Topic Introduction

Have you ever wondered how calculators handle numbers way beyond their normal limits? Or why programming languages sometimes make you jump through hoops just to add two really big numbers together? Welcome to the world of **arithmetic on number representations** — a classic topic that shows up often in coding interviews.

The Concept: Arithmetic on Number Representations

In many programming languages, numbers are limited by their data types. But what if you need to work with numbers that are hundreds or thousands of digits long? That's where representing numbers as arrays or strings comes in. Instead of relying on built-in arithmetic, you process numbers digit by digit, just like how you would on paper.

How does it work?

• Each digit of the number is stored in a string or an array, most often with the most significant digit at the front (just like how we write numbers).

• Arithmetic operations (addition, multiplication, etc.) are then performed by simulating the process you would use if you were doing the calculation by hand, from right to left.

When and Why is this useful in interviews?

- These problems test your ability to manipulate data structures directly.
- They assess your understanding of loops, indexing, carry-overs, and sometimes string manipulation.
- They're a great way for interviewers to gauge how you handle edge cases and "off-by-one" errors.
- Mastery of these simulations shows you can work beyond built-in library limitations.

Simple Example (not one of our problems today):

Suppose you need to add two numbers, `123` and `789`, represented as arrays `[1,2,3]` and `[7,8,9]`. You'd add from the rightmost digits, carry over as needed, and construct the result as `[9,1,2]`, which is `912`.

The Problems

Today’s trio:

- **Plus One** (LeetCode 66)
- **Add Binary** (LeetCode 67)
- **Multiply Strings** (LeetCode 43)

What's the link?

All three require you to perform arithmetic operations on numbers that are not stored as plain integers. Whether adding one to a number, summing two binary numbers as strings, or multiplying two huge numbers, you’ll need to simulate the process digit by digit, often from right to left, handling carries along the way.

Let’s break them down, starting simple and ramping up!

Problem 1: Plus One

Problem:

Given a non-empty array of decimal digits representing a non-negative integer, add one to the integer. The digits are stored such that the most significant digit is at the head of the list.

Problem link

Example

Input: `[1, 2, 3]`
Output: `[1, 2, 4]`
Because 123 + 1 = 124

The Thinking Process

- Imagine you have the number written on paper: `1 2 3`.
- You want to add 1 — so start from the rightmost digit.
- If adding 1 causes a carry (like going from 9 to 0), continue left, carrying over.
- If you finish and there's a carry left, you need to add a new digit at the front.

Try this one yourself:
Input: `[9, 9, 9]`
What should the output be?

Brute Force Approach

You could convert the array to a number, add 1, then split it back into digits.
But for very large numbers, this won't work due to integer overflow.

- **Time Complexity:** O(n) (for conversion and splitting)
- **Space Complexity:** O(n) (for the output array)
- **But:** Not robust for huge numbers.

Optimal Approach: In-Place Digit Manipulation

The core pattern:

- Traverse the digits from right to left.
- Add 1 to the last digit.
- If the digit becomes 10, set it to 0, carry 1 to the next digit.
- If not, you're done.
- If you've carried all the way past the first digit, insert a new `1` at the front.

Let's code it in Python.

```
def plusOne(digits):
    """
```

```
    Adds one to a number represented as an array of digits.

    Args:
    digits (List[int]): Array of digits (most significant at index 0).

    Returns:
    List[int]: The incremented number as an array of digits.
    """
    n = len(digits)
    for i in range(n - 1, -1, -1):  # Start from the last digit
        if digits[i] < 9:
            digits[i] += 1  # No carry, so we can return early
            return digits
        digits[i] = 0  # Set to 0 and carry to next digit
    # If we're here, all digits were 9
    return [1] + digits  # Add 1 at the front

# Time Complexity: O(n)
# Space Complexity: O(1) if modifying in-place, O(n) for the result in worst case (when
new digit is added).
```

What's happening in the code?

- We loop from the end of the digits array.
- If a digit is less than 9, we just add 1 and return.
- If it's 9, we set it to 0 (because 9 + 1 = 10, carry over).
- If all digits are 9, we need to add a new `1` at the front.

Trace with Input `[9, 9, 9]`:

- i = 2: digits[2] = 9 — set to 0, carry
- i = 1: digits[1] = 9 — set to 0, carry
- i = 0: digits[0] = 9 — set to 0, carry
- All done: prepend 1, result is `[1, 0, 0, 0]`

Try this test case:

Input: `[4, 3, 2, 1]`

What do you think the output will be?

Pause here and try to solve it yourself before peeking at the code!

Problem 2: Add Binary

Problem:

Given two binary strings `a` and `b`, return their sum as a binary string.

Problem link

Example

Input:

```
a = "1010"
b = "1011"
```

Output:

```
"10101"
```

How is this similar or different?

