7. Reverse Integer

Medium Math

Problem Description

The task is to take a signed 32-bit integer x and reverse the order of its digits. For example, if the input is 123, the output should be 321. If the input is -123, the output should be -321. The tricky part comes with the boundaries of a 32-bit signed integer, which ranges from -2^31 to 2^31 - 1. If reversing the digits of x would cause the number to fall outside this range, the function should return 0 instead. This means we need to be careful with overflow—an issue that occurs when the reversed integer is too large or too small to be represented by a 32-bit signed integer.

Intuition

signed integer, respectively. These values are -2^31 and 2^31 - 1. We want to build the reversed number digit by digit. We can isolate the last digit of x by taking x % 10 (the remainder when x is

To solve this problem, we first set up two boundaries, mi and mx, which represent the minimum and maximum values of a 32-bit

divided by 10). This last digit, referred to as y in our code, is the next digit to be placed in the reversed number. However, we need to be careful not to cause an overflow when we add this new digit to the reversed number. Before we add y to

the reversed number ans, we check if adding the digit would cause an overflow. To do this, we check if ans is either less than mi / 10 + 1 or greater than mx / 10. If it's outside this range, we return 0.

If it's safe to add the digit, we proceed. We add the digit to ans by multiplying ans by 10 (which "shifts" the current digits to the left) and then adding y. This process effectively reverses the digits of x.

For the next iteration, we need to remove the last digit from x. We do this by subtracting y from x and then dividing by 10. We repeat this process until x has no more digits left. The result is a reversed number that fits within the 32-bit signed integer

range, or 0 if an overflow would have occurred. The time complexity is $0(\log |x|)$ because the process continues for as many digits as x has, and the space complexity is

0(1) as there is a constant amount of memory being used regardless of the size of x.

Solution Approach

The implementation uses a straightforward algorithm that iterates through the digits of the input number x and constructs the

Initialization: We start by setting the initial reversed number ans to 0. We also define the minimum and maximum values mi

reversed number without using additional data structures or complex patterns. Let's detail the steps using the provided Reference Solution Approach:

dividing by 10.

and mx for a 32-bit signed integer, which are -2^31 and $2^31 - 1$. **Reversing Digits:** The while loop runs as long as there are digits left in x. Within the loop, we take the following steps:

- Isolate the last digit y of x by computing x % 10. \circ If x is negative and y is positive, adjust y by subtracting 10 to make it negative. Checking for Overflow: Before appending y to ans, we must confirm that ans *10 + y will not exceed the boundaries set
- by mi and mx. To avoid overflow, we check:
- ∘ If ans is less than mi/10 + 1 or greater than mx/10, we return 0 immediately, as adding another digit would exceed the 32-bit signed integer limits.
- Building the Reversed Number: If it is safe to proceed, we multiply ans by 10 (which shifts the reversed number one place to the left) and add y to ans. This action reverses y from its position in x to its new reversed position in ans.
- **Completion**: The loop repeats this process, accumulating the reversed number in lans until all digits are processed. The core of this approach is predicated on the mathematical guarantees regarding integer division and modulus operations in

Updating the Original Number x: We update x by removing its last digit. This is done by subtracting y from x and then

and only proceeds if the operation stays within bounds. By following the constraints of a 32-bit signed integer at every step and efficiently using arithmetic operations, the reverse

Python. The guard checks for overflow by considering both scale (multiplication by 10) and addition (adding the digit) separately

To illustrate the solution approach, let's take x = 1469 as our example.

Reversing Digits: Begin while loop with x = 1469.

Example Walkthrough

Initialization:

o ans is initialized to 0.

∘ mi is set to -2^31 (-2147483648).

Building the Reversed Number:

Updating the Original Number x:

Next iteration of the loop with x = 146:

• Check for overflow: ans = 9 is still within bounds.

mx is set to 2³¹ - 1 (2147483647).

 Isolate the last digit y by computing 1469 % 10 = 9. \circ x is positive so we keep y = 9.

We multiply ans by 10, which is still 0, and add y to get ans = 9.

function achieves the reversal of digits robustly and efficiently.

- **Checking for Overflow:** o ans is currently 0, which is greater than mi/10 + 1 (-214748364) and less than mx/10 (214748364), so continue without returning 0.

• Isolate y = 146 % 10 = 6.

• Isolate y = 14 % 10 = 4.

- \circ Update x to remove the last digit: x = (1469 9) / 10 which simplifies to x = 146.
- Update ans: ans = 9 * 10 + 6 = 96. • Update x: x = (146 - 6) / 10 which simplifies to x = 14.
- Next iteration with x = 14:
- Check for overflow: ans = 96 is still within bounds.
- Update ans: ans = 96 * 10 + 4 = 964. • Update x: x = (14 - 4) / 10 which simplifies to x = 1.
- Check for overflow: ans = 964 is still within bounds. • Update ans: ans = 964 * 10 + 1 = 9641.

process for each digit.

