<https://leetcode.com/problems/find-in-mountain-array/>

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

You may recall that an array

arr is a **mountain array** if and only if:

arr.length >= 3

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

arr[0] < arr[1] < ... < arr[i - 1] < arr[i]

arr[i] > arr[i + 1] > ... > arr[arr.length - 1]

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

**You cannot 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:**

3 <= mountain\_arr.length() <= 10^4

0 <= target <= 10^9

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

**Attempt 1: 2022-09-21**

**Wrong solution (Similar as L162, even no specific target, but still able to use template similar to Find Target Occurrence, just change the condition as nums[mid] == target to nums[mid] > nums[mid - 1] && nums[mid] > nums[mid + 1], but error out "You made too many calls to MountainArray.get()."), because we call too many times of mountainArr.get() during find peak element**

// Since given array length >= 3, no need check corner case when length = 1 or 2

// Find peak element (Refer to L162)

// Set 'lo' = 1 and 'hi' = array length - 2, to avoid index out of boundary

// when calling mountainArr.get(mid - 1) and mountainArr.get(mid + 1)

int lo = 1;

int hi = mountainArr.length() - 2;

int peak = 0;

while(lo <= hi) {

    int mid = lo + (hi - lo) / 2;

    if(mountainArr.get(mid) > mountainArr.get(mid - 1) && mountainArr.get(mid) > mountainArr.get(mid + 1)) {

        peak = mid;

        break;

    } else if(mountainArr.get(mid) < mountainArr.get(mid - 1)) {

        hi = mid - 1;

    } else {

        lo = mid + 1;

    }

}

**Failure as 78/79, last test case not able to pass**

**You made too many calls to MountainArray.get().**

/\*\*

\* // This is MountainArray's API interface.

\* // You should not implement it, or speculate about its implementation

\* interface MountainArray {

\*    public int get(int index) {}

\*    public int length() {}

\* }

\*/

class Solution {

    public int findInMountainArray(int target, MountainArray mountainArr) {

        // Since given array length >= 3, no need check corner case when length = 1 or 2

        // Find peak element (Refer to L162)

        // Set 'lo' = 1 and 'hi' = array length - 2, to avoid index out of boundary

        // when calling mountainArr.get(mid - 1) and mountainArr.get(mid + 1)

        int lo = 1;

        int hi = mountainArr.length() - 2;

        int peak = 0;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(mountainArr.get(mid) > mountainArr.get(mid - 1) && mountainArr.get(mid) > mountainArr.get(mid + 1)) {

                peak = mid;

                break;

            } else if(mountainArr.get(mid) < mountainArr.get(mid - 1)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // Find target on left monotonic increasing subarray (Refer to L704)

        // If not able to find target and return directly, then move on to

        // right monotonic decreasing subarray, no need to return -1 now

        lo = 0;

        hi = peak;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

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

                return mid;

            } else if(mountainArr.get(mid) > target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // Find target on right monotonic decreasing subarray

        lo = peak;

        hi = mountainArr.length() - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

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

                return mid;

            } else if(mountainArr.get(mid) > target) {

                lo = mid + 1;

            } else {

                hi = mid - 1;

            }

        }

        // If not able to find in either side return -1

        return -1;

    }

}

**Require a new way to find peak element with less call of mountainArr.get() method**

**Two Times Binary Search (360min, too long to figure out how to find peak element with less call of mountainArr.get() method)**

**Template refer to**

<https://leetcode.com/problems/find-peak-element/discuss/788474/General-Binary-Search-Thought-Process-%3A-4-Templates>

class Solution:

    def findPeakElement(self, nums: List[int]) -> int:

        left =0

        right = len(nums)-1

        while left < right:

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

            if nums[mid] > nums[mid+1]: # True Condition # Dec function # go left # Find First True i.e first elem where this condition will be True

                right = mid # include mid # mid is potential solution

            else:

                left = mid +1

        return left

**Summary of 2 most frequently used binary search templates**

<https://leetcode.com/problems/find-in-mountain-array/discuss/317607/JavaC++Python-Triple-Binary-Search/294169>

