LeetCode Training Day 5 Binary Search I

Find the Target

The original idea of binary search is to search a particular value in a sorted array. The process will be you mark the start point (index of 0) and end point (index of size-1) in the array, calculate the middle point by middle = first + (last – first) /2, and check if array[middle] <= target then move last = middle, if not move first = middle + 1 and continue the search until first and last meet.

The code logic in binary search can easily have a bug, for example when first == 1 and last == 2, middle will be 1. (1+ (2-1)/2) = 1. If you do first = middle when array[middle] > target, you will get into an endless loop. Another issue is that we check the condition of first < last and break the loop when first == last. However if first and last meet and we get the target, we may miss the target since we break out the loop already. The ideal logic should be 3 condition blocks, check arr[middle] <, ==, or > target and move the first = middle + 1 if greater, last = middle -1 if less, and check result if equal.

Also notice in C++ and C#, if the sorted data is stored in array, vector, List or so, there is a build-in function of binary search, in C++ it is lower\_bound() and upper\_bound(), in C# it is BinarySearch.

Next Let’s discuss more deeper in the idea of binary search. If you think from another perspective. The process of a binary search can also be a process of discard what we do not need, with keep on discard whatever you do not need, you will get to what you need or confirm it does not exist.

Another trick in binary search is that when we are searching not exactly the target but the closet one, we should record the closer answer every time before we discard the range.

The standard logic in binary search is as below:

int first = 0; int last = arr.size() - 1;

int result = INT\_MIN; // assign an impossible value

while (first <= last) // loop until break

{

int middle = first + (last-first) / 2; // avoid overflow

if (arr[middle] > target)

{

last = middle - 1;

}

else if (arr[middle] < target)

{

first = middle + 1;

}

else

{

result = arr[middle];

break;

}

}

There are also some special patterns which use the idea of binary seach but do not use the logic pattern of binary search, for example, search an element in an sorted 2D matrix or find the most sutiabale item.

## 704. Binary Search

Easy

Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

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

**Example 1:**

**Input:** nums = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**Example 2:**

**Input:** nums = [-1,0,3,5,9,12], target = 2

**Output:** -1

**Explanation:** 2 does not exist in nums so return -1

**Constraints:**

* 1 <= nums.length <= 104
* -104 < nums[i], target < 104
* All the integers in nums are **unique**.
* nums is sorted in ascending order.

### Analysis:

Classical Binary Search.

/// <summary>

/// Leet code #704. Binary Search

///

/// Given a sorted (in ascending order) integer array nums of n elements

/// and a target value, write a function to search target in nums. If

/// target exists, then return its index, otherwise return -1.

///

/// Example 1:

///

/// Input: nums = [-1,0,3,5,9,12], target = 9

/// Output: 4

/// Explanation: 9 exists in nums and its index is 4

///

/// Example 2:

///

/// Input: nums = [-1,0,3,5,9,12], target = 2

/// Output: -1

/// Explanation: 2 does not exist in nums so return -1

///

/// Note:

/// 1. You may assume that all elements in nums are unique.

/// 2. n will be in the range [1, 10000].

/// 3. The value of each element in nums will be in the range

/// [-9999, 9999].

/// </summary>

int LeetCodeBinarySearch::binarySearchI(vector<int>& nums, int target)

{

int first = 0;

int last = nums.size() - 1;

int result = -1;

while (first <= last)

{

int middle = first + (last - first) / 2;

if (nums[middle] < target)

{

first = middle + 1;

}

else if (nums[middle] > target)

{

last = middle - 1;

}

else

{

result = middle;

break;

}

}

return result;

}

## 35. Search Insert Position

Easy

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

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

**Example 1:**

**Input:** nums = [1,3,5,6], target = 5

**Output:** 2

**Example 2:**

**Input:** nums = [1,3,5,6], target = 2

**Output:** 1

**Example 3:**

**Input:** nums = [1,3,5,6], target = 7

**Output:** 4

**Example 4:**

**Input:** nums = [1,3,5,6], target = 0

**Output:** 0

**Example 5:**

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

**Output:** 0

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums contains **distinct** values sorted in **ascending** order.
* -104 <= target <= 104

### Analysis:

If we find a position with value greater than the target value, it may be your insert position, so record it and replace it when you find a better one.

