1. Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.

Input: nums = [4,5,6,7,0,1,2], target = 0

Output: 4

**Solution:**

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

int start = 0;

int end = nums.length - 1;

while (start <= end){



int mid = (start + end) / 2;

if (nums[mid] == target)

return mid;

if (nums[start] <= nums[mid]){

if (target < nums[mid] && target >= nums[start])

end = mid - 1;

else

start = mid + 1;

}

if (nums[mid] <= nums[end]){

if (target > nums[mid] && target <= nums[end])

start = mid + 1;

else

end = mid - 1;

}

}// while loop closed

return -1;

}

TC: O(logn)

SC: O(1)

1. Given a sorted array of non-negative distinct integers, find the smallest missing non-negative element in it.

Input: nums[] = [0, 1, 2, 6, 9, 11, 15]

Output: The smallest missing element is 3

**Solution:**

    public static int findSmallestMissing(int[] nums, int left, int right)

    {

        // base condition

        if (left > right) {

            return left;

        }



        int mid = left + (right - left) / 2;

        // if the mid-index matches with its value, then the mismatch

        // lies on the right half

        if (nums[mid] == mid) {



            return findSmallestMissing(nums, mid + 1, right);



        }

        else {

            // mismatch lies on the left half

            return findSmallestMissing(nums, left, mid - 1);



        }

    }// function closed

TC: O(log n)

SC: O(log n) bcoz of implicit stack call.

1. Given a sorted binary array, efficiently count the total number of 1’s in it.

Input: nums[] = [0, 0, 0, 0, 1, 1, 1]

Output: The total number of 1’s present is 3

**Solution:**

public static int count(int[] nums, int left, int right)

    {

        // base case

        if (nums == null || nums.length == 0) {

            return 0;

        }



        // if the last array element is 0, no 1's can



        // be present since it is sorted



        if (nums[right] == 0) {

            return 0;

        }



        // if the first array element is 1, all its elements

        // are ones only since it is sorted

        if (nums[left] == 1) {

            return (right - left + 1);

        }

        // divide the array into left and right subarray and recur

        int mid = (left + right) / 2;

        return count(nums, left, mid) + count(nums, mid + 1, right);



    }

TC: O(logn)

SC: O(logn)

1. Given an integer array, find the peak element in it. A peak element is an element that is greater than its neighbours. There might be multiple peak elements in an array, and the solution should report any peak element.

Input : [8, 9, 10, 2, 5, 6]

Output: The peak element is 10 (or 6)

**Solution:**

public static int findPeakElement(int[] nums, int left, int right)

    {

        // find the middle element. To avoid overflow, use mid = low + (high - low) / 2



        int mid = (left + right) / 2;

        // check if the middle element is greater than its neighbors

        if ((mid == 0 || nums[mid - 1] <= nums[mid]) &&

                (mid == nums.length - 1 || nums[mid + 1] <= nums[mid])) {

            return mid;

        }

        // If the left neighbor of `mid` is greater than the middle element,

        // find the peak recursively in the left subarray

        if (mid - 1 >= 0 && nums[mid - 1] > nums[mid]) {

            return findPeakElement(nums, left, mid - 1);

        }

        // If the right neighbor of `mid` is greater than the middle element,

        // find the peak recursively in the right subarray

        return findPeakElement(nums, mid + 1, right);

    }

    public static int findPeakElement(int[] nums)

    {

        // base case

        if (nums == null || nums.length == 0) {

            System.exit(-1);

        }

        int index = findPeakElement(nums, 0, nums.length - 1);

        return nums[index];

    }

TC: O(log n)

SC: O(log n)

1. Given a nearly sorted array such that each of the n elements may be misplaced by no more than one position from the correct sorted order, search a given element in it efficiently. Report if the element is not present in the array. An element at index i in a correctly sorted order can be misplaced by the ± 1 position, i.e., it can be present at index i-1 or i or i+1 in the input array.



**nums** = [2, 1, 3, 5, 4, 7, 6, 8, 9]

**target** = 5

**Output**: Element 5 found at index 3

**Solution:**

 public static int searchElement(int[] nums, int target)

    {

        // search space is nums[left…right]

        int left = 0;

        int right = nums.length - 1;

        // loop till the search space is exhausted

        while (left <= right)

        {

            // find middle index `mid` and compare nums[mid-1], nums[mid],

            // and nums[mid+1] with the target number

            int mid = (left + right) / 2;

            // return `mid` if the middle element is equal to the target number

            if (nums[mid] == target) {

                return mid;

            }

            // return `mid-1` if nums[mid-1] is equal to target number

            else if (mid - 1 >= left && nums[mid - 1] == target) {

                return mid - 1;

            }

            // return `mid+1` if nums[mid+1] is equal to target number

            else if (mid + 1 <= right && nums[mid + 1] == target) {

                return mid + 1;

            }

            // if the middle element is less than the target number,

            // reduce search space to the right subarray nums[mid+2…right]

            else if (target > nums[mid]) {

                left = mid + 2;

            }

            // if the middle element is greater than the target number,

            // reduce search space to the right subarray nums[left…mid-2]

            else {

                right = mid - 2;

            }

        }//while loop closed

        // invalid input or number is not present in the array

        return -1;

    }

TC: O(log n)

SC: O(1)

1. Given a sorted integer array, find the index of a given number’s first occurence. If the element is not present in the array, report that as well.



Solution:

 public static int findFirstOccurrence(int[] nums, int target)

    {

        // search space is nums[left…right]

        int left = 0;

        int right = nums.length - 1;

        // initialize the result by -1

        int result = -1;

        // loop till the search space is exhausted

        while (left <= right)

        {

            // find the mid-value in the search space and compares it with the target

            int mid = (left + right) / 2;

            // if the target is located, update the result and

            // search towards the left (lower indices)

            if (target == nums[mid])

            {

                result = mid;

                right = mid - 1;

            }



            // if the target is less than the middle element, discard the right half

            else if (target < nums[mid]) {

                right = mid - 1;

            }

            // if the target is more than the middle element, discard the left half

            else {

                left = mid + 1;

            }

        }

        // return the leftmost index, or -1 if the element is not found

        return result;

    }

TC: O(log n)

SC: O(1)