

# 2774. Array Upper Bound

## 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.

- The `upperBound(target: number): number` method must be defined for all arrays.
- The array provided (`nums`) is sorted in ascending order.
- The `target` is the number whose upper bound index we need to find.
- The "upper bound index" is defined as the index of the last occurrence of the `target` in the array.
- 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:

- Initialization:** Set two pointers, `left` at the start of the array and `right` at the end (technically, just past the last element).
- 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 ( $>>1$  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`.
- 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:

- Extending the Prototype:** First, the `Array` prototype is extended by defining the `upperBound` function, allowing all arrays to use this new method.
- 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 calculations.
- 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.,  $(left + right) >> 1$ ).
- 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.
- 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`.
- 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:

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

- First, since we want to extend all arrays with the `upperBound` method, we define this method in the `Array` prototype.
- 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`).
- 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`.
- 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`.
- 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`.
- 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`.
- At this point, `left` and `right` are equal, indicating that the search space is empty so the loop ends.
- 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`.
- 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

```
1 class ExtendedList(list):
2     # Implementation of the 'upper_bound' method.
3     # It finds the index of the first element in the array greater than the given target.
4     # If the target is not found or every element is less than or equal to the target,
5     # it returns -1.
6
7     def upper_bound(self, target: int) -> int:
8         left = 0 # start of the range to search
9         right = len(self) # end of the range to search
10
11         while left < right:
12             mid = (left + right) // 2 # find the middle index
13
14             # If the middle element is greater than the target,
15             # the upper bound must be to the left of mid (inclusive)
16             if self[mid] > target:
17                 right = mid
18             else:
19                 # Otherwise, the upper bound is to the right of mid
20                 left = mid + 1
21
22         # Return the index of the first element greater than the target
23         # or return -1 if the target is not found
24         return left if left < len(self) else -1
25
26
27 # Example usage:
28 result1 = ExtendedList([3, 4, 5]).upper_bound(5) # result1 should be 2
29 result2 = ExtendedList([1, 4, 5]).upper_bound(2) # result2 should be -1
30 result3 = ExtendedList([3, 4, 6, 6, 6, 6, 7]).upper_bound(6) # result3 should be 6
31
32 # Print results to verify
33 print("Result 1:", result1)
34 print("Result 2:", result2)
35 print("Result 3:", result3)
36
```

## Java Solution

```
1 import java.util.ArrayList;
2
3 // To add a method to the ArrayList class, we're going to create a MyArrayList class that extends ArrayList
4 public class MyArrayList<T extends Comparable<T>> extends ArrayList<T> {
5
6     // Method 'upperBound' finds the index of the first element that is greater than the given target.
7     // If all elements are less than or equal to the target, it returns the size of the list.
8     public int upperBound(T target) {
9         int left = 0; // Start of the range to search
10        int right = this.size(); // End of the range to search, exclusive
11
12        while (left < right) {
13            int mid = left + (right - left) / 2; // Find the middle index using a safe method to prevent overflow
14
15            // If the middle element is greater than target, move the right pointer to mid
16            // If this narrowing of the range finds the first element greater than the target
17            if (this.get(mid).compareTo(target) > 0) {
18                right = mid;
19            } else {
20                // Otherwise, move the left pointer past mid, as all elements up to mid are less or equal to the target
21                left = mid + 1;
22            }
23        }
24
25        // The loop exits when left == right, which is the position where an element greater than
26        // target would get inserted (hence the upper bound)
27        return left;
28    }
29 }
30
31 // The MyArrayList class can be used as follows:
32 public class TestUpperBound {
33     public static void main(String[] args) {
34         MyArrayList<Integer> list = new MyArrayList<>();
35         list.add(3);
36         list.add(4);
37         list.add(5);
38
39         // The 'upperBound' method is now available for use
40         int result1 = list.upperBound(5); // result1 should be 3
41
42         list.clear();
43         list.add(1);
44         list.add(4);
45         list.add(5);
46         int result2 = list.upperBound(2); // result2 should be 1
47
48         list.clear();
49         list.add(3);
50         list.add(4);
51         list.add(6);
52         list.add(6);
53         list.add(6);
54         list.add(6);
55         list.add(7);
56         int result3 = list.upperBound(6); // result3 should be 6
57
58         // Print the results
59         System.out.println("Result1: " + result1);
60         System.out.println("Result2: " + result2);
61         System.out.println("Result3: " + result3);
62     }
63 }
64
```

