Basic Calculator III

Implement a basic calculator to evaluate a simple expression string.

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.

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.

Solution 1

So far, we have encountered the following series of calculator problems:

1. 224. Basic Calculator
2. 227. Basic Calculator II
3. 772. Basic Calculator III
4. 770. Basic Calculator IV

Though each of them may be solved using different methodologies, in this post I'd like to sort out the complexities and develop one "generic" solution to help us approach these problems.

Note that this post is **NOT** intended to deal with general calculator problems that involve complicated operands/operators. The analyses below are limited to the scope as specified by the problem descriptions of the above four problems.

`I -- Definitions and terminology`

In this section I will spell out some definitions to facilitate the explanation.

- `Expression`: An expression is a string whose value is to be calculated. Every expression must be alternating between **operand**s and **operator**s.
- `Operand`: An operand is a "generalized value" that can be the target of an operator. It must be one of the following three types -- **number**, **variable**, **subexpression**.
- `Number`: A number consists of digits only. The value of an operand of this type is given by the literal value of the number.
- `Variable`: A variable consists of lowercase letters only. It can be either a free variable whose value is unknown, or a bound variable whose value is mapped to some number (can be looked up). Here we define the value of an operand of free variable type is given by the variable itself, while that of bound variable type is given by the value of the number to which the variable is mapped.
- `Subexpression`: A subexpression is a valid expression enclosed in parentheses (which implies a recursive definition back to `Expression`). The value of an operand of this type is given by the calculated value of the subexpression itself.
- `Operator`: An operator is some prescribed action to be taken on the target operands. It must be one of the following four types: `+`, `-`, `*`, `/`, corresponding to addition, subtraction, multiplication and integer division, respectively. Operators have precedences, where `+` and `-` have **level one** precedence, while `*` and `/` have **level two** precedence (the higher the level is, the higher the precedence is).

`II -- Rules for calculations`

In this section, I will specify the general rules for carrying out the actual evaluations

of the expression.

Separation rule:

- We separate the calculations into two different levels corresponding to the two precedence levels.
- For each level of calculation, we maintain two pieces of information: the *partial result* and the *operator in effect*.
- For level one, the partial result starts from `0` and the initial operator in effect is `+`; for level two, the partial result starts from `1` and the initial operator in effect is `*`.
- We will use `l1` and `o1` to denote respectively the partial result and the operator in effect for level one; `l2` and `o2` for level two. The operators have the following mapping:
  `o1 == 1` means `+`; `o1 == -1` means `-`;
  `o2 == 1` means `*`; `o2 == -1` means `/`.
  By default we have `l1 = 0`, `o1 = 1`, and `l2 = 1`, `o2 = 1`.

Precedence rule:

- Each operand in the expression will be associated with a precedence of level two by default, meaning they can only take part in calculations of precedence level two, not level one.
- The operand can be any of the aforementioned types (number, variable or subexpression), and will be evaluated together with `l2` under the action prescribed by `o2`.

Demotion rule:

- The partial result `l2` of precedence level two can be demoted to level one. Upon demotion, `l2` becomes the operand for precedence level one and will be evaluated together with `l1` under the action prescribed by `o1`.
- The demotion happens when either a level one operator (i.e., `+` or `-`) is hit or the end of the expression is reached. After demotion, `l2` and `o2` will be reset for following calculations.

`III -- Algorithm implementations`

In this section, I will lay out the general structure of the algorithm using pseudo-codes. From section `I`, we know there are at most five different types of structs contained in the expression: number, variable, subexpression, level one operators, level two operators. We will check each of them and proceed accordingly.

```
public int calculate(String s) {
    int l1 = 0, o1 = 1; // Initialization of level one
    int l2 = 1, o2 = 1; // Initialization of level two

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

        if (c is a digit) {

           --> we have an operand of type number, so find its value "num"
           --> then evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c is a lowercase letter) {

           --> we have an operand of type variable, so find its name "var"
           --> then look up the variable mapping table to find its value "num";
           --> lastly evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c is an opening parenthesis) {

           --> we have an operand of type subexpression, so find its string represen
tation
           --> then recursively call the "calculate" function to find its value "num
";
           --> lastly evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c is a level two operator) {

            --> o2 needs to be updated: o2 = (c == '*' ? 1 : -1);

        } else if (c is a level one operator) {

            --> demotion happens here: l1 = l1 + o1 * l2;
            --> o1 needs to be updated: o1 = (c == '+' ? 1 : -1);
            --> l2, o2 need to be reset: l2 = 1, o2 = 1;

       }

       return (l1 + o1 * l2); // end of expression reached, so demotion happens agai
n
}
```

`IV -- List of solutions`

It is now straightforward to adapt the algorithm in section `III` to tackle each of the calculator problems. Note that not all the checks are needed for a particular version of the calculator problems. More specifically,

- Basic Calculator I does not have variables and level two operators;
- Basic Calculator II does not contain variables as well as subexpressions;
- Basic Calculator III does not have variables;
- Basic Calculator IV is the most general form but its level two operators do not

include division (`/`).

