2012. Sum of Beauty in the Array

Medium Array Leetcode Link

Problem Description

In this problem, we are provided with a 0-indexed integer array `nums`. We need to determine the "beauty" for each element `nums[i]` at positions `i` from `1` to `nums.length - 2`, according to the following rules:

- The beauty is 2 if `nums[i]` is greater than all preceding elements `nums[j]` where `j < i`, and less than all succeeding elements `nums[k]` where `k > i`.
- The beauty is 1 if `nums[i]` is greater than the immediate preceding element `nums[i - 1]` and less than the immediate succeeding element `nums[i + 1]`, but does not satisfy the condition for beauty 2.
- The beauty is 0 if none of the above conditions is satisfied.

Our goal is to calculate the sum of beauty for all such `nums[i]`.

Intuition

The solution builds on the key observation that to find the beauty of an index `i`, it suffices to determine the maximum element to the left of `i` and the minimum element to the right of `i`. This leads us to an efficient way to evaluate the beauty for each index.

The approach is as follows:

1. Create two additional arrays, `lmx` and `rmi`, to record the maximum element observed from the beginning up to `i - 1` and the minimum element observed from the end down to `i + 1`.
2. Iterate through the `nums` array from left to right, populating `lmx` by recording the maximum value seen so far.
3. Iterate through the `nums` array from right to left, populating `rmi` with the minimum value seen so far.
4. Now, traverse the array again and for each `i` (from `1` to `nums.length - 2`), check the following:
   - If `lmx[i] < nums[i] < rmi[i]`, add `2` to the answer since `nums[i]` satisfies the condition for beauty `2`.
   - Else if `nums[i - 1] < nums[i] < nums[i + 1]` and the first condition is not satisfied, add `1` to the answer, marking beauty `1`.
5. Sum the beauty values calculated for each `i` to get the final answer.

Solution Approach

To implement the solution, we follow these steps, making use of sequential iterations and auxiliary space for the arrays needed to keep track of maximum and minimum boundaries:

1. **Initialization**:
   - Calculate the length `n` of the array `nums`.
   - Initialize two arrays `lmx` and `rmi` of size `n` to keep track of the left maximum and right minimum values, respectively. `lmx[i]` will store the maximum value from the start of the array to index `i - 1`, and `rmi[i]` will store the minimum value from the end of the array to index `i + 1`.
   - `ans` is initially filled with `0` because there is no number before the start of the array.
   - `rmi` is filled with a very large number, `100001`, to ensure that any real number in the array `nums` will be less than this placeholder value.
2. **Populate `lmx`**:
   - Iterate through `nums` from left to right starting from index `1` up to `n - 1`.
   - Update `lmx[i]` such that it holds the maximum value seen up to `nums[i - 1]`. This is done using the formula `lmx[i] = max(lmx[i - 1], nums[i - 1])`.
3. **Populate `rmi`**:
   - Iterate through `nums` from right to left starting from index `n - 2` down to `0`.
   - Update `rmi[i]` such that it holds the minimum value seen from `nums[i + 1]` to the end of the array. The formula used here is `rmi[i] = min(rmi[i + 1], nums[i + 1])`.
4. **Calculate the total beauty**:
   - Initialize a variable `ans` to hold the sum of beauty scores.
   - Iterate through the elements of `nums` from index `1` to `n - 2` (inclusive).
   - Check if the beauty of `nums[i]` is `2` by comparing if `lmx[i] < nums[i] < rmi[i]`. If this condition is true, increment `ans` by `2`.
   - Else if `nums[i]` does not qualify for a beauty of `2`, check if it is greater than the element to its left and less than the element to its right (i.e., check if `nums[i - 1] < nums[i] < nums[i + 1]`). If true, increment `ans` by `1`.
   - If neither condition is satisfied, the beauty for that element is `0`, so `ans` remains the same.
5. **Return the result**:
   - After the loop completes, `ans` contains the sum of beauty for all `nums[i]`, and we return this value.

The main data structures used in this solution are the arrays `lmx` and `rmi` for dynamic programming, which store computed values that can be used to determine the beauty of each element efficiently. The algorithm makes use of `max()` and `min()` functions for comparisons, and a single pass through the array (ignoring the separate passes for `lmx` and `rmi` initializations) to calculate the sum of beauty. This approach ensures that we have all the necessary information to evaluate the beauty of each element without using nested loops, which would result in a higher computational complexity.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the integer array `nums = [1, 2, 3, 4, 2]`.

