Topic Introduction

Today's trio of problems puts the spotlight on **string processing and manipulation**. While strings are a familiar datatype, the art of processing them efficiently is a recurring theme in interviews. Whether you're searching for patterns, performing arithmetic, or finding overlaps, string processing often means going character-by-character, carefully managing indices, and handling edge cases (like empty strings or mismatched lengths).

What is string processing?
String processing is the set of techniques used to examine, modify, or analyze sequences of characters. It can involve searching for substrings, modifying content, or simulating operations (like addition or multiplication) that we usually perform on numbers.

How does it work?
At its core, string processing often involves looping through the string(s), comparing or manipulating characters, and building up results using string concatenation or auxiliary data structures.

When and why is it useful in interviews?
Interviewers love string problems because they test your attention to detail, understanding of data structures, and ability to handle corner cases. Many real-world applications (like text editors, search engines, and data validation) rely on string processing.

Example (not one of today's problems):
Suppose you need to check if a word is a palindrome (the same forwards and backwards). You might use two pointers, one at the start and one at the end, comparing characters and moving inward. If all corresponding pairs match, it's a palindrome!

Why These 3 Problems?

Today's problems are all about **string manipulation**:

- **Longest Common Prefix** finds the shared starting substring among a group of strings.
- **Add Binary** simulates addition for two binary numbers, each given as a string.
- **Multiply Strings** simulates multiplication for two large numbers represented as strings.

They're grouped because they all focus on processing strings character-by-character, handling different string lengths, and building up answers from smaller components. Each tests your ability to simulate operations you'd usually do with numbers — but using only string operations. Let's dive in!

Problem 1: Longest Common Prefix

Leetcode 14: Longest Common Prefix

Rephrased Problem Statement:
Given a list of strings, find the longest prefix that's shared by all of them. If there is no common prefix, return an empty string.

Example Input/Output:

- Input: `["flower", "flow", "flight"]`

- Output: `"fl"`

How to Think About It:

Imagine lining up the words vertically and comparing their letters column by column from the start. As soon as you find a mismatch in a column, you know that's where the common prefix ends.

Try This One:

Input: `["dog", "racecar", "car"]`

Expected Output: `""` (no common prefix)

Brute-Force Approach:

Check each prefix of the first string (one character at a time), and for each, see if all other strings start with that prefix. This is inefficient — you might be repeating many checks.

- **Time Complexity:** O(N * M^2), where N is the number of strings and M is the length of the first string.

Optimal Approach: Vertical Scanning Pattern

Instead of checking all prefixes, scan one character at a time in all strings, comparing the same index. If all strings have the same character, add it to the prefix. Stop at the first mismatch or when any string runs out of characters.

Step-by-Step Logic:

- If the list is empty, return `""`.
- For each character index in the first string:
  - Check that every string has that character at the same index.
  - If not, return the prefix up to (but not including) this index.
- If you finish scanning the whole first string, it's the common prefix.

Python Solution:

```
def longestCommonPrefix(strs):
    if not strs:
        return ""

    # Loop through each character index of the first string
    for i in range(len(strs[0])):
        char = strs[0][i]
        # Compare with the same position in all other strings
        for s in strs[1:]:
            # If index is out of range or characters don't match
            if i >= len(s) or s[i] != char:
                return strs[0][:i]
    # If we get through the whole first string, that's the prefix
    return strs[0]
```

- **Time Complexity:** O(N * M), where N is number of strings, M is length of the shortest string.
- **Space Complexity:** O(1) (ignoring the output string).

How the Code Works:

- The outer loop goes over each character index of the first string.
- The inner loop checks that every other string matches at the current index.
- If a mismatch is found or a string is too short, the prefix so far is returned.
- If no mismatches, the entire first string is the prefix.

Trace with Example:

Input: `["flower", "flow", "flight"]`

- i = 0: All have 'f' at index 0.
- i = 1: All have 'l' at index 1.
- i = 2: "flower" and "flow" have 'o', but "flight" has 'i' at index 2.
- Mismatch found at index 2. Return `strs[0][:2]` which is `"fl"`.

Try This Test Case Yourself:

Input: `["cir", "car"]`

What's the output?

Take a moment to solve this on your own before jumping into the solution. Remember: careful indexing is key!

Problem 2: Add Binary

Leetcode 67: Add Binary

Rephrased Problem Statement:

You are given two non-empty binary strings. Return their sum as a binary string.

Why is this similar or different?

Like the previous problem, you're processing strings character-by-character. But here, you're simulating digit-wise addition (like you would on paper), handling carry-over as you go. This is a great example of simulating number operations with strings.

