**Find First and Last Position of Element in Sorted Array**

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [5,7,7,8,8,10], target = 8

**Output:** [3,4]

**Example 2:**

**Input:** nums = [5,7,7,8,8,10], target = 6

**Output:** [-1,-1]

**Example 3:**

**Input:** nums = [], target = 0

**Output:** [-1,-1]

**Constraints:**

* 0 <= nums.length <= 105
* -109 <= nums[i] <= 109
* nums is a non-decreasing array.
* -109 <= target <= 109

/\*\*

\* @param {number[]} nums

\* @param {number} target

\* @return {number[]}

\*/

var searchRange = function(nums, target) {

};

Solution

Overview

Let's briefly look at a brute-force way of solving this problem. Given a target element, we can simply do a linear scan over the entire array to find the first and the last position. The first occurrence will be the first time when we encounter this target. Thereafter, we continue to scan elements until we find one that is greater than the target or until we reach the end of the array. This will help us determine the last position of the target.

The downside of this approach is that it doesn't take advantage of the *sorted* nature of the array. This linear scan approach has a time complexity of O(N)*O*(*N*) because there are N*N* elements in the array. That doesn't sound too bad, right? Well, it does if we compare it to an approach with logarithmic time complexity. We'll look at a *binary search-based approach* to solve this problem which will take advantage of the sorted nature of the array.

Approach: Binary Search

**Intuition**

Let's review binary search a bit. Given a sorted array, binary search works by looking at the middle element of the given array, and based on the value of the middle element, it decides to discard one half of the array. At each step, we reduce the length of the array to search by half and that is what leads to the logarithmic time complexity of the algorithm. Usually, we employ the binary search algorithm to determine if an element is in a sorted array. Here, we can tweak the binary search algorithm to find the first and the last position of a given element.

Let's look at the basic binary search algorithm one step at a time:

* We use 2 variables to keep track of the subarray that we are scanning. Let's call them begin and end. Initially, begin is set to 0 and end is set to the last index of the array.
* We iterate until begin is greater than or equal to end.
* At each step, we calculate the middle element mid = (begin + end) / 2. We use the value of the middle element to decide which half of the array we need to search.
  + If the target that we're searching for has a value lower than the mid element, we discard the right half of the array i.e. end = mid - 1.
  + If the target that we're searching for has a value higher than the mid element, we discard the left half of the array i.e. begin = mid + 1.
  + If nums[mid] == element, then we found our target and we return from there.

**Binary Search and Bidirectional Scan**

A naive way to use binary search to find the first and the last position of a target is to first determine the index of **any** occurrence of the given target. Suppose we know that the target is at the index i in the array. From there on, we do a linear scan to the left and keep going until we find the first occurrence of this target. Similarly, we do a linear scan to the right to find the last position. This works just fine. However, in the worst case when our entire array (or say 90% or more of it) is filled with the target, then this is a linear-time algorithm. In that case, the linear scan will end up taking more time than the binary-search itself.

**Two Binary Searches**

Instead of using a linear-scan approach to find the boundaries once the target has been found, let's use two binary searches to find the first and last position of the target. We can make a small tweak to the checks we perform on the middle element. This tweak will help us determine the first and the last position of an element

