:
print(fibonacci(5)) # Output will be 3
```

This walkthrough demonstrates how the algorithm works with a small example and assures us that the result of `fibonacci(5)` is indeed the 5th number in the Fibonacci sequence according to our zero-indexing in the sequence definition.

Solution Implementation

Python

```
class Solution:
    def fib(self, N: int) -> int:
        # Initialize the first two Fibonacci numbers
        previous, current = 0, 1

        # Iterate N times to calculate the N-th Fibonacci number
        for _ in range(N):
            # Update the previous and current values to move one step forward in the Fibonacci sequence
            previous, current = current, previous + current

        # After N iterations, previous holds the value of the N-th Fibonacci number
        return previous
```

Java

```
class Solution {
    public int fib(int n) {
        // Initializing the first two numbers of the Fibonacci sequence.
        int previousNumber = 0; // Previously known as 'a'.
        int currentNumber = 1; // Previously known as 'b'.

        // Looping to calculate Fibonacci sequence until the nth number.
        while (n-- > 0) {
            // Calculate the next number in the Fibonacci sequence.
            int nextNumber = previousNumber + currentNumber;

            // Update the previous number to be the current number.
            previousNumber = currentNumber;

            // Update the current number to be the next number.
            currentNumber = nextNumber;
        }

        // After completing the loop, 'previousNumber' holds the nth Fibonacci number.
        return previousNumber;
    }
}
```

C++

```
class Solution {
public:
    int fib(int n) {
        // Initialize the first two fibonacci numbers
        int previous = 0, current = 1;

        // Loop to calculate the nth fibonacci number
        while (n--) {
            // Calculate the next fibonacci number
            int next = previous + current;

            // Update the previous and current values for the next iteration
            previous = current;
            current = next;
        }

        // Return the nth fibonacci number which is now stored in 'previous'
        return previous;
    }
};
```

TypeScript

```
// Function to calculate the nth Fibonacci number
function fib(n: number): number {
    // Initialize the first two Fibonacci numbers
    let currentFib = 0; // The first Fibonacci number, F(0)
    let nextFib = 1; // The second Fibonacci number, F(1)

    // Iterate until the nth number
    for (let i = 0; i < n; i++) {
        // Update the current and next numbers using tuple assignment
        // currentFib becomes the nextFib, and nextFib becomes the sum of currentFib and nextFib
        [currentFib, nextFib] = [nextFib, currentFib + nextFib];
    }

    // Return the nth Fibonacci number
    return currentFib;
}

class Solution:
    def fib(self, N: int) -> int:
        # Initialize the first two Fibonacci numbers
        previous, current = 0, 1

        # Iterate N times to calculate the N-th Fibonacci number
        for _ in range(N):
            # Update the previous and current values to move one step forward in the Fibonacci sequence
            previous, current = current, previous + current

        # After N iterations, previous holds the value of the N-th Fibonacci number
        return previous
```

Time and Space Complexity

Time Complexity

The provided code consists of a single loop that iterates `n` times, where `n` is the input number to calculate the Fibonacci sequence. In each iteration of the loop, a constant number of operations are performed, specifically the assignments `a, b = b, a + b`. Therefore, this loop runs in linear time with respect to the input `n`. This gives us a time complexity of $O(n)$.

Space Complexity

The space complexity of the code is constant, as it only uses a fixed number of variables (`a` and `b`) regardless of the input size. This means no additional space is used that scales with the input size `n`, leading to a space complexity of $O(1)$.