In this section, I will list two solutions for Basic Calculator III (recursive and iterative), and one solution for Basic Calculator IV (recursive).

Basic Calculator III: Solutions for this version pretty much follow the general structure in section `III`, except that we do not need to check for variables since the input expression does not contain any.

- Recursive solution: `O(n^2)` time, `O(n)` space

```
public int basicCalculatorIII(String s) {
    int l1 = 0, o1 = 1;
    int l2 = 1, o2 = 1;

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

            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '(') {
            int j = i;

            for (int cnt = 0; i < s.length(); i++) {
                if (s.charAt(i) == '(') cnt++;
                if (s.charAt(i) == ')') cnt--;
                if (cnt == 0) break;
            }

            int num = basicCalculatorIII(s.substring(j + 1, i));

            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '*' || c == '/') {
            o2 = (c == '*' ? 1 : -1);

        } else if (c == '+' || c == '-') {
            l1 = l1 + o1 * l2;
            o1 = (c == '+' ? 1 : -1);

            l2 = 1; o2 = 1;
        }
    }

    return (l1 + o1 * l2);
}
```

- Iterative solution: `O(n)` time, `O(n)` space

```java
public int basicCalculatorIII(String s) {
    int l1 = 0, o1 = 1;
    int l2 = 1, o2 = 1;

    Deque<Integer> stack = new ArrayDeque<>(); // stack to simulate recursion

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

            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '(') {
            // First preserve current calculation status
            stack.offerFirst(l1); stack.offerFirst(o1);
            stack.offerFirst(l2); stack.offerFirst(o2);

            // Then reset it for next calculation
            l1 = 0; o1 = 1;
            l2 = 1; o2 = 1;

        } else if (c == ')') {
            // First preserve the result of current calculation
            int num = l1 + o1 * l2;

            // Then restore previous calculation status
            o2 = stack.poll(); l2 = stack.poll();
            o1 = stack.poll(); l1 = stack.poll();

            // Previous calculation status is now in effect
            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '*' || c == '/') {
            o2 = (c == '*' ? 1 : -1);

        } else if (c == '+' || c == '-') {
            l1 = l1 + o1 * l2;
            o1 = (c == '+' ? 1 : -1);

            l2 = 1; o2 = 1;
        }
    }

    return (l1 + o1 * l2);
}
```

Basic Calculator IV: Solutions for this version, however, require some extra effort

apart from the general structure in section `III`. Due to the presence of variables (free variables, to be exact), the partial results for each level of calculations may not be pure numbers, but instead expressions (simplified, of course). So we have to come up with some structure to represent these partial results.

Though multiple options exist as to designing this structure, we do want it to support operations like addition, subtraction and multiplication with relative ease. Here I represent each expression as a collection of mappings from terms to coefficients, where each term is just a list of free variables sorted in lexicographic order. Some quick examples:

1. `"2 * a * b - d * c"`: this expression contains two terms, where the first term is `["a", "b"]` and the second is `["c", "d"]`. The former has a coefficient of `2` while the latter has a coefficient of `-1`. So we represent the expression as two mappings: `["a", "b"] --> 2` and `["c", "d"] --> -1`.
2. "`4`": this expression contains a single term, where the term has**zero** free variables and thus will be written as `[]`. The coefficient is `4` so we have the mapping `[] --> 4`. More generally, for any expression formed only by a pure number `num`, we have `[] --> num`.

See below for a detailed definition of the `Term` and `Expression` classes.

The `Term` class will support operations for comparing two terms according to their degrees as well as for generating a customized string representation.

The `Expression` class will support operations for adding additional terms to the existing mapping.

Addition of two expressions is done by adding all mappings in the second expression to those of the first one, and if possible, combine the coefficients of duplicate terms.

Subtraction of two expressions is implemented by first negating the coefficients of terms in the second expression and then applying addition (and thus can be combined with addition).

Multiplication of two expressions is done by collecting terms formed by merging every pair of terms from the two expressions (as well as their coefficients).

