<https://leetcode.com/problems/find-peak-element/>

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

Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

**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: 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.

**Constraints:**

1 <= nums.length <= 1000

-2^31 <= nums[i] <= 2^31 - 1

nums[i] != nums[i + 1] for all valid i.

**Attempt 1: 2022-09-18**

**Solution 1: Binary Search solution (10min, 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])**

class Solution {

    public int findPeakElement(int[] nums) {

        int len = nums.length;

        if(len == 1) {

            return 0;

        }

        if(nums[0] > nums[1]) {

            return 0;

        }

        if(nums[len - 1] > nums[len - 2]) {

            return len - 1;

        }

        int lo = 1;

        int hi = len - 2;

        while(lo <= hi) {

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

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

                return mid;

            } else if(nums[mid] < nums[mid - 1]) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return -1;

    }

}

Space Complexity: O(1)

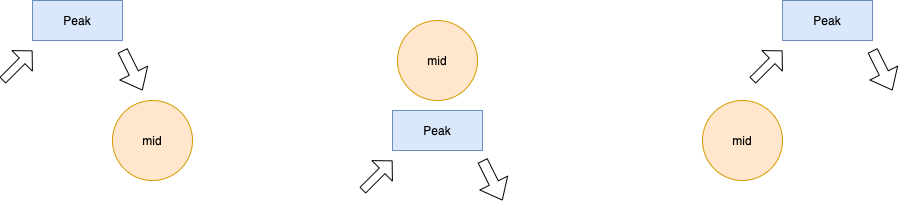
Time Complexity: O(logn)

**Binary Search solution refer to**

<https://leetcode.com/problems/find-peak-element/discuss/1290642/intuition-behind-conditions-complete-explanation-diagram-binary-search>

Given an array, we need to find the peak element. As the sub portions of the array are increasing/decreasing ( only then we would be able to find peak ), there are sub portions of array which are sorted, so we could use binary search to get this problem done. But exactly how ? This is an interesting part.

For a mid element, there could be three possible cases :



Case 1 : mid lies on the right of our result peak ( Observation : Our peak element search space is left side )

Case 2 : mid is equal to the peak element ( Observation : mid element is greater than its neighbors )

Case 3 : mid lies on the left. ( Observation : Our peak element search space is right side )

so, the code becomes

int start = 0;

int end = n-1;

while(start <= end) {

int mid = start + (end - start)/2;

if(nums[mid] > nums[mid-1] && nums[mid] > nums[mid+1]) return mid;  // if mid == peak ( case 2 )

else if(nums[mid] < nums[mid-1]) end = mid - 1; // downward slope and search space left side ( case 1)

else if(nums[mid] < nums[mid+1]) start = mid + 1; // upward slope and search space right side ( case 3 )

}

Some base cases :

The array could be strictly increasing or strictly decreasing and as we have to return any of the possible peaks, so we could add a condition to check whether the 1st element/last element could be the peak ). This point is also supported by the fact that, we are looking for mid-1/ mid+1 ( and these indices are compromised for 0th index / n-1th index respectively.

So, our complete code becomes

if(nums.length == 1) return 0; // single element

int n = nums.length; // check if 0th/n-1th index is the peak element

if(nums[0] > nums[1]) return 0;

if(nums[n-1] > nums[n-2]) return n-1; // search in the remaining array

int start = 1;

int end = n-2;

while(start <= end) {

    int mid = start + (end - start)/2;

    if(nums[mid] > nums[mid-1] && nums[mid] > nums[mid+1]) return mid;

    else if(nums[mid] < nums[mid-1]) end = mid - 1;

    else if(nums[mid] < nums[mid+1]) start = mid + 1;

}

return -1; // dummy return statement

**Solution 2: Linear Scan solution (10 min)**

class Solution {

    public int findPeakElement(int[] nums) {

        for(int i = 0; i < nums.length - 1; i++) {

            if(nums[i] > nums[i + 1]) {

                return i;

            }

        }

        return nums.length - 1;

    }

}

Space Complexity: O(1)

Time Complexity: O(n)

