2774. Array Upper Bound

Easy

Leetcode Link

Problem Description

The task is to extend all arrays in a programming environment to include a new method called `upperBound()`. This method should be applicable to any array, and when called, it returns the index of the last occurrence of a specified `target` number within a given array. The array referenced by `nums` is sorted in ascending order and it can consist of repeated numbers, indicative of duplicates being allowed.

1. The `upperBound(target: number): number` method must be defined for all arrays.
2. The array provided (`nums`) is sorted in ascending order.
3. The `target` is the number whose upper bound index we need to find.
4. The "upper bound index" is defined as the index of the last occurrence of the `target` in the array.
5. If the `target` number is not present in the array, the method is expected to return `-1`.

In simple terms, if you've got an array like `[1,2,2,3]` and you call `upperBound(2)` on it, you should get `2`, which is the index of the last occurrence of the number `2`. If you search for a number not present in the array, such as `upperBound(4)`, it should return `-1`.

Intuition

The solution for the `upperBound` function utilizes a modified binary search algorithm. Since the array is sorted, binary search is an ideal choice for the task due to its efficiency in logarithmic time complexity (O(log n)).

Here is a breakdown of the intuition behind the solution approach:

1. **Initialization**: Set two pointers, `left` at the start of the array and `right` at the end (technically, just past the last element).
2. **Binary Search Loop**: Continue to narrow the search range by adjusting `left` and `right`.
    - Calculate the middle index `mid` by averaging `left` and `right` and shifting right (`>>`) is equivalent to dividing by 2).
    - If the element at `mid` is greater than the `target`, move the `right` pointer to `mid`, as the `target`, if it exists, must be to the left of `mid`.
    - Otherwise, move the `left` pointer to `mid + 1`, since we're looking for the last occurrence of `target`.
3. **Result**: Once the loop ends, if `left` is greater than 0 and the element at `left - 1` equals the `target`, then the upper bound is found, and the index `left - 1` is returned.
    - If the `target` is not found, return `-1`.

In this way, the solution exploits the sorted nature of the array and efficiently finds the upper bound of a given `target` number, if present.

Solution Approach

The implementation of the `upperBound` method on the `Array` prototype in TypeScript follows a binary search approach to locate the last occurrence of the `target` number within an array. Binary search is a well-known algorithm for finding an item in a sorted list, and it works by repeatedly dividing the search interval in half.

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

1. **Extending the Prototype**: First, the `Array` prototype is extended by defining the `upperBound` function, allowing all arrays to use this new method.
2. **Initialization**:
    - `left` is initialized to `0`, representing the start of the array.
    - `right` is initialized to the length of the array, which is an index one past the last element, as this is a common pattern in binary search implementations to facilitate the calculation with `mid`.
3. **Binary Search Loop**:
    - The search continues until `left < right`, which means there is still a range to be checked.
    - In each iteration, the `mid` point of the current range is calculated using the bit shift operator `>>`, which is a quick way to perform integer division by 2 (i.e., `(left + right) >> 1`).
4. **Midpoint Evaluation**:
    - If the value at the `mid` index is greater than the `target`, the `target` cannot be to the right of `mid`, so `right` is updated to `mid`. This effectively discards the second half of the current range.
    - If the value at `mid` is less than or equal to the `target`, the `target` can be at `mid` or to the right of it, so `left` is updated to `mid + 1`. Since we're looking for the last occurrence, we can safely ignore `mid` (even if it matches the `target`) because there might be another occurrence of `target` further to the right.
5. **Post-Loop Check**:
    - After the loop finishes, `left` is the index where the `target` either just surpasses the last `target` number or where the `target` would be inserted to maintain the sorted order of the array.
    - There's a check to see if `left` is greater than `0` and if the element at `left - 1` equals `target`, returning `left - 1` as it represents the last occurrence of `target`.
6. **Default Case**:
    - If the number is not found, the method defaults to returning `-1`, indicating the absence of the `target` number.

This approach ensures that the `upperBound` function can operate efficiently, typically requiring `O(log n)` time to find the upper bound of the `target` in the array, where `n` is the number of elements in the array.

Example usage of the `upperBound` method could look like this:

```
console.log([3,4,5].upperBound(5)); // Output: 2
console.log([1,2,2,3].upperBound(2)); // Output: 2
console.log([3,4,6,6,6,7].upperBound(6)); // Output: 5
```

This implementation assumes that the environment supports modifications to the `Array` prototype and that the arrays in question are sorted in ascending order.

