2648. Generate Fibonacci Sequence

Easy

Leetcode Link

Problem Description

The task is to implement a generator function in TypeScript that produces a generator object capable of yielding values from the Fibonacci sequence indefinitely. The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, typically starting with 0 and 1. That is to say, $X_n = X_{(n-1)} + X_{(n-2)}$ for n greater than 1, with X_0 being 0 and X_1 being 1. The function we create should be able to produce each Fibonacci number when requested.

Intuition

To solve this problem, we have to understand what a generator is and how it works. A generator in TypeScript is a function that can be paused and resumed, and it produces a sequence of results instead of a single value. This fits perfectly with the requirement to yield an indefinite sequence, like the Fibonacci series, since we don't want to compute all the numbers at once, which would be inefficient and impractical for a potentially infinite series.

The intuition behind the Fibonacci sequence generator is relatively straightforward. We start with two variables, `a` and `b`, which represent the two latest numbers in the sequence. We initialize `a` to 0 and `b` to 1, which are the first two numbers in the Fibonacci sequence. In each iteration of our generator function, we yield the current value of `a`, which is part of the sequence, and then update `a` and `b` to the next two numbers.

This update is done by setting `a` to `b` and `b` to `a + b`, effectively shifting the sequence by one. The array destructuring syntax `[a, b] = [b, a + b]` in TypeScript makes this step concise and clear. Because our generator is designed to run indefinitely and the Fibonacci sequence has no end, we enclose our logic in an infinite `while` loop. This allows consumers of the generator to keep requesting the next number in the sequence for as long as they need without ever running out of values to yield.

Solution Approach

The implementation of the Fibonacci sequence generator is elegantly simple, leaning heavily on the capabilities of TypeScript's generator functions. The generator pattern is ideal for this kind of problem since it allows for the creation of a potentially infinite sequence where each element is produced on demand.

Here's a breakdown of the algorithm step by step:

1. We first declare a generator function `fibGenerator` by using the `function*` syntax. This indicates that the function will be a generator.
2. Inside the function, we initialize two variables `a` and `b` to start the Fibonacci sequence. `a` starts at `0`, and `b` starts at `1`.
3. We then enter an infinite `while` loop with the condition `true`. This loop will run indefinitely until the consumer of the generator decides to stop requesting values.
4. Inside the loop, we `yield a`. The `yield` keyword is what makes this function a generator. It pauses the function execution and sends the value of `a` to the consumer. When the consumer asks for the next value, the function resumes execution right after the `yield` statement.
5. After yielding `a`, we perform the Fibonacci update step using array destructuring: `[a, b] = [b, a + b]`. This step calculates the next Fibonacci number by summing the two most recent numbers in the sequence. `a` is updated to the current value of `b`, and `b` is updated to the sum of the old `a` and `b`.
6. The loop then iterates, and the process repeats, yielding the next number in the Fibonacci sequence.

By using a generator function, we can maintain the state of our sequence (the last two numbers) across multiple calls to `.next()`. We avoid having to store the entire sequence in memory, which allows the function to produce Fibonacci numbers as far into the sequence as the consumer requires without any predetermined limits.

The `fibGenerator` generator function can be used by creating a generator object, say `gen`, and repeatedly calling `gen.next().value` to get the next number in the Fibonacci sequence. This process can go on as long as needed to generate Fibonacci numbers on the fly.

There are no complex data structures needed: the algorithm only ever keeps track of the two most recent values. This is an excellent example of how a generator can manage state internally without the need for external data structures or class properties.

Example Walkthrough

To illustrate the solution approach, let's manually walk through the initial part of running our generator function `fibGenerator`.

1. A generator object `gen` is created by calling the `fibGenerator()` function.
2. We call `gen.next().value` for the first time:
   a. The generator function starts executing and initializes `a` to `0` and `b` to `1`.
   b. It enters the infinite `while` loop.
   c. It hits the `yield a` statement. At this point, `a` is `0`, so it yields `0`.
   d. The `gen.next().value` call returns `0`, which is the first number in the Fibonacci sequence.
3. We call `gen.next().value` for the second time:
   a. The generator function resumes execution right after the `yield` statement.
   b. The Fibonacci update happens: `[a, b] = [b, a + b]` results in `a = 1` and `b = 1`.
   c. The next iteration of the loop starts, and `yield a` yields `1`.
   d. The call returns `1`, which is the second number in the Fibonacci sequence.
4. We call `gen.next().value` for the third time:
   a. The function resumes and updates `a` and `b` again. Now `a = 1` (previous `b`) and `b = 2` (previous `a + b`).
   b. It yields `a`, which is `1`.
   c. The call returns `1`, which is the third Fibonacci number.
5. This process can continue indefinitely, with `gen.next().value` being called to get the next Fibonacci number each time. The next few calls would return `2`, `3`, `5`, `8`, `13`, and so on.

