Documentation for the rangeBitwiseAnd Problem

Problem Overview:

- The problem is to compute the bitwise AND of all integers within a given inclusive range [left, right]. The bitwise AND operation on a set of integers results in a new integer where each bit is set to 1 if all the integers in the set have that bit set to 1, otherwise it is set to 0. The goal is to efficiently find the result of this operation for any two integers left and right.

Key Observations:

1. Effect of Bitwise AND on Consecutive Numbers:

- As the numbers increase within the range, more bits flip from 1 to 0. This means that the more the numbers differ, the less likely their bitwise AND will result in non-zero values, especially for large ranges.

- For example, the numbers between 5 and 7 are 5 (101), 6 (110), and 7 (111). The bitwise AND of these three numbers results in 100 (4 in decimal), because the last two bits differ between the numbers, but the first bit remains the same.

2. Common Bit Patterns:

- The result of the bitwise AND operation is determined by the common prefix of the binary representations of left and right. As numbers in the range differ, their binary representations lose matching bits from the right.

- The core insight is that the bitwise AND of a range of numbers will maintain only the bits that are the same in the binary representations of left and right. Once we encounter a difference, all subsequent bits (to the right) will be zero in the result.

3. **Bit Shifting to Identify the Common Prefix:**

- To find the common prefix, we can repeatedly right-shift both left and right until they become identical. Each right shift eliminates the least significant bit, which helps in focusing only on the common higher-order bits. Once they are the same, the remaining part is the common prefix.

- After identifying the common prefix, we need to shift it back to its original position (left-shifting by the number of shifts performed) to get the final result.

Approach Explanation:

The approach is based on the following steps:

1. **Right Shift Until Equal:**
   - We keep shifting both left and right to the right until they become the same number. This step progressively reduces the differences in the lower bits until only the common higher bits remain.

2. **Count the Number of Shifts:**
   - Every time we shift, we count how many times we've shifted. This is important because once left and right become equal, we need to shift the result back to its original position.

3. **Reconstruct the Result:**
   - After identifying the common prefix by shifting, the final result is obtained by shifting this common prefix back to the left by the number of times we shifted the numbers during the process. This restores the original significance of the bits.

Example 1:

- **Input:** left = 5, right = 7
- **Explanation:**
  - Binary representation of 5 is 101, 6 is 110, and 7 is 111.
  - The common bits between 5 and 7 are only the highest bit. After right-shifting both numbers until they become equal, we get 1.
  - We shift 1 back by the number of times we shifted (2 times), resulting in 4 (100 in binary).
- **Output:** 4.

Example 2:

- **Input:** left = 0, right = 0
- **Explanation:**
  - Since both left and right are already the same, no shifts are needed.
- **Output:** 0.

Example 3:

- **Input:** left = 1, right = 2147483647
- **Explanation:**
  - The binary representations of 1 and 2147483647 differ significantly in their higher bits. By shifting both numbers, the common bits will eventually be reduced to 0, since the range is too large.
- **Output:** 0.

Time Complexity:

- **Time Complexity: O(log(right))**
  - The number of shifts we need to perform is proportional to the number of bits in right, which is approximately log(right) in the worst case. This is efficient even for large values of right.

Space Complexity:

- **Space Complexity: O(1)**
  - The space complexity is constant because no extra space is required apart from a few variables to store the numbers and the shift count.

Constraints:

- 0 <= left <= right <= 2^{31} 1
- The range of inputs can be large, up to 2^31 1 (2147483647). The solution must handle this efficiently without iterating through all numbers in the range.

Edge Cases:

1. **Equal left and right:**
   - If left is equal to right, the result is simply left, as the range contains only one number.

2. **Large Range:**
   - If left is significantly smaller than right, especially when right is close to 2^31 1, the result is often 0 because all bits from the lower numbers will eventually differ from the higher numbers.

3. Zero as Input:

- If left or right is 0, the result will always be 0 because ANDing any number with 0 results in 0.

Conclusion:

- This problem involves efficiently computing the bitwise AND of a range of numbers by identifying the common bit prefix. Instead of iterating through the entire range, we use bitwise operations and shifting to reduce the problem to a simpler form, making the solution highly efficient for large inputs.