Problem 1868: Product of Two Run-Length Encoded Arrays

Problem Information

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

Problem Description

Run-length encoding

is a compression algorithm that allows for an integer array

nums

with many segments of

consecutive repeated

numbers to be represented by a (generally smaller) 2D array

encoded

. Each

encoded[i] = [val

i

, freq

i

]

describes the

i

th

segment of repeated numbers in

nums

where

val

i

is the value that is repeated

freq

i

times.

For example,

nums = [1,1,1,2,2,2,2,2]

is represented by the

run-length encoded

array

encoded = [[1,3],[2,5]]

. Another way to read this is "three

1

's followed by five

2

's".

The

product

of two run-length encoded arrays

encoded1

and

encoded2

can be calculated using the following steps:

Expand

both

encoded1

and

encoded2

into the full arrays

nums1

and

nums2

respectively.

Create a new array

prodNums

of length

nums1.length

and set

prodNums[i] = nums1[i] * nums2[i]

.

Compress

prodNums

into a run-length encoded array and return it.

You are given two

run-length encoded

arrays

encoded1

and

encoded2

representing full arrays

nums1

and

nums2

respectively. Both

nums1

and

nums2

have the

same length

. Each

encoded1[i] = [val

i

, freq

i

]

describes the

i

th

segment of

nums1

, and each

encoded2[j] = [val

j

, freq

j

]

describes the

j

th

segment of

nums2

.

Return

the

product

of

encoded1

and

encoded2

.

Note:

Compression should be done such that the run-length encoded array has the

minimum

possible length.

Example 1:

Input:

encoded1 = [[1,3],[2,3]], encoded2 = [[6,3],[3,3]]

Output:

[[6,6]]

Explanation:

encoded1 expands to [1,1,1,2,2,2] and encoded2 expands to [6,6,6,3,3,3]. prodNums = [6,6,6,6,6,6], which is compressed into the run-length encoded array [[6,6]].

Example 2:

Input:

encoded1 = [[1,3],[2,1],[3,2]], encoded2 = [[2,3],[3,3]]

Output:

[[2,3],[6,1],[9,2]]

Explanation:

encoded1 expands to [1,1,1,2,3,3] and encoded2 expands to [2,2,2,3,3,3]. prodNums = [2,2,2,6,9,9], which is compressed into the run-length encoded array [[2,3],[6,1],[9,2]].

Constraints:

1 <= encoded1.length, encoded2.length <= 10

5

encoded1[i].length == 2

encoded2[j].length == 2

1 <= val

i

, freq

i

<= 10

4

for each

encoded1[i]

.

1 <= val

j

, freq

j

<= 10

4

for each

encoded2[j]

.

The full arrays that

encoded1

and

encoded2

represent are the same length.

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> findRLEArray(vector<vector<int>>& encoded1,
vector<vector<int>>& encoded2) {

}
};
```

Java:

```
class Solution {
public List<List<Integer>> findRLEArray(int[][] encoded1, int[][] encoded2) {

}
}
```

Python3:

```
class Solution:
def findRLEArray(self, encoded1: List[List[int]], encoded2: List[List[int]])
```

```
-> List[List[int]]:
```

Python:

```
class Solution(object):
def findRLEArray(self, encoded1, encoded2):
"""
:type encoded1: List[List[int]]
:type encoded2: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number[][]} encoded1
* @param {number[][]} encoded2
* @return {number[][]}
*/
var findRLEArray = function(encoded1, encoded2) {

};
```

TypeScript:

```
function findRLEArray(encoded1: number[][], encoded2: number[][]): number[][]
{

};
```

C#:

```
public class Solution {
public IList<IList<int>> FindRLEArray(int[][] encoded1, int[][] encoded2) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
```

```
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** findRLEArray(int** encoded1, int encoded1Size, int* encoded1ColSize,
int** encoded2, int encoded2Size, int* encoded2ColSize, int* returnSize,
int** returnColumnSizes) {

}
```

Go:

```
func findRLEArray(encoded1 [][]int, encoded2 [][]int) [][]int {

}
```

Kotlin:

```
class Solution {
fun findRLEArray(encoded1: Array<IntArray>, encoded2: Array<IntArray>):
List<List<Int>> {

}
}
```

Swift:

```
class Solution {
func findRLEArray(_ encoded1: [[Int]], _ encoded2: [[Int]]) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn find_rle_array(encoded1: Vec<Vec<i32>>, encoded2: Vec<Vec<i32>>) ->
Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer[][]} encoded1
# @param {Integer[][]} encoded2
# @return {Integer[][]}
def find_rle_array(encoded1, encoded2)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $encoded1
* @param Integer[][] $encoded2
* @return Integer[][]
*/
function findRLEArray($encoded1, $encoded2) {

}
}
```

Dart:

```
class Solution {
List<List<int>> findRLEArray(List<List<int>> encoded1, List<List<int>>
encoded2) {

}
}
```

Scala:

```
object Solution {
def findRLEArray(encoded1: Array[Array[Int]], encoded2: Array[Array[Int]]):
List[List[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec find_rle_array(encoded1 :: [[integer]], encoded2 :: [[integer]]) ::
```

```
[[integer]]
def find_rle_array(encoded1, encoded2) do

end
end
```

Erlang:

```
-spec find_rle_array(Encoded1 :: [[integer()]], Encoded2 :: [[integer()]]) ->
[[integer()]].
find_rle_array(Encoded1, