**Modified Binary Search: Introduction**

Let’s understand the Modified Binary Search pattern, its real-world applications, and some problems we can solve with it.

**We'll cover the following**

* [Overview](https://www.educative.io/courses/grokking-coding-interview-patterns-java/modified-binary-search-introduction#Overview)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/modified-binary-search-introduction#Examples)
* [Does my problem match this pattern?](https://www.educative.io/courses/grokking-coding-interview-patterns-java/modified-binary-search-introduction#Does-my-problem-match-this-pattern)
* [Real-world problems](https://www.educative.io/courses/grokking-coding-interview-patterns-java/modified-binary-search-introduction#Real-world-problems)
* [Strategy time!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/modified-binary-search-introduction#Strategy-time)

**Overview**

The modified binary search pattern is an extension of the traditional binary search algorithm and can be applied to a wide range of problems. Before we delve into the modified version, let’s first recap the classic binary search algorithm.

**Classic Binary Search**

Binary search is an efficient search algorithm for searching a target value in sorted arrays or sorted lists that support direct addressing (also known as [random access](https://www.educative.io/answers/difference-between-sequential-and-random-access-to-storage)). It follows a divide-and-conquer approach, significantly reducing the search space with each iteration. The algorithm uses three indexes—start, end, and middle—and proceeds as follows:

1. Set the start and end indexes to the first and last elements of the array, respectively.
2. Calculate the position of the middle index by taking the average of the start and end indexes. For example, if *start*=0 and *end*=7, then *middle*=⌊(0+7)/2⌋=3.
3. Compare the target value with the element at the middle index.
4. If the target value is equal to the middle index element, we have found the target, and the search terminates.
5. If the target value is less than the middle index element, update the end index to *middle*−1 and repeat from step 22 onwards. Since the array is sorted, all the values between the middle and the end indexes will also be greater than the target value. Therefore, there’s no reason to consider that half of the search space.
6. If the target value is greater than the middle index element, update the start index to *middle*+1 and repeat from step 22. Again, since the array is sorted, all the values between the start and the middle indexes will also be less than the target value. Therefore, there’s no reason to consider that half of the search space.
7. Continue the process until the target value is found or if the search space is exhausted, that is, if the start index has crossed the end index. This means that the algorithm has explored all possible values, which implies that the search space is now empty and the target value is not present.

**Note:** We’re assuming the array is sorted in ascending order. If it’s sorted in descending order, we’ll do the opposite when changing the positions of the start and end pointers.

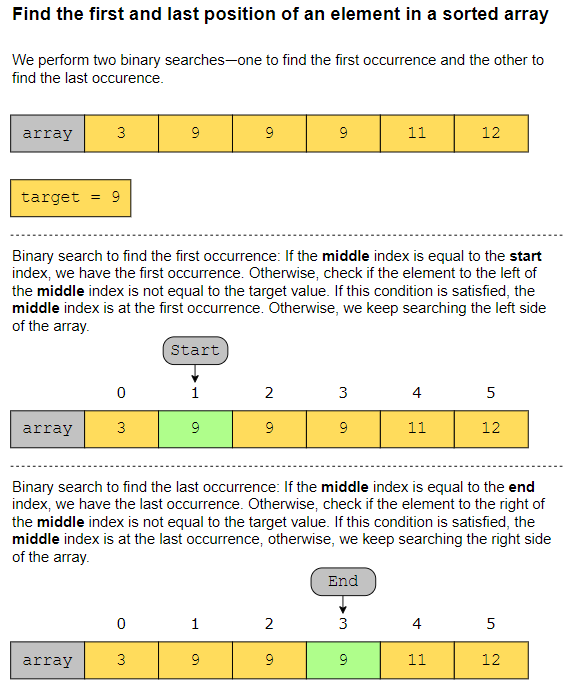
Binary search reaches the target value in *O*(log(*n*)) time, since we divide the array into two halves at each step, and then focus on only one of these halves. Had we opted for the brute-force approach, we would have had to traverse the entire array, without any partitioning, to search for the target value, which would take *O*(*n*) in the worst case.

