29. Divide Two Integers

Medium Bit Manipulation Math

Problem Description

The problem at hand requires us to divide two integers, `dividend` and `divisor`, without using multiplication, division, and mod operator. The result should be the integer part of the quotient, with the division result truncated towards zero, meaning that any fractional part is disregarded. This operation should be handled with care as the problem also specifies dealing with overflows by capping the return value at 32-bit signed integer limits.

In essence, we are to implement a form of division that replicates how integer division works in programming languages where the result is truncated towards zero, ensuring we work within the 32-bit integer range. We have to be cautious, as the direct operations that normally achieve this (/, *, %) are not permitted.

Intuition

The solution relies on the concept of subtraction and bit manipulation to accomplish the division. Since we can't use the division operator, we think about what division actually means. Division is essentially repeated subtraction. For instance, when we say 10 divided by 2, it means how many times we can subtract 2 from 10 until we reach a number less than 2.

The intuition behind the solution is to use a subtraction-based approach where we keep subtracting the divisor from the dividend and count how many times we can do this until the dividend is less than the divisor. This is a valid first step but not efficient enough for large numbers, which is where bit manipulation comes in handy.

To improve efficiency, instead of subtracting the divisor once at a time, we exponentially increase the subtracted amount by left shifting the divisor (which is equivalent to multiplying by powers of 2) and subtract this from the dividend if possible. This approach is much faster, as left shifting effectively doubles the subtracted amount each time, allowing us to subtract large chunks in logarithmic time compared to a linear approach.

The whole process loops, increasing the amount being subtracted each time (as long as the double of the current subtraction amount is still less than or equal to the remaining dividend) and adding the corresponding power of 2 to our total. This loop represents a divide and conquer strategy that works through the problem using bit-level operations to mimic standard division while ensuring the result stays within the specified integer range.

Time and space complexity is considered, especially since we are working within a constrained environment that doesn't allow typical operations. The time complexity here is O(log(a) * log(b)), with 'a' being the dividend and 'b' the divisor. This complexity arises because the algorithm processes parts of the dividend in a time proportional to the logarithm of its size, and likewise for the divisor since the subtraction step is proportional to its logarithm as well. The space complexity is constant, O(1), since we use a fixed number of variables regardless of the size of the inputs.

Solution Approach

The approach to solving the division problem without multiplication, division, or mod operator involves a few key steps and utilizes simple yet powerful concepts of bit manipulation to efficiently find the quotient.

Here's a step-by-step walkthrough of the implementation:

1. **Handle Signs:**
   - First, we need to handle the sign of the quotient. If the `dividend` and `divisor` have the same sign, the result is positive; otherwise, it's negative. We define the variable `sign` and use a simple comparison to set its value to `-1` for a negative result or `1` for a positive one.
   - The actual division operation will be conducted on the absolute values of the `dividend` and `divisor`.
2. **Initialize Variables:**
   - Set `INT_MAX` as `(1 << 31) - 1`, which represents the maximum positive value for a 32-bit integer.
   - Set `INT_MIN` as `-(1 << 31)`, representing the minimum negative value for a 32-bit integer.
   - Initialize the quotient `tot` to `0`.
3. **Bitwise Shift for Division:**
   - We start a loop where we continue to subtract the `divisor` from the `dividend` until `dividend` is smaller than `divisor`. For each subtraction:
     - Initialize a counter `cnt` with `0`.
     - Inside an inner loop, left shift the `divisor` by `cnt + 1` positions, effectively multiplying the `divisor` by `2` each time, until it would exceed the current `dividend`.
     - After finding the maximum amount by which we can multiply the `divisor` without exceeding the `dividend`, we add `1 << cnt` to our running total `tot`. This is equivalent to adding `2^cnt`.
     - Reduce the `dividend` by `divisor << cnt`.
4. **Finalizing Result:**
   - Multiply the `tot` by the `sign` to apply the correct sign to the result.
   - Handle potential integer overflow by comparing the result against `INT_MAX` and `INT_MIN`:
     - If the result is within the range `[INT_MIN, INT_MAX]`, return the result.
     - If the result exceeds the range, return `INT_MAX`.

