Basic Calculator IV

Given an `expression` such as `expression = "e + 8 - a + 5"` and an evaluation map such as `{"e": 1}` (given in terms of `evalvars = ["e"]` and `evalints = [1]`), return a list of tokens representing the simplified expression, such as `["-1*a","14"]`

- An expression alternates chunks and symbols, with a space separating each chunk and symbol.
- A chunk is either an expression in parentheses, a variable, or a non-negative integer.
- A variable is a string of lowercase letters (not including digits.) Note that variables can be multiple letters, and note that variables never have a leading coefficient or unary operator like `"2x"` or `"-x"`.

Expressions are evaluated in the usual order: brackets first, then multiplication, then addition and subtraction. For example, `expression = "1 + 2 * 3"` has an answer of `["7"]`.

The format of the output is as follows:

- For each term of free variables with non-zero coefficient, we write the free variables within a term in sorted order lexicographically. For example, we would never write a term like `"b*a*c"`, only `"a*b*c"`.
- Terms have degree equal to the number of free variables being multiplied, counting multiplicity. (For example, `"a*a*b*c"` has degree 4.) We write the largest degree terms of our answer first, breaking ties by lexicographic order ignoring the leading coefficient of the term.
- The leading coefficient of the term is placed directly to the left with an asterisk separating it from the variables (if they exist.) A leading coefficient of 1 is still printed.
- An example of a well formatted answer is `["-2*a*a*a", "3*a*a*b", "3*b*b", "4*a", "5*c", "-6"]`
- Terms (including constant terms) with coefficient 0 are not included. For example, an expression of "0" has an output of [].

Examples:

```
Input: expression = "e + 8 - a + 5", evalvars = ["e"], evalints = [1]
Output: ["-1*a","14"]

Input: expression = "e - 8 + temperature - pressure",
evalvars = ["e", "temperature"], evalints = [1, 12]
Output: ["-1*pressure","5"]

Input: expression = "(e + 8) * (e - 8)", evalvars = [], evalints = []
Output: ["1*e*e","-64"]

Input: expression = "7 - 7", evalvars = [], evalints = []
Output: []

Input: expression = "a * b * c + b * a * c * 4", evalvars = [], evalints = []
Output: ["5*a*b*c"]

Input: expression = "((a - b) * (b - c) + (c - a)) * ((a - b) + (b - c) * (c - a))",
evalvars = [], evalints = []
Output: ["-1*a*a*b*b","2*a*a*b*c","-1*a*a*c*c","1*a*b*b*b","-1*a*b*b*c","-1*a*b*c*c",
"1*a*c*c*c","-1*b*b*b*c","2*b*b*c*c","-1*b*c*c*c","2*a*a*b","-2*a*a*c","-2*a*b*b","2*
a*c*c","1*b*b*b","-1*b*b*c","1*b*c*c","-1*c*c*c","-1*a*a","1*a*b","1*a*c","-1*b*c"]
```

Note:

1. `expression` will have length in range `[1, 1000]`.
2. `evalvars, evalints` will have equal lengths in range `[0, 1000]`.

Solution 1

The solution is similar to other calculator questions using stack. In C++, we can implicitly use stack in recursion, i.e. passing current index by reference.

I use unordered_map<string, int> to represent each operand, including both single variable or nested parentheses such as (a * b - a * c + 5).

```
For example, we can represent the following operands by unordered_map<string, int> m
p.
2*a*b: mp[a*b] = 2
15: mp[""] = 15
a*b*c - 5*e*f + 12*var*tmp: mp[a*b*c] = 1; mp[e*f] = -5; mp[var*tmp] = 12;
```

In addition, we need reorder the variable string after multiplication. For example, `a*b*c` is the same as `c*b*a`. And we should combine them.

