878. Nth Magical Number

**Binary Search** 

# **Problem Description**

**Math** 

Hard

The problem is about finding the nth number that is divisible by at least one of the two given numbers, a or b. These kinds of numbers are referred to as magical numbers. For instance, if a is 2 and b is 3, then the first few magical numbers would be 2, 3, 4 (divisible by 2), 6 (divisible by both 2 and 3), 8 (divisible by 2), and so on. The function nthMagicalNumber should return the nth

magical number in this sequence. However, due to the potentially large size of the output, we don't want the exact value of the nth magical number but instead its remainder when divided by 10^9 + 7, which is a commonly used large prime number in modular arithmetic problems to prevent

integer overflow errors. Intuition

### When solving this problem, we need to consider a vital property of Icm (Least Common Multiple) of a and b, which will further help us understand the distribution of magical numbers. Specifically, every multiple of the *lcm* of a and b is a magical number

To arrive at the solution, we observe that: 1. Each multiple of a is a magical number.

which is also the least common period in which all the magical numbers repeat.

3. Some numbers are multiples of both a and b, specifically multiples of the lcm of a and b.

2. Each multiple of **b** is a magical number.

- We use a binary search to find the smallest number x where the count of magical numbers up to x is at least n. To count the
- number of magical numbers up to x, we sum up x // a (the count of multiples of a up to x), x // b (the count of multiples of b up to x), and subtract x // c (to account for over-counting the numbers that are multiples of both a and b, hence multiples

sequence up to r = (a + b) \* n (an upper bound for the nth magical number) to bisect\_left with a custom key function that calculates the aforementioned count. Then, it takes the result x of the binary search, which is essentially the nth magical number, and applies the modulo to maintain the result within the bounds required by the problem statement. **Solution Approach** 

The code exploits the bisect\_left function from the bisect module to perform the binary search efficiently. It passes a virtual

The implementation of the solution includes the following concepts: Binary Search: A classical algorithm to find the position of an element in a sorted array in logarithmic time complexity. Here, it is used to find the smallest number x such that there are at least n magical numbers less than or equal to x.

Least Common Multiple (LCM): Used to calculate the period at which the multiples of a and b repeat. This is critical, as we

# must adjust the counts to avoid counting any number more than once if it is divisible by both a and b.

of their lcm c).

**Efficient Counting:** Using integer division to count multiples up to x for a, b, and their lcm.

• The mod variable is set to 10\*\*9 + 7 to prepare for the final output to be given modulo this large prime.

Looking at the implementation step-by-step:

**Modular Arithmetic:** This is used to avoid large number overflows by taking the modulus of the result with 10\*\*9 + 7.

product of a and b divided by their greatest common divisor (gcd)). • The variable r defines an upper bound for the nth magical number, which can be set as the sum of a and b multiplied by n. This is based on the assumption that magical numbers will appear frequently enough that we do not need to check any number greater than this bound.

• The magic happens with bisect\_left(range(r), x=n, key=lambda x: x // a + x // b - x // c). This call is a bit unusual because it uses

• The variable c calculates the lcm of a and b using a built-in lcm function (not shown in the snippet given, but can be implemented as the

using much memory. The key function calculates the count of magical numbers up to x, and bisect\_left will find the position where this count reaches at least n. • The % mod at the end of the function ensures the result is within the required modulus to prevent overflow and match the problem's constraints.

bisect\_left on a range, leveraging the fact that Python ranges are lazy (they do not instantiate all values) and thus can be enormous without

properties and algorithmic patterns to do so within the computing constraints. **Example Walkthrough** 

In summary, the code uses a binary search to efficiently find the nth magical number, leveraging knowledge of mathematical

Following the given solution approach:

Calculate the least common multiple (LCM) of a and b. The LCM of 2 and 3 is 6, because 6 is the smallest number that is

We set an upper bound for the binary search. For n magical numbers, a rough upper bound would be (a+b) \* n. In this case,

do this by checking the count of multiples of a and b subtracted by the count of their LCM up to certain x. For each

To illustrate the solution approach with an example, let's choose a = 2, b = 3, and we want to find the 5th magical number.

divisible by both 2 and 3. Therefore, every 6 steps, the pattern of magical numbers will repeat.

candidate number x, we calculate the count as x // a + x // b - x // c, where c is the LCM.

