1671. Minimum Number of Removals to Make Mountain Array

Hard Greedy Array Binary Search Dynamic Programming Leetcode Link

Problem Description

The problem provides an array of integers named `nums` and defines a **mountain array** as an array that increases in value to a certain point (the peak) and then strictly decreases in value thereafter. A mountain array must have at least one element before and after the peak, meaning at minimum the array size should be 3. The goal is to find the minimum number of elements that must be removed from `nums` so that the remaining array is a mountain array. Thus, the task is to transform `nums` into a mountain array using the fewest possible deletions.

Intuition

To solve this problem, the intuition is to find the longest increasing subsequence (LIS) from the left to a peak and from the peak to the right for each element in the array, considering that element as the peak. Since we need at least one element before and after the peak for the array to be a mountain array, we exclude peaks at the boundaries (first and last element).

The key insight is that the number of elements we need to delete from the array to form a mountain array is equal to the total number of elements minus the length of the longest subsequence that forms a mountain array. This longest subsequence will be the combination of the longest strictly increasing subsequence that reaches up to the peak (but not including the peak) and the longest strictly decreasing subsequence that starts just after the peak.

For each possible peak in the array, we determine the length of the LIS up to (but not including) that peak and the length of the longest decreasing subsequence (LDS) starting just after that peak and ending at the last element. Adding the lengths of these two subsequences together and subtracting one (to account for the peak being counted twice) gives us the length of the longest mountain subsequence that has that specific element as the peak. By iterating over all the elements (treating each as the peak) and applying this logic, and then maximizing the resultant lengths, we find the length of the desired longest mountain subsequence within the array. Subtracting this length from the total number of elements in the original array gives the minimum number of elements that need to be removed to form a valid mountain array.

Solution Approach

The solution leverages dynamic programming to find the Longest Increasing Subsequence (LIS) from the left and the longest decreasing subsequence from the right for each element.

To achieve this, two arrays `left` and `right` of the same size as the input `nums` are created. Initially, all elements in `left` and `right` are set to 1, meaning at the very least, every element can be considered a subsequence of length 1.

1. LIS from the left: We iterate through each element `i` from index 1 to `n-1`, where `n` is the size of `nums`. For each `i`, we also iterate through each previous element `j` from 0 to `i-1`. If `nums[i] > nums[j]`, it means the sequence can be extended, and we update `left[i]` to be the maximum of its current value and `left[j] + 1`.

2. Longest decreasing subsequence from the right: We mirror what we did for the LIS from the left, but in reverse. Starting from `n-2` down to 0, iterate the current element `i` and for each `i`, iterate over every following element `j` from `i+1` to `n-1`. If `nums[i] > nums[j]`, it indicates that we can extend a decreasing subsequence and we update `right[i]` with the max of its current value and `right[j] + 1`.

3. Once we have both `left` and `right`, we iterate through them together using `zip(left, right)` to calculate the length of the longest mountain subsequence that each element can form, as the peak, where we add the values of `left` and `right` together and subtract 1 (to not double-count the peak). We are only interested in the peaks that are not at the boundaries, hence we check if both `left[i] > 1` and `right[i] > 1`.

4. The length of the largest subsequence forming a mountain is the maximum value of the sequence length found in step 3. Since we want to find the minimum number of removals, we subtract this value from the total length of the input array, `n`.

By utilizing dynamic programming, this solution builds up subproblems (LIS to the left and right of every element) that help to solve the overall problem (minimum removals to form a mountain array). This avoids the inefficiency of checking every possible subsequence directly, which would be computationally expensive.

Example Walkthrough

Let's consider the array `nums` equal to `[2, 1, 1, 5, 6, 2, 3, 1]`. The goal is to remove the minimum number of elements to turn `nums` into a mountain array. Following the solution approach:

