227. Basic Calculator II

Medium Stack Math String

Problem Description

The problem requires us to evaluate a mathematical expression given as a string. The expression includes integers and the operators '+', '-', '*', and '/'. We need to process the expression and return the result as an integer. The division operation should truncate the result toward zero, meaning it should not result in any fractional part. We need to handle the expression correctly, keeping in mind the different precedence of the operators: multiplication and division have higher precedence over addition and subtraction.

All the numbers and the result of the computation are guaranteed to be within the range of a 32-bit signed integer. It is also worth noting that the given expression will be valid, so there is no need to handle any syntax errors.

Additionally, it is important to mention that we cannot use any built-in functions that can evaluate strings as mathematical expressions directly, such as the `eval()` function in Python.

Intuition

To solve this problem, we need to implement an algorithm that can parse and evaluate the expression respecting the correct operator precedence. Since we can't use built-in evaluation functions, we have to simulate the calculation manually. One common approach to evaluating expressions is to use a stack.

Here's the intuition behind using a stack in this solution:

1. When we encounter a number, we store it for the next operation.
2. When we encounter an operator, we decide what to do based on the previous operator.
3. For '+' or '-', we need to add or subtract the number to/from our result. But since multiplication and division have higher precedence, we cannot directly add/subtract the number – we push it onto the stack.
4. For '*' or '/', we immediately perform the operation with the previous number because they have higher precedence.
5. Updating the current operator as we go allows us to know which operation to perform next.

To achieve the evaluation:

- We iterate over the string.
- We keep track of the current number and the last operator we have seen.
- When we see a new operator, we perform the necessary operation based on the last operator, which could mean pushing to the stack or popping from the stack, performing an operation, and then pushing back the result.
- At the end of the expression, we perform the operation for the last seen operator.
- Since all numbers and results are within the range of a signed 32-bit integer, we do not need to worry about overflow.

After traversing the entire string and performing all stack operations, the result of the expression will be the sum of all numbers present in the stack.

Solution Approach

The implementation follows these steps, making use of a stack and a variable to keep track of the current number (initially `0`) and another to remember the last operator (initially `'+'`):

1. **Initialize Variables**: A stack `stk` is used to handle the numbers and the intermediate results. The variable `v` is used to build the current number as we parse the string. Another variable, `sign`, stores the last encountered operator, which is initially set to `'+'`.

2. **Parse the String**: The string `s` is parsed character by character using a `for` loop. If the current character `c` is a digit, it's a part of the current number `v`. We update `v` by shifting the current value by one digit to the left (multiplying by 10) and adding the digit (`v * 10 + int(c)`).

3. **Handle Operators and End of String**: We need to perform an action whenever we encounter an operator or the end of the string (because the last number might not be followed by an operator). So we check if the character `c` is in '+-*/', or if it is the last character in the string (`i == n - 1`).

4. **Perform Operations Based on Last Sign**:

   - If the last sign was `'+'`, push the current value of `v` onto the stack.
   - If it was `'-'`, push the negative of `v` onto the stack.
   - If it was `'*'`, pop the top value from the stack, multiply it by `v`, and push the result back onto the stack.
   - If it was `'/'`, pop the top value from the stack, truncate divide it by `v`, and push the result back onto the stack. This is achieved by using `int(stk.pop() / v)` which truncates towards zero in Python.

   After processing the current number and operator, `sign` is updated to the current operator `c`, and `v` is reset to `0` to hold the next number.

5. **Evaluate the Stack**: After the loop is finished, all values left in the stack are parts of the expression where addition and subtraction should be applied. As we pushed the results of multiplications and divisions and also considered the sign for addition and subtraction, simply summing all the values in the stack using `sum(stk)` gives us the final result.

The function finally returns the sum of all elements in the stack, which is the evaluated result of the given expression.

Example Walkthrough

Let's walk through an example to illustrate the solution approach by evaluating the expression `s = "3+5*2-4/2"` step by step.

1. **Initialize Variables**: We start by initializing our stack `stk`, our variable `v` to `0`, and our operator variable `sign` to `'+'`.

2. **Parse the String**: We parse each character of our string `s`.

   - For the first character `'3'`, since it is a digit, we update `v` to be `v * 10 + 3`, which is `3`.
   - The next character is `'+'`. Since we encounter an operator, we perform the action for the last sign ('+'). We push `3` onto the stack (as our `v` is `3` and the sign is `'+'`). We then update `sign` to `'+'`, and reset `v` to `0`.

3. Continue parsing:

   - Next, we encounter `'5'`, which we multiply by 10 and add to our `v` (0 at this moment), resulting in `5`.
   - Similarly, we encounter the character `'2'` after `'5'` and the `'*'` operator, but we don't do anything yet because the number `2` is not complete. After `'2'`, our current `v` is `52`.
   - Then we see the `'*'` operator. Now we need to act on the previous sign which was `'+'`, so we push `5` into our stack and reset `v` to `2` because `2` is part of the next operation which involves multiplication. We update `sign` to be `'*'`.

