2939. Maximum Xor Product

Medium Greedy Bit Manipulation Math

Problem Description

Given three integers `a`, `b`, and `n`, the task is to find the maximum value of `(a XOR x) * (b XOR x)` where `x` can be any integer from `0` to $2^n - 1$ (both inclusive). Here, `XOR` refers to the bitwise exclusive OR operation. To ensure the return value is within a manageable size, we are asked to return the result after taking the modulo of $10^9 + 7$.

We know that the XOR operation can flip bits in the binary representation of a number, depending on the bits of another number it is being XORed with. The catch here is to intelligently select the value of `x` that, when XORed with `a` and `b`, results in the maximum possible product `(a XOR x) * (b XOR x)`.

Intuition

The solution follows a greedy approach combined with bitwise operations. The intuition behind the solution lies in understanding how the XOR operation affects the bits of the operands and consequently, the product of the two results. The XOR operation with `1` flips a bit, while XOR with `0` leaves it unchanged. The goal is to carefully set each bit of `x` to either `0` or `1` in such a way that the resulting product is maximized.

The process starts with ensuring that the most significant bits (those beyond `n`) of `a` and `b` are preserved as they will always contribute to the larger value in the product when XORed with `0`. These parts are denoted as `ax` and `bx` respectively.

Then, for each bit from the most significant bit `n-1` to the least significant bit `0`, we decide whether to set the corresponding bit in `ax` and `bx` to `1`. If the current bits of `a` and `b` are equal, setting the bit to `1` in both `ax` and `bx` does not change their product, but as these bits are less significant than the preserved parts, it is optimal to set them to `1`.

However, when the bits of `a` and `b` are different, we have a decision to make. To increase the product, we want to flip the bit of the smaller operand among `ax` and `bx`. This way, we are increasing the smaller number without decreasing the other, thus maximizing the product.

By iterating from the most significant bit to the least significant bit in this manner and updating `ax` and `bx` accordingly, we ensure that we arrive at the highest possible product. Once we have determined the optimal `ax` and `bx`, the final result is their product, modulo $10^9 + 7$.

Solution Approach

The solution approach makes clever use of bitwise operations to arrive at the maximum product. Here's how the algorithm proceeds:

1. The first step is to preserve the bits of `a` and `b` that are above the `n-th` bit since those are not affected by XOR with any `x` in the given range `0` to $2^n - 1$.
   - This is done by right-shifting `a` and `b` by `n` and then left-shifting back by `n`. The result is stored in two new variables `ax` and `bx`. This effectively zeroes out the bits in positions `0` to `n-1` since right-shifting followed by left-shifting with the same number of bits fills the vacated bits with zeroes.
2. The core of the algorithm then iterates over each bit position from `n-1` down to `0`:
   - For each bit position `i`, we extract the bit value of `a` and `b` at position `i` using the bitwise AND operation with `1 << i`.
   - Next, we determine if the current bits of `a` and `b`, let's call them `x` and `y`, are the same or not. If they are the same, then regardless of what value `i` takes for bit position `i`, the product will be maximized if the bit position `i` is set to `1` for both `ax` and `bx`.
3. However, if `x` and `y` are not equal, we compare the values of `ax` and `bx`. We only want to flip the bit of the smaller value between the two:
   - If `ax` is greater than `bx`, then we increase `bx` by setting its `i-th` bit to ensure `bx` is increased.
   - Conversely, if `bx` is greater or equal to `ax`, then we increase `ax` by setting its `i-th` bit.
   - This operation intends to make `bx` and `ax` more equal in value, as the maximum product of two numbers happens when they are closest to each other in value.
4. After determining the optimal `ax` and `bx`, we calculate the product and take the modulo with $10^9 + 7$ to get the final result:
   - `ax * bx % mod`

Each of these steps complies with the greedy paradigm of algorithm design, as we are making the locally optimal choice of setting bit `i` in hope of finding the global maximum product. By considering bit positions in a descending order, we prioritize the influence of higher-order bits on the product, which is key since they contribute significantly more to the value than lower-order bits.

In terms of data structures, no additional structures are necessary; the solution uses basic integer variables to store the modified versions of `a` and `b`, as well as to handle intermediate values as we iterate over the bits.

By breaking down the bitwise operations and decision-making process, we can see exactly how the algorithm maximizes the final product and ensure that it conforms to the constraints and requirements described in the problem.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Assume we are given `a = 22`, `b = 27`, and `n = 5`. The binary representations of `a` and `b` within 5 bits are `10110` for `a` and `11011` for `b`. We aim to find an `x` such that `(a XOR x) * (b XOR x)` is maximized, where `x` ranges from `0` to $2^5 - 1$, i.e., `0` to `31`.

Step 1: Since `n` equals `5`, we do not need to alter `a` and `b` for this particular example because any `x` within the range will not affect bits beyond the 5th bit.

