1969. Minimum Non-Zero Product of the Array Elements

Medium Greedy Recursion Math Leetcode Link

Problem Description

The LeetCode problem presents a scenario where you have a positive integer `p`, and asks for the operations to be performed on an array of integers `nums`, containing the numbers 1 through `2^p - 1` in binary form. The goal is to find the minimum non-zero product of the array elements after any number of a specific operation: you can choose any two elements `x` and `y` from `nums` and swap a corresponding bit between the two (a bit at a certain position in `x` gets swapped with the bit in the same position in `y`).

The key challenge is to determine how to perform these operations to minimize the product of all numbers in the array, and then return the minimal product modulo (`10^9 + 7`). Note that the product must be calculated before applying the modulo.

Intuition

To arrive at the solution for this problem, we need to consider the properties of binary numbers and the effect of swapping bits. Since we are dealing with numbers from 1 to `2^p - 1`, we know that in binary, these numbers will look like a sequence from `1` to `111...111` (with `p-1` ones).

If we attempt to perform operations to minimize the product, we should aim to make the numbers as close to each other as possible because the product of any set of numbers is minimized when the numbers are equal (or as close to each other as possible). This means we should try to lower the value of the maximum numbers and increase the value of the minimum numbers. However, the smallest number cannot be changed because it's `1` and has no zeros to swap with.

Considering this, the maximum product reduction happens when we only modify the most significant bits of the largest numbers in the array. The maximum number `2^p - 1` can't be changed since all of its bits are `1`s, but the second-largest number, `2^p - 2`, has exactly one `0` bit, and it can be swapped with `1` bits of the numbers just below it. Luckily, since the array includes all numbers in the range, we have plenty of `1`s to swap with.

Through this process, the numbers `2^p - 2`, `2^p - 3`, ..., all become `2^p - 1`, all except the least significant bits. We need to perform this bit-swap operation `2^(p-1) - 1` times because there are `2^(p-1)` numbers that can be reduced to one less than their maximum value, and we don't need to consider the smallest number `1` itself.

Then the product of all numbers can be calculated as the constant value `2^p - 1` multiplied by `(2^p - 2)^{2^(p-1) - 1}`. However, calculating such large exponentiation directly is impractical due to possible integer overflow and inefficiency. Therefore, we use the `pow` function with three arguments in Python that calculates the power and applies the modulo at the same time, effectively managing large numbers efficiently. This operation is modulo `10^9 + 7`, a large prime number often used to prevent integer overflow in competitive programming.

Solution Approach

The solution to this problem leverages modular exponentiation to compute the product of array elements after performing the optimal bit-swap operations. Let's dive into the algorithm, and the data structures used along with the pattern that the solution capitalizes on:

1. **Observation of Pattern**:
   - The array begins with all possible `p`-bit numbers.
   - The product is initially the product of all these numbers.
   - The goal is to swap bits to minimize this product.
2. **Optimal Bit Swaps**:
   - The optimal bit swaps will make as many numbers as possible equal to the largest number in the range, `2^p - 1`, which has all bits set to 1. Swapping bits will not affect this number.
   - Every number except `1` and `2^p - 1` can be increased to `2^p - 1` by swapping bits with a larger number that has a corresponding `1` bit.
3. **Mathematical Insight**:
   - Every number from `2` to `2^p - 2` can be paired with a unique `1` bit from numbers larger than it.
   - Because the numbers `2` to `2^(p-1) - 1` are all less than `2^(p-1)`, they can be made into `2^p - 1` by swapping with the larger half of the array, which will have a complementary `1` in every position where the smaller half has a `0`.
   - The minimal product is then `((2^p - 1) * (2^p - 2)^(2^(p-1) - 1)) % (10^9 + 7)`
4. **Algorithm**:
   - Calculate `(2^p - 1)`, the largest number which will be a multiple in the final product.
   - Raise `(2^p - 2)` to the power of `(2^(p-1) - 1)` using modular exponentiation.
   - Multiply the two results and apply modulo operation to get the final result.
5. **Data Structures**:
   - No complex data structures are needed since the calculation involves only integers and the use of exponentiation and modulo operations provided by the language's standard library.
6. **Use of Python's `pow` Function**:
   - The `pow` function in Python is used to efficiently compute large powers under modulo. Its signature is `pow(base, exp, mod)`.
   - This function is crucial because calculating `2^p - 2` raised to `2^(p - 1) - 1` would result in astronomically large numbers that can't be handled by standard integer operations.
   - Instead, `pow` calculates each step of the exponentiation process modulo `10^9 + 7`, keeping the intermediate results manageable and avoiding overflow.
7. **Code**:

```python
class Solution:
    def minNonZeroProduct(self, p: int) -> int:
        mod = 10**9 + 7
        return (2**p - 1) * pow(2**p - 2, 2 ** (p - 1) - 1, mod) % mod
```

   - The code simply applies the formula derived from the insight and mathematical operations with modular arithmetic using the `pow` function for efficient calculation.