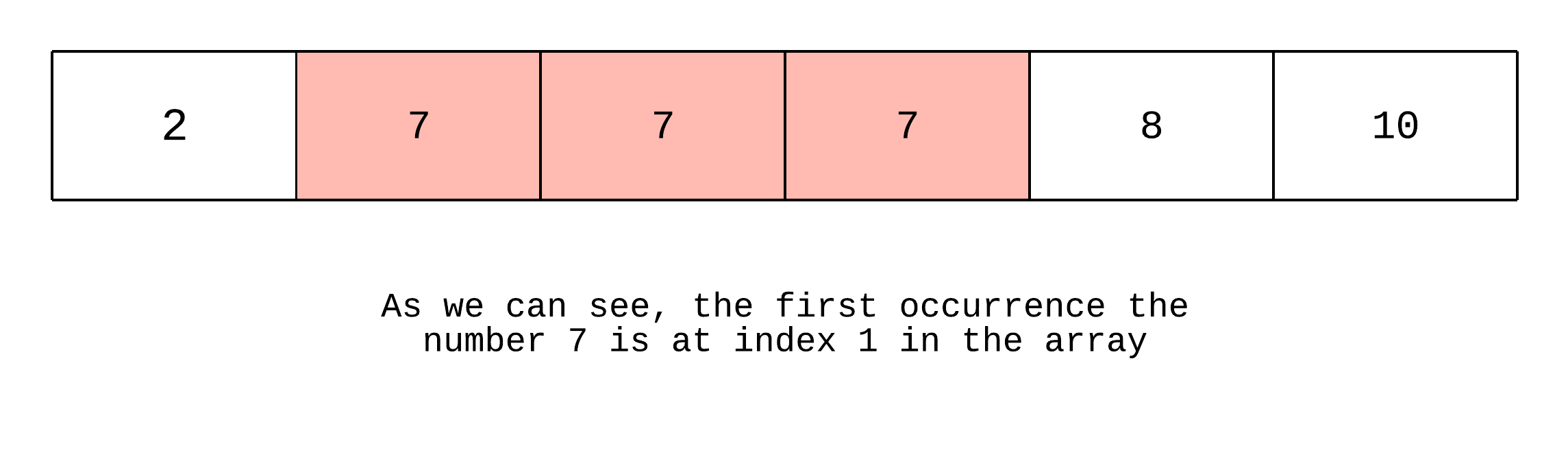
Normally, we compare nums[mid] == target because we simply need to check if we found our target or not. But now, apart from checking for equality, we also need to check if mid is the first or the last index where the target occurs. Let's see how we can do that.

**First position in the array**

There are two situations where an index will be the first occurrence of the target in the array.

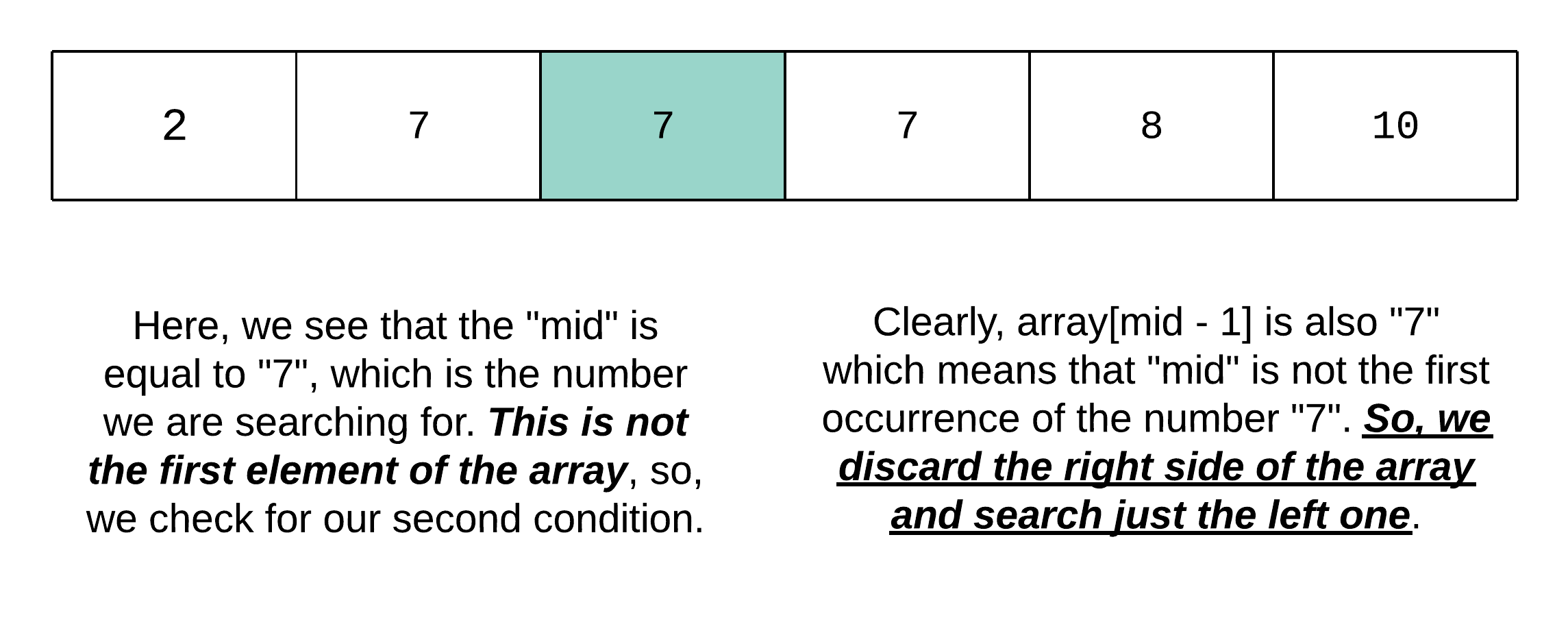
1. If mid is the same as begin which implies our mid element is the *first* element in the remaining subarray.
2. The element to the left of this index is not equal to the target that we are searching for. I.e. nums[mid - 1] != target. If this condition is not met, we should keep searching on the left side of the array for the first occurrence of the target.

