Problem 1111: Maximum Nesting Depth of Two Valid Parentheses Strings

Problem Information

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

Problem Description

A string is a

valid parentheses string

(denoted VPS) if and only if it consists of

"("

and

")"

characters only, and:

It is the empty string, or

It can be written as

AB

(

A

concatenated with

B

), where

A

and

B

are VPS's, or

It can be written as

(A)

, where

A

is a VPS.

We can similarly define the

nesting depth

depth(S)

of any VPS

S

as follows:

depth("") = 0

depth(A + B) = max(depth(A), depth(B))

, where

A

and

B

are VPS's

depth("(" + A + ")") = 1 + depth(A)

, where

A

is a VPS.

For example,

""

,

"()()"

, and

"()(()())"

are VPS's (with nesting depths 0, 1, and 2), and

")("

and

"(()"

are not VPS's.

Given a VPS

seq

, split it into two disjoint subsequences

A

and

B

, such that

A

and

B

are VPS's (and

A.length + B.length = seq.length

).

Now choose

any

such

A

and

B

such that

max(depth(A), depth(B))

is the minimum possible value.

Return an

answer

array (of length

seq.length

) that encodes such a choice of

A

and

B

:

answer[i] = 0

if

seq[i]

is part of

A

, else

answer[i] = 1

. Note that even though multiple answers may exist, you may return any of them.

Example 1:

Input:

seq = "(()())"

Output:

[0,1,1,1,1,0]

Example 2:

Input:

seq = "()(())()"

Output:

[0,0,0,1,1,0,1,1]

Constraints:

1 <= seq.size <= 10000

Code Snippets

C++:

```
class Solution {
public:
vector<int> maxDepthAfterSplit(string seq) {

}
};
```

Java:

```
class Solution {
public int[] maxDepthAfterSplit(String seq) {
```

```
}
}
```

Python3:

```
class Solution:
def maxDepthAfterSplit(self, seq: str) -> List[int]:
```

Python:

```
class Solution(object):
def maxDepthAfterSplit(self, seq):
"""
:type seq: str
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {string} seq
* @return {number[]}
*/
var maxDepthAfterSplit = function(seq) {

};
```

TypeScript:

```
function maxDepthAfterSplit(seq: string): number[] {

};
```

C#:

```
public class Solution {
public int[] MaxDepthAfterSplit(string seq) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* maxDepthAfterSplit(char* seq, int* returnSize) {

}
```

Go:

```
func maxDepthAfterSplit(seq string) []int {

}
```

Kotlin:

```
class Solution {
fun maxDepthAfterSplit(seq: String): IntArray {

}
}
```

Swift:

```
class Solution {
func maxDepthAfterSplit(_ seq: String) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn max_depth_after_split(seq: String) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {String} seq
# @return {Integer[]}
```

```
def max_depth_after_split(seq)

end
```

PHP:

```
class Solution {

/**
* @param String $seq
* @return Integer[]
*/
function maxDepthAfterSplit($seq) {

}
}
```

Dart:

```
class Solution {
List<int> maxDepthAfterSplit(String seq) {

}
}
```

Scala:

```
object Solution {
def maxDepthAfterSplit(seq: String): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_depth_after_split(seq :: String.t) :: [integer]
def max_depth_after_split(seq) do

end
end
```

Erlang:

```
-spec max_depth_after_split(Seq :: unicode:unicode_binary()) -> [integer()].
max_depth_after_split(Seq) ->
.
```

Racket:

```
(define/contract (max-depth-after-split seq)
(-> string? (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
* Difficulty: Medium
* Tags: array, string, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<int> maxDepthAfterSplit(string seq) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
* Difficulty: Medium
* Tags: array, string, stack
*
* Approach: Use two pointers or sliding window technique
```

```
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int[] maxDepthAfterSplit(String seq) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
Difficulty: Medium
Tags: array, string, stack

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def maxDepthAfterSplit(self, seq: str) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maxDepthAfterSplit(self, seq):
"""
:type seq: str
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
* Difficulty: Medium
```

```
* Tags: array, string, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {string} seq
* @return {number[]}
*/
var maxDepthAfterSplit = function(seq) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
* Difficulty: Medium
* Tags: array, string, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function maxDepthAfterSplit(seq: string): number[] {

};
```

C# Solution:

```
/*
* Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
* Difficulty: Medium
* Tags: array, string, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
```

```
*/

public class Solution {
public int[] MaxDepthAfterSplit(string seq) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
* Difficulty: Medium
* Tags: array, string, stack
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* maxDepthAfterSplit(char* seq, int* returnSize) {

}
```

Go Solution:

```
// Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
// Difficulty: Medium
// Tags: array, string, stack
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func maxDepthAfterSplit(seq string) []int {

}
```

Kotlin Solution:

```
class Solution {
fun maxDepthAfterSplit(seq: String): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func maxDepthAfterSplit(_ seq: String) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Maximum Nesting Depth of Two Valid Parentheses Strings
// Difficulty: Medium
// Tags: array, string, stack
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn max_depth_after_split(seq: String) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {String} seq
# @return {Integer[]}
def max_depth_after_split(seq)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $seq
* @return Integer[]
*/
function maxDepthAfterSplit($seq) {

}
}
```

Dart Solution:

```
class Solution {
List<int> maxDepthAfterSplit(String seq) {

}
}
```

Scala Solution:

```
object Solution {
def maxDepthAfterSplit(seq: String): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_depth_after_split(seq :: String.t) :: [integer]
def max_depth_after_split(seq) do

end
end
```

Erlang Solution:

```
-spec max_depth_after_split(Seq :: unicode:unicode_binary()) -> [integer()].
max_depth_after_split(Seq) ->
.
```

Racket Solution:

```
(define/contract (max-depth-after-split seq)
(-> string? (listof exact-integer?))
)
```