3064. Guess the Number Using Bitwise Questions I

Medium Bit Manipulation Interactive

Problem Description

In this problem, we have to find a certain number `n` using an API called `commonSetBits`. This API takes an integer `num` as its input and returns the number of set bits (1 bits) at the same positions in both `n` and `num`. The bit positions are counted from the least significant bit, which is position 0. The API is essentially calculating the number of 1 bits in the result of `n` AND `num`.

The task is to determine `n` exactly.

To clarify with an example: if `n` is `5`, which has a binary representation of `101`, and we call `commonSetBits(1)` which is `001` in binary, it will return `1`, because they have one `1` in common at the 0th position. However, if we call `commonSetBits(2)` which is `010` in binary, it will return `0`, because they don't have any `1` bits in common.

The constraints ensure that:

- The unknown number `n` is between `1` and `2^30 - 1`.
- The input to `commonSetBits`, `num`, is between `0` and `2^30 - 1`.
- We cannot rely on the output of `commonSetBits` if the input `num` is outside the specified range.

Intuition

Given that we need to find the exact value of `n` and we have an API that tells us which bits are in common between `n` and any given `num`, the intuition is to call `commonSetBits` with inputs that help us isolate each bit of `n`.

The evident choice for such isolating calls would be powers of 2. Why? Because the binary representation of `2^i` has a `1` at the ith position and `0` everywhere else. By calling `commonSetBits(2^i)` for every bit position from 0 to 31, we can check if that particular bit is set in `n`.

Solution Approach

The solution involves a simple yet efficient approach which is a straightforward implementation of the intuition we've discussed.

Since we know the number of bits in `n` is limited (up to 32 for an integer within the constraint of `2^30 - 1`), we can iteratively check each bit position to see whether it is set in `n`. We do this by using the power of 2 for each bit position as input to the `commonSetBits` API: `2^0`, `2^1`, `2^2`, ..., `2^31`.

The provided code in the solution uses list comprehension to create a list of such powers of `2` only where the corresponding bit is set in `n`, determined by whether `commonSetBits(1 << i)` is greater than 0.

Here is the breakdown:

- For each bit position `i` from 0 to 31:
  - Calculate `1 << i`, which is a number with only the ith bit set (indicating `2^i`).
  - Call `commonSetBits(1 << i)` and check if the result is greater than 0. If yes, it means that the ith bit is also set in `n` since the common set bits are counted.
  - If the ith bit in `n` is set, include `1 << i` in the sum.

Algorithmically, the list comprehension `[1 << i for i in range(32) if commonSetBits(1 << i)]` returns a list of all `2^i` values that contribute to the final number `n`.

The `sum()` function then adds up all these individual powers of 2, which, when combined, reconstruct the original number `n`.

This solution is effective since it requires only 32 calls to `commonSetBits` (one for each bit) and uses no additional data structures. Overall, the code's time complexity is O(32), which is effectively O(1) since it's a constant time operation, and the space complexity is also O(1) as there are no data structures using space proportional to the size of the input.

Here's a visualization of the algorithm for a number n's binary representation `1101` (13 in decimal):

```
Bit position:  3210
n:             1101
2^0 (1):       0001 (API returns 1, bit is set)
2^1 (2):       0010 (API returns 0, bit is not set)
2^2 (4):       0100 (API returns 1, bit is set)
2^3 (8):       1000 (API returns 1, bit is set)

Resulting sum: 1 + 0 + 4 + 8 = 13 (which is n)
```

The use of bitwise shifts (`<<`) and checks for values greater than zero in the list comprehension makes the code both concise and optimal for the task at hand.

Example Walkthrough

Let's go through a small example to illustrate the solution approach.

Suppose the unknown number `n` we're trying to find is `11`, which is `1011` in binary. We want to use the `commonSetBits` API to identify the positions of the `1` bits in `n`.

Here's how we would proceed, step-by-step:

1. We begin by checking if the least significant bit (LSB, bit position 0) is a `1` in `n` by calling `commonSetBits(1)`, since `1` in binary is `0001`. If `n` has a `1` at this position, the API will return `1`.
   - `commonSetBits(1)` → returns `1` because both `n` (1011) and the input (0001) have a `1` at LSB.
2. Next, we check bit position 1 by calling `commonSetBits(2)`, which corresponds to binary `0010`. The API will return `1` if `n` also has a `1` in this position.
   - `commonSetBits(2)` → returns `1` because `n` (1011) and the input (0010) both have a `1` at position 1.
3. We then check bit position 2 with `commonSetBits(4)` (binary `0100`). If `n` contains a `1` at this position, the API will return `1`.
   - `commonSetBits(4)` → returns `0` indicating that position 2 in `n` is not set (`n` is 1011 and input is 0100).
4. Moving on to bit position 3, we call `commonSetBits(8)`, which is `1000` in binary. If `n` has a `1` bit here, the API will return `1`.
   - `commonSetBits(8)` → returns `1` because both `n` (1011) and the input (1000) have a `1` at position 3.

Now that we have called the API with powers of `2` for each bit position up to the bit position 3, we have the following information:

- The API returned `1` for inputs `1`, `2`, and `8`.
- The API returned `0` for input `4`.

Using this information, we can sum up all powers of `2` corresponding to the API returning `1`:

[\text{Resulting sum} = 2^0 + 2^1 + 2^3 = 1 + 2 + 8 = 11]

The sum `11` is the original number `n` we aimed to find.

This example demonstrates how the solution uses powers of `2` to isolate each bit of the unknown number and correctly sum them to discover `n` with only a few calls to the API.