Step 1: Initialization

- `n = 5` (length of `nums`)
- Initialize `lmx = [0, 0, 0, 0, 0]` and `rmi = [100001, 100001, 100001, 100001, 100001]`

Step 2: Populate `lmx`

- Starting from `i = 1`, iterate to `i = 4`:
  - `lmx[1] = max(0, nums[0]) = 1`
  - `lmx[2] = max(1, nums[1]) = 2`
  - `lmx[3] = max(2, nums[2]) = 3`
  - `lmx[4] = max(3, nums[3]) = 4`
- Now `lmx = [0, 1, 2, 3, 4]`

Step 3: Populate `rmi`

- Starting from `i = 3`, iterate to `i = 0`:
  - `rmi[3] = min(100001, nums[4]) = 2`
  - `rmi[2] = min(2, nums[3]) = 2`
  - `rmi[1] = min(2, nums[2]) = 2`
  - `rmi[0] = min(2, nums[1]) = 2`
- Now `rmi = [2, 2, 2, 2, 100001]`

Step 4: Calculate the total beauty

- Initialize `ans = 0`
- For `i = 1` to `n - 2`:
  - For `i = 1`:
    - `lmx[1]` is `1`, `nums[1]` is `2`, `rmi[1]` is `2`.
    - `nums[1]` is greater than `lmx[1]` but not less than `rmi[1]`, so check next condition.
    - `nums[0]` is `1`, `nums[1]` is `2`, `nums[2]` is `3`. It satisfies `nums[0] < nums[1] < nums[2]`, so add `1` to `ans`.
  - For `i = 2`:
    - `lmx[2]` is `2`, `nums[2]` is `3`, `rmi[2]` is `2`.
    - `nums[2]` does not satisfy any beauty conditions, so `ans` stays the same.
  - For `i = 3`:
    - `lmx[3]` is `3`, `nums[3]` is `4`, `rmi[3]` is `2`.
    - `nums[3]` is greater than both `lmx[3]` and `rmi[3]`, so it adds nothing to `ans`.
- The final value of `ans` after the loop is `1`.

Step 5: Return the result

The result, which is the sum of beauty for all `nums[i]`, is `1`. This is the final answer to the problem.

In this particular example, the only element to contribute to the beauty sum was `nums[1]` with a beauty of `1`.

Python Solution

```python
from typing import List

class Solution:
    def sumOfBeauties(self, nums: List[int]) -> int:
        num_elements = len(nums)  # Get the number of elements in the list
        max_left = [0] * num_elements  # Initialize a list to store the maximum to the left of each position
        min_right = [100001] * num_elements  # Initialize a list to store the minimum to the right of each position

        # Populate max_left by finding the maximum on the left for each position in nums
        for i in range(1, num_elements):
            max_left[i] = max(max_left[i - 1], nums[i - 1])

        # Populate min_right by finding the minimum on the right for each position in nums
        for i in range(num_elements - 2, -1, -1):
            min_right[i] = min(min_right[i + 1], nums[i + 1])

        beauty_sum = 0  # This variable will hold the cumulative beauty of the array

        # Loop through each element of the array except the first and last
        for i in range(1, num_elements - 1):
            # Check if the element is greater than the max to the left and less than the min to the right
            if max_left[i] < nums[i] < min_right[i]:
                beauty_sum += 2  # If it is, the beauty score for this number is 2
            # Otherwise, check if the element is greater than its previous and less than its next element
            elif nums[i - 1] < nums[i] < nums[i + 1]:
                beauty_sum += 1  # If so, the beauty score for this number is 1

        return beauty_sum  # Return the total accumulated beauty
```

