762. Prime Number of Set Bits in Binary Representation

Easy Bit Manipulation Math

Leetcode Link

Problem Description

The problem provides us with two integers, `left` and `right`, and asks us to calculate how many numbers in the range from `left` to `right` (inclusive) have a prime number of set bits in their binary representations. The term "set bits" refers to the number of 1's that appear in the binary form of a given number. For example, the number `21` has a binary representation of `10101`, which contains three 1's, or three set bits.

Intuition

The intuition behind the solution can be broken down into a few logical steps:

1. **Identify the Prime Numbers**: Since we are interested in numbers that have a prime number of set bits, we first need a set of prime numbers to check against. However, we don't need all prime numbers, just the ones within the possible range of set bits for numbers in our input range. Given that the max value for a 32-bit integer is 2^31 - 1, it has at maximum 31 set bits. The prime numbers within this range are `{2, 3, 5, 7, 11, 13, 17, 19}`.

2. **Count Set Bits**: For each number in the inclusive range between `left` and `right`, we need to determine the number of set bits. There is a built-in method in Python called `bit_count` for integers that do just that, counting the number of bits set to `1` in binary representation.

3. **Check Primes**: Once we have the count of set bits for a number, we check if this count is in our pre-identified set of prime numbers.

4. **Summing Up Prime Set Bits Counts**: Finally, we need to sum up how many numbers fall within our criteria. This is done using sum comprehension in Python, where for each number in our range, we count it only if the number of set bits is a prime number.

By combining these steps, we are able to arrive at the final count efficiently.

Solution Approach

The implementation of the solution uses a simple but effective approach:

1. **Define a Set of Prime Numbers**:
   - We create a set called `primes` which contains the prime numbers less than 20 (as calculated above, since 32-bit integers have a maximum of 31 set bits, and 19 is the largest prime number less than 31).
   - The set data structure is chosen due to its O(1) time complexity for membership checking, as we will need to check if a count (number of set bits) is a prime number.
2. **Iterate Over the Range**:
   - We loop over each number from `left` to `right` (inclusive) using a range in the loop: `for i in range(left, right + 1)`.
   - This iteration is simple and direct, covering each integer in the provided range.
3. **Count the Set Bits**:
   - For each number `i` in the range, we use the `bit_count()` method to find the number of set bits for that number.
   - The use of `bit_count()` is a Python-specific method introduced in version 3.10 which simplifies and optimizes the operation of counting set bits.
4. **Check for Prime Number of Set Bits**:
   - The comprehension part `i.bit_count() in primes` checks if the number of set bits is in our set of prime numbers.
   - If the number of set bits is a prime number, this evaluates to `True`, which, when summed using the `sum()` function, is treated as `1`.
5. **Calculate the Total Count**:
   - Finally, the `sum()` function adds up all the `1`s (for each `True` result) reflecting the count of numbers that satisfy the condition (having a prime number of set bits).

The entire operation returns the sum, which is precisely the count of numbers with a prime number of set bits in the specified range. This approach is efficient because it minimizes the calls to check for prime numbers by predefining the possible primes and uses bitwise operation optimized functions for count calculation.

Example Walkthrough

Let's take the range from `left = 10` to `right = 15` and illustrate the solution approach step by step.

1. **Define the Set of Prime Numbers**: We have a pre-determined set of prime numbers, which are `{2, 3, 5, 7, 11, 13, 17, 19}` because we are working within the 32-bit integer range.

2. **Iterate Over the Range**: We review the numbers 10, 11, 12, 13, 14, and 15.

3. **Count the Set Bits** and **Check for Prime Number of Set Bits**:
   - For the number `10` which in binary is `1010`, there are two set bits. Two is a prime number.
   - For the number `11` which in binary is `1011`, there are three set bits. Three is also a prime number.
   - For the number `12` which in binary is `1100`, there are two set bits. Two is a prime number.
   - For the number `13` which in binary is `1101`, there are three set bits. Three is a prime number.
   - For the number `14` which in binary is `1110`, there are three set bits. Three is a prime number.
   - For the number `15` which in binary is `1111`, there are four set bits. Four is not a prime number.
4. **Calculate the Total Count**: Out of the numbers from 10 to 15:
   - Four numbers (`10`, `11`, `12`, `13`, `14`) have a prime number of set bits.
   - Two numbers (`15`) do not.

Therefore, the final count of numbers with a prime number of set bits between 10 and 15, inclusive, is 4.