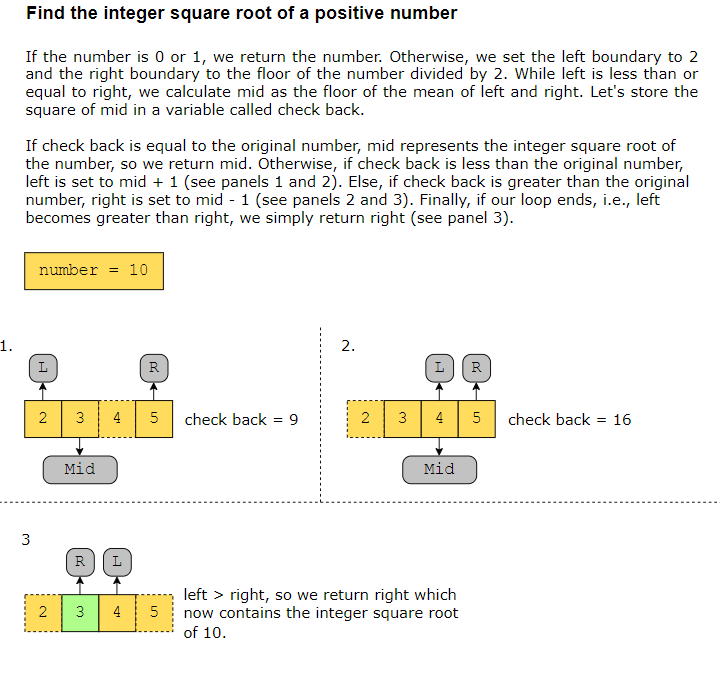
**Modified Binary Search**

The modified binary search pattern builds upon the basic binary search algorithm discussed above. It involves adapting the traditional binary search approach by applying certain conditions or transformations, allowing us to solve more complex problems.

A few common variations of the modified binary search pattern are:

1. **Binary search on a modified array:** Sometimes, the array may be modified in a certain way, which affects the search process. For example, the array might be sorted and then rotated around some unknown pivot. Alternatively, some elements in a sorted array might be modified based on a specific condition. To handle such scenarios, we can modify the basic binary search technique to detect anomalies in the sorted order.
2. **Binary search with multiple conditions:** When searching for a target satisfying multiple conditions, a modified binary search can be used. It involves adapting the comparison logic within the binary search to accommodate multiple conditions. Examples include finding a target range rather than a single target, or finding the leftmost, or the rightmost occurrence of a target value.





**Does my problem match this pattern?**

* Yes, if either of these conditions is fulfilled:
  + The problem requires us to find a target value (or its first or last occurrence) in a sorted array or a sorted list that supports direct addressing.
  + We may use this pattern when segments of an input array are sorted, even if the whole list does not, at first sight, seem to be sorted. For example, we can modify the binary search technique to find a target in a sorted array that has been rotated around an arbitrary pivot.
* No, if either of these conditions is fulfilled:
  + The input data structure does not support direct addressing.
  + The data to search is not sorted according to criteria relevant to the search. For example, if we’re looking for a particular date in a list of dates, but the list is sorted in alphabetical order (and not chronologically), we cannot use binary search.
  + The solution does not require us to find a particular value or range of values.

**Real-world problems**

Many problems in the real world use the modified binary search pattern. Let’s look at some examples.

* Dictionary: A dictionary contains words that are alphabetically sorted. Therefore, we can use classic binary search to find the required word quickly. If we wanted to find all the words in the dictionary that share a common prefix, we could modify the comparison operation used in the classic binary search algorithm.
* Debugging with minimal support: Let’s suppose that a code script consists of �*n* lines, and that there is a bug somewhere in the script. Binary search is performed to find the bug by dividing the code based on line numbers. For example, if the code does not run for the first �/2*n*/2 lines, we further divide the code and check if it runs for the first �/4*n*/4 lines and so on.
* Student documents: Given a list of students sorted by their scores on a test, find all the students who scored between 40%40% and 55%55%.

**Strategy time!**

Match the problems that can be solved using the modified binary search pattern.

**Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.

Q**. Detect a cycle in a graph.**

A. Detecting a cycle in a graph involves traversing and exploring the entire graph rather than searching for a specific value or condition. Graph traversal algorithms like Depth-first Search (DFS) or Breadth-first Search (BFS) are more suitable for solving this problem.

Q. **Check if a number is the majority element in a sorted array.**

**A.** We can use binary search to find the index of the first occurrence of the given number. We can then calculate if that number exists more than n/2 times in the array.

Q. **Find a peak element in an array, where the peak element is the one that’s strictly greater than its immediate left and right neighbors. Assume that adjacent elements in the array can never be equal.**

**A.** The Modified Binary Search approach efficiently narrows down the search space by comparing the middle element with its adjacent neighbors. We move to the particular half of the array based on this comparison. The first and last elements can also be considered as peaks if they’re greater than their only neighbor.

**Q. Shift all 0s in an array to the right.**

**A.** This problem requires iterating through the array, and identifying and moving 0s to the right, while maintaining the order of other elements. A different approach, such as Two Pointers, would be more suitable for solving this problem efficiently.

**Binary Search**

Try to solve the Binary Search problem.

**We'll cover the following**

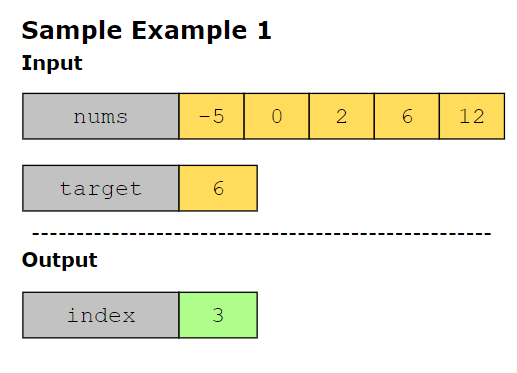
* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/binary-search#Statement)
* [Example](https://www.educative.io/courses/grokking-coding-interview-patterns-java/binary-search#Example)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/binary-search#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/binary-search#Figure-it-out)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/binary-search#Try-it-yourself)

**Statement**

We are given an array of integers, nums, sorted in ascending order, and an integer value, target. If the target exists in the array, return its index. If the target does not exist, return -1.

**Constraints:**

* 11≤≤ nums.length ≤≤ 103103
* −104−104≤≤ nums[i] , target ≤≤ 104104
* All integers in nums are unique.
* nums is sorted in ascending order.



## Understand the problem

Let’s take a moment to make sure you’ve correctly understood the problem. The quiz below helps you check if you’re solving the correct problem:

Find the index of the number 0 from the following array using binary search.

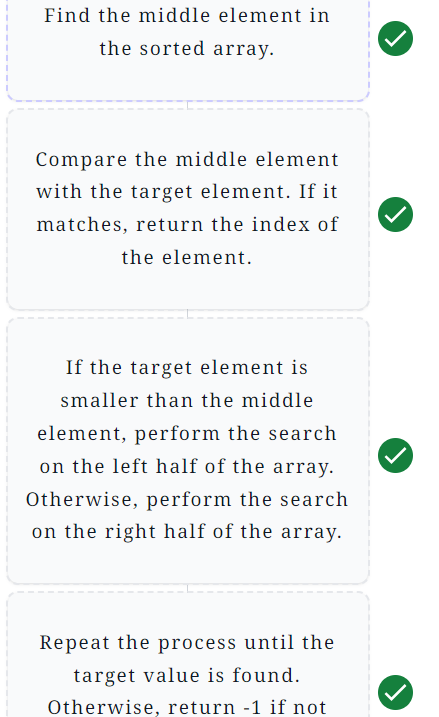
array = [-30, 0, 1, 2, 3,15]

Also, calculate the number of comparisons required to find the number.

1. index: 1, steps: 3
   1. We first calculate the mid element of the array, which is 1. The target element is less than 1, so we calculate the mid of the left subarray [-30, 0]. The mid of this array is -30, which is less than the target element. Now, we calculate the mid of the right subarray [0], which is 0, equal to the target element. So, 0 exists at index 1, and we found it in three comparisons.