Step 2: We start iterating from the 4th bit (most significant bit) to the 0th bit (least significant bit):

For the 4th bit (2^4 or `16` place):

- `a` at 4th bit is `1`, `b` at 4th bit is `1`. Both are the same. We choose `x` to have `0` at this bit to keep the bits intact and maintain the high value.

For the 3rd bit (2^3 or `8` place):

- `a` at 3rd bit is `0`, `b` at 3rd bit is `1`. They differ. We want to flip the bit for the smaller operand of the two (for our purposes, since their bits beyond `n` are zero, `ax` and `bx` correspond to `a` and `b`). So `x` will have `1` at this bit to flip `a`'s 3rd bit and make it larger.

For the 2nd bit (2^2 or `4` place):

- Both `a` and `b` have `1` at this bit. No changes needed; we choose `x` to have `0` at this bit.

For the 1st bit (2^1 or `2` place):

- `a` at 1st bit is `1`, `b` at 1st bit is `0`. They are different again, and `b` is smaller at the 1st bit. We choose `x` to have `1` at this bit to flip `b`'s 1st bit.

For the 0th bit (2^0 or `1` place):

- `a` at 0th bit is `0`, `b` at 0th bit is `1`. We flip `x`'s 0th bit by choosing `x` to have `1` at this bit.

Based on the decisions above, for each bit, `x` will be `01111` in binary or `15` in decimal.

Step 3: Now we compute `(a XOR x) * (b XOR x)`:

- `(a XOR x)` in binary: `10110 XOR 01111` equals `11001` (decimal `25`)
- `(b XOR x)` in binary: `11011 XOR 01111` equals `10100` (decimal `20`)

Step 4: The product is `25 * 20 = 500`. The modulo operation is not needed in this specific case because the product is already less than $10^9 + 7$.

Thus, the maximum value for `(a XOR x) * (b XOR x)` is `500` for `x = 15`.

This example demonstrates the step-by-step process described in the solution approach by carefully constructing the optimal value of `x` using bitwise operations to maximize the product.

Solution Implementation

Python

```
class Solution:
    def maximum_xor_product(self, num1: int, num2: int, num_bits: int) -> int:
        # Define the modulo for taking the result % mod
        MODULUS = 10**9 + 7

        # Initialize variables to store the maximum XOR values based on the most significant bits
        xor_num1, xor_num2 = (num1 >> num_bits) << num_bits, (num2 >> num_bits) << num_bits

        # Iterate over each bit, from most significant bit to least significant bit
        for i in range(num_bits - 1, -1, -1):
            # Extract the current bit of num1 and num2
            bit_num1 = (num1 >> i) & 1
            bit_num2 = (num2 >> i) & 1

            # If the current bits are equal, set the current bit in both xor_num1 and xor_num2
            if bit_num1 == bit_num2:
                xor_num1 |= 1 << i
                xor_num2 |= 1 << i
            # If xor_num1 is greater, set the current bit in xor_num2
            elif xor_num1 > xor_num2:
                xor_num2 |= 1 << i
            # Otherwise, set the current bit in xor_num1
            else:
                xor_num1 |= 1 << i

        # Return the product of xor_num1 and xor_num2 modulo MODULUS
        return (xor_num1 * xor_num2) % MODULUS
```

Java

```
class Solution {
    public int maximumXorProduct(long num1, long num2, int bits) {
        final int MODULUS = (int) 1e9 + 7; // The modulus value for the result

        // These two variables represent num1 and num2 with the last 'bits' bits set to 0
        long adjustedNum1 = (num1 >> bits) << bits;
        long adjustedNum2 = (num2 >> bits) << bits;

        // Iterate over each bit from 'bits-1' down to 0
        for (int i = bits - 1; i >= 0; --i) {
            long bitNum1 = (num1 >> i) & 1; // Get the ith bit of num1
            long bitNum2 = (num2 >> i) & 1; // Get the ith bit of num2

            // If ith bits of num1 and num2 are equal, set ith bits of adjustedNum1 and adjustedNum2 to 1
            if (bitNum1 == bitNum2) {
                adjustedNum1 |= 1L << i;
                adjustedNum2 |= 1L << i;
            }
            // Otherwise, if adjustedNum1 is less than adjustedNum2, set ith bit of adjustedNum1 to 1
            else if (adjustedNum1 < adjustedNum2) {
                adjustedNum1 |= 1L << i;
            }
            // Otherwise, set ith bit of adjustedNum2 to 1
            else {
                adjustedNum2 |= 1L << i;
            }
        }

        // Apply modulus operation to both adjusted numbers to ensure the product is within the range
        adjustedNum1 %= MODULUS;
        adjustedNum2 %= MODULUS;

        // Calculate the product of adjustedNum1 and adjustedNum2, apply modulus operation, and cast to integer
        return (int) (adjustedNum1 * adjustedNum2 % MODULUS);
    }
}
```

