2763. Sum of Imbalance Numbers of All Subarrays

Hard | Array | Hash Table | Ordered Set | Leetcode Link

Problem Description

The problem requires calculating the "imbalance number" of all subarrays of a given array `nums`. The *imbalance number* of an array is defined as the count of instances where, in the sorted version of an array, consecutive elements differ by more than 1. Essentially, it measures how many times a pair of neighboring elements in a sorted list have a gap larger than 1.

To compute the *sum of imbalance numbers* of all subarrays, we must consider every continuous slice of the original array, sort each one, and calculate its imbalance number, then add these up for all possible subarrays.

A simple example to illustrate this concept would be:

Given the array [1,2,4], it has three subarrays: [1], [1,2], [1,2,4]. Their sorted versions are [1], [1,2], and [1,2,4] respectively. There is no pair of neighboring elements that differ by more than 1 in the first two subarrays, so their imbalance numbers are 0. However, in the last subarray, 2 and 4 have a difference of 2, which is greater than 1; thus, its imbalance number is 1. So the sum of all imbalance numbers is 0 + 0 + 1 = 1.

Intuition

To solve this problem, we adopt an approach that iteratively considers every possible subarray starting from each index. We initialize the imbalance number for each subarray as zero and update it as we grow the subarray window.

The solution involves two nested loops. The outer loop fixes the starting point of the subarray, while the inner loop extends the subarray by incrementing the endpoint. This way, we examine all contiguous subarrays of the array `nums`.

For each element added to the current subarray, we maintain a sorted list of all elements in the subarray to quickly determine the neighbors of the new element. This ordered list is crucial because it allows us to assess imbalances in constant or logarithmic time, rather than sorting the subarray every time which would be computationally expensive.

When a new number is added:

- We find its direct neighbors in the sorted list to see if it introduces any new imbalances.
- If the new number creates an imbalance with either (or both) of its neighbors, we update our imbalance counter.
- If the new number is inserted between two other numbers that were previously causing an imbalance, the presence of the new number may fix this imbalance, and we decrease our counter.

The variable `cnt` maintains the imbalance number of the current subarray, which is updated on each iteration based on the conditions explained above. After iterating through all subarrays, the `ans` variable holds the sum of the imbalance numbers from all subarrays, which is returned as the solution.

Thus, this algorithm effectively combines the enumeration of subarrays with the maintenance of a dynamically-updated sorted list, which leads to a much more efficient calculation of the sum of imbalance numbers than a naive approach.

Solution Approach

The implementation capitalizes on an important data structure—an ordered set. In the Python solution provided, the `SortedList` from the `sortedcontainers` module serves as a replacement for an ordered set that is not natively available in Python. Using this data structure, the algorithm keeps track of a subarray's elements in sorted order — an essential step for calculating the imbalance number efficiently.

For every subarray processed, the solution steps through the following algorithm:

1. Initialize an empty `SortedList` called `sl` and a counter `cnt` for the imbalance number.
2. Iterate over the array with two pointers, marked by the indices `i` and `j`, where `i` is fixed by the outer loop and `j` is varied by the inner loop. Each iteration of the inner loop examines a new subarray extending from `i` to `j`.
3. Before adding the new element at `j` to the subarray:
   - Use the `bisect_left` method to find the position `k` in the sorted list where the new element would be inserted. This gives us the direct higher neighbor in the sorted subarray.
   - Determine the immediate lower neighbor in the sorted subarray by subtracting one from the index `k`, resulting in index `h`.
4. Check for imbalances:
   - If there is an element at index `k`, and the difference between this element and `nums[j]` is greater than 1, increment `cnt` as a new imbalance is introduced.
   - Similarly, if there is an element at index `h`, and the difference between `nums[j]` and this element is greater than 1, increment `cnt` for this new imbalance.
   - If both conditions are met, check if the new element `nums[j]` is filling a gap between `sl[h]` and `sl[k]` that was previously an imbalance; if so, decrement `cnt` since an imbalance is resolved.
5. Insert the new element `nums[j]` into the sorted list to update the subarray.
6. Add the current `cnt` to a running total `ans` which accumulates the imbalance numbers for all subarrays examined so far.
7. Repeat steps 3-6 for each `j` until all subarrays starting with index `i` have been processed.

