Problem 2561: Rearranging Fruits

Problem Information

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

Problem Description

You have two fruit baskets containing

n

fruits each. You are given two

0-indexed

integer arrays

basket1

and

basket2

representing the cost of fruit in each basket. You want to make both baskets

equal

. To do so, you can use the following operation as many times as you want:

Choose two indices

i

and

j

, and swap the

i

th

fruit of

basket1

with the

j

th

fruit of

basket2

.

The cost of the swap is

min(basket1[i], basket2[j])

.

Two baskets are considered equal if sorting them according to the fruit cost makes them exactly the same baskets.

Return

the minimum cost to make both the baskets equal or

-1

if impossible.

Example 1:

Input:

basket1 = [4,2,2,2], basket2 = [1,4,1,2]

Output:

1

Explanation:

Swap index 1 of basket1 with index 0 of basket2, which has cost 1. Now basket1 = [4,1,2,2] and basket2 = [2,4,1,2]. Rearranging both the arrays makes them equal.

Example 2:

Input:

basket1 = [2,3,4,1], basket2 = [3,2,5,1]

Output:

-1

Explanation:

It can be shown that it is impossible to make both the baskets equal.

Constraints:

basket1.length == basket2.length

1 <= basket1.length <= 10

5

1 <= basket1[i], basket2[i] <= 10

9

Code Snippets

C++:

```
class Solution {
public:
long long minCost(vector<int>& basket1, vector<int>& basket2) {

}
};
```

Java:

```
class Solution {
public long minCost(int[] basket1, int[] basket2) {

}
}
```

Python3:

```
class Solution:
def minCost(self, basket1: List[int], basket2: List[int]) -> int:
```

Python:

```
class Solution(object):
def minCost(self, basket1, basket2):
"""
:type basket1: List[int]
:type basket2: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} basket1
* @param {number[]} basket2
* @return {number}
*/
var minCost = function(basket1, basket2) {

};
```

TypeScript:

```
function minCost(basket1: number[], basket2: number[]): number {

};
```

C#:

```
public class Solution {
public long MinCost(int[] basket1, int[] basket2) {

}
}
```

C:

```
long long minCost(int* basket1, int basket1Size, int* basket2, int
basket2Size) {

}
```

Go:

```
func minCost(basket1 []int, basket2 []int) int64 {

}
```

Kotlin:

```
class Solution {
fun minCost(basket1: IntArray, basket2: IntArray): Long {

}
}
```

Swift:

```
class Solution {
func minCost(_ basket1: [Int], _ basket2: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn min_cost(basket1: Vec<i32>, basket2: Vec<i32>) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[]} basket1
# @param {Integer[]} basket2
# @return {Integer}
def min_cost(basket1, basket2)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $basket1
* @param Integer[] $basket2
* @return Integer
*/
function minCost($basket1, $basket2) {

}
}
```

Dart:

```
class Solution {
int minCost(List<int> basket1, List<int> basket2) {



}
}
```

Scala:

```
object Solution {
def minCost(basket1: Array[Int], basket2: Array[Int]): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_cost(basket1 :: [integer], basket2 :: [integer]) :: integer
def min_cost(basket1, basket2) do

end
end
```

Erlang:

```
-spec min_cost(Basket1 :: [integer()], Basket2 :: [integer()]) -> integer().
min_cost(Basket1, Basket2) ->
.
```

Racket:

```
(define/contract (min-cost basket1 basket2)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Rearranging Fruits
* Difficulty: Hard
* Tags: array, greedy, 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:
long long minCost(vector<int>& basket1, vector<int>& basket2) {

}
};
```

Java Solution:

```
/**
* Problem: Rearranging Fruits
* Difficulty: Hard
* Tags: array, greedy, 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 long minCost(int[] basket1, int[] basket2) {

}
}
```

Python3 Solution:

```
"""
Problem: Rearranging Fruits
Difficulty: Hard
Tags: array, greedy, 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 minCost(self, basket1: List[int], basket2: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minCost(self, basket1, basket2):
"""
:type basket1: List[int]
:type basket2: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Rearranging Fruits
* Difficulty: Hard
* Tags: array, greedy, 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[]} basket1
* @param {number[]} basket2
* @return {number}
*/
var minCost = function(basket1, basket2) {

};
```

TypeScript Solution:

```
/**
* Problem: Rearranging Fruits
* Difficulty: Hard
* Tags: array, greedy, 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 minCost(basket1: number[], basket2: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Rearranging Fruits
* Difficulty: Hard
* Tags: array, greedy, 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 long MinCost(int[] basket1, int[] basket2) {

}
}
```

C Solution:

```
/*
* Problem: Rearranging Fruits
* Difficulty: Hard
* Tags: array, greedy, 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
```

```
*/

long long minCost(int* basket1, int basket1Size, int* basket2, int
basket2Size) {

}
```

Go Solution:

```
// Problem: Rearranging Fruits
// Difficulty: Hard
// Tags: array, greedy, 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 minCost(basket1 []int, basket2 []int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun minCost(basket1: IntArray, basket2: IntArray): Long {

}
}
```

Swift Solution:

```
class Solution {
func minCost(_ basket1: [Int], _ basket2: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Rearranging Fruits
// Difficulty: Hard
// Tags: array, greedy, 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 min_cost(basket1: Vec<i32>, basket2: Vec<i32>) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} basket1
# @param {Integer[]} basket2
# @return {Integer}
def min_cost(basket1, basket2)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $basket1
* @param Integer[] $basket2
* @return Integer
*/
function minCost($basket1, $basket2) {

}
}
```

Dart Solution:

```
class Solution {
int minCost(List<int> basket1, List<int> basket2) {

}
}
```

Scala Solution:

```
object Solution {
def minCost(basket1: Array[Int], basket2: Array[Int]): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_cost(basket1 :: [integer], basket2 :: [integer]) :: integer
def min_cost(basket1, basket2) do

end
end
```

Erlang Solution:

```
-spec min_cost(Basket1 :: [integer()], Basket2 :: [integer()]) -> integer().
min_cost(Basket1, Basket2) ->
.
```

Racket Solution:

```
(define/contract (min-cost basket1 basket2)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```