## Figure it out!

We have a game for you to play. Rearrange the logical building blocks to develop a clearer understanding of how to solve this problem.



**Solution: Binary Search**

Let's solve the Binary Search problem using the Modified Binary Search pattern.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-binary-search#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-binary-search#Solution)
  + [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-binary-search#Time-complexity)
  + [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-binary-search#Space-complexity)

**Statement**

We are given an array of integers, nums, sorted in ascending order, and an integer value, target. If the target exists in the array, return its index. If the target does not exist, return -1.

**Constraints:**

* 11≤≤ nums.length ≤≤ 103103
* −104−104≤≤ nums[i] , target ≤≤ 104104
* All integers in nums are unique.
* nums is sorted in ascending order.

**Solution**

The array provided is sorted, so whenever target is greater than any number in the array, target must be present in the subarray to the right of the number. Similarly, whenever target is less than any number in the array, target must be present in the subarray to the left of the number. We can solve the above problem using the iterative approach by following the steps below:

1. Let’s initialize the low and high variables to 00 and nums.length-1, respectively.
2. Calculate the mid index using the formula *mid*=*low*+((*high*−*low*)/2).
   1. If nums[mid] is equal to the target value, we return mid.
   2. If nums[mid] is greater than target, point high to mid-1, and the value of low remains the same. Now we will search the left part of the array.
   3. If nums[mid] is less than target, point low to mid + 1, and the value of high remains the same. Now we will search the right part of the array.
3. When the value of low is greater than the value of high, this indicates that the target is not present in the array. This is because we’ve checked all possible positions where target might exist. So, -1 is returned.
4. import java.util.\*;
5. public class main {
6. public static int binarySearch(int[] nums, int target) {
7. int low = 0;
8. int high = nums.length - 1;
9. while (low <= high) {
10. int mid = low + (high - low) / 2;
11. if (nums[mid] == target) {
12. return mid;
13. }
14. else if (target < nums[mid]) {
15. high = mid - 1;
16. }
17. else if (target > nums[mid]) {
18. low = mid + 1;
19. }
20. }
21. return -1;
22. }
23. public static void main(String[] args) {
24. int[][] numsLists = {
25. {},
26. {0, 1},
27. {1, 2, 3},
28. {-1, 0, 3, 5, 9, 12},
29. {-100, -67, -55, -50, -49, -40, -33, -22, -10, -5}
30. };
31. int[] targetList = {12, 1, 3, 9, -22};
33. for (int i = 0; i < numsLists.length; i++) {
34. int[] nums = numsLists[i];
35. int target = targetList[i];
36. int index = binarySearch(nums, target);
37. System.out.println((i + 1) + ".\tArray to search: " + Arrays.toString(nums));
38. System.out.println("\tTarget: " + target);
39. if (index != -1) {
40. System.out.println("\t" + target + " exists in the array at index " + index);
41. } else {
42. System.out.println("\t" + target + " does not exist in the array, so the return value is " + index);
43. }
44. System.out.println(new String(new char[100]).replace('\0', '-'));
45. }
46. }
47. }

### Time complexity

The time complexity of this solution is logarithmic, *O*(log *n*), where *n* is the number of elements in the array.

### Space complexity

The space complexity of this solution is constant, *O*(1).

# Solution: Search in Rotated Sorted Array

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

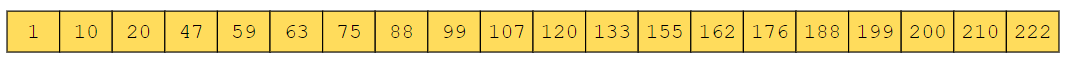
**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Statement)
* [Solution](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Solution)
  + [Naive approach](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Naive-approach)
  + [Optimized approach using modified binary search](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Optimized-approach-using-modified-binary-search)
    - [Step-by-step solution construction](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Step-by-step-solution-construction)
    - [Just the code](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Just-the-code)
    - [Solution summary](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Solution-summary)
    - [Time complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#Space-complexity)

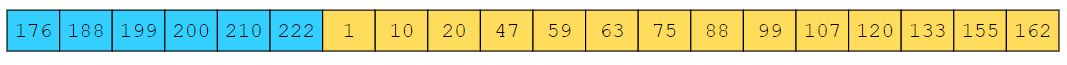
## Statement

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 -11.