Let's take a look at an example depicting this idea. In the example below, we are searching for the first occurrence of the number 7.



*Figure 1. Find the first position of "7" in the array.*

Initially, the variables begin and end are 0 and 5 respectively. So, our mid element is (0 + 5) / 2 = 2.



*Figure 2. The middle element is a match but it is not the first occurrence.*

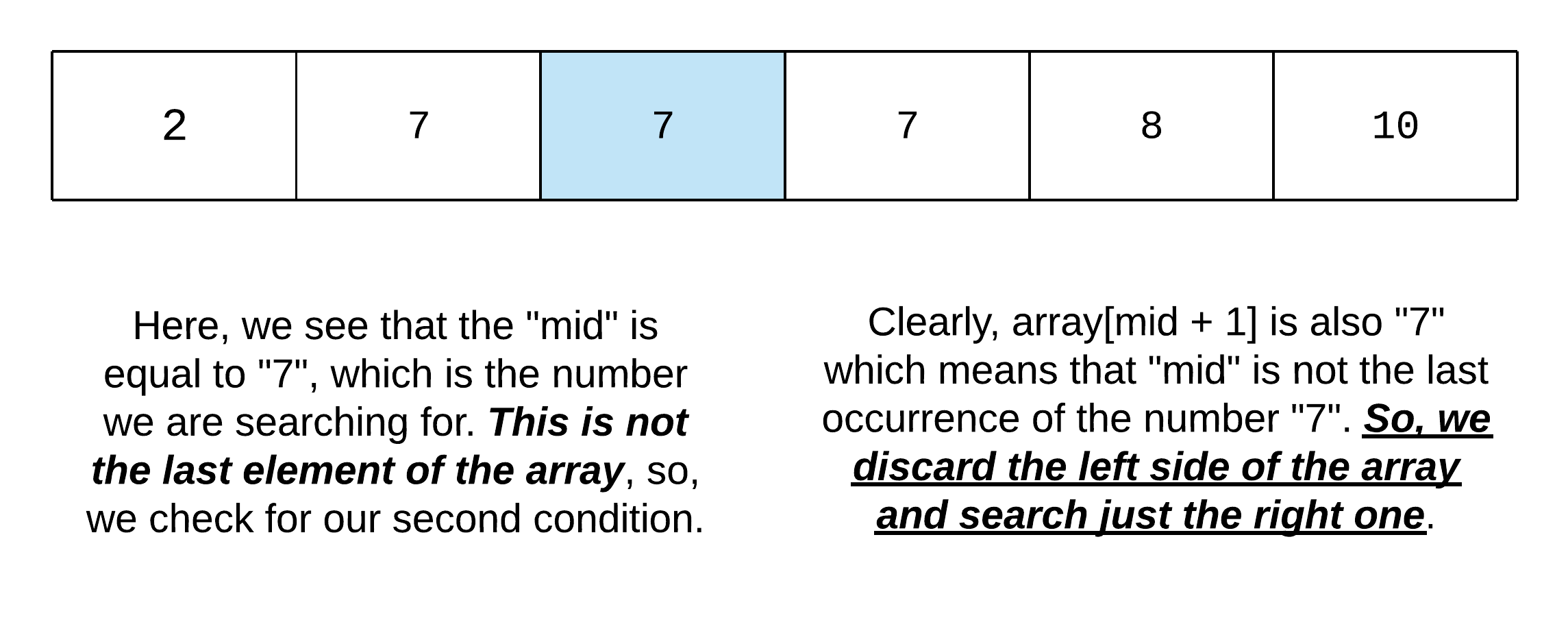
Now, begin is still 0 but end has moved to mid - 1 = 1. Now we have narrowed our search down to the first two elements of the array. Our updated mid is (0 + 1) / 2 = 0. This element is lower than the target we are looking for 2 < 7. So, we discard the "left" side of this subarray and update begin = mid + 1. This leaves us with a single index, which is in fact the first occurrence of the element "7".