Solution Implementation

Python

```python
# Let's assume there is a predefined API commonSetBits which, when given an integer,
# returns the number of set bits (1's) common amongst all the elements upto 'num'.

# Here we have a class 'Solution' with a method 'findNumber'
class Solution:
    def findNumber(self) -> int:
        # Initialize 'result' to store sum of powers of 2 that meet the condition
        result = 0
        # Iterate over each bit position from 0 to 31 (for 32-bit integers)
        for i in range(32):
            # Use commonSetBits with a power of 2 (only that bit set to 1)
            # Check if 'i'th bit is common amongst all numbers
            # with the bit set using the API
            if commonSetBits(1 << i):
                # If common, add the corresponding power of 2 to the result
                result += 1 << i
        # Once done, return the sum which represents the number we are trying to find
        return result
```

Java

```java
/**
 * The following class Solution is meant to find a number based on the set bits
 * that it has in common with other numbers through the `commonSetBits` API.
 */
public class Solution extends Problem {
    /**
     * This method determines the number that has common set bits with other numbers.
     *
     * @return the number which has the maximum number of common set bits with others.
     */
    public int findNumber() {
        // Initialize the result variable that will store the number with common set bits.
        int result = 0;

        // Iterate through each of the 32 bits of an integer.
        for (int i = 0; i < 32; ++i) {
            // Create a bitmask with only the i-th bit set.
            int bitmask = 1 << i;

            // Use the commonSetBits API to check if the current bit is set in the common number.
            if (commonSetBits(bitmask) > 0) {
                // If the i-th bit is common, include it in the result using bitwise OR.
                result |= bitmask;
            }
        }

        // Return the number with all common set bits combined.
        return result;
    }
}
```

C++

```cpp
/**
 * Forward declaration of the API.
 * int commonSetBits(int num);
 * The API is used to check how many bits set (to 1) in the input 'num' are common
 * with a specific unknown number.
 *
 * @param num The number to test against the unknown number.
 * @return The number of common bits set to 1 in the input 'num' and the unknown number.
 */

class Solution {
public:
    /**
     * Finds the unknown number with which the `commonSetBits` API compares.
     *
     * @return The unknown number that shares the common set bits with the input
     *         to the `commonSetBits` API.
     */
    int findNumber() {
        int result = 0; // Initialize result to 0.

        // Loop through all bit positions from 0 to 31 (assuming 32-bit integer)
        for (int i = 0; i < 32; ++i) {
            // Check if bit at position 'i' is set in the unknown number by
            // passing a number with only the 'i'th bit set to the API.
            int bitMask = 1 << i; // Create a mask with only the 'i'th bit set.
            if (commonSetBits(bitMask)) {
                // If the 'i'th bit is common, set the 'i'th bit in the result.
                result |= bitMask;
            }
        }

        // Return the result which now represents the unknown number.
        return result;
    }
};
```

TypeScript

```typescript
// Declaration of the commonSetBits API. This function takes a number and
// returns the number of common set bits.
declare function commonSetBits(num: number): number;

// This function finds a number composed of bits that are commonly set
// when using the `commonSetBits` function on powers of 2 up to 2^31.
function findNumber(): number {
    // Initialize the number to be returned.
    let resultNumber = 0;

    // Iterate through all bits positions from 0 to 31 (32-bit number assumption).
    for (let i = 0; i < 32; ++i) {
        // Create a number with only the i-th bit set.
        const bitMask = 1 << i;

        // Check if this bit is commonly set using the provided API function.
        if (commonSetBits(bitMask)) {
            // If so, set the i-th bit in the result number.
            resultNumber |= bitMask;
        }
    }

    // Return the composed number containing all the common set bits.
    return resultNumber;
}
```

```python
# Let's assume there is a predefined API commonSetBits which, when given an integer,
# returns the number of set bits (1's) common amongst all the elements upto 'num'.

# Here we have a class 'Solution' with a method 'findNumber'
class Solution:
    def findNumber(self) -> int:
        # Initialize 'result' to store sum of powers of 2 that meet the condition
        result = 0
        # Iterate over each bit position from 0 to 31 (for 32-bit integers)
        for i in range(32):
            # Use commonSetBits with a power of 2 (only that bit set to 1)
            # Check if 'i'th bit is common amongst all numbers
            # with the bit set using the API
            if commonSetBits(1 << i):
                # If common, add the corresponding power of 2 to the result
                result += 1 << i
        # Once done, return the sum which represents the number we are trying to find
        return result
```

Time and Space Complexity

Time Complexity

The time complexity of the code is mainly determined by the for-loop that iterates over the range of `32`, which stands for the number of bits in an integer data type in Python (assuming the standard 32-bit integer size). Inside the loop, the `commonSetBits` is called once for each bit position from `0` to `31`.

Even though the loop runs 32 times which is a fixed number, the given reference mentions the time complexity as `O(log n)`, where `n <= 2^30`. This could imply that the `commonSetBits` function itself has a time complexity of `O(log n)` due to potentially using operations that depend logarithmically on the value of the number passed to it. However, analyzing without the specifics of `commonSetBits`, the time complexity would seem to be `O(1)` since the loop runs a constant number of times.

Assuming that the `commonSetBits` function is indeed dependent on the number of bits set to 1 in a given number (which would not exceed `30` as per the given constraints and the context of the problem), the overall time complexity is `O(log n)` based on the problem's constraint that `n <= 2^30`.

Space Complexity

The space complexity of the code is `O(1)` since it does not allocate any additional space that grows with the size of the input. The only extra space used is for the variable `i` in the for-loop and the space needed for calculating the sum, which do not depend on the input size and are thus constant.