Problem 2868: The Wording Game

Problem Information

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

Problem Description

Alice and Bob each have a

lexicographically sorted

array of strings named

a

and

b

respectively.

They are playing a wording game with the following rules:

On each turn, the current player should play a word from their list such that the new word is

closely greater

than the last played word; then it's the other player's turn.

If a player can't play a word on their turn, they lose.

Alice starts the game by playing her

lexicographically

smallest

word.

Given

a

and

b

, return

true

if Alice can win knowing that both players play their best, and

false

otherwise.

A word

w

is

closely greater

than a word

z

if the following conditions are met:

w

is

lexicographically greater

than

z

.

If

w

1

is the first letter of

w

and

z

1

is the first letter of

z

,

w

1

should either be

equal

to

z

1

or be the

letter after

z

1

in the alphabet.

For example, the word

"care"

is closely greater than

"book"

and

"car"

, but is not closely greater than

"ant"

or

"cook"

.

A string

s

is

lexicographically

greater

than a string

t

if in the first position where

s

and

t

differ, string

s

has a letter that appears later in the alphabet than the corresponding letter in

t

. If the first

min(s.length, t.length)

characters do not differ, then the longer string is the lexicographically greater one.

Example 1:

Input:

a = ["avokado","dabar"], b = ["brazil"]

Output:

false

Explanation:

Alice must start the game by playing the word "avokado" since it's her smallest word, then Bob plays his only word, "brazil", which he can play because its first letter, 'b', is the letter after Alice's word's first letter, 'a'. Alice can't play a word since the first letter of the only word left is not equal to 'b' or the letter after 'b', 'c'. So, Alice loses, and the game ends.

Example 2:

Input:

a = ["ananas","atlas","banana"], b = ["albatros","cikla","nogomet"]

Output:

true

Explanation:

Alice must start the game by playing the word "ananas". Bob can't play a word since the only word he has that starts with the letter 'a' or 'b' is "albatros", which is smaller than Alice's word. So Alice wins, and the game ends.

Example 3:

Input:

a = ["hrvatska","zastava"], b = ["bijeli","galeb"]

Output:

true

Explanation:

Alice must start the game by playing the word "hrvatska". Bob can't play a word since the first letter of both of his words are smaller than the first letter of Alice's word, 'h'. So Alice wins, and the game ends.

Constraints:

1 <= a.length, b.length <= 10

5

a[i]

and

b[i]

consist only of lowercase English letters.

a

and

b

are

lexicographically sorted

.

All the words in

a

and

b

combined are

distinct

.

The sum of the lengths of all the words in

a

and

b

combined does not exceed

10

6

.

Code Snippets

C++:

```
class Solution {
public:
bool canAliceWin(vector<string>& a, vector<string>& b) {

}
};
```

Java:

```
class Solution {
public boolean canAliceWin(String[] a, String[] b) {
```

```
}
}
```

Python3:

```
class Solution:
def canAliceWin(self, a: List[str], b: List[str]) -> bool:
```

Python:

```
class Solution(object):
def canAliceWin(self, a, b):
"""
:type a: List[str]
:type b: List[str]
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {string[]} a
* @param {string[]} b
* @return {boolean}
*/
var canAliceWin = function(a, b) {

};
```

TypeScript:

```
function canAliceWin(a: string[], b: string[]): boolean {

};
```

C#:

```
public class Solution {
public bool CanAliceWin(string[] a, string[] b) {

}
```

```
}
```

C:

```
bool canAliceWin(char** a, int aSize, char** b, int bSize) {

}
```

Go:

```
func canAliceWin(a []string, b []string) bool {

}
```

Kotlin:

```
class Solution {
fun canAliceWin(a: Array<String>, b: Array<String>): Boolean {

}
}
```

Swift:

```
class Solution {
func canAliceWin(_ a: [String], _ b: [String]) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn can_alice_win(a: Vec<String>, b: Vec<String>) -> bool {

}
}
```

Ruby:

```
# @param {String[]} a
# @param {String[]} b
```

```
# @return {Boolean}
def can_alice_win(a, b)

end
```