1. **Determine LIS from the left**: Starting from the second element, compare it with all the previous elements to calculate how long the increasing subsequence can be up to that point.
   - Initialize `left` array with `[1, 1, 1, 1, 1, 1, 1, 1]`.
   - Traverse the elements from left to right:
   - The subsequence `[2]` cannot be extended by `1`, so `left` stays the same.
   - The subsequence `[1, 1]` doesn't extend, so `left` remains the same.
   - `5` extends `[2]` and `[1]`, so `left[3] = 3`.
   - `6` extends `[2, 1, 5]`, so `left[4] = 4`.
   - `2` can't extend any subsequence, so `left[5]` is `1`.
   - `3` can extend `[2, 1, 5, 6]`, but is only a continuation of `[2]`, so `left[6] = 2`.
   - `1` doesn't extend any sequences, so `left` doesn't change.
   - The `left` array after iterations: `[1, 1, 1, 3, 4, 1, 2, 1]`.
2. **Find the longest decreasing subsequences from the right**: Apply analogous logic as LIS from the left, but in reverse.
   - Initialize `right` array with `[1, 1, 1, 1, 1, 1, 1, 1]`.
   - Traverse the elements from right to left:
   - `3` starts a new subsequence, so `right` stays the same.
   - `2` can extend `3`, so `right[5] = 2`.
   - `6` and `5` don't find smaller following elements, no changes.
   - `1` and `1` can extend multiple subsequences, so `right[1]` and `right[2]` become `2` because they can extend `2` and `3`.
   - The `right` array after iterations: `[1, 2, 2, 1, 1, 2, 1, 1]`.
3. **Combine the LIS and decreasing subsequences**:
   - We calculate the possible length for each element as the peak ignoring the first and last element:
   - Peaks `[1, 3, 4, 6]`: their combined lengths `[1, 3, 4, 2] + [2, 1, 1, 2] - 1`.
   - Lengths are `[2, 3, 4, 3]` for those peaks.
4. **Find the longest mountain and determine minimum removals**:
   - The longest mountain has length `4` (from peak `6`).
   - Minimum removals = Total length of `nums` (8) - length of the longest mountain (4).
   - Therefore, the minimum number of elements that needs to be removed to form a mountain array is `8 - 4 = 4`.

Using this approach, the elements at positions `[3, 4, 5, 6]` (zero-based index) form the longest mountain subsequence, `[1, 5, 6, 2]`, and removing the elements at positions `[0, 1, 2, 7]` results in a valid mountain array with the fewest deletions.

Python Solution

```python
from typing import List

class Solution:
    def minimumMountainRemovals(self, nums: List[int]) -> int:
        # Length of the input list
        length = len(nums)

        # Initialize DP arrays to store the longest increasing subsequence
        # from the left and from the right for each element
        left_lis = [1] * length
        right_lis = [1] * length

        # Populate the left LIS DP array
        for i in range(1, length):
            for j in range(i):
                # If the current number is greater than a number before it,
                # update the DP array to include the longer subsequence
                if nums[i] > nums[j]:
                    left_lis[i] = max(left_lis[i], left_lis[j] + 1)

        # Populate the right LIS DP array
        for i in range(length - 2, -1, -1):
            for j in range(i + 1, length):
                # If the current number is greater than a number after it,
                # update the DP array to include the longer subsequence
                if nums[i] > nums[j]:
                    right_lis[i] = max(right_lis[i], right_lis[j] + 1)

        # Calculate the maximum length of bitonic subsequence
        # which is a sum of left and right sequences minus 1
        # because the peak element is counted twice
        # Only consider peak elements which are part of both LIS and LDS
        max_bitonic_len = max(
            (left + right - 1)
            for left, right in zip(left_lis, right_lis)
            if left > 1 and right > 1
        )

        # The minimum number of removals is the total length of the array
        # minus the maximum length of the bitonic subsequence
        return length - max_bitonic_len

```

Java Solution

```java
class Solution {
    public int minimumMountainRemovals(int[] nums) {
        int length = nums.length;

        // Arrays to store the longest increasing subsequence ending at each index from the left and right
        int[] longestIncreasingLeft = new int[length];
        int[] longestIncreasingRight = new int[length];

        // Initialize the arrays with a default value of 1
        Arrays.fill(longestIncreasingLeft, 1);
        Arrays.fill(longestIncreasingRight, 1);

        // Calculate the longest increasing subsequence for each index from the left
        for (int i = 1; i < length; ++i) {
            for (int j = 0; j < i; ++j) {
                if (nums[i] > nums[j]) {
                    longestIncreasingLeft[i] = Math.max(longestIncreasingLeft[i], longestIncreasingLeft[j] + 1);
                }
            }
        }

        // Calculate the longest increasing subsequence for each index from the right
        for (int i = length - 2; i >= 0; --i) {
            for (int j = i + 1; j < length; ++j) {
                if (nums[i] > nums[j]) {
                    longestIncreasingRight[i] = Math.max(longestIncreasingRight[i], longestIncreasingRight[j] + 1);
                }
            }
        }

        int maxMountainSize = 0;
        // Find the maximum size of a valid mountain subsequence
        for (int i = 0; i < length; ++i) {
            // To form a mountain, the element nums[i] must be increasing and decreasing, both sides at least by 1 element
            if (longestIncreasingLeft[i] > 1 && longestIncreasingRight[i] > 1) {
                // Calculate the length of the mountain and update the maximum mountain size
                int mountainSize = longestIncreasingLeft[i] + longestIncreasingRight[i] - 1;
                maxMountainSize = Math.max(maxMountainSize, mountainSize);
            }
        }

        // The minimum removals is the array length minus the maximum mountain size
        return length - maxMountainSize;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minimumMountainRemovals(vector<int>& nums) {
        int size = nums.size();
        vector<int> left(size, 1), right(size, 1); // Initialize LIS vectors for left and right

        // Compute the length of longest increasing subsequence (LIS) from the left
        for (int i = 1; i < size; ++i) {
            for (int j = 0; j < i; ++j) {
                if (nums[i] > nums[j]) {
                    left[i] = max(left[i], left[j] + 1); // Update the LIS at i based on j
                }
            }
        }

        // Compute the length of LIS from the right
        for (int i = size - 2; i >= 0; --i) {
            for (int j = i + 1; j < size; ++j) {
                if (nums[i] > nums[j]) {
                    right[i] = max(right[i], right[j] + 1); // Update the LIS at i based on j
                }
            }
        }

        int maxMountainLength = 0; // Variable to keep track of the longest mountain length

        // Find the maximum length of a bitonic subsequence (peak of the mountain)
        for (int i = 0; i < size; ++i) {
            // Ensure we have an increasing and decreasing subsequence, i.e., a peak
            if (left[i] > 1 && right[i] > 1) {
                // Update the maxMountainLength if left[i] + right[i] - 1 is greater
                maxMountainLength = max(maxMountainLength, left[i] + right[i] - 1);
            }
        }

        // The minimum removals will be the total number of elements minus the longest mountain length
        return size - maxMountainLength;
    }
};

