2513. Minimize the Maximum of Two Arrays

Medium | Math | Binary Search | Number Theory | Leetcode Link

Problem Description

The essence of this problem is to fill two initially empty arrays arr1 and arr2 with distinct positive integers that are not divisible by respective divisors: divisor1 for arr1 and divisor2 for arr2. Additionally, we must ensure that no integer is present in both arrays. The goal is to figure out the minimum value of the maximum integer that can be present in either of the arrays while meeting the following conditions:

- `arr1` must contain `uniqueCnt1` distinct positive integers, none of which should be divisible by `divisor1`.
- `arr2` must contain `uniqueCnt2` distinct positive integers, none of which should be divisible by `divisor2`.
- There must not be any overlap between the integers in `arr1` and `arr2`.

The challenge thus revolves around selecting numbers carefully to minimize the highest number placed in either array, all the while respecting the given divisibility rules.

Intuition

The solution to this problem can be derived using a binary search on the range of possible maximum values for the arrays. Since a binary search requires a sorted range and certain conditions to find a target, we define the conditions based on understanding how many numbers will be allowed in each array up to a certain value `x`. The binary search begins with an initial range from 0 to an arbitrarily large number (in this case, 10^10), looking for the smallest `x` that satisfies our conditions.

Firstly, we need a function that checks if `x` can be the maximum possible integer in either array. I will define `f(x)`, which calculates the count of distinct non-divisible integers up to `x` for `arr1` and `arr2` separately, and a combined count for both arrays together. It does so by incorporating the floor division and modulo operations for each divisor as well as the lowest common multiple (LCM) of both divisors.

The intuition behind these calculations is that every `divisor1` consecutive numbers, `divisor1 - 1` will be admissible for `arr1` since one will be divisible and thus excluded. The same logic applies to `arr2` with `divisor2`. For common elements to be excluded from both arrays, we consider the LCM of both divisors, as any number divisible by the LCM would be divisible by both `divisor1` and `divisor2`.

The function `f(x)` checks whether, up to the number `x`, there are at least `uniqueCnt1` non-divisible numbers for `arr1`, `uniqueCnt2` for `arr2`, and a combined `uniqueCnt1 + uniqueCnt2` for both with overlap removed. If this condition holds, then `x` is a candidate for the minimum possible maximum integer.

The use of `bisect_left` in Python is how we do our binary search here. It essentially searches for the place where `True` would be inserted in the range to keep it sorted, as per the `f(x)` condition. This gives us the smallest value `x` that satisfies `f(x)`.

By initializing the binary search with a large enough range, we ensure that we can find a cut-off point where `f(x)` transitions from `False` to `True`, indicating that we've found the minimum possible maximum integer for the arrays.

Solution Approach

The implementation of the solution utilizes several important concepts from algorithms and mathematics:

1. **Least Common Multiple (LCM):** Before we start the binary search, we need to find the least common multiple of the two divisors. This is crucial as we need to exclude numbers divisible by both `divisor1` and `divisor2` when counting the numbers admissible in both arrays combined. As the LCM is not provided in the solution code, it must be implemented in helper functions or use external libraries.

2. **Counting Function (`f`):** We define a counting function `f(x)` that uses arithmetic to determine the count of distinct non-divisible integers up to a certain value `x` for each array, as well as combined. In more detail:
   - We calculate `cnt1` by determining how many groups of `divisor1` can fit into `x` (given by `x // divisor1`) and then adding the remaining numbers (given by `x % divisor1`). We exclude one number for each full group since one number will be divisible by `divisor1`.
   - Similarly, `cnt2` is calculated for `arr2` and `divisor2`.
   - For the combined count `cnt`, we apply the same logic using the LCM of the two divisors since we want to exclude numbers that are disallowed in both arrays.
3. **Binary Search Algorithm:** With `bisect_left`, we are efficiently conducting a binary search. The range `[0, 10**10]` is just a representation of an upper bound that we are confident is much higher than our target maximum integer. Binary search is applied by using `f(x)` as the key function for `bisect_left` which allows us to find the smallest number `x` for which `f(x)` returns `True`, i.e., the counting function indicates that both `arr1` and `arr2` have their required distinct positive integers.

