Reverse Integer LeetCode

Given an integer, write a program to reverse the digits of the integer.

Example Testcase:

Input: 123

Output: 321

Explanation:

The input integer is 123. To reverse its digits, we need to extract each digit of the integer starting from the last digit, and append it to a new integer variable.

First, we extract the last digit 3, and append it to the new integer variable. So the new integer variable becomes 3.

Then, we extract the second last digit 2, and append it to the new integer variable. So the new integer variable becomes 32.

Finally, we extract the first digit 1, and append it to the new integer variable. So the new integer variable becomes 321, which is the reversed integer.

Thus, the program should return the output as 321 for the given input integer 123.

Approach 1: Using Arithmetic Operations

In this approach, we can use arithmetic operations to extract each digit of the integer starting from the last digit, and append it to a new integer variable.

Initialize a new integer variable to 0.

While the input integer is not equal to 0:

Get the last digit of the input integer using the modulo (%) operator.

Append the last digit to the new integer variable by multiplying the new integer variable by 10 and adding the last digit.

Remove the last digit from the input integer by dividing it by 10 and taking the floor of the result.

Return the new integer variable as the reversed integer.

Approach 2: Using Recursion

In this approach, we can use recursion to extract each digit of the integer starting from the last digit, and append it to a new integer variable.

If the input integer is less than 10, return the input integer as the reversed integer.

Otherwise, get the last digit of the input integer using the modulo (%) operator.

Divide the input integer by 10 and pass the result recursively to the function to get the reversed integer of the remaining digits.

Multiply the reversed integer of the remaining digits by 10 and add the last digit to get the reversed integer of the original integer.

Return the reversed integer as the output.

Approach 3: Using String Manipulation

In this approach, we can convert the integer to a string, reverse the string, and then convert it back to an integer.

Convert the input integer to a string.

Reverse the string.

Convert the reversed string back to an integer.

Return the integer as the reversed integer.

Approach 1:

The reverseDigit function takes an integer num as input and returns the reversed integer. Here's how it works:

Determine the sign of the input integer:

int sign = (num < 0) ? -1 : 1;

This line of code uses the ternary operator (? :) to assign -1 to sign if num is negative, and 1 otherwise.

Take the absolute value of the input integer:

num = abs(num);

This line of code ensures that we are only dealing with positive integers in the rest of the function.

Reverse the digits of the input integer:

```
while(num != 0) {
   int reminder = num % 10;
   if((tempNum > (INT_MAX)/10) || (tempNum < (INT_MIN)/10))
```

```
        return 0;
    tempNum = tempNum * 10 + reminder;
    num /= 10;
}
```

This while loop takes the rightmost digit of num (the remainder when num is divided by 10), appends it to the right of tempNum, and then removes the rightmost digit of num by dividing it by 10. This process is repeated until all of the digits of num have been reversed.

Note that the if statement inside the while loop checks whether tempNum has overflowed the integer range. If it has, the function returns 0 to indicate that an overflow has occurred.

Multiply the result by the sign of the input integer:

return sign * tempNum;

This line of code ensures that the result has the same sign as the input integer.

For example, if the input integer is -123, the reverseDigit function will perform the following steps:

Determine the sign of the input integer:

sign = -1;

Take the absolute value of the input integer:

num = 123;

Reverse the digits of the input integer:

iteration 1:

reminder = 3;

tempNum = 3;

num = 12;

iteration 2:

reminder = 2;

tempNum = 32;

num = 1;

iteration 3:

reminder = 1;

tempNum = 321;

num = 0;

Multiply the result by the sign of the input integer:

return -1 * 321 = -321;

So the reversed integer of -123 is -321.

Time And Space Complexity:

The space complexity of the code is O(1) because it only uses a constant amount of extra space to store a few integer variables.

The time complexity of the code is O(log n), where n is the value of the input number. This is because the number of times the loop executes is proportional to the number of digits in the input number, which is log base 10 of n. In each iteration of the loop, the code performs a constant amount of arithmetic and comparison operations, so the overall time complexity is O(log n).

Approach 2:

This code reverses the digits of a given integer recursively, without using a loop or a secondary helper function. Here's how it works:

The reverseDigitRecursively function takes an integer num as input and initializes a variable sign to store the sign of the input number (-1 if the number is negative, 1 otherwise).

The function then takes the absolute value of the input number using the abs function, so that we can reverse the digits regardless of the sign of the input number.

