201. Bitwise AND of Numbers Range

Medium Bit Manipulation

Leetcode Link

Problem Description

The problem presents us with a task to calculate the bitwise AND for all numbers in the range from 'left' to 'right' inclusive. The 'bitwise AND' operation takes two bit operands and performs the logical AND operation on each pair of corresponding bits. The result is 1 if both bits are 1, and 0 otherwise. This problem thus requires us to perform this operation on a sequence of integers, starting from 'left' up to 'right'.

Here is a step-by-step understanding of the problem:

1. We take two integers, 'left' and 'right'.
2. We apply the bitwise AND to every pair of numbers in that range.
3. We want to find the cumulative result of the AND operation applied in sequence from 'left' to 'right'.

To illustrate, if our range is [5, 7], we need to calculate the bitwise AND of 5, 6, and 7.

- The binary representation of 5 is 101.
- The binary representation of 6 is 110.
- The binary representation of 7 is 111.

The bitwise AND of 5 AND 6 is 100 (in binary), and the bitwise AND of 100 AND 111 is 100, so the result for our example would be 4.

Intuition

The intuition behind the provided solution leverages the idea that the result of the bitwise AND operation of a range of numbers is determined by the common bits i.e., the bits that don't change across all numbers in the range. The moment a bit changes from 1 to 0 within the range, due to the nature of the AND operation (which requires all operands for a particular bit position to be 1 in order to yield a 1), that bit will be 0 in the final outcome.

The crucial observation is that if we progressively strip off the lowest bit of the 'right' number until it is less than or equal to 'left', the bits that remain align with the bits of the final AND operation result.

Here's why:

- If 'left' and 'right' are the same, the bits remain unchanged as there's nothing to compare or operate against, so the bitwise AND is equal to the 'left' or 'right' value.
- If 'left' is less than 'right', at some point, the least significant bit that differs between 'left' and 'right' will become 0 in 'right' after a certain number of operations because 'right' will be decremented for every bit that is 1 starting from the least significant bit.
- Since 'right' is getting smaller with right & (right - 1) and 'left' is constant, eventually 'right' will be less than or equal to 'left' or the differing bits at the end of 'right' have been turned off.
- The remaining bits before the changed bit in 'right' are common with 'left', therefore they remain unchanged and represent the bitwise AND of all numbers in the range.

By employing this method, we avoid the necessity of iterating through all numbers in the range and directly arrive at the result by focusing only on the bits that won't be altered by the AND operation in the range.

Solution Approach

The provided solution uses a simple while loop and a bitwise operation to iteratively narrow down the range to finally arrive at the bitwise AND of all numbers within the range `[left, right]`. Here's a step-by-step explanation of how the solution works:

1. The solution approach starts with a condition `while left < right:` which means that we will keep iterating as long as our `right` value is greater than our `left` value.

2. Inside the loop, the key operation is `right &= right - 1`. This operation takes the current value of `right` and performs a bitwise AND with the number one less than `right` (`right - 1`).

3. The expression `right - 1` has the effect of flipping the least significant bit (LSB) of `right` that is set to 1 to 0, and if there are any bits to the right of this bit, it sets them to 1.

4. The bitwise AND operation `&=` then turns off the least significant bit that was just flipped in `right` because the LSB in `right` is 1, while the corresponding bit in `right - 1` is 0.

5. As a result of this operation, the `right` value becomes smaller with each iteration, which means we are effectively stripping off the lowest set bit one by one until `right` is no longer greater than `left`.

6. This iterative process continues to turn off the variable bits starting from least significant until all the bits that would change in the range `[left, right]` become 0.

7. The loop exits when `left` is the same as or greater than `right`. At this point, all the differentiating bits of `right` have been turned off, and the remaining common unchanged bits represent the result of the bitwise AND for the range.

8. The return statement `return right` gives us the final answer, which by now holds the bitwise AND of all numbers in the range `[left, right]`.

This implementation does not require any additional data structures and thus operates in-place with O(1) space complexity. The time complexity is based on the number of set bits in `right`, specifically, it is O(log n), where n is the value of `right`, since each iteration turns off one set bit.

No extra patterns or algorithms are used here, just bit manipulation to iteratively approach the solution. This approach is very efficient because it avoids the explicit bitwise AND of every two numbers in the range, which can be computationally expensive especially for large ranges.

Example Walkthrough

Let's illustrate the solution approach with a small example where `left = 10` and `right = 15`. In binary, these numbers are represented as:

`left = 10` = `1010` (in binary)
`right = 15` = `1111` (in binary)

Following the steps of the provided solution approach:

1. We check the condition `while left < right:`. Since `10 < 15`, we enter the loop.

2. The key operation of the loop is `right &= right - 1`. Initially, `right` is `15`, which is `1111` in binary. Then, `right - 1` is `14`, which is `1110` in binary.

