29. Divide Two Integers

Medium Bit Manipulation **Math**

The problem at hand requires us to divide two integers, dividend and divisor, without using multiplication, division, and mod

Problem Description

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.

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

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.

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.

To improve efficiency, instead of subtracting the divisor once at a time, we exponentially increase the subtracted amount by left

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

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

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:

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.

Handle Signs:

 The actual division operation will be conducted on the absolute values of the dividend and divisor. **Initialize Variables:** Set INT_MAX as (1 << 31) − 1, which represents the maximum positive value for a 32-bit integer.

• 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

• We start a loop where we continue to subtract the divisor from the dividend until dividend is smaller than divisor. For each

■ Reduce the dividend by divisor << cnt.

running total tot. This is equivalent to adding 2^cnt.

Multiply the tot by the sign to apply the correct sign to the result.

Initialize the quotient tot to 0.

 \circ Set INT_MIN as -(1 << 31), representing the minimum negative value for a 32-bit integer.

while ensuring the result stays within the specified integer range.

a fixed number of variables regardless of the size of the inputs.

- **Bitwise Shift for Division:**
 - 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

Finalizing Result:

Example Walkthrough

1. Handle Signs:

 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 $0(\log a * \log b)$ time complexity, where a is the dividend and b

is the divisor. Space complexity is 0(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.

• 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).

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.

■ 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.

3. Bitwise Shift for Division:

Initialize total quotient tot = 0.

Begin the loop to subtract divisor from dividend.

Add 1 << cnt which is 1 << 1 or 2 to tot.

Add 1 << cnt which is 1 to tot. Now, tot = 2 + 1 = 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.

• With the new dividend of 4, repeat the process:

∘ Subtract divisor << cnt (which is 3) from dividend. Now, dividend = 4 - 3 = 1.

• 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.

∘ Subtract divisor << cnt which is 3 << 1 or 6 from dividend. Now, dividend = 10 - 6 = 4.

On the first iteration, cnt = 0. We check if (divisor << cnt + 1) <= dividend:</p>

Since we began with tot = 0 and added 2 first and then 1 to it, we have tot = 3.

the process by allowing us to subtract larger powers of 2 wherever possible.

• Because dividend is now less than divisor, we can conclude our calculation.

Python

 $INT_MIN = -2**31$

dividend = abs(dividend)

Initialize the total quotient

dividend -= divisor << count

if ((dividend < 0) != (divisor < 0)) {</pre>

while (longDividend >= longDivisor) {

// Use long to avoid integer overflow issues

long longDivisor = Math.abs((long) divisor);

longDividend -= longDivisor << count;</pre>

// Use number to accommodate for JavaScript's safe integer range

// Decrease dividend by the found multiple of the divisor

Define the boundaries for an integer (32-bit signed integer)

Work with positive values for both dividend and divisor

while dividend >= (divisor << (count + 1)):</pre>

Loop to find how many times the divisor can fit into the dividend

Double the divisor as much as possible without exceeding the dividend

Increment total quotient by the number of times we doubled the divisor

Decrease dividend by the matched part which we just calculated

not depend on the size of the input. Hence, the space used is constant.

absDividend -= absDivisor * Math.pow(2, shiftCount);

if (result $>= -(2 ** 31) \&\& result <= (2 ** 31) - 1) {$

def divide(self, dividend: int, divisor: int) -> int:

sign = -1 if (dividend * divisor) < 0 else 1

while (absDividend >= (absDivisor * Math.pow(2, shiftCount + 1))) {

// Accumulate the quotient by the power of 2 corresponding to the shift count

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

// Handle overflow by returning the maximum safe integer value if the result is not within 32-bit signed integer range

Determine the sign of the output. If dividend and divisor have different signs, result will be negative

Count will keep track of the number of times we can double the divisor while still being less than or equal to dividend

// and to avoid precision issues with bitwise operations

let absDividend: number = Math.abs(dividend);

let absDivisor: number = Math.abs(divisor);

result += Math.pow(2, shiftCount);

while (absDividend >= absDivisor) {

shiftCount++;

// Apply the sign of the result

return Math.trunc(result);

result *= resultSign;

return (2 ** 31) - 1;

INT MAX = 2**31 - 1

dividend = abs(dividend)

while dividend >= divisor:

count += 1

total quotient += 1 << count

dividend -= divisor << count</pre>

Multiply the result by the sign

result = sign * total_quotient

Initialize the total quotient

divisor = abs(divisor)

total_quotient = 0

count = 0

 $INT_MIN = -2**31$

class Solution:

let result: number = 0; // Initialize result

// Loop until the dividend is smaller than divisor

// Multiply the sign back into the total

long longDividend = Math.abs((long) dividend);

// This will accumulate the result of the division

// Loop to find how many times the divisor can be subtracted from the dividend

// Double the divisor until it is less than or equal to the dividend

// Add the number of times we could double the divisor to the total

// Subtract the final doubled divisor value from the dividend

// This counter will keep track of the number of left shifts

while (longDividend >= (longDivisor << (count + 1))) {</pre>

Multiply the result by the sign

result = sign * total_quotient

if result < INT MIN:</pre>

return INT MIN

elif result > INT MAX:

return INT_MAX

return result

divisor = abs(divisor)

total_quotient = 0

4. Finalizing Result:

Solution Implementation

Work with positive values for both dividend and divisor

Loop to find how many times the divisor can fit into the dividend

Decrease dividend by the matched part which we just calculated

sign = -1 if (dividend * divisor) < 0 else 1

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

Determine the sign of the output. If dividend and divisor have different signs, result will be negative

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)):</pre> count += 1# Increment total quotient by the number of times we doubled the divisor total quotient += 1 << count

Check and correct for overflow: if result is out of the 32-bit signed integer range, clamp it to INT_MAX

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

Java class Solution { public int divide(int dividend, int divisor) { // Determine the sign of the result

int sign = 1;

sign = -1;

long total = 0;

int count = 0;

count++;

total += 1L << count;

long result = sign * total;

else:

```
// Handle overflow cases by clamping to the Integer range
        if (result >= Integer.MIN_VALUE && result <= Integer.MAX_VALUE) {</pre>
            return (int) result;
        // If the result is still outside the range, return the max integer value
        return Integer.MAX_VALUE;
C++
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) \land (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))) {</pre>
                ++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;</pre>
        // 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) {</pre>
            return static_cast<int>(result);
        return INT32_MAX;
};
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;
```

```
# 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:</pre>
    return INT MIN
elif result > INT MAX:
    return INT_MAX
else:
    return result
```

Time and Space Complexity

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.

Time Complexity

Therefore, these two loops combined yield the time complexity mentioned. **Space Complexity** The space complexity of the given code is 0(1). Only a fixed number of integer variables sign, tot, and cnt are used, which do

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