2644. Find the Maximum Divisibility Score

Easy Array Leetcode Link

Problem Description

The problem provides us with two arrays: `nums` (containing integers) and `divisors` (containing possible divisors). We are tasked with finding the integer from `divisors` which has the highest divisibility score, where the divisibility score of an element in `divisors` is the count of elements in `nums` that it can divide evenly (without leaving a remainder). In case more than one divisor has the same highest score, we need to return the smallest among them.

To put it simply, we want to answer the question: "Which number in `divisors` divides the most numbers in `nums` with no remainder?" and if there's a tie, we select the smallest number.

Intuition

This problem is suited for a brute force approach that checks each number in `divisors` against every number in `nums` to calculate its divisibility score.

- We begin by setting a variable `mx` to 0, to keep track of the maximum score found so far, and another variable `ans` to keep the divisor with the maximum score. We initialize `ans` with the first element of `divisors` since we need to return an element from `divisors` in case all scores are zero.

- We iterate over each divisor in `divisors`. For each divisor, we count how many numbers in `nums` are divisible by it. This is done using a summation with a conditional check, where we use the modulo operator `%` to test if the remainder is zero (meaning divisibility).

- If the current divisor's score is greater than the maximum score we've seen (`mx`), we update `mx` with this new score and also update `ans` with the current divisor.

- If the current divisor's score is equal to the maximum score but less than the current `ans`, we update `ans` to this divisor since it is smaller and we want the smallest divisor in case of a tie.

By the end of the iteration, `ans` will hold the divisor with the highest divisibility score, and in the case of a tie, the smallest one amongst them. This algorithm guarantees that we look at each possibility, and eventually, the correct answer is identified and returned.

Solution Approach

The solution provided in the reference code implements the brute force approach described in the intuition section. Here is a detailed breakdown of the approach:

- We have a `for` loop that goes through each element `div` in the `divisors` array.
- Inside the loop, we have a key expression `sum(x % div == 0 for x in nums)`. This expression counts the number of elements in `nums` that are divisible by `div` (the `x % div == 0` part checks if `x` is divisible by `div` without a remainder).
  - This is a generator comprehension within the `sum` function that goes over each element `x` in `nums` and yields `1` if `x` is divisible by `div`, and `0` otherwise. The `sum` function then adds up these 0s and 1s to get the total count.
- The `cnt` variable is used to hold this count, which represents the divisibility score of the current divisor `div`.
- We compare `cnt` against `mx` to determine if we have found a higher divisibility score:
  - If `cnt` is greater than `mx`, we have indeed found a new divisor with a higher score. We assign `cnt` to `mx`, and `div` to `ans`.
  - If `cnt` is equal to `mx` but `div` is smaller than the current `ans`, then we update `ans` with `div`. This ensures that among divisors with the same highest score, we will return the smallest `divisor`.
- No additional data structures are required, making this approach efficient in terms of space complexity. The time complexity can be considered as O(n * m), where 'n' is the length of the `nums` array and 'm' is the length of the `divisors` array, since each `divisor` is checked against all `nums`.

The algorithm employs a simple comparison-based technique, and its strength lies in its straightforwardness and direct mapping to the problem statement without any need for optimization tricks or complicated data structures. This is highly suitable for scenarios where the array sizes are manageable and high efficiency is not a critical requirement.

Example Walkthrough

Let's consider two small arrays to illustrate the solution approach:

- `nums = [4, 8, 12]`
- `divisors = [2, 3, 4]`

Now, we want to figure out which number in `divisors` can divide the most numbers in `nums` without leaving a remainder, and if there's a tie, we choose the smallest number.

1. We start with an initial maximum score `mx` set to 0 and the current answer `ans` set to the first element in `divisors`, which is 2.

2. We then iterate through each number in `divisors` to calculate its divisibility score:

   - When `div = 2`:
     - `4 % 2 == 0` (true, score is 1)
     - `8 % 2 == 0` (true, score is 2)
     - `12 % 2 == 0` (true, score is 3)
     - Total score for 2 is 3. `mx` is updated to 3, `ans` is updated to 2.
   - When `div = 3`:
     - `4 % 3 != 0` (false, score remains 0)
     - `8 % 3 != 0` (false, score remains 0)
     - `12 % 3 == 0` (true, score is 1)
     - Total score for 3 is 1, which is less than `mx`(3). We don't update `mx` or `ans`.
   - When `div = 4`:
     - `4 % 4 == 0` (true, score is 1)
     - `8 % 4 == 0` (true, score is 2)
     - `12 % 4 == 0` (true, score is 3)
     - Total score for 4 is also 3, equal to `mx`. However, since `ans` is smaller (2 < 4), we do not update `ans`.
3. After the iteration, the highest score is 3 with `ans` being 2. Since there's a tie between 2 and 4, we take the smaller number which is 2.

Thus, the final answer is 2, because it can divide all three numbers in `nums` with no remainder and is the smallest among the divisors with the highest divisibility score.