One is return index during the search:

while lo <= hi:

  mid = (lo+hi)/2

  if nums[mid] == target:

    return mid

  if nums[mid] > target:

    hi = mid-1

  else:

    lo = mid+1

return -1

Another more frequently used binary search template is for searching lowest element satisfy function(i) == True (the array should satisfy function(x) == False for 0 to i-1, and function(x) == True for i to n-1, and it is up to the question to define the function, like in the find peak element problem, function(x) can be nums[x] < nums[x+1] ), there are 2 ways to write it:

while lo <= hi:

  mid = (lo+hi)/2

  if function(mid):

    hi = mid-1

  else:

    lo = mid+1

return lo

or

while lo < hi:

  mid = (lo+hi)/2

  if function(mid):

    hi = mid

  else:

    lo = mid+1

return lo

No matter which one you use, just be careful about updating the hi and lo, which could easily lead to infinite loop. Some binary question is searching a floating number and normally the question will give you a precision, in which case you don't need to worry too much about the infinite loop but your while condition will become something like "while lo+1e-7<hi"

**Implementation based on above second pattern in L1095**

int lo = 0;

int hi = mountainArr.length() - 1;

// Find the peak index

while (lo < hi) {

    int mid = lo + (hi - lo) / 2;

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

        // include mid since mid is potential solution of peak

        hi = mid;

    } else {

        lo = mid + 1;

    }

}

**Correct Solution**

<https://leetcode.com/problems/find-in-mountain-array/discuss/378052/Binary-Search-Thinking-Process>

If we divide A in half, one subarray is in the increasing / decreasing order, the other is a mountain array that has the peak index. That helps abandon one subarray. Then in the mountain subarray, we keep dividing...Until the searching space is exhausted, we can find the peak index. The loop invariant is the peak index is in [left, right].

The peak index divides A in half, one subarray is in the increasing order, the other is in the decreasing order. We aim to find the index of target in A if target exists. We can do Binary Search on two subarrays separately. The loop invariant is the target index is in [left, right].

/\*\*

\* // This is MountainArray's API interface.

\* // You should not implement it, or speculate about its implementation

\* interface MountainArray {

\*    public int get(int index) {}

\*    public int length() {}

\* }

\*/

class Solution {

    public int findInMountainArray(int target, MountainArray mountainArr) {

        int lo = 0;

        int hi = mountainArr.length() - 1;

        // Find the peak index

        while (lo < hi) {

            int mid = lo + (hi - lo) / 2;

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

                hi = mid;

            } else {

                lo = mid + 1;

            }

        }

        int peak = lo;

        // Find target on left monotonic increasing subarray (Refer to L704)

        // If not able to find target and return directly, then move on to

        // right monotonic decreasing subarray, no need to return -1 now

        lo = 0;

        hi = peak;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

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

                return mid;

            } else if(mountainArr.get(mid) > target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // Find target on right monotonic decreasing subarray

        lo = peak;

        hi = mountainArr.length() - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

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

                return mid;

            } else if(mountainArr.get(mid) > target) {

                lo = mid + 1;

            } else {

                hi = mid - 1;

            }

        }

        // If not able to find in either side return -1

        return -1;

    }

}

Space Complexity: O(1)

Time Complexity: O(logn)

