2634. Filter Elements from Array

Easy

Leetcode Link

Problem Description

In this problem, you are provided with two main elements: an integer array `arr` and a function `fn` which acts as a filter. The goal is to construct a new array, `filteredArr`, that contains only certain elements from the original array `arr`. To decide which elements make it into `filteredArr`, you use the filter function `fn`.

The function `fn` can accept one or two parameters. The first parameter is an element from the array (`arr[i]`), and the second parameter is the index of that element within the array (`i`). The role of `fn` is to evaluate each element (and its index) and return a truthy value if the element should be included in the filtered array or a falsy value if it should not. In JavaScript, a truthy value is any value that, when converted to a boolean, becomes `true`.

The requirement of the problem is to write your own filter logic without using the built-in `Array.filter` method provided by JavaScript. This is a fundamental programming task that reinforces your understanding of array iteration and conditional statements.

Intuition

The intuition behind the solution involves iterating through the given array, analyzing each element using the provided function `fn`, and selectively gathering items into a new array based on the result of that function. The steps to arrive at the solution are quite straightforward:

1. Initialize a new array, let's call it `ans`, to store the elements that satisfy the filtering condition.
2. Loop through each element of the original array `arr` using a `for` loop. As we loop through, we have access to both the current element `arr[i]` and its index `i`.
3. On each iteration, apply the filter function `fn` to the current element and its index. If the function returns a truthy value, include this element in the `ans` array by adding (pushing) it.
4. After going through all the elements in `arr`, end the loop and return the `ans` array as the filtered array.

By following these steps, you effectively replicate the behavior of the `Array.filter` method using basic iteration and conditional logic. This method ensures that the original array remains unchanged while producing a new array that meets the specified filtering criteria.

Solution Approach

The implementation of the solution uses a simple and effective algorithm that involves array traversal and conditional logic:

1. **Array traversal**: We start by iterating through each element of the input array `arr` using a `for` loop. The loop runs from the beginning to the end of the array, granting us access to every element and its corresponding index.

2. **Apply the filter function**: During each iteration of the loop, we invoke the function `fn` with two arguments: the current element `arr[i]` and its index `i`. The purpose of `fn` is to determine whether the element meets the filtering criteria specified.

3. **Conditional logic**: If the function `fn` returns a truthy value, it signifies that the current element passes the filter condition and should be included in the resulting array. We check the result of `fn(arr[i], i)` with a simple `if` condition.

4. **Building the answer array**: When the `if` condition is satisfied, we use the `Array.prototype.push` method to add the current element `arr[i]` to a new array called `ans`. This array is initialized beforehand to store all elements that meet the condition.

5. **Return the result**: After the loop has finished processing all the elements in the input array, the `ans` array now contains all the elements for which `fn(arr[i], i)` returned a truthy value. We return `ans` as the filtered array.

By following these steps, the code effectively filters the input array based on the logic defined in the filter function `fn` without utilizing the built-in `Array.filter` method. This approach demonstrates fundamental programming concepts like loops, conditionals, and array manipulation in TypeScript, which is the language of the provided solution.

Here is the code snippet provided using the steps mentioned above:

```
function filter(arr: number[], fn: (n: number, i: number) => any): number[] {
    const ans: number[] = [];
    for (let i = 0; i < arr.length; ++i) { // Loop through array
        if (fn(arr[i], i)) { // Apply filter function and check for truthy
            ans.push(arr[i]); // Add element to answer array if condition is met
        }
    }
    return ans; // Return filtered array
}
```

Notice that no special data structures are needed for this task. An additional array `ans` is the only requirement to gather filtered results. The given solution is straightforward, adhering to basic algorithmic patterns—an essential practice for any programmer to master.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach:

Assume we have an integer array `arr` and a filter function `fn` defined as follows:

```
const arr = [1, 2, 3, 4, 5];
const fn = (element: number, index: number) => (element % 2 === 0);
```

In this example, our function `fn` is designed to return `true` for even numbers and `false` for odd numbers.

Now, let's use these to walk through the filter function's steps:

1. **Array traversal**: We start by looking at each element of the array `arr`. The array elements are `[1, 2, 3, 4, 5]`.