Example Walkthrough

Let's go through the solution approach with a small example. Suppose we have an array `nums` and we want to find the upper bound index of the target number `2` in this array:

```
nums = [1, 2, 2, 3, 4].
```

1. First, since we want to extend all arrays with the `upperBound` method, we define this method in the `Array` prototype.
2. To start the binary search, we initialize two variables, `left` to `0` and `right` to `nums.length` (which is 5 in this case, as the array has elements from index 0 to 4).
3. Enter the binary search loop. Our target is `2`, so we check the middle of the array between `left` and `right`. Since our current `left` is 0 and `right` is 5, the middle index `mid` is calculated as `(0 + 5) >> 1`, which is 2.
4. Now, we check the value at index 2, which is also `2`. Since we are looking for the last occurrence and the middle value is equal to our target, we move the `left` pointer up, to `mid + 1` to continue the search in the right half of the current range. After this, `left` is now 3, and `right` remains 5.
5. We iterate again, and now `mid` is `(3 + 5) >> 1`, which is 4. The element at index 4 is 4, which is greater than our target `2`. Hence, we move the `right` pointer down to `mid`, making `right` now equal to 4.
6. The next iteration starts, but since `left` is still less than `right`, we calculate a new `mid` `(3 + 4) >> 1`, which is 3. The element at index 3 is `3`, which is greater than our target `2`, so again we move `right` to `mid`. The value of `right` is now 3.
7. At this point, `left` and `right` are equal, indicating that the search space is empty so the loop ends.
8. After the loop, we perform a final check. Since `left` is 3 and we are asked to return `left - 1`, we check if the element at index 2 is equal to `2`, and it is. Therefore, the upper bound index is indeed `2`.
9. If the target had been a number not present in the array, like `5`, we would have eventually narrowed down to a point where `left` would equal `right` and the check of `nums[left - 1]` would not match the target, thus, the function would return `-1`.

This example illustrates how the `upperBound` method works on a simple array using binary search principles. It efficiently narrows down the search and accurately finds the last occurrence of the target number `2`, or confirms the target's absence if it's not in the array.

Python Solution

```
class ExtendedList(list):
    # Implementation of the `upper_bound` method.
    # It finds the index of the first element in the array greater than the given target.
    # If the target is not found or no element is less than or equal to the target,
    # it returns -1.
    def upper_bound(self, target: int) -> int:
        left = 0  # start of the range to search
        right = len(self)  # end of the range to search

        while left < right:
            mid = (left + right) // 2  # find the middle index

            # If the middle element is greater than the target,
            # the upper bound must be to the left of mid (inclusive)
            if self[mid] > target:
                right = mid
            else:
                # Otherwise, the upper bound is to the right of mid
                left = mid + 1

        # Return the index of the first element greater than the target
        # or return -1 if the target is not found
        return left if left > 0 and self[left - 1] == target else -1

# Example usage:
results = ExtendedList([3, 4, 5]).upper_bound(5)  # result1 should be 2
results = ExtendedList([1, 2, 2, 3]).upper_bound(2)  # result2 should be 2
results = ExtendedList([3, 4, 6, 6, 6, 7]).upper_bound(6)  # result3 should be 6

# Print results to verify
print("Result 1:", result1)
print("Result 2:", result2)
print("Result 3:", result3)
```

