227. Basic Calculator II Medium Stack Math String **Leetcode Link** 

## **Problem Description**

'+', '-', '\*', 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

The problem requires us to evaluate a mathematical expression given as a string. The expression includes integers and the operators

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

Here's the intuition behind using a stack in this solution: 1. When we encounter a number, we store it for the next operation.

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.

another to remember the last operator (initially '+'):

character in the string (i == n - 1).

4. Perform Operations Based on Last Sign:

approach to evaluating expressions is to use a stack.

- 4. For '\*' or '/', we immediately perform the operation with the previous number because they have higher precedence.

2. When we encounter an operator, we decide what to do based on the previous operator.

- 5. Updating the current operator as we go allows us to know which operation to perform next. To achieve the evaluation:
- 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.

We iterate over the string.

- 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**

achieved by using int(stk.pop() / v) which truncates towards zero in Python.

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 '+'.

The implementation follows these steps, making use of a stack and a variable to keep track of the current number (initially 0) and

### 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).

number.

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

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

 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

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

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.

5. Evaluate the Stack: After the loop is finished, all values left in the stack are parts of the expression where addition and

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

Similarly, we encounter the character '2' after '5' and the '\*' operator, but we don't do anything yet because the number 2

3. Continue parsing:

stack (5) by v (which is 2), and push the result 10 back onto the stack. Update sign to '-' and v to 0.

○ Next, we encounter '5', which we multiply by 10 and add to our v (0 at this moment), resulting in 5.

stack (as our v is 3 and the sign is '+'). We then update sign to '+', and reset v to 0.

#### 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

Parsing the next number '4', just increment v to 4.

which is 2, truncate towards zero, and push the result back to the stack (-2).

7 # stk = [10, -2] after processing "-4/2" (since the division truncates towards zero)

# Initialize the value to hold current number and stack to store values.

Lastly, the character '2' sets v to 2.

1 # Stack operations in detail:

**Python Solution** 

1 class Solution:

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

6

9

10

11

12

13

14

15

16

17

Java Solution

1 import java.util.Deque;

class Solution {

2 import java.util.ArrayDeque;

2 # stk = [] after initialization

3 # stk = [3] after processing "3+"

**4** # stk = [3, 5] after processing "5"

5 # stk = [10] after processing "5\*2"

6 # stk = [10, -4] after processing "-4"

8 # Final sum of stack = 3 + 10 - 2 = 11

def calculate(self, s: str) -> int:

# Set the initial sign to '+' (plus).

elif sign == '-':

elif sign == '\*':

elif sign == '/':

sign = char

public int calculate(String s) {

value = 0

return sum(stack)

stack.append(-value)

stack.append(stack.pop() \* value)

# Use integer division to round towards zero.

stack.append(int(stack.pop() / value))

# Update the sign to the current operator.

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

char operation = '+'; // Initialize the operation as addition

// If the current character is a digit, accumulate into value

int value = 0; // This will hold the current number

for (int i = 0; i < s.length(); ++i) {</pre>

char currentChar = s.charAt(i);

if (Character.isDigit(currentChar)) {

// Iterate through each character in the input string

value = value \* 10 + (currentChar - '0');

# Reset 'value' for the next number.

value, n = 0, len(s)

sign = '+'

Example Walkthrough

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:

○ 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.

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,

○ Continue parsing: next, we encounter '-', and since our last operator was '\*', we multiply the most recent number in the

- 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.
- 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.

```
# 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)
```

# If the sign is times, multiply the top of the stack by the current value and push it back.

# If the sign is divide, take the top of the stack, divide by current value and push the result back.

# If the sign is minus, push the negative of the value onto the stack.

Deque<Integer> stack = new ArrayDeque<>(); // Use a stack to manage intermediate results

```
34
35
36
37
```

C++ Solution

#include <cctype>

#include <string>

int calculate(std::string s) {

// Set the initial sign to '+'

// Stack to hold intermediate results

for (int i = 0; i < stringSize; ++i) {</pre>

switch (operationSign) {

break;

break;

break;

break;

currentValue = 0;

operationSign = currentChar;

// Reset the current value

case '/':

std::stack<int> calculationStack;

char currentChar = s[i];

case '+':

case '-':

case '\*':

char operationSign = '+';

// Initialize the current value and the total size of the string

// If we reach the end of the string or encounter an operation

calculationStack.push(currentValue);

calculationStack.push(-currentValue);

calculationStack.pop();

calculationStack.pop();

// Update the operation sign for the next operation

int topValue = calculationStack.top();

int topValue = calculationStack.top();

calculationStack.push(topValue \* currentValue);

calculationStack.push(topValue / currentValue);

