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

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

arr.length >= 3

There exists some index i (0-indexed) 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 an integer array arr, return the length of the longest subarray, which is a mountain. Return 0 if there is no mountain subarray.

**Example 1:**

Input: arr = [2,1,4,7,3,2,5]

Output: 5

Explanation: The largest mountain is [1,4,7,3,2] which has length 5.

**Example 2:**

Input: arr = [2,2,2]

Output: 0

Explanation: There is no mountain.

**Constraints:**

1 <= arr.length <= 10^4

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

**Follow up:**

Can you solve it using only one pass?

Can you solve it in O(1) space?

**Attempt 1: 2023-02-22**

**Solution 1: Two passes from left to right/right to left with O(N) space (30 min)**

**Style 1: No need fill in up & down with 1, just up[i] + down[i] + 1**

class Solution {

public int longestMountain(int[] arr) {

int count = 0;

int len = arr.length;

int[] up = new int[len];

int[] down = new int[len];

for(int i = 0; i < len; i++) {

if(i > 0 && arr[i] > arr[i - 1]) {

up[i] = up[i - 1] + 1;

}

}

for(int i = len - 2; i >= 0; i--) {

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

down[i] = down[i + 1] + 1;

}

if(up[i] > 0 && down[i] > 0) {

count = Math.max(count, up[i] + down[i] + 1);

}

}

return count;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2: Fill in up & down with 1, and up[i] + down[i] - 1**

class Solution {

    public int longestMountain(int[] arr) {

        int count = 0;

        int n = arr.length;

        int[] up = new int[n];

        int[] down = new int[n];

        Arrays.fill(up, 1);

        Arrays.fill(down, 1);

        for(int i = 1; i < n; i++) {

            if(arr[i] > arr[i - 1]) {

                up[i] = up[i - 1] + 1;

            }

        }

        for(int i = n - 2; i >= 0; i--) {

            if(arr[i] > arr[i + 1]) {

                down[i] = down[i + 1] + 1;

            }

            if(up[i] > 1 && down[i] > 1) {

                count = Math.max(count, up[i] + down[i] - 1);

            }

        }

        return count;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 2: Find peak and expand on both left and right side (60 min)**

class Solution {

    public int longestMountain(int[] arr) {

        int count = 0;

        int n = arr.length;

        // Not enough elements to form a mountain

        if(n < 3) {

            return 0;

        }

        for(int i = 1; i < n - 1; i++) {

            // Check if the current element is a peak

            if(arr[i] > arr[i - 1] && arr[i] > arr[i + 1]) {

                int left = i - 1;

                int right = i + 1;

                // Expand on left

                while(left > 0 && arr[left] > arr[left - 1]) {

                    left--;

                }

                // Expand on right

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

                    right++;

                }

                count = Math.max(count, right - left + 1);

                // Move the index to the end of the current mountain

                i = right;

            }

        }

        return count;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to chatGPT**

LeetCode problem 845, "Longest Mountain in Array," asks for the length of the longest mountain subarray in a given array. A mountain subarray is defined as one that has at least three elements, with the middle element being greater than its neighbors.

Here's a Java solution that efficiently solves the problem:

class Solution {

    public int longestMountain(int[] arr) {

        if (arr == null || arr.length < 3) {

            return 0; // Not enough elements to form a mountain

        }

        int maxLength = 0;

        int n = arr.length;

        for (int i = 1; i < n - 1; i++) {

            // Check if the current element is a peak

            if (arr[i] > arr[i - 1] && arr[i] > arr[i + 1]) {

                int left = i - 1;

                int right = i + 1;

                // Expand to the left

                while (left > 0 && arr[left] > arr[left - 1]) {

                    left--;

                }

                // Expand to the right

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

                    right++;

                }

                // Calculate the length of the mountain

                int currentLength = right - left + 1;

                maxLength = Math.max(maxLength, currentLength);

                // Move the index to the end of the current mountain

                i = right;

            }

        }

        return maxLength;

    }

}

**Explanation:**

Initial Check: If the array is null or has fewer than three elements, it immediately returns 0 because a valid mountain cannot exist.

Iterate through the array: We start at index 1 and go up to n - 2 because we need at least one element on both sides of a peak to form a mountain.

Check for a Peak: A peak is an element that is greater than its neighbors. If a peak is found, we then look for the extent of the mountain.

Expand Left: Move left from the peak as long as each element is greater than the previous one.

Expand Right: Move right from the peak as long as each element is greater than the next one.

Calculate Length: The length of the mountain is right - left + 1.

Update Max Length: Keep track of the longest mountain found.

Move the Index: Once a mountain is processed, move the index to the end of this mountain to avoid redundant checks.

**Complexity:**

Time Complexity: O(n), where n is the length of the input array. **Each element is processed at most twice**.

Space Complexity: O(1), as we are using only a few extra variables.

This solution efficiently finds the longest mountain by checking for peaks and expanding outwards to find the full mountain range.