3. The expression `right - 1` affects `right` by:

   - Flipping the least significant bit set to 1 to 0 (the rightmost bit in this case).
   - In the binary form `1110`, we see that it's the last digit that changes when subtracting 1 from `1111`.
4. Applying the bitwise AND operation:
   `1111` (right)
   `1110` (right - 1)
   `------`
   `1110` (result of right & (right - 1))

5. After the first iteration, `right` is now `14` (`1110` in binary).

6. We run the loop again since `left` (`1010` in binary) is still less than `right` (`1110` in binary).

7. Subtract 1 from `right`: `right - 1` = `13` = `1101` in binary.

8. Apply the bitwise AND operation again:
   `1110` (right)
   `1101` (right - 1)
   `------`
   `1100` (result of right & (right - 1))

9. `right` is now `12` (`1100` in binary), and we proceed with the loop.

10. Once more, as `left` (`1010` in binary) is still less than `right` (`1100` in binary), we continue.

    - `right - 1` = `11` = `1011` in binary.
    - The AND operation:
      `1100` (right)
      `1011` (right - 1)
      `------`
      `1000` (result of right & (right - 1))
11. `right` is now `8` (`1000` in binary), and `left` is `10` (`1010` in binary). Now, since `left` is not less than `right`, the loop ends.

12. The last value left in `right` which stands now at `8` (`1000` in binary) is the bitwise AND of all numbers from `10` to `15`.

We return `right` which is `8`. This value is indeed the bitwise AND of the range `[10, 15]` because any sequence of consecutive integers will eventually result in the bits that change within the range being set to 0, leaving the common unchanged bits as the result. In this case, `1000` in binary or `8` in decimal.

Python Solution

```python
class Solution:
    # This function finds the bitwise AND of all numbers within the inclusive range [left, right]
    def rangeBitwiseAnd(self, left: int, right: int) -> int:
        # Loop until 'left' is not less than 'right'
        while left < right:
            # We use the trick "right &= right - 1" to clear the least significant bit of 'right'
            # This effectively reduces 'right' each time, moving towards 'left'
            right &= right - 1

        # At the point where the while loop stops, 'left' and 'right' have the same prefix for
        # their binary representations which is the answer to the problem.
        return right

```

Java Solution

```java
class Solution {

    // Function to find the bitwise AND of all numbers in the range [left, right]
    public int rangeBitwiseAnd(int left, int right) {

        // Keep iterating until the right value is greater than or equal to the left value
        while (left < right) {
            // Goal is to remove the least significant bit of 'right' by doing 'right' & (right - 1)
            // to reduce right. This is because we know if left < right, the least significant
            // bits will eventually become zero due to the range of AND operations before we reach 'left'.
            right &= (right - 1);
        }

        // By the end, 'right' is modified such that it represents the common bits of
        // all numbers in the range [left, right], thus the bitwise AND result.
        return right;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int rangeBitwiseAnd(int m, int n) {
        // Loop until m is no longer less than n.
        while (m < n) {
            // The idea is to remove the rightmost bit of 'n' to make it smaller.
            // For example, n = 1010 (10 in decimal), n - 1 = 1001 (9 in decimal),
            // then n & (n - 1) = 1000 (8 in decimal), effectively removing the least significant bit.
            // This helps move 'n' closer to 'm' without affecting the result of bitwise AND of the range.
            n = n & (n - 1);
        }
        // Once m and n become equal, all bits to the right of the current bit position
        // have had differing values within the range [m, n], and therefore result in zero when ANDed.
        // The current value of 'n' is the bitwise AND of all numbers between the original 'm' and 'n'.
        return n;
    }
};

```

Typescript Solution

```typescript
/**
 * Applies a bitwise AND operation to all numbers within a given range.
 *
 * @param {number} left - The starting point of the range (inclusive).
 * @param {number} right - The end point of the range (inclusive).
 * @return {number} - The result of the bitwise AND operation applied sequentially from left to right.
 */
function rangeBitwiseAnd(left: number, right: number): number {
    // Loop until the 'left' value is greater than or equal to 'right'
    while (left < right) {
        // Apply bitwise AND between 'right' and 'right-1' then store it back in 'right'
        // This operation removes the least significant bit of 'right'
        right &= right - 1;
    }

    // Return the value of 'right' which at this point equals the bitwise AND of the range
    return right;
}

```

Time and Space Complexity

The provided code snippet is designed to find the bitwise AND of all numbers between two integers, `left` and `right` inclusive. The while loop continues to strip the least significant bit from `right` until `right` is no longer greater than `left`.

- **Time Complexity:** The time complexity of the algorithm is `O(log(n))`, where `n` is the distance between `left` and `right`. In the worst-case scenario, we will need to iterate through all bits of `right` until `right` becomes equal to `left`. Since an integer in Python has at most `log2(max_int)` bits, the loop runs at most `log2(right - left + 1)` times which simplifies to `O(log(n))`.

- **Space Complexity:** The space complexity of the algorithm is `O(1)`. This is because the algorithm only uses a constant amount of space, as it only modifies `right` without allocating any other data structures proportional to the size of the input range.