2344. Minimum Deletions to Make Array Divisible

Hard | Array | Math | Number Theory | Sorting | Heap (Priority Queue) | Leetcode Link

Problem Description

This problem presents us with two arrays of positive integers, `nums` and `numsDivide`. The goal is to perform the minimum number of deletions in `nums` to ensure that the smallest number in `nums` divides all numbers in `numsDivide`. If no element in `nums` can be the divisor for every element in `numsDivide`, the function should return `-1`.

The division rule here is that an integer `x` divides another integer `y` if the remainder when `y` is divided by `x` is zero, denoted as `y % x == 0`.

To solve this problem, you need to consider two main steps:

1. Identify the smallest number in `nums` that can be a divisor for all elements in `numsDivide`.
2. Find the minimum number of deletions in `nums` required to make this number the smallest in `nums`.

Intuition

To begin, since we want a number from `nums` that divides all elements in `numsDivide`, we need to find the Greatest Common Divisor (GCD) of `numsDivide`. All elements of `numsDivide` must be divisible by this GCD, so our potential divisor in `nums` must also divide the GCD of `numsDivide`. Therefore, the smallest number in `nums` that also divides the GCD of `numsDivide` will serve as the required divisor.

With that in mind, the first step is to calculate the GCD of all elements in `numsDivide` using a built-in function.

Next, we search for the smallest number in `nums` that can divide this GCD, which can be done using a generator expression within the `min` function. If no such number exists in `nums` (when `min` returns the `default=0`), we cannot perform the desired operation, therefore we return `-1`.

If a divisor is found, we count the number of elements in `nums` that are smaller than this divisor since any such elements must be deleted for the divisor to be the smallest element in `nums`. The count of these elements gives us the minimum number of deletions needed. The summing using a generator expression in the `return` statement calculates this count and returns it.

Solution Approach

The implementation of the solution for this problem follows a straightforward approach, with the primary focus being the calculation of the Greatest Common Divisor (GCD) and the minimization of deletions.

Here's the step-by-step breakdown of the solution:

1. Calculate the GCD of elements in `numsDivide` array:

   - The `gcd` function from the Python standard library can compute GCD of two numbers. Leveraging the unpacking operator `*`, we can pass all elements of `numsDivide` to this function to find their collective GCD.

   ```
   x = gcd(*numsDivide)
   ```

2. Identify the smallest number in `nums` that divides the GCD:

   - We use a generator expression to iterate over all the values `v` in `nums`.
   - The condition `x % v == 0` ensures we only consider those elements that properly divide the GCD `x`.
   - The `min` function is used to find the smallest of such elements.
   - The `default=0` is used to handle the scenario where no such divisibility is possible, which will lead to the `min` function returning 0.

   ```
   y = min((v for v in nums if x % v == 0), default=0)
   ```

3. Count and return the minimum deletions:

   - If we found a divisor `y`, then we count the number of elements in `nums` that are less than `y`. These elements would prevent `y` from being the smallest element if not deleted.
   - This count is computed with the sum of a generator expression.
   - For every element `v` in `nums`, if `v < y`, it adds 1 to the sum.

   ```
   return sum(v < y for v in nums)
   ```

4. Handle the case where no suitable divisor is found:

   - If no element in `nums` can divide the GCD, which means `y` is 0, we cannot perform the operation, so we return `-1`.

   ```
   if y else -1
   ```

In summary, the solution leverages mathematical properties (divisibility and GCD), along with Python's built-in `gcd` function, generator expressions for memory-efficient iteration, and conditional logic to directly return the minimum number of deletions or `-1` if the problem conditions cannot be met.

Example Walkthrough

Let's consider an example to illustrate the solution approach. Suppose we have the following arrays:

`nums = [4, 3, 6]`

`numsDivide = [24, 48]`

Calculating the GCD of Elements in `numsDivide`

First, we find the GCD of all elements in the `numsDivide` array:

The GCD of `24` and `48` can be found as:

`GCD(24, 48) = 24`

This means any divisor in `nums` needs to divide `24` to be a valid divisor for all numbers in `numsDivide`.

