770. Basic Calculator IV

Leetcode Link

Problem

You are given an algebraic expression in a string format and a map that gives the values to some of the variables in the expression. Your task is to simply the expression by substituting the variables with their values and performing the algebraic operations.

The expression only contains lowercase alphabet variables, integers, addition, subtraction, and multiplication operations, and parentheses. It's guaranteed that there are spaces between different parts of the expression. The result should be presented as a list of strings where each string is a term in the simplified expression.

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

Here, the value of variable e is given as 1. When substituted we get "1 + 8 - a + 5". This simplifies to "14 - a" which is represented as two terms in the result, "-1*a" and "14".

Approach

The key steps in the solution are as follows.

1. Tokenize the expression. This involves breaking the expression into smaller units such as variables, integers, and operations.
2. Convert the infix expression to postfix for easier evaluation. This is achieved using a Stack. The infix to postfix conversion is based on the precedence of the operators where multiplication comes before addition and subtraction.
3. Evaluate the postfix expression.
4. Simplify the terms and represent them in the required format.

Example

Let's see an example for a better understanding.

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

In this case, we have an expression with multiplication and addition operations. Since there are no evalvars, we aren't substituting any variables.

But the expression can still be simplified by combining like terms. The expression "a * b * c" appears twice. So, it can be combined to "5 * a * b * c".

The final output is ["5*a*b*c"].

C# Solution

```

C#
public class Solution
{
    public IList<string> BasicCalculatorIV(string expression, string[] evalvars, int[] evalints)
    {
        Dictionary<string, int> dic = new Dictionary<string, int>();
        for(int i = 0; i < evalvars.Length; i++)
        {
            dic.Add(evalvars[i], evalints[i]);
        }
        return Calculate(expression, dic, true);
    }
    private bool IsLowerCase(string s)
    {
        if(s[0] <= '9')
            return false;
        return true;
    }
    private IList<string> Calculate(string exp, Dictionary<string, int> dic, bool add)
    {
        List<string> ans = new List<string>();
        if(!exp.Contains("(") && !exp.Contains(")"))
        {
            string[] str = exp.Split(' ');
            for(int i = 0; i < str.Length; i += 2)
            {
                bool neg = str[i] == "-";
                bool isLower = IsLowerCase(str[i+1]);
                if(!add)
                    neg = !neg;
                if(!dic.ContainsKey(str[i+1]) || isLower)
                {
                    if(neg)
                        ans.Add("-" + str[i+1]);
                    else
                        ans.Add(str[i+1]);
                }
                else
                {
                    if(neg)
                        ans.Add("-" + dic[str[i+1]].ToString());
                    else
                        ans.Add(dic[str[i+1]].ToString());
                }
            }
            return ans;
        }
        int p = 0, start = 0, cnt = 0;
        for(int i = 0; i < exp.Length; i++)
        {
            if(exp[i] == '(')
            {
                if(cnt == 0)
                    start = i;
                cnt++;
            }
            else if(exp[i] == ')')
            {
                cnt--;
                if(cnt == 0)
                {
                    string temp = exp.Substring(0, p);
                    if(i + 1 < exp.Length)
                        temp += exp.Substring(i+1);
                    List<string> t1 = Calculate(temp, dic, add);
                    List<string> t2 = Calculate(exp.Substring(start + 1, i - start - 1), dic, add);
                    return Merge(t1, t2, exp[p-1] == '+');
                }
            }
            else if(exp[i] != ' ' && cnt == 0)
            {
                p = i+1;
            }
        }
        return new List<string>();
    }
    private IList<string> Merge(List<string> str1, List<string> str2, bool add)
    {
        List<string> ans = new List<string>();
        for(int i = 0; i < str1.Count; i++)
        {
            for(int j = 0; j < str2.Count; j++)
            {
                if(add)
                    ans.Add(str1[i] + "*" + str2[j]);
                else
                    ans.Add(str1[i] + "-" + str2[j]);
            }
        }
        return ans;
    }
}
```

Python Solution

```

python
class Solution:
    def basicCalculatorIV(self, expression: str, evalvars: List[str], evalints: List[int]) -> List[str]:
        count = collections.Counter(evalvars, evalints)
        def combine(left, right, symbol):
            if symbol == '-': right = [-x for x in right]
            return list(map(int.__add__, left, right))
        def multiply(left, right):
            return list(map(int.__mul__, left, right))

        def calculate(tokens):
            total = [0]
            symbols, stack = ['+'], []
            while tokens:
                token = tokens.popleft()
                if token in '+-':
                    while stack and symbols[-1] == '*':
                        total = multiply(stack.pop(), total)
                        symbols.pop()
                    total = combine(stack.pop() if stack else [0], total, symbols.pop())
                    symbols.append(token)
                elif token == '*':
                    symbols.append(token)
                elif token in count:
                    stack.append([count[token]])
                elif token.isdigit():
                    stack.append([int(token)])
                else:
                    stack.append([0, 1 if token.islower() else 0, int(token.isupper())])
            while stack:
                total = combine(stack.pop() if stack else [0], total, symbols.pop())
            return total
        return calculate(collections.deque(expression.split()))
```

