<https://leetcode.com/problems/valid-mountain-array/description/>

Given an array of integers arr, return true if and only if it is a valid mountain array.

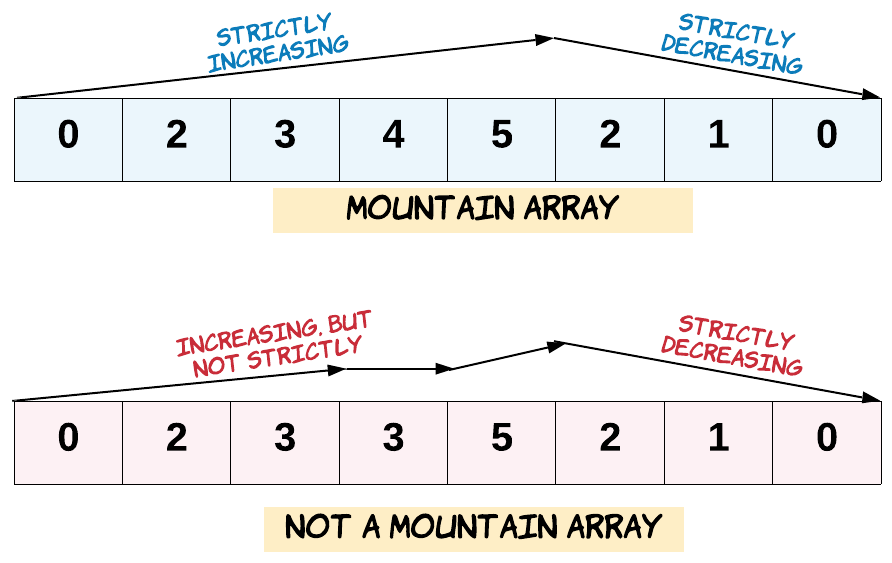
Recall that arr is a mountain array if and only if:

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]



**Example 1:**

**Input:** arr = [2,1]

**Output:** false

**Example 2:**

**Input:** arr = [3,5,5]

**Output:** false

**Example 3:**

**Input:** arr = [0,3,2,1]

**Output:** true

**Constraints:**

1 <= arr.length <= 10^4

0 <= arr[i] <= 10^4

**Attempt 1: 2024-08-31**

**Solution 1: One Pass (10min)**

class Solution {

    public boolean validMountainArray(int[] arr) {

        int n = arr.length;

        int i = 0;

        // Walk up the mountain

        while(i < n - 1 && arr[i] < arr[i + 1]) {

            i++;

        }

        // Peak can't be the first or the last element

        if(i == 0 || i == n - 1) {

            return false;

        }

        // Walk down the mountain

        while(i < n - 1 && arr[i] > arr[i + 1]) {

            i++;

        }

        // If we have reached the end of the array, it's a valid mountain

        return i == n - 1;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to chatGPT**

**Java Solution**

The problem can be solved efficiently by checking the conditions using a linear scan. We'll use two passes to find the peak and verify the ascending and descending order around the peak.

Here's a Java solution:

public class Solution {

public boolean validMountainArray(int[] arr) {

if (arr.length < 3) {

return false; // Not enough elements to form a mountain

}

int i = 0;

int n = arr.length;

// Walk up the mountain

while (i + 1 < n && arr[i] < arr[i + 1]) {

i++;

}

// Peak can't be the first or the last element

if (i == 0 || i == n - 1) {

return false;

}

// Walk down the mountain

while (i + 1 < n && arr[i] > arr[i + 1]) {

i++;

}

// If we have reached the end of the array, it's a valid mountain

return i == n - 1;

}

}

**Explanation:**

Initial Check: If the length of arr is less than 3, it's immediately invalid because it can't form a mountain.

Walking Up the Mountain:

We start with i = 0 and increment i as long as the current element is less than the next (arr[i] < arr[i + 1]).

This loop will stop when we reach the peak (the first point where the array stops increasing).

Checking the Peak:

If the peak is at the first position (i == 0) or the last position (i == n - 1), it's not a valid mountain (it needs to have an increase and a decrease).

Walking Down the Mountain:

We continue incrementing i while the current element is greater than the next (arr[i] > arr[i + 1]).

This loop will stop when the array stops decreasing.

Validation:

If i reaches the end of the array (i == n - 1), the array is a valid mountain.

Otherwise, it's not.

**Complexity Analysis:**

Time Complexity: O(n), where n is the length of the array. We are making a single pass over the array in both the ascending and descending parts.

Space Complexity: O(1), as we are using a constant amount of extra space.

This solution efficiently checks whether the given array is a valid mountain by simulating a walk up and down the mountain using simple pointer manipulation.