C++

```
class Solution {
public:
    int maximumXorProduct(long long num1, long long num2, int n) {
        const int MOD = 1e9 + 7; // Define the modulo constant for the result

        // Initialize `aXor` and `bXor` by preserving the higher bits up to `n` and setting lower bits to 0
        long long aXor = (num1 >> n) << n;
        long long bXor = (num2 >> n) << n;

        // Iterate bit positions from n-1 down to 0
        for (int bitPos = n - 1; bitPos >= 0; --bitPos) {
            // Extract the bit at `bitPos` for both `num1` and `num2`
            int bit1 = (num1 >> bitPos) & 1;
            int bit2 = (num2 >> bitPos) & 1;

            // If the bits are the same, set the bit at `bitPos` in both `aXor` and `bXor`
            if (bit1 == bit2) {
                aXor |= 1LL << bitPos;
                bXor |= 1LL << bitPos;
            } else if (aXor < bXor) {
                // If `aXor` is less than `bXor`, set the bit in `aXor`
                aXor |= 1LL << bitPos;
            } else {
                // Otherwise, set the bit in `bXor`
                bXor |= 1LL << bitPos;
            }
        }

        // Apply the modulo operation to both `aXor` and `bXor`
        aXor %= MOD;
        bXor %= MOD;

        // Return the product of `aXor` and `bXor` under modulo
        return (aXor * bXor) % MOD;
    }
};
```

TypeScript

```
function maximumXorProduct(a: number, b: number, n: number): number {
    // Define the modulus for the final result to avoid large numbers
    const MOD = BigInt(1e9 + 7);

    // Initialize ax and bx to only include the bits from a and b above the nth bit
    // This effectively zeroes out the last n bits
    let ax = (BigInt(a) >> BigInt(n)) << BigInt(n);
    let bx = (BigInt(b) >> BigInt(n)) << BigInt(n);

    // Loop over each bit from n-1 down to 0
    for (let i = BigInt(n - 1); i >= 0n; --i) {
        // Extract the ith bit from both a and b
        const abit = (BigInt(a) >> i) & 1n;
        const bbit = (BigInt(b) >> i) & 1n;

        // If the bits are equal, set the ith bit of both ax and bx
        if (abit === bbit) {
            ax |= 1n << i;
            bx |= 1n << i;
        }
        // If ax is currently less than bx, set the ith bit of ax
        else if (ax < bx) {
            ax |= 1n << i;
        }
        // Otherwise, set the ith bit of bx
        else {
            bx |= 1n << i;
        }
    }

    // Reduce ax and bx by the modulus to handle overflow
    ax %= MOD;
    bx %= MOD;

    // Calculate the product of ax and bx, reduce it by the modulus,
    // and return the number representation of the result
    return Number((ax * bx) % MOD);
}
```

```
class Solution:
    def maximum_xor_product(self, num1: int, num2: int, num_bits: int) -> int:
        # Define the modulo for taking the result % mod
        MODULUS = 10**9 + 7

        # Initialize variables to store the maximum XOR values based on the most significant bits
        xor_num1, xor_num2 = (num1 >> num_bits) << num_bits, (num2 >> num_bits) << num_bits

        # Iterate over each bit, from most significant bit to least significant bit
        for i in range(num_bits - 1, -1, -1):
            # Extract the current bit of num1 and num2
            bit_num1 = (num1 >> i) & 1
            bit_num2 = (num2 >> i) & 1

            # If the current bits are equal, set the current bit in both xor_num1 and xor_num2
            if bit_num1 == bit_num2:
                xor_num1 |= 1 << i
                xor_num2 |= 1 << i
            # If xor_num1 is greater, set the current bit in xor_num2
            elif xor_num1 > xor_num2:
                xor_num2 |= 1 << i
            # Otherwise, set the current bit in xor_num1
            else:
                xor_num1 |= 1 << i

        # Return the product of xor_num1 and xor_num2 modulo MODULUS
        return (xor_num1 * xor_num2) % MODULUS
```

Time and Space Complexity

Time Complexity

The time complexity of the code is $O(n)$. Here, `n` refers to the value given as an argument to the function, not the size of any input array or list, as might typically be the case in algorithmic problems. The code iterates from `n - 1` down to `0` via the for loop, resulting in exactly `n` iterations, hence the time complexity is linear with respect to `n`.

Space Complexity

The space complexity of the code is $O(1)$. This analysis is straightforward as the function operates in constant space, using a fixed number of integer variables (`mod`, `ax`, `bx`, `i`, `x`, `y`) regardless of the value of `n` or the size of the input numbers `a` and `b`. There are no data structures used that grow with the size of the input, which confirms that the space requirements remain constant.