This algorithm effectively simulates division by breaking it down into a combination of subtraction and left bitwise shift operations, replicating multiplication by powers of 2. It's a logarithmic solution in the sense that it reduces the problem size by approximately half with each recursive subtraction, hence the `O(log a * log b)` time complexity, where `a` is the dividend and `b` is the divisor. Space complexity is `O(1)` since the number of variables used does not scale with input size.

The pattern used here can be thought of as a "divide and conquer" as well as "bit manipulation". By using these principles, we can efficiently and accurately divide two integers in a constrained environment.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach as described. Suppose our `dividend` is 10 and our `divisor` is 3. We need to find out how many times we can subtract 3 from 10, with the operations restricted as per the problem statement.

1. Handle Signs:

- Both the `dividend` (10) and `divisor` (3) are positive, so our result will also be positive. Thus, `sign = 1`.

2. Initialize Variables:

- `INT_MAX` is set as `(1 << 31) - 1` and `INT_MIN` is set as `-(1 << 31)`.
- Initialize total quotient `tot = 0`.

3. Bitwise Shift for Division:

- Begin the loop to subtract `divisor` from `dividend`.
  - On the first iteration, `cnt = 0`. We check if `(divisor << cnt + 1) <= dividend`:
    - `3 << 0` is `3`, and `3 << 1` (which is `6`) is still `<= 10 (dividend)`.
    - `3 << 2` would be `12`, which exceeds `10`. So, we can stop at `cnt = 1`.
  - Add `1 << cnt` which is `1 << 1` or `2` to `tot`.
  - Subtract `divisor << cnt` which is `3 << 1` or `6` from `dividend`. Now, `dividend = 10 - 6 = 4`.
- With the new `dividend` of 4, repeat the process:
  - On the second iteration, `cnt = 0`. `3 << cnt + 1` is `6`, which is greater than `4`. So we can't shift `cnt` to 1 this time.
  - Add `1 << cnt` which is `1` to `tot`. Now, `tot = 2 + 1 = 3`.
  - Subtract `divisor << cnt` (which is `3`) from `dividend`. Now, `dividend = 4 - 3 = 1`.

4. Finalizing Result:

- Because `dividend` is now less than `divisor`, we can conclude our calculation.
- Since we began with `tot = 0` and added `2` first and then `1` to it, we have `tot = 3`.
- Multiply `tot` by `sign`. Since `sign = 1`, the result remains `tot = 3`.
- Check for overflow, which isn't the case here, so the final result is `3`.

Thus, dividing 10 by 3 yields a quotient of `3` using this approach. Since we are only concerned with the integer part of the division, the remainder is disregarded, aligning with the truncation towards zero rule. The bit manipulation significantly speeds up the process by allowing us to subtract larger powers of 2 wherever possible.

Python Solution

```python
class Solution:
    def divide(self, dividend: int, divisor: int) -> int:
        # Define the boundaries for an integer (32-bit signed integer)
        INT_MAX = 2**31 - 1
        INT_MIN = -2**31

        # Determine the sign of the output. If dividend and divisor have different signs, result will be negative
        sign = -1 if (dividend * divisor) < 0 else 1

        # Work with positive values for both dividend and divisor
        dividend = abs(dividend)
        divisor = abs(divisor)

        # Initialize the total quotient
        total_quotient = 0

        # Loop to find how many times the divisor can fit into the dividend
        while dividend >= divisor:
            # Count will keep track of the number of times we can double the divisor while still being less than or equal to dividenc
            count = 0
            # Double the divisor as much as possible without exceeding the dividend
            while dividend >= (divisor << (count + 1)):
                count += 1
            # Increment total_quotient by the number of times we doubled the divisor
            total_quotient += 1 << count
            # Decrease dividend by the matched part which we just calculated
            dividend -= divisor << count

        # Multiply the result by the sign
        result = sign * total_quotient

        # Check and correct for overflow: if result is out of the 32-bit signed integer range, clamp it to INT_MAX
        if result < INT_MIN:
            return INT_MIN
        elif result > INT_MAX:
            return INT_MAX
        else:
            return result

