2629. Function Composition

Easy

Leetcode Link

Problem Description

Given an array of functions such as `[f1, f2, f3, ..., fn]`, the task is to construct a new function `fn` that represents the function composition of all these functions. Essentially, what this means is that if you have functions `f(x)`, `g(x)`, and `h(x)`, the new function `fn(x)` would apply these functions in a sequence where the output of one function becomes the input to the next. For example, `fn(x)` with the given functions would be equivalent to `f(g(h(x)))`. This nesting should work with any number of functions provided in the input array.

A special rule applies when the array is empty: in this case, the expected function to return is an identity function, which is a function that returns its input value `f(x) = x`. This function does not modify the input value in any way, essentially leaving it unchanged. Every function in the provided array is to be considered as a unary function, meaning it takes a single integer input and produces a single integer output.

Intuition

To construct the composite function `fn`, we can apply a concept from functional programming called `reduceRight`. This allows us to start from the last function in the array and iteratively apply the next function to the result of the current one, accumulating the results backward. Essentially, for any starting input `x`, we pass it through the last function, take that output and pass it through the second-to-last function, and so on until we reach the first function in the array; the final result of this first function is then the result of the composite function `fn(x)`.

The use of `reduceRight` is essential as it ensures the correct order of function application: from the end of the array to the beginning, mimicking the nesting of functions. As mentioned, if the array is empty, then according to functional programming principles, the accumulation will simply return the initial value `x`, which is exactly the behavior of an identity function, hence satisfying that special case as well.

This solution elegantly leverages the array's `reduceRight` JavaScript method to handle the composition logic, which abstracts away the details of iterating through the array and accumulating results for us.

Solution Approach

The implementation for the problem at hand uses the `reduceRight` method, which is a built-in array method in JavaScript. This method applies a function against an accumulator and each value of the array (from right-to-left) to reduce it to a single value. Here, the "accumulator" is not a numerical value, as it often is with sums or products, but a function.

The main goal is to generate a new function that when called with an argument, it will process this argument through a sequence of the given functions from right-to-left.

Here's the step-by-step algorithm described with relevance to the provided TypeScript code:

1. We define a function `compose` that takes an array of functions `functions: F[]` as its parameter. `F` is a type alias for functions that accept a number and return a number `(x: number) => number`.
2. `compose` returns a new function, effectively creating the composition. This returned function uses `reduceRight` to process an input `x: number`.
3. The `reduceRight` method is called on the `functions` array. It takes two parameters: an accumulator (initially the input `x`) and a function from the array `fn`.
4. The callback for `reduceRight` applies the current function in the array `fn` to the accumulator `acc` and returns the result, which becomes the accumulator for the next iteration.
5. This process continues until `reduceRight` has applied every function in the array to the input, starting from the last function and ending with the first.
6. The result of the `reduceRight` operation is the return value of the last (or first, depending on perspective) function application, which is the fully composed function being applied to the initial input `x`.
7. When `compose` is called with an array of functions, it constructs this new function described above, and when the new function is eventually called with an argument, it executes the composed operations in the correct order.

The key to this solution is understanding the `reduceRight` function and how it can be leveraged to perform operations in a specific order, which in this case is the function composition from right-to-left (or last-to-first in terms of array indices).

Example Walkthrough

Let's consider a simple example with three functions `[f1, f2, f3]`. Here, `f1(x) = x + 1`, `f2(x) = x * 2`, and `f3(x) = x - 3`. Our task is to create a new function that composes these three functions, so running `fn(x)` will perform the operations in sequence as `f1(f2(f3(x)))`.

Here's how we would walk through this using the solution approach:

1. We start by defining our three functions:

```
const f1 = x => x + 1;
const f2 = x => x * 2;
const f3 = x => x - 3;
```

2. We then use these functions to create an array of functions:

```
const functions = [f1, f2, f3];
```

3. Next, we write a `compose` function that takes this array of functions:

```
function compose(functions){
  return (x) => functions.reduceRight((acc, fn) => fn(acc), x);
}
```

4. Now we utilize the `compose` function to create our composite function `fn`:

```
const fn = compose(functions);
```

5. Let's walk through the execution of `fn(5)`:
   - The `reduceRight` method starts with the last function, `f3`.
   - For the initial value `x`, we pass 5: `f3(5)`, which gives us `2`.
   - Next, the output of `f3`, which is `2`, is passed to `f2`: `f2(2)`, which gives us `4`.
   - Finally, the output of `f2`, which is `4`, is passed to `f1`: `f1(4)`, which gives us `5`.
6. Therefore, `fn(5)` processes as `f1(f2(f3(5))) = f1(f2(2)) = f1(4) = 5`.

In conclusion, by using the `compose` function, we have successfully created a new function `fn` that will apply a sequence of operations from our original array of functions. If we were to call `fn(x)` with any number as the input, it would apply `f1`, `f2`, and `f3` to that number in the sequence from the last function to the first function.