## C++ Solution

```
1 #include <vector>
2 using namespace std;
3
4 // Adding a member function called 'upper_bound' to the std::vector class template for number vectors.
5 template <typename T>
6 class VectorWithUpperBound : public vector<T> {
7 public:
8     // Constructor forwarding to std::vector's constructor.
9     using vector<T>::vector;
10
11     // Function upper_bound
12     // Finds the index of the first element in the array greater than the given target.
13     // If no such element is found, it returns the size of the vector.
14     size_t upper_bound(T target) const {
15         size_t left = 0; // Start of the range to search
16         size_t right = this->size(); // End of the range to search, exclusive
17
18         while (left < right) {
19             size_t mid = left + (right - left) / 2; // Find the middle index without integer overflow
20
21             // If the middle element is greater than target, move the right boundary in
22             if ((*this)[mid] > target) {
23                 right = mid;
24             } else {
25                 // Otherwise, move the left boundary out
26                 left = mid + 1;
27             }
28         }
29
30         // Return the index of the upper bound
31         return left;
32     }
33 };
34
35 int main() {
36     // Using the upper_bound method after extending the vector class
37     VectorWithUpperBound<int> vec1 = {3, 4, 5};
38     size_t result1 = vec1.upper_bound(5); // result1 should be 3
39
40     VectorWithUpperBound<int> vec2 = {1, 4, 5};
41     size_t result2 = vec2.upper_bound(2); // result2 should be 1
42
43     VectorWithUpperBound<int> vec3 = {3, 4, 6, 6, 6, 6, 7};
44     size_t result3 = vec3.upper_bound(6); // result3 should be 6
45
46     // Your result handling
47 }
48
49 // Note: Ensure that the above code is included in a proper header and source file structure
50 // and that you include the appropriate header files where the extended vector is used.
51
```

## Typescript Solution

```
1 // Extending the global Array interface to include the upperBound method for number arrays
2 declare global {
3     interface Array<T> {
4         upperBound(target: number): number;
5     }
6 }
7
8 // Implementation of the 'upperBound' method for the Array prototype.
9 // It finds the index of the first element in the array greater than the given target.
10 // If the target is not found, it returns -1.
11 Array.prototype.upperBound = function (target: number): number {
12     let left = 0; // Start of the range to search
13     let right = this.length; // End of the range to search, exclusive
14
15     while (left < right) {
16         const mid = Math.floor((left + right) / 2); // Find the middle index
17
18         // If the middle element is greater than target,
19         // the upper bound must be to the left of mid (inclusive)
20         if (this[mid] > target) {
21             right = mid;
22         } else {
23             // Otherwise, the upper bound is to the right of mid
24             left = mid + 1;
25         }
26     }
27
28     // Return the index of the upper bound if it is at the left border,
29     // or return -1 if the target is not found
30     return left > 0 && this[left - 1] == target ? left - 1 : -1;
31 };
32
33 // Ensure that these changes do not break outside the module scope
34 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:

```
1 import './path/to/extension'; // Replace with actual path to your extended Array prototype
2
3 const result1 = [3,4,5].upperBound(5); // result1 should be 2
4 const result2 = [1,4,5].upperBound(2); // result2 should be -1
5 const result3 = [3,4,6,6,6,6,7].upperBound(6); // result3 should be 5
6
```

## 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`).