/// <summary>

/// Leet code #35. Search Insert Position

/// Given a sorted array and a target value, return the index if the target

/// is found. If not, return the index where it would be if it were inserted

/// in order.

/// You may assume no duplicates in the array.

/// Here are few examples.

/// [1,3,5,6], 5 -> 2

/// [1,3,5,6], 2 -> 1

/// [1,3,5,6], 7 -> 4

/// [1,3,5,6], 0 -> 0

/// </summary>

int LeetCodeBinarySearch::searchInsert(vector<int>& nums, int target)

{

// assign the first as 0, last can be the last item or out of boundary

int first = 0, last = nums.size()-1;

int result = nums.size();

// when first == last break out

while (first <= last)

{

// check the middle point, make sure out overflow

// please remember this is an int, so always left bias

int middle = first + (last - first) / 2;

// if target is located at second half, should go

// beyond middle point to avoid dead loop.

if (nums[middle] >= target)

{

result = middle;

last = middle - 1;

}

else

{

first = middle + 1;

}

}

// you can return middle or first, return first is better

// because no need to keep track middle

return result;

}

## 275. H-Index II

Medium

Given an array of integers citations where citations[i] is the number of citations a researcher received for their ith paper and citations is sorted in an **ascending order**, return compute the researcher's h**-index**.

According to the [definition of h-index on Wikipedia](https://en.wikipedia.org/wiki/H-index): A scientist has an index h if h of their n papers have at least h citations each, and the other n − h papers have no more than h citations each.

If there are several possible values for h, the maximum one is taken as the h**-index**.

You must write an algorithm that runs in logarithmic time.

**Example 1:**

**Input:** citations = [0,1,3,5,6]

**Output:** 3

**Explanation:** [0,1,3,5,6] means the researcher has 5 papers in total and each of them had received 0, 1, 3, 5, 6 citations respectively.

Since the researcher has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, their h-index is 3.

**Example 2:**

**Input:** citations = [1,2,100]

**Output:** 2

**Constraints:**

* n == citations.length
* 1 <= n <= 105
* 0 <= citations[i] <= 1000
* citations is sorted in **ascending order**.

### Analysis:

We are search for max position with value greater or equal to the index + 1.

/// <summary>

/// Leet Code 275. H-Index II

///

/// Medium

///

/// Given an array of integers citations where citations[i] is the number

/// of citations a researcher received for their ith paper and citations

/// is sorted in an ascending order, return compute the researcher's

/// h-index.

///

/// According to the definition of h-index on Wikipedia: A scientist has

/// an index h if h of their n papers have at least h citations each, and

/// the other n - h papers have no more than h citations each.

///

/// If there are several possible values for h, the maximum one is taken

/// as the h-index.

///

/// You must write an algorithm that runs in logarithmic time.

///

/// Example 1:

/// Input: citations = [0,1,3,5,6]

/// Output: 3

/// Explanation: [0,1,3,5,6] means the researcher has 5 papers in total

/// and each of them had received 0, 1, 3, 5, 6 citations respectively.

/// Since the researcher has 3 papers with at least 3 citations each and

/// the remaining two with no more than 3 citations each, their h-index

/// is 3.

///

/// Example 2:

/// Input: citations = [1,2,100]

/// Output: 2

///

/// Constraints:

/// 1. n == citations.length

/// 2. 1 <= n <= 10^5

/// 3. 0 <= citations[i] <= 1000

/// 4. citations is sorted in ascending order.

/// </summary>

int LeetCodeBinarySearch::hIndexII(vector<int>& citations)

{

int first = 0;

int last = citations.size() - 1;

int result = 0;

while (first <= last)

{

int middle = first + (last - first) / 2;

if (citations[middle] >= (int)citations.size() - middle)

{

result = (int)citations.size() - middle;

last = middle - 1;

}

else

{

first = middle + 1;

}

}

return result;

}

## 162. Find Peak Element

Medium

A peak element is an element that is greater than its neighbors.

Given an input array nums, where nums[i] ≠ nums[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that nums[-1] = nums[n] = -∞.