- Similar: Need to process numbers digit-by-digit, from right to left, handling carry.
- Different: Now, both numbers are strings, and the base is 2 (binary), not 10.

Brute Force

You could convert both strings to integers, add, and convert back to binary string.
But again, this fails for very large inputs.

Optimal Approach: Manual Binary Addition

- Use two pointers, one for each string, starting from the right end.
- Track a `carry` variable.
- At each step, add the corresponding digits (0 or 1) and the carry.
- Append the sum modulo 2 to the result.
- Update the carry (sum // 2).
- Continue until both strings and carry are processed.
- Reverse the result at the end.

Step-by-step for the example:

```
a:  1 0 1 0
b:  1 0 1 1
        ^
sum: 0 + 1 + 0 (carry) = 1 -> append '1', carry 0
     ^
sum: 1 + 1 + 0 = 2 -> append '0', carry 1
   ^
sum: 0 + 0 + 1 = 1 -> append '1', carry 0
 ^
sum: 1 + 1 + 0 = 2 -> append '0', carry 1
Out of digits, carry is 1 -> append '1'
Reverse: '1', '0', '1', '0', '1' => "10101"
```

Another test case to try:

`a = "11"`, `b = "1"`

Expected output: `"100"`

Pseudocode:

```
Initialize i = len(a) - 1, j = len(b) - 1, carry = 0, result = []
While i >= 0 or j >= 0 or carry:
    digitA = int(a[i]) if i >= 0 else 0
    digitB = int(b[j]) if j >= 0 else 0
    total = digitA + digitB + carry
    result.append(str(total % 2))
    carry = total // 2
    i -= 1
    j -= 1
Reverse result and join as string
Return result
```

Trace with `a = "11"`, `b = "1"`:

- i=1, j=0: 1+1+0=2, append '0', carry=1
- i=0, j=-1: 1+0+1=2, append '0', carry=1
- i=-1, j=-2: 0+0+1=1, append '1', carry=0
- Result: ['0','0','1'] -> Reverse: '100'

Time Complexity: O(max(len(a), len(b)))

Space Complexity: O(max(len(a), len(b))) (for the result string)

Problem 3: Multiply Strings

Problem:

Given two non-negative integers represented as strings, return the product as a string. (Do not use built-in BigInteger libraries.)

Problem link

What's different or more challenging?

- Now, you're multiplying (not just adding) very large numbers represented as strings.
- You need to simulate the grade-school multiplication algorithm.
- Each digit in one number must be multiplied by every digit in the other, and results properly summed.

Example

Input:

`num1 = "123"`

`num2 = "456"`

Output:
`"56088"`

Manual Multiplication Approach

- Initialize a result array of size `len(num1) + len(num2)` (maximum possible digits).
- For each digit in `num1` (from right to left):
    - For each digit in `num2` (from right to left):
        - Multiply digits, add to the current position in result (taking care of carry).
- At the end, skip leading zeros and combine digits to form the answer string.

Pseudocode:

```
Initialize result array of zeros with length len(num1) + len(num2)
For i from len(num1)-1 down to 0:
    For j from len(num2)-1 down to 0:
        mul = int(num1[i]) * int(num2[j])
        sum = mul + result[i+j+1]
        result[i+j+1] = sum % 10
        result[i+j] += sum // 10
Skip leading zeros in result array
Join result to form string
Return result string
```

Try this test case yourself:
`num1 = "99"`, `num2 = "99"`
Expected output: `"9801"`

Trace (short version):

- Multiply 9*9 (units place), add to result.
- Carry over as appropriate.
- Final result after joining: "9801".

Time Complexity: O(n * m), where n and m are the lengths of num1 and num2.
Space Complexity: O(n + m) (for the result array)

Summary and Next Steps

Today, you explored three classic problems that require simulating arithmetic without relying on built-in number types. The big picture:

- **Pattern:** Process digits from right to left, handle carries properly, build up the result.
- **Insight:** These techniques are essential when working with very large numbers or when numbers are given as strings/arrays.

- **Variations:** The base can change (decimal, binary, etc.), or the operation (addition, multiplication).

Common Traps:

- Forgetting to handle carry at the end.
- Off-by-one errors when indexing.
- Not reversing the result when needed.
- Dropping leading zeros (for multiplication).

Action List:

- Try solving all three problems on your own, even the one with code provided!
- For Add Binary and Multiply Strings, see if you can write the code from scratch.
- Dry-run your solutions with edge cases: lots of 9s, lots of 1s, or different string lengths.
- Try to adapt your code to work with other bases (e.g., base-7).
- Compare your solutions with others online—look for alternative techniques or optimizations.
- Don’t worry if you get stuck—practice is key, and every attempt builds your skill!

Keep practicing, and soon you’ll be comfortable with any “simulate arithmetic” challenge interviews throw your way. Happy coding!