1862. Sum of Floored Pairs

Hard Array Math Binary Search Prefix Sum

Problem Description

The task is to find the sum of the integer parts of the fractions formed by dividing each element in the given integer array `nums` by every other element in the same array. This means for each pair of indices `(i, j)` in the array, we're calculating `floor(nums[i] / nums[j])`, where `floor()` is a mathematical function that returns the largest integer less than or equal to a given number (the integer part of a division). All possible pairwise divisions are included. The result could potentially be very large, so we return the final sum modulo `10**9 + 7` to keep the number within the bounds of typical 32-bit integer representation and to manage big numbers in a reasonable way, as often required in computational problems.

Intuition

To solve this problem, we employ several algorithmic strategies to make the calculations efficient. Since we are dealing with all possible pairs, a naive solution could take up too much time, especially for large arrays. Here are some observations and steps to approach the solution more efficiently:

1. **Counting Occurrences**: We first count how many times each number appears in the array `nums`. This is important because the same division will occur as many times as the number appears. We keep these counts in a dictionary or array, referred to as `cnt`.
2. **Prefix Sum Array**: Next, we construct a prefix sum array `s` that keeps a running sum of counts of numbers up to each index. `s[i]` contains the total number of array numbers less than or equal to `i`. This step will greatly accelerate finding the sum of counts between any two number ranges since it'll be a simple subtraction, `s[j] - s[i - 1]`.
3. **Enumerating Divisors and Quotients**: Now, instead of looping over every pair, we iterate over all possible divisors `y` that appear in `nums` and then for each `y`, over all possible quotients `d`. For each quotient, we calculate how many numerators give this specific quotient when divided by `y` using the prefix sum array. This is the number of pairs `(i, j)` where `nums[i] / nums[j]` yields quotient `d`.
4. **Calculating the Sum**: For each `d` and `y`, we can find the sum of `floor(nums[i] / nums[j])` by taking the count of such numerators from the prefix sum array, multiplying it with count of `y` and quotient `d`. Finally, we accumulate these into the `ans` variable.
5. **Modulo Arithmetic**: We have to frequently follow modulo to avoid integer overflow issues. Each intermediate sum is taken modulo `10**9 + 7` to maintain a manageable number size.

By implementing these steps, we optimize the brute force enumeration approach into an efficient calculation involving counting, prefix sums, and careful iteration to avoid unnecessary recalculations, leading to a manageable time complexity for large input sizes.

Solution Approach

The implementation of the given solution follows a structured approach that utilizes algorithms, data structures, and patterns to efficiently compute the required sum. Here's a step-by-step explanation of how the solution approach mentioned above is implemented:

1. **Initialize Data Structures:**
   - A dictionary or counter, named `cnt`, to store counts of each unique number in `nums`.
   - A list `s` of size `mx + 1`, where `mx` is the maximum number found in `nums`, to store the prefix sum of counts.
2. **Build Counters and Prefix Sum:**
   - Use the `Counter` class from the `collections` module to count occurrences of numbers in `nums`: `cnt = Counter(nums)`.
   - Create the prefix sum array. The element `s[i]` is computed by adding `cnt[i]` to the prefix sum up to `i - 1`: `s[i] = s[i - 1] + cnt[i]`.
3. **Enumerate Denominators:**
   - Loop through possible divisors `y` ranging from `1` to `mx`. Here `y` will be the denominator when calculating `floor(nums[i] / nums[j])`.
   - For each `y`, we only continue if `y` was present in `nums` (`if cnt[y]:`), to avoid unnecessary computations.
4. **Enumerate Quotients:**
   - For the chosen `y`, iterate through possible quotients `d` starting from `1`, incrementing `d` till `d * y` exceeds `mx`.
   - During each iteration, calculate the count of numerators that would yield the quotient `d` when divided by `y`. This is done by subtracting the prefix sum up to `d * y - 1` from the prefix sum up to `min(mx, (d + 1) * y - 1)`.
   - The result `s[min(mx, (d + 1) * y - 1)] - s[(d + 1) * y - 1]` gives us the number of numerators in the specified range.
5. **Compute Contribution to Answer:**
   - Multiply the count of numerators by `cnt[y]` and the quotient `d` to find the total contribution of that quotient with the denominator `y` to the final answer.
   - Add this value to `ans`, ensuring at each step to apply modulo `10**9 + 7` to keep the value within bounds: `ans += cnt[y] * d * (s[min(mx, d * y + y - 1)] - s[d * y - 1]) % mod`.
6. **Return Result:**
   - After completing the enumeration of `y` and `d`, return the accumulated `ans`.

Each step incorporates careful use of algorithms (use of counting and prefix sums) and data structures (arrays for prefix sums, counters for occurrences) alongside modular arithmetic to not only bring down the complexity but also manage the large numbers. This approach provides a significant performance advancement compared to the brute force method that would have had a prohibitive time complexity for large inputs.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose our input array `nums` is `[1, 2, 3, 4, 6]`. We want to find the sum modulo `10**9 + 7` of the integer parts of the fractions formed by dividing each number in `nums` by every other number in the same array.