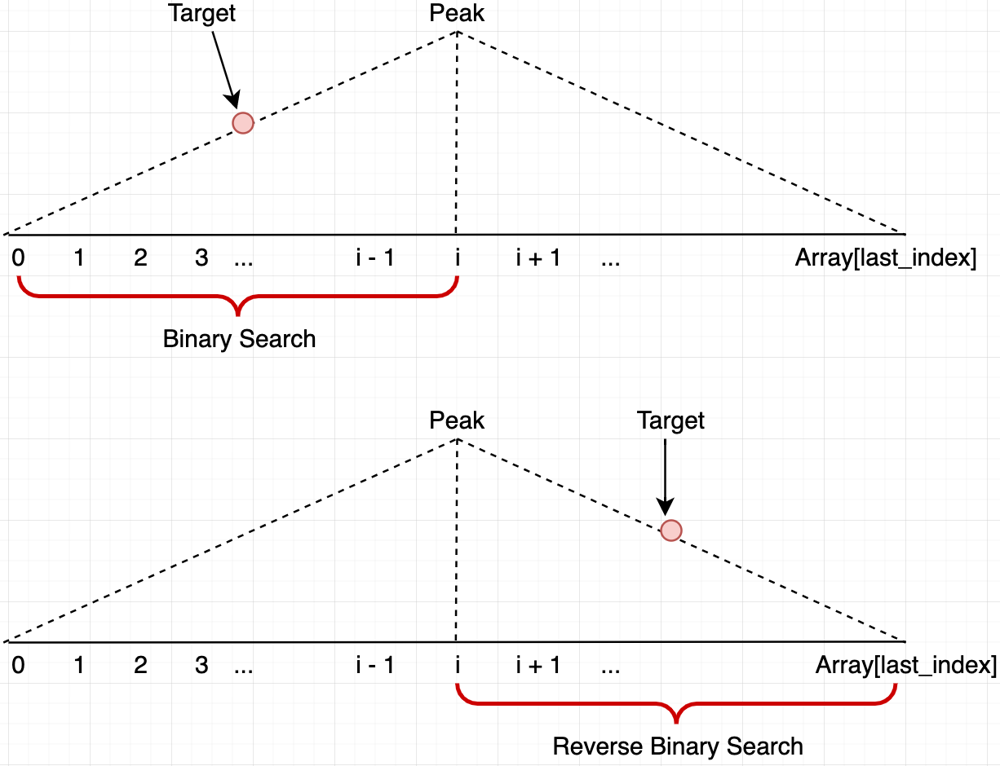
**What's the logic behind triple binary search ?**

**Refer to**

<https://leetcode.com/problems/find-in-mountain-array/solutions/317607/java-c-python-triple-binary-search/>

**Intuition**

Triple Binary Search, Triple Happiness.



Binary find peak in the mountain.[852. Peak Index in a Mountain Array](https://leetcode.com/problems/peak-index-in-a-mountain-array/discuss/139848/C%2B%2BJavaPython-Better-than-Binary-Search/294082)

Binary find the target in strict increasing array

Binary find the target in strict decreasing array

**What's the difference between above 78/79 too many calls solution and correction solution ?**

**Refer to chatGPT**

The difference between the two solutions mainly lies in how they find the peak element of the mountain array, which significantly affects the number of get() calls made to the MountainArray interface. The second solution is more efficient because it avoids unnecessary checks and redundant calls to mountainArr.get().

**Differences Between the Two Solutions**

**1. Finding the Peak Element:**

First Solution:

Starts the peak search with lo = 1 and hi = mountainArr.length() - 2.

Uses a condition that checks mountainArr.get(mid) > mountainArr.get(mid - 1) and mountainArr.get(mid) > mountainArr.get(mid + 1) to find the peak.

This condition results in multiple calls to mountainArr.get() for both mid - 1 and mid + 1 in each iteration.

Second Solution:

Starts with lo = 0 and hi = mountainArr.length() - 1.

Uses a simpler condition that compares mountainArr.get(mid) with mountainArr.get(mid + 1) to decide the direction of the search.

Only makes one call to mountainArr.get() for mid and one for mid + 1 in each iteration.

The second solution efficiently narrows down the search to find the peak with fewer get() calls.

**2. Number of get() Calls:**

The first solution calls mountainArr.get(mid), mountainArr.get(mid - 1), and mountainArr.get(mid + 1) multiple times in the while loop. This leads to many unnecessary calls, especially if the array length is large.

The second solution minimizes these calls by only making the necessary comparisons (mid and mid + 1), ensuring it stays within the limits imposed by the problem constraints.

