1808. Maximize Number of Nice Divisors

Hard

Recursion

Problem Description

Math

The given problem involves constructing a positive integer n with certain constraints involving prime factors and "nice" divisors. A key term in the problem is "prime factors," which are the prime numbers that multiply together to give the number n. Another key term is "nice divisors," which are specific divisors of n that are also divisible by all of n's prime factors. The constraints of the problem are as follows:

We need to maximize the number of nice divisors that n can have.

assumption that we want to only use prime numbers as the base.

Here's the reasoning behind each step of the solution:

3, modulo $10^9 + 7$, as it maximizes the product.

• The integer n must have at most a given number of prime factors (primeFactors).

The goal is to return the number of nice divisors of the constructed number n. However, because this number has the potential to

be very large, the answer should be given modulo 10^9 + 7.

To reach the solution, we need to understand that to maximize the number of nice divisors, n should be composed in a way that leverages the power of 3 to the greatest extent possible. This is based on the mathematical fact that for a fixed sum of exponents, the product of repeated multiplication of the base number is maximized when the base is the number 3, under the

• When primeFactors leaves a remainder of 1 when divided by 3 (primeFactors % 3 == 1), it's better to take a factor of 4 out (since 4 is 2^2, and 22 > 31) and then raise 3 to the power of (primeFactors // 3) - 1.

• If primeFactors is less than 4, we should just return primeFactors since we can't do better than multiplying the prime factors directly.

• When primeFactors is a multiple of 3 (primeFactors % 3 == 0), we can simply return 3 raised to the power of the quotient of primeFactors and

• When primeFactors leaves a remainder of 2 when divided by 3 (primeFactors % 3 == 2), we can multiply by 2 once (using one of the prime factors) and then raise 3 to the largest power possible with the remaining factors.

Python's pow function is used to efficiently compute the large exponents modulo 10^9 + 7, which is necessary due to the size of

- the numbers involved and the need to return the result within the limitations of computable integer ranges. Solution Approach
 - divisors, n should be comprised mostly of the prime number 3. This conclusion is based on the optimization principle that given a fixed sum of natural numbers, their product is maximized when the numbers are as close to each other as possible — and for prime factors, 3 is the smallest prime that enables us to get the most 'prime factors' within our constraint.

The Python code provided is a direct implementation of the insight that for any integer n, to maximize the number of "nice/prime"

1. When primeFactors is less than 4:

best solution is to just multiply prime factors 2 and/or 3 (the smallest primes) to get the maximum number of nice divisors, which would be equal to primeFactors itself. There is no room for using powers greater than one since that would reduce the number of distinct prime factors we can use. if primeFactors < 4:</pre> return primeFactors

The second condition checks if primeFactors is perfectly divisible by 3. If so, then n is best constructed by multiplying 3 with

itself primeFactors / 3 times (raising 3 to the power of primeFactors / 3). This is because we're making the most of all available

The third condition accounts for when primeFactors divided by 3 leaves a remainder of 1. In this case, taking all of them as 3's

would leave us with a prime factor of 1, which is not utilizable. Instead, we take a '4' out (which is 2 * 2, using two prime factors

to still keep n as a product of primes) and multiply it by the largest power of 3 we can form with the remaining factors

The first if condition in the code checks if primeFactors is less than 4. Due to there being so few prime factors, it's clear that the

2. When primeFactors is a multiple of 3:

prime factors to get the largest n with the most number of nice divisors.

(primeFactors - 4), which will be (primeFactors / 3) - 1 threes.

Let's walk through the implementation and the thought process for each condition in the code:

return pow(3, primeFactors // 3, mod) % mod 3. When primeFactors gives a remainder of 1 upon division by 3:

statement.

Example Walkthrough

number of nice divisors here is 2.

if primeFactors % 3 == 0:

if primeFactors % 3 == 1: return 4 * pow(3, primeFactors // 3 - 1, mod) % mod

4. When primeFactors gives a remainder of 2 upon division by 3: The final condition covers when there is a remainder of 2. This scenario suggests we can multiply one '2' with the maximum power of 3 possible with the remaining number of available prime factors. if primeFactors % 3 == 2: return 2 * pow(3, primeFactors // 3, mod) % mod

In all these mathematical operations, the % mod ensures that we always stay within the bounds of the specified modulus (10^9 +

7), which prevents integer overflows in environments where this might be an issue and adheres to the constraints of the problem

Overall, the solution doesn't require the use of complex data structures or sophisticated algorithms — it relies principally on

mathematical insight and the efficient computation of large powers modulo a number, which is a common operation in number theory and modular arithmetic.