// If the sign is '+', push the current value to the stack

int currentValue = 0, stringSize = s.size();

class Solution {

public:

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

2 #include <stack>

```
18
                // If we've reached the end of the string or we encounter an operator
19
                if (i == s.length() - 1 || currentChar == '+' || currentChar == '-'
20
                                          || currentChar == '*' || currentChar == '/') {
21
22
23
                    // Perform the operation based on the previous sign
24
                    switch (operation) {
                        case '+':
25
26
                            stack.push(value); // Add the value to the stack
27
                            break;
28
                        case '-':
29
                            stack.push(-value); // Push the negated value for subtraction
30
                            break;
                        case '*':
31
                            stack.push(stack.pop() * value); // Multiply with the top value on the stack
32
                            break;
33
                        case '/':
                            stack.push(stack.pop() / value); // Divide the top value with current value
                            break;
38
39
                    operation = currentChar; // Update the operation for the next iteration
                    value = 0; // Reset value for the next number
40
41
42
43
44
            int result = 0; // Initialize result to accumulate stack values
45
           // Pop values from the stack and add them to calculate the result
46
           while (!stack.isEmpty()) {
47
                result += stack.pop();
48
49
50
            return result; // Return the final calculated result
51
52
53
```

// If character is a digit, build the number by multiplying the current value by 10 and adding the digit's integer valu

// If the sign is '\*', multiply the top element of the stack with the current value and push the result back to

// If the sign is '/', divide the top element of the stack by the current value and push the result back to the

if (i == stringSize - 1 || currentChar == '+' || currentChar == '-' || currentChar == '\*' || currentChar == '/') {

if (std::isdigit(currentChar)) currentValue = currentValue \* 10 + (currentChar - '0');

// If the sign is '-', push the negative of the current value to the stack

```
50
 51
 52
             // Initialize the answer to 0
 53
             int result = 0;
 54
             // Add up all values in the stack to get the final result
             while (!calculationStack.empty()) {
 55
 56
                 result += calculationStack.top();
                 calculationStack.pop();
 57
 58
 59
             // Return the result of expression evaluation
 60
             return result;
 61
 62 };
 63
Typescript Solution
   // Import Stack data structure for use in calculation
  2 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
  8
         let expressionSize: number = expression.length;
  9
 10
         // Start with a default operationSign of '+'
         let operationSign: string = '+';
 11
 12
         // Use a stack to keep track of calculation results
 13
         let calculationStack = new Stack<number>();
 14
         for (let i = 0; i < expressionSize; ++i) {</pre>
 15
 16
             let currentChar = expression[i];
 17
             // If the current character is a digit, construct the number
 18
             if (!isNaN(parseInt(currentChar))) {
 19
                 currentValue = currentValue * 10 + (parseInt(currentChar));
 20
 21
 22
 23
             // Perform operations at the end of the string or at an operator
             if (i === expressionSize - 1 || currentChar === '+' || currentChar === '-' || currentChar === '*' || currentChar === '/') {
 24
                 switch (operationSign) {
 25
 26
                     // Push current value to stack for a '+' operation
                     case '+':
 27
                         calculationStack.push(currentValue);
 28
 29
                         break;
 30
                     // Push the negative of current value for a '-' operation
 31
                     case '-':
 32
                         calculationStack.push(-currentValue);
 33
                         break;
                     // Multiply the stack's top with current value for a '*' operation
 34
 35
                     case '*':
 36
 37
                             let topValue = calculationStack.pop();
                             if(topValue !== undefined) {
 38
 39
                                 calculationStack.push(topValue * currentValue);
 40
 41
 42
                         break;
 43
                     // Divide the stack's top by current value for a '/' operation
                     case '/':
 44
 45
 46
                             let topValue = calculationStack.pop();
 47
                             if(topValue !== undefined) {
 48
                                 calculationStack.push(Math.trunc(topValue / currentValue));
 49
 50
 51
                         break;
 52
 53
                 // Update the operationSign with current character
 54
                 operationSign = currentChar;
 55
                 // Reset currentValue for the next part of the string
 56
                 currentValue = 0;
 57
 58
 59
 60
         // Calculate the final result by summing up the values in the stack
 61
         let result: number = 0;
 62
         while (!calculationStack.isEmpty()) {
 63
             let value = calculationStack.pop();
 64
             if(value !== undefined) {
                 result += value;
 65
```

#### 69 // Return the result of the expression evaluation 70 return result; 71 72 // Note: Since this bit of code uses the 'Stack' implementation from the 'typescript-collections' library,

Time and Space Complexity

// you would have to install that npm package to use this stack data structure.

67

68

75

**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.