65. Valid Number

Hard String

Problem Description

The problem is about verifying if a given string `s` represents a valid number according to certain rules defined for decimal numbers and integers. A valid number can be an integer or a decimal number, optionally followed by an exponent. An exponent consists of an 'e' or 'E' followed by an integer. A decimal number includes an optional sign ('+' or '-'), digits, and a decimal point in specific arrangements. An integer consists of an optional sign and one or more digits. A valid number must conform to these rules to be considered as such.

Valid examples include: `"2"`, `"0089"`, `"-0.1"`, `"4."`, `"2e10"`, `"3e+7"`, `"53.5e93"`, etc. Invalid examples are: `"abc"`, `"1a"`, `"1e"`, `"--6"`, `"95a54e53"`, etc.

The aim is to write a function that takes the string `s` and returns `true` if `s` represents a valid number, otherwise `false`.

Intuition

The solution approach starts by checking the string for the necessary conditions and the optional components, like sign, decimal point, and exponent, in their correct order and format.

1. We scan the string to find a valid sequence of digits, accepting an optional leading sign.
2. Next, we look for a decimal point, but we must handle it carefully since it might be part of a valid decimal or an invalid character sequence.
3. Then, if we have an exponent symbol ('e' or 'E'), it must be followed by an integer (optionally with its own sign), but cannot be the first or the last character.
4. Along the way, we must also reject any characters that do not belong in a number, like alphabetic characters other than 'e' or 'E', or unexpected symbols.

Solution Approach

The solution uses string scanning and simple conditional checking to validate the format of the number.

- The algorithm begins by iterating over each character of the string `s` while keeping count of decimal points and exponents encountered.
- A sign character is allowed at the beginning of `s` or immediately after an exponent marker.
- Initially, an attempt is made to skip the optional sign at the beginning of the string, as it does not impact the format in terms of digits or decimal places.
- An edge case is handled where a string could be just a sign or could start with a decimal point with no digits following or preceding it or followed by an exponent marker; such strings are considered invalid.

Next, a while loop commences which iterates over each remaining character:

- It checks for the presence of a decimal point. If a decimal point is found, it confirms whether one has already been encountered or if an exponent has been encountered prior. Since a number can only have one decimal point and it cannot appear after an exponent, such cases are flagged as invalid by returning `false`.
- If an 'e' or 'E' is encountered, it checks if an exponent has already been seen (as there can only be one) or if it's at the beginning of the string (there should be digits before an exponent) or at the end of the string (there must be an integer part following it).
- If the character immediately following an exponent is a sign, it is allowed, but there must be digits following this sign (an exponent cannot be followed by a sign that is the last character).
- Any non-numeric character encountered (excluding signs immediately following an exponent) invalidates the number, triggering an immediate return of `false`.
- The loop continues until all characters are verified. If the string passes all checks, the function returns `true`, indicating that `s` is a valid number.

This algorithm neither necessitates complex data structures nor applies intricate patterns, relying instead on sequential character checking and state tracking with simple boolean flags indicating the presence of specific characters ('.', 'e', 'E'). Notably, the solution is fine-tuned to the specific validation rules set out in the problem description.

Example Walkthrough

Let's take the string `s = "3.5e+2"` and walk through the steps to determine if it represents a valid number according to the solution approach:

1. The algorithm starts by looking for an optional sign. The first character is '3', which is a valid digit and not a sign, so the algorithm moves on.
2. As the algorithm continues, it finds a decimal point after '3'. Since no decimal point has been encountered yet and an exponent has not appeared, this is still a potential valid number.
3. The next character is '5', which is a digit, so the reading continues without any issue.
4. Following the digit '5', the algorithm encounters an 'e', indicating the start of an exponent. Since this is the first exponent character and there have been digits before, the pattern is still valid.
5. It then sees a '+', which is an allowed sign for the exponent as long as it's immediately after the 'e' and is not the last character in the string.
6. Finally, the algorithm finds a '2', which is a digit following the exponent and its sign. This confirms a valid integer part of the exponent.

