371. Sum of Two Integers

Medium Bit Manipulation Math

Problem Description

The problem presents a situation where we need to calculate the sum of two integers `a` and `b`. However, the challenge here is that we are restricted from using the usual addition and subtraction operators, which are `+` and `-`. This requires a different approach to perform the addition operation by means other than direct arithmetic operators.

Intuition

The intuition behind the solution is based on bitwise operations. Since we cannot use the `+` and `-` operators, we need to simulate the addition process. A common way to achieve this is by using bitwise XOR and AND operations. The XOR operation can be used to simulate the addition without carrying, and the AND operation followed by a left shift can be used to determine the carry from the addition.

Here's a step-by-step process of the thinking:

1. **XOR for Addition**: The XOR operation gives a binary number that represents the summation of two bits where carrying is not accounted for. For example, 1 XOR 1 is 0, 1 XOR 0 is 1, which aligns with addition of bits without carrying.
2. **AND for Carry**: The AND operation gives a binary number that represents the bits that would need to be carried in a standard addition operation. For example, 1 AND 1 is 1, which means there is a carry.
3. **Shift for Carry Position**: The carried value from the AND operation is then left-shifted one bit because the carry from any particular bit would go to the next higher bit in a binary addition.
4. **Iteration Until No Carry**: We need to repeat this process until there is no carry left, i.e., the result of the AND operation is 0.
5. **Masking with 0xFFFFFFFF**: To ensure compatibility with negative numbers and to simulate 32-bit integer overflow behavior, bitwise operations are performed with a mask of 0xFFFFFFFF.
6. **Check for Negative Result**: If the result has its leftmost bit (bit 31) set, it is negative (in 32-bit two's complement form). To convert it to a negative number that Python recognizes, the inverse operation is applied (via `~(a ^ 0xFFFFFFFF)`).

Considering the above intuition, the solution approach simulates the addition of two integers using bitwise operations that reflect the mechanics of binary addition while respecting the constraints of 32-bit signed integer overflow.

The implementation of the solution uses bitwise operations to mimic the addition process. Below is a detailed walk-through of how the provided code implements the algorithm:

1. **Masking for 32-bit Integer Overflow**: The code starts by initializing `a` and `b` with a mask of `0xFFFFFFFF`:

   ```
   a, b = a & 0xFFFFFFFF, b & 0xFFFFFFFF
   ```

   This ensures that we're only considering the lower 32 bits of the integers, which simulates the overflow behavior of a 32-bit machine. In a 32-bit environment, integers wrap around when they exceed their maximum or minimum limits, and bitwise operations in Python can produce similar results by using this mask.

2. **Loop for Iterative Addition**: Next, a `while` loop is started, which continues as long as `b` is non-zero:

   ```
   while b:
   ```

   The value of `b` holds the carry bits that need to be added to `a`. The loop iterates until there are no carry bits left to add.

3. **Calculating Carry**: Inside the loop, the carry is calculated by performing a bitwise AND between `a` and `b`, and then left shifting the result by 1 bit:

   ```
   carry = ((a & b) << 1) & 0xFFFFFFFF
   ```

   The result is again masked to ensure it stays within 32 bits.

4. **Adding Without Carry**: The XOR operation is then used to add `a` and `b` without considering the carry:

   ```
   a, b = a ^ b, carry
   ```

   Here, `a` now contains the intermediate sum without the carry bits, while `b` takes on the value of the carry bits that need to be added in the next iteration.

5. **Handling Negative Numbers**: Once the loop completes (when there is no carry left), the function checks if the result is a negative number:

   ```
   return a if a < 0x80000000 else ~(a ^ 0xFFFFFFFF)
   ```

   If the most significant bit of `a` (bit 31) is not set, `a` is returned directly. If it is set, it indicates a negative number in 32-bit two's complement form, so the bitwise complement operator `~` is used in conjunction with `^(XOR)` to get the negative integer that Python can interpret correctly.

The code leverages the concept of two's complement for negative numbers. In two's complement, a negative number is represented by inverting all the bits of its positive counterpart and then adding 1 to the least significant bit (which happens here indirectly through the XOR and AND operations).

No additional data structures are used in this implementation; the solution is carried out with simple integer variables and bitwise operations.

In conclusion, this implementation carefully applies fundamental binary arithmetic operations in a loop to achieve the same result as the addition operator while conforming to the specified constraints relating to arithmetic operators and integer overflow behavior.

Example Walkthrough

Let's take smaller, more manageable integers to explain the logic clearly. We will add two integers, `a = 13` and `b = 8`, without using the `+` and `-` operators. In binary form, `a` is `1101` and `b` is `1000`.

1. Mask both `a` and `b` with `0xFFFFFFFF`. For the value ranges in our example, this has no effect because both `a` and `b` are positive and within the 32-bit limit.

   ```
   a, b = 13 & 0xFFFFFFFF, 8 & 0xFFFFFFFF
   ```

2. Enter the `while` loop where `b` is not zero. Initially, `b` is `1000` (or `8`), so the loop commences.

3. Calculate the carry by ANDing `a` and `b`, then left shifting by one (`<<1`):

   ```
   carry = ((a & b) << 1) & 0xFFFFFFFF
   # carry = ((1101 & 1000) << 1) & 0xFFFFFFFF
   # carry = (1000 << 1) & 0xFFFFFFFF
   # carry = 10000 & 0xFFFFFFFF
   # carry = 16, in binary: 10000
   ```

4. Perform the addition without carry using `XOR`, and update the carry from step 3:

   ```
   a, b = a ^ b, carry
   # a = 1101 ^ 1000
   # a = 0101, which is 5 in decimal
   # b = carry = 10000, which is 16 in decimal
   ```