4. Handle Operators and End of String:

   - Continue parsing: next, we encounter `'-'`, and since our last operator was `'*'`, we multiply the most recent number in the stack (`5`) by `v` (which is `2`), and push the result `10` back onto the stack. Update `sign` to `'-'` and `v` to `0`.
   - Parsing the next number `'4'`, just increment `v` to `4`.
   - We see the `'/'` operator, which means—since the last sign was `'-'`—we push `-4` onto the stack. Update `sign` to `'/'` and reset `v` to `0`.
   - Lastly, the character `'2'` sets `v` to `2`.

5. At this point, we've reached the end of the string. The last operator is `'/'`, so we must divide the top value of the stack `-4` by `v`, which is `2`, truncate towards zero, and push the result back to the stack (`-2`).

6. **Evaluate the Stack**: We now evaluate the contents of the stack, which has `[3, 10, -2]`. By summing them up, we get the final result of `3 + 10 - 2 = 11`.

```
# Stack operations in detail:
# stk = [] after initialization
# stk = [3] after processing "3+"
# stk = [3, 5] after processing "5"
# stk = [10] after processing "5*2"
# stk = [10, -4] after processing "-4"
# stk = [10, -2] after processing "-4/2" (since the division truncates towards zero)
# Final sum of stack = 3 + 10 - 2 = 11
```

The above step-by-step processing adheres to the operator precedence and accurately evaluates the mathematical expression without the use of any built-in evaluators. The final result for the expression `s = "3+5*2-4/2"` is `11`.

Python Solution

```python
class Solution:
    def calculate(self, s: str) -> int:
        # Initialize the value to hold current number and stack to store values.
        value, n = 0, len(s)
        # Set the initial sign to '+' (plus).
        sign = '+'
        # Initialize an empty stack to keep numbers and intermediate results.
        stack = []

        # Iterate over each character in the input string.
        for i, char in enumerate(s):
            # If the character is a digit, update 'value' to be the current multi-digit number.
            if char.isdigit():
                value = value * 10 + int(char)

            # If we reach the end of the string or encounter an operator.
            if i == n - 1 or char in '+-*/':
                # Perform an action depending on the last sign observed.
                if sign == '+':
                    # If the sign is plus, push the current value onto the stack.
                    stack.append(value)
                elif sign == '-':
                    # If the sign is minus, push the negative of the value onto the stack.
                    stack.append(-value)
                elif sign == '*':
                    # If the sign is times, multiply the top of the stack by the current value and push it back.
                    stack.append(stack.pop() * value)
                elif sign == '/':
                    # If the sign is divide, take the top of the stack, divide by current value and push the result back.
                    # Use integer division to round towards zero.
                    stack.append(int(stack.pop() / value))

                # Update the sign to the current operator.
                sign = char
                # Reset 'value' for the next number.
                value = 0

        # The final result is the sum of the values in the stack.
        return sum(stack)

```

Java Solution

```java
import java.util.Deque;
import java.util.ArrayDeque;

class Solution {
    public int calculate(String s) {
        Deque<Integer> stack = new ArrayDeque<>(); // Use a stack to manage intermediate results
        char operation = '+'; // Initialize the operation as addition
        int value = 0; // This will hold the current number

        // Iterate through each character in the input string
        for (int i = 0; i < s.length(); ++i) {
            char currentChar = s.charAt(i);

            // If the current character is a digit, accumulate into value
            if (Character.isDigit(currentChar)) {
                value = value * 10 + (currentChar - '0');
            }

            // If we've reached the end of the string or we encounter an operator
            if (i == s.length() - 1 || currentChar == '+' || currentChar == '-'
                                    || currentChar == '*' || currentChar == '/') {

                // Perform the operation based on the previous sign
                switch (operation) {
                    case '+':
                        stack.push(value); // Add the value to the stack
                        break;
                    case '-':
                        stack.push(-value); // Push the negated value for subtraction
                        break;
                    case '*':
                        stack.push(stack.pop() * value); // Multiply with the top value on the stack
                        break;
                    case '/':
                        stack.push(stack.pop() / value); // Divide the top value with current value
                        break;
                }

                operation = currentChar; // Update the operation for the next iteration
                value = 0; // Reset value for the next number
            }
        }

        int result = 0; // Initialize result to accumulate stack values
        // Pop values from the stack and add them to calculate the result
        while (!stack.isEmpty()) {
            result += stack.pop();
        }

        return result; // Return the final calculated result
    }
}

