1862. Sum of Floored Pairs

Binary Search Prefix Sum

## **Problem Description**

Hard

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.

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

# ntuition

cnt.

variable.

sizes.

possible pairs, a naïve solution could take up too much time, especially for large arrays. Here are some observations and steps to approach the solution more efficiently: 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

To solve this problem, we employ several algorithmic strategies to make the calculations efficient. Since we are dealing with all

- 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]. 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. 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

**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

**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

### 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.

implemented:

**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]).

∘ 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].

• Use the Counter class from the collections module to count occurrences of numbers in nums: cnt = Counter(nums).

For each y, we only continue if y was present in nums (if cnt[y]:), to avoid unnecessary computations.

**Initialize Data Structures:** 

**Build Counters and Prefix Sum:** 

**Compute Contribution to Answer:** 

had a prohibitive time complexity for large inputs.

- **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 <u>prefix sum</u> up to d \* y - 1 from the prefix sum up to min(mx, d \* y + y - 1). • The result s[min(mx, d \* y + y - 1)] - s[d \* y - 1] gives us the number of numerators in the specified range.
- 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. **Return Result:**
  - 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
- 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

same array.

**Example Walkthrough** 

**Initialize Data Structures:** 

• The complete s becomes [0, 1, 2, 3, 4, 4, 5].

After completing the enumeration of y and d, return the accumulated ans.

• 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. **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:

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

• We loop through possible divisors y from 1 to 6. **Enumerate Quotients:**  $\circ$  For each y, iterate through possible quotients d. For example, for y = 2:

This walkthrough has demonstrated how each step in the solution works together. By avoiding the brute force enumeration for

 Quotient d = 2, numerator range is [4, 5]. s[5] - s[3] = 1 numerator.  $\circ$  And so on until d \* y > mx.

same steps).

Solution Implementation

class Solution:

**Enumerate Denominators:** 

**Compute Contribution to Answer:** For each d, calculate its contribution:

 $\bullet$  s[1] = 1, s[2] = s[1] + 1, s[3] = s[2] + 1, and so on until s[6] = s[5] + 1.

**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

Finally, ans % mod is returned as the result.

def sum\_of\_floored\_pairs(self, nums):

for i in range(1, max\_num + 1):

public int sumOfFlooredPairs(int[] nums) {

// Create a frequency array

for (int num : nums) {

++frequency[num];

maxElement = Math.max(maxElement, num);

// Create an array to hold the prefix sum of the frequency array

int[] frequency = new int[maxElement + 1];

int[] prefixSum = new int[maxElement + 1];

for (int i = 1; i <= maxElement; ++i) {</pre>

// Count the frequency of each number in the array

// Calculate the prefix sum of the frequency array

every pair and applying more efficient arithmetic and data structures, the overall computation becomes significantly faster, making it viable even for larger arrays.

Quotient d = 1, numerator range is [2, 3].s[3] - s[1] = 2 numerators.

-d = 1: cnt[2] \*1 \* (s[3] - s[1]) % mod = 1 <math>\*1 \* (2 - 1) % mod = 1

-d = 2: cnt[2] \*2 \* (s[5] - s[3]) % mod = <math>1 \* 2 \* (1 - 0) % mod = 2

Repeat above steps for each divisor y and accumulate the answer into ans.

**Python** from collections import Counter

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

prefix\_sum = [0] \* (max\_num + 1) # Create a list for prefix sum of counts

answer += num\_counts[divisor] \* multiple \* range\_sum

multiple += 1 # Increment the multiple for the next iteration

answer %= MODULO # Take mod to avoid large numbers

max\_num = max(nums) # Find the maximum number in the list

prefix\_sum[i] = prefix\_sum[i - 1] + num\_counts[i]

# Calculate the prefix sum for counts of each number

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 = 1 # For each divisor, calculate floor pairs for its multiples while multiple \* divisor <= max\_num:</pre> # 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]

```
final 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) {
```

class Solution {

Java

return answer

```
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) {</pre>
            if (frequency[num] > 0) {
                // For each unique element, find the sum of floored pairs
                for (int divisor = 1; divisor * num <= maxElement; ++divisor) {</pre>
                    // Add the contribution of the current divisor to the total sum
                    answer += (long) frequency[num] * divisor * (prefixSum[Math.min(maxElement, divisor * num + num - 1)] - prefixed
                    answer %= MOD; // Modulo operation to keep the sum within bounds
        // Return the answer cast to an integer
        return (int) answer;
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<int> 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) {</pre>
            prefixSum[i] = prefixSum[i - 1] + count[i];
        long long answer = 0; // Accumulate the final answer
        // Iterate over all numbers in count array
        for (int y = 1; y <= maxNum; ++y) {</pre>
            if (count[y]) { // If this number appears in nums
                // For each multiple of the number y, calculate contributions
                for (int multiplier = 1; multiplier * y <= maxNum; ++multiplier) {</pre>
                    int nextBound = min(maxNum, multiplier * y + y - 1);
                    answer += static_cast<long long>(count[y]) * multiplier *
                               (prefixSum[nextBound] - prefixSum[multiplier * y - 1]);
                    answer %= MOD; // Ensure we are within the modulo value
        return answer; // Return the computed result
```

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

let answer = 0;

**}**;

**TypeScript** 

function sumOfFlooredPairs(nums: number[]): number {

const frequency: number[] = Array(maxNum + 1).fill(0);

const prefixSum: number[] = Array(maxNum + 1).fill(0);

// Initialize the answer variable to store the final result

const modulo = 1e9 + 7; // Modulus to prevent overflow

const maxNum = Math.max(...nums);

for (const num of nums) {

// Find the maximum value in the array to define the range of prefixes

// Initialize a prefix sum array to store the running sum of frequencies

// Fill the frequency array with the frequency of each number in 'nums'

answer %= MODULO # Take mod to avoid large numbers

multiple += 1 # Increment the multiple for the next iteration

// Initialize a counter array to store the frequency of each number up to the maximum

```
// Traverse through the range of numbers
      for (let y = 1; y <= maxNum; ++y) {</pre>
          if (frequency[y]) { // Only proceed if the current number has a non-zero frequency
              for (let d = 1; d * y <= maxNum; ++d) {
                  answer += frequency[y] * d * (prefixSum[Math.min((d + 1) * y - 1, maxNum)] - prefixSum[d * y - 1]);
                  answer %= modulo;
      // Return the final answer
      return answer;
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 = 1
               # For each divisor, calculate floor pairs for its multiples
               while multiple * divisor <= max_num:</pre>
                    # 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
```

## **Time Complexity** The time complexity of the function sumOfFlooredPairs is largely determined by the nested loop structure where the outer loop

**Time and Space Complexity** 

return answer

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** 

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

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 0(M). The cnt Counter also does not exceed O(M) in size, so the overall space complexity remains O(M).