Problem 1408: String Matching in an Array

Problem Information

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

Problem Description

Given an array of string

words

, return all strings in

words

that are a

substring

of another word. You can return the answer in

any order

.

Example 1:

Input:

words = ["mass","as","hero","superhero"]

Output:

["as","hero"]

Explanation:

"as" is substring of "mass" and "hero" is substring of "superhero". ["hero","as"] is also a valid answer.

Example 2:

Input:

words = ["leetcode","et","code"]

Output:

["et","code"]

Explanation:

"et", "code" are substring of "leetcode".

Example 3:

Input:

words = ["blue","green","bu"]

Output:

[]

Explanation:

No string of words is substring of another string.

Constraints:

1 <= words.length <= 100

1 <= words[i].length <= 30

words[i]

contains only lowercase English letters.

All the strings of

words

are

unique

.

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public List<String> stringMatching(String[] words) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public IList<string> StringMatching(string[] words) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** stringMatching(char** words, int wordsSize, int* returnSize) {

}
```

Go:

```
func stringMatching(words []string) []string {

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

/**
* @param String[] $words
* @return String[]
```

```
*/
function stringMatching($words) {

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

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

```
)
```

Solutions

C++ Solution:

```
/*
* Problem: String Matching in an Array
* Difficulty: Easy
* Tags: array, string, tree
*
* 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<string> stringMatching(vector<string>& words) {

}
};
```

Java Solution:

```
/**
* Problem: String Matching in an Array
* Difficulty: Easy
* Tags: array, string, tree
*
* 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<String> stringMatching(String[] words) {

}
}
```

Python3 Solution:

```
"""
Problem: String Matching in an Array
Difficulty: Easy
Tags: array, string, tree

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: String Matching in an Array
* Difficulty: Easy
* Tags: array, string, tree
*
* 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[]} words
* @return {string[]}
*/
```

```
var stringMatching = function(words) {

};
```

TypeScript Solution:

```
/**
* Problem: String Matching in an Array
* Difficulty: Easy
* Tags: array, string, tree
*
* 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 stringMatching(words: string[]): string[] {

};
```

C# Solution:

```
/*
* Problem: String Matching in an Array
* Difficulty: Easy
* Tags: array, string, tree
*
* 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<string> StringMatching(string[] words) {

}
}
```

C Solution:

```
/*
* Problem: String Matching in an Array
* Difficulty: Easy
* Tags: array, string, tree
*
* 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().
*/
char** stringMatching(char** words, int wordsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: String Matching in an Array
// Difficulty: Easy
// Tags: array, string, tree
//
// 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 stringMatching(words []string) []string {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: String Matching in an Array
// Difficulty: Easy
// Tags: array, string, tree
//
// 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 string_matching(words: Vec<String>) -> Vec<String> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

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