<https://leetcode.com/problems/reverse-integer/description/>

Given a signed 32-bit integer x, return x with its digits reversed. If reversing x causes the value to go outside the signed 32-bit integer range [-2^31, 2^31 - 1], then return 0.

Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

**Example 1:**

**Input:** x = 123

**Output:** 321

**Example 2:**

**Input:** x = -123

**Output:** -321

**Example 3:**

**Input:** x = 120

**Output:** 21

**Constraints:**

-2^31 <= x <= 2^31 - 1

**Attempt 1: 2024-06-29**

**Solution 1: Math (10 min)**

class Solution {

    public int reverse(int x) {

        int result = 0;

        while(x != 0) {

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

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

// No need to add '- 1' since 1 / 10 = 0, not impact on math calculation perspective

// if(result < Integer.MIN\_VALUE / 10 || result > (Integer.MAX\_VALUE - 1) / 10) {

            if(result < Integer.MIN\_VALUE / 10 || result > Integer.MAX\_VALUE / 10) {

                return 0;

            }

            result = result \* 10 + x % 10;

            x /= 10;

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/reverse-integer/solutions/4060/my-accepted-15-lines-of-code-for-java/comments/168051>

Nice solution, but, as many others, I didn't get the line if ((newResult - tail) / 10 != result).

We're interested on what happens when an integer *overflows*. Well, it is rolled over. Practically speaking, if you would try

public static void main(String[] args) {

int rollMeOver= Integer.MAX\_VALUE + 1;

System.out.println(rollMeOver);

}

You will get as an output -2147483648 which represents the lowest value for an integer (Integer.MIN\_VALUE).

**Thus, in our problem, if newResult is going to overflow we are sure that newResult / 10 != result (this is the reason that**[**@Inception\_wzd**](https://leetcode.com/inception_wzd)**said that we don't need to subtract the tail first because by /10 we are already losing the last digit).**

By the way, the same thing happens for the *underflow*.

public static void main(String[] args) {

int rollMeOver= Integer.MIN\_VALUE - 1;

System.out.println(rollMeOver);

}

This is going to output the Integer.MAX\_VALUE which is 2147483647.

**Refer to**

<https://algo.monster/liteproblems/7>

**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**

To solve this problem, we first set up two boundaries, mi and mx, which represent the minimum and maximum values of a 32-bit 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 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 O(\log |x|) because the process continues for as many digits as x has, and the space complexity is O(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 reversed number without using additional data structures or complex patterns. Let's detail the steps using the provided Reference Solution Approach:

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

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.

Updating the Original Number x: We update x by removing its last digit. This is done by subtracting y from x and then dividing by 10.

Completion: The loop repeats this process, accumulating the reversed number in ans until all digits are processed.

The core of this approach is predicated on the mathematical guarantees regarding integer division and modulus operations in Python. The guard checks for overflow by considering both scale (multiplication by 10) and addition (adding the digit) separately 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 function achieves the reversal of digits robustly and efficiently.

**Solution Implementation**

class Solution {

public int reverse(int x) {

// Initialize answer to hold the reversed number

int reversedNumber = 0;

// Loop until x becomes 0

while (x != 0) {

// 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) {

return 0;

}

// Add the last digit of x to reversedNumber

reversedNumber = reversedNumber \* 10 + x % 10;

// Remove the last digit from x

x /= 10;

}

// Return the reversed number

return reversedNumber;

}

}

**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 O(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.