**Example 1:**

**Input:** **nums** = [1,2,3,1]

**Output:** 2

**Explanation:** 3 is a peak element and your function should return the index number 2.

**Example 2:**

**Input:** **nums** = [1,2,1,3,5,6,4]

**Output:** 1 or 5

**Explanation:** Your function can return either index number 1 where the peak element is 2,

  or index number 5 where the peak element is 6.

**Note:**

Your solution should be in logarithmic complexity.

### Analysis:

Assume we are at middle position, if the previous element is lower than current position and next position is higher than current position, we are at uphill, we should set the first as current + 1 position and discard other lower uphill points, if previous position is higher than current position, then we are at down hill, we can discard down hill position from current.

/// <summary>

/// Leet code #162. Find Peak Element

///

/// A peak element is an element that is greater than its neighbors.

///

/// Given an input array nums, where nums[i] ≠ nums[i+1], find a peak element

/// and return its index.

///

/// The array may contain multiple peaks, in that case return the index to

/// any one of the peaks is fine.

///

/// You may imagine that nums[-1] = nums[n] = INT\_MIN.

///

/// Example 1:

///

/// Input: nums = [1,2,3,1]

/// Output: 2

/// Explanation: 3 is a peak element and your function should return the

/// index number 2.

///

/// Example 2:

///

/// Input: nums = [1,2,1,3,5,6,4]

/// Output: 1 or 5

/// Explanation: Your function can return either index number 1 where the

/// peak element is 2, or index number 5 where the peak element is 6.

///

/// Note:

/// Your solution should be in logarithmic complexity.

/// </summary>

int LeetCodeBinarySearch::findPeakElement(vector<int>& nums)

{

int first = 0;

int last = nums.size() - 1;

int middle = last;

int result = last;

while (first < last)

{

middle = first + (last - first) / 2;

if (nums[middle] < nums[middle + 1])

{

result = middle + 1;

first = middle + 1;

}

else

{

result = middle;

last = middle;

}

}

return result;

}

## 33. Search in Rotated Sorted Array

Medium

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

**Example 1:**

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

**Output:** 4

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2], target = 3

**Output:** -1

### Analysis:

For a rotated array, you will always have one part is in order and another may or may not be in order. You can compare if the target number is within the range of the ordered part, if yes, discard the other part, is no, discard the ordered part.

/// <summary>

/// Leet code #33. Search in Rotated Sorted Array

///

/// Suppose an array sorted in ascending order is rotated at some pivot

/// unknown to you beforehand.

///

/// (i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

///

/// You are given a target value to search. If found in the array return

/// its index, otherwise return -1.

///

/// You may assume no duplicate exists in the array.

///

/// Your algorithm's runtime complexity must be in the order of O(log n).

///

/// Example 1:

///

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

/// Output: 4

///

/// Example 2:

///

/// Input: nums = [4,5,6,7,0,1,2], target = 3

/// Output: -1

/// </summary>

int LeetCodeBinarySearch::search(vector<int>& nums, int target)

{

int first = 0, last = nums.size() - 1;

while (first <= last)

{

size\_t middle = first + (last - first) / 2;

if (target == nums[middle])

{

return middle;

}

// because a range can be one item, so = is important.

else if (nums[first] <= nums[middle])

{

// if the target is not within ordered range

if ((target < nums[first]) || (target > nums[middle]))

{

first = middle + 1;

}

else // if not in another range.

{

last = middle - 1;

}

}

else

{

if ((target > nums[middle]) && (target <= nums[last]))

{

first = middle + 1;

}

else

{

last = middle - 1;

}

}

}

return -1;

}

## 81. Search in Rotated Sorted Array II

Medium

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., [0,0,1,2,2,5,6] might become [2,5,6,0,0,1,2]).

You are given a target value to search. If found in the array return true, otherwise return false.

**Example 1:**

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

**Output:** true

**Example 2:**

**Input:** nums = [2,5,6,0,0,1,2], target = 3

**Output:** false

**Follow up:**

* This is a follow up problem to [Search in Rotated Sorted Array](https://leetcode.com/problems/search-in-rotated-sorted-array/description/), where nums may contain duplicates.
* Would this affect the run-time complexity? How and why?

