2635. Apply Transform Over Each Element in Array

Easy

Leetcode Link

Problem Description

In this problem, we are given two inputs: an array of integers `arr` and a function `fn`. The function `fn` is a mapping function that takes two arguments, an element from the array and its index, and returns a new value based on some logic. Our goal is to create a new array where each element is the result of applying the mapping function `fn` to the corresponding element in the original array, using its value and index as arguments. The transformed array should follow the rule that `returnedArray[i] = fn(arr[i], i)`. Importantly, we need to achieve this without using the `Array.map` method, which is a built-in method in JavaScript and TypeScript that essentially solves this problem by default. This means we have to manually iterate over the original array and construct the new array.

Intuition

To solve the problem without the built-in `Array.map` method, the intuitive approach is to directly iterate over the array. During iteration, each element can be accessed using its index. We can then apply the mapping function `fn` to both the element and its index to get the transformed value. The original array element is replaced with this new value, building up a transformed array in-place. The key steps in our solution approach involve:

1. Looping through each index of the array: We start from the first element and go all the way to the last.
2. Applying the mapping function: For each element, we call the function `fn` with the element value and its index.
3. Updating the array in-place: We directly replace the original element with the result of the mapping function.

By directly manipulating the original array, we eliminate the need for extra space that would be required for a new array, making the solution efficient in terms of space complexity. This traversal method ensures that every element is touched once and transformed accordingly, leading to a new array that is returned at the end of the function.

Keep in mind that since the original array is modified, it can no longer be used in its original form after the function executes. If preservation of the original array is necessary, a new array would need to be created to hold the transformed values.

Solution Approach

The provided TypeScript function `map` implements the logic required to transform an array based on a provided mapping function `fn`. The key components of the solution are:

1. **Traversal Algorithm**: The core of our solution is a simple for-loop which iterates over each element of the input array `arr`. Inside the loop, the index `i` goes from 0 to `arr.length - 1`, ensuring all elements are covered.

2. **In-Place Update**: For each iteration, the mapping function `fn` is called with the current element `arr[i]` and the current index `i` as arguments. The result of this mapping function, which is expressed mathematically as `fn(arr[i], i)`, replaces the current element in the array. This is done in-place, meaning that the original array is updated with the new values without the use of additional data structures.

3. **Return Structure**: Once the loop has processed all elements, the updated array `arr` is returned. The updated array now contains the transformed values as per the rules of the mapping function `fn`.

The TypeScript code for this solution is as follows:

```typescript
function map(arr: number[], fn: (n: number, i: number) => number): number[] {
    for (let i = 0; i < arr.length; ++i) {
        arr[i] = fn(arr[i], i);
    }
    return arr;
}
```

The for-loop is a fundamental algorithm for array traversal. It accesses elements sequentially, which is efficient especially for an array data structure that allows direct access to its elements via their indices.

This method uses no additional data structures apart from the given array and variables for iteration. There's minimal overhead, making it a space-efficient pattern.

In summary, the solution approach involves iteratively replacing each element in the original array with its mapped value using the provided function `fn`. This approach is straightforward and avoids the complexity that might come from adding or removing elements from arrays or dealing with extra arrays. It's an in-place and space-efficient method that relies on the mechanics of array indexing and loops, fundamental principles in algorithm design.

Example Walkthrough

Let's illustrate the solution approach with a simple example. Suppose we have an array `arr` containing `[1, 2, 3, 4]`, and we want to transform each element by multiplying it by its index using the provided function `fn`.

Our mapping function `fn` could be defined in TypeScript as follows:

```typescript
function multiplyByIndex(value: number, index: number): number {
  return value * index;
}
```

Now, let's walk through the process of transforming the array with this `fn` function:

1. Initialize the array: `arr = [1, 2, 3, 4]`
2. Using the given `map` function, we start the looping process with index `i = 0`.
   - At `i = 0`: Apply `fn(arr[0], 0)`, which is `multiplyByIndex(1, 0)`, resulting in `1 * 0 = 0`. The array now looks like `[0, 2, 3, 4]`.
3. Next, increment the index to `i = 1`.
   - At `i = 1`: Apply `fn(arr[1], 1)`, which is `multiplyByIndex(2, 1)`, resulting in `2 * 1 = 2`. The array updates to `[0, 2, 3, 4]`.
4. Increment the index to `i = 2`.
   - At `i = 2`: Apply `fn(arr[2], 2)`, which is `multiplyByIndex(3, 2)`, resulting in `3 * 2 = 6`. The array now looks like `[0, 2, 6, 4]`.
5. Finally, increment the index to `i = 3`.
   - At `i = 3`: Apply `fn(arr[3], 3)`, which is `multiplyByIndex(4, 3)`, resulting in `4 * 3 = 12`. The array updates to `[0, 2, 6, 12]`.

