Problem 2947: Count Beautiful Substrings I

Problem Information

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

Problem Description

You are given a string

s

and a positive integer

k

.

Let

vowels

and

consonants

be the number of vowels and consonants in a string.

A string is

beautiful

if:

vowels == consonants

.

(vowels * consonants) % k == 0

, in other terms the multiplication of

vowels

and

consonants

is divisible by

k

.

Return

the number of

non-empty beautiful substrings

in the given string

s

.

A

substring

is a contiguous sequence of characters in a string.

Vowel letters

in English are

'a'

,

'e'

,

'i'

,

'o'

, and

'u'

.

Consonant letters

in English are every letter except vowels.

Example 1:

Input:

s = "baeyh", k = 2

Output:

2

Explanation:

There are 2 beautiful substrings in the given string. - Substring "b

aeyh

", vowels = 2 (["a",e"]), consonants = 2 (["y","h"]). You can see that string "aeyh" is beautiful as vowels == consonants and vowels * consonants % k == 0. - Substring "

baey

h", vowels = 2 (["a",e"]), consonants = 2 (["b","y"]). You can see that string "baey" is beautiful as vowels == consonants and vowels * consonants % k == 0. It can be shown that there are only 2 beautiful substrings in the given string.

Example 2:

Input:

s = "abba", k = 1

Output:

3

Explanation:

There are 3 beautiful substrings in the given string. - Substring "

ab

ba", vowels = 1 (["a"]), consonants = 1 (["b"]). - Substring "ab

ba

", vowels = 1 (["a"]), consonants = 1 (["b"]). - Substring "

abba

", vowels = 2 (["a","a"]), consonants = 2 (["b","b"]). It can be shown that there are only 3 beautiful substrings in the given string.

Example 3:

Input:

s = "bcdf", k = 1

Output:

0

Explanation:

There are no beautiful substrings in the given string.

Constraints:

1 <= s.length <= 1000

1 <= k <= 1000

s

consists of only English lowercase letters.

Code Snippets

C++:

```
class Solution {
public:
int beautifulSubstrings(string s, int k) {

}
};
```

Java:

```
class Solution {
public int beautifulSubstrings(String s, int k) {
```

```
}
}
```

Python3:

```
class Solution:
def beautifulSubstrings(self, s: str, k: int) -> int:
```

Python:

```
class Solution(object):
def beautifulSubstrings(self, s, k):
"""
:type s: str
:type k: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {string} s
* @param {number} k
* @return {number}
*/
var beautifulSubstrings = function(s, k) {

};
```

TypeScript:

```
function beautifulSubstrings(s: string, k: number): number {

};
```

C#:

```
public class Solution {
public int BeautifulSubstrings(string s, int k) {

}
```

```
}
```

C:

```
int beautifulSubstrings(char* s, int k) {

}
```

Go:

```
func beautifulSubstrings(s string, k int) int {

}
```

Kotlin:

```
class Solution {
fun beautifulSubstrings(s: String, k: Int): Int {

}
}
```

Swift:

```
class Solution {
func beautifulSubstrings(_ s: String, _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn beautiful_substrings(s: String, k: i32) -> i32 {

}
}
```

Ruby:

```
# @param {String} s
# @param {Integer} k
```

```
# @return {Integer}
def beautiful_substrings(s, k)

end
```

PHP:

```
class Solution {

/**
* @param String $s
* @param Integer $k
* @return Integer
*/
function beautifulSubstrings($s, $k) {

}
}
```

Dart:

```
class Solution {
int beautifulSubstrings(String s, int k) {

}
}
```

Scala:

```
object Solution {
def beautifulSubstrings(s: String, k: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec beautiful_substrings(s :: String.t, k :: integer) :: integer
def beautiful_substrings(s, k) do

end
```

```
end
```

Erlang:

```
-spec beautiful_substrings(S :: unicode:unicode_binary(), K :: integer()) ->
integer().
beautiful_substrings(S, K) ->
.
```

Racket:

```
(define/contract (beautiful-substrings s k)
(-> string? exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Count Beautiful Substrings I
* Difficulty: Medium
* Tags: array, string, tree, math, hash
*
* 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:
int beautifulSubstrings(string s, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Count Beautiful Substrings I
```

```
* Difficulty: Medium
* Tags: array, string, tree, math, hash
*
* 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 int beautifulSubstrings(String s, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Count Beautiful Substrings I
Difficulty: Medium
Tags: array, string, tree, math, hash

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 beautifulSubstrings(self, s: str, k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def beautifulSubstrings(self, s, k):
"""
:type s: str
:type k: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Count Beautiful Substrings I
* Difficulty: Medium
* Tags: array, string, tree, math, hash
*
* 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} k
* @return {number}
*/
var beautifulSubstrings = function(s, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Count Beautiful Substrings I
* Difficulty: Medium
* Tags: array, string, tree, math, hash
*
* 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 beautifulSubstrings(s: string, k: number): number {

};
```

C# Solution:

```
/*
* Problem: Count Beautiful Substrings I
* Difficulty: Medium
* Tags: array, string, tree, math, hash
```

```
*
* 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 int BeautifulSubstrings(string s, int k) {

}
}
```

C Solution:

```
/*
* Problem: Count Beautiful Substrings I
* Difficulty: Medium
* Tags: array, string, tree, math, hash
*
* 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
*/

int beautifulSubstrings(char* s, int k) {

}
```

Go Solution:

```
// Problem: Count Beautiful Substrings I
// Difficulty: Medium
// Tags: array, string, tree, math, hash
//
// 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 beautifulSubstrings(s string, k int) int {

}
```

Kotlin Solution:

```
class Solution {
fun beautifulSubstrings(s: String, k: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func beautifulSubstrings(_ s: String, _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Count Beautiful Substrings I
// Difficulty: Medium
// Tags: array, string, tree, math, hash
//
// 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 beautiful_substrings(s: String, k: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {String} s
# @param {Integer} k
# @return {Integer}
def beautiful_substrings(s, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $s
* @param Integer $k
* @return Integer
*/
function beautifulSubstrings($s, $k) {

}
}
```

Dart Solution:

```
class Solution {
int beautifulSubstrings(String s, int k) {

}
}
```

Scala Solution:

```
object Solution {
def beautifulSubstrings(s: String, k: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec beautiful_substrings(s :: String.t, k :: integer) :: integer
def beautiful_substrings(s, k) do

end
end
```

Erlang Solution:

```
-spec beautiful_substrings(S :: unicode:unicode_binary(), K :: integer()) ->
integer().
beautiful_substrings(S, K) ->
.
```

Racket Solution:

```
(define/contract (beautiful-substrings s k)
(-> string? exact-integer? exact-integer?)
)
```