AI Prompting and ChatGPT's Shortcomings

Phillip Feldman

April 2023

1 Introduction

With the recent rise in popularity of using AI to solve coding questions, I decided to test ChatGPT's ability. I picked a LeetCode problem and modified it in a way so that ChatGPT would fail to solve on a naive prompt. I then attempted to prompt ChatGPT towards a correct solution.

2 The Problem

I chose the following question from https://leetcode.com/problems/reverse-integer/ and changed the bold "64" to 33.
The problem is ranked medium difficulty on LeetCode. With my modification, it is certainly harder. The problem is as follows:
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 33-bit integers (signed or unsigned).
I chose the problem for the following reasons:

- The restriction to 64-bit integers is mostly irrelevant to the solutions of the original problem
- Changing 64 to 33 in the bold instruction makes the problem more complex in a significant way.
- Most posted solutions to the original problem will not pass the modified version.
- There are a few posted solutions that are robust enough to pass the modified problem.
- Changing programming languages can lead to more complex solutions

The prevailing flaws I saw with the incorrect solutions had to do with returning 0 appropriately or the restriction to 32-bit integers. If we let x be the reverse of $2^{31} - 1$, then $33 > log_2(x) > 32$. So, x is a signed 34-bit integer or unsigned 33-bit integer. So, the program should return 0 on certain large inputs. Almost all of the solutions would fail on that instruction.

3 The Prompt

Me: Can you solve this python problem for me?
ChatGPT: Sure, I'd be happy to help you with your Python problem! Please provide me with the problem statement and any relevant details or specifications you have.
Me: 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 ˆ 31, 2 ˆ 31 - 1], then return 0.

Assume the environment does not allow you to store 33-bit integers (signed or unsigned).
ChatGPT: Certainly! Here's one way to implement the solution to this problem in Python:

```
class Solution:
    def reverse(self, x: int) -> int:
        # Check if the number is negative
        negative = x < 0

        # Convert the number to a string and reverse it
        x_str = str(abs(x))
        x_str_rev = x_str[::-1]

        # Convert the reversed string back to an integer
        x_rev = int(x_str_rev)

        # Apply the sign to the reversed integer
        if negative:
            x_rev = -x_rev

        # Check if the reversed integer is within the 32-bit range
        if x_rev < -2**31 or x_rev > 2**31 - 1:
            return 0

        return x_rev
```

This solution defines a Solution class with a reverse method that takes an integer x as input and returns the integer with its digits reversed. It first checks if x is negative and stores that information in the negative variable.

It then converts the absolute value of x to a string using the str function and reverses the string using Python's string slicing notation, which uses a step value of -1 to step through the string in reverse order. The resulting reversed string is stored in the x_str_rev variable.

The reversed string is then converted back to an integer using the int function, and the sign is applied to the integer using the negative variable. If the resulting integer is outside the range of a 32-bit signed integer, the function returns 0. Otherwise, it returns the reversed integer.

Note that this solution assumes that the input integer x is a signed 32-bit integer, as specified in the problem statement. If you need to handle integers of a different size or sign, you may need to modify the implementation accordingly.

Me: Your solution is incorrect.

ChatGPT: I apologize if my solution was incorrect. Could you please provide me with more information about what went wrong with my solution so that I can help you better? Perhaps you can share the error message you received or describe the issue you encountered. Additionally, if you have any specific requirements or constraints for your solution, please let me know.

Me: When you cast 'x_str_rev' to integer you could be storing a 33-bit integer in the variable 'x_rev'. **ChatGPT:**You are correct that casting the reversed string x_str_rev to an integer can result in an integer that requires more than 32 bits to represent. To handle this scenario, we can use a try-except block to catch any OverflowError that occurs when casting the reversed string to an integer. If an OverflowError occurs, we know that the reversed integer is outside the 32-bit range, so we can return 0.