After the outer loop concludes, `ans` holds the final result: the sum of imbalance numbers for all subarrays in `nums`. This algorithm leverages the efficient search and insertion operations provided by `SortedList` to calculate the imbalance number dynamically as each element is considered, thus avoiding the need for repeated sorting of each subarray.

Example Walkthrough

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

We need to find the sum of imbalance numbers for all subarrays of this array.

1. Start with an empty `SortedList` called `sl` and a variable `cnt` initialized to 0. The variable `ans` will accumulate the sum of imbalance numbers and is also initialized to 0.

2. Begin with the outer loop, where `i` starts at 0. That is the starting point of our subarray.

3. Now, let's move to the inner loop, where `j` will go from `i` to the length of `nums` - 1. For each value of `j`, we follow the steps below.

4. At `i = 0, j = 0`, the subarray is `[3]`. Add `3` to `sl` (which was empty). There are no neighbors to check for imbalance as this is the first element. No need to update `cnt`, and add `cnt` (which is 0) to `ans`.

5. At `i = 0, j = 1`, the subarray is `[3, 1]`. We plan to insert `1` into `sl`, but before that, we check for imbalances:

   - The position `k` found by `bisect_left` for `1` is `0` as `1` is smaller than `3`.
   - There is no element at `h = k - 1` since `k` is `0`.
   - Position `k` points to `3`, and since `abs(3 - 1) > 1`, we increment `cnt` to `1`.
   - Insert `1` into `sl`, which is now `[1, 3]`.
   - Add the current `cnt` (which is 1) to `ans`.
6. At `i = 0, j = 2`, the subarray is `[3, 1, 4]`. Inserting `4`:

   - The position `k` for `4` is `2` as `4` should be inserted after `3`.
   - The lower neighbor index `h` is `1`, which corresponds to `3` in `sl`.
   - Position `k` is out of bounds, so there's no upper neighbor to compare with `4`.
   - For the lower neighbor (`sl[h]` is `3`), `abs(4 - 3) = 1` which doesn't increment `cnt`.
   - Insert `4` into `sl`, now `[1, 3, 4]`.
   - Add the current `cnt` (still 1) to `ans`.
7. At `i = 0, j = 3`, the subarray is `[3, 1, 4, 2]`, insert `2`:

   - The position `k` for `2` is `2` as `2` should be inserted between `1` and `3`.
   - The lower neighbor index `h` is `1`, which corresponds to `1` in `sl`.
   - There is an imbalance between `2` and `3` since `abs(3 - 2) = 1`. No update to `cnt`.
   - There is no imbalance between `2` and `1`, since `abs(2 - 1) = 1`. No update to `cnt`.
   - No existing imbalance is corrected by adding `2` because there were no elements between `1` and `3` causing an imbalance before.
   - Insert `2` into `sl`, now `[1, 2, 3, 4]`.
   - Add the current `cnt` (still 1) to `ans`.
8. Now, the outer loop moves to `i = 1` and the process repeats with a new starting point, checking for potential imbalances as elements are added and updating the counters accordingly.

After all iterations of both `i` and `j`, `ans` holds the sum of the imbalance numbers for all subarrays in `nums`.

This approach dynamically calculates the imbalance number as each element is considered and efficiently updates both the imbalance counters and the sorted list of the subarray elements without needing to sort subarrays repeatedly.