**Solution 2: Two Pointers (10min)**

class Solution {

    public boolean validMountainArray(int[] arr) {

        int n = arr.length;

        if(n < 3) {

            return false;

        }

        int i = 0;

        int j = n - 1;

        // Walk up the mountain from left

        while(i < n - 1 && arr[i] < arr[i + 1]) {

            i++;

        }

        // Test out by [0,1,2,3,4,5,6,7,8,9] since

        // its not a mountain but just monotonic

        // increasing sequences

        if(i == n - 1) {

            return false;

        }

        // Walk down the mountain from right

        while(j > 1 && arr[j - 1] > arr[j]) {

            j--;

        }

        // If we can meet at the peak of the mountain

        return i == j;

    }

}

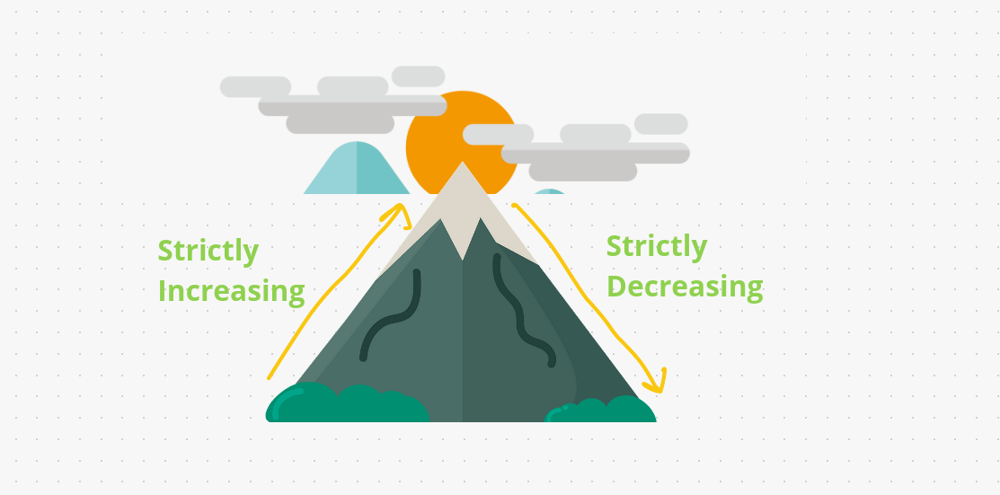
Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/valid-mountain-array/solutions/1717377/java-c-python-easy-to-go-through-solution-explanation/>

**So, how's the mountain we want, something like this:**



Similarly, we have given some **arrays value** and we have to check by that array Is our **mountain can possible or not.**

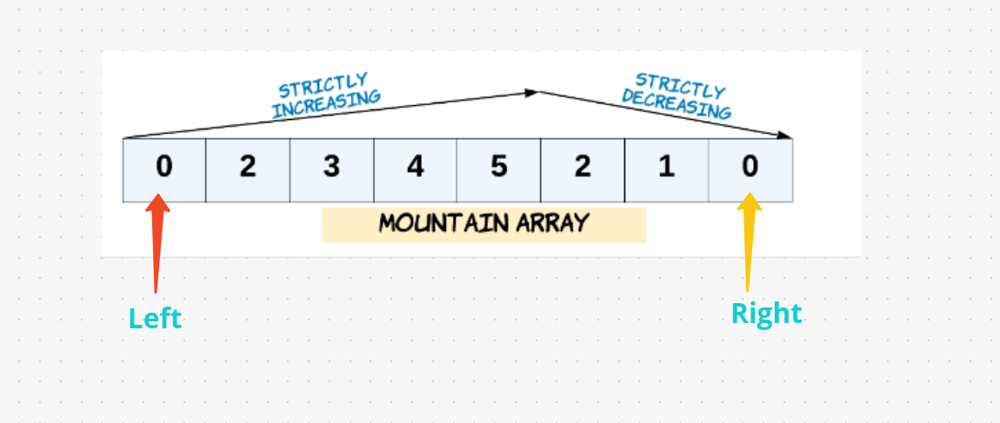
There are some conditons as well to form a mountain:

The **array size** has to be **> 3**

It has to be **strictly increasing** like **[0, 3, 5]** and the values has to be different not same

Similarly it has to be **strictly decreasing** like **[4 , 2, 1]** amd the values has to be different not same

So, how we can check it. For that one we will use the help of **2 pointers** one will start from **left** & another will start from **right**. If left and right **meets on same index value** then we **return true**, because it's a **stricly increasing and decreasing mountain**.



class Solution {

public boolean validMountainArray(int[] arr) {

if(arr.length < 3) return false;

int l = 0;

int r = arr.length - 1;

while(l + 1 < arr.length - 1 && arr[l] < arr[l + 1]) l++;

while(r - 1 > 0 && arr[r] < arr[r - 1]) r--;

return l == r;

}

}

ANALYSIS :-

Time Complexity : O(N) as we are traversing the array only once.

Space Complexity : O(1) as we are not using any extra space

**Refer to**

[L1671.Minimum Number of Removals to Make Mountain Array (Ref.L300)](note://A717168457D646DF8CE8A9054DD61C12)