2626. Array Reduce Transformation

Easy

Leetcode Link

Problem Description

In this problem, we are provided with an array of integers called `nums`, a reducer function `fn`, and an initial value `init`. The task is to produce a single value using a reduction process that applies `fn` to each successive element of `nums`, starting with `init` as the first argument and the first array element as the second argument, and then using the result as the new first argument in the next application with the subsequent array element. This process continues until every element in the array has been processed. The output of the final call to `fn` is what we will return as the result. In case `nums` is an empty array, we should simply return `init`. We are instructed to solve the problem without resorting to the built-in `Array.reduce` method available in many programming languages.

Intuition

To approach this solution, we naturally think of iterating through the array, element by element, and at each step applying the reducer function `fn` to the current result `acc` and the current element `x`. Starting with `init` as our accumulator `acc`, we update `acc` with the result of `fn(acc, x)` after each array element is processed. The use of a loop for iteration is a fundamental tool in programming, allowing us to apply the same operation for each element in a sequence - this fits perfectly with the task of reducing an array. The strength of the reducer pattern is that it's a general concept; the specific behavior is defined by the provided function `fn`, which makes this approach both flexible and powerful.

The given TypeScript solution follows this intuition directly:

1. Initialize the accumulator `acc` with the provided initial value `init`.
2. Iterate through each element `x` in the `nums` array, one by one.
3. For each element `x`, apply the reducer function `fn` to the accumulator `acc` and `x` and update `acc` with the result.
4. After processing all elements, return `acc` as it now represents the accumulation of the sequential applications of `fn`.

By following these steps, we adhere to the instructions and compute the reduced value expected by the problem description.

Solution Approach

The implementation follows a straightforward pattern that's commonly used in functional programming, known as reduction or folding. The idea is to maintain a running total or combined result as we process a sequence of values.

Here's a step-by-step explanation of the solution's implementation:

1. **Initialization**: We start by initializing an accumulator variable `acc` with the value of `init`, which is the initial value provided to us. In functional reduction, the accumulator is the placeholder for the ongoing result.

2. **Looping Through Elements**: We then enter a loop that will iterate over each element in the `nums` array. This is done using a `for...of` statement in TypeScript, a language construct particularly well-suited for iterating over array elements.

3. **Applying the Reducer Function**: Inside the loop, for each element `x`, we call the reducer function `fn` with `acc` and `x` as arguments. The return value of `fn(acc, x)` is then assigned back to `acc`. In mathematical terms, if `fn` is denoted as `f`, this step could be written as $acc = f(acc, x)$. This updates the accumulator with the "reduced" value that incorporates the contribution of the current array element `x`.

4. **Returning the Result**: After the loop has processed all of the elements, the final value of `acc` holds the reduced result of the entire array. We then exit the loop, and the function returns this final value.

5. **Handling Edge Cases**: As per the problem description, if the array is empty (`nums.length === 0`), we simply return the initial value `init`. In the provided implementation, this is implicitly handled, as the loop will not iterate at all for an empty array, and `acc`, which still equals `init`, will be returned.

The algorithm runs in O(n) time complexity, where n is the number of elements in the array since it processes each element exactly once. The space complexity is O(1), as we only use a fixed amount of extra space for the accumulator.

The TypeScript implementation provided uses no complex data structures or patterns beyond a for loop, making it very accessible and easily understandable.

The implementation provided is algorithmically efficient and straightforward, which stands as a testament to the elegance of the reduce pattern.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we are given the following:

- An array `nums` of integers: `[3, 5, 2, 4]`
- A reducer function `fn`, which takes two integers and returns their sum.
- An initial value `init` of `10`.

Our goal is to reduce the array using the function `fn` and `init` as the starting point. Now, let's walk through the solution step by step:

1. **Initialization**: We initialize an accumulator `acc` with the value of `init` which is `10`.
2. **Looping Through Elements**: We start an iteration over each element in the `nums` array:
    - First iteration: Take the first element `3`, apply the reducer function `fn(acc, x) => fn(10, 3)` which returns `13`. Update `acc` to `13`.
    - Second iteration: Take the next element `5`, apply the reducer function `fn(acc, x) => fn(13, 5)` which returns `18`. Update `acc` to `18`.
    - Third iteration: Take the next element `2`, apply the reducer function `fn(acc, x) => fn(18, 2)` which returns `20`. Update `acc` to `20`.
    - Fourth iteration: Take the last element `4`, apply the reducer function `fn(acc, x) => fn(20, 4)` which returns `24`. Update `acc` to `24`.
3. **Returning the Result**: Having processed all elements in the array, `acc` now holds the value `24`, which is the final reduced result.
4. **Handling Edge Cases**: If `nums` were an empty array, the for loop would not run and we would simply return `init`, which is `10` in this case.