4. **Binary Search with a Custom Condition (`f`):** Instead of looking for an actual value in a sorted list, as is typical with binary search, we are using the binary search to find the point at which a condition changes from `False` to `True`. This binary search is abstract in the sense that it operates on the truth value of `f(x)` over a range of numbers.

By combining the above techniques, the solution finds the minimum possible maximum integer with the least amount of computation necessary, avoiding the need to directly generate the arrays or to iterate over large ranges of numbers, which would be inefficient.

The most crucial part of the solution is ensuring that the key function `f(x)` accurately reflects the conditions we need for `arr1` and `arr2`. Once `f(x)` is correctly defined, `bisect_left` takes over to execute the binary search, taking advantage of the fact that `f(x)` will be `False` until it isn't—at which point we've found our minimum maximum.

It's worth noting that mathematical intuition is key in converting the problem into one that can be solved with binary search. Specifically, understanding how to calculate the number of admissible numbers given the constraints on divisibility and ensuring the arrays remain disjoint are the fundamental subproblems addressed by the counting function `f(x)`.

Example Walkthrough

Let's consider an example scenario to illustrate the solution approach:

Assume we have `divisor1 = 2`, `divisor2 = 3`, `uniqueCnt1 = 2`, and `uniqueCnt2 = 2`.

We need to create two arrays, `arr1` and `arr2`, such that:

- `arr1` contains two distinct integers not divisible by `2`.
- `arr2` contains two distinct integers not divisible by `3`.
- None of the integers are overlapping between `arr1` and `arr2`.

Step-by-Step Process

1. **Finding the Least Common Multiple (LCM):**
   - The LCM of `2` and `3` is `6`. We need this for counting common numbers that would be excluded from both arrays.
2. **Defining the Counting Function (`f`):**
   - Let's say we are checking if `x = 7` can be the maximum integer in either array.
   - For `divisor1 = 2`, up to `x = 7`, the numbers `1, 3, 5, 7` (four numbers) are not divisible by `2`.
   - For `divisor2 = 3`, up to `x = 7`, the numbers `1, 2, 4, 5, 7` (five numbers) are not divisible by `3`.
   - For numbers up to `x = 7` that are not divisible by `2` or `3` (using LCM of `6`), we get `1, 5, 7` (three numbers).
3. **Applying Binary Search with Custom Condition (`f`):**
   - Since we require at least `2` numbers for each array and to avoid overlap, we need to confirm if the combined total without overlap is at least `4`. For `x = 7`, we have `3` such numbers, so `f(7)` is `False`.
   - Using binary search, we increase `x` and check again. Let's say `x = 8`. We repeat the counting:
     - For `divisor1 = 2`, the numbers are `1, 3, 5, 7` (still four numbers).
     - For `divisor2 = 3`, the numbers are `1, 2, 4, 5, 7, 8` (six numbers).
     - Using the LCM of `6`, the numbers are `1, 5, 7` (still three numbers).
4. **Finding the Transition Point:**
   - For `x = 8`, `f(8)` is `False` as we still don't have at least `4` non-overlapping numbers. We proceed with the binary search.
   - Increment `x` and check again. At `x = 10`, we repeat the counting:
     - Non-divisible by `2`: `1, 3, 5, 7, 9` (five numbers).
     - Non-divisible by `3`: `1, 2, 4, 5, 7, 8, 10` (seven numbers).
     - Non-divisible by both (`LCM = 6`): `1, 5, 7, 11` (four numbers – `11` is now included since `x = 10`).
   - Since `f(10)` is `True` for the first time (we have at least `2` numbers for each array without overlap), the binary search concludes, and we consider `10` to be the smallest possible maximum integer that satisfies all conditions.

Therefore, for our example with the chosen parameters, the minimum possible maximum integer that can be placed in either array while satisfying all conditions is `10`. This example demonstrates the method through which we can utilize a binary search to efficiently solve for the required parameters.