Identifying the Smallest Number in `nums` Dividing the GCD

Now, we look for the smallest number in `nums` that can divide the GCD `24`:

- For `4`, we check if `24 % 4 == 0`, which is `True`.
- For `3`, we check `24 % 3` and find it's also `True` as `24` is divisible by `3`.
- For `6`, we check `24 % 6` and see that this is `True` as well.

All numbers [4, 3, 6] in `nums` can divide `24`, but we need the smallest one; hence we find the min which is `3`.

Counting and Returning Minimum Deletions

To make `3` the smallest number in `nums`, we must delete all numbers that are smaller than `3`. In `nums`, there are no such numbers.

Therefore, the minimum number of deletions is `0`.

This means no numbers from `nums` need to be deleted for the smallest number within it to divide all numbers in `numsDivide`.

Handling the Case of No Appropriate Divisor Found

Since we did find the number `3` as a suitable divisor, we did not encounter the scenario of the `min` function returning `0`, which would have led us to return `-1`.

Using the aforementioned steps, the solution successfully calculates that no deletions are needed from `nums` so that the smallest number in `nums` can divide all elements in `numsDivide`.

Python Solution

```python
from math import gcd
from typing import List

class Solution:
    def minOperations(self, nums: List[int], numsDivide: List[int]) -> int:
        # Calculate the greatest common divisor (GCD) of all elements in numsDivide
        common_divisor = gcd(*numsDivide)

        # Find the smallest element in nums that is a divisor of the GCD
        # If there is no such element, the default value will be 0
        min_divisible = min((value for value in nums if common_divisor % value == 0), default=0)

        # If the smallest element is not found, return -1 as it's not possible
        # to make all numsDivide elements divisible by any number in nums
        if min_divisible == 0:
            return -1

        # Calculate the number of operations needed by counting elements in nums
        # that are smaller than the smallest valid divisor (min_divisible)
        operations = sum(value < min_divisible for value in nums)

        # Return the count of operations needed
        return operations

```

Java Solution

```java
class Solution {
    // This method finds the minimum number of operations required
    // to make every number in numsDivide divisible by some number
    // in nums by removing the smallest numbers in nums.
    public int minOperations(int[] nums, int[] numsDivide) {
        // Initialize the greatest common divisor (gcd) of all numbers in numsDivide.
        int gcdValue = 0;
        for (int value : numsDivide) {
            gcdValue = gcd(gcdValue, value);
        }

        // Set an initial high value to find the minimum value in nums that divides the gcd without remainder.
        int minDivisibleValue = Integer.MAX_VALUE;
        for (int value : nums) {
            if (gcdValue % value == 0) {
                minDivisibleValue = Math.min(minDivisibleValue, value);
            }
        }

        // If no number was found, return -1 as it's not possible to satisfy the condition with any deletions.
        if (minDivisibleValue == Integer.MAX_VALUE) {
            return -1;
        }

        // Count the numbers of operations (number of elements smaller than the minDivisibleValue to be deleted).
        int operations = 0;
        for (int value : nums) {
            if (value < minDivisibleValue) {
                operations++;
            }
        }

        // Return the number of operations required.
        return operations;
    }

    // Helper method to compute the gcd of two numbers using the Euclidean algorithm.
    private int gcd(int a, int b) {
        // If b is zero, a is the gcd by definition.
        return b == 0 ? a : gcd(b, a % b);
    }
}

```

C++ Solution

```cpp
#include <vector>  // Required to include vector
#include <algorithm> // Required for std::min function
using namespace std;

class Solution {
public:
    int minOperations(vector<int>& nums, vector<int>& numsDivide) {
        // Initializing gcdValue with 0 to calculate GCD of all values in numsDivide
        int gcdValue = 0;
        // Calculating GCD of all elements in numsDivide
        for (int& value : numsDivide) {
            gcdValue = gcd(gcdValue, value);
        }
        // Setting the minimum possible value greater than all elements in nums
        int minValueGreaterThanAll = 1 << 30; // Large value as upper limit.
        // Finding the smallest number in nums that divides the gcdValue without remainder
        for (int& value : nums) {
            if (gcdValue % value == 0) {
                minValueGreaterThanAll = min(minValueGreaterThanAll, value);
            }
        }

        // If minValueGreaterThanAll is not changed, it means no such number is found. Return -1.
        if (minValueGreaterThanAll == 1 << 30) {
            return -1;
        }
        // Counting the number of operations to remove numbers smaller than minValueGreaterThanAll
        int operationsCount = 0;
        for (int& value : nums) {
            operationsCount += value < minValueGreaterThanAll;
        }

        return operationsCount; // Returning the minimum number of operations.
    }

    // Function to calculate the gcd of two numbers
    int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }
};

