# 1966. Binary Searchable Numbers in an Unsorted Array

Medium Array Binary Search Leetcode Link

## Problem Description

In this problem, we are given an array `nums` that contains a sequence of unique integers. The sequence may be sorted or not sorted. We are tasked with implementing an algorithm that simulates a binary search-like process, but instead of choosing the middle element as a pivot, it chooses a random pivot during each iteration. If the pivot matches the `target`, the function returns `true`. If the pivot is less than the target, all elements to the left of the pivot (including the pivot itself) are removed from the sequence, and the search continues with the remaining elements. Similarly, if the pivot is greater than the target, all elements to the right of the pivot are removed from the sequence. This continues until either the target is found (and `true` is returned) or the sequence is empty (and `false` is returned).

The crux of the problem is to find out how many values in the `nums` array are *guaranteed* to be found using this function, regardless of which pivots are randomly selected in the process. A "guaranteed" value here means that no matter how the elements are chosen as pivots in each iteration, it will always end up being found if it is indeed in the sequence.

## Intuition

To solve this problem, we need to identify which elements in the sequence would always be found, regardless of the random selection of the pivot. To do this, we need to find out whether choosing any pivot would lead to accidentally discarding the target element when it shouldn't have been discarded. To be a 'binary search-able' number, a target value in the sequence must never be in the same subsequence as any incorrect pivot (an incorrect pivot being any element that would not be the target and would mistakenly remove the target from consideration).

For example, let's say our target is the number 3 in an unsorted sequence. If at any point during the function execution, the number 3 is in a subsequence with a pivot greater than 3, it might get discarded even though it hasn't been evaluated yet.

Hence, the process involves two checks:

1. From left to right, we check if any element is smaller than any element to its left. If it is, then in some random pivot selection, it could be incorrectly disregarded as a pivot.
2. From right to left, we check if any element is greater than any element to its right. If it is, then it could also lead to the same problem but in the opposite direction.

We maintain an array, `ok`, to keep track of whether each element in `nums` is binary search-able. We initially set all elements in this array to `1` (representing 'true'), indicating that we assume all elements are binary search-able.

During the execution, we utilize two variables `mx` and `mi` to keep track of the maximum value encountered so far from the left and the minimum value from the right, respectively. Starting from the left of the array, for each element `x`, if it is smaller than `mx`, then it cannot be binary search-able as there exists a larger value to the left. Thus, we set `ok[i]` to `0` for such an element. We then update `mx` with the maximum value seen so far.

We perform a similar procedure starting from the right of the array, using `mi` to track the minimum value seen so far. If any element `nums[i]` is greater than `mi`, we set `ok[i]` to `0`, indicating that the element can't be guaranteed to be searchable, and then we update `mi`.

At the end of the process, `sum(ok)` will yield the total number of elements in the `nums` array that are guaranteed to be found, which is the solution to the problem.

## Solution Approach

The solution provided leverages the concept of maintaining two boundary conditions as we iterate over the input array, which is a common technique in problems involving subarrays or sequence behavior modifications based on certain conditions.

Here is a closer look at the step-by-step approach:

1. Initialize an array `ok` of the same length as `nums`, with all elements set to `1`. This array will track whether each element in `nums` could be binary search-able (`1` for yes, `0` for no).

2. Set `mx` (maximum boundary) to a very small number (the problem uses `-1000000` as an initially small value) and `mi` (minimum boundary) to a very large number (the problem uses `1000000`), ensuring these boundaries will not affect the first comparison.

3. Iterate over `nums` from left to right, updating `mx` to be the maximum value seen so far. If the current element `x` is less than `mx`, it means `x` had a larger number before it, and thus, `x` can't be guaranteed to always be searchable. Set `ok[i]` to `0`.

4. Iterate over `nums` from right to left, updating `mi` to be the minimum value seen so far. If the current element `nums[i]` is greater than `mi`, it implies that there's a smaller element to its right, so if `nums[i]` were chosen as a pivot, it would mistakenly discard the smaller number. Set `ok[i]` to `0` for such cases.

5. Finally, sum up all elements in `ok`. This sum represents the count of elements that are guaranteed to be binary search-able.

Let's relate the steps with the actual code segments:

- The `ok` array initialization is directly represented by `ok = [1] * n`.
- Setting the `mx` and `mi` boundaries uses `mx, mi = -1000000, 1000000`.
- The two `for` loops represent the left-to-right and right-to-left iterations with corresponding logic to update `ok`.
- The sum of elements in `ok` is obtained by `return sum(ok)`.

By combining the edge constraints from both ends of the array, we ensure that only those elements that satisfy the conditions for both leftward and rightward iterations are considered binary search-able. The nature of the algorithm is such that it has O(n) time complexity since it passes over the array twice—once from each end—and performs constant-time operations within each iteration.

In terms of algorithmic patterns, this approach can be recognized as a variant of the two-pointer technique where instead of moving two pointers towards each other, we move boundaries in a way that encodes the validity of each element based on the search-ability condition described in the problem.