**Solution 3: One pass from left to right with O(1) space (60 min)**

**Style 1**

class Solution {

public int longestMountain(int[] arr) {

int count = 0;

int up = 0;

int down = 0;

int len = arr.length;

for(int i = 1; i < len; i++) {

// If we are going down and current element is greater than

// prev means last mountain ended

// And the current mountain began ('up' will be updated below)

if(down > 0 && arr[i - 1] < arr[i] || arr[i - 1] == arr[i]) {

up = 0;

down = 0;

}

// If current element is greater then previous then we are going

// up, else we are going down the mountain

if(arr[i - 1] < arr[i]) {

up++;

}

if(arr[i - 1] > arr[i]) {

down++;

}

if(up > 0 && down > 0) {

count = Math.max(count, up + down + 1);

}

}

return count;

}

}

**Style 2**

class Solution {

public int longestMountain(int[] arr) {

int count = 0;

int len = arr.length;

int i = 1;

while(i < len) {

while(i < len && arr[i - 1] == arr[i]) {

i++;

}

int up = 0;

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

i++;

up++;

}

int down = 0;

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

i++;

down++;

}

if(up > 0 && down > 0) {

count = Math.max(count, up + down + 1);

}

}

return count;

}

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/longest-mountain-in-array/solutions/135593/c-java-python-1-pass-and-o-1-space/>

**Intuition**: We have already many 2-pass or 3-pass problems, like 821. Shortest Distance to a Character. They have almost the same idea. One forward pass and one backward pass. Maybe another pass to get the final result, or you can merge it in one previous pass.

**Explanation**: In this problem, we take one forward pass to count up hill length (to every point).We take another backward pass to count down hill length (from every point). Finally a pass to find max(up[i] + down[i] + 1) where up[i] and down[i] should be positives.

**Time Complexity**: O(N)

public int longestMountain(int[] A) {

int N = A.length, res = 0;

int[] up = new int[N], down = new int[N];

for (int i = N - 2; i >= 0; --i) if (A[i] > A[i + 1]) down[i] = down[i + 1] + 1;

for (int i = 0; i < N; ++i) {

if (i > 0 && A[i] > A[i - 1]) up[i] = up[i - 1] + 1;

if (up[i] > 0 && down[i] > 0) res = Math.max(res, up[i] + down[i] + 1);

}

return res;

}

**Follow up**

Can you solve this problem with only one pass? Can you solve this problem in O(1) space?

**In this solution, I count up length and down length. Both up and down length are clear to 0 when A[i - 1] == A[i] or down > 0 && A[i - 1] < A[i].**

public int longestMountain(int[] A) {

int res = 0, up = 0, down = 0;

for (int i = 1; i < A.length; ++i) {

if (down > 0 && A[i - 1] < A[i] || A[i - 1] == A[i]) up = down = 0;

if (A[i - 1] < A[i]) up++;

if (A[i - 1] > A[i]) down++;

if (up > 0 && down > 0 && up + down + 1 > res) res = up + down + 1;

}

return res;

}

**Refer to**

<https://leetcode.com/problems/longest-mountain-in-array/solutions/135593/c-java-python-1-pass-and-o-1-space/comments/189334>

int longestMountain(vector<int> A)

{

int res = 0, up = 0, down = 0;

for (int i = 1; i < A.size(); ++i)

{

//If we are going down and current element is greater than prev MEANS last mountain ended

//And the current mountain began (Up will be updated below)

if(down>0 && A[i - 1] < A[i] || A[i - 1] == A[i])

up = down = 0;

//If current element is greater then previous then we are going up

//Else we are going down the mountain

A[i] > A[i -1] ? up++ : down++;

if (up>0 && down>0)

res = max(res, up + down + 1);

}

return res;

}

**Refer to**

<https://leetcode.com/problems/longest-mountain-in-array/solutions/135593/c-java-python-1-pass-and-o-1-space/comments/143182>

Updated list of problems that involved 1 or 2 passes from left to right/right to left:

[L53.Maximum Subarray (Ref.L821)](note://872ABD256AD34054B5D1F2E9992E2CAA)

[L121.Best Time to Buy and Sell Stock (Ref.L53)](note://CA1ACF4D33934A4CB3BF8D6ECB726E33)

[L152.Maximum Product Subarray (Ref.L53)](note://39691ECD970548F1999CAA98021DD7FC)

[L238.Product of Array Except Self (Ref.L845,L53)](note://0B52134081E7482F9337DF69820CC8E4)

[L739.Daily Temperatures](note://6FB3A11E2B7746DCA411301F941DD948)

[L769.Max Chunks To Make Sorted (Ref.L768,L739)](note://AE47A8D25BF048A49FE5365860A3CCCF)

[L768.Max Chunks To Make Sorted II (Ref.L769,L739)](note://BB68399B4CC048E6A2560D993E38F665)

[L821.Shortest Distance to a Character (Ref.L845)](note://WEB790653af463124a332a0f62fb0d60ff3)

[L581.P3.11.Shortest Unsorted Continuous Subarray (Ref.L845)](note://B2FA6CB7DCA84D0C9F2B6CF89ADC9CD7)

[L42.Trapping Rain Water (Ref.L238,L11)](note://638F207C2A4D4B94864DE5494C729557)

[L896.Monotonic Array](note://705FB59737A9406887D44F0DB529611D)