<https://leetcode.ca/all/772.html>

<https://www.lintcode.com/problem/849/description>

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, non-negative integers and empty spaces .

The expression string contains only non-negative integers, +, -, \*, / operators , open ( and closing parentheses ) and empty spaces . The integer division should truncate toward zero.

You may assume that **the given expression is always valid**. All intermediate results will be in the range of [-2147483648, 2147483647].

Some examples:

"1 + 1" = 2

" 6-4 / 2 " = 4

"2\*(5+5\*2)/3+(6/2+8)" = 21

"(2+6\* 3+5- (3\*14/7+2)\*5)+3"=-12

Note: Do not use the eval built-in library function.

**L772的解法可以同时作用于L224, L227, 属于通用解法**

**Attempt 1: 2024-07-02**

**Solution 1: Two Stacks (720 min)**

public class Solution {

    /\*\*

     \* @param s: the expression string

     \* @return: the answer

     \*/

    public int calculate(String s) {

        Stack<Integer> nums = new Stack<>();

        Stack<Character> ops = new Stack<>();

        int n = s.length();

        for(int i = 0; i < n; i++) {

            char c = s.charAt(i);

            if(c == ' ') {

                continue;

            }

            if(Character.isDigit(c)) {

                int num = 0;

                while(i < n && Character.isDigit(s.charAt(i))) {

                    num = num \* 10 + s.charAt(i) - '0';

                    i++;

                }

                nums.push(num);

                i--;

            } else if(c == '(') {

                ops.push(c);

            } else if(c == ')') {

                while(ops.peek() != '(') {

                    int b = nums.pop();

                    int a = nums.pop();

                    nums.push(apply\_ops(ops.pop(), a, b));

                }

                ops.pop();

            } else {

                // Check if previous operator not '(' and higher precedence than current operator,

                // then we have to calculate based on previous operator first.

                // e.g 2\*3+4, we will see '\*' as previous operator at stack peek when we encounter '+',

                // so we have to calculate based on '\*' first by pop out 2 and 3 to get multiple as 6,

                // rather than calcualte '+' with 4

                while(!ops.isEmpty() && ops.peek() != '('

                && precedence(ops.peek()) >= precedence(c)) {

                    int b = nums.pop();

                    int a = nums.pop();

                    nums.push(apply\_ops(ops.pop(), a, b));

                }

                ops.push(c);

            }

        }

        while(!ops.isEmpty()) {

            int b = nums.pop();

            int a = nums.pop();

            nums.push(apply\_ops(ops.pop(), a, b));

        }

        return nums.peek();

    }

    private int precedence(char op) {

        if(op == '(' || op == ')') {

            return 0;

        } else if(op == '+' || op == '-') {

            return 1; // '+' and '-' has lower precedence as 1

        } else {

            return 2; // '\*' and '/' has higher precedence as 2

        }

    }

    private int apply\_ops(char op, int a, int b) {

        if(op == '+') {

            return a + b;

        } else if(op == '-') {

            return a - b;

        } else if(op == '\*') {

            return a \* b;

        } else {

            return a / b;

        }

    }

}

**Refer to**

<https://algo.monster/liteproblems/772>

**Problem Explanation**

You are required to implement a basic calculator that evaluates simple expression strings. The expression string may contain open and closing parentheses, plus and minus signs, non-negative integers, multiplication and division operators, and empty spaces. For integer division, you should truncate towards zero. The expression is always valid and all intermediate results will be within the range of [-2147483648, 2147483647].

Here are some examples:

"1 + 1" should output: 2

" 6-4 / 2 " should output: 4

"2\*(5+5\*2)/3+(6/2+8)" should output: 21

"(2+6\* 3+5- (3\*14/7+2)\*5)+3" should output: -12

We are not allowed to use the eval built-in library function for solving this problem.

**Solution Approach**

In this solution, a stack data structure is used to store operands and operators until they are calculated. The algorithm goes through the string expression and breaks it down into numbers and operators.

Create two stacks, one for numbers and the other for operators.

Loop through the string.

If the current character is a number, calculate the number and push it on the numbers stack.

If the current character is an operator or a parenthesis, do the following:

If it's an opening parenthesis, push it onto the operators stack.

If it's a closing parenthesis, perform calculations until an opening parenthesis is found and then pop the opening parenthesis from the stack.

If it's an operator '+', '-', '\*', '/', check the precedence of the operator in stack with the current operator. If the operator in the stack has higher or equal precedence, perform calculation and then push the current operator in the stack.