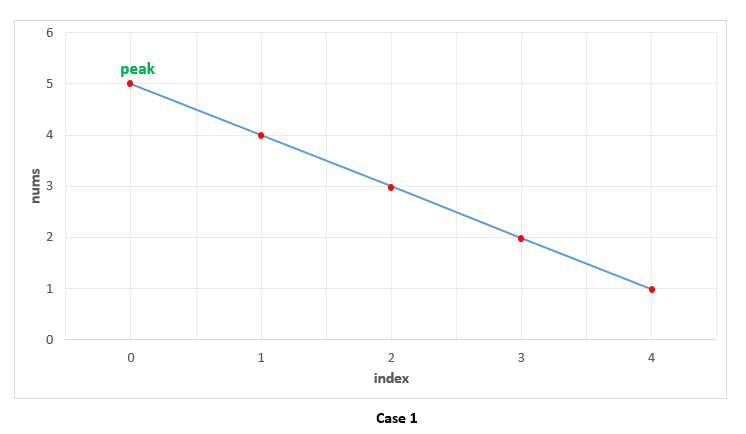
**Linear Scan solution refer to**

<https://leetcode.com/problems/find-peak-element/solution/>

**Approach 1: Linear Scan**

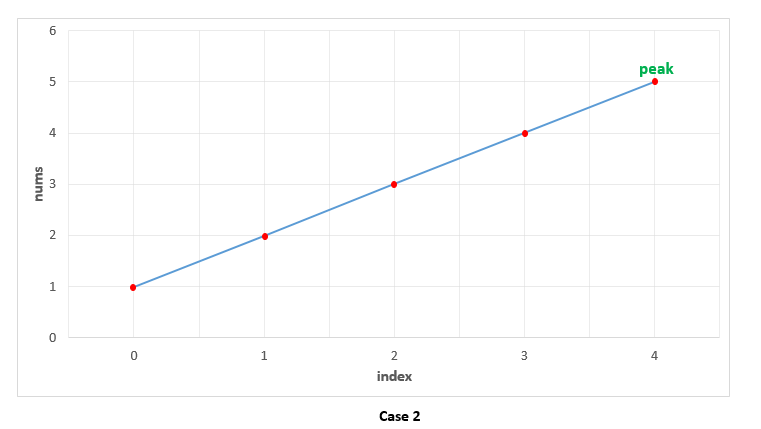
In this approach, we make use of the fact that two consecutive numbers nums[j]and nums[j + 1]are never equal. Thus, we can traverse over the numsarray starting from the beginning. Whenever, we find a number nums[i], we only need to check if it is larger than the next number nums[i+1]for determining if nums[i]is the peak element. The reasoning behind this can be understood by taking the following three cases which cover every case into which any problem can be divided.

Case 1. All the numbers appear in a descending order. In this case, the first element corresponds to the peak element. We start off by checking if the current element is larger than the next one. The first element satisfies this criteria, and is hence identified as the peak correctly. In this case, we didn't reach a point where we needed to compare nums[i]with nums[i-1] also, to determine if it is the peak element or not.

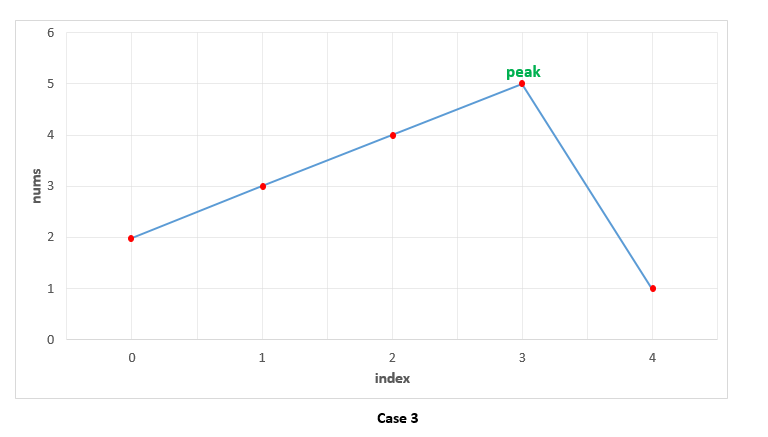


Case 2. All the elements appear in ascending order. In this case, we keep on comparing nums[i] with  nums[i+1]to determine if nums[i]

is the peak element or not. None of the elements satisfy this criteria, indicating that we are currently on a rising slope and not on a peak. Thus, at the end, we need to return the last element as the peak element, which turns out to be correct. In this case also, we need not compare nums[i]with nums[i-1], since being on the rising slope is a sufficient condition to ensure that nums[i] isn't the peak element.



Case 3. The peak appears somewhere in the middle. In this case, when we are traversing on the rising edge, as in Case 2, none of the elements will satisfy nums[i] > nums[i + 1]. We need not compare nums[i]with nums[i-1]on the rising slope as discussed above. When we finally reach the peak element, the condition nums[i] > nums[i + 1]is satisfied. We again, need not compare nums[i]with nums[i-1]. This is because, we could reach nums[i]as the current element only when the check nums[i] > nums[i + 1]failed for the previous(i-1)^{th} element, indicating that nums[i-1] < nums[i]. Thus, we are able to identify the peak element correctly in this case as well.



public class Solution {

    public int findPeakElement(int[] nums) {

        for (int i = 0; i < nums.length - 1; i++) {

            if (nums[i] > nums[i + 1])

                return i;

        }

        return nums.length - 1;

    }

}

**Complexity Analysis**

Time complexity : O(n). We traverse the numsarray of size nonce only.

Space complexity : O(1). Constant extra space is used.

**Refer to**

[L1095.P12.8.Find in Mountain Array (Ref.L162,L704)](note://F02F0200C53D40F48B3CE54558FB47E6)