An original sorted array before rotation is given below:



After rotating this array 6 times, it changes to:



**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≤1≤ nums.length ≤1000≤1000
* −104≤−104≤ nums[i] ≤104≤104
* −104≤−104≤ target ≤104≤104

## 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:

**Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to [just the code](https://www.educative.io/courses/grokking-coding-interview-patterns-java/solution-search-in-rotated-sorted-array#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.

import java.util.\*;

class Solution {

public static int binarySearch(List<Integer> nums, int target) {

int start = 0;

int end = nums.size() - 1;

int mid = 0;

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;

}

// Driver code

public static void main(String args[]) {

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(12, 24, 35, 47, 58, 69, 72, 83, 94),

Arrays.asList(5, 13, 28, 41, 56, 63, 77, 82, 99, 105),

Arrays.asList(5, 7, 12, 17, 21, 28, 33, 37, 41, 48, 52, 57, 62, 68, 72)

);

List<Integer> targetList = Arrays.asList(1, 50, 12, 56, 5);

for (int i = 0; i < targetList.size(); i++) {

System.out.println((i + 1) + ".\tSorted array: " + numList.get(i) +

"\n\ttarget " + targetList.get(i) +

" found at index " + binarySearch(numList.get(i), targetList.get(i)));

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

Next, let’s look at the recursive version.

import java.util.\*;

class Solution {

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);

}

// Driver code

public static void main(String args[]) {

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(12, 24, 35, 47, 58, 69, 72, 83, 94),

Arrays.asList(5, 13, 28, 41, 56, 63, 77, 82, 99, 105),

Arrays.asList(5, 7, 12, 17, 21, 28, 33, 37, 41, 48, 52, 57, 62, 68, 72)

);

List<Integer> targetList = Arrays.asList(1, 50, 12, 56, 5);

for (int i = 0; i < targetList.size(); i++) {

System.out.println((i + 1) + ".\tSorted array: " + numList.get(i) +

"\n\ttarget " + targetList.get(i) +

" found at index " + binarySearch(numList.get(i), 0, numList.get(i).size() - 1, targetList.get(i)));

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

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:

import java.util.\*;

class Solution {

public static int binarySearchRotated(List<Integer> nums, int target) {

int start = 0;

int end = nums.size() - 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 unsorted second half

}

}

// 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

end = mid - 1; // target is not within the sorted second half, so let’s examine the unsorted first half

}

}

return -1;

}

// Driver code

public static void main(String args[]) {

List<List<Integer>> numList = Arrays.asList(

Arrays.asList(5, 6, 7, 1, 2, 3, 4),

Arrays.asList(40, 50, 60, 10, 20, 30),

Arrays.asList(47, 58, 69, 72, 83, 94, 12, 24, 35),

Arrays.asList(77, 82, 99, 105, 5, 13, 28, 41, 56, 63),

Arrays.asList(48, 52, 57, 62, 68, 72, 5, 7, 12, 17, 21, 28, 33, 37, 41)

);

List<Integer> targetList = Arrays.asList(1, 50, 12, 56, 5);

for (int i = 0; i < targetList.size(); i++) {

System.out.println((i + 1) + ".\tSorted array: " + numList.get(i) +

"\n\ttarget " + targetList.get(i) +

" found at index " + binarySearchRotated(numList.get(i), targetList.get(i)));