Java Solution

```
import java.util.ArrayList;

// To add a method to the ArrayList class, we're going to create a MyArrayList class that extends ArrayList
public class MyArrayList<T extends Comparable<T>> extends ArrayList<T> {

    // Method `upperBound` finds the index of the first element that is greater than the given target.
    // If all elements are less than or equal to the target, it returns the size of the list.
    public int upperBound(T target) {
        int left = 0; // Start of the range to search
        int right = this.size(); // End of the range to search, exclusive

        while (left < right) {
            int mid = left + (right - left) / 2; // Find the middle index using a safe method to prevent overflow

            // If the middle element is greater than target, move the right pointer to mid
            // This is because we are looking for the first element greater than the target
            if (this.get(mid).compareTo(target) > 0) {
                right = mid;
            } else {
                // Otherwise, move the left pointer past mid, as all elements up to mid are less or equal to the target
                left = mid + 1;
            }
        }

        // The loop exits when left == right, which is the position where an element greater than
        // target would get inserted (hence the upper bound)
        return left;
    }
}

// The MyArrayList class can be used as follows:
public class TestUpperBound {
    public static void main(String[] args) {
        MyArrayList<Integer> list = new MyArrayList<>();
        list.add(3);
        list.add(4);
        list.add(5);

        // The `upperBound` method is now available for use
        int result1 = list.upperBound(5); // result1 should be 3

        list.clear();
        list.add(1);
        list.add(2);
        list.add(2);
        list.add(3);
        int result2 = list.upperBound(2); // result2 should be 1

        list.clear();
        list.add(3);
        list.add(4);
        list.add(6);
        list.add(6);
        list.add(6);
        list.add(7);
        int result3 = list.upperBound(6); // result3 should be 6

        // Print the results
        System.out.println("Result1: " + result1);
        System.out.println("Result2: " + result2);
        System.out.println("Result3: " + result3);
    }
}
```

C++ Solution

```
#include <vector>
using namespace std;

// Adding a member function called `upper_bound` to the std::vector class template for number vectors.
template <typename T>
class VectorWithUpperBound : public vector<T> {
public:
    // Constructor forwarding to std::vector's constructor.
    using vector<T>::vector;

    // Function upper_bound
    // It finds the index of the first element in the array greater than the given target.
    // If no such element is found, it returns the size of the vector.
    size_t upper_bound(T target) const {
        size_t left = 0; // Start of the range to search
        size_t right = this->size(); // End of the range to search, exclusive

        while (left < right) {
            size_t mid = left + (right - left) / 2; // Find the middle index without integer overflow

            // If the middle element is greater than target, move the right boundary in
            if (this->at(mid) > target) {
                right = mid;
            } else {
                // Otherwise, move the left boundary out
                left = mid + 1;
            }
        }

        // Return the index of the upper bound
        return left;
    }
};

int main() {
    // Using the upper_bound method after extending the vector class
    VectorWithUpperBound<int> vec1 = {3, 4, 5};
    size_t result1 = vec1.upper_bound(5); // result1 should be 3

    VectorWithUpperBound<int> vec2 = {1, 2, 2, 3};
    size_t result2 = vec2.upper_bound(2); // result2 should be 1

    VectorWithUpperBound<int> vec3 = {3, 4, 6, 6, 6, 7};
    size_t result3 = vec3.upper_bound(6); // result3 should be 6

    // Your result handling
}
```

Typescript Solution

```
// Extending the global Array interface to include the upperBound method for number arrays
declare global {
    interface Array<T> {
        upperBound(target: number): number;
    }
}

// Implementation of the `upperBound` method for the Array prototype.
// It finds the index of the first element in the array greater than the given target.
// If the target is not found, it returns -1.
Array.prototype.upperBound = function (target: number): number {
    let left = 0; // Start of the range to search
    let right = this.length; // End of the range to search, exclusive

    while (left < right) {
        const mid = Math.floor((left + right) / 2); // Find the middle index

        // If the middle element is greater than target,
        // the upper bound must be to the left of mid (inclusive)
        if (this[mid] > target) {
            right = mid;
        } else {
            // Otherwise, the upper bound is to the right of mid
            left = mid + 1;
        }
    }

    // Return the index of the upper bound if it is at the left border,
    // or return -1 if the target is not found
    return left > 0 && this[left - 1] === target ? left - 1 : -1;
};

// Ensure that these changes do not break outside the module scope
export {};
```

To use the `upperBound` function, you should include this code in a module, and then you can import it where it's needed. Here's how you would use the `upperBound` method after including the code:

```
import "./path/to/extension"; // Replace with actual path to your extended Array prototype

const result1 = [3,4,5].upperBound(5); // result1 should be 2
const result2 = [1,2,2,3].upperBound(2); // result2 should be 2
const result3 = [3,4,6,6,6,7].upperBound(6); // result3 should be 5
```

Time and Space Complexity

The time complexity of the `Array.prototype.upperBound` function is $O(\log n)$, where n is the number of elements in the array. This is because the function uses a binary search approach, which repeatedly divides the array in half and thus has a logarithmic time complexity.

The space complexity of the function is $O(1)$. It uses only a constant amount of additional space regardless of the size of the input array since all operations are performed in place and it only uses a fixed number of variables (`left`, `right`, and `mid`).