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

Given a string s which represents an expression, *evaluate this expression and return its value*.

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 [-231, 231 - 1].

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

**Example 1:**

Input: s = "3+2\*2"

Output: 7

**Example 2:**

Input: s = " 3/2 "

Output: 1

**Example 3:**

Input: s = " 3+5 / 2 "

Output: 5

**Constraints:**

* 1 <= s.length <= 3 \* 105
* s consists of integers and operators ('+', '-', '\*', '/') separated by some number of spaces.
* s represents **a valid expression**.
* All the integers in the expression are non-negative integers in the range [0, 231 - 1].
* The answer is **guaranteed** to fit in a **32-bit integer**.

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

**Wrong solution:  Not reserve last operation**

**Take sample input as "3+2\*2", the major difference is wrong solution still recognize the the same, the first operation is '+', then second operation is '\*', but that will causing after push 3 onto stack, then will apply 3 \*2 directly which is wrong, the correct solution treat sample input as "+3+2\*2", the first operation is '+', the second operation is also '+', and finally the third operation is '\*', so after push 3 and 2 both stack, it will apply 2\*2 with popping out the first 2, which is correct logic**

Input: "3+2\*2"

Expected: 7

OUtput: 6

class Solution {

public int calculate(String s) {

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

int curNum = 0;

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

char c = s.charAt(i);

if(Character.isDigit(c)) {

curNum = curNum \* 10 + c - '0';

}

if(!Character.isDigit(c) && c != ' ' || i == s.length() - 1) {

if(c == '+') {

stack.push(curNum);

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

stack.push(-curNum);

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

stack.push(stack.pop() \* curNum);

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

stack.push(stack.pop() / curNum);

}

curNum = 0;

}

}

int result = 0;

while(!stack.isEmpty()) {

result += stack.pop();

}

return result;

}

}

**Solution 1:  With Stack**

**Style 1: Stack with reserve 'last operation' exactly before pushing current number (120min, just remember the trick !!!)**

class Solution {

public int calculate(String s) {

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

int curNum = 0;

// Tricky point:

// Constraints: All the integers in the expression are non-negative integers in the range [0, 231 - 1].

// e.g "-3+2\*2" is invalid since -3 is negative

// Based on constraints, we can pre-define operation as '+' as last operation

// which transfer a input from "3+2\*2" into "+3+2\*2"

// 'last operation' exactly define as latest operation before current number

char lastOperation = '+';

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

char c = s.charAt(i);

if(Character.isDigit(c)) {

curNum = curNum \* 10 + c - '0';

}

// Must include 'i = s.length() - 1' as condition, even the last character is

// not operation, we have to push current number since no more operation to

// terminate the calculation loop

// Test case: "3+2\*2"

if(!Character.isDigit(c) && c != ' ' || i == s.length() - 1) {

if(lastOperation == '+') {

stack.push(curNum);

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

stack.push(-curNum);

} else if(lastOperation == '\*') {

stack.push(stack.pop() \* curNum);

} else if(lastOperation == '/') {

stack.push(stack.pop() / curNum);

}

curNum = 0;

lastOperation = c;

}

}

int result = 0;

while(!stack.isEmpty()) {

result += stack.pop();

}

return result;

}

}

Time Complexity: O(n), where n is the length of the string s. We iterate over the string s at most twice.

Space Complexity: O(n), where n is the length of the string s.

**Refer to**

<https://leetcode.com/problems/basic-calculator-ii/discuss/1645655/C%2B%2B-Intuitive-Solution-(W-explanation)-or-Stack>

**APPROACH :**

* To begin with, we add '+' to the s=input string (Reason will be understood later).
* We initilaize a few variables int ans=0(to store the final ans), char sign='+'(to store the previous operator), long long int curr=0 (to store the value computed until we encounter some operator) (The final ans is guaranteed to be an integer, but the values in between might exceed integer values).
* While traversing the string :

1. If we encounter an empty space, we just continue.

2. Else if we encounter a digit/num (The string might also contain more than 1 digit numbers, so until we find any operator, we'll keep on forming the multi digit number (num = num\*10 + s[i] ) ).

