Topic Introduction

Today we’re diving deep into **string parsing and conversion** problems. These are classic interview questions that look deceptively simple, but often require careful thinking and attention to detail.

What is string parsing?
String parsing means analyzing a string and breaking it down into parts based on some rules. This could involve extracting numbers, changing formats, or validating that a string matches a certain pattern.

When do you use it?
You’ll find string parsing everywhere: reading user input, processing file data, interpreting configuration, or even comparing software versions. Interviewers love these questions because they test your ability to work precisely with text, handle tricky edge cases, and write robust code.

A quick example:
Suppose you have `"apple,banana,pear"` and you want a list of fruits. You can use a split operation to get `["apple", "banana", "pear"]`. Simple enough! But what if there are spaces, empty items, or invalid characters? That’s when parsing gets interesting.

Why are these useful in interviews?
String parsing problems test your:

- Attention to detail (handling spaces, signs, invalid input)
- Ability to write code that’s robust and clear
- Understanding of string manipulation techniques

Let’s see how this plays out in three popular problems.

Why Are These Problems Grouped Together?

- **ZigZag Conversion**: Rearranges a string into a zigzag pattern, then reads it line by line.
- **String to Integer (atoi)**: Converts a string to an integer, following specific parsing rules and validation.
- **Compare Version Numbers**: Compares two version strings (like "1.0.3" and "1.1") by parsing and comparing each component.

All three require you to **parse strings and transform or compare their contents**. Each one has its own twist, but they share a foundation: careful, step-by-step string processing.

Let’s start with the most visual of the three!

Problem 1: ZigZag Conversion

LeetCode Problem Link

Problem Statement (in plain English):
Given a string and a number of rows, write the string in a zigzag pattern on the given number of rows. Then, read the characters row

by row, top to bottom, to form a new string.

Example:

Input:

`s = "PAYPALISHIRING"`, `numRows = 3`

Pattern:

```
P   A   H   N
A P L S I I G
Y   I   R
```

Output: `"PAHNAPLSIIGYIR"`

How to Think About It:

- Imagine writing letters diagonally down to the bottom row, then diagonally up to the top, and repeat.
- If `numRows` is 1, the output is just the original string.

Tip:

Try drawing the zigzag on paper for small examples!

Try this input yourself:

`s = "HELLOWORLD"`, `numRows = 4`

Brute Force Idea:

- Build a 2D grid with `numRows` rows and as many columns as needed.
- Fill the grid following the zigzag, then read row by row.
- **Time Complexity**: O(N), but extra space for the grid.

Optimal Approach:

- Use an array of strings, one for each row.
- Track the current row and direction (down or up).
- For each character:
    - Add it to the current row.
    - Change direction at the top or bottom row.

Step-by-Step Logic:

- If `numRows` is 1 or greater than the string length, return the input.
- Prepare a list to hold strings for each row.
- Use variables: `current_row`, `going_down` (True/False).
- Loop through the string:
    - Append the character to `rows[current_row]`.
    - If at top or bottom, reverse `going_down`.
    - Move `current_row` accordingly.
- Join all rows to get the final string.

Python Solution:

```python
def convert(s: str, numRows: int) -> str:
    # Special case: only one row, or string shorter than numRows
    if numRows == 1 or numRows >= len(s):
        return s

    # Create a list for each row
    rows = [''] * numRows
    current_row = 0
    going_down = False

    for char in s:
        rows[current_row] += char
        # Reverse direction at the top or bottom row
        if current_row == 0 or current_row == numRows - 1:
            going_down = not going_down
        # Move up or down
        current_row += 1 if going_down else -1

    # Join all rows to form the result
    return ''.join(rows)
```

Time Complexity: O(N) (where N is the length of `s`)

Space Complexity: O(N) for the rows

Explanation:

- The `rows` list holds the characters for each row.
- We iterate through each character, adding it to the correct row.
- `current_row` tracks where we are.
- `going_down` flips whenever we hit the top or bottom row.

Trace Example:

`s = "PAYPALISHIRING"`, `numRows = 3`

- Start at row 0, going down.
- Add 'P' to row 0, go to row 1.
- Add 'A' to row 1, go to row 2.
- Add 'Y' to row 2 (bottom), reverse to going up, go to row 1.
- Continue...

Try this manually:

`s = "ABCDEFG"`, `numRows = 2`

What should the output be?

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

Reflect:

Did you realize this could also be solved by calculating each character’s position mathematically, instead of simulating the pattern? If you’re up for a challenge, give that a try!