When the string is exhausted, calculate the remaining expressions in the stack.

The top of the numbers stack will hold the final result.

**Sample Step**

Let's take the example of "6-4 /2".

Initially, both stacks are empty.

The first character '6' is a number, so push it in the number stack. number stack: [6].

The next character '-' is an operator, so push it in the operator stack. operator stack: ['-'].

The next characters '4', '/', and '2' similarly end up in the stack. number stack: [6, 4, 2], operator stack: ['-', '/'].

We have finished parsing the string, pop items out of the stack and calculate. The '/' has higher precedence than '-', so pop out 4 and 2, calculate 4/2 with integer division and get 2, then push 2 back to number stack: [6,2]. The operator stack is now ['-'].

Continue to calculate, pop out 6 and 2 from number stack, and '-' from operator stack, calculate 6 - 2 = 4. Now both stacks are empty and we return 4.

**Java Solution**

class Solution {

    /\*\*

     \* @param s: the expression string

     \* @return: the answer

     \*/

    private int precedence(char op) {

        if (op == '(' || op == ')') return 0;

        else if (op == '+' || op == '-') return 1; // '+' and '-' has lower precedence as 1

        else return 2; // '\*' and '/' has higher precedence as 2

    }

    private int apply\_operand(char op, int a, int b) {

        switch (op) {

            case '+':

                return a + b;

            case '-':

                return a - b;

            case '\*':

                return a \* b;

            default:

                return a / b;

        }

    }

    public int calculate(String s) {

        Stack<Integer> nums = new Stack<>();

        Stack<Character> ops = new Stack<>();

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

            if (s.charAt(i) == ' ') continue;

            if (Character.isDigit(s.charAt(i))) {

                int num = 0;

                while (i < s.length() && Character.isDigit(s.charAt(i))) {

                    num = num \* 10 + s.charAt(i) - '0';

                    i++;

                }

                nums.push(num);

                i--;

            }

            else if (s.charAt(i) == '(') {

                ops.push(s.charAt(i));

            }

            else if (s.charAt(i) == ')') {

                while (ops.peek() != '(') {

                    int b = nums.pop();

                    int a = nums.pop();

                    nums.push(apply\_operand(ops.pop(), a, b));

                }

// Pop out corresponding open bracket '(' for current close bracket ')'

                ops.pop();

            } else {

// Check if previous operator not '(' and higher precedence than current operator,

// then we have to calculate based on previous operator first.

// e.g 2\*3+4, we will see '\*' as previous operator at stack peek when we encounter '+',

// so we have to calculate based on '\*' first by pop out 2 and 3 to get multiple as 6,

// rather than calcualte '+' with 4

                while (!ops.empty() && ops.peek() != '('

&& this.precedence(ops.peek()) >= this.precedence(s.charAt(i))) {

                    int b = nums.pop();

                    int a = nums.pop();

                    nums.push(this.apply\_operand(ops.pop(), a, b));

                }

                ops.push(s.charAt(i));

            }

        }

        while (!ops.empty()) {

            int b = nums.pop();

            int a = nums.pop();

            nums.push(this.apply\_operand(ops.pop(), a, b));

        }

        return nums.peek();

    }

}

**Follow up: If input contains negative numbers, how to handle ?**

**Test out by input = "1-( -2)" against above original solution**

java.util.EmptyStackException

at line 103, java.base/java.util.Stack.peek

at line 85, java.base/java.util.Stack.pop

at line 63, Solution.calculate

at line 56, \_\_DriverSolution\_\_.\_\_helper\_\_

at line 86, \_\_Driver\_\_.main

**We have to add negative case handling, when detect the '-' as first char of the input or the first char after '(', the handling logic will invoke as "push a 0 onto 'num' stack" to simulate the action '0 - n = -n' to get negative value**

public class Solution {

    /\*\*

     \* @param s: the expression string

     \* @return: the answer

     \*/

    public int calculate(String s) {

        Stack<Integer> nums = new Stack<>();

        Stack<Character> ops = new Stack<>();

        int n = s.length();

        for(int i = 0; i < n; i++) {

            char c = s.charAt(i);

            if(c == ' ') {

                continue;

            }

            if(Character.isDigit(c)) {

                int num = 0;

                while(i < n && Character.isDigit(s.charAt(i))) {

                    num = num \* 10 + s.charAt(i) - '0';

                    i++;

                }

                nums.push(num);

                i--;

            } else if(c == '(') {

                ops.push(c);

            } else if(c == ')') {

                while(ops.peek() != '(') {

                    int b = nums.pop();

                    int a = nums.pop();

                    nums.push(apply\_ops(ops.pop(), a, b));

                }

                ops.pop();

            } else {

                // Check if previous operator not '(' and higher precedence than current operator,

                // then we have to calculate based on previous operator first.