### Analysis:

For a rotated array, you will always have one part is in order and another may or may not be in order. You can compare if the target number is within the range of the ordered part, if yes, discard the other part, is no, discard the ordered part. Please notice that if you see an disordered part it is useless, because we cannot determine whether the target number is in there or not. If we can not find any ordered part, we have to discard first or last until we see the ordered part.

One optimization you can consider for the following implementation is that you use recursive call to resolve the problem with given first, last. This may speed up in the case where you have large number of duplicated number either in front or in back.

/// <summary>

/// Leetcode #81. Search in Rotated Sorted Array II

///

/// Suppose an array sorted in ascending order is rotated at some pivot

/// unknown to you beforehand.

///

/// (i.e., [0,0,1,2,2,5,6] might become [2,5,6,0,0,1,2]).

///

/// You are given a target value to search. If found in the array return true,

/// otherwise return false.

///

/// Example 1:

///

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

/// Output: true

///

/// Example 2:

///

/// Input: nums = [2,5,6,0,0,1,2], target = 3

/// Output: false

/// Follow up:

///

/// This is a follow up problem to Search in Rotated Sorted Array, where nums

/// may contain duplicates.

/// Would this affect the run-time complexity? How and why?

/// </summary>

bool LeetCodeBinarySearch::searchII(vector<int>& nums, int target)

{

int first = 0;

// because we need to compare with right end, so must point to a solid item.

int last = nums.size() - 1;

while (first <= last)

{

int middle = first + (last - first) / 2;

if (target == nums[middle])

{

return true;

}

else if (nums[first] < nums[middle])

{

// first check the target is within the ordered range, if not must be

// in a disordered range

if (target < nums[first] || target > nums[middle])

{

first = middle + 1;

}

else

{

last = middle - 1;

}

}

else if (nums[middle] < nums[last])

{

if (target < nums[middle] || target > nums[last])

{

last = middle - 1;

}

else

{

first = middle + 1;

}

}

// otherwise we move the first or the last until we find a disordered one

else if (nums[first] == nums[middle])

{

first++;

}

else if (nums[last] == nums[middle])

{

last--;

}

}

return false;

}

## 153. Find Minimum in Rotated Sorted Array

Medium

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e.,  [0,1,2,4,5,6,7] might become  [4,5,6,7,0,1,2]).

Find the minimum element.

You may assume no duplicate exists in the array.

**Example 1:**

**Input:** [3,4,5,1,2]

**Output:** 1

**Example 2:**

**Input:** [4,5,6,7,0,1,2]

**Output:** 0

### Analysis:

For a rotated array, you will always have one part is in order and another may or may not be in order. If you see the part which is disordered, then the lowest number must be there, if you do not see any disordered part, it must be in the lower half.

/// <summary>

/// Leet code #153. Find Minimum in Rotated Sorted Array

///

/// Suppose an array sorted in ascending order is rotated at some pivot

/// unknown to you beforehand.

///

/// (i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

///

/// Find the minimum element.

///

/// You may assume no duplicate exists in the array.

///

/// Example 1:

///

/// Input: [3,4,5,1,2]

/// Output: 1

///

/// Example 2:

///

/// Input: [4,5,6,7,0,1,2]

/// Output: 0

/// </summary>

int LeetCodeBinarySearch::findMin(vector<int>& nums)

{

int first = 0;

int last = nums.size() - 1;

int result = INT\_MAX;

while (first <= last)

{

int middle = first + (last - first) / 2;

// If first is greater than last, then the minimum item must be

// within the range

if (nums[first] > nums[middle])

{

result = min(result, nums[middle]);

last = middle - 1;

}

else if (nums[middle] > nums[last])

{

result = min(result, nums[last]);

first = middle + 1;

}

else

{

result = min(result, nums[first]);

break;

}

}

return result;

}

## 74. Search a 2D Matrix

Medium

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

* Integers in each row are sorted from left to right.
* The first integer of each row is greater than the last integer of the previous row.

**Example 1:**

Calendar

Description automatically generated

**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

**Output:** true

**Example 2:**

Calendar

Description automatically generated

**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

**Output:** false

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 100
* -104 <= matrix[i][j], target <= 104

### Analysis:

We start the top right corner, if the matrix value is greater than the target, we know the target can be in the current row but in left column, if the matrix value is less than target, we know the target can be in the rows below.