Java Solution

```

java
package Calculator;

import java.util.*;

public class Solution {
    HashMap<String, Integer> map;
    PriorityQueue<String> q;

    public List<String> basicCalculatorIV(String expression, String[] evalvars, int[] evalints) {
        map = new HashMap<>();
        for (int i = 0; i < evalvars.length; i++) map.put(evalvars[i], evalints[i]);
        q = new PriorityQueue<>((a, b) -> (b.split("\\*").length - a.split("\\*").length != 0) ? b.split("\\*").length - a.split("\\
        Map<String, Integer> counts = count(expression);
        for (String c : counts.keySet())
            if (counts.get(c) != 0) {
                q.offer(c);
            }
        List<String> ans = new ArrayList();
        while (!q.isEmpty()) {
            String c = q.poll();
            ans.add(coefToString(counts.get(c)) + (c.equals("1") ? "" : "*" + c));
        }
        return ans;
    }

    public String coefToString(int x) {
        return x > 0 ? "+" + x : String.valueOf(x);
    }

    public Map<String, Integer> combine(Map<String, Integer> counter, int coef, Map<String, Integer> val) {
        Map<String, Integer> ans = new HashMap();
        for (String k : counter.keySet()) {
            for (String v : val.keySet()) {
                List<String> keys = new ArrayList<>(Arrays.asList((k + "*" + v).split("\\*")));
                Collections.sort(keys);
                String key = String.join("*", keys);
                ans.put(key, ans.getOrDefault(key, 0) + counter.get(k) * val.get(v) * coef);
            }
        }
        return ans;
    }

    public Map<String, Integer> count(String expr) {
        Map<String, Integer> ans = new HashMap();
        boolean prevNum = false;
        List<String> symbols = new ArrayList<>();
        symbols.add("+");
        List<Map<String, Integer>> stack = new ArrayList<>();
        int i = 0;
        while (i < expr.length()) {
            char c = expr.charAt(i);
            if (c == '(') {
                if (prevNum) {
                    stack.add(ans);
                    symbols.add("*");
                    ans = new HashMap();
                    prevNum = false;
                } else {
                    stack.add(new HashMap<>());
                    symbols.add(symbols.remove(symbols.size() - 1));
                }
                i++;
            } else if (c == ')') {
                Map<String, Integer> temp = ans;
                ans = stack.remove(stack.size() - 1);
                String symbol = symbols.remove(symbols.size() - 1);
                ans = combine(ans, symbol.equals("-") ? -1 : 1, temp);
                i++;
                prevNum = true;
            } else if (c == ' ') {
                i++;
            } else if (Character.isLetter(c)) {
                int j = i;
                while (j < expr.length() && Character.isLetter(expr.charAt(j))) j++;
                String var = expr.substring(i, j);
                i = j;
                if (map.containsKey(var)) {
                    ans.put("1", ans.getOrDefault("1", 0) + map.get(var) * (symbols.remove(symbols.size() - 1).equals("-") ? -1 : 1))
                    prevNum = true;
                } else {
                    ans = combine(ans, 1, new HashMap<String, Integer>() {{ put(var, symbols.remove(symbols.size() - 1).equals("-")
                    prevNum = false;
                }
            } else if (Character.isDigit(c)) {
                int j = i;
                while (j < expr.length() && Character.isDigit(expr.charAt(j))) j++;
                String num = expr.substring(i, j);
                i = j;
                ans.put("1", ans.getOrDefault("1", 0) + Integer.valueOf(num) * (symbols.remove(symbols.size() - 1).equals("-") ? -1 :
                prevNum = true;
            } else {
                symbols.add(c == '+' ? "+" : "-");
                prevNum = false;
                i++;
            }
        }
        return ans;
     }
}
```

