1201. Ugly Number III

Medium Math Binary Search Number Theory Leetcode Link

Problem Description

In this problem, you are asked to find the `n`th ugly number. An ugly number is defined as a positive integer that is divisible by any of the given three integers `a`, `b`, or `c`. The integers `a`, `b`, and `c` are the only prime factors considered for ugliness. You need to return the smallest `n`th number that meets the condition of being an ugly number.

Intuition

The intuition behind the solution is to use binary search to efficiently find the `n`th ugly number. A straightforward way to solve it would be to iterate over numbers starting from 1, and count how many of them are divisible by `a`, `b`, or `c` until we reach the `n`th ugly number. However, that would be very inefficient for large `n`.

Instead, we can binary search for the answer because the `n`th ugly number is within a known range. The smallest ugly number is 1, and by setting an upper bound (like 2 * 10^9), we can use binary search to narrow down the number that is exactly the `n`th ugly number.

Concretely, we calculate `mid` in our search range as the potential `n`th ugly number, and check how many numbers less than or equal to `mid` are divisible by `a`, `b`, or `c`. To avoid counting numbers more than once that are divisible by any two or all three of `a`, `b`, and `c`, we use the inclusion-exclusion principle. This involves adding and subtracting counts of multiples, like adding the count of numbers divisible by `a` and by `b` but then subtracting the count of numbers divisible by both `a` and `b` to remove the duplicates.

Here's how we apply inclusion-exclusion in this context:

- We count numbers divisible by `a`, `b`, and `c` separately.
- We subtract the numbers that are divisible by the least common multiple (lcm) of (`a` and `b`), (`b` and `c`), and (`a` and `c`) because these numbers were counted more than once.
- We add the numbers divisible by the lcm of (`a`, `b`, and `c`) since those numbers were subtracted one time too many.

Once we have the count of ugly numbers less than or equal to `mid`, we compare it with `n`. If our count is equal to or greater than `n`, the `n`th ugly number is less than or equal to `mid`, and we continue searching to the left. Otherwise, we continue searching to the right.

The process repeats, narrowing the search range until the left and right boundaries converge, at which point `l` or `r` will be the `n`th ugly number.

Solution Approach

The solution uses a binary search algorithm to find the `n`th ugly number. Binary search is a widely used algorithm for finding an item from a sorted list or in scenarios like this one where the condition is monotonically increasing or decreasing.

Here is a step-by-step explanation of the implementation:

1. Calculate the least common multiple (LCM) for all pairs and all three numbers: `a`, `b`, and `c`. The LCMs are necessary for the inclusion-exclusion principle to avoid counting duplicates.

```
ab = lcm(a, b)
bc = lcm(b, c)
ac = lcm(a, c)
abc = lcm(a, bc)  # lcm for all three numbers
```

2. Initialize the binary search boundaries `l` and `r`. The left boundary (`l`) starts at 1, as the smallest ugly number is 1. The right boundary (`r`) is set to a high value, ensuring that the `n`th ugly number lies within this range.

```
l, r = 1, 2 * 10**9
```

3. The main binary search loop continues until `l < r`, meaning we haven't yet narrowed down to a single potential option for the `n`th ugly number.

4. Calculate the middle point (`mid`) between `l` and `r`, which we will test to see if it has exactly `n` ugly numbers less than or equal to it.

```
mid = (l + r) >> 1  # equivalent to (l + r) // 2 but often faster
```

5. Apply the inclusion-exclusion principle to count ugly numbers less than or equal to `mid`:

```
count = (
    mid // a +
    mid // b +
    mid // c -
    mid // ab -
    mid // bc -
    mid // ac +
    mid // abc
)
```

6. Compare `count` with `n`:

   - If `count` is greater than or equal to `n`, it means there are at least `n` ugly numbers less than or equal to `mid`, and we need to continue searching in the left half including `mid`:

   ```
   r = mid
   ```

   - If `count` is less than `n`, it means there are fewer than `n` ugly numbers up to `mid`, and we need to consider larger numbers by searching in the right half:

   ```
   l = mid + 1
   ```

7. The loop continues, narrowing the range until `l` equals `r`. At this point, `l` (or `r`) is the `n`th ugly number, which we return:

```
return l
```

This approach is efficient because it reduces the problem space exponentially with each iteration of the binary search rather than iterating sequentially through all numbers, which wouldn't be feasible for large values of `n`.

Note: This solution assumes the existence of a function `lcm` which computes the least common multiple of the given numbers. Implementing or using a library function for LCM calculations is beyond the scope of the explanation but is critical for the solution to work correctly.

Example Walkthrough

Let's walk through a small example by applying the solution approach outlined above. Suppose we have `a = 2`, `b = 3`, and `c = 5`, and we want to find the 5th ugly number.

