<https://leetcode.com/problems/minimum-number-of-removals-to-make-mountain-array/>

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 nums, return *the* ***minimum*** *number of elements to remove to make* nums *a* ***mountain array****.*

**Example 1:**

Input: nums = [1,3,1]

Output: 0

Explanation: The array itself is a mountain array so we do not need to remove any elements.

**Example 2:**

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

Output: 3

Explanation: One solution is to remove the elements at indices 0, 1, and 5, making the array nums = [1,5,6,3,1].

**Constraints:**

3 <= nums.length <= 1000

1 <= nums[i] <= 10^9

It is guaranteed that you can make a mountain array out of nums.

**Attempt 1: 2023-04-12**

**Solution 1: Two Pass Longest Increasing Subsequence (10 min)**

class Solution {

    public int minimumMountainRemovals(int[] nums) {

        int len = nums.length;

        int[] left\_to\_right = new int[len];

        int[] right\_to\_left = new int[len];

        Arrays.fill(left\_to\_right, 1);

        Arrays.fill(right\_to\_left, 1);

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

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

                if(nums[j] < nums[i]) {

                    left\_to\_right[i] = Math.max(left\_to\_right[i], left\_to\_right[j] + 1);

                }

            }

        }

        for(int i = len - 1; i >= 0; i--) {

            for(int j = len - 1; j > i; j--) {

                if(nums[j] < nums[i]) {

                    right\_to\_left[i] = Math.max(right\_to\_left[i], right\_to\_left[j] + 1);

                }

            }

        }

        int max\_mountain\_len = 0;

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

            if(left\_to\_right[i] > 1 && right\_to\_left[i] > 1) {

                max\_mountain\_len = Math.max(max\_mountain\_len, left\_to\_right[i] + right\_to\_left[i] - 1);

            }

        }

        return len - max\_mountain\_len;

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/minimum-number-of-removals-to-make-mountain-array/solutions/952016/java-lis-with-detailed-explanation-and-comments-o-n-2-time-and-o-n-space-revised/>

/\*

  Concept: We need to find the maximum number of elements of the array that can be involved in a mountain array.

  We know, that a mountain array contains a peak element and there is an increasing subsequence in the left of

  the peak and a decreasing subsequence in the right. So, we need to find out the element(peak), for which the

  total number of elements from the original array involved in the left increasing subsequence and the right

  decreasing subsequence, in maximum. This will create a mountain array with the peak element. Then, we can delete

  the rest of the elements of the array not involved in this mountain array.

\*/

class Solution {

    public int minimumMountainRemovals(int[] nums) {

        int n=nums.length;

        int []left=new int [n]; // maximum increasing subsequence in the left of an element.

        int []right=new int [n]; // maximum increasing subsequence in the right of an element.

        Arrays.fill(left,1);

        Arrays.fill(right,1);

        // calculating maximum increasing subsequence for the left of an index.

        for(int i=1;i<n;i++){

            for(int j=0;j<i;j++){

                if(nums[j]<nums[i]&&left[i]<left[j]+1)

                    left[i]=left[j]+1;

            }

        }

        // calculating maximum increasing subsequence for the right of an index.

        for(int i=n-2;i>=0;i--){

            for(int j=n-1;j>i;j--){

                if(nums[j]<nums[i]&&right[i]<right[j]+1)

                    right[i]=right[j]+1;

            }

        }

        // calculating the maximum number of elements that can be involved in a mountain array.

        int max=0;

        for(int i=1;i<n-1;i++){

            /\*

              If the below conditional statement is not given, then strictly increasing or strictly

              decreasing sequences will also be considered. It will hence fail in,

              Test case: [10, 9, 8, 7, 6, 5, 4, 5, 4].

              ---Thanks to @chejianchao for suggesting the test case.

              We need to make sure both the LIS on the left and right, ending at index i, has length > 1.

            \*/

            if(right[i]>1&&left[i]>1) // if element nums[i] is a valid peak,

                max=Math.max(max,left[i]+right[i]-1);

        }

        // we need to delete the rest of the elements.

        return n-max;

    }

}

// O(N^2) time and O(N) space.

**Why a Conventional Monotonic Stack Isn't Directly Applicable to LeetCode 1671**

**Nature of the Problem:** LeetCode 1671 involves finding the longest increasing subsequence (LIS) from the left and the longest decreasing subsequence (LDS) from the right to form a mountain array. **The problem is fundamentally about finding and combining subsequences, not just finding the nearest greater or smaller elements.** A mountain array requires both a valid increasing sequence up to a peak and a valid decreasing sequence after the peak.

**Global vs. Local Information: Monotonic stacks are typically used to handle local comparisons, i.e., nearest smaller/larger neighbors. However, finding LIS or LDS involves global comparisons — every element needs to consider potentially all previous (or next) elements to determine its position in a longer subsequence. This requires more than just the nearest elements, as an element's inclusion in a subsequence depends on a cumulative history of comparisons.**

**Subsequence Tracking:** In the LIS and LDS approach, we are interested in cumulative subsequences — the longest subsequence that ends or starts at a particular index. **Monotonic stacks don't directly handle subsequences but are more suited for immediate comparisons. Tracking the LIS/LDS needs dynamic programming or binary search for efficient insertion and updating of sequences.**

**Order and Combination of Subproblems:** For LeetCode 1671, we are required to combine two independent subsequences (LIS from the left and LDS from the right). Monotonic stacks don't naturally handle this type of combination efficiently. While a stack could manage a simple sequence check (e.g., strictly increasing or decreasing order), combining two sequences to form a mountain peak requires more comprehensive tracking of indices and values.

**Example: Using Monotonic Stack for Different Problems**

A problem where a monotonic stack fits perfectly is LeetCode 84 ("Largest Rectangle in Histogram"). Here, each bar's nearest smaller bar is critical in defining the maximum area. A monotonic stack efficiently tracks these nearest smaller elements by pushing and popping bars based on height. However, in the context of LIS/LDS for mountain arrays, the stack doesn't naturally track the "longest subsequence ending/starting at" information.

**Refer to**

[L300.Longest Increasing Subsequence](note://20C954956F6E4AA6A3575EA3146A1FCA)