(2+3)\*5 = 25. Therefore, we can start our binary search from 1 to 25. Use a binary search to find the smallest number x such that there are at least n magical numbers less than or equal to x. We

## Suppose we test x = 10. To count the magic numbers up to 10, we find:

 Multiples of 6 up to 10: 10 // 6 = 1 (6). Now we sum the counts of multiples of a and b then subtract the count of LCM of a and b: 5 + 3 - 1 = 7. This means

there are 7 magical numbers up to 10. Since we are looking for the 5th magical number, 10 is an upper bound (too high) and

Summing up and subtracting gives us 4 + 2 - 1 = 5. There are 5 magical numbers up to 8, so 8 is our candidate for the 5th

Finally, we apply the modulo operation. Since the number is lower than 10\*\*\*9 + 7, the modulus doesn't change the number.

we need to search lower. Adjust the binary search range accordingly. Let's try x = 8. The calculations give us:

- Repeat the binary search process until the range is narrowed down to a single number. Since we already found that there are 5 magical numbers up to 8 and we are looking for the 5th magical number, we can conclude that 8 is the 5th magical number.
- This example demonstrates the binary search-based approach in a clear and step-by-step manner, following the key concepts outlined in the initial solution approach.

So the 5th magical number given a = 2 and b = 3 is 8.

# Function to calculate the least common multiple (LCM).

# Use binary search to find the nth magical number.

# Return the nth magical number modulo MOD.

// Function to find the nth magical number

// Initialize binary search bounds

while (lowerBound < upperBound) {</pre>

upperBound = mid;

lowerBound = mid + 1;

const MOD: number = 1e9 + 7; // Use uppercase for constants

// Calculate the least common multiple of a and b

// Return the nth magical number modulo MOD

let lcmAB: number = lcm(a, b);

while (left < right) {</pre>

else

from math import gcd

return left % MOD;

from bisect import bisect\_left

MOD = 10\*\*9 + 7

r = (a + b) \* n

return nth\_magical % MOD

Time and Space Complexity

complexity of O(log(min(a, b)).

**Time Complexity** 

function nthMagicalNumber(n: number, a: number, b: number): number {

// Perform a binary search to find the nth magical number

right = mid; // Too high, adjust the right boundary

left = mid + 1; // Too low, adjust the left boundary

def nth magical number(self, n: int, a: int, b: int) -> int:

# Use binary search to find the nth magical number.

# Return the nth magical number modulo MOD.

# The key function calculates the number of magical numbers

# Define the modulo for the result as per the problem statement.

# Calculate an upper boundary for the search space (this is not tight).

# less than or equal to 'x' by summing the number of multiples of 'a' and 'b',

# then subtracting the number of multiples of 'lcm of a b' to avoid double counting.

nth\_magical = bisect\_left(range(r), n, key=lambda x:  $x // a + x // b - x // lcm_of_a_b$ )

let left: number = 0, right: number = (a + b) \* n; // Use `number` type for the range

let mid: number = Math.floor((left + right) / 2); // Calculate the middle value

if (Math.floor(mid / a) + Math.floor(mid / b) - Math.floor(mid / lcmAB) >= n)

// Check if the mid value has n or more multiples of a or b, considering overlaps

// let result = nthMagicalNumber(n, a, b); // Invoke the function with desired values of n, a, and b

long upperBound = (long) (a + b) \* n;

// Middle of the current bounds

long mid = (lowerBound + upperBound) >>> 1;

public int nthMagicalNumber(int n, int a, int b) {

int leastCommonMultiple = a \* b / gcd(a, b);

# The key function calculates the number of magical numbers

# less than or equal to 'x' by summing the number of multiples of 'a' and 'b',

# then subtracting the number of multiples of 'lcm of a b' to avoid double counting.

// Calculate least common multiple of a and b using gcd (Greatest Common Divisor)

// Binary search to find the smallest number that is the nth magical number

// Check if the mid number is a valid magical number by counting

// If count is equal or greater than n, we shrink the right bound

if (mid / a + mid / b - mid / leastCommonMultiple >= n) {

// Otherwise, we need to look for a larger number

// all multiples of a and b up to mid, minus those of their lcm to avoid double-counting

nth\_magical = bisect\_left(range(r), n, key=lambda x:  $x // a + x // b - x // lcm_of_a_b$ )

Multiples of 2 up to 10: 10 // 2 = 5 (2, 4, 6, 8, 10).

