2438. Range Product Queries of Powers

Medium | Bit Manipulation | Array | Prefix Sum | Leetcode Link

Problem Description

The problem presents us with two tasks:

1. Construct an array called `powers` from a given positive integer `n`, where the array consists of the smallest set of powers of 2 that add up to `n`. It's important to note that the array is indexed starting from 0, sorted in non-decreasing order, and has a unique possible configuration for any given `n`.

2. Process multiple queries described by the `queries` 2D array, where each subarray `[left, right]` means we need to calculate the product of `powers[j]` for all values of `j` from `left` to `right` inclusive.

The array of products, one for each query, must be computed and returned, and to manage potentially large numbers, the final products should be computed modulo `10^9 + 7`.

An example can illustrate this better: Suppose `n = 10`, which in binary is `1010`. The `powers` array would consist of the powers of 2 that sum up to `n`: `[2^1, 2^3]` or `[2, 8]`. For a query `[0, 1]` which means calculate the product `powers[0] * powers[1]`, the answer would be `2 * 8 = 16`.

Intuition

To address the problem systematically, we can approach it as follows:

Creating the `powers` Array

1. Find the minimum set of powers of 2 that sum up to `n`:

   - We iterate through the binary representation of `n` from the least significant bit to the most significant bit.
   - For each bit that is set (i.e., is 1), we calculate the corresponding power of 2. This is achieved by utilizing the operation `n & -n`, which isolates the least significant bit that is set and yields the smallest power of 2 contributing to `n`.
   - After finding the current smallest power of 2, we subtract it from `n`, and continue the process until `n` becomes 0.
2. The `powers` array is constructed implicitly in reverse order (from larger to smaller powers of 2) because we're starting with the least significant bit (smallest powers of 2) in `n`'s binary representation.

Processing the Queries

1. Now, with the `powers` array ready, we loop through each query:
   - Initialize a variable to accumulate the product (starting with 1) within the range `[left, right]`.
   - Multiply the components of the `powers` array that are within the range specified by each query. Since we want to avoid integer overflow issues, we take the modulus of the product with `10^9 + 7` at each step.
   - Append the resulting product to our answer list.

The intuition behind modulo operation at each multiplication step is to ensure that we stay within the maximum limit for integer values and correctly handle cases where the product may be very large, as the final product must be returned modulo `10^9 + 7`.

Solution Approach

The solution to this LeetCode problem involves both bitwise manipulation to construct the `powers` array and modular arithmetic to calculate and store the query results. Here's how the implementation goes step-by-step, corresponding to the given solution code:

1. **Initializing the `powers` array**: We need a list `powers` to store the powers of 2 that make up the number `n`. Since a positive integer can be represented as a sum of unique powers of 2 (according to binary representation), we find these powers and store them.

2. **Finding the powers of 2 from `n`**: We use a while loop that continues until `n` becomes 0. Inside this loop, we use `x = n & -n` to get the lowest power of 2 in `n`. This operation works because in binary, `-n` is `n` with all bits inverted plus 1 (two's complement representation), so `n & -n` isolates the least significant 1-bit. We append `x` to the `powers` list and subtract `x` from `n` using `n -= x`.

3. **Modular exponentiation**: We take note that when dealing with large numbers, we should use modular arithmetic to avoid overflow. Modulo operations are distributive over multiplication, which is crucial for our solution. We set `mod` to `10**9 + 7`.

4. **Processing each query**: We need to iterate over the range of powers specified by each query:

   - We initialize a variable `x = 1` for the product result of the current query.
   - The slice `powers[l : r + 1]` gets the relevant segment of the `powers` array based on the current query range.
   - For each element `y` in the above slice, we update `x` as `x = (x * y) % mod` to get the product of the current segment, applying the modulus at each multiplication step to ensure the result stays within bounds of the integer limits.
5. **Storing the result**: For each query, after we calculate the product, we append it to the answer list `ans`. The list `ans` is our final result array that will be returned.

This solution utilizes simple bitwise operations to deconstruct a number into powers of 2, and then employs modular arithmetic with a simple iteration to answer the range product queries.

Example Walkthrough

Let's take `n = 10` and a single query `[0, 1]` to illustrate the solution approach step-by-step.

