8. String to Integer (atoi)

Medium String

Leetcode Link

Problem Description

The `myAtoi` function is designed to mimic the behavior of the `atoi` function in C/C++, converting a string to a 32-bit signed integer following a set of specific rules. The steps of the algorithm are:

1. Start by skipping any leading whitespace in the input string, as these are irrelevant for number conversion.
2. Identify if the first non-whitespace character is a sign indicator ('-' for negative, '+' for positive). If found, note the sign for the final number. If no sign is specified, the number is considered positive by default.
3. Read the subsequent characters until a non-digit is encountered or the end of the string is reached. These characters represent the numerical digits of the string.
4. Convert the sequence of digits into an integer. If no digits are found, the result should be 0. Apply the previously noted sign to this integer.
5. Make sure the resulting integer fits within the range of a 32-bit signed integer. If it does not, "clamp" the value to the nearest boundary of the range, which is either -2^{31} or $2^{31} - 1$.
6. Return the processed integer as the final result.

A couple of important notes to consider:

- The only whitespace character to be considered is the space character `' '`.
- All characters after the numeric part of the string should not be ignored, as they can influence the final output if they are non-digits.

Intuition

To convert the string to an integer while following the rules, we need to go step by step:

1. **Skip Whitespaces:** Check each character until we encounter a non-whitespace character or the end of the string. This is achieved by using a while loop that increments an index whenever a space is found.
2. **Check for a Sign:** Once we hit a non-whitespace character, we decide on the sign of our result. If it's a '-' we set the sign to -1, otherwise, we assume it's positive (1).
3. **Convert String to Integer:** We parse the string character by character as long as they are numeric (digits). We need to be mindful of possible overflow past the 32-bit integer range, which happens when our number exceeds the maximum allowed value when multiplied by 10 or when adding the next digit.
4. **Handle Overflows:** Since we're working within the bounds of a signed 32-bit integer, we ensure that if our number goes beyond the limits is clamped to the closest limit, either the lower bound -2^{31} or the upper bound $2^{31} - 1$.

By carefully following each of these steps, we simulate the atoi function's behavior, producing a 32-bit signed integer from an input string.

Solution Approach

The solution implements the conversion process through a series of steps that correspond to the rules provided in the problem description. Here's a walk-through of the key components:

1. **Initialization:** The function first checks if the input string `s` is not empty. It then initializes important variables such as `n` for the length of the string, `i` for the current index, and `res` for the resulting integer value.
2. **Whitespace Skipping:** Using a `while` loop, the implementation skips over the leading whitespace characters `' '` until a non-whitespace character is encountered by incrementing the `i` index. If `i` becomes equal to `n` (the length of the string), that means the string contained only whitespace, and we return `0`.
3. **Sign Detection:** The next character is checked to determine the sign of the resultant integer. If it's a `'-'`, the variable `sign` is set to `-1`, otherwise `1`. If a sign character is found, `i` is incremented to move past it.
4. **Integer Conversion:** A `while` loop is used to iterate over the remaining characters of the string, as long as they are digits. Each digit is converted to an integer using `int(s[i])` and added to the result `res` after multiplying the current `res` by `10` (shifting the number one decimal place to the left). The loop stops if a non-digit character appears.
5. **Overflow Handling:** Before adding the new digit to `res`, we check for potential overflow. For positive numbers, it checks if `res` is greater than $(2^{31} - 1) // 10$, or if `res` is equal to $(2^{31} - 1) // 10$ and the next digit (`i`) is greater than 7 (since $2^{31} - 1$ ends in 7). For negative numbers, the same logic is applied but with the limits of -2^{31}. If an overflow is detected, the maximum or minimum 32-bit signed integer value is returned based on the `sign`.
6. **Result Return:** Once the loop finishes, the integer is adjusted with the sign and returned as the result.

The code demonstrates good practices such as:

- Early exits to avoid unnecessary processing (empty string or string with only spaces).
- Overflow protection by checking the conditions before the actual overflow could occur.
- Efficient string traversal by using index arithmetic rather than creating new strings or arrays.