Python Solution

```python
class Solution:
    def count_prime_set_bits(self, left: int, right: int) -> int:
        # Define a set of prime numbers that could represent set bit counts
        primes = {2, 3, 5, 7, 11, 13, 17, 19}

        # Use list comprehension to count the number of integers in the specified range
        # which have a prime number of set bits (1s in their binary representation)
        prime_set_bits_count = sum(
            bin(i).count('1') in primes
            for i in range(left, right + 1)
        )

        return prime_set_bits_count

```

Java Solution

```java
import java.util.Set;

class Solution {
    // Initialization of a Set containing prime numbers.
    private static final Set<Integer> PRIME_NUMBERS = Set.of(2, 3, 5, 7, 11, 13, 17, 19);

    // Method to count the numbers in the given range with a prime number of set bits.
    public int countPrimeSetBits(int left, int right) {
        // Initialize a counter for the numbers meeting the criteria.
        int primeSetBitsCount = 0;

        // Iterate over the range from left to right, inclusive.
        for (int i = left; i <= right; ++i) {
            // Use Integer.bitCount to determine the number of set bits in the binary representation of 'i'.
            // If the number of set bits is in the PRIME_NUMBERS set, increment the counter.
            if (PRIME_NUMBERS.contains(Integer.bitCount(i))) {
                primeSetBitsCount++;
            }
        }

        // Return the total count of numbers with a prime number of set bits.
        return primeSetBitsCount;
    }
}

```

C++ Solution

```cpp
#include <unordered_set>

class Solution {
public:
    // Function to count the number of integers within a range [left, right]
    // that have a prime number of set bits in their binary representation
    int countPrimeSetBits(int left, int right) {
        // Define a set of primes that are less than or equal to 19, since the maximum
        // number of bits for an int (32 bits) has at most 19 set bits to be a prime.
        std::unordered_set<int> primeSet{2, 3, 5, 7, 11, 13, 17, 19};

        int count = 0; // This variable will store the count of numbers satisfying the condition

        // Iterate over each number in the given range
        for (int i = left; i <= right; ++i) {
            // Use __builtin_popcount to count the number of set bits in the current number
            // Increase the count if the number of set bits is in the primeSet
            count += primeSet.count(__builtin_popcount(i));
        }

        // Return the final count
        return count;
    }
};

```

Typescript Solution

```typescript
// Define a set of prime numbers less than or equal to 19
const primeSet: Set<number> = new Set([2, 3, 5, 7, 11, 13, 17, 19]);

// Function to count the number of integers within a range [left, right]
// that have a prime number of set bits in their binary representation.
function countPrimeSetBits(left: number, right: number): number {
    let count = 0; // Initialize count of numbers satisfying the condition

    // Iterate over each number within the range from left to right inclusive
    for (let i = left; i <= right; i++) {
        // Count the number of set bits (1s) in the binary representation of the number
        const setBits = countSetBits(i);

        // If the number of set bits is prime, increment the count
        if (primeSet.has(setBits)) {
            count++;
        }
    }

    // Return the final count
    return count;
}

// Helper function to count the number of set bits in the binary representation of a number
function countSetBits(num: number): number {
    let setBits = 0; // Initialize count of set bits to 0

    while (num > 0) {
        setBits += num & 1; // Increment setBits if the least significant bit is 1
        num >>= 1; // Right shift the number to check the next bit
    }

    return setBits;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is mainly determined by the for loop, which iterates from `left` to `right` inclusive.

We let `N` represent the number of integers in the range `[left, right]`, then `N` equals `right - left + 1`.

For each `i` in the range `[left, right]`, we calculate the number of set bits using `i.bit_count()`. Since the maximum number of bits for an integer is bounded by the logarithm of the integer, let's denote the number of bits as `k`. The operation `bit_count()` has a time complexity of `O(k)`.

However, since the values of `left` and `right` are not restricted by input size as in traditional algorithm analyses, where `N` would normally depend on some input parameter like array size, defining `k` is trickier. Generally, for a 32-bit integer, `k` would be at most `32`. Thus we can consider `bit_count()` operation to be `O(1)` for practical purposes.

Therefore, since `bit_count()` takes constant time and is called `N` times, the time complexity of the entire loop is `O(N)`.

The set membership test `in primes` is `O(1)` because lookup in a set of prime numbers (of fixed small size) is constant time.

Combining these, the loop has a time complexity of `O(N * 1) = O(N)`.

Space Complexity

The space complexity of the given code is low.

We have a set, `primes`, containing a fixed number of prime numbers which is independent of the input size. This means it consumes a constant amount of space, `O(1)`.

The temporary variables for the loop and the sum operation are also constant space, so they do not contribute to the space complexity in terms of `N`.

Therefore, the overall space complexity of the code is `O(1)`.