1. **Initializing the `powers` array**: We begin by creating an empty array `powers` to store powers of 2. Since we are provided with `n = 10` (which is `1010` in binary), we need to find which powers of 2 add up to 10.

2. **Finding the powers of 2 from `n`**: We loop while `n` is not zero,

   - In the first iteration, `n` is `10`. We do `x = n & -n`, which gives us `2` (`10` in binary), because `10 & -10` isolates the least significant bit which represents the power of 2.
   - We append `2` to the `powers` array, which now becomes `[2]`.
   - We subtract `2` from `n`, making `n = 10 - 2 = 8`.
   - In the next iteration, `n` is `8`. Again `x = n & -n` gives us `8` because `8` is a power of 2.
   - We append `8` to the `powers` array, which now becomes `[2, 8]`.
   - Subtract `8` from `n`, `n` becomes `0`. The loop ends here.

After these steps, the `powers` array represents the powers of 2 that sum up to `n`: `[2, 8]`.

3. **Modular exponentiation**: Set the variable `mod` to `10^9 + 7` to ensure all operations are performed modulo this number to prevent integer overflow.

4. **Processing each query**: With the `powers` array ready, we process the query `[0, 1]`:

   - We initialize `x = 1` which will hold our product.
   - Based on the query, we need the slice `powers[0 : 1+1]`, which is the entire `powers` array `[2, 8]`.
   - Loop through the slice and for each element `y`, update `x` as `(x * y) % mod`.
   - First, `x = (1 * 2) % mod = 2`. Next, `x = (2 * 8) % mod = 16`.
   - No more elements in the slice, so the final product of this query is `16`.
5. **Storing the result**: We append `16` to the answer list `ans`. If there were more queries, each product would be appended to `ans` as well.

In this example, the answer list `ans` contains just one element `[16]`, because there was only one query. This final array is what the function would return.

Python Solution

```python
class Solution:
    def product_queries(self, n: int, queries: List[List[int]]) -> List[int]:
        # Initialize an array to store the powers of 2 found in the binary representation of n
        powers_of_two = []

        # Extract powers of 2 from n by finding the rightmost set bit continuously
        while n:
            # x is the largest power of 2 in n, given by the bitwise AND of n and its two's complement
            x = n & -n
            # Add x to the list
            powers_of_two.append(x)
            # Remove the largest power of 2 from n
            n -= x

        # Set modulo as per the problem statement to avoid large integer overflow
        mod = 10**9 + 7

        # Initialize an array to store the results of the queries
        results = []

        # Loop through each query, which is a pair of indices (l, r)
        for l, r in queries:
            # Start with a product result of 1 for each query
            product_result = 1

            # Multiply the values from powers_of_two[l] to powers_of_two[r]
            # and take the modulo to prevent overflow
            for power in powers_of_two[l : r + 1]:
                product_result = (product_result * power) % mod

            # Append the product result to the results list
            results.append(product_result)

        # Return the final list of results for all queries
        return results

```

Java Solution

```java
class Solution {
    // Define the mod constant for the problem (10^9 + 7)
    private static final int MOD = (int) 1e9 + 7;

    // Method to solve the product queries on a range of powers of two that compose n
    public int[] productQueries(int n, int[][] queries) {
        // Count the number of set bits in n to determine the size of the powers array
        int[] powers = new int[Integer.bitCount(n)];

        // Extract the powers of two which compose n
        for (int i = 0; n > 0; ++i) {
            int lowestOneBit = n & -n; // Get the lowest set bit (the rightmost one)
            powers[i] = lowestOneBit; // Store the power of two in the array
            n -= lowestOneBit; // Remove the extracted power of two from n
        }

        // Create an array to store the answers to the queries
        int[] ans = new int[queries.length];

        // Process each query
        for (int i = 0; i < ans.length; ++i) {
            long product = 1; // Store the product as a long to prevent overflow
            int left = queries[i][0], right = queries[i][1]; // Get the left and right indices

            // Calculate the product of the powers from left to right index inclusive
            for (int j = left; j <= right; ++j) {
                product = (product * powers[j]) % MOD; // Multiply the current power and take mod
            }

            // Store the result as an int after casting from long
            ans[i] = (int) product;
        }

        // Return the array containing the results for all the queries
        return ans;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Constants should be capitalized and use static constexpr for compile time initialization
    static constexpr int MOD = 1e9 + 7;

    // This function takes an integer and a set of queries and returns the product of powers for each query
    vector<int> productQueries(int n, vector<vector<int>>& queries) {
        vector<int> powersOfTwo;

        // Extract powers of 2 from n and store them in the powersOfTwo vector
        while (n > 0) {
            // Get the rightmost set bit (largest power of 2 not greater than n)
            int largestPowerOfTwo = n & -n;
            // Add the power of 2 to the list
            powersOfTwo.emplace_back(largestPowerOfTwo);
            // Subtract the power of 2 from n to remove that bit
            n -= largestPowerOfTwo;
        }

        vector<int> answer;
        // Iterate through each query
        for (auto& query : queries) {
            int start = query[0], end = query[1];
            long long product = 1;  // Define the product as a long long to prevent integer overflow

            // Calculate the product of powers from start to end index
            for (int i = start; i <= end; ++i) {
                product = (product * powersOfTwo[i]) % MOD;
            }

            // Store the result in answer using modulo to fit within integer range
            answer.emplace_back(static_cast<int>(product));
        }
        // Return the final answers for all queries
        return answer;
    }
};

