Topic Introduction

Welcome to another PrepLetter, where we break down tricky interview concepts into bite-sized, practical lessons! Today, we're diving into **Integer Manipulation**.

What is Integer Manipulation?

Integer manipulation is about working directly with the digits and structure of numbers, without converting them into strings (unless absolutely necessary). This means extracting, reversing, or altering digits using arithmetic operations like division and modulo.

How does it work?

Imagine you need to reverse a number, check if it's a palindrome, or add one to a list of digits. You'll use operations like:

- **Modulo (%)** to grab the last digit.
- **Integer division (//)** to remove the last digit.
- **Multiplication and addition** to build new numbers.

Why is it useful in interviews?

These problems test your ability to:

- Work precisely with number logic.
- Avoid overflows and edge cases (like negative numbers or leading zeros).
- Write code that's efficient, readable, and not reliant on string tricks.

Simple Example (not from today's problems):

Extracting all digits of a number:

```
n = 1234
while n > 0:
    digit = n % 10
    print(digit)
    n //= 10
```

This prints 4, 3, 2, 1 — the digits from least to most significant.

Today's Problems:

- Reverse Integer
- Palindrome Number
- Plus One

Why are these grouped together?

All three require you to manipulate integers at the digit level. "Reverse Integer" flips the digits and handles overflow. "Palindrome Number" checks if an integer reads the same forwards and backwards. "Plus One" operates on a list of digits, simulating addition.

The best order for learning is:

- Reverse Integer (fundamental digit handling and overflow)
- Palindrome Number (similar digit extraction, different goal)
- Plus One (manipulating digit arrays, with carry logic)

Problem 1: Reverse Integer

Problem Statement (LeetCode Link):

Given a 32-bit signed integer, reverse its digits. If reversing causes the value to go outside the signed 32-bit integer range, return 0.

Example:

- Input: `x = 123`
- Output: `321`

Walkthrough:

- Start with 123.
- Reverse the digits: 321.

Extra Test Case:

- Input: `x = -2040`
- Output: `-402`

(Note: The output does NOT have leading zeros.)

Solving Strategy:

- Extract digits one by one using modulo and division.
- Build the reversed number.
- Check for overflow before appending each digit (since 32-bit signed integers go from -2^31 to 2^31 - 1).

Try it on Paper:

Take `x = 1200`. What should your code output?

Brute-force Approach:

- Convert the number to a string, reverse it, and convert back to int.
- *Drawbacks*: Not allowed in interviews if the problem says “Don’t convert to string.” Also, doesn’t teach you digit manipulation.

Time complexity: O(k), where k is number of digits.

Optimal Approach:

- Use math to reverse the digits:
  - Initialize `rev = 0`.
  - While `x` is not 0:
    - Pop the last digit: `pop = x % 10` (for negatives, use `pop = x % 10` carefully or adjust the logic).
    - Append it to `rev`: `rev = rev * 10 + pop`.
    - Check if `rev` will go out of 32-bit integer range before multiplying or adding.
    - Update `x` by removing the last digit: `x //= 10` (for negatives, Python division rounds toward minus infinity).

Python Solution:

```
def reverse(x):
    # 32-bit signed integer range
    INT_MIN, INT_MAX = -2**31, 2**31 - 1
    rev = 0
```

```
    neg = x < 0
    x = abs(x)

    while x != 0:
        pop = x % 10
        x //= 10

        # Check for overflow before actually multiplying
        if rev > (INT_MAX - pop) // 10:
            return 0
        rev = rev * 10 + pop

    if neg:
        rev = -rev
    if rev < INT_MIN or rev > INT_MAX:
        return 0
    return rev
```

Time Complexity: O(k)

Space Complexity: O(1)

How the solution works:

- We store if the number is negative.
- Loop: pop off the last digit and push it to the reversed number.
- Before every push, check if multiplying by 10 would go out of bounds.
- Return 0 if overflow would occur.

Trace Example:

`x = -120`

- neg = True, x = 120
- First iteration: pop = 0, rev = 0
- Second iteration: pop = 2, rev = 2
- Third iteration: pop = 1, rev = 21
- x is now 0
- Apply sign: rev = -21

Try this test case on your own:

`x = 1534236469` (Does this overflow? What should your function return?)

