Problem 2108: Find First Palindromic String in the Array

Problem Information

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

Problem Description

Given an array of strings

words

, return

the first

palindromic

string in the array

. If there is no such string, return

an

empty string

""

.

A string is

palindromic

if it reads the same forward and backward.

Example 1:

Input:

words = ["abc","car","ada","racecar","cool"]

Output:

"ada"

Explanation:

The first string that is palindromic is "ada". Note that "racecar" is also palindromic, but it is not the first.

Example 2:

Input:

words = ["notapalindrome","racecar"]

Output:

"racecar"

Explanation:

The first and only string that is palindromic is "racecar".

Example 3:

Input:

words = ["def","ghi"]

Output:

""

Explanation:

There are no palindromic strings, so the empty string is returned.

Constraints:

1 <= words.length <= 100

1 <= words[i].length <= 100

words[i]

consists only of lowercase English letters.

Code Snippets

C++:

```
class Solution {
public:
string firstPalindrome(vector<string>& words) {

}
};
```

Java:

```
class Solution {
public String firstPalindrome(String[] words) {

}
}
```

Python3:

```
class Solution:
def firstPalindrome(self, words: List[str]) -> str:
```

Python:

```
class Solution(object):
def firstPalindrome(self, words):
"""
:type words: List[str]
:rtype: str
"""
```

JavaScript:

```
/**
* @param {string[]} words
* @return {string}
*/
var firstPalindrome = function(words) {

};
```

TypeScript:

```
function firstPalindrome(words: string[]): string {

};
```

C#:

```
public class Solution {
public string FirstPalindrome(string[] words) {

}
}
```

C:

```
char* firstPalindrome(char** words, int wordsSize) {

}
```

Go:

```
func firstPalindrome(words []string) string {
}
```

Kotlin:

```
class Solution {
fun firstPalindrome(words: Array<String>): String {

}
}
```

Swift:

```
class Solution {
func firstPalindrome(_ words: [String]) -> String {

}
}
```

Rust:

```
impl Solution {
pub fn first_palindrome(words: Vec<String>) -> String {

}
}
```

Ruby:

```
# @param {String[]} words
# @return {String}
def first_palindrome(words)

end
```

PHP:

```
class Solution {

/**
* @param String[] $words
* @return String
*/
function firstPalindrome($words) {

}
}
```

Dart:

```
class Solution {
String firstPalindrome(List<String> words) {

}
}
```

Scala:

```
object Solution {
def firstPalindrome(words: Array[String]): String = {

}
}
```

Elixir:

```
defmodule Solution do
@spec first_palindrome(words :: [String.t]) :: String.t
def first_palindrome(words) do

end
end
```

Erlang:

```
-spec first_palindrome(Words :: [unicode:unicode_binary()]) ->
unicode:unicode_binary().
first_palindrome(Words) ->
.
```

Racket:

```
(define/contract (first-palindrome words)
(-> (listof string?) string?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Find First Palindromic String in the Array
* Difficulty: Easy
* Tags: array, string
*
* 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:
string firstPalindrome(vector<string>& words) {

}
};
```

Java Solution:

```
/**
* Problem: Find First Palindromic String in the Array
* Difficulty: Easy
* Tags: array, string
*
* 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 String firstPalindrome(String[] words) {

}
}
```

Python3 Solution:

```
"""
Problem: Find First Palindromic String in the Array
Difficulty: Easy
Tags: array, string

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 firstPalindrome(self, words: List[str]) -> str:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def firstPalindrome(self, words):
"""
:type words: List[str]
:rtype: str
"""
```

JavaScript Solution:

```
/**
* Problem: Find First Palindromic String in the Array
* Difficulty: Easy
* Tags: array, string
*
* 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[]} words
* @return {string}
*/
var firstPalindrome = function(words) {
};
```

TypeScript Solution:

```
/**
* Problem: Find First Palindromic String in the Array
* Difficulty: Easy
* Tags: array, string
*
* 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 firstPalindrome(words: string[]): string {

};
```

C# Solution:

```
/*
* Problem: Find First Palindromic String in the Array
* Difficulty: Easy
* Tags: array, string
*
* 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 string FirstPalindrome(string[] words) {

}
}
```

C Solution:

```
/*
* Problem: Find First Palindromic String in the Array
* Difficulty: Easy
* Tags: array, string
*
* 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
*/

char* firstPalindrome(char** words, int wordsSize) {

}
```

Go Solution:

```
// Problem: Find First Palindromic String in the Array
// Difficulty: Easy
// Tags: array, string
//
// 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 firstPalindrome(words []string) string {

}
```

Kotlin Solution:

```
class Solution {
fun firstPalindrome(words: Array<String>): String {

}
}
```

Swift Solution:

```
class Solution {
func firstPalindrome(_ words: [String]) -> String {

}
}
```

Rust Solution:

```
// Problem: Find First Palindromic String in the Array
// Difficulty: Easy
// Tags: array, string
//
// 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 first_palindrome(words: Vec<String>) -> String {


}
}
```

Ruby Solution:

```
# @param {String[]} words
# @return {String}
def first_palindrome(words)

end
```

PHP Solution:

```
class Solution {

/**
* @param String[] $words
* @return String
*/
function firstPalindrome($words) {

}
}
```

Dart Solution:

```
class Solution {
String firstPalindrome(List<String> words) {
}
}
```

Scala Solution:

```
object Solution {
def firstPalindrome(words: Array[String]): String = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec first_palindrome(words :: [String.t]) :: String.t
def first_palindrome(words) do

end
end
```

Erlang Solution:

```
-spec first_palindrome(Words :: [unicode:unicode_binary()]) ->
unicode:unicode_binary().
first_palindrome(Words) ->
.
```

Racket Solution:

```
(define/contract (first-palindrome words)
(-> (listof string?) string?)
)
```