```

Typescript Solution

```typescript
function minimumMountainRemovals(nums: number[]): number {
    // Get the length of the input array.
    const length = nums.length;

    // Initialize arrays to keep track of the length of increasing subsequence from the left and right.
    const increasingFromLeft = new Array(length).fill(1);
    const increasingFromRight = new Array(length).fill(1);

    // Populate the increasingFromLeft array with the longest increasing subsequence ending at each index.
    for (let i = 1; i < length; ++i) {
        for (let j = 0; j < i; ++j) {
            if (nums[i] > nums[j]) {
                increasingFromLeft[i] = Math.max(increasingFromLeft[i], increasingFromLeft[j] + 1);
            }
        }
    }

    // Populate the increasingFromRight array with the longest increasing subsequence starting at each index.
    for (let i = length - 2; i >= 0; --i) {
        for (let j = i + 1; j < length; ++j) {
            if (nums[i] > nums[j]) {
                increasingFromRight[i] = Math.max(increasingFromRight[i], increasingFromRight[j] + 1);
            }
        }
    }

    // Initialize a variable to keep track of the max length of a bitonic subsequence.
    let maxLengthBitonic = 0;

    // Calculate the maximum length of the bitonic subsequence.
    // A bitonic subsequence increases and then decreases.
    // It must increase and decrease by at least one element on each side.
    for (let i = 0; i < length; ++i) {
        if (increasingFromLeft[i] > 1 && increasingFromRight[i] > 1) {
            maxLengthBitonic = Math.max(maxLengthBitonic, increasingFromLeft[i] + increasingFromRight[i] - 1);
        }
    }

    // The minimum mountain removals is the total length minus the max length of bitonic subsequence.
    return length - maxLengthBitonic;
}

```

Time and Space Complexity

The given Python code defines a function `minimumMountainRemovals` that finds the minimum number of elements to remove from an array to make the remaining array a mountain array, where a mountain array is defined as an array where elements first strictly increase then strictly decrease.

Time Complexity

The time complexity of the provided solution can be analyzed by examining its nested loops.

- The first `for` loop, responsible for populating the `left` list, contains a nested loop that compares each element to all previous elements to find the longest increasing subsequence up to the current index. This loop runs in `O(n^2)` time, where `n` is the length of the `nums` array.

- The second `for` loop, which populates the `right` list, also contains a nested loop that runs in reverse to find the longest decreasing subsequence from the current index to the end of the array. This loop similarly has a time complexity of `O(n^2)`.

- After computing the `left` and `right` lists, the final line involves a single loop that iterates over both lists to find the maximum value of `a + b - 1` where `a` and `b` refer to corresponding values of `left` and `right`. This loop runs in `O(n)` time.

Adding up the complexities of these loops, the first two dominant ones give us a time complexity of `O(n^2 + n^2)` which simplifies to `O(n^2)` since we drop constants and lower order terms when expressing big O notation.

Therefore, the total time complexity of the function is `O(n^2)`.

Space Complexity

The space complexity is determined by the amount of additional memory the algorithm uses in relation to the input size:

- We have two lists, `left` and `right`, each of size `n`. Therefore, the space used by these lists is `2n`, which in big O notation is `O(n)`.

- Aside from the `left` and `right` lists, there are only a few integer variables used, which do not scale with the input and thus contribute `O(1)` to the space complexity.

Hence, the total space complexity of the function is `O(n)`.