2724. Sort By

Easy

Problem Description

In this problem, you are provided with two inputs: an array `arr` which can contain any type of elements, and a function `fn` which takes one of the elements of `arr` as an argument and returns a number. The task is to sort the array `arr` not by its actual values but by the numbers returned when each element is passed through the function `fn`. The output should be a new array `sortedArr` where the elements are ordered in ascending order based on the corresponding numbers returned from `fn`. It is guaranteed that `fn` will give a unique number for each element in the array, ensuring that there is a clear sort order.

Intuition

The given TypeScript function `sortBy` takes in `arr` and `fn` and uses the `sort` function to rearrange the elements in `arr`. In TypeScript and JavaScript, the `sort` function allows for a custom comparator, a function that takes in two elements and decides their order.

In this case, the comparator is `((a, b) => fn(a) - fn(b))`. This is a function that calls `fn` on both elements `a` and `b`, subtracts the result of calling `fn` on `b` from the result of calling `fn` on `a`, and uses the result of the subtraction to determine their order:

- If `fn(a) - fn(b)` is less than 0, `a` comes before `b` in the sorted array.
- If `fn(a) - fn(b)` is greater than 0, `b` comes before `a` in the sorted array.
- If `fn(a) - fn(b)` is 0 (which won't happen as `fn` returns unique numbers), `a` and `b` would be considered equal in terms of sorting, but this scenario is excluded per the problem assumptions.

Solution Approach

The implementation of the solution involves using the `sort` method which is a built-in functionality of JavaScript and TypeScript arrays. This method sorts the elements of an array in place and can order them according to the return value of a provided function.

In our `sortBy` function, we use `arr.sort((a, b) => fn(a) - fn(b))`. Here's how it breaks down:

- `arr`: This is the array of elements we need to sort.
- `.sort()`: This method accepts a comparator function that determines the sort order.
- `(a, b)`: The comparator function receives two elements from the array at a time.
- `fn(a) - fn(b)`: Inside the comparator, we apply the `fn` function to each element `a` and `b`, then subtract the result of `b` from `a`. The resulting number is used by `sort` to determine their order.

Remember, the `sort` method by default converts elements into strings and compares their sequences of UTF-16 code units values. However, when provided with a comparator function, it behaves as per the logic you provide in that function.

For the algorithms, data structures, or patterns used:

- **Algorithm**: The specific sort algorithm used by `.sort()` is dependent on the JavaScript engine implementation. It could be quicksort, mergesort, or another algorithm optimized for different types of arrays and sizes. However, you don't need to know these specifics to use the method.
- **Data Structures**: Since we're sorting an array and not using any additional data structures, the only relevant data structure is the input array itself.
- **Patterns**: A common programming pattern used here is the usage of higher-order functions. `fn` is a higher-order function since it takes a function as an argument (our comparator function) and returns a value based on the invocation of that function.

By using this approach, we get a sorted array based on the specified conditions using concise and effective code.

Example Walkthrough

Let's consider `arr` is an array of objects where each object has a `name` and an `age` property. We want to sort this array by the age of each person in ascending order.

```
const people = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 20 },
  { name: "Charlie", age: 30 }
];

function getAge(person) {
  return person.age;
}

const sortedPeople = sortBy(people, getAge);
```

In the example above, our `arr` is the `people` array and our `fn` is the `getAge` function which extracts the age from an object.

Here's what happens step-by-step when our `sortBy` function processes the `people` array using the `getAge` function as `fn`:

1. The `sortBy` function calls `arr.sort((a,b) => fn(a) - fn(b))`, passing in our custom comparator.
2. The `sort` method begins to compare elements in the array using the comparator function:
   - It compares two elements of `people`, say `Alice` and `Bob`, by calling `getAge(Alice)` which returns 25 and `getAge(Bob)` which returns 20.
   - The computation `fn(a) - fn(b)` translates to `25 - 20`, which is `5`. Since the result is positive, `Bob` will come before `Alice` in the sorted array.
3. This process repeats for each pair of elements in the array, effectively organizing the entire `people` array according to the ages of the people in ascending order.
4. The `sortBy` function returns a new array `sortedPeople`:
   - `[ { name: "Bob", age: 20 }, { name: "Alice", age: 25 }, { name: "Charlie", age: 30 } ]`

After the execution of `sortBy`, the `sortedPeople` array is sorted by the `age` property. This walk-through illustrates the elegant and efficient use of the `sort` method with a custom comparator function to sort objects by their specified properties.