3. When we encounter a sign, we stop & check the previous operator :

\*\* If sign = '+' (i.e; the previous operator was **+**), we just push curr into the stack ('Cause this value needs to be added to ans).

\*\* If sign = '-', (i.e; the previous operator was **-**), we toggle curr's sign and push it into the stack ('Cause this value needs to be subtracted from ans).

\*\* If sign = '\*', (i.e; we encountered a **x** sign previously), then the curr value should be multiplied with it and then added to ans, so we pop the top of the stack, multiply it with the current value and push the result into the stack.

\*\* If sign = '/', i.e; we encountered a **/** previously, then the curr value should divide the previous value ans, so we pop the top of the stack, divide it with curr and push the result into the stack.

4. At the end, we encounter another '+' sign, this is to push the last curr value into the stack.

5. Finally we have obtained a stack of values all of which only need to be added to obtain the ans.

6. So pop the values from stack one by one, add them to ans, and return the ans

**Time Complexity :** O(n) - n = size of the input string

**Auxiliary Space :** O(n) - For the stack

**Code :**

class Solution {

public:

int calculate(string s) {

s += '+';

stack<int> stk;

long long int ans = 0, curr = 0;

char sign = '+'; //to store the previously encountered sign

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

if(isdigit(s[i])) curr = curr\*10 + (s[i]-'0'); //keep forming the number, until you encounter an operator

else if(s[i]=='+' || s[i]=='-' || s[i]=='\*' || s[i]=='/'){

if(sign == '+') stk.push(curr); //'Cause it has to added to the ans

else if(sign == '-') stk.push(curr\*(-1)); //'Cause it has to be subtracted from ans

else if(sign == '\*'){

int num = stk.top(); stk.pop(); //Pop the top of the stack

stk.push(num\*curr); //Multiply it with the current value & push the result into stack

}

else if(sign == '/'){

int num = stk.top();stk.pop();

stk.push(num/curr); //Divide it with curr value & push it into the stack

}

curr = 0;

sign = s[i];

}

}

while(stk.size()){

ans += stk.top(); stk.pop();

}

return ans;

}

};

**Refer to**

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

**Overview**

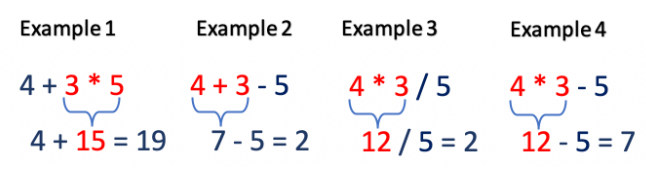
There are multiple variations of this problem like [Basic Calculator](https://leetcode.com/problems/basic-calculator/) and [Basic Calculator III](https://leetcode.com/problems/basic-calculator-iii/). This problem is relatively simpler to solve, as we don't have to take care of the parenthesis.

The aim is to evaluate the given mathematical expression by applying the basic mathematical rules. The expressions are evaluated from left to right and the order of evaluation depends on the [Operator Precedence](https://en.wikipedia.org/wiki/Order_of_operations). Let's understand how we could implement the problem using different approaches.

**Approach 1: Using Stack**

**Intuition**

We know that there could be 4 types of operations - addition (+), subtraction (-), multiplication (\*) and division (/). Without parenthesis, we know that, multiplication (\*) and (\) operations would always have higher precedence than addition (+) and subtraction (-) based on operator precedence rules.



If we look at the above examples, we can make the following observations -

* If the current operation is addition (+) or subtraction (-), then the expression is evaluated based on the precedence of the next operation.