Lastly to conform to the format of the output, we have a dedicated function to convert the mappings in the result expression to a list of strings, where each string consists of the coefficient and the term connected by the `*` operator. Note that terms with `0` coefficient are ignored and terms without free variables contains numbers only.

As to complexity analysis, the nominal runtime complexity of this solution is similar to the recursive one for Basic Calculator III -- `O(n^2)`. It is also possible to use stacks to simulate the recursion process and cut the nominal time complexity down to `O(n)` (I will leave this as an exercise for you).

Here I used the word "nominal" because the above analyses assumed that the addition, subtraction and multiplication operations of two expressions take constant time, as is the case when the expressions are pure numbers. Apparently this assumption no longer stands true, since the number of terms may grow exponentially (think about this expression `(a + b) * (c + d) * (e + f) * ...`). Nevertheless, this solution should work against average/general test cases.

```
private static class Term implements Comparable<Term> {
    List<String> vars;
    static final Term C = new Term(Arrays.asList()); // this is the term for pure nu
mbers

    Term(List<String> vars) {
        this.vars = vars;
    }

    public int hashCode() {
        return vars.hashCode();
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;

        if (!(obj instanceof Term)) return false;

        Term other = (Term)obj;

        return this.vars.equals(other.vars);
    }

    public String toString() {
        return String.join("*", vars);
    }

    public int compareTo(Term other) {
        if (this.vars.size() != other.vars.size()) {
            return Integer.compare(other.vars.size(), this.vars.size());
        } else {
            for (int i = 0; i < this.vars.size(); i++) {
                int cmp = this.vars.get(i).compareTo(other.vars.get(i));
                if (cmp != 0) return cmp;
            }
        }

        return 0;
    }
}

private static class Expression {
    Map<Term, Integer> terms;

    Expression(Term term, int coeff) {
        terms = new HashMap<>();
        terms.put(term, coeff);
    }
```

```
    void addTerm(Term term, int coeff) {
        terms.put(term, coeff + terms.getOrDefault(term, 0));
    }
}

private Term merge(Term term1, Term term2) {
    List<String> vars = new ArrayList<>();

    vars.addAll(term1.vars);
    vars.addAll(term2.vars);
    Collections.sort(vars);

    return new Term(vars);
}

private Expression add(Expression expression1, Expression expression2, int sign) {
    for (Map.Entry<Term, Integer> e : expression2.terms.entrySet()) {
        expression1.addTerm(e.getKey(), sign * e.getValue());
    }

    return expression1;
}

private Expression mult(Expression expression1, Expression expression2) {
    Expression res = new Expression(Term.C, 0);

    for (Map.Entry<Term, Integer> e1 : expression1.terms.entrySet()) {
        for (Map.Entry<Term, Integer> e2 : expression2.terms.entrySet()) {
            res.addTerm(merge(e1.getKey(), e2.getKey()), e1.getValue() * e2.getValue
());
        }
    }

    return res;
}

private Expression calculate(String s, Map<String, Integer> map) {
    Expression l1 = new Expression(Term.C, 0);
    int o1 = 1;

    Expression l2 = new Expression(Term.C, 1);
    // we don't need 'o2' because the precedence level two operators contain '*' onl
y

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

        if (Character.isDigit(c)) {  // this is a number
            int num = c - '0';

            while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1))) {
                num = num * 10 + (s.charAt(++i) - '0');
            }

            l2 = mult(l2, new Expression(Term.C, num));
```

```
        } else if (Character.isLowerCase(c)) { // this is a variable
            int j = i;

            while (i + 1 < s.length() && Character.isLowerCase(s.charAt(i + 1))) i+
+;

            String var = s.substring(j, i + 1);
            Term term = map.containsKey(var) ? Term.C : new Term(Arrays.asList(var)
);
            int num = map.getOrDefault(var, 1);

            l2 = mult(l2, new Expression(term, num));

        } else if (c == '(') { // this is a subexpression
            int j = i;

            for (int cnt = 0; i < s.length(); i++) {
                if (s.charAt(i) == '(') cnt++;
                if (s.charAt(i) == ')') cnt--;
                if (cnt == 0) break;
            }

            l2 = mult(l2, calculate(s.substring(j + 1, i), map));

        } else if (c == '+' || c == '-') { // level one operators
            l1 = add(l1, l2, o1);
            o1 = (c == '+' ? 1 : -1);

            l2 = new Expression(Term.C, 1);
        }
    }

    return add(l1, l2, o1);
}

private List<String> format(Expression expression) {
    List<Term> terms = new ArrayList<>(expression.terms.keySet());

    Collections.sort(terms);

    List<String> res = new ArrayList<>(terms.size());

    for (Term term : terms) {
        int coeff = expression.terms.get(term);

        if (coeff == 0) continue;

        res.add(coeff + (term.equals(Term.C) ? "" : "*" + term.toString()));
    }

    return res;
}

public List<String> basicCalculatorIV(String expression, String[] evalvars, int[] ev
alints) {
    Map<String, Integer> map = new HashMap<>();
```

```
    for (int i = 0; i < evalvars.length; i++) {
        map.put(evalvars[i], evalints[i]);
    }

    return format(calculate(expression, map));
}
```