PHP:

```
class Solution {

/**
* @param String[] $a
* @param String[] $b
* @return Boolean
*/
function canAliceWin($a, $b) {

}
}
```

Dart:

```
class Solution {
bool canAliceWin(List<String> a, List<String> b) {

}
}
```

Scala:

```
object Solution {
def canAliceWin(a: Array[String], b: Array[String]): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec can_alice_win(a :: [String.t], b :: [String.t]) :: boolean
def can_alice_win(a, b) do

end
```

```
end
```

Erlang:

```
-spec can_alice_win(A :: [unicode:unicode_binary()], B ::
[unicode:unicode_binary()]) -> boolean().
can_alice_win(A, B) ->
.
```

Racket:

```
(define/contract (can-alice-win a b)
(-> (listof string?) (listof string?) boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: The Wording Game
* Difficulty: Hard
* Tags: array, string, graph, greedy, math, sort
*
* 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:
bool canAliceWin(vector<string>& a, vector<string>& b) {

}
};
```

Java Solution:

```
/**
* Problem: The Wording Game
```

```
* Difficulty: Hard
* Tags: array, string, graph, greedy, math, sort
*
* 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 boolean canAliceWin(String[] a, String[] b) {

}
}
```

Python3 Solution:

```
"""
Problem: The Wording Game
Difficulty: Hard
Tags: array, string, graph, greedy, math, sort

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 canAliceWin(self, a: List[str], b: List[str]) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def canAliceWin(self, a, b):
"""
:type a: List[str]
:type b: List[str]
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: The Wording Game
* Difficulty: Hard
* Tags: array, string, graph, greedy, math, sort
*
* 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[]} a
* @param {string[]} b
* @return {boolean}
*/
var canAliceWin = function(a, b) {

};
```

TypeScript Solution:

```
/**
* Problem: The Wording Game
* Difficulty: Hard
* Tags: array, string, graph, greedy, math, sort
*
* 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 canAliceWin(a: string[], b: string[]): boolean {

};
```

C# Solution:

```
/*
* Problem: The Wording Game
* Difficulty: Hard
* Tags: array, string, graph, greedy, math, sort
```

```
*
* 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 bool CanAliceWin(string[] a, string[] b) {

}
}
```

C Solution:

```
/*
* Problem: The Wording Game
* Difficulty: Hard
* Tags: array, string, graph, greedy, math, sort
*
* 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
*/

bool canAliceWin(char** a, int aSize, char** b, int bSize) {

}
```

Go Solution:

```
// Problem: The Wording Game
// Difficulty: Hard
// Tags: array, string, graph, greedy, math, sort
//
// 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 canAliceWin(a []string, b []string) bool {

}
```

Kotlin Solution:

```
class Solution {
fun canAliceWin(a: Array<String>, b: Array<String>): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func canAliceWin(_ a: [String], _ b: [String]) -> Bool {

}
}
```

Rust Solution:

```
// Problem: The Wording Game
// Difficulty: Hard
// Tags: array, string, graph, greedy, math, sort
//
// 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 can_alice_win(a: Vec<String>, b: Vec<String>) -> bool {

}
}
```

Ruby Solution:

```
# @param {String[]} a
# @param {String[]} b
# @return {Boolean}
def can_alice_win(a, b)

end
```

PHP Solution:

```
class Solution {

/**
* @param String[] $a
* @param String[] $b
* @return Boolean
*/
function canAliceWin($a, $b) {

}
}
```

Dart Solution:

```
class Solution {
bool canAliceWin(List<String> a, List<String> b) {

}
}
```

Scala Solution:

```
object Solution {
def canAliceWin(a: Array[String], b: Array[String]): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec can_alice_win(a :: [String.t], b :: [String.t]) :: boolean
def can_alice_win(a, b) do

end
end
```

Erlang Solution:

```
-spec can_alice_win(A :: [unicode:unicode_binary()], B ::
[unicode:unicode_binary()]) -> boolean().
can_alice_win(A, B) ->
.
```

Racket Solution:

```
(define/contract (can-alice-win a b)
(-> (listof string?) (listof string?) boolean?)
)
```