Python Solution

```python
from math import gcd
from bisect import bisect_left

class Solution:
    def minimizeSet(self, divisor1: int, divisor2: int, unique_cnt1: int, unique_cnt2: int) -> int:
        # Auxiliary function to determine the lowest common multiple (LCM) of two numbers
        def lcm(a, b):
            return a * b // gcd(a, b)

        # Define the condition function that will be used with binary search
        def condition(x: int) -> bool:
            # Calculate the number of integers not divisible by divisor1 up to x
            nondiv_cnt1 = x // divisor1 * (divisor1 - 1) + x % divisor1
            # Calculate the number of integers not divisible by divisor2 up to x
            nondiv_cnt2 = x // divisor2 * (divisor2 - 1) + x % divisor2
            # Calculate the number of integers not divisible by the LCM of divisor1 and divisor2 up to x
            nondiv_cnt_total = x // lcm_value * (lcm_value - 1) + x % lcm_value
            # Check if the counted non-divisible integers meet or exceed the unique count requirements
            return (
                nondiv_cnt1 >= unique_cnt1 and
                nondiv_cnt2 >= unique_cnt2 and
                nondiv_cnt_total >= unique_cnt1 + unique_cnt2
            )

        # Calculate the least common multiple of the two divisors
        lcm_value = lcm(divisor1, divisor2)

        # Perform binary search to find the smallest integer x for which condition(x) is True
        # The search space is assumed to be up to 10^10
        result = bisect_left(range(10**10), True, key=condition)

        return result

```

Java Solution

```java
class Solution {

    // Method to minimize the set size based on provided divisors and unique count requirements
    public int minimizeSet(int divisor1, int divisor2, int uniqueCount1, int uniqueCount2) {
        // Calculate the least common multiple of the two divisors
        long leastCommonMultiple = calculateLCM(divisor1, divisor2);

        // Initialize search space for the minimum set size
        long left = 1, right = 10000000000L;

        // Binary search to find the minimum set size
        while (left < right) {
            long mid = (left + right) >> 1;  // Calculate the midpoint

            // Calculate the number of unique elements for divisor1 and divisor2,
            // and both combined in the range [1, mid]
            long count1 = mid / divisor1 * (divisor1 - 1) + mid % divisor1;
            long count2 = mid / divisor2 * (divisor2 - 1) + mid % divisor2;
            long combinedCount = mid / leastCommonMultiple * (leastCommonMultiple - 1) + mid % leastCommonMultiple;

            // Check if the current midpoint meets the unique count requirements
            if (count1 >= uniqueCount1 && count2 >= uniqueCount2 && combinedCount >= uniqueCount1 + uniqueCount2) {
                right = mid; // Midpoint is valid, try to find smaller number
            } else {
                left = mid + 1; // Midpoint is too small, increase lower bound
            }
        }

        // Cast to int as per the problem constraints, the resulting set size is safe to be within integer range
        return (int) left;
    }

    // Helper method to calculate the least common multiple (LCM) of two numbers
    private long calculateLCM(int a, int b) {
        return (long) a * b / calculateGCD(a, b);
    }

    // Helper method to calculate the greatest common divisor (GCD) of two numbers using Euclid's algorithm
    private int calculateGCD(int a, int b) {
        return b == 0 ? a : calculateGCD(b, a % b);
    }
}

```

C++ Solution

```cpp
#include <algorithm> // For std::lcm

class Solution {
public:
    int minimizeSet(int divisor1, int divisor2, int uniqueCount1, int uniqueCount2) {
        long left = 1, right = 1e10; // Define search space for binary search
        long leastCommonMultiple = std::lcm(static_cast<long>(divisor1), static_cast<long>(divisor2)); // Compute LCM

        // Perform a binary search to find the smallest number that satisfies the conditions.
        while (left < right) {
            long mid = (left + right) / 2; // Find the midpoint of the current search space

            // Calculate how many unique numbers exist for 'mid' when considering divisor1 and divisor2 separately
            long count1 = mid / divisor1 * (divisor1 - 1) + mid % divisor1;
            long count2 = mid / divisor2 * (divisor2 - 1) + mid % divisor2;

            // Calculate the unique count for both divisors when they are combined
            long combinedCount = mid / leastCommonMultiple * (leastCommonMultiple - 1) + mid % leastCommonMultiple;

            // Check if 'mid' satisfies all of the minimum requirements for unique numbers for both divisors
            if (count1 >= uniqueCount1 && count2 >= uniqueCount2 && combinedCount >= uniqueCount1 + uniqueCount2) {
                // If so, narrow the search to the lower half, including 'mid'
                right = mid;
            } else {
                // Otherwise, narrow the search to the upper half, excluding 'mid'
                left = mid + 1;
            }
        }

        // After exiting the loop, 'left' will be the minimum number that satisfies all requirements.
        return left;
    }
};