**Last position in the array**

There are two situations where an index will be the last occurrence of the target in the array.

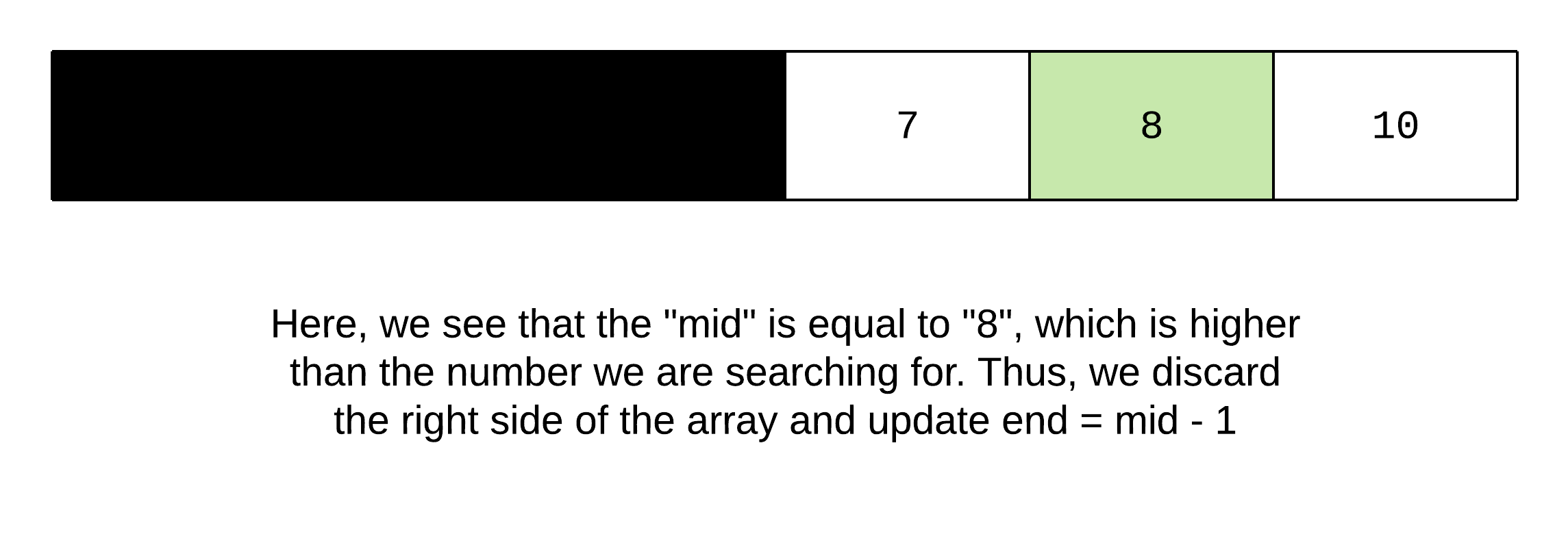
1. If mid is the same as end which implies our mid element is the *last* element of the remaining subarray.
2. If the element to the right of mid is not equal to the target we are searching for. I.e. nums[mid + 1] != target. If this condition is not met, we should keep searching on the right side of the array for the last occurrence of the target.

