Problem 2116: Check if a Parentheses String Can Be Valid

Problem Information

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

Problem Description

A parentheses string is a

non-empty

string consisting only of

'('

and

')'

. It is valid if

any

of the following conditions is

true

:

It is

()

.

It can be written as

AB

(

A

concatenated with

B

), where

A

and

B

are valid parentheses strings.

It can be written as

(A)

, where

A

is a valid parentheses string.

You are given a parentheses string

s

and a string

locked

, both of length

n

.

locked

is a binary string consisting only of

'0'

s and

'1'

s. For

each

index

i

of

locked

,

If

locked[i]

is

'1'

, you

cannot

change

s[i]

.

But if

locked[i]

is

'0'

, you

can

change

s[i]

to either

'('

or

')'

.

Return

true

if you can make

s

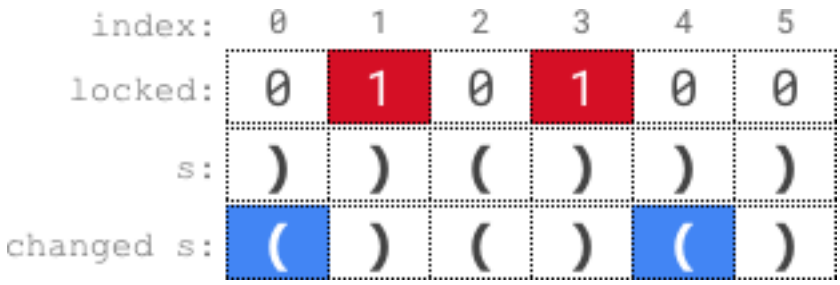
a valid parentheses string

. Otherwise, return

false

.

Example 1:



Input:

s = "))()))", locked = "010100"

Output:

true

Explanation:

locked[1] == '1' and locked[3] == '1', so we cannot change s[1] or s[3]. We change s[0] and s[4] to '(' while leaving s[2] and s[5] unchanged to make s valid.

Example 2:

Input:

s = "()()", locked = "0000"

Output:

true

Explanation:

We do not need to make any changes because s is already valid.

Example 3:

Input:

s = ")", locked = "0"

Output:

false

Explanation:

locked permits us to change s[0]. Changing s[0] to either '(' or ')' will not make s valid.

Example 4:

Input:

s = "((()))((()))", locked = "111111010111"

Output:

true

Explanation:

locked permits us to change s[6] and s[8]. We change s[6] and s[8] to ')' to make s valid.

Constraints:

n == s.length == locked.length

1 <= n <= 10

5

s[i]

is either

'('

or

')'

.

locked[i]

is either

'0'

or

'1'

.

Code Snippets

C++:

```
class Solution {
public:
bool canBeValid(string s, string locked) {

}
};
```

Java:

```
class Solution {
public boolean canBeValid(String s, String locked) {

}
}
```

Python3:

```
class Solution:
def canBeValid(self, s: str, locked: str) -> bool:
```

Python:

```
class Solution(object):
def canBeValid(self, s, locked):
"""
:type s: str
:type locked: str
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {string} s
* @param {string} locked
* @return {boolean}
*/
var canBeValid = function(s, locked) {

};
```

TypeScript:

```
function canBeValid(s: string, locked: string): boolean {

};
```

C#:

```
public class Solution {
public bool CanBeValid(string s, string locked) {

}
}
```

C:

```
bool canBeValid(char* s, char* locked) {

}
```

Go:

```
func canBeValid(s string, locked string) bool {

}
```

Kotlin:

```
class Solution {
fun canBeValid(s: String, locked: String): Boolean {

}
}
```

Swift:

```
class Solution {
func canBeValid(_ s: String, _ locked: String) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn can_be_valid(s: String, locked: String) -> bool {

}
}
```

Ruby:

```
# @param {String} s
# @param {String} locked
# @return {Boolean}
def can_be_valid(s, locked)

end
```

PHP:

```
class Solution {

/**
* @param String $s
* @param String $locked
* @return Boolean
*/
function canBeValid($s, $locked) {

}
}
```

Dart:

```
class Solution {
bool canBeValid(String s, String locked) {

}
}
```

Scala:

```
object Solution {
def canBeValid(s: String, locked: String): Boolean = {

}
```

```
}
```

Elixir:

```
defmodule Solution do
@spec can_be_valid(s :: String.t, locked :: String.t) :: boolean
def can_be_valid(s, locked) do

end
end
```

Erlang:

```
-spec can_be_valid(S :: unicode:unicode_binary(), Locked ::
unicode:unicode_binary()) -> boolean().
can_be_valid(S, Locked) ->
.
```

Racket:

```
(define/contract (can-be-valid s locked)
(-> string? string? boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Check if a Parentheses String Can Be Valid
* Difficulty: Medium
* Tags: string, greedy, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
```

```
bool canBeValid(string s, string locked) {

}
};
```

Java Solution:

```
/**
* Problem: Check if a Parentheses String Can Be Valid
* Difficulty: Medium
* Tags: string, greedy, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public boolean canBeValid(String s, String locked) {

}
}
```

Python3 Solution:

```
"""
Problem: Check if a Parentheses String Can Be Valid
Difficulty: Medium
Tags: string, greedy, stack

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def canBeValid(self, s: str, locked: str) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def canBeValid(self, s, locked):
"""
:type s: str
:type locked: str
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Check if a Parentheses String Can Be Valid
* Difficulty: Medium
* Tags: string, greedy, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {string} s
* @param {string} locked
* @return {boolean}
*/
var canBeValid = function(s, locked) {

};
```

TypeScript Solution:

```
/**
* Problem: Check if a Parentheses String Can Be Valid
* Difficulty: Medium
* Tags: string, greedy, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function canBeValid(s: string, locked: string): boolean {
```

```
};
```

C# Solution:

```
/*
* Problem: Check if a Parentheses String Can Be Valid
* Difficulty: Medium
* Tags: string, greedy, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public bool CanBeValid(string s, string locked) {

}
}
```

C Solution:

```
/*
* Problem: Check if a Parentheses String Can Be Valid
* Difficulty: Medium
* Tags: string, greedy, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

bool canBeValid(char* s, char* locked) {

}
```

Go Solution:

```
// Problem: Check if a Parentheses String Can Be Valid
// Difficulty: Medium
```

```
// Tags: string, greedy, stack
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func canBeValid(s string, locked string) bool {

}
```

Kotlin Solution:

```
class Solution {
fun canBeValid(s: String, locked: String): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func canBeValid(_ s: String, _ locked: String) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Check if a Parentheses String Can Be Valid
// Difficulty: Medium
// Tags: string, greedy, stack
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn can_be_valid(s: String, locked: String) -> bool {

}
}
```

Ruby Solution:

```
# @param {String} s
# @param {String} locked
# @return {Boolean}
def can_be_valid(s, locked)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $s
* @param String $locked
* @return Boolean
*/
function canBeValid($s, $locked) {

}
}
```

Dart Solution:

```
class Solution {
bool canBeValid(String s, String locked) {

}
}
```

Scala Solution:

```
object Solution {
def canBeValid(s: String, locked: String): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec can_be_valid(s :: String.t, locked :: String.t) :: boolean
def can_be_valid(s, locked) do

end
end
```

Erlang Solution:

```
-spec can_be_valid(S :: unicode:unicode_binary(), Locked ::
unicode:unicode_binary()) -> boolean().
can_be_valid(S, Locked) ->
.
```

Racket Solution:

```
(define/contract (can-be-valid s locked)
(-> string? string? boolean?)
)
```