Problem 3424: Minimum Cost to Make Arrays Identical

Problem Information

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

Problem Description

You are given two integer arrays

arr

and

brr

of length

n

, and an integer

k

. You can perform the following operations on

arr

any

number of times:

Split

arr

into

any

number of

contiguous

subarrays

and rearrange these subarrays in

any order

. This operation has a fixed cost of

k

.

Choose any element in

arr

and add or subtract a positive integer

x

to it. The cost of this operation is

x

.

Return the

minimum

total cost to make

arr

equal

to

brr

.

Example 1:

Input:

arr = [-7,9,5], brr = [7,-2,-5], k = 2

Output:

13

Explanation:

Split

arr

into two contiguous subarrays:

[-7]

and

[9, 5]

and rearrange them as

[9, 5, -7]

, with a cost of 2.

Subtract 2 from element

arr[0]

. The array becomes

[7, 5, -7]

. The cost of this operation is 2.

Subtract 7 from element

arr[1]

. The array becomes

[7, -2, -7]

. The cost of this operation is 7.

Add 2 to element

arr[2]

. The array becomes

[7, -2, -5]

. The cost of this operation is 2.

The total cost to make the arrays equal is

2 + 2 + 7 + 2 = 13

.

Example 2:

Input:

arr = [2,1], brr = [2,1], k = 0

Output:

0

Explanation:

Since the arrays are already equal, no operations are needed, and the total cost is 0.

Constraints:

1 <= arr.length == brr.length <= 10

5

0 <= k <= 2 * 10

10

-10

5

<= arr[i] <= 10

5

-10

5

<= brr[i] <= 10

5

Code Snippets

C++:

```
class Solution {
public:
long long minCost(vector<int>& arr, vector<int>& brr, long long k) {

}
};
```

Java:

```
class Solution {
public long minCost(int[] arr, int[] brr, long k) {

}
}
```

Python3:

```
class Solution:
def minCost(self, arr: List[int], brr: List[int], k: int) -> int:
```

Python:

```
class Solution(object):
def minCost(self, arr, brr, k):
"""
:type arr: List[int]
:type brr: List[int]
:type k: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} arr
* @param {number[]} brr
* @param {number} k
* @return {number}
*/
var minCost = function(arr, brr, k) {

};
```

TypeScript:

```
function minCost(arr: number[], brr: number[], k: number): number {

};
```

C#:

```
public class Solution {
public long MinCost(int[] arr, int[] brr, long k) {

}
}
```

C:

```
long long minCost(int* arr, int arrSize, int* brr, int brrSize, long long k)
{

}
```

Go:

```
func minCost(arr []int, brr []int, k int64) int64 {

}
```

Kotlin:

```
class Solution {
fun minCost(arr: IntArray, brr: IntArray, k: Long): Long {

}
}
```

Swift:

```
class Solution {
func minCost(_ arr: [Int], _ brr: [Int], _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn min_cost(arr: Vec<i32>, brr: Vec<i32>, k: i64) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[]} arr
# @param {Integer[]} brr
# @param {Integer} k
# @return {Integer}
def min_cost(arr, brr, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $arr
* @param Integer[] $brr
* @param Integer $k
* @return Integer
*/
function minCost($arr, $brr, $k) {

}
}
```

Dart:

```
class Solution {
int minCost(List<int> arr, List<int> brr, int k) {

}
}
```

Scala:

```
object Solution {
def minCost(arr: Array[Int], brr: Array[Int], k: Long): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_cost(arr :: [integer], brr :: [integer], k :: integer) :: integer
def min_cost(arr, brr, k) do

end
end
```

Erlang:

```
-spec min_cost(Arr :: [integer()], Brr :: [integer()], K :: integer()) ->
integer().
min_cost(Arr, Brr, K) ->
.
```

Racket:

```
(define/contract (min-cost arr brr k)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Cost to Make Arrays Identical
* Difficulty: Medium
* Tags: array, greedy, sort
*
* 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:
long long minCost(vector<int>& arr, vector<int>& brr, long long k) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Cost to Make Arrays Identical
* Difficulty: Medium
* Tags: array, greedy, sort
*
* 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 long minCost(int[] arr, int[] brr, long k) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Cost to Make Arrays Identical
Difficulty: Medium
Tags: array, greedy, sort
```

```
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 minCost(self, arr: List[int], brr: List[int], k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minCost(self, arr, brr, k):
"""
:type arr: List[int]
:type brr: List[int]
:type k: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Minimum Cost to Make Arrays Identical
* Difficulty: Medium
* Tags: array, greedy, sort
*
* 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 {number[]} arr
* @param {number[]} brr
* @param {number} k
* @return {number}
*/
var minCost = function(arr, brr, k) {
};
```

TypeScript Solution:

```
/**
* Problem: Minimum Cost to Make Arrays Identical
* Difficulty: Medium
* Tags: array, greedy, sort
*
* 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 minCost(arr: number[], brr: number[], k: number): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Cost to Make Arrays Identical
* Difficulty: Medium
* Tags: array, greedy, sort
*
* 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 long MinCost(int[] arr, int[] brr, long k) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Cost to Make Arrays Identical
* Difficulty: Medium
* Tags: array, greedy, sort
*
* 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
*/

long long minCost(int* arr, int arrSize, int* brr, int brrSize, long long k)
{

}
```

Go Solution:

```
// Problem: Minimum Cost to Make Arrays Identical
// Difficulty: Medium
// Tags: array, greedy, sort
//
// 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 minCost(arr []int, brr []int, k int64) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun minCost(arr: IntArray, brr: IntArray, k: Long): Long {

}
}
```

Swift Solution:

```
class Solution {
func minCost(_ arr: [Int], _ brr: [Int], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Cost to Make Arrays Identical
// Difficulty: Medium
// Tags: array, greedy, sort
//
// 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 min_cost(arr: Vec<i32>, brr: Vec<i32>, k: i64) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} arr
# @param {Integer[]} brr
# @param {Integer} k
# @return {Integer}
def min_cost(arr, brr, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $arr
* @param Integer[] $brr
* @param Integer $k
* @return Integer
*/
function minCost($arr, $brr, $k) {

}
}
```

Dart Solution:

```
class Solution {
int minCost(List<int> arr, List<int> brr, int k) {

}
}
```

Scala Solution:

```
object Solution {
def minCost(arr: Array[Int], brr: Array[Int], k: Long): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_cost(arr :: [integer], brr :: [integer], k :: integer) :: integer
def min_cost(arr, brr, k) do

end
end
```

Erlang Solution:

```
-spec min_cost(Arr :: [integer()], Brr :: [integer()], K :: integer()) ->
integer().
min_cost(Arr, Brr, K) ->
.
```

Racket Solution:

```
(define/contract (min-cost arr brr k)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?
exact-integer?)
)
```