Solution: Search in Rotated Sorted Array

Let's solve the Search in Rotated Sorted Array problem using the Modified Binary Search pattern.

We'll cover the following

Statement

Given a sorted integer array, `nums`, and an integer value, `target`, the array is rotated by some arbitrary number. Search and return the index of `target` in this array. If the `target` does not exist, return `-1`.

An original sorted array before rotation is given below:

1	10	20	47	59	63	75	88	99	107	120	133	155	162	176	188	199	200	210	222

After rotating this array 6 times, it changes to:

176	188	199	200	210	222	1	10	20	47	59	63	75	88	99	107	120	133	155	162

Constraints:

- All values in `nums` are unique.
- The values in `nums` are sorted in ascending order.
- The array may have been rotated by some arbitrary number.
- $1 \leq$ `nums.length` ≤ 5000
- $-10^4 \leq$ `nums[i]` $\leq 10^4$
- $-10^4 \leq$ `target` $\leq 10^4$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

A naive approach is to traverse the whole array while searching for our target.

We get the required solution, but at what cost? The time complexity is $O(n)$ because we traverse the array only once, and the space complexity is $O(1)$. Let's see if we can use the modified binary search pattern to design a more efficient solution.

Optimized approach using modified binary search

We've been provided with a rotated array to apply binary search, which is faster than the above linear approach. We can change the part we have to search based on our three pointers—`low`, `mid`, and `high`.

The slides below illustrate how we would like the algorithm to run:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
176	188	199	200	210	222	1	10	20	47	59	63	75	88	99	107	120	133	155	162

target: 200

This is our initial setup.

1 of 13

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Let's start with learning how to use binary search to find a target value in an unrotated sorted array. We can do this either iteratively or recursively. Let's look at the iterative version first.

Java

```java
import java.util.*;

class RotatedSearch {
    public static int binarySearch(List<Integer> nums, int target) {
        int start = 0;
        int end = nums.size() - 1;
        int mid = 0;
        if (start > end)
            return -1;

        while (start <= end) {
            // Finding the mid using integer division
            mid = start + (end - start) / 2;
            // Target value is present at the middle of the array
            if (nums.get(mid) == target)
                return mid;
            // If the target value is greater than the middle, ignore the first half
            else if (nums.get(mid) < target)
                start = mid + 1;
            // If the target value is less than the middle, ignore the second half
            else
                end = mid - 1;
        }
        return -1;
    }

    public static void main(String args[]) {
```

```
        List<Integer> targetList = Arrays.asList(3, 60, 15, 11);
```

Search in Rotated Sorted Array

Next, let's look at the recursive version.

Java

```
import java.util.*;

class RotatedSearch {
    public static int binarySearch(List<Integer> nums, int start, int end, int target) {
        if (start > end)
            return -1;

        // Finding the mid using integer division
        int mid = start + (end - start) / 2;
        // Target value is present at the middle of the array
        if (nums.get(mid) == target)
            return mid;
        // If the target value is greater than the middle, ignore the first half
        else if (nums.get(mid) < target)
            return binarySearch(nums, mid + 1, end, target);
        // If the target value is less than the middle, ignore the second half
        return binarySearch(nums, start, mid - 1, target);
    }

    public static void main(String args[]) {
        List<Integer> targetList = Arrays.asList(3, 60, 15, 11);
        List<List<Integer>> numList = Arrays.asList(
            Arrays.asList(1, 2, 3, 4, 5, 6, 7),
            Arrays.asList(10, 20, 30, 40, 50, 60),
            Arrays.asList(11, 15, 200, 432, 765, 1000),
            Arrays.asList(3, 5, 7, 9, 11, 13)
        );
        for (int i = 0; i < targetList.size(); i++) {
```

Search in Rotated Sorted Array

Binary search works with arrays that are completely sorted. However, the `nums` array that we're provided may not have this property if it's rotated. Therefore, we have to modify our binary search to accommodate this rotation.

You may notice that at least one half of the array is always sorted—if the array is rotated by *less* than half the length of the array, at least the second half of the array will still be sorted. Contrarily, if the array is rotated by *more* than half the length of the array, then at least the first half of the array will be sorted. We can use this property to our advantage and modify the `binarySearch` function as follows:

- If the `target` value lies within the sorted half of the array, our problem is a basic binary search.
- Otherwise, discard the sorted half and keep examining the unsorted half.