In example 1, 4+3 is evaluated later because the next operation is multiplication (3\*5) which has higher precedence. But, in example 2, 4+3 is evaluated first because the next operation is subtraction (3-5) which has equal precedence.

* If the current operator is multiplication (\*) or division (/), then the expression is evaluated irrespective of the next operation. This is because in the given set of operations (+,-,\*,/), the \* and / operations have the highest precedence and therefore must be evaluated first.

In the above example 2 and 3, 4\*3 is always evaluated first irrespective of the next operation.

Using this intuition let's look at the algorithm to implement the problem.

**Algorithm**

Scan the input string s from left to right and evaluate the expressions based on the following rules

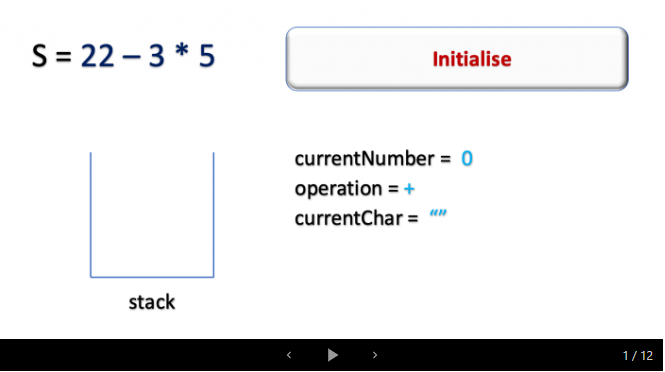
1. If the current character is a digit 0-9 ( operand ), add it to the number currentNumber.
2. Otherwise, the current character must be an operation (+,-,\*, /). Evaluate the expression based on the type of operation.

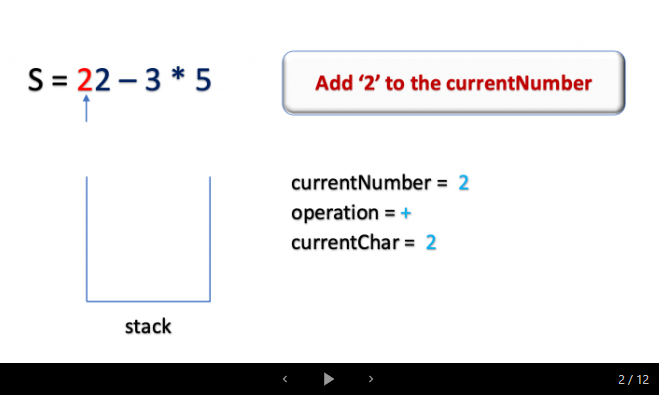
* Addition (+) or Subtraction (-): We must evaluate the expression later based on the next operation. So, we must store the currentNumber to be used later. Let's push the currentNumber in the Stack.

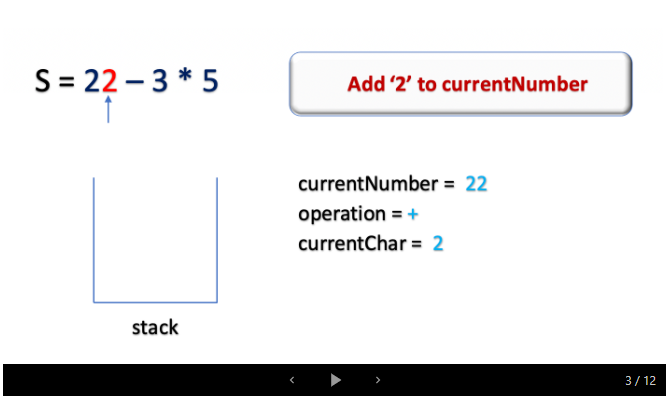
[Stack data structure](https://leetcode.com/explore/learn/card/queue-stack/230/usage-stack/) follows Last In First Out (LIFO) principle. Hence, the last pushed number in the stack would be popped out first for evaluation. In addition, when we pop from the stack and evaluate this expression in the future, we need a way to determine if the operation was Addition (+) or Subtraction (-). To simplify our evaluation, we can push -currentNumber in a stack if the current operation is subtraction (-) and assume that the operation for all the values in the stack is addition (+). This works because (a - currentNumber) is equivalent to (a + (-currentNumber)).