**Why the Second Solution is More Efficient**

Reduced **get() Calls**: The second solution only makes necessary comparisons:

In the peak finding phase, it only makes two get() calls per iteration (mid and mid + 1).

This reduces the overall number of calls compared to the first solution, which can make up to three get() calls in each iteration of the loop.

Binary Search Method: The way binary search is utilized:

The second solution effectively halves the search space by moving either lo or hi based on the comparison between mid and mid + 1, which directly leads to finding the peak.

In contrast, the first solution sometimes performs unnecessary comparisons and has more conditional checks, making it less efficient.

Efficient Peak Detection: The second solution's approach to finding the peak is based on the property that if mountainArr.get(mid) > mountainArr.get(mid + 1), then the peak must be on the left side (including mid). This leads to fewer comparisons and quicker peak detection.

**Conclusion**

The second solution is more efficient due to its simpler peak-finding logic and fewer calls to mountainArr.get(). It optimally leverages binary search principles to reduce unnecessary operations.

The first solution, while logically correct, makes too many get() calls, which results in exceeding the allowed number of calls in some test cases, leading to the "too many calls" error.

In problems where the number of calls to a provided API is restricted, optimizing every step of the solution (like peak finding) is crucial to ensure that the solution remains within the limits. The second solution's efficiency allows it to pass all test cases without exceeding the call limits.

**Why in above second solution (correct solution) we set 'int peak = lo' not 'int peak = hi' ?**

**Refer to chatGPT**

In the second solution (the correct one), we use int peak = lo; instead of int peak = hi; because of the way the binary search loop is structured. Let me explain why lo and hi are equivalent at the end of this loop, and why lo is chosen:

**Understanding the Loop**

Loop Condition (**while (lo < hi))**:

The loop continues as long as lo < hi. This condition ensures that we are narrowing down the search to a single point.

**The loop breaks when lo is equal to hi, which means we have found the peak.**

Mid Calculation and Movement:

**We calculate mid as lo + (hi - lo) / 2, which gives the middle index between lo and hi.**

**If mountainArr.get(mid) > mountainArr.get(mid + 1), it implies we are in the descending part of the mountain (or at the peak), so the peak must be at mid or before mid. We then set hi = mid to continue searching in the left part, including mid.**

**If mountainArr.get(mid) <= mountainArr.get(mid + 1), it implies we are still in the ascending part of the mountain, so we set lo = mid + 1 to continue searching to the right.**

**Why Use int peak = lo; Instead of int peak = hi;**

Termination Condition: The loop terminates when lo equals hi. This means we have narrowed down our search to a single element, which is the peak of the mountain. At this point, both lo and hi point to the same index, and this index is the peak.

**lo and hi are the Same**: **Since the loop only breaks when lo == hi, assigning either lo or hi to the peak variable would give the same result. Using int peak = lo; is just a convention; we could equivalently use int peak = hi;.**

**Why We Use lo**

Readability and Consistency: In binary search algorithms, it's common to use lo as the final value when the loop terminates because lo represents the lower bound of our search range. This makes it intuitive to use lo to assign the peak index after narrowing down the range to a single point.

Pattern in Binary Search: Often in binary search problems, lo is used as the final index when the search condition is while (lo < hi). It becomes a natural choice for clarity and consistency in understanding.

**Conclusion**

Using int peak = lo; in the final assignment is more of a coding convention, and it's consistent with how we track the progress of binary search. Both lo and hi point to the same value when the loop exits, so it doesn't matter which one we choose to assign the peak index. This approach maintains readability and follows common patterns in binary search implementation.

**Refer to**

[L162.Find Peak Element](note://E6EA30EA7B614C149A1F4FE68B8116F4)

[L704.Binary Search](note://484532A5D2CB4B98A28218674438D5D8)

[L852.P12.7.Peak Index in a Mountain Array (Ref.L162,L1095)](note://109C71BED3454E10AB2F5D1347A8269D)