                // e.g 2\*3+4, we will see '\*' as previous operator at stack peek when we encounter '+',

                // so we have to calculate based on '\*' first by pop out 2 and 3 to get multiple as 6,

                // rather than calcualte '+' with 4

                while(!ops.isEmpty() && ops.peek() != '('

                && precedence(ops.peek()) >= precedence(c)) {

                    int b = nums.pop();

                    int a = nums.pop();

                    nums.push(apply\_ops(ops.pop(), a, b));

                }

                // Negative scenario

// Test out by input = "1-( -2)" against above original solution

                if(c == '-') {

                    // The first num is negative

                    if(nums.isEmpty()) {

                        nums.push(0);

                    // The first num in parentheses is negative

                    } else {

                        int index = i - 1;

                        while(index >= 0 && s.charAt(index) == ' ') {

                            index--;

                        }

                        if(s.charAt(index) == '(') {

                            nums.push(0);

                        }

                    }

                }

                ops.push(c);

            }

        }

        while(!ops.isEmpty()) {

            int b = nums.pop();

            int a = nums.pop();

            nums.push(apply\_ops(ops.pop(), a, b));

        }

        return nums.peek();

    }

    private int precedence(char op) {

        if(op == '(' || op == ')') {

            return 0;

        } else if(op == '+' || op == '-') {

            return 1; // '+' and '-' has lower precedence as 1

        } else {

            return 2; // '\*' and '/' has higher precedence as 2

        }

    }

    private int apply\_ops(char op, int a, int b) {

        if(op == '+') {

            return a + b;

        } else if(op == '-') {

            return a - b;

        } else if(op == '\*') {

            return a \* b;

        } else {

            return a / b;

        }

    }

}

**Refer to**

<https://www.lintcode.com/problem/849/solution/17761>

精选的答案无法处理负数情况，需要添加负数的判断。

第一个数字就是负数

括号里的第一个数字是负数

这个代码是根据lc上的讨论 添加了一个判断。

class Solution {

public int calculate(String s) {

if (s == null || s.length() == 0) return 0;

Stack<Integer> nums = new Stack<>();

Stack<Character> ops = new Stack<>();

int num = 0;

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

char c = s.charAt(i);

if (c == ' '){

continue;

}

if (Character.isDigit(c)) {

num = c - '0';

while (i < s.length() - 1 && Character.isDigit(s.charAt(i + 1))) {

num = num \* 10 + (s.charAt(i+1) - '0');

i++;

}

nums.push(num);

num = 0;

} else if (c == '(') {

ops.push(c);

} else if (c == ')') {

while (ops.peek() != '('){

nums.push(operation(ops.pop(), nums.pop(), nums.pop()));

}

ops.pop();

} else if (c == '+' || c == '-' || c == '\*' || c == '/') {

while (!ops.isEmpty() && precedence(c, ops.peek())){

nums.push(operation(ops.pop(), nums.pop(),nums.pop()));

}

if(c == '-'){//negative scenario

if(nums.isEmpty()){ //the first num is negative

nums.push(0);

}else{// the first num in parenthess is negative

int index = i - 1;

while(index >= 0 && s.charAt(index) == ' '){

index--;

}

if(s.charAt(index) == '('){

nums.push(0);

}

}

}

ops.push(c);

}

}

while (!ops.isEmpty()) {

nums.push(operation(ops.pop(), nums.pop(), nums.pop()));

}

return nums.pop();

}

private boolean precedence(char ops1, char ops2){

if(ops2 == '(' || ops2 == ')'){

return false;

}

if((ops1 == '\*' || ops1 == '/') && (ops2 == '+' || ops2 == '-')){

return false;

}

return true;

}

private static int operation(char op, int b, int a) {

switch (op) {

case '+': return a + b; //加法

case '-': return a - b; //减法

case '\*': return a \* b; //乘法

case '/': return a / b; //除法

}

return 0;

}

}

**Refer to**

[L224.P11.7.Basic Calculator (Ref.L227,L772)](note://4489B9E4FD5E45449DC8776D72CFA9CE)

[L227.P11.7.Basic Calculator II (Ref.L224,L772)](note://49816B7E855C4C25BAEAEC863EE22738)