Since the end of the string is reached without any invalid character or sequence, the algorithm concludes that `s = "3.5e+2"` is a valid number and returns `true`. This example successfully represents a number with both a decimal and an exponent, including an optional sign for the exponent.

Python Solution

```python
class Solution:
    def isNumber(self, s: str) -> bool:
        # Length of the input string.
        length = len(s)
        # Start index for traversing the string.
        index = 0

        # Check for optional sign at the beginning.
        if s[index] in '+-':
            index += 1

        # Empty string after a sign or no numeric part is invalid.
        if index == length:
            return False

        # Single dot without digits or dot directly followed by exponent is invalid.
        if s[index] == '.' and (index + 1 == length or s[index + 1] in 'eE'):
            return False

        # Counters for dots and exponent characters.
        dot_count = exponent_count = 0

        # Traverse the string starting from the current index.
        while index < length:
            if s[index] == '.':
                # If there's already a dot or an exponent, it's invalid.
                if exponent_count or dot_count:
                    return False
                dot_count += 1
            elif s[index] in 'eE':
                # If there's already an exponent, or this is the first character, or there isn't a number following, it's invalid.
                if exponent_count or index == 0 or index == length - 1:
                    return False
                exponent_count += 1
                # Check for an optional sign after the exponent.
                if s[index + 1] in '+-':
                    index += 1
                    # If the string ends after the sign, it's invalid.
                    if index == length - 1:
                        return False
            # Non-numeric, non-dot, and non-exponent characters are invalid.
            elif not s[index].isdigit():
                return False
            index += 1

        # If all checks pass, the string represents a valid number.
        return True

```

Java Solution

```java
class Solution {
    public boolean isNumber(String s) {
        int length = s.length();
        int index = 0;

        // Check for an optional sign at the beginning
        if (s.charAt(index) == '+' || s.charAt(index) == '-') {
            index++;
        }

        // Check if the string is non-empty after optional sign
        if (index == length) {
            return false;
        }

        // Check for string starting with a dot followed by e/E or end of string
        if (s.charAt(index) == '.' && (index + 1 == length || s.charAt(index + 1) == 'e' ||
            s.charAt(index + 1) == 'E')) {
            return false;
        }

        int dotCount = 0; // Count of dots in the string
        int eCount = 0;   // Count of 'e's or 'E's in the string

        // Iterate over the characters in the string
        for (int i = index; i < length; ++i) {
            char currentChar = s.charAt(i);

            if (currentChar == '.') {
                // If there's an 'e/E' before the dot or it's a second dot, it's invalid
                if (eCount > 0 || dotCount > 0) {
                    return false;
                }
                dotCount++;
            } else if (currentChar == 'e' || currentChar == 'E') {
                // Check for multiple 'e/E', 'e/E' at start/end or directly after a sign
                if (eCount > 0 || i == index || i == length - 1) {
                    return false;
                }
                eCount++;
                // Check for a sign immediately after 'e/E'
                if (s.charAt(i + 1) == '+' || s.charAt(i + 1) == '-') {
                    // Skip the next character if it's a sign
                    // If it leads to end of the string, it's invalid
                    if (++i == length - 1) {
                        return false;
                    }
                }
            } else if (currentChar < '0' || currentChar > '9') {
                // If the character is not a digit, it's invalid
                return false;
            }
        }
        // If all checks pass, it's a number
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine if a given string is a valid number
    bool isNumber(string s) {
        int length = s.size(); // Store the size of the string
        int index = 0; // Start index for scanning the string

        // Optional sign in front; increment index if it exists
        if (s[index] == '+' || s[index] == '-') index++;

        // If string is empty or has only a sign, return false
        if (index == length) return false;

        // If string starts with a dot and is not followed by a digit or exponent,
        // it is not a valid number
        if (s[index] == '.' && (index + 1 == length || s[index + 1] == 'e' || s[index + 1] == 'E')) return false;

        int dotCount = 0, eCount = 0; // Counters for the dots and exponents encountered

        // Loop over the rest of the string
        for (int j = index; j < length; ++j) {
            if (s[j] == '.') {
                // If we encounter a dot after an exponent or if it's a second dot, it's invalid
                if (eCount || dotCount) return false;
                dotCount++; // Increment dot counter
            } else if (s[j] == 'e' || s[j] == 'E') {
                // If we encounter an exponent after another exponent, or if it's at the start
                // or end of the number, it's invalid
                if (eCount || j == index || j == length - 1) return false;
                eCount++; // Increment exponent counter
                // Skip the sign of the exponent part if it's there
                if (s[j + 1] == '+' || s[j + 1] == '-') {
                    if (++j == length - 1) return false; // If only a sign follows the exponent, it's invalid
                }
            } else if (s[j] < '0' || s[j] > '9') {
                // If the character is not a digit, it's invalid
                return false;
            }
        }
        // If all conditions are met, it's a valid number
        return true;
    }
};