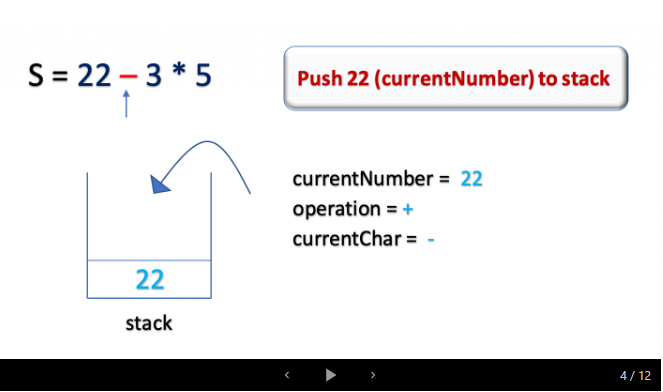
* Multiplication (\*) or Division (/): Pop the top values from the stack and evaluate the current expression. Push the evaluated value back to the stack.

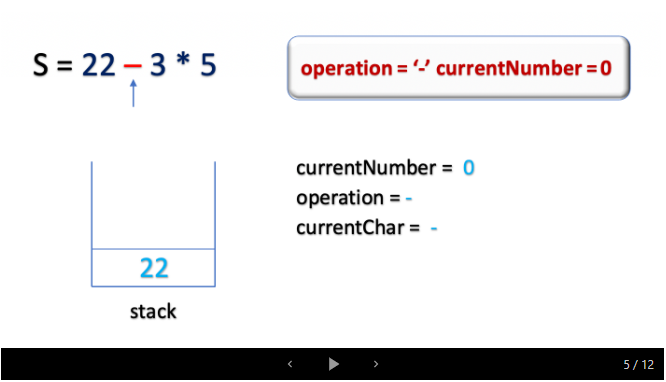
Once the string is scanned, pop from the stack and add to the result.

