<https://leetcode.com/problems/basic-calculator/>

Given a string s representing a valid expression, implement a basic calculator to evaluate it, and return *the result of the evaluation*.

**Note:** You are **not** allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

**Example 1:**

Input: s = "1 + 1"

Output: 2

**Example 2:**

Input: s = " 2-1 + 2 "

Output: 3

**Example 3:**

Input: s = "(1+(4+5+2)-3)+(6+8)"

Output: 23

**Constraints:**

1 <= s.length <= 3 \* 10^5

s consists of digits, '+', '-', '(', ')', and ' '.

s represents a valid expression.

'+' is **not** used as a unary operation (i.e., "+1" and "+(2 + 3)" is invalid).

'-' could be used as a unary operation (i.e., "-1" and "-(2 + 3)" is valid).

There will be no two consecutive operators in the input.

Every number and running calculation will fit in a signed 32-bit integer.

**Attempt 1: 2022-11-26**

**Solution 1:  Stack**

**Style 1: Stack store both number and sign for current brace [当前所属括号的符号] (120 min)**

class Solution {

    public int calculate(String s) {

        int result = 0;

        int sign = 1;

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

        s = s.replaceAll(" ", "");

        int i = 0;

        while(i < s.length()) {

            char c = s.charAt(i);

            if(Character.isDigit(c)) {

                int num = c - '0';

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

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

                    i++;

                }

                result += num \* sign;

            } else if(c == '+') {

                sign = 1;

            } else if(c == '-') {

                sign = -1;

            } else if(c == '(') {

// result记录的是在遇到当前'('前的所有数加权和

                stack.push(result);

// sign记录的是在遇到当前'('前的最后一个符号

                stack.push(sign);

// 对于'('开始后新的result和sign我们需要重开一个计算

                result = 0;

                sign = 1;

            } else if(c == ')') {

// 当遇到')'时我们需要结算当前'(...)'中新运算出的result,

// 然后需要这个result与stack顶部记录的'('前最后的sign结合

// 才能获得带符号的当前'(...)'的结果, 然后需要与stack次顶

// 部记录的'('前的所有数加权和结合，这样才能完成更新result

// 来记录整体到当前')'的所有数加权和

                result = result \* stack.pop() + stack.pop();

            }

            i++;

        }

        return result;

    }

}

**Refer to**

<https://leetcode.com/problems/basic-calculator/discuss/62362/JAVA-Easy-Version-To-Understand>!!!!!

**Refer to**

<https://algo.monster/liteproblems/224>

**Problem Description**

The problem tasks us with creating a program that can evaluate a string expression containing integers, '+', '-', '(', and ')', representing the addition and subtraction of integers as well as grouping with parentheses. No built-in functions or libraries for evaluating mathematical expressions are allowed. The objective is to parse and calculate the result of the given string as the expression would be evaluated in arithmetic.

**Intuition**

To solve this challenge, we simulate the mechanics of an arithmetic expression evaluator by manually parsing the string. This involves the following steps:

**Linear Scanning:** We process the string from left to right, character by character, to evaluate the numbers and the operators.

**Dealing with Numbers:** Since the numbers may have more than one digit, we need to convert the sequence of digit characters into actual numerical values. This is done by initializing a number x to 0 and then for each digit 'd' we find, multiplying x by 10 and adding d to x, effectively constructing the number in decimal.

**Handling Operators:** We keep track of the last seen operator which will be either a '+' or a '-', through a sign variable which is 1 or -1, respectively. When a number is completely parsed, we combine it with the current sign to add or subtract from the accumulated answer.

**Using Stack for Parentheses: To evaluate expressions within parentheses, we use a stack. When we encounter an opening parenthesis '(', we push the current accumulated answer and the current sign onto the stack, then reset the answer and sign to begin evaluating the expression inside the parentheses. When we encounter a closing parenthesis ')', we pop the sign and the accumulated answer before the parenthesis from the stack, and combine them with the current answer inside the parentheses (apply the sign and then add the two numbers).**

**Final Computation:** After we process all characters, the result of the entire expression will be in the answer variable. Parentheses are handled during the process as they are encountered, ensuring that the subexpressions are calculated at the correct times.

This approach systematically breaks down the string into components that we can evaluate in isolation, handling the precedence of operations in expressions with parentheses appropriately, leading to the correct final answer.

**Solution Approach**

The solution's approach primarily involves a while loop that iterates through the string s, evaluating expressions and handling arithmetic operations and parentheses. The core components of the solution include:

**Stack (stk):** Used to store previous results and signs when an opening parenthesis is found. We push the current ans and sign and restart their values for evaluating the new embedded expression.

