Problem 770: Basic Calculator IV

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

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 a 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 degrees equal to the number of free variables being multiplied, counting multiplicity. We write the largest degree terms of our answer first, breaking ties by lexicographic order ignoring the leading coefficient of the term.

For example,

"a*a*b*c"

has degree

4

.

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

[]

.

Note:

You may assume that the given expression is always valid. All intermediate results will be in the range of

[-2

31

, 2

31

- 1]

.

Example 1:

Input:

expression = "e + 8 - a + 5", evalvars = ["e"], evalints = [1]

Output:

["-1*a","14"]

Example 2:

Input:

expression = "e - 8 + temperature - pressure", evalvars = ["e", "temperature"], evalints = [1, 12]

Output:

["-1*pressure","5"]

Example 3:

Input:

expression = "(e + 8) * (e - 8)", evalvars = [], evalints = []

Output:

["1*e*e","-64"]

Constraints:

1 <= expression.length <= 250

expression

consists of lowercase English letters, digits,

'+'

,

'-'

,

'*'

,

'('

,

')'

,

' '

.

expression

does not contain any leading or trailing spaces.

All the tokens in

expression

are separated by a single space.

0 <= evalvars.length <= 100

1 <= evalvars[i].length <= 20

evalvars[i]

consists of lowercase English letters.

evalints.length == evalvars.length

-100 <= evalints[i] <= 100

Code Snippets

C++:

```
class Solution {
public:
vector<string> basicCalculatorIV(string expression, vector<string>& evalvars,
vector<int>& evalints) {

}
};
```

Java:

```
class Solution {
public List<String> basicCalculatorIV(String expression, String[] evalvars,
int[] evalints) {

}
}
```

Python3:

```
class Solution:
def basicCalculatorIV(self, expression: str, evalvars: List[str], evalints:
List[int]) -> List[str]:
```

Python:

```
class Solution(object):
def basicCalculatorIV(self, expression, evalvars, evalints):
"""
:type expression: str
:type evalvars: List[str]
:type evalints: List[int]
:rtype: List[str]
"""
```

JavaScript:

```
/**
* @param {string} expression
* @param {string[]} evalvars
* @param {number[]} evalints
* @return {string[]}
*/
var basicCalculatorIV = function(expression, evalvars, evalints) {

};
```

TypeScript:

```
function basicCalculatorIV(expression: string, evalvars: string[], evalints:
number[]): string[] {

};
```

C#:

```
public class Solution {
public IList<string> BasicCalculatorIV(string expression, string[] evalvars,
int[] evalints) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** basicCalculatorIV(char* expression, char** evalvars, int evalvarsSize,
int* evalints, int evalintsSize, int* returnSize) {

}
```

Go:

```
func basicCalculatorIV(expression string, evalvars []string, evalints []int)
[]string {

}
```

Kotlin:

```
class Solution {
fun basicCalculatorIV(expression: String, evalvars: Array<String>, evalints:
IntArray): List<String> {

}
}
```

Swift:

```
class Solution {
func basicCalculatorIV(_ expression: String, _ evalvars: [String], _
evalints: [Int]) -> [String] {

}
}
```

Rust:

```
impl Solution {
pub fn basic_calculator_iv(expression: String, evalvars: Vec<String>,
evalints: Vec<i32>) -> Vec<String> {

}
}
```

Ruby:

```
# @param {String} expression
# @param {String[]} evalvars
# @param {Integer[]} evalints
# @return {String[]}
def basic_calculator_iv(expression, evalvars, evalints)

end
```

PHP:

```
class Solution {

/**
* @param String $expression
* @param String[] $evalvars
* @param Integer[] $evalints
* @return String[]
*/
function basicCalculatorIV($expression, $evalvars, $evalints) {

}
}
```

Dart:

```
class Solution {
List<String> basicCalculatorIV(String expression, List<String> evalvars,
List<int> evalints) {

}
}
```

Scala:

```
object Solution {
def basicCalculatorIV(expression: String, evalvars: Array[String], evalints:
Array[Int]): List[String] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec basic_calculator_iv(expression :: String.t, evalvars :: [String.t],
evalints :: [integer]) :: [String.t]
def basic_calculator_iv(expression, evalvars, evalints) do

end
end
```

Erlang:

```
-spec basic_calculator_iv(Expression :: unicode:unicode_binary(), Evalvars ::
[unicode:unicode_binary()], Evalints :: [integer()]) ->
[unicode:unicode_binary()].
basic_calculator_iv(Expression, Evalvars, Evalints) ->
.
```

Racket:

```
(define/contract (basic-calculator-iv expression evalvars evalints)
(-> string? (listof string?) (listof exact-integer?) (listof string?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Basic Calculator IV
* Difficulty: Hard
* Tags: string, graph, math, hash, sort, stack
```

```
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
vector<string> basicCalculatorIV(string expression, vector<string>& evalvars,
vector<int>& evalints) {

}
};
```

Java Solution:

```
/**
* Problem: Basic Calculator IV
* Difficulty: Hard
* Tags: string, graph, math, hash, sort, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public List<String> basicCalculatorIV(String expression, String[] evalvars,
int[] evalints) {

}
}
```

Python3 Solution:

```
"""
Problem: Basic Calculator IV
Difficulty: Hard
Tags: string, graph, math, hash, sort, stack

Approach: String manipulation with hash map or two pointers
```

```
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def basicCalculatorIV(self, expression: str, evalvars: List[str], evalints:
List[int]) -> List[str]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def basicCalculatorIV(self, expression, evalvars, evalints):
"""
:type expression: str
:type evalvars: List[str]
:type evalints: List[int]
:rtype: List[str]
"""
```

JavaScript Solution:

```
/**
* Problem: Basic Calculator IV
* Difficulty: Hard
* Tags: string, graph, math, hash, sort, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {string} expression
* @param {string[]} evalvars
* @param {number[]} evalints
* @return {string[]}
*/
var basicCalculatorIV = function(expression, evalvars, evalints) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Basic Calculator IV
* Difficulty: Hard
* Tags: string, graph, math, hash, sort, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function basicCalculatorIV(expression: string, evalvars: string[], evalints:
number[]): string[] {

};
```

C# Solution:

```
/*
* Problem: Basic Calculator IV
* Difficulty: Hard
* Tags: string, graph, math, hash, sort, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public IList<string> BasicCalculatorIV(string expression, string[] evalvars,
int[] evalints) {

}
}
```

C Solution:

```
/*
* Problem: Basic Calculator IV
* Difficulty: Hard
* Tags: string, graph, math, hash, sort, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** basicCalculatorIV(char* expression, char** evalvars, int evalvarsSize,
int* evalints, int evalintsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Basic Calculator IV
// Difficulty: Hard
// Tags: string, graph, math, hash, sort, stack
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func basicCalculatorIV(expression string, evalvars []string, evalints []int)
[]string {

}
```

Kotlin Solution:

```
class Solution {
fun basicCalculatorIV(expression: String, evalvars: Array<String>, evalints:
IntArray): List<String> {

}
}
```

Swift Solution:

```
class Solution {
func basicCalculatorIV(_ expression: String, _ evalvars: [String], _
evalints: [Int]) -> [String] {

}
}
```

Rust Solution:

```
// Problem: Basic Calculator IV
// Difficulty: Hard
// Tags: string, graph, math, hash, sort, stack
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn basic_calculator_iv(expression: String, evalvars: Vec<String>,
evalints: Vec<i32>) -> Vec<String> {

}
}
```

Ruby Solution:

```
# @param {String} expression
# @param {String[]} evalvars
# @param {Integer[]} evalints
# @return {String[]}
def basic_calculator_iv(expression, evalvars, evalints)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $expression
```

```
* @param String[] $evalvars
* @param Integer[] $evalints
* @return String[]
*/
function basicCalculatorIV($expression, $evalvars, $evalints) {

}
}
```

Dart Solution:

```
class Solution {
List<String> basicCalculatorIV(String expression, List<String> evalvars,
List<int> evalints) {

}
}
```

Scala Solution:

```
object Solution {
def basicCalculatorIV(expression: String, evalvars: Array[String], evalints:
Array[Int]): List[String] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec basic_calculator_iv(expression :: String.t, evalvars :: [String.t],
evalints :: [integer]) :: [String.t]
def basic_calculator_iv(expression, evalvars, evalints) do

end
end
```

Erlang Solution:

```
-spec basic_calculator_iv(Expression :: unicode:unicode_binary(), Evalvars ::
[unicode:unicode_binary()], Evalints :: [integer()]) ->
```

```
[unicode:unicode_binary()].
basic_calculator_iv(Expression, Evalvars, Evalints) ->
.
```

Racket Solution:

```
(define/contract (basic-calculator-iv expression evalvars evalints)
(-> string? (listof string?) (listof exact-integer?) (listof string?))
)
```