1. **Initialize Data Structures:**
   - Let `cnt` be our dictionary which will store occurrences of each number. `cnt = {1:1, 2:1, 3:1, 4:1, 6:1}`
   - The list `s` will store the prefix sum of counts. Since our `mx` is `6`, `s = [0, 0, 0, 0, 0, 0, 0]` initially.
2. **Build Counters and Prefix Sum:**
   - `cnt` will be populated straight from our array: `cnt = Counter([1, 2, 3, 4, 6])`.
   - To build the prefix sum array, we update `s`:
     - `s[1] = 1`, `s[2] = s[1] + 1`, `s[3] = s[2] + 1`, and so on until `s[6] = s[5] + 1`.
   - The complete `s` becomes `[0, 1, 2, 3, 4, 4, 5]`.
3. **Enumerate Denominators:**
   - We loop through possible divisors `y` from `1` to `6`.
4. **Enumerate Quotients:**
   - For each `y`, iterate through possible quotients `d`. For example, for `y = 2`:
   - Quotient `d = 1`, numerator range is `[2, 3]`. `s[3] - s[1] = 2` numerators.
   - Quotient `d = 2`, numerator range is `[4, 5]`. `s[5] - s[3] = 1` numerator.
   - And so on until `d * y > mx`.
5. **Compute Contribution to Answer:**
   - For each `d`, calculate its contribution:
     - `d = 1: cnt[2] * 1 * (s[3] - s[1]) % mod = 1 * 1 * (2 - 1) % mod = 1`
     - `d = 2: cnt[2] * 2 * (s[5] - s[3]) % mod = 1 * 2 * (5 - 4) % mod = 2`
   - Repeat above steps for each divisor `y` and accumulate the answer into `ans`.
6. **Return Result:**
   - After iterating over all `y` and `d` pairs, the `ans` might look something like this: `ans = 1 + 2 + (other contributions calculated using the same steps)`.
   - Finally, `ans % mod` is returned as the result.

This walkthrough has demonstrated how each step in the solution works together. By avoiding the brute force enumeration for every pair and applying more efficient arithmetic and data structures, the overall computation becomes significantly faster, making it viable even for larger arrays.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def sum_of_floored_pairs(self, nums):
        MODULO = 10 ** 9 + 7  # Constant for the modulo operation to prevent large integer values
        max_num = max(nums)  # Find the maximum number in the list
        num_counts = Counter(nums)  # Count the occurrences of each number in nums
        max_num = max(nums)  # Find the maximum number in the list
        prefix_sum = [0] * (max_num + 1)  # Create a list for prefix sum of counts

        # Calculate the prefix sum for counts of each number
        for i in range(1, max_num + 1):
            prefix_sum[i] = prefix_sum[i - 1] + num_counts[i]

        answer = 0  # This will store the final answer

        # Iterate over each unique number to calculate contributions
        for divisor in range(1, max_num + 1):
            if num_counts[divisor]:
                multiple = 0
                # For each divisor, calculate floor pairs for its multiples
                while multiple * divisor <= max_num:
                    # Increment answer by number of pairs times current multiple
                    range_sum = prefix_sum[min(max_num, multiple * divisor + divisor - 1)] - prefix_sum[multiple * divisor - 1]
                    answer += num_counts[divisor] * multiple * range_sum
                    answer %= MODULO  # Take mod to avoid large numbers
                    multiple += 1  # Increment the multiple for the next iteration

        return answer
```

Java

```java
class Solution {
    public int sumOfFlooredPairs(int[] nums) {
        int mod = (int) 1e9 + 7; // Define the mod for the final answer
        int maxElement = 0;

        // Find the maximum element in the array
        for (int num : nums) {
            maxElement = Math.max(maxElement, num);
        }

        // Create a frequency array
        int[] frequency = new int[maxElement + 1];

        // Create an array to hold the prefix sum of the frequency array
        int[] prefixSum = new int[maxElement + 1];

        // Count the frequency of each number in the array
        for (int num : nums) {
            ++frequency[num];
        }

        // Calculate the prefix sum of the frequency array
        for (int i = 1; i <= maxElement; ++i) {
            prefixSum[i] = prefixSum[i - 1] + frequency[i];
        }

        long answer = 0; // This will store the final result

        // Iterate through all the unique elements in nums (as per the frequency)
        for (int num = 1; num <= maxElement; ++num) {
            if (frequency[num] > 0) {
                // For each unique element, find the sum of floored pairs
                for (int divisor = 1; divisor * num <= maxElement; ++divisor) {
                    // Add the contribution of the current divisor to the total sum
                    answer += (long) frequency[divisor] * (prefixSum[Math.min(maxElement, divisor * num + num - 1)] - prefixSum[divisor * num - 1]) * divisor;
                    answer %= mod; // Modulo operation to keep the sum within bounds
                }
            }
        }