```

Java Solution

```java
class Solution {
    public int divide(int dividend, int divisor) {
        // Determine the sign of the result
        int sign = 1;
        if ((dividend < 0) != (divisor < 0)) {
            sign = -1;
        }

        // Use long to avoid integer overflow issues
        long longDividend = Math.abs((long) dividend);
        long longDivisor = Math.abs((long) divisor);

        // This will accumulate the result of the division
        long total = 0;

        // Loop to find how many times the divisor can be subtracted from the dividend
        while (longDividend >= longDivisor) {
            // This counter will keep track of the number of left shifts
            int count = 0;

            // Double the divisor until it is less than or equal to the dividend
            while (longDividend >= (longDivisor << (count + 1))) {
                count++;
            }

            // Add the number of times we could double the divisor to the total
            total += 1L << count;

            // Subtract the final doubled divisor value from the dividend
            longDividend -= longDivisor << count;
        }

        // Multiply the sign back into the total
        long result = sign * total;

        // Handle overflow cases by clamping to the Integer range
        if (result >= Integer.MIN_VALUE && result <= Integer.MAX_VALUE) {
            return (int) result;
        }

        // If the result is still outside the range, return the max integer value
        return Integer.MAX_VALUE;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int divide(int dividend, int divisor) {
        // Determine sign of the result based on the signs of dividend and divisor
        int resultSign = (dividend < 0) ^ (divisor < 0) ? -1 : 1;

        // Use long long to avoid overflow issues for abs(INT32_MIN)
        long long absDividend = abs(static_cast<long long>(dividend));
        long long absDivisor = abs(static_cast<long long>(divisor));
        long long result = 0; // Initialize result

        // Loop until the dividend is smaller than divisor
        while (absDividend >= absDivisor) {
            int shiftCount = 0; // Count how many times the divisor has been left-shifted

            // Find the largest shift where the shifted divisor is smaller than or equal to dividend
            while (absDividend >= (absDivisor << (shiftCount + 1))) {
                ++shiftCount;
            }

            // Add to the result the number represented by the bit at the found position
            result += 1ll << shiftCount;

            // Reduce dividend by the found multiple of divisor
            absDividend -= absDivisor << shiftCount;
        }

        // Apply sign of result
        result *= resultSign;

        // Handle overflow by returning INT32_MAX if the result is not within int range
        if (result >= INT32_MIN && result <= INT32_MAX) {
            return static_cast<int>(result);
        }
        return INT32_MAX;
    }
};

```

Typescript Solution

```typescript
// Global function for division without using division operator
function divide(dividend: number, divisor: number): number {
    // Determine sign of the result based on the signs of dividend and divisor
    let resultSign: number = (dividend < 0) ^ (divisor < 0) ? -1 : 1;

    // Use number to accommodate for JavaScript's safe integer range
    // and to avoid precision issues with bitwise operations
    let absDividend: number = Math.abs(dividend);
    let absDivisor: number = Math.abs(divisor);
    let result: number = 0; // Initialize result

    // Loop until the dividend is smaller than divisor
    while (absDividend >= absDivisor) {
        let shiftCount: number = 0; // Count how many times the divisor has been multiplied by 2

        // Find the largest multiple of 2 for divisor that is still less than or equal to dividend
        while (absDividend >= (absDivisor * Math.pow(2, shiftCount + 1))) {
            shiftCount++;
        }

        // Accumulate the quotient by the power of 2 corresponding to the shift count
        result += Math.pow(2, shiftCount);

        // Decrease dividend by the found multiple of the divisor
        absDividend -= absDivisor * Math.pow(2, shiftCount);
    }

    // Apply the sign of the result
    result *= resultSign;

    // Handle overflow by returning the maximum safe integer value if the result is not within 32-bit signed integer range
    if (result >= -(2 ** 31) && result <= (2 ** 31) - 1) {
        return Math.trunc(result);
    }
    return (2 ** 31) - 1;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(log(a) * log(b))`. This is because in the first `while` loop, we're checking if `a` is greater than or equal to `b`, which requires `O(log(a))` time since in each iteration `a` is reduced roughly by a factor of two or more. The inner `while` loop is responsible for finding the largest shift of `b` that `a` can handle, which will execute at most `O(log(b))` times, as shifting `b` left by one doubles its value, and `cnt` increases until `a` is no longer greater than `b` shifted by `cnt + 1`. Therefore, these two loops combined yield the time complexity mentioned.

Space Complexity

The space complexity of the given code is `O(1)`. Only a fixed number of integer variables `sign`, `tot`, and `cnt` are used, which do not depend on the size of the input. Hence, the space used is constant.