/// <summary>

/// Leet code #74. Search a 2D Matrix

///

/// Write an efficient algorithm that searches for a value in an m x n matrix.

/// This matrix has the following properties:

/// Integers in each row are sorted from left to right.

/// The first integer of each row is greater than the last integer of the

/// previous row.

/// For example,

/// Consider the following matrix:

/// [

/// [1, 3, 5, 7],

/// [10, 11, 16, 20],

/// [23, 30, 34, 50]

/// ]

/// Given target = 3, return true.

/// </summary>

bool LeetCodeBinarySearch::searchMatrix(vector<vector<int>>& matrix, int target)

{

int n = matrix.size();

int m = matrix[0].size();

int r = 0;

int c = matrix[0].size() - 1;

while (c >= 0 && r < n)

{

if (matrix[r][c] == target) return true;

else if (matrix[r][c] > target)

{

c--;

}

else

{

r++;

}

}

return false;

}

# Advance Problems

## 154. Find Minimum in Rotated Sorted Array II

Hard

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e.,  [0,1,2,4,5,6,7] might become  [4,5,6,7,0,1,2]).

Find the minimum element.

The array may contain duplicates.

**Example 1:**

**Input:** [1,3,5]

**Output:** 1

**Example 2:**

**Input:** [2,2,2,0,1]

**Output:** 0

**Note:**

* This is a follow up problem to [Find Minimum in Rotated Sorted Array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/description/).
* Would allow duplicates affect the run-time complexity? How and why?

### Analysis:

For a rotated array, the minimum number must be either in the disordered part or the first half of the ordered part. If the second half is either disordered or ordered, we know where to find. The only problem is that the fist, middle and last are same, in this case we keep on discard last one, since our job is not to find all the minimum values, until we have the clue.

/// <summary>

/// Leet code #154. Find Minimum in Rotated Sorted Array II

///

/// Suppose an array sorted in ascending order is rotated at some pivot

/// unknown to you beforehand.

///

/// (i.e., [0,1,2,4,5,6,7] might become [4,5,6,7,0,1,2]).

///

/// Find the minimum element.

///

/// The array may contain duplicates.

///

/// Example 1:

///

/// Input: [1,3,5]

/// Output: 1

///

/// Example 2:

///

/// Input: [2,2,2,0,1]

/// Output: 0

/// Note:

///

/// This is a follow up problem to Find Minimum in Rotated Sorted Array.

/// Would allow duplicates affect the run-time complexity? How and why?

/// </summary>

int LeetCodeBinarySearch::findMinII(vector<int>& nums)

{

if (nums.size() == 0) return 0;

int first = 0;

int last = nums.size() - 1;

while (first < last)

{

int middle = first + (last - first) / 2;

// The first part disordered, the minimum is in first part

if (nums[first] > nums[middle])

{

last = middle;

}

// The second part is disordered, the minimum is in the second part.

else if (nums[middle] > nums[last])

{

first = middle + 1;

}

// The second part is in order, the minimum is in the first part.

// regardless first part is in order or not.

else if (nums[middle] < nums[last])

{

last = middle;

}

// We can not determine the second part is in order or not, discard last.

else if (nums[middle] == nums[last])

{

last--;

}

}

return nums[first];

}

## 1095. Find in Mountain Array

Hard

*(This problem is an****interactive problem****.)*

You may recall that an array A is a *mountain array* if and only if:

* A.length >= 3
* There exists some i with 0 < i < A.length - 1 such that:
  + A[0] < A[1] < ... A[i-1] < A[i]
  + A[i] > A[i+1] > ... > A[A.length - 1]

Given a mountain array mountainArr, return the **minimum** index such that mountainArr.get(index) == target.  If such an index doesn't exist, return -1.

**You can't access the mountain array directly.**  You may only access the array using a MountainArray interface:

* MountainArray.get(k) returns the element of the array at index k (0-indexed).
* MountainArray.length() returns the length of the array.

Submissions making more than 100 calls to MountainArray.get will be judged *Wrong Answer*.  Also, any solutions that attempt to circumvent the judge will result in disqualification.