def max_nice_divisors(self, prime_factors: int) -> int:

return pow(3, prime_factors // 3, MOD)

return (2 * pow(3, prime_factors // 3, MOD)) % MOD

public int maxNiceDivisors(int primeFactors) {

if (primeFactors < 4) {</pre>

return (int) result;

int maxNiceDivisors(int primeFactors) {

return primeFactors;

long long result = 1;

while(exponent > 0) {

if (primeFactors % 3 == 0) {

if (primeFactors % 3 == 1) {

if (primeFactors < 4) {</pre>

const int MOD = 1e9 + 7;

// Calculate the maximum product of the given number of prime factors

// If the number of prime factors is less than 4, return it as is

auto quickPower = [&](long long base, long long exponent) -> int {

result = (result * base) % MOD;

base = (base * base) % MOD;

return quickPower(3, primeFactors / 3);

// then use the (primeFactors - 4) / 3 threes.

return static_cast<int>(result);

// Square the base and move to the next bit

// Define the modulo value as constant for easy changes and readability

exponent >>= 1; // equivalent to dividing exponent by 2

// If primeFactors is a multiple of 3, simply return 3^(primeFactors/3)

// Define a power function that computes a^n % mod using binary exponentiation

if (exponent & 1) { // If the current bit is set, multiply the result with base

// If primeFactors leaves a remainder of 1 when divided by 3, use one 2 and one 3 to make a four,

return static_cast<int>((quickPower(3, primeFactors / 3 - 1) * 4L) % MOD);

// If primeFactors leaves a remainder of 2 when divided by 3, pair one two with the threes

C++

public:

class Solution {

};

return primeFactors;

if prime_factors % 3 == 0:

if prime_factors % 3 == 1:

Define the modulo value as a constant according to the problem statement

If there is a remainder of 1 when the number of prime factors is divided by 3,

we use a group of 4 (2 * 2) and the rest as groups of 3 to maximize the product.

The first part comes from taking a single '3' out and combining it with the '1'

If there is a remainder of 2 when the number of prime factors is divided by 3,

to make a '4', and use the remaining (prime_factors // 3 - 1) groups of 3.

we can simply use one group of 2 with the maximum number of groups of 3.

// Function to compute the maximum product of primeFactors with the largest sum.

// If the total number of prime factors is less than 4, return the number itself.

This optimizes the product of divisors, making them as 'nice' as possible.

return (4 * pow(3, (prime_factors // 3) - 1, MOD)) % MOD

integer n using these prime factors such that we maximize the number of nice divisors.

Firstly, we need to decide how to distribute these 5 prime factors. Since 5 is not a multiple of 3 and leaves a remainder of 2 when divided by 3 (primeFactors % 3 == 2), the third condition in our solution approach applies here. According to the condition, it's best to use a factor of 2 once and then use the prime number 3 with the remaining factors. Therefore, we will have $3^1 * 2^1 = 3$ * 2 = 6 as our n.

After constructing the integer, we calculate the number of nice divisors of n (which is 6 in this case). The divisors of 6 are 1, 2, 3,

and 6. However, only 3 and 6 are "nice" because they are divisible by all of n's prime factors (which are 2 and 3). Therefore, the

To calculate the power of 3 used in constructing n, we need to use the pow function in Python, as the numbers can be very large.

(because we can only use one '3' after using a '2' to construct n), so our use of the pow function would be pow(3, 1, 10**9 + 7).

The final result would be calculated as 2 * pow(3, 1, 10***9 + 7), which equals 2 * 3 = 6. Since we're returning the number of

The pow function takes three arguments: the base, the exponent, and the modulus. For our example, we have an exponent of 1

Since 3 to the power of 1 is just 3, and this is less than the modulus, the pow function wouldn't change the number.

Let's use a small example to illustrate the solution approach. Suppose we have primeFactors = 5. We want to construct a positive

nice divisors and not n itself, and the result fits within normal computational ranges, we don't need the modulus for this small example. But in the actual solution approach, the modulus would be used because we're often dealing with much larger numbers where the result could easily exceed normal computational ranges. Solution Implementation

MOD = 10**9 + 7# If the number of prime factors is less than 4, the maximum product is the number itself if prime_factors < 4:</pre> return prime_factors # If the number of prime factors is divisible by 3, the product of equal-sized groups of 3 # yields the maximum product. Use modular exponentiation to find 3 to the power of the number # of groups (prime_factors // 3), and take the modulo.

class Solution { // Define the modulo constant for all operations. private final int MODULO = (int) 1e9 + 7;

