2803. Factorial Generator

Easy

Problem Description

The task is to write a generator function that produces a sequence of factorials for a given integer `n`. A factorial of a non-negative integer `n` is the product of all positive integers less than or equal to `n`. It's denoted as `n!` and is calculated as `n! = n * (n - 1) * (n - 2) * ... * 2 * 1` with the special case of `0!` equaling `1`. The challenge here involves creating a function that, when invoked, generates each value in the factorial sequence up to `n` on-demand, rather than computing them all at once.

Intuition

To solve this problem, we use a generator in TypeScript, which allows us to yield values one at a time whenever the generator's `next()` method is called. This fits our goal perfectly, as we want to produce factorial values in a sequence, without computing them all upfront.

The intuition behind the solution is to maintain a running product that starts at 1 (`0!` is 1). As we iterate from 1 to `n`, we multiply the running product by the current number `i` to get `i!`, and then we yield this result. This way, we get each factorial value in turn—from `1!` to `n!`—and any code using this generator can retrieve the next value in the factorial sequence whenever required without calculating the entire sequence ahead of time.

Solution Approach

To implement the solution, a Generator function in TypeScript is used, identified by the `function*` syntax and enabling the `yield` keyword within its body. Here's the step-by-step approach used in the provided solution:

1. The generator checks if the input `n` is 0. Since the factorial of 0 is defined as 1, it yields `1` immediately.

```
if (n === 0) {
    yield 1;
}
```

2. We declare a variable `ans` to keep track of the running product of the series, initializing it to 1 (`0!`).
3. We use a `for` loop to iterate from 1 to `n`. On each iteration, we update `ans` by multiplying it by the current number `i`, which effectively calculates `i!`.

```
for (let i = 1; i <= n; ++i) {
    ans *= i;
    yield ans;
}
```

4. During each iteration of the loop, after updating `ans` to hold the factorial of `i`, we use `yield` to produce the current value of `ans`. This allows the caller to retrieve the values of the factorial sequence one by one on-demand.

The algorithm efficiently computes each factorial value in the sequence by building on the previous value, thus avoiding recomputation of factorials. This is a fundamental principle in dynamic programming, although the solution does not store all previously computed values, just the most recent product.

Data structures:

- The generator itself serves as a custom iterable data structure, allowing the caller of this function to receive factorial values incrementally.

Patterns used:

- **Generator Pattern** - Allows the function to yield multiple values over time, rather than computing them all at once and returning them.
- **Iteration** - Uses a simple loop to traverse from 1 to `n`, aligning with the sequence of factorial calculation.
- **In-place Computation** - Updates the `ans` variable in each iteration which holds the latest factorial value.

Overall, this approach maximizes efficiency minimizing both time and space complexity, as it calculates each factorial product when needed without storing all results or performing redundant calculations.

Example Walkthrough

Let's illustrate the solution approach with a small example where `n = 4`. The goal is to generate the sequence of factorials for the numbers `0` through `4`, which are `1`, `1`, `2`, `6`, and `24`, respectively.

1. The generator checks if `n` is `0`. In our case, since `n` is `4`, it doesn't yield `1` immediately and moves on to the next steps.
2. A variable `ans` is declared and initialized to `1`, which represents the factorial for `0` (`0!`).
3. We start a loop from `1` up to and including `n`, which is `4` in our case. In each iteration of the loop, `ans` would be updated as follows:
   - **First iteration (`i = 1`):**
     - `ans *= i` → `ans = 1 * 1`
     - The generator yields the current value of `ans`, which is `1`.
   - **Second iteration (`i = 2`):**
     - `ans *= i` → `ans = 1 * 2`
     - The generator yields the current value of `ans`, which is `2`.
   - **Third iteration (`i = 3`):**
     - `ans *= i` → `ans = 2 * 3`
     - The generator yields the current value of `ans`, which is `6`.
   - **Fourth iteration (`i = 4`):**
     - `ans *= i` → `ans = 6 * 4`
     - The generator yields the current value of `ans`, which is `24`.
4. During each iteration, after `ans` is updated to hold the factorial of `i`, `yield` is used to produce the current `ans` value. This allows the caller to get the factorial values one at a time on-demand.

Assuming the calling code continually invokes the generator's `next()` method after each value is yielded, it would receive the sequence of factorial values one by one:

- `next()` → `1` (which is `1!`)
- `next()` → `2` (which is `2!`)
- `next()` → `6` (which is `3!`)
- `next()` → `24` (which is `4!`)

In real-world usage, this can be very memory efficient, as we compute each factorial as needed without storing every single result. This method is particularly useful when working with very large sequences where memory consumption is a concern.