```

Typescript Solution

```typescript
// Define module-level constants in uppercase with `const` (no static needed in global context)
const MOD = 1e9 + 7;

// This function takes an integer and an array of queries and returns the product of powers for each query
function productQueries(n: number, queries: number[][]): number[] {
    let powersOfTwo: number[] = [];

    // Extract powers of 2 from n and store them in the powersOfTwo array
    while (n > 0) {
        // Get the rightmost set bit (largest power of 2 not greater than n)
        let largestPowerOfTwo = n & -n;
        // Add the power of 2 to the array
        powersOfTwo.push(largestPowerOfTwo);
        // Subtract the power of 2 from n to remove that bit
        n -= largestPowerOfTwo;
    }

    let answers: number[] = [];
    // Iterate through each query
    queries.forEach(query => {
        let start = query[0], end = query[1];
        // Define the product as a bigint to prevent integer overflow and use BigInt literals for calculations
        let product: bigint = BigInt(1);

        // Calculate the product of powers from start to end indices
        for (let i = start; i <= end; ++i) {
            product = (product * BigInt(powersOfTwo[i])) % BigInt(MOD);
        }

        // Store the result in answers using modulo to fit within integer range and convert BigInt to number
        answers.push(Number(product));
    });

    // Returns the final answers for all queries
    return answers;
}

```

Time and Space Complexity

Time Complexity

The given Python code consists of two main parts: extracting powers of 2 from `n`, and computing the product of subsets as specified by `queries`.

1. **Extracting Powers of 2 from `n`**:

   - This is done with a loop that runs until `n` becomes 0.
   - For each iteration, we perform a bitwise AND of `n` and `-n` to isolate the lowest power of 2 in `n`.
   - Since each number can have at most `O(log n)` bits, where `log` refers to the logarithm base 2, the loop runs for `O(log n)` iterations.
   - The operation within each loop is `O(1)`.
   - Therefore, this step has a time complexity of `O(log n)`.
2. **Processing Queries**:

   - There is a loop iterating over each query in `queries`. Let's denote the number of queries as `q`.
   - Inside this loop, there is another loop over the powers from index `l` to `r`, which in the worst case includes all the powers extracted previously.
   - The number of powers is at most `O(log n)`, as previously derived.
   - Multiplying and taking the modulo has a constant time complexity `O(1)`.
   - Therefore, processing all queries has a worst-case time complexity of `O(q * log n)`.

Overall, the time complexity of the entire function is `O(log n) + O(q * log n) = O((q + 1) * log n)`.

Space Complexity

The space complexity of the code is determined by the extra space used apart from the input:

1. **Storing Powers**:

   - We store each power of 2 that exists in the binary representation of `n`.
   - This takes `O(log n)` space.
2. **Answer List**:

   - We store one integer for each query, so this takes `O(q)` space.
3. **Temporary Variables**:

   - Temporary variables such as `x`, `y`, and `mod` are `O(1)` space.

Overall, the resultant space complexity is `O(log n) + O(q) = O(log n + q)`.