Let's take a look at an example depicting this idea. In the example below, we are searching for the last occurrence of the number 7.



*Figure 3. Find the last position of "7" in the array.*

Initially, the variables begin and end are 0 and 5 respectively. So, our mid element is (0 + 5) / 2 = 2.



*Figure 4. The middle element is not a match.*

The updated mid is greater than "7". So, we discard the right side of the array. This leaves us with just a single element in the array which is "7" and it is also the last occurrence.

**Algorithm**

1. Define a function called findBound which takes three arguments: the array, the target to search for, and a boolean value isFirst which indicates if we are trying to find the first or the last occurrence of target.
2. We use 2 variables to keep track of the subarray that we are scanning. Let's call them begin and end. Initially, begin is set to 0 and end is set to the last index of the array.
3. We iterate until begin is greater than or equal to end.
4. At each step, we calculate the middle element mid = (begin + end) / 2. We use the value of the middle element to decide which half of the array we need to search.
   * *nums[mid] == target*
     + *isFirst is true* ~ This implies that we are trying to find the first occurrence of the element. If mid == begin or nums[mid - 1] != target, then we return mid as the first occurrence of the target. Otherwise, we update end = mid - 1
     + *isFirst is false* ~ This implies we are trying to find the last occurrence of the element. If mid == end or nums[mid + 1] != target, then we return mid as the last occurrence of the target. Otherwise, we update begin = mid + 1
   * *nums[mid] > target* ~ We update end = mid - 1 since we must discard the right side of the array as the middle element is greater than target.
   * *nums[mid] < target* ~ We update begin = mid + 1 since we must discard the left side of the array as the middle element is less than target.
5. We return a value of -1 at the end of our function which indicates that target was not found in the array.
6. In the main searchRange function, we first call findBound with isFirst set to true. If this value is -1, we can simply return [-1, -1]. Otherwise, we call findBound with isFirst set to false to get the last occurrence and then return the result.

**Implementation**

class Solution {

public int[] searchRange(int[] nums, int target) {

int firstOccurrence = this.findBound(nums, target, true);

if (firstOccurrence == -1) {

return new int[]{-1, -1};

}

int lastOccurrence = this.findBound(nums, target, false);

return new int[]{firstOccurrence, lastOccurrence};

}

private int findBound(int[] nums, int target, boolean isFirst) {

int N = nums.length;

int begin = 0, end = N - 1;

while (begin <= end) {

int mid = (begin + end) / 2;

if (nums[mid] == target) {

if (isFirst) {

// This means we found our lower bound.

if (mid == begin || nums[mid - 1] != target) {

return mid;

}

// Search on the left side for the bound.

end = mid - 1;

} else {

// This means we found our upper bound.

if (mid == end || nums[mid + 1] != target) {

return mid;

}

// Search on the right side for the bound.

begin = mid + 1;

}

} else if (nums[mid] > target) {

end = mid - 1;

} else {

begin = mid + 1;

}

}

return -1;

}

}

**Complexity Analysis**

* Time Complexity: O(\text{log} N)*O*(log*N*) considering there are N*N* elements in the array. This is because binary search takes logarithmic time to scan an array of N*N* elements. Why? Because at each step we discard half of the array we are scanning and hence, we're done after a logarithmic number of steps. We simply perform binary search twice in this case.
* Space Complexity: O(1)*O*(1) since we only use space for a few variables and our result array, all of which require constant space.