1. Compute the least common multiples (LCM) for the pairs and all three numbers:

```
ab = lcm(a, b)  # lcm(2, 3) = 6
bc = lcm(b, c)  # lcm(3, 5) = 15
ac = lcm(a, c)  # lcm(2, 5) = 10
abc = lcm(a, bc)  # lcm(2, 15) = 30
```

2. Initialize binary search bounds:

```
l, r = 1, 2 * 10**9
```

3. Enter binary search loop:

4. Calculate the midpoint to test:

```
mid = (l + r) >> 1  # Let's assume the midpoint turns out to be 10 for our first iteration
```

5. Apply inclusion-exclusion to count the ugly numbers less than or equal to `mid`:

```
count = (
    10 // 2 +  # Numbers divisible by 2
    10 // 3 +  # Numbers divisible by 3
    10 // 5 -  # Numbers divisible by 5
    10 // 6 -  # Numbers divisible by both 2 and 3
    10 // 15 -  # Numbers divisible by both 3 and 5
    10 // 10 +  # Numbers divisible by both 2 and 5
    10 // 30   # Numbers divisible by 2, 3, and 5
)
# This gives us 5 + 3 + 2 - 1 - 0 - 1 + 0 = 8.
```

6. Since `count` (8) is less than `n` (5), we need to consider larger numbers and move `l` right:

```
l = mid + 1  # Our new `l` is now 11
```

7. Continue the binary search:

   After several iterations, we will find that when `mid = 10`, the count is equal to 8, and our `l` was updated to 11. This process continues until `l` equals `r`.

   Ultimately, we will find that `l = r = 10` because that is the smallest number for which there are exactly 5 or more numbers that are divisible by `a`, `b`, or `c`. So the 5th ugly number is 10.

This example illustrates how the solution uses binary search and the inclusion-exclusion principle to efficiently find the target ugly number without iterating over every single number up to `n`.

Python Solution

```python
from math import gcd

# The lcm function computes the least common multiple of two or more numbers
def lcm(x, y, z=None):
    def lcm_two(a, b):
        return a * b // gcd(a, b)

    if z: # if three numbers are provided
        return lcm_two(lcm_two(x, y), z)
    else: # if only two numbers are provided
        return lcm_two(x, y)

class Solution:
    def nthUglyNumber(self, n: int, a: int, b: int, c: int) -> int:
        # Computing least common multiples of pairs and triplet of a, b, c
        ab_lcm = lcm(a, b)
        bc_lcm = lcm(b, c)
        ac_lcm = lcm(a, c)
        abc_lcm = lcm(a, b, c)

        # Binary search range - start with 1, end with a value large enough to ensure the nth ugly number is within the range
        left, right = 1, 2 * 10**9

        # Binary search to find the nth ugly number
        while left < right:
            mid = (left + right) // 2

            # Counting the number of ugly numbers up to `mid`
            # by adding the count for each prime and subtracting the count for
            # their least common multiples to avoid double counting.
            count = (mid // a + mid // b + mid // c
                     - mid // ab_lcm
                     - mid // bc_lcm
                     - mid // ac_lcm
                     + mid // abc_lcm)

            # If the current count is at least `n`, move `right` to mid
            # indicating that the nth ugly number is lesser or equal to mid.
            if count >= n:
                right = mid
            else:
                # Otherwise, move `left` just above mid as the nth ugly
                # number must be greater than mid.
                left = mid + 1

        # Since 'left' will end up at the smallest number
        # where the count is at least n, it is our answer.
        return left