 $\circ$  Multiples of 3 up to 10: 10 // 3 = 3 (3, 6, 9).

 $\circ$  Multiples of 2 up to 8: 8 // 2 = 4 (2, 4, 6, 8).

 $\circ$  Multiples of 3 up to 8: 8 // 3 = 2 (3, 6).

Multiples of 6 up to 8: 8 // 6 = 1 (6).

magical number.

**Python** 

Java

from math import gcd

from bisect import bisect\_left

def lcm(x, y):

Solution Implementation

return x \* y // gcd(x, y)

class Solution: def nth magical number(self, n: int, a: int, b: int) -> int: # Define the modulo for the result as per the problem statement. MOD = 10\*\*9 + 7

# Calculate the least common multiple of a and b.  $lcm_of_a_b = lcm(a, b)$ # Calculate an upper boundary for the search space (this is not tight). r = (a + b) \* n

### class Solution { // Define modulus constant for the problem private static final int MOD = (int) 1e9 + 7;

long lowerBound = 0;

} else {

return nth\_magical % MOD

```
// After binary search, lower bound is our nth magical number
        // Return it modulo MOD
        return (int) (lowerBound % MOD);
    // Utility method to calculate the greatest common divisor using Euclid's algorithm
    private int qcd(int a, int b) {
        // Recursive calculation of gcd
        return b == 0 ? a : gcd(b, a % b);
C++
#include <numeric> // Include necessary library for std::lcm
class Solution {
public:
    const int MOD = 1e9 + 7; // Use uppercase for constants
    int nthMagicalNumber(int n, int a, int b) {
        int lcm ab = std::lcm(a, b); // Calculate the least common multiple of a and b
        long long left = 0, right = 1LL * (a + b) * n; // Use long long for large ranges
        // Perform a binary search to find the nth magical number
        while (left < right) {</pre>
            long long mid = (left + right) / 2; // Calculate the middle value
            // Check if the mid value has n or more multiples of a or b, considering overlaps
            if (mid / a + mid / b - mid / lcm ab >= n)
                right = mid; // Too high, decrease right
            else
                left = mid + 1; // Too low, increase left
        return left % MOD; // Return the nth magical number modulo MOD
};
TypeScript
function lcm(a: number, b: number): number {
    // Calculate the greatest common divisor (GCD) using Euclid's algorithm
    function gcd(a: number, b: number): number {
        while (b !== 0) {
            let t = b;
            b = a % b;
            a = t;
        return a;
    // Calculate the least common multiple (LCM) using the relationship between GCD and LCM
    return (a * b) / gcd(a, b);
```

#### # Function to calculate the least common multiple (LCM). def lcm(x, v): return x \* y // gcd(x, y)# Calculate the least common multiple of a and b. $lcm_of_a_b = lcm(a, b)$

class Solution:

// Usage

To analyze the time complexity, let's consider the operations performed: Calculating the least common multiple (lcm) of a and b. The time complexity of finding the lcm depends on the algorithm

The given code snippet finds the nth magical number where a magical number is a number divisible by either a or b.

The bisect\_left function performs a binary search. The parameter range(r) has a length of about Nx(a+b)/lcm(a,b). The binary search algorithm has a time complexity of O(log n), but since it's not searching through a simple list but using a

used. If the Euclidean algorithm is used to find the gcd (greatest common divisor), then this part of the code has a time

lambda function to generate values on the fly, the key function is calculated for every middle element in each iteration of the

binary search. This, therefore, results in a time complexity of  $O(\log(Nx(a+b)/lcm(a,b)) * \log(min(a,b))$  because each

key calculation is O(log(min(a, b))). The modulo operation is constant time, 0(1). Given these points, the overall time complexity is  $O(\log(Nx(a+b)/lcm(a,b)) * \log(min(a,b))$ .

The code uses a constant amount of extra space: 1. Storing variables such as mod, c, and r requires O(1) space.

**Space Complexity** 

2. The bisect\_left function does not create additional data structures that scale with the input size, since the range does not actually create a list but is an iterator.

The lambda function within bisect\_left is stateless and does not consume more memory proportional to the size of the input. Therefore, the overall space complexity of the provided function is 0(1) (constant space).