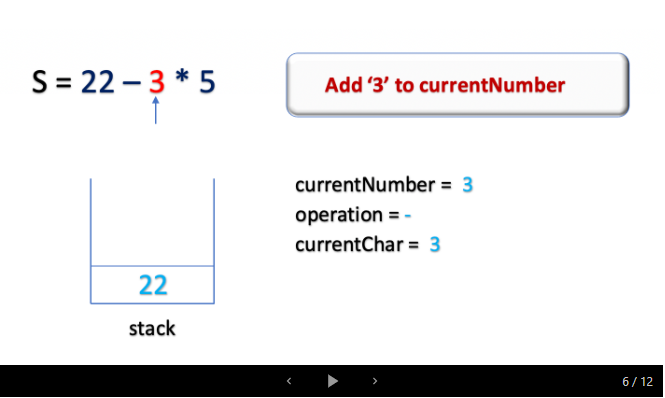


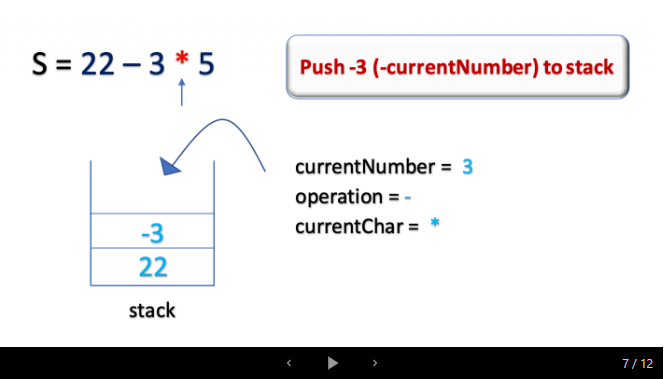


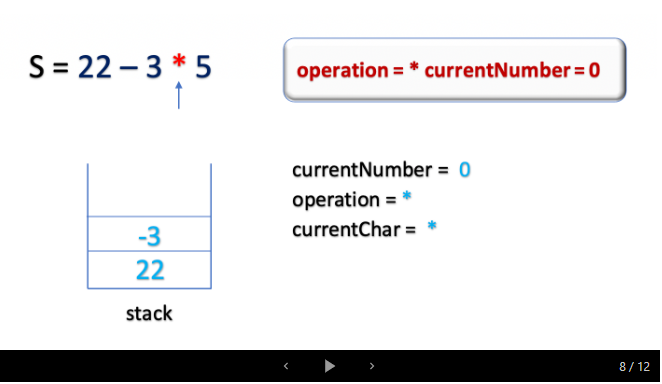


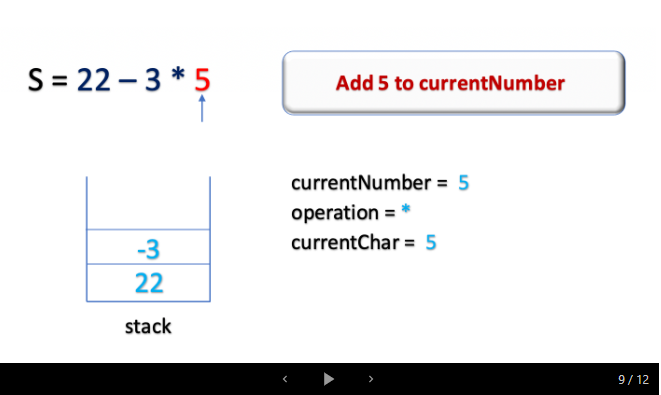


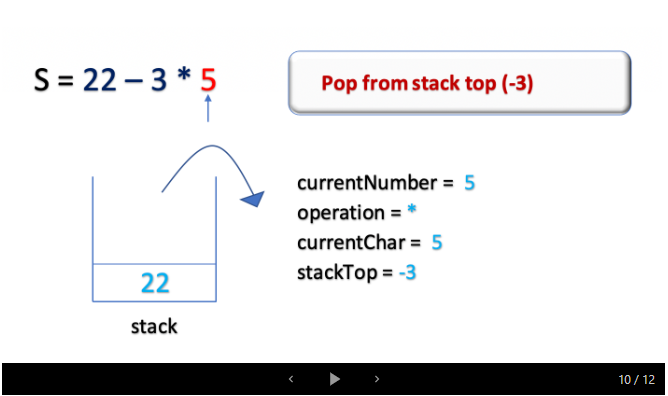


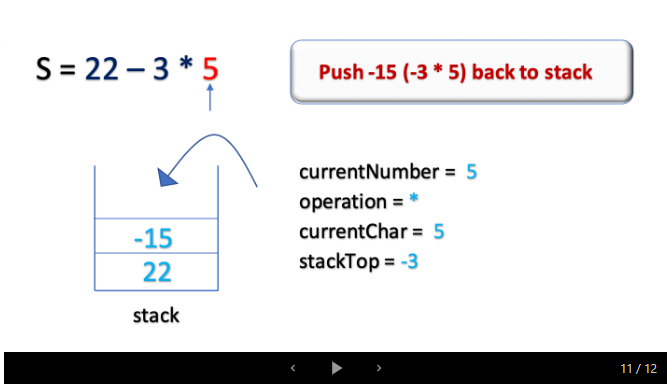


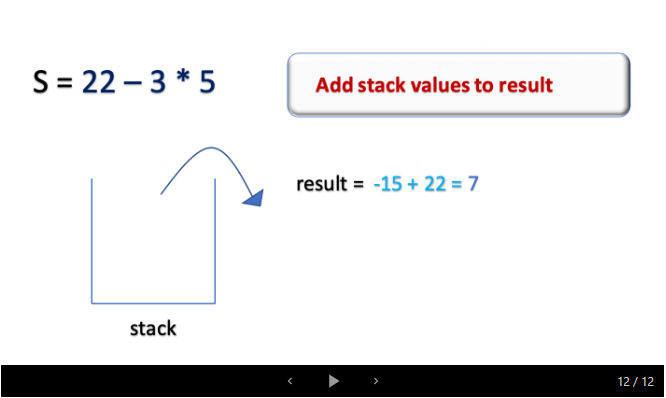












class Solution {

public int calculate(String s) {

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

int len = s.length();

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

int currentNumber = 0;

char operation = '+';

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

char currentChar = s.charAt(i);

if (Character.isDigit(currentChar)) {

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

}

if (!Character.isDigit(currentChar) && !Character.isWhitespace(currentChar) || i == len - 1) {

if (operation == '-') {

stack.push(-currentNumber);

}

else if (operation == '+') {

stack.push(currentNumber);

}

else if (operation == '\*') {

stack.push(stack.pop() \* currentNumber);

}

else if (operation == '/') {

stack.push(stack.pop() / currentNumber);

}

operation = currentChar;

currentNumber = 0;

}

}

int result = 0;

while (!stack.isEmpty()) {

result += stack.pop();

}

return result;

}

}

**Complexity Analysis**

* Time Complexity: O(*n*), where n is the length of the string s. We iterate over the string s at most twice.
* Space Complexity: O(*n*), where nis the length of the string s.

**Style 2:  Push first number onto Stack before any other operations (60 min, more intuitive)**

class Solution {

public int calculate(String s) {

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

s = s.replaceAll(" ", "");

String firstNum = getNum(0, s);

stack.push(Integer.parseInt(firstNum));

int i = firstNum.length();

while(i < s.length()) {

char c = s.charAt(i);

String nextNum = getNum(i + 1, s);

if(c == '+') {

stack.push(Integer.parseInt(nextNum));

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

stack.push(-Integer.parseInt(nextNum));

