Problem 1807: Evaluate the Bracket Pairs of a String

Problem Information

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

Problem Description

You are given a string

s

that contains some bracket pairs, with each pair containing a

non-empty

key.

For example, in the string

"(name)is(age)yearsold"

, there are

two

bracket pairs that contain the keys

"name"

and

"age"

.

You know the values of a wide range of keys. This is represented by a 2D string array

knowledge

where each

knowledge[i] = [key

i

, value

i

]

indicates that key

key

i

has a value of

value

i

.

You are tasked to evaluate

all

of the bracket pairs. When you evaluate a bracket pair that contains some key

key

i

, you will:

Replace

key

i

and the bracket pair with the key's corresponding

value

i

.

If you do not know the value of the key, you will replace

key

i

and the bracket pair with a question mark

"?"

(without the quotation marks).

Each key will appear at most once in your

knowledge

. There will not be any nested brackets in

s

.

Return

the resulting string after evaluating

all

of the bracket pairs.

Example 1:

Input:

s = "(name)is(age)yearsold", knowledge = [["name","bob"],["age","two"]]

Output:

"bobistwoyearsold"

Explanation:

The key "name" has a value of "bob", so replace "(name)" with "bob". The key "age" has a value of "two", so replace "(age)" with "two".

Example 2:

Input:

s = "hi(name)", knowledge = [["a","b"]]

Output:

"hi?"

Explanation:

As you do not know the value of the key "name", replace "(name)" with "?".

Example 3:

Input:

s = "(a)(a)(a)aaa", knowledge = [["a","yes"]]

Output:

"yesyesyesaaa"

Explanation:

The same key can appear multiple times. The key "a" has a value of "yes", so replace all occurrences of "(a)" with "yes". Notice that the "a"s not in a bracket pair are not evaluated.

Constraints:

1 <= s.length <= 10

5

0 <= knowledge.length <= 10

5

knowledge[i].length == 2

1 <= key

i

.length, value

i

.length <= 10

s

consists of lowercase English letters and round brackets

'('

and

')'

.

Every open bracket

'('

in

s

will have a corresponding close bracket

')'

.

The key in each bracket pair of

s

will be non-empty.

There will not be any nested bracket pairs in

s

.

key

i

and

value

i

consist of lowercase English letters.

Each

key

i

in

knowledge

is unique.

Code Snippets

C++:

```
class Solution {
public:
string evaluate(string s, vector<vector<string>>& knowledge) {

}
};
```

Java:

```
class Solution {
public String evaluate(String s, List<List<String>> knowledge) {

}
```

```
}
```

Python3:

```
class Solution:
def evaluate(self, s: str, knowledge: List[List[str]]) -> str:
```

Python:

```
class Solution(object):
def evaluate(self, s, knowledge):
"""
:type s: str
:type knowledge: List[List[str]]
:rtype: str
"""
```

JavaScript:

```
/**
* @param {string} s
* @param {string[][]} knowledge
* @return {string}
*/
var evaluate = function(s, knowledge) {

};
```

TypeScript:

```
function evaluate(s: string, knowledge: string[][]): string {

};
```

C#:

```
public class Solution {
public string Evaluate(string s, IList<IList<string>> knowledge) {

}
}
```

C:

```
char* evaluate(char* s, char*** knowledge, int knowledgeSize, int*
knowledgeColSize) {

}
```

Go:

```
func evaluate(s string, knowledge [][]string) string {

}
```

Kotlin:

```
class Solution {
fun evaluate(s: String, knowledge: List<List<String>>): String {

}
}
```

Swift:

```
class Solution {
func evaluate(_ s: String, _ knowledge: [[String]]) -> String {

}
}
```

Rust:

```
impl Solution {
pub fn evaluate(s: String, knowledge: Vec<Vec<String>>) -> String {

}
}
```

Ruby:

```
# @param {String} s
# @param {String[][]} knowledge
# @return {String}
def evaluate(s, knowledge)
```

```
end
```

PHP:

```
class Solution {

/**
* @param String $s
* @param String[][] $knowledge
* @return String
*/
function evaluate($s, $knowledge) {

}
}
```