```

Typescript Solution

```typescript
function lcm(a: number, b: number): number {
    // Helper function to find the least common multiple (LCM) using the greatest common divisor (GCD) method
    let gcd = function gcd(a: number, b: number): number {
        return b ? gcd(b, a % b) : a;
    };
    // Formula for LCM: |a * b| / gcd(a, b)
    return Math.abs(a * b) / gcd(a, b);
}

function minimizeSet(divisor1: number, divisor2: number, uniqueCount1: number, uniqueCount2: number): number {
    let left: number = 1, right: number = 1e10; // Define search space for binary search
    let leastCommonMultiple: number = lcm(divisor1, divisor2); // Compute LCM using the helper function

    // Perform a binary search to find the smallest number that satisfies the conditions.
    while (left < right) {
        let mid: number = Math.floor((left + right) / 2); // Find the midpoint of the current search space

        // Calculate how many unique numbers exist for 'mid' when considering divisor1 and divisor2 separately
        let count1: number = Math.floor(mid / divisor1) * (divisor1 - 1) + mid % divisor1;
        let count2: number = Math.floor(mid / divisor2) * (divisor2 - 1) + mid % divisor2;

        // Calculate the unique count for both divisors when they are combined
        let combinedCount: number = Math.floor(mid / leastCommonMultiple) * (leastCommonMultiple - 1) + mid % leastCommonMultiple;

        // Check if 'mid' satisfies all of the minimum requirements for unique numbers for both divisors
        if (count1 >= uniqueCount1 && count2 >= uniqueCount2 && combinedCount >= uniqueCount1 + uniqueCount2) {
            // If so, narrow the search to the lower half, including 'mid'
            right = mid;
        } else {
            // Otherwise, narrow the search to the upper half, excluding 'mid'
            left = mid + 1;
        }
    }

    // After exiting the loop, 'left' will be the minimum number that satisfies all requirements.
    return left;
}

```

Time and Space Complexity

The given Python code aims to find the minimum number `x` that satisfies certain divisibility conditions. The code is expected to use the `bisect_left` function from the `bisect` module to perform a binary search for the smallest value of `x` where function `f(x)` returns `True`. Unfortunately, the provided code contains errors and it's incomplete (there's a reference to an undefined variable `divisor` inside the `f(x)` function). Assuming these issues were fixed and the `lcm` function (least common multiple) and `bisect_left` were correctly implemented, we can proceed with the time and space complexity analysis.

Time complexity:

1. `bisect_left`: The binary search implemented by `bisect_left` has a time complexity of `O(log N)`, where `N` is the size of the range. In this case, `N` is set to `10**10`, which is a constant, so the `bisect_left` call alone would have a complexity of `O(log 10**10)` which simplifies to `O(log 10)` due to the base not being specified in the Big O notation.

2. Function `f`: This function is called by `bisect_left` at each step of the binary search. The operations inside function `f` are performed in constant time (basic arithmetic operations) and do not depend on the size of the input.

Combining the two above, we get `O(log 10)` multiplied by the time complexity of `f`, which is `O(1)`, resulting in `O(log 10)`. However, the improper references in the `f(x)` function can alter this analysis if, for example, computing the `lcm` has a non-constant complexity.

Space complexity:

1. `bisect_left`: The space complexity of the binary search is `O(1)` since it doesn't allocate any additional space that grows with the input.

2. LCM computation: If the `lcm` is computed each time `f(x)` is called, and the computation of `lcm` is not in `O(1)` space, this would affect the space complexity. If the `lcm` is computed once and used multiple times with a constant space complexity, the space complexity will remain `O(1)`.

3. Function `f`: Since the function `f` only uses a fixed number of variables and does not use any data structures that scale with the size of the input, its space complexity is `O(1)`.

Considering all the above, the overall space complexity of the code is `O(1)`, provided the computationally-intensive tasks such as finding the least common multiple have constant space complexity and are computed outside of the `f(x)` function or are otherwise cached.

It's important to note that the presence of the `range(10**10)` in the `bisect_left` function does not impact space complexity since `range` in Python 3 creates a range object that does not generate all numbers in memory but computes them on-the-fly.