Python Solution

```python
from typing import List, Callable

# Define a type alias 'FunctionType' for functions that take a number and return a number
FunctionType = Callable[[float], float]

def compose(functions: List[FunctionType]) -> FunctionType:
    """
    Composes an array of functions into a single function.
    The functions are composed from right to left.

    :param functions: A list of functions to be composed.
    :return: A new function that is the result of composing the input functions.
    """
    # Define and return a new function that takes a single argument 'x'
    def composed_function(x: float) -> float:
        # Use 'reduce' to apply each function in the list from right to left
        # to the accumulator 'acc', starting with the initial value 'x'
        return reduce(lambda acc, fn: fn(acc), reversed(functions), x)
    return composed_function

# Use functools.reduce to provide the reduce functionality in Python
from functools import reduce

# Example usage:
# Create a new function 'composed_fn' by composing functions:
# first incrementing a number, then doubling it.
composed_fn = compose([lambda x: x + 1, lambda x: 2 * x])
# Applying 'composed_fn' to 4 should first double it (2 * 4 = 8),
# and then increment the result (8 + 1 = 9), so 'composed_fn(4)' should return 9.
print(composed_fn(4))  # Output: 9
```

Java Solution

```java
import java.util.function.Function;
import java.util.List;
import java.util.Collections;

public class FunctionComposition {

    // Define a functional interface 'FunctionType' for functions that take a number and return a number
    @FunctionalInterface
    interface FunctionType extends Function<Integer, Integer> {
    }

    /**
     * Composes a list of functions into a single function.
     * The functions are composed from right to left.
     *
     * @param functions - A list of functions to be composed.
     * @return A new function that is the result of composing the input functions.
     */
    public static FunctionType compose(List<FunctionType> functions) {
        // Return a function that takes a single argument 'x'
        return (Integer x) -> {
            // Use 'reduce' on the reversed list to apply each function from right to left
            // to the current result 'result', starting with the initial value 'x'
            return functions.stream()
                    .reduce((FunctionType) result -> result,
                            (nextFunction, currentComposition) ->
                                    (x2) -> currentComposition.apply(nextFunction.apply(x2)))
                    .apply(x);
        };
    }

    public static void main(String[] args) {
        // Example usage:
        // Create a new function 'fn' by composing functions:
        // first incrementing a number, then doubling it.
        FunctionType fn = compose(List.of(
            x -> 2 * x, // Doubling function
            x -> x + 1  // Incrementing function
        ));
        // Applying 'fn' to 4 should first double it (2 * 4 = 8),
        // and then increment the result (8 + 1 = 9), so 'fn.apply(4)' should return 9.
        System.out.println(fn.apply(4)); // Output: 9
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional>
#include <numeric>

// Define a type 'FunctionType' for functions that take an int and return an int
using FunctionType = std::function<int(int)>;

/**
 * Composes an array of functions into a single function.
 * The functions are composed from right to left.
 *
 * @param functions - A vector of functions to be composed.
 * @returns A new function that is the result of composing the input functions.
 */
FunctionType compose(const std::vector<FunctionType>& functions) {
    // Return a function that takes a single argument 'x'
    return [functions](int x) -> int {
        // Use 'std::accumulate' with reverse iterators to apply each function
        // in the vector from right to left to the accumulator 'acc',
        // starting with the initial value 'x'
        return std::accumulate(functions.rbegin(), functions.rend(), x,
            [](int acc, const FunctionType& fn) -> int {
                // Apply the current function 'fn' to the accumulator 'acc'
                return fn(acc);
            }); // Initial value for 'acc' is 'x'
    };
}

// Example usage:
int main() {
    // Create a new function 'fn' by composing functions:
    // first incrementing an integer, then doubling it.
    FunctionType fn = compose({[](int x) { return x + 1; }, [](int x) { return 2 * x; }});

    // Applying 'fn' to 4 should first double it (2 * 4 = 8),
    // and then increment the result (8 + 1 = 9), so 'fn(4)' should return 9.
    std::cout << fn(4) << std::endl; // Output: 9

    return 0;
}
```

Typescript Solution

```typescript
// Define a type 'FunctionType' for functions that take a number and return a number
type FunctionType = (x: number) => number;

/**
 * Composes an array of functions into a single function.
 * The functions are composed from right to left.
 *
 * @param functions - An array of functions to be composed.
 * @returns A new function that is the result of composing the input functions.
 */
function compose(functions: FunctionType[]): FunctionType {
    // Return a function that takes a single argument 'x'
    return (x: number): number => {
        // Use 'reduceRight' to apply each function in the array from right to left
        // to the accumulator 'acc', starting with the initial value 'x'
        return functions.reduceRight((acc: number, fn: FunctionType): number => {
            // Apply the current function 'fn' to the accumulator 'acc'
            return fn(acc);
        }, x); // Initial value for 'acc' is 'x'
    };
}

// Example usage:
// Create a new function 'fn' by composing functions:
// first incrementing a number, then doubling it.
const fn = compose([x => x + 1, x => 2 * x]);
// Applying 'fn' to 4 should first double it (2 x 4 = 8),
// and then increment the result (8 + 1 = 9), so 'fn(4)' should return 9.
console.log(fn(4)); // Output: 9
```

Time and Space Complexity

Time Complexity

The time complexity of the `compose` function is $O(mn)$, where n is the number of functions in the `functions` array, and m is the complexity of the individual functions being composed. If we assume that each function in the array has a constant time complexity, then the time complexity would simplify to $O(n)$. This is because the composed function produced by `compose` calls each function in the array exactly once for each invocation, and it does so in a linear sequence using `reduceRight`.

For each call of the returned composed function, `reduceRight` iteratively applies the function calls one by one, starting from the last function to the first function in the array, using the return value of the last function as the input to the previous function. Therefore, the time it takes to execute scales linearly with the number of functions in the array.

Space Complexity

The space complexity of the `compose` function is $O(n)$ primarily due to functional closure. When `compose` returns the composed function, it keeps a closure over the `functions` array. This requires storing a reference to each function in the array, so space used scales linearly with the number of functions.

The actual execution may require additional space, which also depends on the nature of the functions. If none of the functions in the array creates additional space that depends on the size of the input or the number of functions, the overall space complexity of the composed function remains $O(n)$. However, if any individual function has a greater space complexity, this would necessarily increase the space complexity of the composed function.