```

C++ Solution

```cpp
#include <cctype>
#include <stack>
#include <string>

class Solution {
public:
    int calculate(std::string s) {
        // Initialize the current value and the total size of the string
        int currentValue = 0, stringSize = s.size();
        // Set the initial sign to '+'
        char operationSign = '+';
        // Stack to hold intermediate results
        std::stack<int> calculationStack;
        for (int i = 0; i < stringSize; ++i) {
            char currentChar = s[i];
            // If character is a digit, build the number by multiplying the current value by 10 and adding the digit's integer valu
            if (std::isdigit(currentChar)) currentValue = currentValue * 10 + (currentChar - '0');
            // If we reach the end of the string or encounter an operation
            if (i == stringSize - 1 || currentChar == '+' || currentChar == '-' || currentChar == '*' || currentChar == '/') {
                switch (operationSign) {
                    // If the sign is '+', push the current value to the stack
                    case '+':
                        calculationStack.push(currentValue);
                        break;
                    // If the sign is '-', push the negative of the current value to the stack
                    case '-':
                        calculationStack.push(-currentValue);
                        break;
                    // If the sign is '*', multiply the top element of the stack with the current value and push the result back to
                    case '*':
                        {
                            int topValue = calculationStack.top();
                            calculationStack.pop();
                            calculationStack.push(topValue * currentValue);
                        }
                        break;
                    // If the sign is '/', divide the top element of the stack by the current value and push the result back to the
                    case '/':
                        {
                            int topValue = calculationStack.top();
                            calculationStack.pop();
                            calculationStack.push(topValue / currentValue);
                        }
                        break;
                }
                // Update the operation sign for the next operation
                operationSign = currentChar;
                // Reset the current value
                currentValue = 0;
            }
        }
        // Initialize the answer to 0
        int result = 0;
        // Add up all values in the stack to get the final result
        while (!calculationStack.empty()) {
            result += calculationStack.top();
            calculationStack.pop();
        }
        // Return the result of expression evaluation
        return result;
    }
};

```

Typescript Solution

```typescript
// Import Stack data structure for use in calculation
import { Stack } from "typescript-collections";

// Function to evaluate the arithmetic expression within a string
function calculate(expression: string): number {
    // Initialize currentValue to hold the numbers as they are parsed
    let currentValue: number = 0;
    // Determine the length of the expression string
    let expressionSize: number = expression.length;
    // Start with a default operationSign of '+'
    let operationSign: string = '+';
    // Use a stack to keep track of calculation results
    let calculationStack = new Stack<number>();

    for (let i = 0; i < expressionSize; ++i) {
        let currentChar = expression[i];

        // If the current character is a digit, construct the number
        if (!isNaN(parseInt(currentChar))) {
            currentValue = currentValue * 10 + (parseInt(currentChar));
        }

        // Perform operations at the end of the string or at an operator
        if (i === expressionSize - 1 || currentChar === '+' || currentChar === '-' || currentChar === '*' || currentChar === '/') {
            switch (operationSign) {
                // Push current value to stack for a '+' operation
                case '+':
                    calculationStack.push(currentValue);
                    break;
                // Push the negative of current value for a '-' operation
                case '-':
                    calculationStack.push(-currentValue);
                    break;
                // Multiply the stack's top with current value for a '*' operation
                case '*':
                    {
                        let topValue = calculationStack.pop();
                        if(topValue !== undefined) {
                            calculationStack.push(topValue * currentValue);
                        }
                    }
                    break;
                // Divide the stack's top by current value for a '/' operation
                case '/':
                    {
                        let topValue = calculationStack.pop();
                        if(topValue !== undefined) {
                            calculationStack.push(Math.trunc(topValue / currentValue));
                        }
                    }
                    break;
            }
            // Update the operationSign with current character
            operationSign = currentChar;
            // Reset currentValue for the next part of the string
            currentValue = 0;
        }
    }

    // Calculate the final result by summing up the values in the stack
    let result: number = 0;
    while (!calculationStack.isEmpty()) {
        let value = calculationStack.pop();
        if(value !== undefined) {
            result += value;
        }
    }

    // Return the result of the expression evaluation
    return result;
}

// Note: Since this bit of code uses the 'Stack' implementation from the 'typescript-collections' library,
// you would have to install that npm package to use this stack data structure.

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the input string. The algorithm iterates through each character of the string exactly once, and the operations within the loop are constant time operations, including digit extraction, basic arithmetic operations, and stack manipulation. The final summation of stack elements also takes `O(n)` time in the worst case where all characters result in individual numbers pushed onto the stack.

Space Complexity

The space complexity is `O(n)` for the stack that stores numbers based on the operations. In the worst case, where the string is composed of numbers separated by addition signs, each number is pushed onto the stack, reaching a maximum stack size proportional to the length of the input string.