Python Solution

```python
from sortedcontainers import SortedList
from typing import List

class Solution:
    def sum_imbalance_numbers(self, nums: List[int]) -> int:
        # Initialize the number of elements and the answer variable
        n = len(nums)
        total_imbalance = 0

        # Outer loop - iterate over all elements in nums
        for i in range(n):
            sorted_list = SortedList()  # Create a new sorted list for the subarray starting at i
            imbalance_count = 0  # Initialize the imbalance count for the current subarray

            # Inner loop - iterate over the subarray starting at index i
            for j in range(i, n):
                # Find the position where nums[j] should be inserted to keep the list sorted
                insertion_index = sorted_list.bisect_left(nums[j])

                # Check the difference between the number before insertion_index, if any,
                # to see if an imbalance occurs
                if insertion_index > 0 and nums[j] - sorted_list[insertion_index - 1] > 1:
                    imbalance_count += 1

                # Check the difference between the number after insertion_index, if any,
                # to see if an imbalance occurs
                if insertion_index < len(sorted_list) and sorted_list[insertion_index] - nums[j] > 1:
                    imbalance_count += 1

                # Adjust count for the previous imbalance if new insertion fixes an existing imbalance
                if insertion_index > 0 and insertion_index < len(sorted_list) and \
                   sorted_list[insertion_index] - sorted_list[insertion_index - 1] > 1:
                    imbalance_count -= 1

                # Add the current number to the sorted list
                sorted_list.add(nums[j])

                # Add the current count of imbalances to the total
                total_imbalance += imbalance_count

        # Return the total imbalance count over all subarrays
        return total_imbalance

```

Java Solution

```java
class Solution {
    public int sumImbalanceNumbers(int[] nums) {
        int n = nums.length; // Get the length of the array
        int imbalanceSum = 0; // Initialize the sum of imbalance numbers

        // Iterate through each element in the array as the starting point
        for (int i = 0; i < n; ++i) {
            TreeMap<Integer, Integer> frequencyMap = new TreeMap<>(); // TreeMap to keep the numbers sorted and frequency count
            int imbalanceCount = 0; // Initialize the imbalance count for the current subarray

            // Iterate through the array starting at i to calculate the imbalances
            for (int j = i; j < n; ++j) {
                // Find the smallest number greater than or equal to nums[j]
                Integer nextHigherNumber = frequencyMap.ceilingKey(nums[j]);

                // If such a number exists and the difference is greater than 1, increase imbalance
                if (nextHigherNumber != null && nextHigherNumber - nums[j] > 1) {
                    ++imbalanceCount;
                }

                // Find the greatest number less than or equal to nums[j]
                Integer nextLowerNumber = frequencyMap.floorKey(nums[j]);

                // If such a number exists and the difference is greater than 1, increase imbalance
                if (nextLowerNumber != null && nums[j] - nextLowerNumber > 1) {
                    ++imbalanceCount;
                }

                // If both numbers exist and their difference is greater than 1, decrease imbalance
                if (nextLowerNumber != null && nextHigherNumber != null && nextHigherNumber - nextLowerNumber > 1) {
                    --imbalanceCount;
                }

                // Record the frequency of the current number, merging with the existing count
                frequencyMap.merge(nums[j], 1, Integer::sum);

                // Add the current imbalance count to the total sum of imbalances
                imbalanceSum += imbalanceCount;
            }
        }
        // Return the total sum of imbalances found in all subarrays
        return imbalanceSum;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <set>

class Solution {
public:
    int sumImbalanceNumbers(vector<int>& nums) {
        int n = nums.size(); // Get the size of input vector 'nums'.
        int imbalanceSum = 0; // Initialize the result to store the sum of imbalances.

        // Loop over the elements of 'nums' considering each element as the start of the subarray.
        for (int i = 0; i < n; ++i) {
            multiset<int> sortedElements; // Create a multiset to store the elements in sorted order.
            int countImbalance = 0; // Count to keep track of imbalance instances in the current subarray.

            // Loop over the elements of 'nums' from the 'i'th element to the end to form subarrays.
            for (int j = i; j < n; ++j) {
                // Find the position of the current element in the sorted multiset.
                auto it = sortedElements.lower_bound(nums[j]);
                // If there's an element greater than the current element and the difference is greater than 1, it contributes to imb
                if (it != sortedElements.end() && *it - nums[j] > 1) {
                    ++countImbalance;
                }
                // If there's an element smaller than the current element and the difference is greater than 1, it contributes to imb
                if (it != sortedElements.begin() && nums[j] - *prev(it) > 1) {
                    ++countImbalance;
                }
                // If both conditions are met, we have counted the imbalance one extra time, so decrement the counter.
                if (it != sortedElements.end() && it != sortedElements.begin() && *it - *prev(it) > 1) {
                    --countImbalance;
                }
                // Insert the current element into the multiset to update the subarray.
                sortedElements.insert(nums[j]);
                // Add the current imbalance count to our running sum.
                imbalanceSum += countImbalance;
            }
        }
        // Return the total imbalance sum.
        return imbalanceSum;
    }
};