After the loop concludes, the entire array has been transformed in-place. Each element has been replaced by the product of the original value and its index.

The resulting transformed array is `[0, 2, 6, 12]`. The `map` function then returns this array.

This walkthrough provides an illustration of how the `map` function applies a mapping function `fn` to each element of an array along with its index and updates the array in-place without using any additional data structures or space.

Python Solution

```python
# Applies a transformation function `transform_fn` to each element
# in the input list `input_list` and returns a new list with the
# transformed elements. The `transform_fn` function accepts two
# arguments: the element value and its index in the list.

def map(input_list, transform_fn):
    # Create a new list to hold the transformed elements.
    result_list = []

    # Iterate over each element in the input list.
    for i, value in enumerate(input_list):
        # Apply the transformation function to the current element and its index.
        # Then, store the result in the result list.
        result_list.append(transform_fn(value, i))

    # Return the new list with the transformed elements.
    return result_list
```

Java Solution

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

public class ArrayTransformer {

    /**
     * Applies a transformation function `transformFn` to each element in the input array `inputArray`
     * and returns a new array with the transformed elements.
     * The `transformFn` function accepts two arguments: the element value and its index in the array.
     *
     * @param inputArray  the array of integers to be transformed
     * @param transformFn the transformation function that takes an element and its index
     * @return a new array containing the transformed elements
     */
    public static int[] map(int[] inputArray, BiFunction<Integer, Integer, Integer> transformFn) {
        // Create a new array to hold the transformed elements
        int[] resultArray = new int[inputArray.length];

        // Iterate over each element in the input array
        for (int i = 0; i < inputArray.length; i++) {
            // Apply the transformation function to the current element and its index
            // Then store the result in the corresponding position of the result array
            resultArray[i] = transformFn.apply(inputArray[i], i);
        }

        // Return the new array with the transformed elements
        return resultArray;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional> // For std::function

// Applies a transformation function `transformFn` to each element in the input vector `inputVector`
// and returns a new vector with the transformed elements.
// The `transformFn` function accepts two arguments: the element value and its index in the vector.

std::vector<int> map(const std::vector<int>& inputVector, std::function<int(int, int)> transformFn) {
    // Create a new vector to hold the transformed elements.
    std::vector<int> resultVector;

    // Reserve space in the vector to optimize reallocation if inputVector's size is known.
    resultVector.reserve(inputVector.size());

    // Iterate over each element in the input vector using a traditional for loop
    // to have access to the current element's index.
    for (int i = 0; i < inputVector.size(); ++i) {
        // Apply the transformation function to the current element and its index.
        // Then, push the result onto the result vector.
        resultVector.push_back(transformFn(inputVector[i], i));
    }

    // Return the new vector with the transformed elements.
    return resultVector;
}
```

Typescript Solution

```typescript
// Applies a transformation function `transformFn` to each element in the input array `inputArray`
// and returns a new array with the transformed elements.
// The `transformFn` function accepts two arguments: the element value and its index in the array.

function map(inputArray: number[], transformFn: (value: number, index: number) => number): number[] {
    // Create a new array to hold the transformed elements.
    const resultArray: number[] = [];

    // Iterate over each element in the input array.
    for (let i = 0; i < inputArray.length; i++) {
        // Apply the transformation function to the current element and its index.
        // Then, store the result in the corresponding position of the result array.
        resultArray[i] = transformFn(inputArray[i], i);
    }

    // Return the new array with the transformed elements.
    return resultArray;
}
```

Time and Space Complexity

The time complexity of the provided function is `O(n)`, where `n` is the length of the array `arr`. This is because the function includes a for loop that iterates over each element of the array exactly once.

The space complexity of the function is `O(1)`. This is due to the function modifying the input array in place and not allocating any additional significant space that grows with the input size. The only additional memory used is for the loop counter `i`, which is a constant amount of space.