```

Java Solution

```java
class Solution {
    // Method to find the nth ugly number that is divisible by a, b, or c
    public int nthUglyNumber(int n, int a, int b, int c) {
        // Find the least common multiple (LCM) of the pairs (a, b), (b, c), (a, c),
        // and the three numbers (a, b, c)
        long lcmAB = lcm(a, b);
        long lcmBC = lcm(b, c);
        long lcmAC = lcm(a, c);
        long lcmABC = lcm(lcmAB, c);

        // Use binary search in the range [1, 2000000000] to find the nth ugly number
        long left = 1, right = 2000000000;
        while (left < right) {
            long mid = (left + right) >> 1; // Calculate the midpoint of the range
            // Check if the count of numbers divisible by a, b, or c up to mid is >= n
            if (count(mid, a, b, c, lcmAB, lcmBC, lcmAC, lcmABC) >= n) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        // The left pointer will point to the nth ugly number
        return (int) left;
    }

    // Helper method to count numbers divisible by a, b, or c up to a limit
    private long count(long mid, int a, int b, int c, long lcmAB, long lcmBC, long lcmAC, long lcmABC) {
        // Calculate the inclusive count of divisible numbers by a, b, and c separately
        long countA = mid / a;
        long countB = mid / b;
        long countC = mid / c;

        // Subtract the counts for pairs of (a, b), (b, c), (a, c) to exclude double counted numbers
        long countAB = mid / lcmAB;
        long countBC = mid / lcmBC;
        long countAC = mid / lcmAC;

        // Add the count for (a, b, c) to include numbers that are divisible by all three
        long countABC = mid / lcmABC;

        // Apply inclusion-exclusion principle and return the result
        return countA + countB + countC - countAB - countBC - countAC + countABC;
    }

    // Helper method to find the greatest common divisor (GCD) of two numbers
    private long gcd(long a, long b) {
        return b == 0 ? a : gcd(b, a % b);
    }

    // Helper method to find the least common multiple (LCM) of two numbers
    private long lcm(long a, long b) {
        return a * b / gcd(a, b);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Utility function to calculate the least common multiple (LCM) of two numbers
    long long lcm(long long a, long long b) {
        return a / gcd(a, b) * b;  // Calculate the product and then divide by the greatest common divisor (GCD)
    }

    // Utility function to calculate the greatest common divisor (GCD) of two numbers
    long long gcd(long long a, long long b) {
        if (b == 0) return a; // Base case: if the second number is 0, return the first number
        return gcd(b, a % b); // Recursive case: return the GCD of b and the remainder of a divided by b
    }

    // Function to find the nth ugly number that is divisible by at least one of the given numbers a, b, or c
    int nthUglyNumber(int n, int a, int b, int c) {
        // Compute pairwise least common multiples
        long long lcmAB = lcm(a, b);
        long long lcmBC = lcm(b, c);
        long long lcmAC = lcm(a, c);
        // Compute the least common multiple of all three numbers
        long long lcmABC = lcm(lcmAB, c);

        long long left = 1, right = 2000000000;
        // Binary search to find the smallest integer that has at least n multiples of a, b, or c
        while (left < right) {
            long long mid = (left + right) / 2;
            // Count the number of multiples of a, b, c, and subtract the multiples of lcmAB, lcmBC, lcmAC
            // Add the multiples of lcmABC to correct for over-subtraction
            long long count = mid / a + mid / b + mid / c
                              - mid / lcmAB - mid / lcmBC - mid / lcmAC
                              + mid / lcmABC;
            if (count >= n) {  // If there are at least n ugly numbers up to mid, search the left half
                right = mid;
            } else {           // Otherwise, search the right half
                left = mid + 1;
            }
        }
        // The left index now points to the nth ugly number
        return left;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the gcd of two numbers using the Euclidean algorithm.
function gcd(a: bigint, b: bigint): bigint {
    while (b !== 0n) {
        let temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Function to calculate the lcm of two numbers based on the gcd.
function lcm(a: bigint, b: bigint): bigint {
    return (a / gcd(a, b)) * b;
}

// Function to find the nth ugly number that is divisible by either a, b, or c.
function nthUglyNumber(n: number, a: number, b: number, c: number): number {
    // Convert inputs to bigint for proper lcm and gcd calculations.
    const bigA = BigInt(a);
    const bigB = BigInt(b);
    const bigC = BigInt(c);

    // Calculate least common multiples for combinations of a, b, and c.
    const abLCM = lcm(bigA, bigB);
    const bcLCM = lcm(bigB, bigC);
    const acLCM = lcm(bigA, bigC);
    const abcLCM = lcm(bigA, bcLCM);

    // Binary search to find the nth ugly number.
    let low = 1n;
    let high = BigInt(2e9);
    while (low < high) {
        const mid = (low + high) >> 1n;
        // Calculate the count of numbers divisible by a, b, or c up to `mid`.
        const count =
            mid / bigA +
            mid / bigB +
            mid / bigC -
            mid / abLCM -
            mid / bcLCM -
            mid / acLCM +
            mid / abcLCM;
        // Narrow down search space based on `count` compared to `n`.
        if (count >= BigInt(n)) {
            high = mid;
        } else {
            low = mid + 1n;
        }
    }

    // Return the nth ugly number as a Number type.
    return Number(low);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code primarily depends upon the binary search used to find the `nth` ugly number. The binary search operates in the range between `1` and `2 * 10**9`. In each iteration of the binary search, we perform a constant number of operations including the computation of `lcm` (Least Common Multiple), divisions, and some arithmetic operations.

The binary search halves the search range with each iteration, which results in a time complexity of `O(log(maxVal))`, where `maxVal` is `2 * 10**9`.

Additionally, the calculations of `lcm(a, b)`, `lcm(b, c)`, `lcm(a, c)`, and `lcm(a, b, c)` are executed only once and outside of the loop, assuming that the `lcm` function has a time complexity of `O(log(min(x, y)))` for two numbers `x` and `y`, where `lcm(a, b, c)` uses the pairwise method to calculate LCM of three numbers which can be expressed as `lcm(a, lcm(b, c))`.

Thus, the overall time complexity is `O(4 * log(min(a, b, c)) + log(maxVal))`. Because `log(min(a, b, c))` is negligible compared to `log(maxVal)`, the practical time complexity simplifies to `O(log(maxVal))`, which can be stated as `O(log(2 * 10**9))` for this specific problem.

Space Complexity

The space complexity of the algorithm is `O(1)`. No additional space that scales with the input size is used. The space is used for a constant number of integer variables (`ab`, `bc`, `ac`, `abc`, `l`, `r`, `mid`, and the LCM calculations), regardless of the size of the input `n`, `a`, `b`, or `c`. Thus, the space complexity remains constant.