Javascript Solution

```

javascript
var basicCalculatorIV = function(expression, evalvars, evalints) {
    let HashMap = require("collections/hash-map");
    let root = new Node();
    let i = 0;
    let evalMap = new HashMap();

    for (let i = 0; i < evalvars.length; i++) evalMap.set(evalvars[i], evalints[i]);

    while (i < expression.length) {
        if(expression.charAt(i) === ' ') {
            i++;
            continue;
        }
        if(expression.charAt(i) === "+" || expression.charAt(i) === "-") {
            new Node(root, expression.charAt(i++));
        } else if(expression.charAt(i) === "(") {
            let node = new Node(root, "+");
            root = node;
            i++;
        } else if(expression.charAt(i) === ")") {
            root = root.parent;
            i++;
        } else {
            let j = i;
            while(j<expression.length && expression.charAt(j)>='a' && expression.charAt(j)<='z') j++;
            let str = expression.substring(i,j);
            i=j;

            let num = 1;
            if(evalMap.has(str)) {
                num = evalMap.get(str);
                str = "";
            }

            if(i<expression.length && expression.charAt(i) === "*") {
                i++;
                j=i;
                while(j<expression.length && expression.charAt(j)>='0' && expression.charAt(j)<='9') j++;
                num *= parseInt(expression.substring(i,j));
                i=j;
            }
            root.children.get(root.children.size()-1).coe *= root.children.get(root.children.size()-1).term.getOrDefault(str,0)+num
            if(str !== "") root.children.get(root.children.size()-1).term.set(str, 1);
        }
    }

    let res = new Array(), map = new HashMap();
    calculate(root, map);

    let keys = Array.from(map.keys());

    keys.sort((a,b)=>(b.length() - a.length() !== 0) ? b.length() - a.length() : a.compareTo(b));

    for(let key of keys) {
        if(map.get(key) === 0) continue;
        if(key === "") {
            res.push(map.get(key) + "");
        } else {
            res.push(map.get(key) + "*" + key);
        }
    }
    return res;
};

calculate = function(root, map) {
    for(let node of root.children) {
        if(node.symbol === "*") {
            let childMap = new HashMap();
            calculate(node, childMap);
            let keys = Array.from(childMap.keys());
            for(let key of keys) {
                for(let term of key.split("\\*")) {
                    let val = node.term.getOrDefault(term,0) + childMap.get(key);
                    if(val === 0) node.term.delete(term);
                    else node.term.set(term,val);
                }
                let res = new Array();
                let keys = Array.from(node.term.keySet());
                for(let term of keys) {
                    for(let i=0;i<node.term.get(term);i++) res.add(term);
                }
                res.sort();
                map.set(String.join("*",res), node.coe);
            }
        } else if(node.symbol === "+") {
            let temp = calculate(node, map);
            for(let key of temp.keySet()) {
                let val = map.getOrDefault(key,0) + temp.get(key);
                if(val === 0) map.delete(key);
                else map.set(key,val);
            }
        } else {
            let res = new Array(), keys = Array.from(node.term.keySet());
            for(let term of keys) {
                for(let i=0;i<node.term.get(term);i++) res.push(term);
            }
            res.sort();
            return new HashMap().set(String.join("*",res), node.coe);
        }
    }
    return map;
};

class Node {
    constructor(parent, symbol) {
        this.parent = parent;
        if(parent !== undefined) {
            parent.children.push(this);
        }
        this.symbol = symbol;
        this.term = undefined;
        this.coe = undefined;

        if(symbol === "*" || symbol === undefined) {
            this.term = new Map();
            this.coe = 1;
        } else {
            this.coe = 0;
            this.term = null;
        }

        this.children = new Array();
        if(symbol !== undefined) this.children.push(new Node());
    }
};
```

As you can see from the solutions given above, the problem can be solved using a variety of programming languages including C#, Python, Java, and Javascript. Each of these solutions implements the same basic strategy – tokenize the expression, convert the infix expression to postfix for easier evaluation, evaluate the postfix expression and simplify the resulting terms.

In the C# solution, we first prepare a dictionary to map the variable values. Then the expression is divided into fragments based on whether parentheses are present or not. These fragments are further processed depending on whether they contain operations or variables. The Combine and Merge functions are used to combine or merge fragments after calculations have been done on them. The entire process is done recursively till the final simplified form is attained.

The Python solution also uses a similar approach but represents the expression as a deque collection for efficient retrieval and processing.

The Java solution follows a similar approach but employs Hashmaps to store variables and their corresponding integers.

Finally, the Javascript solution, which is slightly more complex, builds a tree-like structure to store and process the variables and operation. The tree nodes represent either an operation or an integer and are processed accordingly.

Despite the difference in syntax and certain functions, the core logic and the approach remains same across all these languages - breaking down the algebraic expression and evaluating it step by step.

Got a question? Ask the Teaching Assistant anything you don't understand.