Here's an updated implementation of the reverse method that handles this scenario:

```
class Solution:
    def reverse(self, x: int) -> int:
        # Check if the number is negative
        negative = x < 0

        # Convert the number to a string and reverse it
        x_str = str(abs(x))
        x_str_rev = x_str[::-1]

        # Try to convert the reversed string back to an integer
        try:
            x_rev = int(x_str_rev)
        except OverflowError:
            # Return 0 if the reversed integer is outside the 32-bit range
            return 0

        # Apply the sign to the reversed integer
        if negative:
            x_rev = -x_rev

        # Check if the reversed integer is within the 32-bit range
```

```
        if x_rev < -2**31 or x_rev > 2**31 - 1:
            return 0

        return x_rev
```

This implementation uses a try-except block to catch any OverflowError that occurs when casting the reversed string x_str_rev to an integer. If an OverflowError occurs, the function returns 0.

With this modification, the function should now handle cases where reversing the integer results in a value that is outside the 32-bit signed integer range.
Me: Python will not raise an 'OverflowError' on 33-bit integers.
ChatGPT: You are correct that Python will not raise an OverflowError when casting a 33-bit integer to an int, as the int type in Python can represent arbitrarily large integers. I apologize for any confusion I may have caused.

To correctly handle the constraint of not being able to store 33-bit integers, we can use the following implementation:

```
class Solution:
    def reverse(self, x: int) -> int:
        # Check if the number is negative
        negative = x < 0

        # Convert the number to a string and reverse it
        x_str = str(abs(x))
        x_str_rev = x_str[::-1]

        # Convert the reversed string back to an integer digit by digit
        x_rev = 0
        for digit in x_str_rev:
            x_rev = x_rev * 10 + int(digit)

        # Apply the sign to the reversed integer
        if negative:
            x_rev = -x_rev

        # Check if the reversed integer is within the 32-bit range
        if x_rev < -2**31 or x_rev > 2**31 - 1:
            return 0

        return x_rev
```

Me: Now, the for loop might store a 33-bit integer.
ChatGPT: You are correct that the for loop in my previous implementation may produce an integer with more than 32 bits. Specifically, if the reversed integer is greater than or equal to 2,147,483,647, then the value of x_rev * 10 will be greater than 2,147,483,647, which will cause x_rev * 10 + int(digit) to overflow and produce an integer with more than 32 bits.

To handle this scenario, we can modify the loop to perform the multiplication and addition in a separate step, and check for overflow before updating the value of x_rev. One way to do this is to use the divmod function to compute the quotient and remainder of the multiplication operation x_rev * 10, and then add the remainder and the current digit to obtain the new value of x_rev. If the quotient is nonzero, then we know that the multiplication operation caused an overflow, and we can return 0.

Here's an updated implementation of the reverse method that handles this scenario:

```
class Solution:
    def reverse(self, x: int) -> int:
        # Check if the number is negative
        negative = x < 0

        # Convert the number to a string and reverse it
        x_str = str(abs(x))
        x_str_rev = x_str[::-1]

        # Convert the reversed string back to an integer digit by digit
        x_rev = 0
        for digit in x_str_rev:
            quotient, remainder = divmod(x_rev * 10, 2**31)
            if quotient != 0:
                return 0
            x_rev = quotient + remainder + int(digit)

        # Apply the sign to the reversed integer
        if negative:
            x_rev = -x_rev

        return x_rev
```

This implementation uses the divmod function to compute the quotient and remainder of the multiplication operation x_rev * 10, and checks for overflow by testing whether the quotient is nonzero. If the quotient is nonzero, the function returns 0 to indicate that the result is outside the 32-bit signed integer range.

With this modification, the function should now correctly handle cases where the reversed integer has more than 32 bits.

Me:'x_rev * 10' could possibly overflow.

ChatGPT: You are correct that the expression x_rev * 10 could potentially overflow and produce a result outside of the 32-bit signed integer range.