In summary, the implementation uses mathematical analysis and properties of binary numbers to find an efficient formula to compute the answer. It then utilizes the `pow` function in Python for modular exponentiation, keeping all intermediate calculations within an acceptable range to avoid overflow and efficiently compute the final result. The simplicity of the code belies the more involved mathematical reasoning that underpins the solution.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose `p = 3`, which means our array `nums` consists of binary numbers from `1` (`0001` in binary) to `2^3 - 1` (`0111` in binary).

For clarity, let's list out all the numbers (in binary and decimal) from 1 to 2^3 - 1:

- 0001 (1)
- 0010 (2)
- 0011 (3)
- 0100 (4)
- 0101 (5)
- 0110 (6)
- 0111 (7)

We are trying to minimize the product of these numbers by swapping bits according to the rules.

1. We cannot do anything with the smallest number (`0001`) because it has no extra `1` bits to swap.
2. The largest number `0111` doesn't require any swaps since all its bits are already `1`.

Our goal is to increase the smaller numbers and make them as close to `0111` as possible. We observe that:

- The number `0110` (6) can have its `0` bit swapped with a `1` from another number to become `0111`.
- Similarly, `0101` (5) can swap its `0` with a `1` from a larger number to become `0111`.
- The same goes for `0100` (4), `0011` (3), and `0010` (2).

Notice that after performing these swaps, our list of numbers becomes:

- 0001 (1)
- 0111 (7)
- 0111 (7)
- 0111 (7)
- 0111 (7)
- 0111 (7)
- 0111 (7)

So, essentially, we have one `0001` and six `0111`s. The product of these numbers is `0001 * 0111^6`.

Using the algorithm described in the solution approach:

- The largest number `(2^p - 1)` is `0111` (which is `7`).
- We will raise the second largest number `(2^p - 2)` which is `0110` (which is `6`) to the power of `(2^(p-1) - 1)`, which in our case is `2^(3-1) - 1 = 2^2 - 1 = 3`.

The actual computation is:

```
(2^p - 1) * (2^p - 2)^(2^(p-1) - 1) % (10^9 + 7)
= 7 * 6^3 % 1000000007
= 7 * 216 % 1000000007
= 1512 % 1000000007
= 1512
```

In Python, using the `pow` function, our code becomes:

```python
mod = 10**9 + 7
result = (2**3 - 1) * pow(2**3 - 2, 2 ** (3 - 1) - 1, mod) % mod
# This should evaluate to 1512
```

So, after the optimal bit swaps, the minimum non-zero product of the array elements for `p = 3` is `1512`, subject to a modulo of `10^9 + 7`.

Python Solution

```python
class Solution:
    def minNonZeroProduct(self, p: int) -> int:
        # Define the modulo value since it will be used multiple times in the calculation.
        modulo = 10**9 + 7

        # Compute the maximum value that can be generated with p bits, which is 2**p - 1.
        # This maximum value is part of the final product.
        max_val = 2**p - 1

        # Compute the base for exponentiation which is one less than the maximum value.
        base = max_val - 1

        # Compute the exponent, which is half the quantity of numbers with p bits,
        # minus one for the non-zero constraint, which is 2**(p-1) - 1.
        exponent = 2**(p - 1) - 1

        # Calculate the power with modulo operation to prevent large number computations.
        # The pow function here uses the third argument as the modulo.
        power_mod = pow(base, exponent, modulo)

        # Compute the final result as the product of the maximum value
        # and the power_mod, modulo the defined modulo.
        result = (max_val * power_mod) % modulo

        # Return the final result.
        return result