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

stack.push(stack.pop() \* Integer.parseInt(nextNum));

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

stack.push(stack.pop() / Integer.parseInt(nextNum));

}

i += nextNum.length() + 1;

}

int result = 0;

while(!stack.isEmpty()) {

result += stack.pop();

}

return result;

}

private String getNum(int index, String s) {

StringBuilder sb = new StringBuilder();

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

sb.append(s.charAt(index));

index++;

}

return sb.toString();

}

}

**Refer to**

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

**栈法**

**复杂度**

时间 O(N) 空间 O(N)

**思路**

因为乘法和除法不仅要知道下一个数，也要知道上一个数。所以我们用一个栈把上次的数存起来，遇到加减法就直接将数字压入栈中，遇到乘除法就把栈顶拿出来乘或除一下新数，再压回去。最后我们把栈里所有数加起来就行了。

**注意**

先用String.replace()去掉所有的空格

**代码**

public class Solution {

public int calculate(String s) {

s = s.replace(" ", "");

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

String firstNum = getNum(0, s);

stk.push(Long.parseLong(firstNum));

int i = firstNum.length();

while(i < s.length()){

char c = s.charAt(i);

// 拿出下一个数字

String numStr = getNum(i + 1, s);

if(c == '+'){

stk.push(Long.parseLong(numStr));

}

if(c == '-'){

stk.push(-Long.parseLong(numStr));

}

if(c == '\*'){

stk.push(stk.pop()\*Long.parseLong(numStr));

}

if(c == '/'){

stk.push(stk.pop()/Long.parseLong(numStr));

}

i = i+ numStr.length() + 1;

}

long res = 0;

while(!stk.isEmpty()){

res += stk.pop();

}

return (int)res;

}

private String getNum(int i, String s){

StringBuilder num = new StringBuilder();

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

num.append(s.charAt(i));

i++;

}

return num.toString();

}

}

**Solution 2:  Without Stack**

**Style 1: (120min)**

class Solution {

public int calculate(String s) {

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

int curNum = 0;

int lastNum = 0;

int result = 0;

// Tricky point:

// Constraints: All the integers in the expression are non-negative integers in the range [0, 231 - 1].