Java

Python

class Solution:

```
// If the total number of prime factors divided by 3 leaves no remainder,
   // return 3 raised to the power of primeFactors/3, modulo MODULO.
   if (primeFactors % 3 == 0) {
        return quickPower(3, primeFactors / 3);
   // If the remainder is 1 when divided by 3, calculate power for primeFactors/3 - 1
   // and multiply the result by 4, then take modulo MODULO.
   if (primeFactors % 3 == 1) {
        return (int) (4L * quickPower(3, primeFactors / 3 - 1) % MODULO);
   // If the remainder is 2, multiply 2 with 3 raised to the power of primeFactors/3,
    // then take modulo MODULO.
   return 2 * quickPower(3, primeFactors / 3) % MODULO;
// Helper function to perform quick exponentiation with modulo.
private int quickPower(long base, long expo) {
    long result = 1;
   // Loop until the exponent becomes zero.
   while (expo > 0) {
       // If the current bit in the binary representation of the exponent is 1,
       // multiply result with base and take modulo.
        if ((expo & 1) == 1) {
            result = result * base % MODULO;
       // Square the base and take modulo at each iteration.
        base = base * base % MODULO;
        // Right shift the exponent by 1 (equivalent to dividing by 2).
        expo >>= 1;
   // Cast the result back to int before returning.
```

```
// to maximize the product.
       return static_cast<int>((quickPower(3, primeFactors / 3) * 2) % MOD);
};
TypeScript
/**
* Calculates the maximum "nice" divisors for a given number of prime factors.
* A "nice" divisor of a number is defined as the number which only contains
 * prime numbers that divide the original number.
 * For example, given 4 prime factors, the product with maximum "nice" divisors is 4 itself, which divides into 2 * 2.
 * @param {number} primeFactors - The number of prime factors
 * @returns {number} The maximum number of "nice" divisors
const maxNiceDivisors = (primeFactors: number): number => {
   // Any number less than 4 is its own maximum
    if (primeFactors < 4) {</pre>
       return primeFactors;
    // Define the modulo value to handle large numbers
    const MOD: number = 1e9 + 7;
    /**
    * Performs exponentiation by squaring, modulo MOD.
    * This is needed to efficiently compute large powers under a modulo.
     * @param {number} base - The base of the exponentiation
     * @param {number} exponent - The exponent
    * @returns {number} Result of (base^exponent) % MOD
    const quickPower = (base: number, exponent: number): number => {
        let result: number = 1;
        for (; exponent; exponent >>= 1) {
            if (exponent & 1) {
                result = Number((BigInt(result) * BigInt(base)) % BigInt(MOD));
            base = Number((BigInt(base) * BigInt(base)) % BigInt(MOD));
       return result;
    };
    // Determine the division of prime factors by 3 to find the major section of divisors
    const k: number = Math.floor(primeFactors / 3);
    // If exactly divisible by 3, return 3 to the power k modulo MOD
    if (primeFactors % 3 === 0) {
       return quickPower(3, k);
   // If remainder is 1 when divided by 3, then use one 2 and decrement k to get the rest as 3's
   if (primeFactors % 3 === 1) {
       return (4 * quickPower(3, k - 1)) % MOD;
   // If remainder is 2 when divided by 3, then use one 2 and keep k to get the rest as 3's
```

return (4 * pow(3, (prime_factors // 3) - 1, MOD)) % MOD # If there is a remainder of 2 when the number of prime factors is divided by 3, # we can simply use one group of 2 with the maximum number of groups of 3. # This optimizes the product of divisors, making them as 'nice' as possible. return (2 * pow(3, prime_factors // 3, MOD)) % MOD

Time and Space Complexity

return (2 * quickPower(3, k)) % MOD;

def max_nice_divisors(self, prime_factors: int) -> int:

of groups (prime_factors // 3), and take the modulo.

return pow(3, prime_factors // 3, MOD)

Define the modulo value as a constant according to the problem statement

If the number of prime factors is less than 4, the maximum product is the number itself

If the number of prime factors is divisible by 3, the product of equal-sized groups of 3

If there is a remainder of 1 when the number of prime factors is divided by 3,

we use a group of 4 (2 * 2) and the rest as groups of 3 to maximize the product.

The first part comes from taking a single '3' out and combining it with the '1'

to make a '4', and use the remaining (prime_factors // 3 - 1) groups of 3.

size. Therefore, the space usage remains constant no matter the value of primeFactors.

yields the maximum product. Use modular exponentiation to find 3 to the power of the number

};

class Solution:

MOD = 10**9 + 7

if prime_factors < 4:</pre>

return prime_factors

if prime_factors % 3 == 0:

if prime_factors % 3 == 1:

The time complexity of the given code is $O(\log n)$ where n is the input primeFactors. This is because the only non-constant operation that depends on the size of the input is the pow function, which calculates x^y % mod in logarithmic time relative to y.

Time Complexity

pow function with the exponent primeFactors // 3 or primeFactors // 3 - 1, both of which are proportional to the input size. **Space Complexity** The space complexity of the code is 0(1). The algorithm uses a fixed amount of space (a few integer variables like mod and

temporary variables for storing the result of the pow function). It does not allocate any additional space that grows with the input

In all three conditions within the function (primeFactors % 3 == 0, primeFactors % 3 == 1, and the else case), the code calls the