**Current answer (ans): This is the running total of the expression evaluated so far, excluding the content within parentheses which haven't been closed yet. When we find a closing parenthesis, we update it to include the result of expression inside the just-closed parenthesis.**

**Current sign (sign):** This variable holds the last seen sign of '+' or '-', initialized as 1 (representing '+'). This is used to multiply with the current number found to add or subtract it from ans.

**Index (i):** To keep track of the current position within the string.

The algorithm proceeds as follows:

Iterate over each character in the string until the end is reached.

If a digit is encountered:

A separate while loop gathers the entire number (as numbers could have multiple digits), constructing the value by multiplying the previously accumulated value by 10 and adding the digit to it.

After the whole number is determined, multiply it by the current sign and add it to the current ans.

Update the index i to the position after the last digit of the number.

When a '+' or '-' is encountered:

The sign is updated to 1 for '+' or -1 for '-'.

**On encountering an opening parenthesis '(':**

**Push the current ans and sign onto the stack (to 'remember' them for after the close parenthesis).**

**Reset ans and sign for the next calculation within the new context (inside the parentheses).**

**On encountering a closing parenthesis ')':**

**The expression inside the parenthesis is complete, so compute its value by multiplying it with the sign popped from the stack.**

**Then add the result to the ans before the parenthesis (also popped from the stack).**

The loop increments the index i at each step, except when processing a whole number, where i is updated accordingly.

After the while loop completes, ans contains the result of evaluating the entire expression, which is then returned.

**Example Walkthrough**

Let's use a simple expression to demonstrate the approach: 1 + (2 - (3 + 4)).

Initialize ans = 0, sign = 1, stk = [], and start at index i = 0.

i = 0: The first character is 1, a digit. So we form the number 1.

Update ans by adding 1 \* sign which is 1. Now ans = 1.

i = 2: The character is +. Set sign to 1.

i = 4: Encounter (. Push the current ans (1) and sign (1) onto stk and reset ans to 0, sign to 1.

i = 5: Next is digit 2. Update ans to 2.

i = 7: Encounter -. Set sign to -1.

i = 9: Encounter ( again. Push the current ans (2) and sign (-1) onto stk and reset ans to 0, sign to 1.

i = 10: Digit 3 is seen. Update ans to 3.

i = 12: The character is +. Set sign to 1.

i = 14: Digit 4 is seen. Update ans by adding 4 \* sign which is 4. Now ans = 3 + 4 = 7.

i = 15: Encounter ). Pop stk which has 2 and -1. Update ans: ans = 2 - (7) = -5.

i: Move forward to the next character, but there is none immediately after the closing parenthesis, so continue.

i = 17: Now, encounter another ). Pop stk which has 1 and 1. Update ans: ans = 1 + (-5) = -4.

At the end of the string, we have ans = -4. Since we've processed every character according to the rules, and handled the nested parentheses correctly, the resultant ans is the evaluation of the entire expression 1 + (2 - (3 + 4)), which is indeed -4.

**Solution Implementation**

class Solution {

public int calculate(String s) {

// Stack to hold the intermediate results and signs

Deque<Integer> stack = new ArrayDeque<>();

// Initialize the sign as positive

int sign = 1;

// This will hold the final result of the evaluation

int result = 0;

// Length of the input string for iteration

int length = s.length();

// Iterating over each character in the string

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

char ch = s.charAt(i);

// Check if the current char is a digit

if (Character.isDigit(ch)) {

int startIndex = i;

int number = 0;

// Build the number till we encounter a non-digit

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

number = number \* 10 + s.charAt(startIndex) - '0';

startIndex++;

}

// Update the result with the current number and sign

result += sign \* number;

// Move the pointer to the end of the number

i = startIndex - 1;

} else if (ch == '+') {

// Set sign as positive for addition

sign = 1;

} else if (ch == '-') {

// Set sign as negative for subtraction

sign = -1;

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

// Push the result and sign on the stack before the parenthesis

stack.push(result);

stack.push(sign);

// Reset result and sign for the expression inside the parenthesis

result = 0;

sign = 1;

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

// After closing parenthesis, result is the evaluated value inside the parenthesis

// Pop the sign before the parenthesis and multiply it with the current result,

// then add the result before the parenthesis

result = stack.pop() \* result + stack.pop();

}

}

// Return the final result of the evaluation

return result;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the code is determined by the number of operations needed to parse and evaluate the expression.