Finally, sort each term by degrees and ignore those with zero coefficient.

```
class Solution {
public:
    vector<string> basicCalculatorIV(string expression, vector<string>& evalvars, v
ector<int>& evalints) {
        unordered_map<string, int> mp;
        int n = evalvars.size();
        // create a map for variable value pairs
        for (int i = 0; i < n; ++i) mp[evalvars[i]] = evalints[i];
        // helper function is recursion using implicit stack
        int pos = 0;
        unordered_map<string, int> output = helper(expression, mp, pos);
        vector<pair<string, int>> ans(output.begin(), output.end());
        // sort result based on variable degree
        sort(ans.begin(), ans.end(), mycompare);
        vector<string> res;
        for (auto& p: ans) {
            // only consider non-zero coefficient variables
            if (p.second == 0) continue;
            res.push_back(to_string(p.second));
            if (p.first != "") res.back() += "*"+p.first;
        }
        return res;
    }
private:
    unordered_map<string, int> helper(string& s, unordered_map<string, int>& mp, in
t& pos) {
        // every operand is an unordered_map, including single variable or nested (a
* b + a * c);
        // if the operand is a number, use pair("", number)
        vector<unordered_map<string, int>> operands;
        vector<char> ops;
        ops.push_back('+');
        int n = s.size();
        while (pos < n && s[pos] != ')') {
            if (s[pos] == '(') {
               pos++;
               operands.push_back(helper(s, mp, pos));
            }
```

```
            else {
                int k = pos;
                while (pos < n && s[pos] != ' ' && s[pos] != ')') pos++;
                string t = s.substr(k, pos-k);
                bool isNum = true;
                for (char c: t) {
                    if (!isdigit(c)) isNum = false;
                }
                unordered_map<string, int> tmp;
                if (isNum)
                    tmp[""] = stoi(t);
                else if (mp.count(t))
                    tmp[""] = mp[t];
                else
                    tmp[t] = 1;
                operands.push_back(tmp);
            }
            if (pos < n && s[pos] == ' ') {
                ops.push_back(s[++pos]);
                pos += 2;
            }
        }
        pos++;
        return calculate(operands, ops);
    }
    unordered_map<string, int> calculate(vector<unordered_map<string, int>>& operan
ds, vector<char>& ops) {
        unordered_map<string, int> ans;
        int n = ops.size();
        for (int i = n-1; i >= 0; --i) {
            unordered_map<string, int> tmp = operands[i];
            while (i >= 0 && ops[i] == '*')
                tmp = multi(tmp, operands[--i]);
            int sign = ops[i] == '+'? 1: -1;
            for (auto& p: tmp) ans[p.first] += sign*p.second;
        }
        return ans;
    }
    unordered_map<string, int> multi(unordered_map<string, int>& lhs, unordered_map
<string, int>& rhs) {
        unordered_map<string, int> ans;
        int m = lhs.size(), n = rhs.size();
        for (auto& p: lhs) {
            for (auto& q: rhs) {
                // combine and sort the product of variables
                string t = combine(p.first, q.first);
                ans[t] += p.second*q.second;
            }
        }
        return ans;
    }
    string combine(const string& a, const string& b) {
        if (a == "") return b;
        if (b == "") return a;
        vector<string> strs = split(a, '*');
        for (auto& s: split(b, '*')) strs.push_back(s);
        sort(strs.begin(), strs.end());
```

```
        sort(strs.begin(), strs.end());
        string s;
        for (auto& t: strs) s += t +'*';
        s.pop_back();
        return s;
    }
    static vector<string> split(const string& s, char c) {
        vector<string> ans;
        int i = 0, n = s.size();
        while (i < n) {
            int j = i;
            i = s.find(c, i);
            if (i == -1) i = n;
            ans.push_back(s.substr(j, i-j));
            i++;
        }
        return ans;
    }
    static bool mycompare(pair<string, int>& a, pair<string, int>& b) {
        string s1 = a.first, s2 = b.first;
        vector<string> left = split(s1, '*');
        vector<string> right = split(s2, '*');
        return left.size() > right.size() || (left.size() == right.size() && left <
right);
    }
};
```

written by zestypanda original link here

Solution 2

```
def basicCalculatorIV(self, expression, evalvars, evalints):
    class C(collections.Counter):
        def __add__(self, other):
            self.update(other)
            return self
        def __sub__(self, other):
            self.subtract(other)
            return self
        def __mul__(self, other):
            product = C()
            for x in self:
                for y in other:
                    xy = tuple(sorted(x + y))
                    product[xy] += self[x] * other[y]
            return product
    vals = dict(zip(evalvars, evalints))
    def f(s):
        s = str(vals.get(s, s))
        return C({(s,): 1}) if s.isalpha() else C({(): int(s)})
    c = eval(re.sub('(\w+)', r'f("\1")', expression))
    return ['*'.join((str(c[x]),) + x)
            for x in sorted(c, key=lambda x: (-len(x), x))
            if c[x]]
```

I let `eval` and `collections.Counter` do most of the work. First I wrap every variable and number in the given expression in a call to `f`. For example the expression `e + 8 - a + 5` becomes `f("e") + f("8") - f("a") + f("5")`. Then when I `eval` that, my function `f` converts its argument to a `C` object, which is a subclass of `Counter`.

A term like `42*a*a*b` is represented by `C({('a', 'a', 'b'): 42})`. That is, the key is the variables as sorted tuple, and the value is the coefficient. So `f` converts free variables to `C({('x',): 1})` (where `x` is the variable name) and converts known variables or numbers to `C({(): x})` (where `x` is the number).