Solution Implementation

Python

```
# This generator function calculates factorial numbers up to n.
# It yields each intermediate factorial result in the sequence.
def factorial(n):
    # Base case: the factorial of 0 is 1.
    if n == 0:
        yield 1

    # Initialize the accumulator variable for the factorial calculation.
    accumulator = 1

    # Iterate from 1 to n, multiplying the accumulator by each number.
    for i in range(1, n + 1):
        accumulator *= i

        # Yield the current result of the factorial up to i.
        yield accumulator

# Usage example:
# Create an instance of the generator for the factorial of 2.
generator = factorial(2)

# Retrieve and print the first result from the generator, which is 1! = 1.
print(next(generator))  # Outputs: 1

# Retrieve and print the second result from the generator, which is 2! = 2.
print(next(generator))  # Outputs: 2
```

Java

```
import java.util.Iterator;

// This class represents an iterable sequence of factorial numbers up to a given n.
public class FactorialSequence implements Iterable<Integer> {
    private final int n;

    // Constructor for the FactorialSequence class
    public FactorialSequence(int n) {
        this.n = n;
    }

    // Method to create and return an iterator over the factorial sequence.
    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            private int current = 0;
            private int accumulator = 1;

            // Checks if the next factorial number is available.
            @Override
            public boolean hasNext() {
                return current <= n;
            }

            // Computes and returns the next factorial number.
            @Override
            public Integer next() {
                if (current == 0 || current == 1) {
                    // Factorial of 0 or 1 is always 1.
                    current++;
                    return accumulator; // 1
                } else {
                    accumulator *= current;
                    current++;
                    return accumulator;
                }
            }
        };
    }

    // Usage example:
    public static void main(String[] args) {
        // Create an instance of the factorial sequence for the factorial of 2.
        FactorialSequence sequence = new FactorialSequence(2);

        // Retrieve and print results from the sequence using an iterator.
        Iterator<Integer> iterator = sequence.iterator();

        // Retrieve and print the first result from the sequence, which is 1! = 1.
        System.out.println(iterator.next()); // Outputs: 1

        // Retrieve and print the second result from the sequence, which is 2! = 2.
        System.out.println(iterator.next()); // Outputs: 2
    }
}
```

C++

```
#include <iostream>
#include <vector>

// This class represents a 'generator' for factorial numbers up to 'n' using precomputation.
class FactorialGenerator {
private:
    std::vector<int> factorials;
    size_t currentIndex;

public:
    // Constructor that precalculates the factorials up to 'n'.
    explicit FactorialGenerator(int n) : currentIndex(0) {
        // Reserve space for the factorial results for optimization.
        factorials.reserve(n + 1);

        // Base case: the factorial of 0 is 1.
        factorials.push_back(1);

        // Compute the factorial of each number in the sequence up to 'n' and store the results.
        int accumulator = 1;
        for (int i = 1; i <= n; ++i) {
            accumulator *= i;
            factorials.push_back(accumulator);
        }
    }

    // Function to get the next factorial in the sequence.
    int next() {
        // Check if there are more factorials to return.
        if (currentIndex < factorials.size()) {
            // Return the next factorial and increment the index.
            return factorials[currentIndex++];
        } else {
            // If there are no more factorials, throw an exception or handle the case as desired.
            throw std::out_of_range("No more factorials in the sequence.");
        }
    }

    // Function to check if there are more factorials to be generated.
    bool hasNext() const {
        return currentIndex < factorials.size();
    }
};

int main() {
    // Usage example:
    // Create an instance of the generator for factorial of 2.
    FactorialGenerator generator(2);

    // Retrieve and print the first result from the generator, which is 1! = 1.
    if (generator.hasNext()) {
        std::cout << generator.next() << std::endl; // Outputs: 1
    }

    // Retrieve and print the second result from the generator, which is 2! = 2.
    if (generator.hasNext()) {
        std::cout << generator.next() << std::endl; // Outputs: 2
    }

    return 0;
}
```

TypeScript

```
// This generator function calculates factorial numbers up to n.
// It yields each intermediate factorial result in the sequence.
function* factorial(n: number): Generator<number> {
    // Base case: the factorial of 0 is 1.
    if (n === 0) {
        yield 1;
    }

    // Initialize the accumulator variable for the factorial calculation.
    let accumulator = 1;

    // Iterate from 1 to n, multiplying the accumulator by each number.
    for (let i = 1; i <= n; ++i) {
        accumulator *= i;

        // Yield the current result of the factorial up to i.
        yield accumulator;
    }
}

// Usage example:
// Create an instance of the generator (iterator) for factorial of 2.
const generator = factorial(2);

// Retrieve and print the first result from the generator, which is 1! = 1.
console.log(generator.next().value); // Outputs: 1

// Retrieve and print the second result from the generator, which is 2! = 2.
console.log(generator.next().value); // Outputs: 2
```

```
# This generator function calculates factorial numbers up to n.
# It yields each intermediate factorial result in the sequence.
def factorial(n):
    # Base case: the factorial of 0 is 1.
    if n == 0:
        yield 1

    # Initialize the accumulator variable for the factorial calculation.
    accumulator = 1

    # Iterate from 1 to n, multiplying the accumulator by each number.
    for i in range(1, n + 1):
        accumulator *= i

        # Yield the current result of the factorial up to i.
        yield accumulator

# Usage example:
# Create an instance of the generator for the factorial of 2.
generator = factorial(2)

# Retrieve and print the first result from the generator, which is 1! = 1.
print(next(generator))  # Outputs: 1

# Retrieve and print the second result from the generator, which is 2! = 2.
print(next(generator))  # Outputs: 2
```

Time and Space Complexity

The time complexity of this generator function is `O(n)`. This is because the loop within the generator runs `n` times, with each iteration involving a constant number of operations (a multiplication and a yield). Therefore, the overall number of operations linearly depends on the input `n`.