Here is how we go about implementing the iterative approach for this:

Java

```
import java.util.*;

class RotatedSearch {

    public static int binarySearchRotated(List<Integer> nums, int target) {
        int start = 0;
        int end = nums.size() - 1;
        if (start > end)
            return -1;
```

```
            return -1;
        while (start <= end) {
            // Finding the mid using integer division
            int mid = start + (end - start) / 2;
            // Target value is present at the middle of the array
            if (nums.get(mid) == target)
                return mid;
            // start to mid is sorted
            if (nums.get(start) <= nums.get(mid)) {
                if (nums.get(start) <= target && target < nums.get(mid)) {
                    end = mid - 1; // target is within the sorted first half of the array
                } else {
                    start = mid + 1; // target is not within the sorted first half, so let's examine the
                }
            }
            // mid to end is sorted
            else {
                if (nums.get(mid) < target && target <= nums.get(end))
                    start = mid + 1; // target is within the sorted second half of the array
                else
```

Search in Rotated Sorted Array

The recursive approach is shown below:

Java

```
import java.util.*;

class RotatedSearch {

    public static int binarySearch(List<Integer> nums, int start, int end, int target) {
        if (start > end)
            return -1;
        // Finding the mid using integer division
        int mid = start + (end - start) / 2;
        // Target value is present at the middle of the array
        if (nums.get(mid) == target)
            return mid;
        // start to mid is sorted
        if (nums.get(start) <= nums.get(mid)) {
            if (nums.get(start) <= target && target < nums.get(mid)) {
                // target is within the sorted first half of the array
                return binarySearch(nums, start, mid - 1, target);
            }
            // target is not within the sorted first half, so let's examine the unsorted second half
            return binarySearch(nums, mid + 1, end, target);
        }
        // mid to end is sorted
        else {
            if (nums.get(mid) < target && target <= nums.get(end))
                return binarySearch(nums, mid + 1, end, target); // target is within the sorted second ha
            return binarySearch(nums, start, mid - 1, target); // target is not within the sorted second
        }
    }
```

Search in Rotated Sorted Array

Just the code

Here's the complete solution to this problem:

The iterative approach:

Java

```
import java.util.*;

```

```java
class RotatedSearch {

    public static int binarySearchRotated(List<Integer> nums, int target) {
        int start = 0;
        int end = nums.size() - 1;
        if (start > end)
            return -1;
        while (start <= end) {
            int mid = start + (end - start) / 2;
            if (nums.get(mid) == target)
                return mid;
            if (nums.get(start) <= nums.get(mid)) {
                if (nums.get(start) <= target && target < nums.get(mid)) {
                    end = mid - 1;
                } else
                    start = mid + 1;
            } else {
                if (nums.get(mid) < target && target <= nums.get(end))
                    start = mid + 1;
                else
                    end = mid - 1;
            }
        }
        return -1;
    }
```

Search in Rotated Sorted Array

The recursive approach:

Java

```java
import java.util.*;

class RotatedSearch {
    public static int binarySearch(List<Integer> nums, int start, int end, int target) {
        if (start > end) return -1;
        int mid = start + (end - start) / 2;
        if (nums.get(mid) == target) return mid;
        if (nums.get(start) <= nums.get(mid)) {
            if (nums.get(start) <= target && target < nums.get(mid)) {
                return binarySearch(nums, start, mid - 1, target);
            }
            return binarySearch(nums, mid + 1, end, target);
        } else {
            if (nums.get(mid) < target && target <= nums.get(end)) {
                return binarySearch(nums, mid + 1, end, target);
            }
            return binarySearch(nums, start, mid - 1, target);
        }
    }

    public static int binarySearchRotated(List<Integer> nums, int target) {
        return binarySearch(nums, 0, nums.size() - 1, target);
    }

    public static void main(String args[]) {
        List<Integer> targetList = Arrays.asList(3, 60, 15, 11);
        List<List<Integer>> numList = Arrays.asList(
            Arrays.asList(1, 2, 3, 4, 5, 6, 7)
```

Search in Rotated Sorted Array

Solution summary

To recap, the solution to this problem can be divided into the following five parts:

2. Check if the first half is sorted, and if it is, do the following:
    - Check if the target lies in this range, and if it does, perform a binary search on this half for the target value.
    - If the target does not lie in this range, move to the second half of the array.
3. If the first half is not sorted, check if the target lies in the second half.
    - If the target lies in this half, perform a binary search on this half for the target value.
    - If the target does not lie in the second half, examine the first half.
4. If the target value is not found at the end of this process, we return `-1`.

Time complexity

The time complexity of both approaches is $O(\log n)$ since we divide the array into two halves at each step.

Space complexity

The space complexity of the iterative solution is $O(1)$ since no new data structure is being created.

The space complexity of the recursive solution is $O(\log n)$, where n is the number of elements present in the array and $\log n$ is the maximum number of recursive calls needed to find the target.

← Back
Search in Rotated Sort...

Next →
First Bad Version

Mark as Completed