### Example Walkthrough

Let's use a small example to illustrate the solution approach. We will walk through both the left-to-right and right-to-left iterations steps to clarify how the `ok` array gets updated.

Consider the array `nums = [4, 2, 5, 7, 6, 3, 1]` and the target value as `3`. We will go through the algorithm to demonstrate how it ensures whether `3` or any other value can be found regardless of the random pivot selections.

1. Initialize an array `ok` to `[1, 1, 1, 1, 1, 1, 1]`, i.e., initially assuming all elements are search-able.
2. Set `mx` to `-1000000` and `mi` to `1000000`.

**Left-to-Right Check:**

- Start with the first element (4). Since `mx` is `-1000000`, `4` is greater than `mx`, so we set `mx` to `4`.
- Next element is `2`. It is less than the current `mx` (which is `4`), so we update `ok` at index 1 to `0 [1, 0, 1, 1, 1, 1, 1]`.
- Move to `5`, which is greater than `mx` (4), so we update `mx` to `5`.
- Continue to `7`, which is greater than `mx` (5), so we update `mx` to `7`.
- Process `6`, it is less than `mx` (7), so we update `ok` at index 4 to `0 [1, 0, 1, 1, 0, 1, 1]`.
- Move to `3`, it is less than `mx` (7), so update `ok` at index 5 to `0 [1, 0, 1, 1, 0, 0, 1]`.
- Finally, `1` is less than `mx` (7), so update `ok` at index 6 to `0 [1, 0, 1, 1, 0, 0, 0]`.

**Right-to-Left Check:**

- Start with the last element of the current `ok` state `[1, 0, 1, 1, 0, 0, 0]` which is `1`. Since `mi` is `1000000`, `1` is less than `mi`, so we set `mi` to `1`.
- Next element is `3`. It's greater than the current `mi` (1), so we update `ok` at index 5 to `0 [1, 0, 1, 1, 0, 0, 0]`.
- Since `ok[5]` was already `0`, this step does not change the state of `ok`.
- Continue to `6`, which is greater than the current `mi` (1), so `ok` at index 4 could potentially be set to `0`, but it's already `0`.
- Process `7`, it is greater than `mi` (1), but `ok` at index 3 is still `1`, and it remains `1` as `7` has no smaller element to its right.
- Move to `5`, which is greater than `mi` (1), `ok` at index 2 remains `1`.
- Next is `2`, which is greater than the current `mi` (1), and since `ok` at index 1 is `0`, it remains unchanged.
- Finally, `4` is greater than `mi` (1), so `ok` remains as `[1, 0, 1, 1, 0, 0, 0]`.

**Final Step:**

- Sum up the values in `ok` to find the total count of elements guaranteed to be searchable. In this case, `sum(ok)` equals to `3`.

Therefore, `3` elements in the `nums` array can be guaranteed to be found using our random-pivot-based binary search algorithm. These elements are `4`, `5`, and `7`. These elements are guaranteed searchable because there are no other values to their left that would cause them to be discarded if they were not chosen first, and similarly, no values to their right that would be incorrectly discarded if these elements were chosen as pivots.

## Python Solution

```python
from typing import List

class Solution:
    def binarySearchableNumbers(self, nums: List[int]) -> int:
        num_elements = len(nums)  # Number of elements in the input list

        # This list holds flags for elements that are binary searchable.
        # Initially set all elements as potentially binary searchable (1).
        is_binary_searchable = [1] * num_elements

        # Initialize the variables to keep track of the maximum value from the left
        # and the minimum value from the right encountered so far.
        max_from_left = float('-inf')
        min_from_right = float('inf')

        # Iterate from left to right to check if there's a larger number before any element.
        for i, value in enumerate(nums):
            if value < max_from_left:
                # If the current value is less than the max found so far,
                # it cannot be found using binary search.
                is_binary_searchable[i] = 0
            else:
                # Update the max value encountered from the left.
                max_from_left = value

        # Iterate from right to left to check if there's a smaller number after any element.
        for i in range(num_elements - 1, -1, -1):
            if nums[i] > min_from_right:
                # If the current value is greater than the min found so far,
                # it cannot be found using binary search.
                is_binary_searchable[i] = 0
            else:
                # Update the min value encountered from the right.
                min_from_right = nums[i]

        # Return the count of elements that are binary searchable.
        # This is the sum of the flags indicating binary searchability.
        return sum(is_binary_searchable)

# Example usage:
# solution = Solution()
# result = solution.binarySearchableNumbers([1, 3, 2, 4, 5])
# print(result)  # Output would be the count of binary searchable numbers in the input list