```

Java Solution

```java
class Solution {

    // This method calculates the minimum non-zero product of the elements of the
    // array created by 'p' given features.
    public int minNonZeroProduct(int p) {
        final int MOD = (int) 1e9 + 7; // Define the modulo as per the problem statement.

        // Calculate the base value 'a' - it's 2^p - 1 modulo MOD.
        long baseValueA = ((1L << p) - 1) % MOD;

        // Calculate the power value 'b' - it requires using a helper method which
        // computes (2^p - 2) raised to the power of (2^(p-1)-1) modulo MOD.
        long powerValueB = qpow(((1L << p) - 2) % MOD, (1L << (p - 1)) - 1, MOD);

        // Return the minimum product modulo MOD.
        return (int) (baseValueA * powerValueB % MOD);
    }

    // This helper method calculates a^b modulo 'mod' using the fast exponentiation method.
    private long qpow(long base, long exponent, int mod) {
        long result = 1;
        while (exponent > 0) {
            // If the current bit is set, multiply the result by the current base modulo 'mod'.
            if ((exponent & 1) == 1) {
                result = (result * base) % mod;
            }

            // Square the base for the next iteration and take modulo 'mod'.
            base = (base * base) % mod;

            // Right shift exponent by 1 (divide by 2) for the next iteration.
            exponent >>= 1;
        }
        return result;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minNonZeroProduct(int p) {
        // Define 'long long' as 'll' for easier use
        using ll = long long;
        // Define the modulus value for the problem (1e9 + 7 is a common choice for mod operations in programming contests)
        const int MOD = 1e9 + 7;

        // Define a quick power (qpow) function using the fast exponentiation method
        auto quickPower = [MOD](ll base, ll exponent) {
            ll result = 1; // Initialize result to 1 (the identity for multiplication)
            for (; exponent; exponent >>= 1) { // Loop until all bits of exponent are processed
                if (exponent & 1) { // If the current bit is set
                    result = (result * base) % MOD; // Multiply with the current base and take modulo
                }
                base = (base * base) % MOD; // Square the base and take modulo at each step
            }
            return result; // Return the result of raising base to the power of exponent modulo MOD
        };

        // Calculate a as the last number in the sequence modulo MOD
        ll maxValModulo = ((1LL << p) - 1) % MOD;
        // Use the quickPower function to compute b, which is the power of all sequence numbers
        // except the last one, raised to a certain exponent and then modulo MOD.
        ll powerOfPrecedingElements = quickPower(((1LL << p) - 2) % MOD, (1LL << (p - 1)) - 1);
        // Calculate the final answer by multiplying maxValModulo with powerOfPrecedingElements and then modulo MOD
        return maxValModulo * powerOfPrecedingElements % MOD;
    }
};

```

Typescript Solution

```typescript
function minNonZeroProduct(p: number): number {
    // Define a constant mod for the modulus operation
    const MOD = BigInt(1e9 + 7);

    /**
     * Quick exponentiation function to calculate (a^n) % MOD
     *
     * @param base The base number a as bigint
     * @param exponent The exponent n as bigint
     * @returns (base^exponent) % MOD as bigint
     */
    const quickPow = (base: bigint, exponent: bigint): bigint => {
        let result = BigInt(1);
        while (exponent) { // Loop as long as exponent is not zero
            if (exponent & BigInt(1)) { // If the exponent is odd
                result = (result * base) % MOD;
            }
            base = (base * base) % MOD; // Square the base
            exponent >>= BigInt(1); // Halve the exponent
        }
        return result;
    };

    // Calculate the maximum value of the last nonzero element in the array
    const lastNonZeroElement = (2n ** BigInt(p) - 1n) % MOD;
    // Use quickPow to calculate the product of all but the last element
    const productOfOtherElements = quickPow((2n ** BigInt(p) - 2n) % MOD,
                                            2n ** (BigInt(p) - 1n) - 1n);

    // Return the product of lastNonZeroElement and productOfOtherElements modulo MOD as number
    return Number((lastNonZeroElement * productOfOtherElements) % MOD);
}

```

Time and Space Complexity

The given Python code calculates the minimum non-zero product of the pixel values of an `p`-bit image where each pixel can have `2^p` possible values, under modulo `10**9 + 7`. It involves the power and modulo operations.

Time complexity:

The time complexity of the code largely depends on the `pow` function which is used with three arguments: the base (`2**p - 2`), the exponent (`2 ** (p - 1) - 1`), and the modulo (`mod`). The optimized Modular Exponentiation implemented in Python computes results in `O(log(exp))` time, where `exp` is the exponent.

Therefore, the time complexity of the pow function in the code is `O(log(2 ** (p - 1) - 1))`. Since the exponent is `2 ** (p - 1) - 1`, its logarithm is `O(p)`. So the time complexity of the modular exponentiation step is `O(p)`.

We also need to calculate `2**p - 1` and `2**p - 2`, and these can be done in `O(p)` time as well.

Overall, considering these operations together, the total time complexity is `O(p)`.

Space complexity:

The space complexity of the code is `O(1)` since it uses only a constant amount of extra space: the variables for intermediate results and the module `mod`, and no complex data structures or recursive call stacks that scale with the input size.