Problem 1772: Sort Features by Popularity

Problem Information

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

Problem Description

You are given a string array

features

where

features[i]

is a single word that represents the name of a feature of the latest product you are working on. You have made a survey where users have reported which features they like. You are given a string array

responses

, where each

responses[i]

is a string containing space-separated words.

The

popularity

of a feature is the number of

responses[i]

that contain the feature. You want to sort the features in non-increasing order by their popularity. If two features have the same popularity, order them by their original index in

features

. Notice that one response could contain the same feature multiple times; this feature is only counted once in its popularity.

Return

the features in sorted order.

Example 1:

Input:

features = ["cooler","lock","touch"], responses = ["i like cooler cooler","lock touch cool","locker like touch"]

Output:

["touch","cooler","lock"]

Explanation:

appearances("cooler") = 1, appearances("lock") = 1, appearances("touch") = 2. Since "cooler" and "lock" both had 1 appearance, "cooler" comes first because "cooler" came first in the features array.

Example 2:

Input:

features = ["a","aa","b","c"], responses = ["a","a aa","a a a a a","b a"]

Output:

["a","aa","b","c"]

Constraints:

1 <= features.length <= 10

4

1 <= features[i].length <= 10

features

contains no duplicates.

features[i]

consists of lowercase letters.

1 <= responses.length <= 10

2

1 <= responses[i].length <= 10

3

responses[i]

consists of lowercase letters and spaces.

responses[i]

contains no two consecutive spaces.

responses[i]

has no leading or trailing spaces.

Code Snippets

C++:

```
class Solution {
public:
vector<string> sortFeatures(vector<string>& features, vector<string>&
responses) {

}
};
```

Java:

```
class Solution {
public String[] sortFeatures(String[] features, String[] responses) {

}
}
```

Python3:

```
class Solution:
def sortFeatures(self, features: List[str], responses: List[str]) ->
List[str]:
```

Python:

```
class Solution(object):
def sortFeatures(self, features, responses):
"""
:type features: List[str]
:type responses: List[str]
:rtype: List[str]
"""
```

JavaScript:

```
/**
* @param {string[]} features
* @param {string[]} responses
* @return {string[]}
*/
```

```
var sortFeatures = function(features, responses) {

};
```

TypeScript:

```
function sortFeatures(features: string[], responses: string[]): string[] {

};
```

C#:

```
public class Solution {
public string[] SortFeatures(string[] features, string[] responses) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** sortFeatures(char** features, int featuresSize, char** responses, int
responsesSize, int* returnSize) {

}
```

Go:

```
func sortFeatures(features []string, responses []string) []string {

}
```

Kotlin:

```
class Solution {
fun sortFeatures(features: Array<String>, responses: Array<String>):
Array<String> {

}
}
```

Swift:

```
class Solution {
func sortFeatures(_ features: [String], _ responses: [String]) -> [String] {

}
}
```

Rust:

```
impl Solution {
pub fn sort_features(features: Vec<String>, responses: Vec<String>) ->
Vec<String> {

}
}
```

Ruby:

```
# @param {String[]} features
# @param {String[]} responses
# @return {String[]}
def sort_features(features, responses)

end
```

PHP:

```
class Solution {

/**
* @param String[] $features
* @param String[] $responses
* @return String[]
*/
function sortFeatures($features, $responses) {

}
}
```

Dart:

```
class Solution {
List<String> sortFeatures(List<String> features, List<String> responses) {

}
}
```

Scala:

```
object Solution {
def sortFeatures(features: Array[String], responses: Array[String]):
Array[String] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec sort_features(features :: [String.t], responses :: [String.t]) ::
[String.t]
def sort_features(features, responses) do

end
end
```

Erlang:

```
-spec sort_features(Features :: [unicode:unicode_binary()], Responses ::
[unicode:unicode_binary()]) -> [unicode:unicode_binary()].
sort_features(Features, Responses) ->
.
```

Racket:

```
(define/contract (sort-features features responses)
(-> (listof string?) (listof string?) (listof string?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Sort Features by Popularity
* Difficulty: Medium
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
vector<string> sortFeatures(vector<string>& features, vector<string>&
responses) {

}
};
```

Java Solution:

```
/**
* Problem: Sort Features by Popularity
* Difficulty: Medium
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public String[] sortFeatures(String[] features, String[] responses) {

}
}
```

Python3 Solution:

```
"""
Problem: Sort Features by Popularity
Difficulty: Medium
Tags: array, string, hash, sort
```

```
Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def sortFeatures(self, features: List[str], responses: List[str]) ->
List[str]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def sortFeatures(self, features, responses):
"""
:type features: List[str]
:type responses: List[str]
:rtype: List[str]
"""
```

JavaScript Solution:

```
/**
* Problem: Sort Features by Popularity
* Difficulty: Medium
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {string[]} features
* @param {string[]} responses
* @return {string[]}
*/
var sortFeatures = function(features, responses) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Sort Features by Popularity
* Difficulty: Medium
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function sortFeatures(features: string[], responses: string[]): string[] {

};
```

C# Solution:

```
/*
* Problem: Sort Features by Popularity
* Difficulty: Medium
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public string[] SortFeatures(string[] features, string[] responses) {

}
}
```

C Solution:

```
/*
* Problem: Sort Features by Popularity
* Difficulty: Medium
```

```
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
char** sortFeatures(char** features, int featuresSize, char** responses, int
responsesSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Sort Features by Popularity
// Difficulty: Medium
// Tags: array, string, hash, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func sortFeatures(features []string, responses []string) []string {

}
```

Kotlin Solution:

```
class Solution {
fun sortFeatures(features: Array<String>, responses: Array<String>):
Array<String> {

}
}
```

Swift Solution:

```
class Solution {
func sortFeatures(_ features: [String], _ responses: [String]) -> [String] {

}
}
```

Rust Solution:

```
// Problem: Sort Features by Popularity
// Difficulty: Medium
// Tags: array, string, hash, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn sort_features(features: Vec<String>, responses: Vec<String>) ->
Vec<String> {

}
}
```

Ruby Solution:

```
# @param {String[]} features
# @param {String[]} responses
# @return {String[]}
def sort_features(features, responses)

end
```

PHP Solution:

```
class Solution {

/**
* @param String[] $features
* @param String[] $responses
* @return String[]
*/
function sortFeatures($features, $responses) {
```

```
}
}
```

Dart Solution:

```
class Solution {
List<String> sortFeatures(List<String> features, List<String> responses) {

}
}
```

Scala Solution:

```
object Solution {
def sortFeatures(features: Array[String], responses: Array[String]):
Array[String] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec sort_features(features :: [String.t], responses :: [String.t]) ::
[String.t]
def sort_features(features, responses) do

end
end
```

Erlang Solution:

```
-spec sort_features(Features :: [unicode:unicode_binary()], Responses ::
[unicode:unicode_binary()]) -> [unicode:unicode_binary()].
sort_features(Features, Responses) ->
.
```

Racket Solution:

```
(define/contract (sort-features features responses)
(-> (listof string?) (listof string?) (listof string?))
)
```