Example Input/Output:

- Input: `a = "11"`, `b = "1"`
- Output: `"100"`

Try This One:

Input: `a = "1010"`, `b = "1011"`

Expected Output: `"10101"`

Brute-Force Approach:

Convert both strings to integers, add them, and convert the sum back to a binary string. But interviews often ask you to avoid integer conversion due to possible overflow and to demonstrate string manipulation skills!

- **Time Complexity:** O(N), but avoids the core challenge.

Optimal Approach: Two-Pointer Addition with Carry

- Start from the rightmost digit of each string.
- Add corresponding digits and carry.
- Append the result to a result list.

- Move left, and repeat until all digits and carry are processed.
- Reverse the result at the end.

Pseudocode:

```
Initialize i = len(a) - 1, j = len(b) - 1, carry = 0, result = empty list
While i >= 0 or j >= 0 or carry:
    sum = carry
    If i >= 0:
        sum += int(a[i])
        i -= 1
    If j >= 0:
        sum += int(b[j])
        j -= 1
    result.append(str(sum % 2))
    carry = sum // 2
Reverse result
Return result joined as string
```

Step-by-Step Example:

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

- i = 1, j = 0, carry = 0
- sum = 1 (from a) + 1 (from b) + 0 = 2
- result: append "0" (2 % 2), carry becomes 1
- i = 0, j = -1, carry = 1
- sum = 1 (from a) + 0 (b exhausted) + 1 = 2
- result: append "0" (2 % 2), carry stays 1
- i = -1, j = -1, carry = 1
- sum = 0 + 0 + 1 = 1
- result: append "1", carry = 0
- Reverse: "100"

Try This Test Case:

a = "111", b = "10"

What's the output?

- **Time Complexity:** O(max(len(a), len(b)))
- **Space Complexity:** O(max(len(a), len(b))) for the result string.

Problem 3: Multiply Strings

Leetcode 43: Multiply Strings

Rephrased Problem Statement:

Given two non-negative numbers as strings, return their product, also as a string. Do not use built-in big integer libraries or direct

integer conversion.

What's different or more challenging here?

Now you're simulating multiplication manually, not just addition. This means you need to handle digit-wise multiplication and carrying over, just like you would if you multiplied numbers on paper.

Example Input/Output:

- Input: `num1 = "123"`, `num2 = "456"`
- Output: `"56088"`

Try This One:

num1 = "25", num2 = "4"

Expected Output: "100"

Optimal Approach: Digit-by-Digit Multiplication Using Arrays

- Reverse both strings to ease addition from least-significant digit.
- For each digit in num1 (from right to left):
    - For each digit in num2:
        - Multiply and add to the correct position in the result array.
- Handle carry for each position in the result.
- Skip leading zeros and return the joined string.

Pseudocode:

```
If either num1 or num2 is "0": return "0"
Initialize result array of size len(num1) + len(num2) with zeros

For i from len(num1) - 1 to 0:
    For j from len(num2) - 1 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 the rest into a string and return
```

Step-by-Step Example:

num1 = "12", num2 = "10"

- result = [0, 0, 0, 0]
- Multiply 2 *0, 2* 1, 1 *0, 1* 1, updating result array.
- Final result: [0, 1, 2, 0] -> "120"

Try This Test Case:

num1 = "13", num2 = "9"

What's the output?

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

Subtle Nudge:
Can you think of other ways to multiply strings, or optimize the carry-handling step?

Summary and Next Steps

Today, we explored three classic string manipulation problems: finding common prefixes, simulating binary addition, and manual multiplication. Each highlights the importance of careful indexing, handling carries, and building results step by step. These problems help build intuition for more advanced string processing patterns, like parsing, conversion, or pattern matching.

Key Patterns:

- Character-by-character processing
- Two-pointer techniques (often from the end of the string)
- Careful carry/overflow management
- Building results in reverse and reversing at the end

Common Pitfalls:

- Off-by-one errors (especially with indices)
- Forgetting to handle carry after the main loop
- Not handling empty strings or leading zeros properly

Action List

- Solve all three problems yourself, even the one with code provided. Hand-simulate a few test cases.
- For Add Binary and Multiply Strings, try writing solutions that don't use any type conversion — pure string manipulation.
- Explore other string processing challenges, such as string to integer conversion (`atoi`), reversing words, or substring search.
- Reflect on common bugs you hit, and discuss your solutions with a friend or mentor.
- Remember: practice is how you master attention to detail and edge-case readiness!

Keep at it — string skills are essential for interviews and beyond!