**Example 1:**

**Input:** array = [1,2,3,4,5,3,1], target = 3

**Output:** 2

**Explanation:** 3 exists in the array, at index=2 and index=5. Return the minimum index, which is 2.

**Example 2:**

**Input:** array = [0,1,2,4,2,1], target = 3

**Output:** -1

**Explanation:** 3 does not exist in the array, so we return -1.

**Constraints:**

1. 3 <= mountain\_arr.length() <= 10000
2. 0 <= target <= 10^9
3. 0 <= mountain\_arr.get(index) <= 10^9

### Analysis:

When you pick a middle point, you first check if this is a uphill or downhill, if this is a uphill and the target is greater than the middle point or if it is a downhill and value is great than the middle point you can discard the first half or second half. If the value is within the uphill range you search in uphill first if not found then search in the second half, or if the value is within the downhill, you search the downhill first, if not found then search the first half.

Please notice that the original problem is represented as a design problem which is not aligned with the latest signature.

/// <summary>

/// Leet code #1095. Find in Mountain Array

///

/// You may recall that an array A is a mountain array if and only if:

/// A.length >= 3

/// There exists some i with 0 < i < A.length - 1 such that:

/// A[0] < A[1] < ... A[i-1] < A[i]

/// A[i] > A[i+1] > ... > A[A.length - 1]

/// Given a mountain array mountainArr, return the minimum index such that

/// mountainArr.get(index) == target. If such an index doesn't exist,

/// return -1.

///

/// You can't access the mountain array directly. You may only access the

/// array using a MountainArray interface:

///

/// MountainArray.get(k) returns the element of the array at index k

/// (0-indexed).

/// MountainArray.length() returns the length of the array.

/// Submissions making more than 100 calls to MountainArray.get will be

/// judged Wrong Answer. Also, any solutions that attempt to circumvent

/// the judge will result in disqualification.

///

/// Example 1:

///

/// Input: array = [1,2,3,4,5,3,1], target = 3

/// Output: 2

/// Explanation: 3 exists in the array, at index=2 and index=5. Return

/// the minimum index, which is 2.

///

/// Example 2:

///

/// Input: array = [0,1,2,4,2,1], target = 3

/// Output: -1

/// Explanation: 3 does not exist in the array, so we return -1.

///

/// Constraints:

/// 1. 3 <= mountain\_arr.length() <= 10000

/// 2. 0 <= target <= 10^9

/// 3. 0 <= mountain\_arr.get(index) <= 10^9

/// </summary>

class MountainArray

{

private:

vector<int> m\_array;

int length()

{

return m\_array.size();

}

int get(int k)

{

return m\_array[k];

}

public:

MountainArray()

{

}

MountainArray(vector<int>& a)

{

m\_array = a;

}

int findInMountainArray(int target, MountainArray& mountainArr, int first, int last)

{

int result = -1;

if (first == last)

{

if (mountainArr.get(first) == target) result = first;

}

else if (first + 1 == last)

{

if (mountainArr.get(first) == target) result = first;

else if (mountainArr.get(last) == target) result = last;

}

else

{

int mid = first + (last - first) / 2;

if (mountainArr.get(mid - 1) < mountainArr.get(mid) &&

mountainArr.get(mid) < mountainArr.get(mid + 1))

{

if (target <= mountainArr.get(mid))

{

result = findInMountainArray(target, mountainArr, first, mid);

}

if (result == -1)

{

result = findInMountainArray(target, mountainArr, mid + 1, last);

}

}

else if (mountainArr.get(mid - 1) > mountainArr.get(mid) &&

mountainArr.get(mid) > mountainArr.get(mid + 1))

{

result = findInMountainArray(target, mountainArr, first, mid);

if (result == -1)

{

if (target <= mountainArr.get(mid + 1))

{

result = findInMountainArray(target, mountainArr, mid + 1, last);

}

}

}

else

{

if (target <= mountainArr.get(mid))

{

result = findInMountainArray(target, mountainArr, first, mid);

}

if (result == -1)

{

if (target <= mountainArr.get(mid + 1))

{

result = findInMountainArray(target, mountainArr, mid + 1, last);

}

}

}

}

return result;

}

int findInMountainArray(int target, MountainArray& mountainArr)

{

int first = 0;

int last = mountainArr.length() - 1;

int result = findInMountainArray(target, mountainArr, first, last);

return result;

}

};