Take a moment to solve this on your own before jumping into the solution!

Problem 2: Palindrome Number

Problem Statement (LeetCode Link):

Check if an integer is a palindrome (reads the same backward as forward). Negative numbers are NOT palindromes.

Example:

- Input: `x = 121`
- Output: `True`

How is this similar to the previous problem?

- Both require extracting digits from the number.
- Both use modulo and division to work with digits.
- Instead of reversing for output, here we compare digits from both ends.

Brute-force Approach:

- Convert to string, reverse it, compare with original.
- *Drawbacks*: String conversion is discouraged in interviews for digit logic problems.

Time complexity: O(k)

Optimal Approach:

- Reverse only half of the number and compare with the other half.
- If the number is negative or ends with 0 (and is not 0), it's not a palindrome.
- While the reversed half is less than the original half, keep extracting digits.

Step-by-step:

- If x < 0 or (x % 10 == 0 and x != 0): return False
- Initialize rev = 0
- While x > rev:
    - pop = x % 10
    - rev = rev * 10 + pop
    - x //= 10
- After the loop, x should equal rev (even length) or x == rev // 10 (odd length)

Pseudocode:

```
if x < 0 or (x % 10 == 0 and x != 0):
    return False
rev = 0
while x > rev:
    pop = x % 10
    rev = rev * 10 + pop
    x //= 10
return x == rev or x == rev // 10
```

Example Trace:

`x = 1221`

- rev = 0, x = 1221
- pop = 1, rev = 1, x = 122
- pop = 2, rev = 12, x = 12 (now x == rev)
- x == rev, so return True

Try This Test Case:

`x = 12321` (Is this a palindrome?)

Time Complexity: O(k)

Space Complexity: O(1)

Problem 3: Plus One

Problem Statement (LeetCode Link):

Given a non-empty array of digits representing a non-negative integer, add one to the number.

How is this related?

- You manipulate digits without converting the whole array to an integer (to avoid overflow for huge numbers).
- The challenge is handling the carry, especially when the number has trailing 9s.

What's different or more challenging?

- Instead of working with a number, you work with a list.
- Carry can ripple through several digits, and sometimes you need to add a new digit at the front.

Brute-force Approach:

- Convert array to integer, add one, then split back to digits.
- *Drawbacks*: Not safe for very large numbers, and not usually allowed.

Optimal Approach:

- Start from the last digit, add one.
- If result is less than 10, done.
- If result is 10, set to 0 and carry over.
- If you finish the loop and still have a carry, insert 1 at the beginning.

Pseudocode:

```
for i from len(digits) - 1 downto 0:
    if digits[i] < 9:
        digits[i] += 1
        return digits
    digits[i] = 0
insert 1 at the beginning of digits
return digits
```

Example:

Input: `[1,2,9]`

- 9 + 1 = 10, set to 0, carry = 1
- 2 + carry = 3, set to 3, done

Output: `[1,3,0]`

Try This Test Case:

Input: `[9,9,9]`

What should the output be?

Time Complexity: O(n)

Space Complexity: O(1) (if in-place), O(n) if you count the output array when a new digit is added

Summary and Next Steps

Recap:

Today, we tackled three classic interview problems that all build your skills with integer manipulation:

- Extracting and rebuilding numbers using modulo and division.
- Handling tricky cases like overflows, negatives, and carries.
- Understanding how to manipulate both numbers and digit arrays efficiently.

Key Patterns and Insights:

- Use `% 10` and `// 10` to work with digits directly.
- Always check for integer overflow when reconstructing numbers.
- When working with digit arrays, handle carries carefully and know when to grow the array.

Common mistakes:

- Forgetting to consider negative inputs or leading zeros.
- Not checking for overflow.
- Mishandling edge cases (like numbers ending in zero, or all-nines input).

Action List

- Solve all three problems on your own, even if a solution was provided here.
- Try Problem 2 and 3 using a different approach (e.g., for palindrome, try full string conversion and compare; for plus one, try using recursion).
- Explore related problems like "String to Integer (atoi)" or "Add Two Numbers" (linked lists).
- Compare your code with solutions from others, paying attention to edge cases and style.
- Don’t worry if you get stuck. Practicing these patterns is the best way to improve.

Keep it up! Every digit you manipulate is a step closer to interview mastery.