Each call to `.next()` incrementally advances the generator function's internal state, computes the next Fibonacci number, and yields it until the next call. By following these steps, we don't calculate all the Fibonacci numbers at once, nor do we run out of numbers to yield — respecting the definition of an infinite series.

Python Solution

```python
def fib_generator():
    """
    A generator function that yields Fibonacci numbers indefinitely.
    """
    current = 0  # The current number in the sequence, initialized to 0.
    next_num = 1  # The next number in the sequence, initialized to 1.

    while True:
        # Yield the current number before updating.
        yield current
        # Update the current and the next number with the next Fibonacci numbers.
        current, next_num = next_num, current + next_num


# Example usage:
generator = fib_generator()  # Create a new Fibonacci sequence generator.
print(next(generator))  # Outputs 0, the first number in the sequence.
print(next(generator))  # Outputs 1, the second number in the sequence.
# To continue obtaining values from the generator, repeatedly call next(generator).

```

Java Solution

```java
import java.util.Iterator;

// Class that implements an Iterator to generate Fibonacci numbers indefinitely.
public class FibGenerator implements Iterator<Integer> {

    private int current = 0; // The current number in the sequence, initialized to 0.
    private int next = 1;    // The next number in the sequence, initialized to 1.

    // hasNext method is always true since Fibonacci sequence is infinite.
    @Override
    public boolean hasNext() {
        return true; // Fibonacci numbers can be generated indefinitely.
    }

    // The next method returns the current number in the sequence and advances.
    @Override
    public Integer next() {
        int temp = current; // To store the current number to be returned.
        current = next;     // Update current to the next number in sequence.
        next = next + temp; // Calculate the next number in the sequence and update.
        return temp;        // Return the previous value of current.
    }
}

/*
// Example usage:
FibGenerator generator = new FibGenerator(); // Create a new Fibonacci sequence generator.
System.out.println(generator.next()); // Outputs 0, the first number in the sequence.
System.out.println(generator.next()); // Outputs 1, the second number in the sequence.
// To continue obtaining values from the generator, call generator.next().
*/

```

C++ Solution

```cpp
#include <iostream>
#include <tuple>

// A generator-like class for Fibonacci numbers.
class FibGenerator {
public:
    // Constructor initializes the first two Fibonacci numbers.
    FibGenerator() : current(0), next(1) {}

    // The function that returns the next Fibonacci number.
    int getNext() {
        int returnValue = current;                // The value to be returned.

        // Update the current and next number to the next pair in the Fibonacci sequence.
        std::tie(current, next) = std::make_pair(next, current + next);

        return returnValue;                       // Return the current Fibonacci number.
    }

private:
    int current;  // The current number in the sequence.
    int next;     // The next number in the sequence.
};

/*
// Example usage:
int main() {
    FibGenerator generator;  // Create a new Fibonacci sequence generator.

    std::cout << generator.getNext() << std::endl; // Outputs 0, the first number in the sequence.
    std::cout << generator.getNext() << std::endl; // Outputs 1, the second number in the sequence.

    // To continue obtaining values from the generator, call generator.getNext().

    return 0;
}
*/

```

Typescript Solution

```typescript
// A generator function that yields Fibonacci numbers indefinitely.
function* fibGenerator(): Generator<number> {
    let current = 0; // The current number in the sequence, initialized to 0.
    let next = 1;    // The next number in the sequence, initialized to 1.

    // An infinite loop to continuously yield Fibonacci numbers.
    while (true) {
        yield current;   // Yield the current number.
        [current, next] = [next, current + next]; // Update the current and next numbers.
    }
}

/*
// Example usage:
const generator = fibGenerator(); // Create a new Fibonacci sequence generator.
console.log(generator.next().value); // Outputs 0, the first number in the sequence.
console.log(generator.next().value); // Outputs 1, the second number in the sequence.
// To continue obtaining values from the generator, call generator.next().value.
*/

```

Time and Space Complexity

Time Complexity

The time complexity of the `fibGenerator` function is `O(n)` for generating the first `n` Fibonacci numbers. This is because the generator yields one Fibonacci number per iteration, and the computation of the next number is a constant-time operation (simple addition and assignment).

Space Complexity

The space complexity of the generator is `O(1)`. The function maintains a fixed number of variables (`a` and `b`) regardless of the number of Fibonacci numbers generated, so it does not use additional space that grows with `n`.