```

Typescript Solution

```typescript
// Importing necessary functionalities from external libraries
import { min } from 'lodash';

// Function to calculate the greatest common divisor (GCD) of two numbers
function calculateGCD(a: number, b: number): number {
    return b === 0 ? a : calculateGCD(b, a % b);
}

// Function to find the minimum number of operations required
function minOperations(nums: number[], numsDivide: number[]): number {
    // Initialize gcdValue with 0 to calculate GCD of all values in numsDivide
    let gcdValue: number = 0;

    // Calculate GCD of all elements in numsDivide
    for (const value of numsDivide) {
        gcdValue = calculateGCD(gcdValue, value);
    }

    // Initialize minValueGreaterThanAll with a large number as an upper limit.
    let minValueGreaterThanAll: number = 1 << 30;

    // Find the smallest number in nums that divides the gcdValue without a remainder
    for (const value of nums) {
        if (gcdValue % value === 0) {
            minValueGreaterThanAll = Math.min(minValueGreaterThanAll, value);
        }
    }

    // If minValueGreaterThanAll is not changed, no such number is found; return -1.
    if (minValueGreaterThanAll === 1 << 30) {
        return -1;
    }

    // Count the number of operations to remove numbers smaller than minValueGreaterThanAll
    let operationsCount: number = 0;
    for (const value of nums) {
        operationsCount += value < minValueGreaterThanAll ? 1 : 0;
    }

    // Return the minimum number of operations.
    return operationsCount;
}

```

Time and Space Complexity

The given Python code aims to find the minimum number of operations to delete elements from `nums` so that the greatest common divisor (GCD) of `numsDivide` can divide all elements in the modified `nums` list. It finds the GCD of the elements of `numsDivide`, then finds the minimum element `y` in `nums` that is a divisor of the GCD, and counts elements in `nums` less than `y`.

Time Complexity

Let `n` be the length of the `nums` list and `m` be the length of the `numsDivide` list.

1. `gcd(*numsDivide)`: The function computes the GCD of all elements in `numsDivide`. The time complexity of the `gcd` function for two numbers is `O(log(min(a, b)))`, where `a` and `b` are the two numbers. The GCD function will be called for every element in `numsDivide` beyond the first two. Therefore, the time complexity for this portion can be considered `O(m * log (A))`, where `A` is the average of the `numsDivide` list.

2. `min((v for v in nums if x % v == 0), default=0)`: This generator expression iterates over all elements `v` in `nums` and checks if `v` divides `x` without remainder. In the worst case, it will iterate through the entire `nums` list, resulting in a time complexity of `O(n)`.

3. `sum(v < y for v in nums)`: This expression iterates over the list `nums`, and for each element, it increments the sum if the element is less than `y`. This operation also has a time complexity of `O(n)` because it goes through all `n` elements.

Combining these, the overall worst-case time complexity is: `O(m * log (A) + 2n)`. Since `m * log(A)` and `n` are not related by a constant factor, we can't simplify it further. However, typically `m` and `log(A)` are much smaller than `n` in practical scenarios, such that we can consider it as `O(n)` for practical purposes.

Space Complexity

Let's analyze the space complexity of the algorithm:

1. `gcd(*numsDivide)`: The `gcd` function itself uses `O(1)` additional space as it only requires a constant space for the computation.

2. `min((v for v in nums if x % v == 0), default=0)`: The generator expression used here does not store intermediate results and computes the minimum on the fly. Therefore, it also uses `O(1)` space.

3. `sum(v < y for v in nums)`: Similar to the `min` function call, this `sum` leverages a generator expression and does not store intermediate results, so it uses `O(1)` additional space.

The space complexity of the entire algorithm is `O(1)` since all additional space used is constant and does not scale with the size of the input `nums` or `numsDivide`.