Looping through each character in the input string s of length n accounts for O(n) because each character is considered exactly once.

Each digit in the string is processed and converted to an integer, which in the worst case, every character could be a digit, resulting in O(n) for the digit conversion process.

The operations related to stack (pushing and popping) occur in constant time O(1) for each operation, but in the worst case, the total time complexity for all such operations is proportional to the number of parentheses in the input string. Since parentheses pairs cannot exceed n/2, the complexity due to stack operations is also O(n).

Therefore, the overall time complexity of the algorithm is O(n).

**Space Complexity**

The space complexity of the code is determined by the amount of additional memory needed to store intermediate results and the call stack (if applicable).

The stack stk is used to store the result and sign at each level of parentheses, and in the worst case, where we have a pair of parentheses for every two characters (like "((...))"), the stack size could grow up to n/2. Therefore, the space complexity due to the stack is O(n/2), which simplifies to O(n).

Variables ans, sign, i, x, j, and n use a constant amount of space, contributing O(1).

Hence, the overall space complexity of the algorithm is O(n) (where n is the length of the string s).

**Style 2: Stack only store sign for current brace [当前所属括号的符号] (120 min)**

class Solution {

    public int calculate(String s) {

        int result = 0;

        int sign = 1;

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

        stack.push(sign);

        s = s.replaceAll(" ", "");

        int i = 0;

        while(i < s.length()) {

            char c = s.charAt(i);

            if(Character.isDigit(c)) {

                int num = c - '0';

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

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

                    i++;

                }

//result += num \* sign;

                result += num \* sign \* stack.peek();

            } else if(c == '+') {

                sign = 1;

            } else if(c == '-') {

                sign = -1;

            } else if(c == '(') {

                //stack.push(result);

//stack.push(sign);

                stack.push(sign \* stack.peek());

                //result = 0;

                sign = 1;

            } else if(c == ')') {

                //result = result \* stack.pop() + stack.pop();

                stack.pop();

            }

            i++;

        }

        return result;

    }

}

**Refer to**

<https://segmentfault.com/a/1190000003796804>

**栈法**

**复杂度**

时间 O(N) 空间 O(N)

**思路**

很多人将该题转换为后缀表达式后（逆波兰表达式）求解，其实不用那么复杂。题目条件说明只有加减法和括号，由于加减法是相同顺序的，我们大可以直接把所有数顺序计算。难点在于多了括号后如何处理正负号。我们想象一下如果没有括号这题该怎们做：因为只有加减号，我们可以用一个变量sign来记录上一次的符号是加还是减，这样把每次读到的数字乘以这个sign就可以加到总的结果中了。有了括号后，整个括号内的东西可一看成一个东西，这些括号内的东西都会受到括号所在区域内的正负号影响（比如括号前面是个负号，然后括号所属的括号前面也是个负号，那该括号的符号就是正号）。但是每多一个括号，都要记录下这个括号所属的正负号，而每当一个括号结束，我们还要知道出来以后所在的括号所属的正负号。根据这个性质，我们可以使用一个栈，来记录这些括号所属的正负号。这样我们每遇到一个数，都可以根据当前符号，和所属括号的符号，计算其真实值。

**注意**

先用String.replace()去掉所有的空格

**代码**

public class Solution {

    public int calculate(String s) {

        // 去掉所有空格

        s = s.replace(" ", "");

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

        // 先压入一个1进栈，可以理解为有个大括号在最外面

        stk.push(1);

        int i = 0, res = 0, sign = 1;

        while(i < s.length()){

            char c = s.charAt(i);

            // 遇到正号，将当前的符号变为正号

            if(c=='+'){

                sign = 1;

                i++;

            // 遇到负号，将当前的符号变为负号

            } else if(c=='-'){

                sign = -1;

                i++;

            // 遇到左括号，计算当前所属的符号，压入栈中

            // 计算方法是当前符号乘以当前所属括号的符号

            } else if(c=='('){

                stk.push(sign \* stk.peek());

                sign = 1;

                i++;

            // 遇到右括号，当前括号结束，[当前所属括号的符号]出栈

            } else if(c==')'){

                stk.pop();

                i++;

            // 遇到数字，计算其正负号并加入总结果中

            } else {

                int num = 0;

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

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

                    i++;

                }

                res += num \* sign \* stk.peek();

            }

        }

        return res;

    }

}

**Style 3: Two Stacks (General solution from L772)**

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**

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

[L772.Basic Calculator III (Ref.L224,L227)](note://WEB8b3c61c712e02a5dd91a90fb4ed6c3b8)