In this example, the final output, after reducing the array starting with an initial value of `10`, is `24`. The step-by-step reduction process consolidates all the values into one, applying the reducer function sequentially to each array element and the running accumulation.

Python Solution

```python
from typing import List, Callable

# Type alias for the reducer function which is a callback that will be passed
# to the reduce function. It takes two numbers, `accumulator` and `current_value`,
# and returns a number.
ReducerFunction = Callable[[int, int], int]

def reduce(numbers: List[int], reducer: ReducerFunction, initial_value: int) -> int:
    """
    Applies a reducer function on each element of the `numbers` array, resulting in a single output value.

    :param numbers: The list of numbers to be reduced.
    :param reducer: The function to execute on each element in the list.
    :param initial_value: The initial value to start the accumulation from.
    :return: The reduced value after all elements have been processed.
    """
    # The variable that will accumulate the result of calling the reducer
    # function repeatedly.
    accumulator = initial_value

    # Iterate through each number in the list
    for current_value in numbers:
        # Apply the reducer function to the current accumulator and the
        # current value, then assign the result back to the accumulator.
        accumulator = reducer(accumulator, current_value)

    # Return the final accumulated value after processing all elements.
    return accumulator

```

Java Solution

```java
import java.util.function.BiFunction;

/**
 * Applies a reducer function on each element of the `numbers` array, resulting in a single output value.
 *
 * @param numbers       The array of numbers to be reduced.
 * @param reducer       The function to execute on each element in the array.
 * @param initialValue  The initial value to start the accumulation from.
 * @return The reduced value after all elements have been processed.
 */
public static int reduce(int[] numbers, BiFunction<Integer, Integer, Integer> reducer, int initialValue) {
    // The variable that will accumulate the result of calling the reducer function repeatedly.
    int accumulator = initialValue;

    // Iterate through each number in the array
    for (int currentValue : numbers) {
        // Apply the reducer function to the current accumulator and the current value,
        // then assign the result back to the accumulator.
        accumulator = reducer.apply(accumulator, currentValue);
    }

    // Return the final accumulated value after processing all elements.
    return accumulator;
}

```

C++ Solution

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

// Type alias for the reducer function which is a callback that will be passed to the reduce function.
// It takes two integers, `accumulator` and `currentValue` and returns an integer.
using ReducerFunction = std::function<int(int, int)>;

/**
 * Applies a reducer function on each element of the `numbers` vector, resulting in a single output value.
 *
 * @param numbers The vector of integers to be reduced.
 * @param reducer The function to execute on each element in the vector.
 * @param initialValue The initial value to start the accumulation from.
 * @returns The reduced value after all elements have been processed.
 */
int reduce(const std::vector<int>& numbers, const ReducerFunction& reducer, int initialValue) {
    // The variable that will accumulate the result of calling the reducer function repeatedly.
    int accumulator = initialValue;

    // Iterate through each number in the vector
    for (int currentValue : numbers) {
        // Apply the reducer function to the current accumulator and the current value,
        // then assign the result back to the accumulator.
        accumulator = reducer(accumulator, currentValue);
    }

    // Return the final accumulated value after processing all elements.
    return accumulator;
}

```

Typescript Solution

```typescript
// Type declaration for the reducer function which is a callback that will be passed to the reduce function.
// It takes two numbers, `accumulator` and `currentValue` and returns a number.
type ReducerFunction = (accumulator: number, currentValue: number) => number;

/**
 * Applies a reducer function on each element of the `numbers` array, resulting in a single output value.
 *
 * @param numbers - The array of numbers to be reduced.
 * @param reducer - The function to execute on each element in the array.
 * @param initialValue - The initial value to start the accumulation from.
 * @returns The reduced value after all elements have been processed.
 */
function reduce(numbers: number[], reducer: ReducerFunction, initialValue: number): number {
    // The variable that will accumulate the result of calling the reducer function repeatedly.
    let accumulator: number = initialValue;

    // Iterate through each number in the array
    for (const currentValue of numbers) {
        // Apply the reducer function to the current accumulator and the current value,
        // then assign the result back to the accumulator.
        accumulator = reducer(accumulator, currentValue);
    }

    // Return the final accumulated value after processing all elements.
    return accumulator;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `reduce` function is `O(n)`, where `n` is the number of elements in the `nums` array. This is because the function iterates through each element in the array exactly once.

Space Complexity

The space complexity of the `reduce` function is `O(1)` as it uses a fixed amount of space. The `acc` variable is updated during each iteration, but no additional space that grows with the input size is utilized.