5. Since `b` is still not zero, the loop continues with the new values of `a` and `b`.

6. Inside the loop again:

   ```
   carry = ((a & b) << 1) & 0xFFFFFFFF
   # carry = ((0101 & 10000) << 1) & 0xFFFFFFFF
   # carry = (00000 << 1) & 0xFFFFFFFF
   # carry = 00000 & 0xFFFFFFFF
   # carry = 0, in binary: 00000
   ```

   And for the `XOR` operation:

   ```
   a, b = a ^ b, carry
   # a = 0101 ^ 10000
   # a = 10101, which is 21 in decimal
   # b = carry = 00000, which is 0 in decimal
   ```

7. Since `b` is now zero, we exit the loop. We check if `a` is negative by looking at bit 31 (it is not in our case).

8. Return `a` because the most significant bit of `a` is not set, implying that the result is a positive number:

   ```
   return a # which is 21 in our example
   ```

Thus, we have successfully added `a` and `b` using bitwise operations, simulating their binary addition process without using the `+` and `-` operators. This example demonstrates how we obtain the sum `21`, which we expect from adding `13` and `8`.

Solution Implementation

Python

```
class Solution:
    def getSum(self, a: int, b: int) -> int:
        # Mask to get 32-bit representation
        MASK = 0xFFFFFFFF

        # Convert a and b to 32-bit integers
        a, b = a & MASK, b & MASK

        # Iterate while there is a carry
        while b:
            # Calculate the carry from a and b,
            # and ensure it's within the 32-bit boundary
            carry = ((a & b) << 1) & MASK

            # XOR a and b to find the sum without considering carries,
            # then consider the carry for the next iteration
            a, b = a ^ b, carry

        # If a is within the 32-bit integer range, return it directly
        # The operation `a < 0x80000000` checks if the result is within the positive range
        # of a 32-bit integer (from 0x80000000 to 0x7FFFFFFF)
        if a < 0x80000000:
            return a

        # If a is outside the 32-bit integer range, which means it is negative,
        # return the two's complement negative value, which is the bitwise negation of a
        # (by XORing with MASK, which is 'all 1's for a 32-bit number') and adding 1
        return ~(a ^ MASK)
```

Java

```
class Solution {
    public int getSum(int a, int b) {
        // If there is no carry (b == 0), then we have found the sum and can return 'a'
        if (b == 0) {
            return a;
        } else {
        // If there is a carry, recursively call getSum with the following parameters:
        // 1. a XOR b - This calculates the sum without carry
        // 2. (a AND b) << 1 - This calculates the carry and shifts it one bit to the left, as carry affects the next higher bit
            return getSum(a ^ b, (a & b) << 1);
        }
    }
}
```

C++

```
class Solution {
public:
    int getSum(int a, int b) {
        // Loop until there is no carry
        while (b != 0) {
            // Calculate the carry and cast to unsigned int to handle overflow
            unsigned int carry = (unsigned int)(a & b) << 1;

            // Perform XOR operation to add 'a' and 'b' without the carry
            a = a ^ b;

            // Assign the carry to 'b' for the next iteration
            b = carry;
        }

        // Once 'b' is zero, there is no carry and 'a' has the sum result; return it.
        return a;
    }
};
```

TypeScript

```
function getSum(a: number, b: number): number {
    // Loop until there is no carry.
    while (b !== 0) {
        // Calculate the carry, and use >>> 0 to ensure it is treated as an unsigned 32-bit integer.
        let carry: number = (a & b) << 1 >>> 0;

        // Perform XOR operation (add without carry).
        a = a ^ b;

        // Assign the carry to 'b' for the next iteration.
        b = carry;
    }

    // Once 'b' is zero, there is no carry and 'a' has the sum; return 'a'.
    return a;
}


class Solution:
    def getSum(self, a: int, b: int) -> int:
        # Mask to get 32-bit representation
        MASK = 0xFFFFFFFF

        # Convert a and b to 32-bit integers
        a, b = a & MASK, b & MASK

        # Iterate while there is a carry
        while b:
            # Calculate the carry from a and b,
            # and ensure it's within the 32-bit boundary
            carry = ((a & b) << 1) & MASK

            # XOR a and b to find the sum without considering carries,
            # then consider the carry for the next iteration
            a, b = a ^ b, carry

        # If a is within the 32-bit integer range, return it directly
        # The operation `a < 0x80000000` checks if the result is within the positive range
        # of a 32-bit integer (from 0x80000000 to 0x7FFFFFFF)
        if a < 0x80000000:
            return a

        # If a is outside the 32-bit integer range, which means it is negative,
        # return the two's complement negative value, which is the bitwise negation of a
        # (by XORing with MASK, which is 'all 1's for a 32-bit number') and adding 1
        return ~(a ^ MASK)
```

Time and Space Complexity

The given Python code is an implementation of bit manipulation to calculate the sum of two integers without using the `+` operator. To analyze the time and space complexity, we'll consider each operation within the loop:

Time Complexity:

The while loop in the code runs as long as `b` is not zero. Each iteration involves constant-time bit operations: `AND`, `SHIFT LEFT`, and `XOR`. These operations themselves have a time complexity of $O(1)$. However, the number of iterations of the loop depends on the values of `a` and `b` and when the carry will become 0. In the worst case, it will iterate approximately `log C` times where `C` is the maximum possible value for `carry`. Considering 32-bit integers, the loop runs at most 32 times (the number of bits), thus the time complexity can be considered $O(1)$ for 32-bit integers, since it does not depend on the size of the input, but rather on the fixed number of bits used to represent the integers in memory.

Space Complexity:

The space complexity of the algorithm is $O(1)$ as well, as it uses a fixed number of integer variables (`a`, `b`, `carry`) regardless of the input size. There is no use of dynamic memory allocation or data structures that grow with input size.