        // Return the answer cast to an integer
        return (int) answer;
    }
}
```

C++

```cpp
class Solution {
public:
    int sumOfFlooredPairs(vector<int>& nums) {
        const int MOD = 1e9 + 7; // Use a constant for the modulo value
        int maxNum = *max_element(nums.begin(), nums.end()); // Find the max number in nums
        vector<int> count(maxNum + 1, 0); // Counts for each number up to maxNum
        vector<long> prefixSum(maxNum + 1, 0); // Prefix sums of the counts
        // Populate the count array
        for (int num : nums) {
            ++count[num];
        }
        // Calculate the prefix sum array
        for (int i = 1; i <= maxNum; ++i) {
            prefixSum[i] = prefixSum[i - 1] + count[i];
        }
        long long answer = 0; // Accumulate the final answer
        // Iterate over all numbers in count array
        for (int x = 1; x <= maxNum; ++x) {
            if (count[x]) { // If this number appears in nums
                // For each multiple of the number, calculate contributions
                for (int multiplier = 1; multiplier * x <= maxNum; ++multiplier) {
                    int nextBound = min(maxNum, multiplier * x + x - 1);
                    answer += (long long)(prefixSum[nextBound] - prefixSum[multiplier * x - 1]) * multiplier * count[x];
                    answer %= MOD; // Ensure we are within the modulo value
                }
            }
        }
        return answer; // Return the computed result
    }
};
```

TypeScript

```typescript
function sumOfFlooredPairs(nums: number[]): number {
    // Find the maximum value in the array to define the range of prefixes
    const maxNum = Math.max(...nums);

    // Initialize a counter array to store the frequency of each number up to the maximum
    const frequency: number[] = Array(maxNum + 1).fill(0);

    // Initialize a prefix sum array to store the running sum of frequencies
    const prefixSum: number[] = Array(maxNum + 1).fill(0);

    // Fill the frequency array with the frequency of each number in 'nums'
    for (const num of nums) {
        ++frequency[num];
    }

    // Compute the prefix sums of frequencies
    for (let i = 1; i <= maxNum; ++i) {
        prefixSum[i] = prefixSum[i - 1] + frequency[i];
    }

    // Initialize the answer variable to store the final result
    let answer = 0;
    const mod = 1e9 + 7; // Modulo to prevent overflow

    // Traverse through the range of numbers
    for (let x = 1; x <= maxNum; ++x) {
        // If frequency[x] is not zero, it means the current number has a non-zero frequency
        for (let d = 1; d * x <= maxNum; ++d) {
            answer += frequency[x] * d * (prefixSum[Math.min((d + 1) * x - 1, maxNum)] - prefixSum[d * x - 1]);
            answer %= mod;
        }
    }

    // Return the final answer
    return answer;
}
```

```python
from collections import Counter

class Solution:
    def sum_of_floored_pairs(self, nums):
        MODULO = 10 ** 9 + 7  # Constant for modulo operation to prevent large integer values
        num_counts = Counter(nums)  # Count the frequency of each number in nums
        max_num = max(nums)  # Find the maximum number in the list
        prefix_sum = [0] * (max_num + 1)  # Create a list for prefix sum of counts

        # Calculate the prefix sum for counts of each number
        for i in range(1, max_num + 1):
            prefix_sum[i] = prefix_sum[i - 1] + num_counts[i]

        answer = 0  # This will store the final answer

        # Iterate over each unique number to calculate contributions
        for divisor in range(1, max_num + 1):
            if num_counts[divisor]:
                multiple = 0
                # For each divisor, calculate floor pairs for its multiples
                while multiple * divisor <= max_num:
                    # Increment answer by number of pairs times current multiple
                    range_sum = prefix_sum[min(max_num, multiple * divisor + divisor - 1)] - prefix_sum[multiple * divisor - 1]
                    answer += num_counts[divisor] * multiple * range_sum
                    answer %= MODULO  # Take mod to avoid large numbers
                    multiple += 1  # Increment the multiple for the next iteration

        return answer
```

Time and Space Complexity

Time Complexity

The time complexity of the function `sumOfFlooredPairs` is largely determined by the nested loop structure where the outer loop runs for numbers from `1` to the maximum number `mx` in the array `nums`, and the inner loop runs while `d * y` is less than or equal to `mx`. For each outer loop iteration, the inner loop runs approximately `mx / y` times. Summing this over all `y` from `1` to `mx` gives the geometric series sum which is proportional to `mx + log(mx)`. Thus, the total time complexity is $O(M + \log(M))$, where M is the maximum value in the array `nums`.

Space Complexity

The space complexity is determined by the additional memory used by the algorithm which includes the `cnt` Counter and the prefix sum array `s`. Both of these data structures have a size that is dependent on the maximum value `mx` in the array `nums`, not the length of the `nums` array itself. Since `s` is an array of length `mx + 1`, the space complexity is $O(M)$. The `cnt` Counter also does not exceed $O(M)$ in size, so the overall space complexity remains $O(M)$.