Problem 3501: Maximize Active Section with Trade II

Problem Information

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

Problem Description

You are given a binary string

s

of length

n

, where:

'1'

represents an

active

section.

'0'

represents an

inactive

section.

You can perform

at most one trade

to maximize the number of active sections in

s

. In a trade, you:

Convert a contiguous block of

'1'

s that is surrounded by

'0'

s to all

'0'

s.

Afterward, convert a contiguous block of

'0'

s that is surrounded by

'1'

s to all

'1'

s.

Additionally, you are given a

2D array

queries

, where

queries[i] = [l

i

, r

i

]

represents a

substring

s[l

i

...r

i

]

.

For each query, determine the

maximum

possible number of active sections in

s

after making the optimal trade on the substring

s[l

i

...r

i

]

.

Return an array

answer

, where

answer[i]

is the result for

queries[i]

.

Note

For each query, treat

s[l

i

...r

i

]

as if it is

augmented

with a

'1'

at both ends, forming

t = '1' + s[l

i

...r

i

] + '1'

. The augmented

'1'

s

do not

contribute to the final count.

The queries are independent of each other.

Example 1:

Input:

s = "01", queries = [[0,1]]

Output:

[1]

Explanation:

Because there is no block of

'1'

s surrounded by

'0'

s, no valid trade is possible. The maximum number of active sections is 1.

Example 2:

Input:

s = "0100", queries = [[0,3],[0,2],[1,3],[2,3]]

Output:

[4,3,1,1]

Explanation:

Query

[0, 3]

→ Substring

"0100"

→ Augmented to

"101001"

Choose

"0100"

, convert

"0100"

→

"0000"

→

"1111"

.

The final string without augmentation is

"1111"

. The maximum number of active sections is 4.

Query

[0, 2]

→ Substring

"010"

→ Augmented to

"10101"

Choose

"010"

, convert

"010"

$\rightarrow$

"000"

$\rightarrow$

"111"

.

The final string without augmentation is

"1110"

. The maximum number of active sections is 3.

Query

[1, 3]

$\rightarrow$ Substring

"100"

$\rightarrow$ Augmented to

"11001"

Because there is no block of

'1'

s surrounded by

'0'

s, no valid trade is possible. The maximum number of active sections is 1.

Query

[2, 3]

→ Substring

"00"

→ Augmented to

"1001"

Because there is no block of

'1'

s surrounded by

'0'

s, no valid trade is possible. The maximum number of active sections is 1.

Example 3:

Input:

s = "1000100", queries = [[1,5],[0,6],[0,4]]

Output:

[6,7,2]

Explanation:

Query

[1, 5]

→ Substring

"00010"

→ Augmented to

"1000101"

Choose

"00010"

, convert

"00010"

→

"00000"

→

"11111"

.

The final string without augmentation is

"1111110"

. The maximum number of active sections is 6.

Query

[0, 6]

$\rightarrow$ Substring

"1000100"

$\rightarrow$ Augmented to

"110001001"

Choose

"000100"

, convert

"000100"

$\rightarrow$

"000000"

$\rightarrow$

"111111"

.

The final string without augmentation is

"1111111"

. The maximum number of active sections is 7.

Query

[0, 4]

→ Substring

"10001"

→ Augmented to

"1100011"

Because there is no block of

'1'

s surrounded by

'0'

s, no valid trade is possible. The maximum number of active sections is 2.

Example 4:

Input:

s = "01010", queries = [[0,3],[1,4],[1,3]]

Output:

[4,4,2]

Explanation:

Query

[0, 3]

→ Substring

"0101"

$\rightarrow$ Augmented to

"101011"

Choose

"010"

, convert

"010"

$\rightarrow$

"000"

$\rightarrow$

"111"

.

The final string without augmentation is

"11110"

. The maximum number of active sections is 4.

Query

[1, 4]

$\rightarrow$ Substring

"1010"

$\rightarrow$ Augmented to

"110101"

Choose

"010"

, convert

"010"

$\to$

"000"

$\to$

"111"

.

The final string without augmentation is

"01111"

. The maximum number of active sections is 4.

Query

[1, 3]

$\to$ Substring

"101"

$\to$ Augmented to

"11011"

Because there is no block of

'1'

s surrounded by

'0'

s, no valid trade is possible. The maximum number of active sections is 2.

Constraints:

1 <= n == s.length <= 10

5

1 <= queries.length <= 10

5

s[i]

is either

'0'

or

'1'

.

queries[i] = [l

i

, r

i

]

0 <= l

i

<= r

i

< n

Code Snippets

C++:

```
class Solution {
public:
vector<int> maxActiveSectionsAfterTrade(string s, vector<vector<int>>&
queries) {

}
};
```

Java:

```
class Solution {
public List<Integer> maxActiveSectionsAfterTrade(String s, int[][] queries) {

}
}
```

Python3:

```
class Solution:
def maxActiveSectionsAfterTrade(self, s: str, queries: List[List[int]]) ->
List[int]:
```

Python:

```
class Solution(object):
def maxActiveSectionsAfterTrade(self, s, queries):
"""
:type s: str
:type queries: List[List[int]]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {string} s
* @param {number[][]} queries
* @return {number[]}
*/
var maxActiveSectionsAfterTrade = function(s, queries) {

};
```