Problem 2: String to Integer (atoi)

LeetCode Problem Link

Problem Statement:

Implement the C function `atoi` which converts a string to an integer.

Rules:

- Ignore leading whitespace.
- Optional '+' or '-' for sign.
- Read digits until a non-digit is found.
- Clamp result to the 32-bit signed integer range.

Example:

Input: `"    -42"`

Output: `-42`

How is this similar to ZigZag Conversion?

Both require **step-by-step parsing** of a string. But here, instead of building a pattern, you're **validating and extracting a number**.

Try this input:

`"   +123abc"`

Brute Force Idea:

- Use Python's `int()` after stripping whitespace and removing letters.
- This skips important validation and clamping; not enough for interviews.

Optimal Approach:

- Process the string character by character, handling all rules.
- Track:
  - Spaces
  - Sign
  - Digits
  - Stopping at first non-digit
  - Clamping to [-2^31, 2^31-1]

Step-by-Step Logic:

- Skip leading spaces.
- Check for '+' or '-' to set sign.
- Read digits, building the number.
- Stop at first non-digit.
- Clamp result to valid 32-bit integer range.

Pseudocode:

```
Initialize index = 0, sign = 1, total = 0
Skip all leading whitespaces
If next char is '+' or '-', set sign
While next char is digit:
    total = total * 10 + int(char)
    If total * sign < INT_MIN: return INT_MIN
    If total * sign > INT_MAX: return INT_MAX
Return total * sign
```

Example:

Input: `"4193 with words"`

- Skip spaces
- No sign, default positive
- Read digits: 4, 1, 9, 3 => 4193
- Stop at space

Output: `4193`

Try this manually:

`"    -91283472332"`

Time Complexity: O(N)

Space Complexity: O(1)

Trace Example:

Input: `"   -42abc"`

- Skip spaces
- '-' sets sign to negative
- Read 4, 2 => 42
- Stop at 'a'
- Return -42

Dry-run test case:

Input: `"0032"`

What should the output be?

Problem 3: Compare Version Numbers

LeetCode Problem Link

Problem Statement:

Given two version numbers as strings (e.g. "1.0.3" and "1.2"), compare them. Return:

- 1 if version1 > version2
- -1 if version1 < version2
- 0 if they are equal

Ignore leading zeros, and missing subversion numbers are treated as 0.

What’s different here?

- Now you’re parsing **multiple numbers separated by dots**.
- The challenge is handling different lengths and leading zeros.

Try this input:

`version1 = "1.01"`, `version2 = "1.001"`

Pseudocode:

```
Split version1 and version2 by '.'
For each pair of numbers (left to right):
    Convert both to int (removes leading zeros)
    If one is greater, return 1 or -1
If one version has more numbers, check if any are > 0
If all numbers are equal, return 0
```

Example:

Input: `"1.0.1"` vs `"1"`

- Compare 1 and 1 -> equal
- Compare 0 and [missing] -> treat as 0, equal
- Compare 1 and [missing] -> 1 > 0, so return 1

Dry-run test case:

`version1 = "1.0.0"`, `version2 = "1"`

Time Complexity: O(N + M) (N, M are lengths of version strings)

Space Complexity: O(N + M) for the split parts

Nudge:

Notice how all three problems involve **breaking strings into meaningful parts** and working through them step by step. Could you use a generator or zip_longest to make this even cleaner?

Summary and Next Steps

We grouped these problems because they all test your ability to **parse and process strings**:

- ZigZag: Transforming a string into a structured pattern
- atoi: Carefully extracting a number from a messy input
- Version comparison: Parsing and comparing multiple numeric parts

Key patterns and insights:

- Be methodical: process one character (or part) at a time.
- Pay attention to edge cases: empty strings, signs, leading/trailing zeros, invalid input.
- Breaking the problem into smaller steps (skip spaces, check sign, read digits) is a winning strategy.

Common mistakes:

- Not handling edge cases (like empty input or extra zeros)
- Using built-in functions without considering required validation
- Forgetting to clamp values, or handle missing subversion numbers

Action Items

- Solve all three problems on your own, including the one with code provided.

- Try a new approach for atoi or version comparison (e.g., regular expressions or itertools).
- Look for other LeetCode problems tagged with "String" and "Parsing" to strengthen your skills.
- Compare your solutions with others, especially for edge case handling.
- Don’t worry if you get stuck. Practicing these techniques will make you a string parsing ninja!

Happy coding!