Python Solution

```python
class Solution:

    def max_div_score(self, nums: List[int], divisors: List[int]) -> int:
        # Initialize max_score with the first element in divisors and
        # max_count with zero to keep track of the highest score and count.
        max_score, max_count = divisors[0], 0

        # Loop through each divisor in the divisors list.
        for divisor in divisors:
            # Count how many numbers in nums are divisible by the current divisor.
            count_divisible = sum(num % divisor == 0 for num in nums)

            # If the current count is higher than the max_count found so far,
            # update max_count and max_score with the current count and divisor
            if max_count < count_divisible:
                max_count, max_score = count_divisible, divisor
            # If the current count is equal to the max_count, but the divisor is smaller
            # than the max_score, update the max_score to the current divisor.
            elif max_count == count_divisible and max_score > divisor:
                max_score = divisor

        # Return the divisor that has the highest divisibility score, giving preference
        # to the smallest divisor in case of a tie.
        return max_score

```

Java Solution

```java
class Solution {
    // Method to find the divisor with the highest divisibility score.
    // Divisibility score is defined by how many numbers in nums are divisible by the divisor.
    public int maxDivScore(int[] nums, int[] divisors) {
        // Initialize the answer with the first divisor, assuming it has the maximum score initially.
        int maxDivisor = divisors[0];
        // Initialize the maximum count of divisible numbers for any divisor.
        int maxCount = 0;
        // Iterate through all the divisors
        for (int divisor : divisors) {
            // Initialize count for the current divisor.
            int count = 0;
            // Count how many numbers in nums are divisible by this divisor.
            for (int num : nums) {
                if (num % divisor == 0) {
                    count++;
                }
            }
            // Update the maxDivisor and maxCount if the current divisor has a higher count.
            if (maxCount < count) {
                maxCount = count;
                maxDivisor = divisor;
            } else if (maxCount == count) {
                // If the current divisor has the same count, choose the smallest one.
                maxDivisor = Math.min(maxDivisor, divisor);
            }
        }
        // Return the divisor with the highest divisibility score.
        return maxDivisor;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // include necessary headers for std::min

class Solution {
public:
    // Function to find the divisor that has the maximum divisibility score.
    // The score for a divisor is defined as the number of elements in 'nums'
    // that are divisible by this divisor.
    int maxDivScore(vector<int>& nums, vector<int>& divisors) {
        // Initialize the answer with the first divisor as a starting point.
        int maxScoreDivisor = divisors[0];
        // Initialize the maximum count of divisible numbers to zero.
        int maxDivisibleCount = 0;

        // Iterate over each divisor.
        for (int divisor : divisors) {
            // Count how many numbers in 'nums' are divisible by 'divisor'.
            int divisibleCount = 0;
            for (int num : nums) {
                if (num % divisor == 0) {
                    ++divisibleCount;
                }
            }
            // If the current count is greater than the maximum found so far, update the maximum count
            // and change the answer to the current divisor.
            if (maxDivisibleCount < divisibleCount) {
                maxDivisibleCount = divisibleCount;
                maxScoreDivisor = divisor;
            // If the current count equals the maximum found so far, select the smaller divisor.
            } else if (maxDivisibleCount == divisibleCount) {
                maxScoreDivisor = std::min(maxScoreDivisor, divisor);
            }
        }

        // Return the divisor with the maximum divisibility score.
        // In case of a tie, the smallest such divisor is returned.
        return maxScoreDivisor;
    }
};

```

Typescript Solution

```typescript
// Function that calculates the maximum division score for a given array of numbers and divisors.
// The division score is defined by the number of times the numbers in the array can be evenly divided by the divisors.
// The function returns the divisor that gives the highest division score. In case of a tie, it returns the smallest divisor.
function maxDivScore(nums: number[], divisors: number[]): number {
    let bestDivisor: number = divisors[0]; // Initialize bestDivisor as the first divisor
    let maxDivisibleCount: number = 0; // Initialize maximum divisible count (division score) as 0

    // Loop through each divisor
    for (const divisor of divisors) {
        // Calculate the division score for the current divisor by reducing the nums array
        const divisibleCount = nums.reduce((count, num) => count + (num % divisor === 0 ? 1 : 0), 0);

        // Update the bestDivisor and maxDivisibleCount if this divisor has a higher division score
        if (maxDivisibleCount < divisibleCount) {
            maxDivisibleCount = divisibleCount;
            bestDivisor = divisor;
        } else if (maxDivisibleCount === divisibleCount && bestDivisor > divisor) {
            // If the division score is the same but the current divisor is smaller, update the bestDivisor
            bestDivisor = divisor;
        }
    }

    // Return the divisor with the highest division score (bestDivisor)
    return bestDivisor;
}

```

Time and Space Complexity

The given Python code defines the method `maxDivScore` which finds the divisor from the list `divisors` that maximizes the number of elements in `nums` that can be evenly divided by it. In case of ties, the smallest such divisor is returned.

Time Complexity:

The time complexity of the method can be determined by analyzing the for loop and the sum function within it. The for loop iterates once for each element in `divisors`. Inside the loop, the sum function iterates over each element in `nums`:

- Let `n` be the length of `nums`.
- Let `d` be the length of `divisors`.

For each divisor, we perform `n` modulus operations and `n` equality checks. Therefore, for all divisors, we perform this operation `d` times. The time complexity is `O(n*d)` because we have two nested loops: one iterating over divisors and the other over nums.

Space Complexity:

The space complexity is determined by the extra space used by the function beyond the input lists. In this case, the only extra space used are a few variables (`ans`, `mx`, `div`, and `cnt`) that remain constant regardless of input size. Hence, the space complexity is `O(1)` as there are no data structures used that scale with the size of the input.