written by fun4LeetCode original link here

Solution 2

We just go through expression once, and for each character, and there are totally 5 cases
case1: digit:
case1.1: if operator now is '+', push digit
case1.2: if operator now is '-', push -digit
case1.3: if operator now is '', *push (poll*digit)
case1.4: if operator nwo is '/', push (poll/digit)
case2: space: do nothing
case3: operators: update operator
case4: (: push the operator into stack2, push '(' into stack1 (use +inf as '(')
case5:): continues poll and sum polled value until poll '(', then poll the operator from stack2, do calculate, then push back to stack1

Use stack1<Long> to store digit and '(', why use long? because we want to use Long.MAX_VALUE to represent special char '('
Use stack2 to store operator before '('

```java
class Solution {
    public int calculate(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }
        // initialize operator
        char sign = '+';
        Deque<Long> stack1 = new LinkedList<>(); // store digit and '('
        Deque<Character> stack2 = new LinkedList<>(); // store sign before '('
        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (Character.isDigit(ch)) {
                long num = 0;
                while (i < s.length() && Character.isDigit(s.charAt(i))) {
                    num = num * 10 + s.charAt(i++) - '0';
                }
                i--;
                stack1.offerFirst(eval(sign, stack1, num));
            } else if (ch == ' ') {
                continue;
            } else if (ch == '(') {
                stack1.offerFirst(Long.MAX_VALUE);
                stack2.offerFirst(sign);
                sign = '+';
            } else if (ch == ')') {
                long num = 0;
                while (stack1.peekFirst() != Long.MAX_VALUE) {
                    num += stack1.pollFirst();
                }
                stack1.pollFirst(); // pop out '(' (Long.MAX_VALUE)
                char operator = stack2.pollFirst();
                stack1.offerFirst(eval(operator, stack1, num));
            } else {
                sign = ch;
            }
        }
        // what we need to do is just sum up all num in stack
        int result = 0;
        while (!stack1.isEmpty()) {
            result += stack1.pollFirst();
        }
        return result;
    }
    private long eval(char sign, Deque<Long> stack1, long num) {
        if (sign == '+') {
            return num;
        } else if (sign == '-') {
            return -num;
        } else if (sign == '*') {
            return stack1.pollFirst() * num;
        } else {
            return stack1.pollFirst() / num;
        }
    }
}
```

written by Stephen_Li original link here

Solution 3

```
public static int calculate(String s) {
        if (s == null || s.length() == 0) return 0;
        Stack<Integer> nums = new Stack<>(); // the stack that stores numbers
        Stack<Character> ops = new Stack<>(); // the stack that stores operators (in
cluding parentheses)
        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';
                // iteratively calculate each number
                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; // reset the number to 0 before next calculation
            } else if (c == '(') {
                ops.push(c);
            } else if (c == ')') {
                // do the math when we encounter a ')' until '('
                while (ops.peek() != '(') nums.push(operation(ops.pop(), nums.pop()
, nums.pop()));
                ops.pop(); // get rid of '(' in the ops stack
            } else if (c == '+' || c == '-' || c == '*' || c == '/') {
                while (!ops.isEmpty() && precedence(c, ops.peek())) nums.push(opera
tion(ops.pop(), nums.pop(),nums.pop()));
                ops.push(c);
            }
        }
        while (!ops.isEmpty()) {
            nums.push(operation(ops.pop(), nums.pop(), nums.pop()));
        }
        return nums.pop();
    }

    private static int operation(char op, int b, int a) {
        switch (op) {
            case '+': return a + b;
            case '-': return a - b;
            case '*': return a * b;
            case '/': return a / b; // assume b is not 0
        }
        return 0;
    }
    // helper function to check precedence of current operator and the uppermost ope
rator in the ops stack
    private static boolean precedence(char op1, char op2) {
        if (op2 == '(' || op2 == ')') return false;
        if ((op1 == '*' || op1 == '/') && (op2 == '+' || op2 == '-')) return false;
        return true;
    }
```

credits to: https://www.geeksforgeeks.org/expression-evaluation/

written by floydchen original link here