The algorithm does not use any complex data structures, as it only requires integer variables to keep track of the result, the index, and the sign. It is a straightforward and efficient implementation governed by control statements to ensure that the resulting integer adheres to the constraints defined in the problem.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the input string `s = " -42"` which we want to convert to a 32-bit signed integer following the defined rules.

1. **Initialization:**
   - The string `s` is not empty, so we initialize `n=5` (= being the string's length), `i=0` for the current index, and `res=0` for the resulting integer value.
2. **Whitespace Skipping:**
   - We use a `while` loop to skip the leading whitespaces. After skipping, `i=1` as the first three characters are spaces.
3. **Sign Detection:**
   - The next character is `'-'`, which indicates a negative number. We set `sign=-1` and increment `i` to move past the sign, and now `i=2`.
4. **Integer Conversion:**
   - We reach the digits now. With a `while` loop, we start adding the digits to `res`. The first character `'4'` is converted to an integer `4`, and `res` becomes `4`. Then `i` is incremented.
   - Next, `i=3` and the character is `'2'`. We multiply `res` by 10 (shifting to the left) and add 2, making `res=42`. `i` is incremented again, and with `i=4`, we have reached the end of the string, and the loop ends.
5. **Overflow Handling:**
   - During integer conversion, we did not encounter any potential overflow cases, as the number `-42` is within the range of a 32-bit signed integer.
6. **Result Return:**
   - The final step is to apply the sign to `res`. With `sign=-1`, the result becomes `-42`, which is then returned.

This example adhered to the steps of the algorithm, resulting in the correct 32-bit signed integer conversion of the input string `s`. The key to solving this problem was carefully handling each character by categorizing it as whitespace, sign, or digit, and applying appropriate operations to avoid overflows.

Python Solution

```python
class Solution:
    def myAtoi(self, str: str) -> int:
        # Return 0 if string is empty
        if not str:
            return 0

        length_of_string = len(str)
        index = 0

        # Skip leading whitespaces
        while index < length_of_string and str[index] == ' ':
            index += 1

        # After skipping whitespaces, if we're at the end it means it's an empty string
        if index == length_of_string:
            return 0

        result = 0
        max_safe_int = (2 ** 31 - 1) // 10  # Precomputed to avoid overflow

        while index < length_of_string:
            # If the current character is not a digit, break from loop
            if not str[index].isdigit():
                break

            digit = int(str[index])

            # Check for overflow cases
            if result > max_safe_int or (result == max_safe_int and digit > 7):
                return 2 ** 31 - 1 if sign > 0 else -2 ** 31  # Clamp to INT_MIN or INT_MAX

            # Append current digit to result
            result = result * 10 + digit
            index += 1

        # Apply sign and return final result
        return sign * result
```

Java Solution

```java
class Solution {
    public int myAtoi(String str) {
        // Ensure the input string is not null
        if (str == null) {
            return 0;
        }

        int length = str.length();

        // If the string is empty, return 0
        if (length == 0) {
            return 0;
        }

        int index = 0;

        // Skip whitespace characters
        while (index < length && str.charAt(index) == ' ') {
            index++;
        }

        // If we reached the end of string after skipping spaces, return 0
        if (index == length) {
            return 0;
        }

        // Determine the sign based on the current character
        int sign = 1;
        if (str.charAt(index) == '-') {
            sign = -1;
            index++;
        } else if (str.charAt(index) == '+') {
            index++;
        }

        int result = 0;
        // Pre-calculate the threshold to check for overflow
        int threshold = Integer.MAX_VALUE / 10;

        // Convert the number
        while (index < length) {
            char currentChar = str.charAt(index);

            // Break if the current character is not a digit
            if (currentChar < '0' || currentChar > '9') {
                break;
            }

            // Check for overflow when adding a new digit
            if (result > threshold || (result == threshold && currentChar > '7')) {
                return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
            }

            // Update result with the new digit
            result = result * 10 + (currentChar - '0');
            index++;
        }

        // Apply the determined sign to the result and return
        return sign * result;
    }
}
```

