Problem 1400: Construct K Palindrome Strings

Problem Information

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

Problem Description

Given a string

s

and an integer

k

, return

true

if you can use all the characters in

s

to construct

non-empty

k

palindrome strings

or

false

otherwise.

Example 1:

Input:

s = "annabelle", k = 2

Output:

true

Explanation:

You can construct two palindromes using all characters in s. Some possible constructions "anna" + "elble", "anbna" + "elle", "anellena" + "b"

Example 2:

Input:

s = "leetcode", k = 3

Output:

false

Explanation:

It is impossible to construct 3 palindromes using all the characters of s.

Example 3:

Input:

s = "true", k = 4

Output:

true

Explanation:

The only possible solution is to put each character in a separate string.

Constraints:

1 <= s.length <= 10

5

s

consists of lowercase English letters.

1 <= k <= 10

5

Code Snippets

C++:

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

}
};
```

Java:

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

}
```

```
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function canConstruct(s: string, k: number): boolean {

};
```

C#:

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

}
}
```

C:

```
bool canConstruct(char* s, int k) {

}
```

Go:

```
func canConstruct(s string, k int) bool {

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

```
end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

```
class Solution {
bool canConstruct(String s, int k) {

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (can-construct s k)
(-> string? exact-integer? boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Construct K Palindrome Strings
* Difficulty: Medium
* Tags: string, greedy, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
bool canConstruct(string s, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Construct K Palindrome Strings
* Difficulty: Medium
* Tags: string, greedy, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
```

```
* Space Complexity: O(n) for hash map
*/

class Solution {
public boolean canConstruct(String s, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Construct K Palindrome Strings
Difficulty: Medium
Tags: string, greedy, hash

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Construct K Palindrome Strings
* Difficulty: Medium
```

```
* Tags: string, greedy, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

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

};
```

TypeScript Solution:

```
/**
* Problem: Construct K Palindrome Strings
* Difficulty: Medium
* Tags: string, greedy, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function canConstruct(s: string, k: number): boolean {

};
```

C# Solution:

```
/*
* Problem: Construct K Palindrome Strings
* Difficulty: Medium
* Tags: string, greedy, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
```

```
* Space Complexity: O(n) for hash map
*/

public class Solution {
public bool CanConstruct(string s, int k) {

}
}
```

C Solution:

```
/*
* Problem: Construct K Palindrome Strings
* Difficulty: Medium
* Tags: string, greedy, hash
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

bool canConstruct(char* s, int k) {

}
```

Go Solution:

```
// Problem: Construct K Palindrome Strings
// Difficulty: Medium
// Tags: string, greedy, hash
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func canConstruct(s string, k int) bool {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Construct K Palindrome Strings
// Difficulty: Medium
// Tags: string, greedy, hash
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn can_construct(s: String, k: i32) -> bool {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

```
class Solution {
bool canConstruct(String s, int k) {

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

```
.
```

Racket Solution:

```
(define/contract (can-construct s k)
(-> string? exact-integer? boolean?)
)
```