Dart:

```
class Solution {
String evaluate(String s, List<List<String>> knowledge) {

}
}
```

Scala:

```
object Solution {
def evaluate(s: String, knowledge: List[List[String]]): String = {

}
}
```

Elixir:

```
defmodule Solution do
@spec evaluate(s :: String.t, knowledge :: [[String.t]]) :: String.t
def evaluate(s, knowledge) do

end
end
```

Erlang:

```
-spec evaluate(S :: unicode:unicode_binary(), Knowledge ::
[[unicode:unicode_binary()]]) -> unicode:unicode_binary().
evaluate(S, Knowledge) ->
.
```

Racket:

```
(define/contract (evaluate s knowledge)
(-> string? (listof (listof string?)) string?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Evaluate the Bracket Pairs of a String
* Difficulty: Medium
* Tags: array, string, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
string evaluate(string s, vector<vector<string>>& knowledge) {

}
};
```

Java Solution:

```
/**
* Problem: Evaluate the Bracket Pairs of a String
* Difficulty: Medium
* Tags: array, string, hash
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public String evaluate(String s, List<List<String>> knowledge) {

}
}
```

Python3 Solution:

```
"""
Problem: Evaluate the Bracket Pairs of a String
Difficulty: Medium
Tags: array, string, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def evaluate(self, s: str, knowledge: List[List[str]]) -> str:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def evaluate(self, s, knowledge):
"""
:type s: str
:type knowledge: List[List[str]]
:rtype: str
"""
```

JavaScript Solution:

```
/**
* Problem: Evaluate the Bracket Pairs of a String
* Difficulty: Medium
* Tags: array, string, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {string} s
* @param {string[][]} knowledge
* @return {string}
*/
var evaluate = function(s, knowledge) {

};
```

TypeScript Solution:

```
/**
* Problem: Evaluate the Bracket Pairs of a String
* Difficulty: Medium
* Tags: array, string, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function evaluate(s: string, knowledge: string[][]): string {

};
```

C# Solution:

```
/*
* Problem: Evaluate the Bracket Pairs of a String
* Difficulty: Medium
* Tags: array, string, hash
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public string Evaluate(string s, IList<IList<string>> knowledge) {

}
}
```

C Solution:

```
/*
* Problem: Evaluate the Bracket Pairs of a String
* Difficulty: Medium
* Tags: array, string, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

char* evaluate(char* s, char*** knowledge, int knowledgeSize, int*
knowledgeColSize) {

}
```

Go Solution:

```
// Problem: Evaluate the Bracket Pairs of a String
// Difficulty: Medium
// Tags: array, string, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func evaluate(s string, knowledge [][]string) string {

}
```

Kotlin Solution:

```
class Solution {
fun evaluate(s: String, knowledge: List<List<String>>): String {

}
}
```

Swift Solution:

```
class Solution {
func evaluate(_ s: String, _ knowledge: [[String]]) -> String {

}
}
```

Rust Solution:

```
// Problem: Evaluate the Bracket Pairs of a String
// Difficulty: Medium
// Tags: array, string, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn evaluate(s: String, knowledge: Vec<Vec<String>>) -> String {

}
}
```

Ruby Solution:

```
# @param {String} s
# @param {String[][]} knowledge
# @return {String}
def evaluate(s, knowledge)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $s
* @param String[][] $knowledge
* @return String
*/
function evaluate($s, $knowledge) {

}
}
```

Dart Solution:

```
class Solution {
String evaluate(String s, List<List<String>> knowledge) {

}
}
```

Scala Solution:

```
object Solution {
def evaluate(s: String, knowledge: List[List[String]]): String = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec evaluate(s :: String.t, knowledge :: [[String.t]]) :: String.t
def evaluate(s, knowledge) do

end
end
```

Erlang Solution:

```
-spec evaluate(S :: unicode:unicode_binary(), Knowledge ::
[[unicode:unicode_binary()]]) -> unicode:unicode_binary().
evaluate(S, Knowledge) ->
.
```

Racket Solution:

```
(define/contract (evaluate s knowledge)
(-> string? (listof (listof string?)) string?)
)
```