// e.g "-3+2\*2" is invalid since -3 is negative

// Based on constraints, we can pre-define operation as '+' as last operation

// which transfer a input from "3+2\*2" into "+3+2\*2"

// 'last operation' exactly define as latest operation before current number

char lastOperation = '+';

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

char c = s.charAt(i);

if(Character.isDigit(c)) {

curNum = curNum \* 10 + c - '0';

}

// Must include 'i = s.length() - 1' as condition, even the last character is

// not operation, we have to push current number since no more operation to

// terminate the calculation loop

// Test case: "3+2\*2"

if(!Character.isDigit(c) && c != ' ' || i == s.length() - 1) {

// The difference between '+-' and '\*/' for '+-':

// For '+-':

// When handle with stack we push 'curNum' onto stack,

// corresponding to when handle without stack we use 'result'

// together with 'lastNum' to simulate stack: push 'lastNum'

// onto 'result', then update 'lastNum' with 'curNum'

// For '\*/':

// When handle with stack we pop peek element from stack

// and update it with 'curNum' then push back to stack,

// corresponding to when handle without stack no need update

// 'result' since pop and push together means update 'lastNum'

// only, this would be added to the result after the entire

// string is scanned.

if(lastOperation == '+') {

//stack.push(curNum);

result += lastNum;

lastNum = curNum;

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

//stack.push(-curNum);

result += lastNum;

lastNum = -curNum;

} else if(lastOperation == '\*') {

//stack.push(stack.pop() \* curNum);

lastNum \*= curNum;

} else if(lastOperation == '/') {

//stack.push(stack.pop() / curNum);

lastNum /= curNum;

}

curNum = 0;

lastOperation = c;

}

}

//int result = 0;

//while(!stack.isEmpty()) {

// result += stack.pop();

//}

//return result;

result += lastNum;

return result;

}

}

**Refer to**

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

**Approach 2: Optimized Approach without the stack**

**Intuition**

In the previous approach, we used a stack to track the values of the evaluated expressions. In the end, we pop all the values from the stack and add to the result. Instead of that, we could add the values to the result beforehand and keep track of the last calculated number, thus eliminating the need for the stack. Let's understand the algorithm in detail.

**Algorithm**

The approach works similar to *Approach 1* with the following differences :

* Instead of using a stack, we use a variable lastNumber to track the value of the last evaluated expression.
* If the operation is Addition (+) or Subtraction (-), add the lastNumber to the result instead of pushing it to the stack. The currentNumber would be updated to lastNumber for the next iteration.
* If the operation is Multiplication (\*) or Division (/), we must evaluate the expression lastNumber \* currentNumber and update the lastNumber with the result of the expression. **This would be added to the result after the entire string is scanned.**

**Implementation**

class Solution {

public int calculate(String s) {

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

int length = s.length();

int currentNumber = 0, lastNumber = 0, result = 0;

char operation = '+';

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

char currentChar = s.charAt(i);

if (Character.isDigit(currentChar)) {

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

}

if (!Character.isDigit(currentChar) && !Character.isWhitespace(currentChar) || i == length - 1) {

if (operation == '+' || operation == '-') {

result += lastNumber;

lastNumber = (operation == '+') ? currentNumber : -currentNumber;

} else if (operation == '\*') {

lastNumber = lastNumber \* currentNumber;

} else if (operation == '/') {

lastNumber = lastNumber / currentNumber;

}

operation = currentChar;

currentNumber = 0;

}

}

result += lastNumber;

return result;

}

}

**Complexity Analysis**

* Time Complexity: O(*n*), where n is the length of the string s*s*.
* Space Complexity:  O(1), as we use constant extra space to store lastNumber, result and so on.