C++ Solution

```cpp
#include <climits>
#include <string>

class Solution {
public:
    /**
     * Converts the string argument to an integer (32-bit signed integer).
     * Trims whitespace, manages sign, and processes characters until a non-digit is found,
     * or the number goes out of the 32-bit signed integer range.
     *
     * @param s - the string to convert to an integer.
     * @return The parsed integer, or 0 if no conversion is possible.
     */
    int myAtoi(std::string s) {
        // Trim leading whitespace
        size_t start = s.find_first_not_of(" \t\n\r");
        if (start == std::string::npos) return 0;  // Return 0 if the string is just whitespace

        s = s.substr(start);  // Trim leading whitespace

        bool isPositive = true;  // Flag indicating if the number is positive
        int i = 0;               // Index for iteration
        int answer = 0;          // Variable to accumulate the parsed number

        // Check if there is a sign in the beginning
        if (s[i] == '+') {
            isPositive = true;  // Positive number
            i++;
        } else if (s[i] == '-') {
            isPositive = false; // Negative number
            i++;
        }

        // Process each character in the string
        for (; i < s.length(); ++i) {
            // Calculate the numeric value of the current character
            int currentValue = s[i] - '0';

            // Break the loop if the character is not a digit
            if (currentValue < 0 || currentValue > 9) break;

            // Check for overflow: if the current answer is already at the risk of overflow
            // with an additional digit, return the respective limit
            if (answer > INT_MAX / 10 ||
                (answer == INT_MAX / 10 && currentValue > INT_MAX % 10)) {
                return isPositive ? INT_MAX : INT_MIN;
            }

            // Update the answer by adding the current digit
            answer = answer * 10 + currentValue;
        }

        // Return the final answer, considering the sign
        return isPositive ? answer : -answer;
    }
};
```

Typescript Solution

```typescript
/**
 * Converts the string argument to an integer (32-bit signed integer).
 * Trims whitespace, manages sign, and processes characters until a non-digit is found,
 * or the number goes out of the 32-bit signed integer range.
 * @param str - the string to convert to an integer.
 * @return The parsed integer, or 0 if no conversion is possible.
 */
function myAtoi(str: string): number {
    // Trim leading or trailing whitespace
    str = str.trim();

    // Return 0 if the string is empty after trimming
    if (!str) return 0;

    let isPositive: boolean = true;  // Flag indicating if the number is positive
    let i: number = 0;               // Index for iteration
    let answer: number = 0;          // Variable to accumulate the parsed number

    // Check if there is a sign in the beginning
    if (str[i] === '+') {
        isPositive = true;  // Positive number
        i++;
    } else if (str[i] === '-') {
        isPositive = false; // Negative number
        i++;
    }

    // Process each character in the string
    for (; i < str.length; i++) {
        // Calculate the numeric value of the current character
        let currentValue: number = str.charCodeAt(i) - '0'.charCodeAt(0);

        // Break the loop if the character is not a digit
        if (currentValue < 0 || currentValue > 9) break;

        // Check for overflow: if the current answer is already at the risk of overflow
        // with an additional digit, return the respective limit
        if (answer > Math.floor(2147483647 / 10) ||
            (answer === Math.floor(2147483647 / 10) &&
             currentValue > 7)) {
            return isPositive ? 2147483647 : -2147483648;
        }

        // Update the answer by adding the current digit
        answer = answer * 10 + currentValue;
    }

    // Return the final answer, considering the sign
    return isPositive ? answer : -answer;
};
```

Time and Space Complexity

Time Complexity

The time complexity of the given function is $O(n)$, where n is the length of the string. Here's the breakdown:

- We iterate once over the string to trim whitespaces, which takes $O(n)$ in the worst case (if all characters are spaces).
- Checking if the character is a sign (`'+'` or `'-'`) is $O(1)$.
- The following while loop iterates over the rest of the string, but it runs at most n times, which is also $O(n)$ in the worst case.
- Each operation inside the while loop, including checking if a character is a digit, converting it to an integer, checking for overflow, and updating the result, is done in constant time $O(1)$.

Therefore, the dominating factor in the time complexity is the length of the string n, resulting in $O(n)$ overall.

Space Complexity

The space complexity of the function is $O(1)$ because we use a fixed number of integer variables (`i`, `sign`, `res`, `flag`, and `n`) and they do not depend on the size of the input string. No additional structures are allocated that would grow with the input size. The space used by `s` is not counted since it is an input to the function and not additional space allocated by the function itself.