Java Solution

```java
class Solution {
    public int sumOfBeauties(int[] nums) {
        int n = nums.length; // Get the length of the input array
        int[] leftMax = new int[n]; // Array to keep track of maximum values from the left
        int[] rightMin = new int[n]; // Initialize an array to keep track of minimum values from the right
        rightMin[n - 1] = 100001; // Set the last element to a high value as a sentinel

        // Fill the leftMax array with the maximum value encountered from the left up to that index
        for (int i = 1; i < n; ++i) {
            leftMax[i] = Math.max(leftMax[i - 1], nums[i - 1]);
        }

        // Fill the rightMin array with the minimum value encountered from the right up to that index
        for (int i = n - 2; i >= 0; --i) {
            rightMin[i] = Math.min(rightMin[i + 1], nums[i + 1]);
        }

        int totalBeauty = 0; // Variable to hold the total sum of beauty
        // Loop through the array, omitting the first and last element
        for (int i = 1; i < n - 1; ++i) {
            // Check if the current element is larger than the maximum to its left and smaller than the minimum to its right
            if (leftMax[i] < nums[i] && nums[i] < rightMin[i]) {
                totalBeauty += 2; // Add 2 to beauty as it satisfies the special condition
            } else if (nums[i - 1] < nums[i] && nums[i] < nums[i + 1]) {
                totalBeauty += 1; // Add 1 to beauty if it's simply larger than its adjacent elements
            }
        }
        // Return the sum of beauty of all elements
        return totalBeauty;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int sumOfBeauties(vector<int>& nums) {
        int size = nums.size();
        vector<int> leftMax(size); // Stores the maximum to the left of each element.
        vector<int> rightMin(size, 100001); // Stores the minimum to the right of each element, initially set high

        // Populate leftMax by keeping track of the maximum number seen so far from the left.
        for (int i = 1; i < size; ++i) {
            leftMax[i] = max(leftMax[i - 1], nums[i - 1]);
        }

        // Populate rightMin by keeping track of the minimum number seen so far from the right.
        for (int i = size - 2; i >= 0; --i) {
            rightMin[i] = min(rightMin[i + 1], nums[i + 1]);
        }

        int totalBeauty = 0;  // This will store the total beauty of the array.

        // Calculate the beauty for each number in the array excluding the first and last element.
        for (int i = 1; i < size - 1; ++i) {
            // If the current element is greater than the maximum on its left
            // and less than the minimum on its right, add 2 to total beauty.
            if (leftMax[i] < nums[i] && nums[i] < rightMin[i]) {
                totalBeauty += 2;
            }
            // If it doesn't meet the first condition but is still increasing with respect
            // to its immediate neighbors, add 1 to total beauty.
            else if (nums[i - 1] < nums[i] && nums[i] < nums[i + 1]) {
                totalBeauty += 1;
            }
        }

        // Return the total beauty of the array.
        return totalBeauty;
    }
};
```

Typescript Solution

```typescript
// Function to calculate the sum of beauties for all elements in the array except the first and last.
function sumOfBeauties(nums: number[]): number {
    // Determine the length of input array.
    let n: number = nums.length;
    // Initialize prefix and postfix arrays to keep track of max and min values seen so far from either end.
    let prefixMax: number[] = new Array(n).fill(0);
    let postfixMin: number[] = new Array(n).fill(0);

    // Set the first element of prefix and the last element of postfix to be the corresponding values from 'nums'.
    prefixMax[0] = nums[0];
    postfixMin[n - 1] = nums[n - 1];

    // Fill the prefixMax and postfixMin arrays.
    for (let i: number = 1, j: number = n - 2; i < n; ++i, --j) {
        prefixMax[i] = Math.max(nums[i], prefixMax[i - 1]);
        postfixMin[j] = Math.min(nums[j], postfixMin[j + 1]);
    }

    // Initialize the sum of beauties.
    let sumOfBeautyPoints: number = 0;

    // Check the beauty for each element, based on the conditions and update the sum accordingly.
    for (let i: number = 1; i < n - 1; ++i) {
        // Check for beauty level 2 condition.
        if (prefixMax[i - 1] < nums[i] && nums[i] < postfixMin[i + 1]) {
            sumOfBeautyPoints += 2;
        // Check for beauty level 1 condition.
        } else if (nums[i - 1] < nums[i] && nums[i] < nums[i + 1]) {
            sumOfBeautyPoints += 1;
        }
    }

    // Return the total sum of beauty points.
    return sumOfBeautyPoints;
}
```

Time and Space Complexity

Time Complexity

The given code consists of three separate `for` loops that are not nested. Each of these loops runs linearly with respect to the length of the input array `nums`, which is denoted as `n`.

1. The first loop initializes the `lmx` array, which takes `O(n)` time as it iterates from `1` to `n-1`.
2. The second loop initializes the `rmi` array, which also takes `O(n)` time as it iterates from `n-2` to `0`.
3. The third loop calculates the `ans` (answer) by iterating once again in linear time over the range from `1` to `n-1`, resulting in `O(n)` time.

When we add these up, since they are executed in sequence and not nested, the overall time complexity of the code is `O(n) + O(n) + O(n)`, which simplifies to `O(n)`.

Space Complexity

The space complexity of the code is due to the additional arrays `lmx` and `rmi` that are both of length `n`, and the space used for variables like `n`, `i`, and `ans`.

1. The `lmx` array uses `O(n)` space.
2. The `rmi` array also uses `O(n)` space.

Besides these arrays, only a constant amount of extra space is used for the loop indices and the `ans` variable. Thus, the total auxiliary space used by the algorithm is `O(n) + O(n)` which simplifies to `O(n)`.

In conclusion, the time complexity of the algorithm is `O(n)` and the space complexity is `O(n)`.