To handle this scenario, we can use the fact that the maximum value of a 32-bit signed integer is 2**31 - 1, which means that the maximum value of x_rev * 10 without overflow is 2**31 // 10. If x_rev * 10 exceeds this value, then we know that the result will be outside of the 32-bit signed integer range and we can return 0.

Here's an updated implementation of the reverse method that checks for overflow in this way:

```
class Solution:
    def reverse(self, x: int) -> int:
        # Check if the number is negative
        negative = x < 0

        # Convert the number to a string and reverse it
        x_str = str(abs(x))
        x_str_rev = x_str[::-1]

        # Convert the reversed string back to an integer digit by digit
        x_rev = 0
        for digit in x_str_rev:
            if x_rev > 2**31 // 10:
                return 0
            x_rev = x_rev * 10 + int(digit)

        # Apply the sign to the reversed integer
        if negative:
            x_rev = -x_rev

        # Check if the reversed integer is within the 32-bit range
        if x_rev < -2**31 or x_rev > 2**31 - 1:
            return 0

        return x_rev
```

This implementation checks whether x_rev * 10 exceeds 2**31 // 10 before updating the value of x_rev. If the check fails, the function returns 0 to indicate that the result is outside the 32-bit signed integer range.

With this modification, the function should correctly handle cases where x_rev * 10 could potentially overflow.

Me: You keep making similar errors. Try solving the problem in Java first, then perform a naive translation to python.

ChatGPT:I apologize for the errors in my previous responses. I will take your suggestion and provide a solution in Java first, then translate it to Python.

Here's an implementation in Java:

```
class Solution {
    public int reverse(int x) {
        int result = 0;
        while (x != 0) {
            int digit = x % 10;
            int newResult = result * 10 + digit;
            if ((newResult - digit) / 10 != result) {
```

```
                return 0;
            }
            result = newResult;
            x /= 10;
        }
        return result;
    }
}
```

This implementation works by extracting the rightmost digit of the input integer x using the modulo operator %, and then adding it to the reversed integer result using the formula result * 10 + digit. We also check for overflow by verifying that (newResult - digit) / 10 == result. If this condition fails, it means that the addition overflowed and produced a result outside the 32-bit signed integer range, so we return 0. If the loop completes successfully, we return the reversed integer result.

Here's a translation of the Java solution to Python:

```
class Solution:
    def reverse(self, x: int) -> int:
        result = 0
        while x != 0:
            digit = x % 10
            new_result = result * 10 + digit
            if (new_result - digit) // 10 != result:
                return 0
            result = new_result
            x //= 10
        return result
```

This implementation follows the same approach as the Java solution, using the modulo operator % to extract the rightmost digit of x and the floor division operator // to shift the digits of x to the right. We also check for overflow using the same formula (new_result - digit) // 10 == result.

I hope this solution meets your requirements. Please let me know if you have any further questions or concerns.

4 Analysis and Conclusion

I tested ChatGPT's solutions by creating a function that forces all constants and variables suspected of overflowing to behave like they would in Java (wrap around for overflow). ChatGPT failed solutions 1,2,3 and passed on solutions 4,5,6. Of course, ChatGPT was not confident in its correctness. I expected ChatGPT to fail on its first attempt because the solutions posted to the similar problem would not work. I did not expect it to continuously fail after repeated hints. ChatGPT seemed able to correct its mistakes when pointed out. However,

it would not actually work around the issue. This caused ChatGPT to simply move the mistake to different parts of the code.
The Python environment allows for storage of 33-bit and higher integers. Java does not. This would invalidate any incorrect solutions provided by ChatGPT for the same reason the bold instruction does. The training data set would likely have plenty of code that deals with this limitation in Java (using longs vs ints), whereas it likely does not have a lot of data relevant to the specific instruction about 33-bit integers. However, there are posted solutions to the similar problem that are robust enough to solve the modified problem. This shows that ChatGPT cannot logically combine knowledge from different sources to solve novel problems.