```

## Java Solution

```java
class Solution {
    public int binarySearchableNumbers(int[] nums) {
        int length = nums.length; // Get the length of the array
        int[] searchable = new int[length]; // Define an array to keep track of searchable numbers
        Arrays.fill(searchable, 1); // Assume all numbers are initially searchable
        int maxLeft = Integer.MIN_VALUE; // Initialize the maximum to the smallest possible integer
        int minRight = Integer.MAX_VALUE; // Initialize the minimum to the largest possible integer

        // Forward pass: Check for each element if there is a larger number on its left
        for (int i = 0; i < length; ++i) {
            if (nums[i] < maxLeft) {
                searchable[i] = 0; // If found, mark the element as unsearchable
            }
            maxLeft = Math.max(maxLeft, nums[i]); // Update the max value from the left
        }

        int count = 0; // Initialize the count of binary searchable numbers

        // Backward pass: Check for each element if there is a smaller number on its right
        for (int i = length - 1; i >= 0; --i) {
            if (nums[i] > minRight) {
                searchable[i] = 0; // If found, mark the element as unsearchable
            }
            minRight = Math.min(minRight, nums[i]); // Update the min value from the right

            count += searchable[i]; // Increment count if the current number is searchable
        }

        return count; // Return the total count of binary searchable numbers
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    // Function to count how many numbers in the array are binary searchable
    int binarySearchableNumbers(vector<int>& nums) {
        int n = nums.size(); // Get the size of the array
        vector<int> isBinarySearchable(n, 1); // Initialize a vector to keep track of binary searchable numbers
        int maxToLeft = INT_MIN; // Initialize the maximum to the left with the smallest int
        int minToRight = INT_MAX; // Initialize the minimum to the right with the largest int

        // Determine which elements are not searchable by traversing the array from left to right
        for (int i = 0; i < n; ++i) {
            if (nums[i] < maxToLeft) {
                isBinarySearchable[i] = 0; // If the current element is smaller than the max to the left, it's not searchable
            }
            maxToLeft = max(maxToLeft, nums[i]); // Update maxToLeft
        }

        int count = 0; // Initialize a count for the binary searchable numbers
        // Determine which elements are not searchable by traversing the array from right to left
        for (int i = n - 1; i >= 0; --i) {
            if (nums[i] > minToRight) {
                isBinarySearchable[i] = 0; // If the current element is larger than the min to the right, it's not searchable
            }
            minToRight = min(minToRight, nums[i]); // Update minToRight
            count += isBinarySearchable[i]; // Add to count if the number is binary searchable
        }

        return count; // Return the total count of binary searchable numbers
    }
};

```

## Typescript Solution

```typescript
// Define the global variables
let maxToLeft: number;
let minToRight: number;
let isBinarySearchable: number[];

// Function to determine if a number is binary searchable
function binarySearchableNumbers(nums: number[]): number {
    let n: number = nums.length; // Get the size of the array
    isBinarySearchable = new Array(n).fill(1); // Initialize an array to keep track of binary searchable numbers
    maxToLeft = Number.MIN_SAFE_INTEGER; // Initialize the maximum to the left with the smallest safe integer
    minToRight = Number.MAX_SAFE_INTEGER; // Initialize the minimum to the right with the largest safe integer

    // Determine which elements are not searchable by traversing the array from left to right
    for (let i = 0; i < n; ++i) {
        if (nums[i] < maxToLeft) {
            isBinarySearchable[i] = 0; // If the current element is smaller than maxToLeft, it's not searchable
        }
        maxToLeft = Math.max(maxToLeft, nums[i]); // Update maxToLeft with the maximum value so far
    }

    let count: number = 0; // Initialize a count for the binary searchable numbers
    // Determine which elements are not searchable by traversing the array from right to left
    for (let i = n - 1; i >= 0; --i) {
        if (nums[i] > minToRight) {
            isBinarySearchable[i] = 0; // If the current element is larger than minToRight, it's not searchable
        }
        minToRight = Math.min(minToRight, nums[i]); // Update minToRight with the minimum value so far
        count += isBinarySearchable[i]; // Increment count if the element is binary searchable
    }

    return count; // Return the total count of binary searchable numbers
}

```

## Time and Space Complexity

### Time Complexity

The given Python code performs a single forward pass through the array to identify elements that are not greater than any element before them, and then performs a single backward pass to identify elements that are not less than any element after them. Each pass involves a simple comparison operation for each element of the list.

- The forward pass processes each element once, marking those elements that are smaller than the maximum element observed so far with a `0`. This constitutes `O(n)` operations where `n` is the size of the `nums` list.

- Similarly, the backward pass also processes each element once, marking those elements that are greater than the minimum element observed so far with a `0`. This also constitutes `O(n)` operations.

- Finally, the summing operation to count the number of `1`s in the `ok` list also takes `O(n)`.

Hence the overall time complexity of the code is `O(n) + O(n) + O(n)` which simplifies to `O(n)`.

### Space Complexity

The space complexity is determined by the additional space used by the program excluding the input.

- The code defines a list `ok` which has the same size as the input list `nums`. Therefore `ok` takes `O(n)` space.

- Variables `mx` and `mi` use constant space, so their space complexity contribution is `O(1)`.

- The loop indices and temporary variables used for logic operations also occupy constant space.

Thus, the overall space complexity of the code is `O(n)` due to the `ok` array. The rest of the variables do not contribute significantly as they are constant space.