2. **Apply the filter function**: For each element, we call the function `fn`:

   - `fn(1, 0)` returns `false` because 1 is odd.
   - `fn(2, 1)` returns `true` because 2 is even.
   - `fn(3, 2)` returns `false` because 3 is odd.
   - `fn(4, 3)` returns `true` because 4 is even.
   - `fn(5, 4)` returns `false` because 5 is odd.

3. **Conditional logic**: We check the result of the function for each occurrence:

   - Because the call for `arr[0]` returned `false`, we do not include `1` in the `ans` array.
   - The call for `arr[1]` returned `true`, so `2` is included in the `ans` array.
   - The call for `arr[2]` returned `false`, so `3` is not included.
   - Because `arr[3]` returned `true`, `4` is added to the `ans` array.
   - Finally, `arr[4]` returned `false`, so `5` is left out.

4. **Building the answer array**: As we apply the filter function and evaluate the result, our `ans` array begins to build up with the even numbers `[2, 4]`.

5. **Return the result**: Once we've finished iterating through the entire `arr` array, the function ends, and the final `ans` array `[2, 4]` is returned, which contains only the elements from the array `arr` that are even.

In conclusion, following the above steps and example, the filter function replicates `Array.filter` behavior by creating a new array that only includes elements passing the filtering condition defined by `fn`. The final result is an array of even numbers from the original array, demonstrating how the algorithm effectively selects elements based on specified criteria.

Python Solution

```
def filter_by_function(array, filter_function):
    """
    Filters elements of an array based on a provided filter function.

    :param array: The array to filter.
    :param filter_function: The filter function used to determine if an element
                            should be included in the result array.
                            It takes the element value and index as its arguments.
    :return: A new list with elements that pass the filter function condition.
    """
    result = []  # Initialize an empty list to hold the filtered results
    for index, value in enumerate(array):  # Loop over each item and its index in the array
        if filter_function(value, index):  # If the filter function returns 'true', include the element in the result list
            result.append(value)  # Append the current element to the result list
    return result  # Return the filtered list

```

Java Solution

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.BiPredicate;

/**
 * Filters elements of an array based on a provided filter function.
 * @param array The array to filter.
 * @param filterFunction The filter function used to determine if an element should be included in the result array.
 * It takes the element value and its index in the array as its arguments.
 * @return A new list with elements that pass the filter function condition.
 */
public List<Integer> filter(int[] array, BiPredicate<Integer, Integer> filterFunction) {
    List<Integer> result = new ArrayList<>(); // Initialize an empty list to hold the filtered results
    for (int index = 0; index < array.length; index++) {
        if (filterFunction.test(array[index], index)) {
            // If the filter function returns 'true', include the element in the result list
            result.add(array[index]);
        }
    }
    return result; // Return the filtered list
}

```

C++ Solution

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

/**
 * Filters elements of a vector based on a provided filter function.
 * @param inputArray - The vector to filter.
 * @param filterFunction - The filter function used to determine if an element should be included in the result vector.
 *     It takes the element value and index as its arguments.
 * @returns A new vector with elements that pass the filter function condition.
 */
std::vector<int> filter(const std::vector<int> &inputArray, const std::function<bool(int, int)> &filterFunction) {
    std::vector<int> result; // Initialize an empty vector to hold the filtered results
    for (int index = 0; index < inputArray.size(); ++index) {
        if (filterFunction(inputArray[index], index)) {
            // If the filter function returns 'true', include the element in the result vector
            result.push_back(inputArray[index]);
        }
    }
    return result; // Return the filtered vector
}

```

Typescript Solution

```
/**
 * Filters elements of an array based on a provided filter function.
 * @param {number[]} array - The array to filter.
 * @param {Function} filterFunction - The filter function used to determine if an element should be included in the result array.
 *    It takes the element value and index as its arguments.
 * @returns {number[]} A new array with elements that pass the filter function condition.
 */
function filter(array: number[], filterFunction: (value: number, index: number) => boolean): number[] {
    const result: number[] = []; // Initialize an empty array to hold the filtered results
    for (let index = 0; index < array.length; ++index) {
        if (filterFunction(array[index], index)) {
            // If the filter function returns 'true', include the element in the result array
            result.push(array[index]);
        }
    }
    return result; // Return the filtered array
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the array `arr`. This is because the function iterates through each element of the array exactly once.

The space complexity of the function is `O(1)`, ignoring the space consumption of the `ans` array, since there are no additional data structures that grow with the size of the input. The variables `ans` and `i` use a fixed amount of space regardless of the input size.