Problem 3668: Restore Finishing Order

Problem Information

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

Problem Description

You are given an integer array

order

of length

n

and an integer array

friends

.

order

contains every integer from 1 to

n

exactly once

, representing the IDs of the participants of a race in their

finishing

order.

friends

contains the IDs of your friends in the race

sorted

in strictly increasing order. Each ID in friends is guaranteed to appear in the

order

array.

Return an array containing your friends' IDs in their

finishing

order.

Example 1:

Input:

order = [3,1,2,5,4], friends = [1,3,4]

Output:

[3,1,4]

Explanation:

The finishing order is

[

3

,

1

, 2, 5,

4

]

. Therefore, the finishing order of your friends is

[3, 1, 4]

.

Example 2:

Input:

order = [1,4,5,3,2], friends = [2,5]

Output:

[5,2]

Explanation:

The finishing order is

[1, 4,

5

, 3,

2

]

. Therefore, the finishing order of your friends is

[5, 2]

.

Constraints:

1 <= n == order.length <= 100

order

contains every integer from 1 to

n

exactly once

1 <= friends.length <= min(8, n)

1 <= friends[i] <= n

friends

is strictly increasing

Code Snippets

C++:

```
class Solution {
public:
vector<int> recoverOrder(vector<int>& order, vector<int>& friends) {

}
};
```

Java:

```
class Solution {
public int[] recoverOrder(int[] order, int[] friends) {

}
}
```

Python3:

```
class Solution:
def recoverOrder(self, order: List[int], friends: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def recoverOrder(self, order, friends):
"""
:type order: List[int]
:type friends: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} order
* @param {number[]} friends
* @return {number[]}
*/
var recoverOrder = function(order, friends) {

};
```

TypeScript:

```
function recoverOrder(order: number[], friends: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] RecoverOrder(int[] order, int[] friends) {
```

```
}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* recoverOrder(int* order, int orderSize, int* friends, int friendsSize,
int* returnSize) {

}
```

Go:

```
func recoverOrder(order []int, friends []int) []int {

}
```

Kotlin:

```
class Solution {
fun recoverOrder(order: IntArray, friends: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func recoverOrder(_ order: [Int], _ friends: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn recover_order(order: Vec<i32>, friends: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} order
# @param {Integer[]} friends
# @return {Integer[]}
def recover_order(order, friends)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $order
* @param Integer[] $friends
* @return Integer[]
*/
function recoverOrder($order, $friends) {

}
}
```

Dart:

```
class Solution {
List<int> recoverOrder(List<int> order, List<int> friends) {

}
}
```

Scala:

```
object Solution {
def recoverOrder(order: Array[Int], friends: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec recover_order(order :: [integer], friends :: [integer]) :: [integer]
```

```
def recover_order(order, friends) do

end
end
```

Erlang:

```
-spec recover_order(Order :: [integer()], Friends :: [integer()]) ->
[integer()].
recover_order(Order, Friends) ->
.
```

Racket:

```
(define/contract (recover-order order friends)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Restore Finishing Order
* Difficulty: Easy
* Tags: array, 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<int> recoverOrder(vector<int>& order, vector<int>& friends) {

}
};
```

Java Solution:

```
/**
* Problem: Restore Finishing Order
* Difficulty: Easy
* Tags: array, 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 int[] recoverOrder(int[] order, int[] friends) {

}
}
```

Python3 Solution:

```
"""
Problem: Restore Finishing Order
Difficulty: Easy
Tags: array, 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 recoverOrder(self, order: List[int], friends: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def recoverOrder(self, order, friends):
"""
:type order: List[int]
:type friends: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Restore Finishing Order
* Difficulty: Easy
* Tags: array, 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 {number[]} order
* @param {number[]} friends
* @return {number[]}
*/
var recoverOrder = function(order, friends) {

};
```

TypeScript Solution:

```
/**
* Problem: Restore Finishing Order
* Difficulty: Easy
* Tags: array, 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 recoverOrder(order: number[], friends: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Restore Finishing Order
* Difficulty: Easy
```

```
* Tags: array, 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 int[] RecoverOrder(int[] order, int[] friends) {

}
}
```

C Solution:

```
/*
* Problem: Restore Finishing Order
* Difficulty: Easy
* Tags: array, 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().
*/
int* recoverOrder(int* order, int orderSize, int* friends, int friendsSize,
int* returnSize) {

}
```

Go Solution:

```
// Problem: Restore Finishing Order
// Difficulty: Easy
// Tags: array, 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 recoverOrder(order []int, friends []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun recoverOrder(order: IntArray, friends: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func recoverOrder(_ order: [Int], _ friends: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Restore Finishing Order
// Difficulty: Easy
// Tags: array, 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 recover_order(order: Vec<i32>, friends: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} order
# @param {Integer[]} friends
# @return {Integer[]}
def recover_order(order, friends)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $order
* @param Integer[] $friends
* @return Integer[]
*/
function recoverOrder($order, $friends) {

}
}
```

Dart Solution:

```
class Solution {
List<int> recoverOrder(List<int> order, List<int> friends) {

}
}
```

Scala Solution:

```
object Solution {
def recoverOrder(order: Array[Int], friends: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec recover_order(order :: [integer], friends :: [integer]) :: [integer]
def recover_order(order, friends) do
```

```
end
end
```

Erlang Solution:

```
-spec recover_order(Order :: [integer()], Friends :: [integer()]) ->
[integer()].
recover_order(Order, Friends) ->
.
```

Racket Solution:

```
(define/contract (recover-order order friends)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```