TypeScript:

```
function maxActiveSectionsAfterTrade(s: string, queries: number[][]):
number[] {

};
```

C#:

```
public class Solution {
public IList<int> MaxActiveSectionsAfterTrade(string s, int[][] queries) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* maxActiveSectionsAfterTrade(char* s, int** queries, int queriesSize,
int* queriesColSize, int* returnSize) {

}
```

Go:

```
func maxActiveSectionsAfterTrade(s string, queries [][]int) []int {

}
```

Kotlin:

```
class Solution {
fun maxActiveSectionsAfterTrade(s: String, queries: Array<IntArray>):
List<Int> {

}
}
```

Swift:

```
class Solution {
func maxActiveSectionsAfterTrade(_ s: String, _ queries: [[Int]]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn max_active_sections_after_trade(s: String, queries: Vec<Vec<i32>>) ->
Vec<i32> {

}
}
```

Ruby:

```
# @param {String} s
# @param {Integer[][]} queries
# @return {Integer[]}
def max_active_sections_after_trade(s, queries)

end
```

PHP:

```
class Solution {

/**
* @param String $s
* @param Integer[][] $queries
* @return Integer[]
*/
function maxActiveSectionsAfterTrade($s, $queries) {

}
}
```

Dart:

```
class Solution {
List<int> maxActiveSectionsAfterTrade(String s, List<List<int>> queries) {

}
}
```

Scala:

```
object Solution {
def maxActiveSectionsAfterTrade(s: String, queries: Array[Array[Int]]):
List[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_active_sections_after_trade(s :: String.t, queries :: [[integer]])
:: [integer]
def max_active_sections_after_trade(s, queries) do

end
end
```

Erlang:

```
-spec max_active_sections_after_trade(S :: unicode:unicode_binary(), Queries
:: [[integer()]]) -> [integer()].
```

```
max_active_sections_after_trade(S, Queries) ->
.
```

Racket:

```
(define/contract (max-active-sections-after-trade s queries)
(-> string? (listof (listof exact-integer?)) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximize Active Section with Trade II
* Difficulty: Hard
* Tags: array, string, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
vector<int> maxActiveSectionsAfterTrade(string s, vector<vector<int>>&
queries) {

}
};
```

Java Solution:

```
/**
* Problem: Maximize Active Section with Trade II
* Difficulty: Hard
* Tags: array, string, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
```

```
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public List<Integer> maxActiveSectionsAfterTrade(String s, int[][] queries) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximize Active Section with Trade II
Difficulty: Hard
Tags: array, string, tree, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

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

Python Solution:

```
class Solution(object):
def maxActiveSectionsAfterTrade(self, s, queries):
"""
:type s: str
:type queries: List[List[int]]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Maximize Active Section with Trade II
```

```
* Difficulty: Hard
* Tags: array, string, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* @param {string} s
* @param {number[][]} queries
* @return {number[]}
*/
var maxActiveSectionsAfterTrade = function(s, queries) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximize Active Section with Trade II
* Difficulty: Hard
* Tags: array, string, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function maxActiveSectionsAfterTrade(s: string, queries: number[][]):
number[] {

};
```

C# Solution:

```
/*
* Problem: Maximize Active Section with Trade II
* Difficulty: Hard
* Tags: array, string, tree, search
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public IList<int> MaxActiveSectionsAfterTrade(string s, int[][] queries) {

}
}
```

C Solution:

```
/*
* Problem: Maximize Active Section with Trade II
* Difficulty: Hard
* Tags: array, string, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* maxActiveSectionsAfterTrade(char* s, int** queries, int queriesSize,
int* queriesColSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Maximize Active Section with Trade II
// Difficulty: Hard
// Tags: array, string, tree, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height
```

```
func maxActiveSectionsAfterTrade(s string, queries [][]int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun maxActiveSectionsAfterTrade(s: String, queries: Array<IntArray>):
List<Int> {

}
}
```

Swift Solution:

```
class Solution {
func maxActiveSectionsAfterTrade(_ s: String, _ queries: [[Int]]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Maximize Active Section with Trade II
// Difficulty: Hard
// Tags: array, string, tree, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn max_active_sections_after_trade(s: String, queries: Vec<Vec<i32>>) ->
Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {String} s
# @param {Integer[][]} queries
# @return {Integer[]}
def max_active_sections_after_trade(s, queries)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $s
* @param Integer[][] $queries
* @return Integer[]
*/
function maxActiveSectionsAfterTrade($s, $queries) {

}
}
```

Dart Solution:

```
class Solution {
List<int> maxActiveSectionsAfterTrade(String s, List<List<int>> queries) {

}
}
```

Scala Solution:

```
object Solution {
def maxActiveSectionsAfterTrade(s: String, queries: Array[Array[Int]]):
List[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_active_sections_after_trade(s :: String.t, queries :: [[integer]])
```

```
:: [integer]
def max_active_sections_after_trade(s, queries) do

end
end
```

Erlang Solution:

```
-spec max_active_sections_after_trade(S :: unicode:unicode_binary(), Queries
:: [[integer()]]) -> [integer()].
max_active_sections_after_trade(S, Queries) ->
.
```

Racket Solution:

```
(define/contract (max-active-sections-after-trade s queries)
(-> string? (listof (listof exact-integer?)) (listof exact-integer?))
)
```