The function calls the reverseDigitHelper function, passing in the absolute value of the input number as the first parameter and 0 as the second parameter.

The reverseDigitHelper function takes two parameters: num and tempNum. num represents the remaining digits of the input number that we need to reverse, and tempNum represents the reversed digits of the input number that we've computed so far.

If num is equal to 0, we've reversed all the digits and can return tempNum.

Otherwise, we take the last digit of num by computing num % 10 and store it in a variable called remainder.

We check if multiplying tempNum by 10 and adding remainder will cause an integer overflow. If so, we return 0.

We update tempNum by multiplying it by 10 and adding remainder.

We call reverseDigitHelper recursively with the remaining digits of num (computed by dividing num by 10) and the updated value of tempNum.

The function returns the result of the recursive call.

reverseDigitRecursively returns the result of reverseDigitHelper multiplied by the sign of the input number.

For example, if we call reverseDigitRecursively(123), the following steps will occur:

sign is set to 1, since 123 is positive.

We take the absolute value of 123, which is 123.

We call reverseDigitHelper(123, 0).

remainder is set to 3 (the last digit of 123).

We update tempNum to 3.

We call reverseDigitHelper(12, 3) recursively.

remainder is set to 2.

We update tempNum to 32.

We call reverseDigitHelper(1, 32) recursively.

remainder is set to 1.

We update tempNum to 321.

We call reverseDigitHelper(0, 321) recursively.

Since num is now 0, we return tempNum, which is 321.

reverseDigitRecursively(123) returns 321 * 1, which is 321.

Therefore, the output of reverseDigitRecursively(123) is 321, which is the reverse of 123.

Space And Time Complexity:

The space complexity of the code is O(1) as the only variables used are num, tempNum, remainder, and sign, which all take constant space.

The time complexity of the code is O(log(n)), where n is the input number. This is because the code uses recursion to reverse the digits, and the recursion depth is proportional to the number of digits in the input number. Since the number of digits in a number is logarithmic in the number itself, the time complexity is logarithmic.

Approach 3:

The third approach for reversing a number using string manipulation involves converting the input integer to a string, then manipulating the characters of the string to reverse the number, and finally converting the result back to an integer. This method is more concise than the previous two approaches and does not require a loop.

Here is an illustration of how the code works:

Suppose we have an input integer num = 12345. We first determine the sign of the number by checking if it is negative or positive. In this case, the number is positive, so sign = 1.

We then convert the input integer to a string using the to_string() function, which gives us the string "12345". We can then use a for loop to iterate over the characters in the string up to its halfway point, which is length/2. In each iteration of the loop, we swap the character at the current index i with the character at the opposite end of the string, which is length-i-1.

After the loop is complete, we have the string "54321". We can then convert this string back to an integer using the stoi() function, which gives us the reversed integer 54321. We multiply this integer by the sign to get the final result of 12345.

To ensure that the result does not cause an integer overflow, we can use a try-catch block to catch the out_of_range exception that may be thrown by the stoi() function. If an exception is caught, we can return 0 to indicate an error in the function.

Overall, this approach is simple and effective, and the code is easy to read and understand.

Space And Time Complexity:

The space complexity of this solution is O(n), where n is the length of the input number's string representation created using to_string() function.

The time complexity of this solution is O(n), where n is the length of the input number's string representation. This is because the for loop runs for half the length of the string representation, which is approximately n/2, making the time complexity O(n/2) = O(n). The stoi() function also has a time complexity of O(n), where n is the length of the input string. Therefore, the overall time complexity is O(n).

Each approach has its own advantages and disadvantages, and the choice of which approach to use will depend on the specific scenario and requirements of the problem.

The iterative approach using arithmetic operations is the most efficient in terms of time and space complexity, and it can handle large numbers without any overflow issues. However, it may not be the most intuitive or readable approach, and may require some understanding of the math behind the algorithm.

The recursive approach using arithmetic operations is also efficient in terms of time and space complexity, and it can be easier to read and understand than the iterative approach. However, it may not be as performant as the iterative approach, and may be subject to stack overflow errors if the input number is very large.

The approach using string manipulation is straightforward and easy to read and understand, but it may be slower and less efficient than the arithmetic approaches, especially for very

large numbers. Additionally, it requires handling the possibility of integer overflow when converting the reversed string back to an integer.