```

Typescript Solution

```typescript
function sumImbalanceNumbers(nums: number[]): number {
    const n: number = nums.length; // Get the length of input array 'nums'.
    let imbalanceSum: number = 0; // Initialize the result to store the sum of imbalances.

    // Loop over the elements of 'nums', considering each element as the start of the subarray.
    for (let i = 0; i < n; ++i) {
        const sortedElements: Set<number> = new Set(); // Create a Set to store unique elements in sorted order.
        let countImbalance: number = 0; // Count to keep track of imbalance instances in the current subarray.

        // Loop over the elements of 'nums' from the 'i'th element to the end to form subarrays.
        for (let j = i; j < n; ++j) {
            // Convert Set to Array and find the position of the current element to handle sorted insertion.
            let sortedArray = Array.from(sortedElements);

            // Find the correct index where current number would be inserted to maintain sorted order.
            let sortedIndex = sortedArray.findIndex(x => x > nums[j]);

            // Check for imbalance on both sides of the found index.
            // If there's a greater element whose difference with the current element is more than 1,
            // it contributes to the imbalance count.
            if (sortedIndex !== -1 && sortedArray[sortedIndex] - nums[j] > 1) {
                countImbalance++;
            }

            // If there's a smaller element (previous index) whose difference with the current element is
            // more than 1, it contributes to the imbalance count.
            let smallerIndex = sortedIndex - 1;
            if (smallerIndex >= 0 && nums[j] - sortedArray[smallerIndex] > 1) {
                countImbalance++;
            }

            // Insert the current element into the right sorted position.
            sortedElements.add(nums[j]);

            // Add the current count of imbalance to the running sum.
            imbalanceSum += countImbalance;
        }
    }

    // Return the total sum of imbalances found.
    return imbalanceSum;
}

```

Time and Space Complexity

Time Complexity

The given code has a nested loop where the outer loop runs `n` times, and the inner loop runs up to `n` times as well, where `n` is the length of the input array `nums`. For each iteration of the inner loop, the code executes operations that include binary searches and insertions into a `SortedList`. Each of these operations (binary search and insertion) in a `SortedList` has a time complexity of `O(log n)`.

This means that for each `i`, we are doing `n - i` insertions, and up to `n - i` binary searches, each taking `O(log n)` time. The sum of `n - i` over all valid `i` is the sum of the first `n` integers, which is `(n(n+1))/2`, simplifying the total number of operations to `(n^2 + n)/2`. Since each operation is `O(log n)`, we multiply the number of operations by the cost of each operation to get the overall time complexity.

Therefore, the time complexity is `O(n^2 * log n)` because the dominant factor is the nested loop with the `SortedList` operations, and the constant factors can be ignored in the Big O notation.

Space Complexity

The space complexity of the code can be analyzed by accounting for the space taken up by the `SortedList`. This list can grow up to `n` elements in size, where `n` is the length of the input array `nums`. No other data structures in the code use more space than this, and the space used by variables such as `i`, `j`, `k`, `h`, and `cnt` is negligible compared to the space taken up by the `SortedList`.

Therefore, the space complexity is `O(n)`, as the `SortedList` is the data structure occupying the most space and it grows linearly with the input size.