Solution Implementation

Python

```
from typing import List, Callable

def sort_by(array: List[T], comparator: Callable[[T], int]) -> List[T]:
    """
    Sorts an array based on a provided comparator function.

    :param array: The array to be sorted.
    :param comparator: A function that takes an item and returns a number,
                       representing that item's position in the sort order.
    :return: The sorted array.
    """

    # Use the list's sort method by providing a lambda that calls
    # the comparator function to determine the sort order.
    # In Python, the sort method sorts the list in place.

    array.sort(key=comparator)
    return array
```

Java

```
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/**
 * Sorts a list based on a provided comparator function.
 * @param <T> The type of elements in the list.
 * @param list The list to be sorted.
 * @param comparator A Comparator that compares two elements.
 * @return The sorted list.
 */
public static <T> List<T> sortBy(List<T> list, Comparator<T> comparator) {
    // Use the Collections.sort method, passing in the list and the comparator.
    Collections.sort(list, comparator);
    return list;
}
```

C++

```
#include <vector>
#include <algorithm>

/**
 * Sorts a vector based on a provided comparator function.
 * @param array The vector to be sorted.
 * @param comparator A function or functor that takes an item and returns a number,
 * representing that item's position in the sort order.
 * @returns The sorted vector.
 */
template<typename T>
std::vector<T> sortBy(std::vector<T> array, const std::function<int(T)> &comparator) {
    // Use the standard library's sort function, passing in a lambda that uses the
    // comparator to determine the sort order.
    std::sort(array.begin(), array.end(), [&comparator](const T &a, const T &b) {
        return comparator(a) < comparator(b);
    });
    return array;
}
```

TypeScript

```
/**
 * Sorts an array based on a provided comparator function.
 * @param {T[]} array The array to be sorted.
 * @param {(item: T) => number} comparator A function that takes an item and returns a number,
 * representing that item's position in the sort order.
 * @returns {T[]} The sorted array.
 */
function sortBy<T>(array: T[], comparator: (item: T) => number): T[] {
    // Use the array sort function, passing in a lambda that uses the
    // comparator to determine the sort order.
    return array.sort((a, b) => comparator(a) - comparator(b));
}
```

```
from typing import List, Callable

def sort_by(array: List[T], comparator: Callable[[T], int]) -> List[T]:
    """
    Sorts an array based on a provided comparator function.

    :param array: The array to be sorted.
    :param comparator: A function that takes an item and returns a number,
                       representing that item's position in the sort order.
    :return: The sorted array.
    """

    # Use the list's sort method by providing a lambda that calls
    # the comparator function to determine the sort order.
    # In Python, the sort method sorts the list in place.

    array.sort(key=comparator)
    return array
```

Time and Space Complexity

The time complexity of the `sortBy` function largely depends on the implementation of the `.sort()` method in JavaScript's V8 engine (used in Chrome and Node.js). This method generally uses the TimSort algorithm for arrays that have more than a certain number of elements, which has a time complexity of `O(n log n)` on average and in the worst case. For smaller arrays, it may use an algorithm similar to insertion sort, which has a worst case time complexity of `O(n^2)`.

The space complexity for TimSort is `O(n)`. This is due to the need for allocating temporary arrays for storing merged sequences during the sorting process.

A key consideration here is the complexity of the `fn` function that is being used to compare elements. If the complexity of this function is `O(f(n))`, it should be multiplied by the sorting complexity. The overall time complexity would then become `O(n log n * f(n))` for large arrays.

In summary:

- TimSort time complexity: `O(n log n)`
- TimSort space complexity: `O(n)`
- Overall time complexity (including `fn`): `O(n log n * f(n))`