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

1. Sorted array: [5, 6, 7, 1, 2, 3, 4] target 1 found at index 3 ---------------------------------------------------------------------------------------------------- 2. Sorted array: [40, 50, 60, 10, 20, 30] target 50 found at index 1 ---------------------------------------------------------------------------------------------------- 3. Sorted array: [47, 58, 69, 72, 83, 94, 12, 24, 35] target 12 found at index 6 ---------------------------------------------------------------------------------------------------- 4. Sorted array: [77, 82, 99, 105, 5, 13, 28, 41, 56, 63] target 56 found at index 8 ---------------------------------------------------------------------------------------------------- 5. Sorted array: [48, 52, 57, 62, 68, 72, 5, 7, 12, 17, 21, 28, 33, 37, 41] target 5 found at index 6 ----------------------------------------------------------------------------------------------------

The recursive approach is shown below:

import java.util.\*;

class Solution {

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 half of the array

return binarySearch(nums, start, mid - 1, target); // target is not within the sorted second half, so let’s examine the unsorted first half

}

}

public static int binarySearchRotated(List<Integer> nums, int target) {

return binarySearch(nums, 0, nums.size() - 1, target);

}

// Driver code

public static void main(String args[]) {

List<List<Integer>> numList = Arrays.asList(

Arrays.asList(5, 6, 7, 1, 2, 3, 4),

Arrays.asList(40, 50, 60, 10, 20, 30),

Arrays.asList(47, 58, 69, 72, 83, 94, 12, 24, 35),

Arrays.asList(77, 82, 99, 105, 5, 13, 28, 41, 56, 63),

Arrays.asList(48, 52, 57, 62, 68, 72, 5, 7, 12, 17, 21, 28, 33, 37, 41)

);

List<Integer> targetList = Arrays.asList(1, 50, 12, 56, 5);

for (int i = 0; i < targetList.size(); i++) {

System.out.println((i + 1) + ".\tSorted array: " + numList.get(i) +

"\n\ttarget " + targetList.get(i) +

" found at index " + binarySearchRotated(numList.get(i), targetList.get(i)));

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

#### Just the code

Here’s the complete solution to this problem:

The iterative approach:

import java.util.\*;

class Solution {

public static int binarySearchRotated(List<Integer> nums, int target) {

int start = 0;

int end = nums.size() - 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;

}

// Driver code

public static void main(String args[]) {

List<List<Integer>> numList = Arrays.asList(

Arrays.asList(5, 6, 7, 1, 2, 3, 4),

Arrays.asList(40, 50, 60, 10, 20, 30),

Arrays.asList(47, 58, 69, 72, 83, 94, 12, 24, 35),

Arrays.asList(77, 82, 99, 105, 5, 13, 28, 41, 56, 63),

Arrays.asList(48, 52, 57, 62, 68, 72, 5, 7, 12, 17, 21, 28, 33, 37, 41)

);

List<Integer> targetList = Arrays.asList(1, 50, 12, 56, 5);

for (int i = 0; i < targetList.size(); i++) {

System.out.println((i + 1) + ".\tSorted array: " + numList.get(i) +

"\n\ttarget " + targetList.get(i) +

" found at index " + binarySearchRotated(numList.get(i), targetList.get(i)));

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

The recursive approach:

import java.util.\*;

class Solution {

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);

}

// Driver code

public static void main(String args[]) {

List<List<Integer>> numList = Arrays.asList(

Arrays.asList(5, 6, 7, 1, 2, 3, 4),

Arrays.asList(40, 50, 60, 10, 20, 30),

Arrays.asList(47, 58, 69, 72, 83, 94, 12, 24, 35),

Arrays.asList(77, 82, 99, 105, 5, 13, 28, 41, 56, 63),

Arrays.asList(48, 52, 57, 62, 68, 72, 5, 7, 12, 17, 21, 28, 33, 37, 41)

);

List<Integer> targetList = Arrays.asList(1, 50, 12, 56, 5);

for (int i = 0; i < targetList.size(); i++) {

System.out.println((i + 1) + ".\tSorted array: " + numList.get(i) +

"\n\ttarget " + targetList.get(i) + " found at index " +

binarySearchRotated(numList.get(i), targetList.get(i)));

System.out.println(new String(new char[100]).replace('\0', '-'));

}

}

}

#### Solution summary

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

1. Divide the array into two halves.
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 -11.

#### 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 llog*n* is the maximum number of recursive calls needed to find the target.