Python

class Solution:

while x:

return 0

x = (x - digit) // 10

return reversed_number

while (x != 0) {

x /= 10;

return 0;

// Return the reversed number

#include <climits> // For INT_MIN and INT_MAX

const INT_MAX: number = 2 ** 31 - 1;

reversed = reversed * 10 + (x % 10);

// Check for potential overflow by comparing with pre-divided limits.

if (reversed < Math.floor(INT_MIN / 10) || reversed > Math.floor(INT_MAX / 10)) {

// Perform the reverse by multiplying the current reversed by 10 and adding the last digit of x.

let reversed: number = 0;

return 0;

while (x !== 0) {

return reversedNumber;

Solution Implementation

• Isolate y = 1 % 10 = 1.

Final iteration with x = 1:

Now x = 0, the while loop terminates, and the reversed number ans = 9641 is returned. There were no issues with overflow throughout the process, so the result 9641 is the correct reversed integer for our example of x = 1469.

• Update x: x = (1 - 1) / 10 which simplifies to x = 0.

def reverse(self, x: int) -> int: # This variable will hold the reversed number reversed_number = 0 # These define the range of acceptable 32-bit signed integer values $min_int, max_int = -2**31, 2**31 - 1$

Check if the reversed number will overflow when multiplied by 10

Return 0 on overflow as per problem constraints

Return the reversed number within the 32-bit signed integer range

// Check for overflow/underflow condition, return 0 if violated

// Integer.MIN VALUE is -2^31 and Integer.MAX VALUE is 2^31 - 1

if (reversedNumber < Integer.MIN_VALUE / 10 || reversedNumber > Integer.MAX_VALUE / 10) {

reversed_number = reversed_number * 10 + digit

Remove the least significant digit from x

// Add the last digit of x to reversedNumber

// Remove the last digit from x

reversedNumber = reversedNumber * 10 + x % 10;

if reversed number < min int // 10 + 1 or reversed number > max_int // 10:

This process demonstrates that by evaluating the overflow conditions before each digit is added to the reversed number, and by

building the reversed number step by step, it's possible to safely reverse the digits of x without using extra space or complex

data structures. Additionally, due to the use of modulo and division operations, the solution efficiently handles the reversal

Extract the least significant digit of the current number digit = x % 10# Adjustments for negative numbers when the extracted digit is non-zero if x < 0 and digit > 0: digit -= 10 # Shift reversed number digits to the left and add the new digit

```
public int reverse(int x) {
    // Initialize answer to hold the reversed number
   int reversedNumber = 0;
   // Loop until x becomes 0
```

class Solution {

Java

C++

```
class Solution {
public:
    int reverse(int x) {
        int reversedNumber = 0;
        // Loop until all digits are processed
        while (x != 0) {
            // Check if multiplying by 10 will cause overflow
            if (reversedNumber < INT MIN / 10 || reversedNumber > INT MAX / 10) {
                return 0; // Return 0 if overflow would occur
            // Pop the last digit from 'x' using modulus and add it to 'reversedNumber'
            reversedNumber = reversedNumber * 10 + x % 10;
            // Remove the last digit from 'x' by dividing it by 10
            x /= 10;
        return reversedNumber; // Return the reversed number
};
TypeScript
/**
 * Reverse an integer.
 * @param \{number\} \times - The integer to be reversed.
 * @return {number} - The reversed integer, or 0 if the reversed integer overflows 32-bit signed integer range.
const reverseInteger = (x: number): number => {
    // Define the minimum and maximum values for 32-bit signed integer.
    const INT MIN: number = -(2 ** 31);
```

```
// Floor division by 10 to get the next digit (in TypeScript \sim \sim is replaced by Math.trunc).
        // Since x can be negative, we use trunc instead of floor to correctly handle negative numbers.
        x = Math.trunc(x / 10);
    return reversed;
};
class Solution:
   def reverse(self, x: int) -> int:
       # This variable will hold the reversed number
        reversed_number = 0
       # These define the range of acceptable 32-bit signed integer values
       min_int, max_int = -2**31, 2**31 - 1
       while x:
           # Check if the reversed number will overflow when multiplied by 10
            if reversed number < min int // 10 + 1 or reversed number > max_int // 10:
               # Return 0 on overflow as per problem constraints
               return 0
           # Extract the least significant digit of the current number
            digit = x % 10
           # Adiustments for negative numbers when the extracted digit is non-zero
           if x < 0 and digit > 0:
               digit -= 10
           # Shift reversed number digits to the left and add the new digit
            reversed_number = reversed_number * 10 + digit
           # Remove the least significant digit from x
           x = (x - digit) // 10
       # Return the reversed number within the 32-bit signed integer range
        return reversed_number
Time and Space Complexity
```

Time Complexity

The time complexity of the given code is dependent on the number of digits in the integer x. Since we are handling the integer digit by digit, the number of operations is linearly proportional to the number of digits. If the integer x has n digits, then the time complexity is O(n).

Space Complexity The space complexity of the provided code is 0(1). This is because we are only using a fixed amount of additional space (ans,

mi, mx, y, and a few variables for control flow) regardless of the input size.