**Style 2: (60 min, more intuitive)**

class Solution {

public int calculate(String s) {

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

s = s.replaceAll(" ", "");

String firstNum = getNum(0, s);

//stack.push(Integer.parseInt(firstNum));

int result = 0;

int preNum = 0;

result = Integer.parseInt(firstNum);

preNum = result;

int i = firstNum.length();

while(i < s.length()) {

char c = s.charAt(i);

String nextNum = getNum(i + 1, s);

int n = Integer.parseInt(nextNum);

if(c == '+') {

//stack.push(Integer.parseInt(nextNum));

result += n;

preNum = n;

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

//stack.push(-Integer.parseInt(nextNum));

result -= n;

preNum = -n;

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

//stack.push(stack.pop() \* Integer.parseInt(nextNum));

result = result - preNum + preNum \* n;

preNum \*= n;

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

//stack.push(stack.pop() / Integer.parseInt(nextNum));

result = result - preNum + preNum / n;

preNum /= n;

}

i += nextNum.length() + 1;

}

//int result = 0;

//while(!stack.isEmpty()) {

// result += stack.pop();

//}

return result;

}

private String getNum(int index, String s) {

StringBuilder sb = new StringBuilder();

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

sb.append(s.charAt(index));

index++;

}

return sb.toString();

}

}

**Refer to**

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

**临时变量法**

**复杂度**

时间 O(N) 空间 O(1)

**思路**

这题很像[Expression Add Operator](http://segmentfault.com/a/1190000003797204)。因为没有括号，其实我们也可以不用栈。首先维护一个当前的结果，加减法的时候，直接把下一个数加上或减去就行了。乘除法的技巧在于，记录下上次的数字，这样我们把上次计算出的结果，减去上次的数字，得到了上上次的结果，就相当于回退到加或减上一个数字之前的情况了。这时候我们再把上一个数字乘上或除以当前的数字，最后再加或减回上上次的结果，就是这次的结果了。比如2+3\*4，当算完3时，结果是5，当算到4时，先用5-3=2，再用2+3\*4=14，就是当前结果。这里要注意的是，对于下一个数，它的上一个数不是我们这轮的数，而是我们这轮的上轮的数乘以或除以这轮的数，如2+3\*4\*5，到4的时候结果14，到5的时候，上一个数是3\*4，而不是4。

**注意**

要单独处理第一个数的情况

**代码**

public class Solution {

public int calculate(String s) {

s = s.replace(" ","");

long currRes = 0, prevNum = 0;

// 拿出第一个数

String firstNum = getNum(0, s);

currRes = Long.parseLong(firstNum);

prevNum = currRes;

int i = firstNum.length();

while(i < s.length()){

char c = s.charAt(i);

String numStr = getNum(i + 1, s);

System.out.println(numStr);

long n = Long.parseLong(numStr);

if(c == '+'){

currRes += n;

prevNum = n;

}

if(c == '-'){

currRes -= n;

prevNum = -n;

}

if(c == '\*'){

// 上次的结果，减去上次的数，再加上上次的数乘以这次的数，就是这次的结果

currRes = currRes - prevNum + prevNum \* n;

prevNum = prevNum \* n;

}

if(c == '/'){

// 上次的结果，减去上次的数，再加上上次的数除以这次的数，就是这次的结果

currRes = currRes - prevNum + prevNum / n;

prevNum = prevNum / n;

}

// 计算完后，跳过当前的运算符和数字

i = i + numStr.length() + 1;

}

return (int)currRes;

}

private String getNum(int i, String s){

StringBuilder num = new StringBuilder();

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

num.append(s.charAt(i));

i++;

}

return num.toString();

}

}