Counters already know how to add and subtract each other, but I had to teach them multiplication. And in the end I need to turn the resulting `C` object into the desired output format.

written by StefanPochmann original link here

Solution 3

my code is following, and the testcase
112 / 122 test cases passed.
Status: Wrong Answer

Submitted: 2 minutes ago
Input: "(1 - 1 - l) - (8 * 5 - 1 - 2 * 0)"
["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "aa", "ab", "ac", "ad", "ae", "af", "ag", "ah", "ai", "aj", "ak", "al", "am", "an", "ao", "ap", "aq", "ar", "as", "at", "au", "av", "aw", "ax", "ay", "az", "ba", "bb", "bc", "bd", "be", "bf", "bg", "bh", "bi", "bj", "bk", "bl", "bm", "bn", "bo", "bp", "bq", "br", "bs", "bt", "bu", "bv", "bw", "bx", "by", "bz", "ca", "cb", "cc", "cd", "ce", "cf", "cg", "ch", "ci", "cj", "ck", "cl", "cm", "cn", "co", "cp", "cq", "cr", "cs", "ct", "cu", "cv"]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 5, 6, 7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4, 10, 11, 12, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Output: ["7057520"]
Expected: ["7057508"]

both expected and output are weired.....
on my computer result is {"50"}

```
namespace {
struct Formatter {
    bool operator()(const multiset<string>& a, const multiset<string>& b) const {
        if (a.size() > b.size()) return true;
        if (a.size() == b.size()) return a < b;
        return false;
    }
};

using Term = map<multiset<string>, int, Formatter>;

Term operator+(const Term& a, const Term& b) {
    Term res(a);
    for (auto& p : b) {
        res[p.first] += p.second;
    }
    return res;
}


Term operator-(const Term& a, const Term& b) {
    Term res(a);
    for (auto& p : b) {
        res[p.first] -= p.second;
    }
    return res;
}

Term operator*(const Term& a, const Term& b) {
```

```
Term operator*(const Term& a, const Term& b) {
    Term res;
    for (auto& pa : a) {
        for (auto& pb : b) {
            multiset<string> f(pa.first);
            f.insert(pb.first.begin(), pb.first.end());
            res[f] += pa.second * pb.second;
        }
    }
    return res;
}

Term calc(const Term& a, const Term& b, char op) {
    switch (op) {
    case '+':
        return a + b;
    case '-':
        return a - b;
    case '*':
        return a * b;
    default:
        __builtin_unreachable();
    }
}
}

class Solution {
public:
    vector<string> basicCalculatorIV(const string expression, const vector<string>&
evalvars, const vector<int>& evalints) {
        unordered_map<string, int> table;
        for (int i = 0; i < (int)evalvars.size(); ++i) {
            table.emplace(evalvars[i], evalints[i]);
        }

        stack<Term> ts;
        stack<char> os;
        int p = 0;
        while (true) {
            auto r = read(expression + "@", table, p);
            if (r.second == "@") {
                while (os.size()) {
                    Term t2 = move(ts.top());
                    ts.pop();
                    Term t1 = move(ts.top());
                    ts.pop();
                    char op = os.top();
                    os.pop();
                    ts.push(calc(t1, t2, op));
                }
                break;
            } else if (r.second == "(") {
                os.push('(');
            } else if (r.second == ")") {
                while (os.top() != '(') {
                    Term t2 = move(ts.top());
                    ts.pop();
                    Term t1 = move(ts.top());
```

```
                Term t1 = move(ts.top());
                ts.pop();
                char op = os.top();
                os.pop();
                ts.push(calc(t1, t2, op));
            }
            os.pop();
        } else if (r.second == "+") {
            if (os.empty() || os.top() == '(') os.push('+');
            else {
                Term t2 = move(ts.top());
                ts.pop();
                Term t1 = move(ts.top());
                ts.pop();
                char op = os.top();
                os.pop();
                ts.push(calc(t1, t2, op));
                os.push('+');
            }
        } else if (r.second == "-") {
            if (os.empty() || os.top() == '(') os.push('-');
            else {
                Term t2 = move(ts.top());
                ts.pop();
                Term t1 = move(ts.top());
                ts.pop();
                char op = os.top();
                os.pop();
                ts.push(calc(t1, t2, op));
                os.push('-');
            }
        } else if (r.second == "*") {
            if (os.empty() || os.top() != '*') os.push('*');
            else {
                Term t2 = move(ts.top());
                ts.pop();
                Term t1 = move(ts.top());
                ts.pop();
                char op = os.top();
                os.pop();
                ts.push(calc(t1, t2, op));
                os.push('*');
            }

        } else {
            ts.push(move(r.first));
        }
    }

    Term ft = move(ts.top());
    ts.pop();

    vector<string> res;
    for (auto& p : ft) {
        if (p.second == 0) continue;
        string s(to_string(p.second));
        for (auto& f : p.first) {
```

```
                s += "*" + f;
            }
            res.push_back(move(s));
        }
        return res;
    }

    pair<Term, string> read(const string& exp, const unordered_map<string, int>& ta
ble, int& p) {
        while (exp[p] == ' ') ++p;
        if (isalnum(exp[p])) {
            string res;
            while (isalnum(exp[p])) res.push_back(exp[p++]);
            if (isdigit(res.front())) return make_pair(Term{{{}, stol(res)}}, "");
            else if (table.count(res)) return make_pair(Term{{{}, table.at(res)}},
"");
            else return make_pair(Term{{{res}, 1}}, "");
        } else {
            return make_pair(Term(), string(1, exp[p++]));
        }
    }
};
```

written by oxFFFFFFFF original link here