```

Typescript Solution

```typescript
// Determines if a given string is a valid number
function isNumber(s: string): boolean {
    // Store the size of the string
    const length: number = s.length;
    // Start index for scanning the string
    let index: number = 0;

    // Optional sign in front; increment index if it exists
    if (s[index] === '+' || s[index] === '-') index++;

    // If string is empty or has only a sign, return false
    if (index === length) return false;

    // If string starts with a dot and is not followed by a digit or exponent, it is not a valid number
    if (s[index] === '.' && (index + 1 === length || s[index + 1] === 'e' || s[index + 1] === 'E')) return false;

    // Counters for the dots and exponents encountered
    let dotCount: number = 0;
    let eCount: number = 0;

    // Loop over the rest of the string
    for (let j = index; j < length; j++) {
        if (s[j] === '.') {
            // If we encounter a dot after an exponent or if it's a second dot, it's invalid
            if (eCount || dotCount) return false;
            dotCount++; // Increment dot counter
        } else if (s[j] === 'e' || s[j] === 'E') {
            // If we encounter an exponent after another exponent, or if it's at the start or end of the number, it's invalid
            if (eCount || j === index || j === length - 1) return false;
            eCount++; // Increment exponent counter
            // Skip the sign of the exponent part if it's there
            if (s[j + 1] === '+' || s[j + 1] === '-') {
                // Increments the index to skip the sign, then it checks if only a sign follows the exponent, it's invalid
                if (++j === length - 1) return false;
            }
        } else if (!isDigit(s[j])) {
            // If the character is not a digit, it's invalid
            return false;
        }
    }
    // If all conditions are met, it's a valid number
    return true;
}

// Helper function to determine if a character is a digit
function isDigit(char: string): boolean {
    return char >= '0' && char <= '9';
}

```

Time and Space Complexity

The given Python code snippet is designed to validate whether the input string `s` represents a valid number according to certain rules. To analyze its computational complexity, let's consider the size of the input string `s`, which is `n`.

Time Complexity

The function primarily utilizes a single while loop that traverses the input string `s`, which runs at most `n` times, where `n` is the length of the string.

- Each check inside the loop (`s[j] == '.'`, `s[j] in 'eE'`, `s[j].isnumeric()`, etc.) can be considered to have a constant time complexity, i.e., `O(1)`.
- The while loop iterates over each character in the string once. The `if` checks within the loop do not contain any nested loops, and each condition is evaluated in constant time.

Thus, the overall time complexity of the function is `O(n)`.

Space Complexity

- The space complexity is mainly due to the variables `i`, `j`, `dot`, `e`, and the input string `s`. There are no data structures that grow with the input size.
- The function uses a constant amount of extra space aside from the input string itself since no additional data structures such as lists or arrays are utilized to process the input.

Therefore, the space complexity of the function is `O(1)`, indicating constant space usage.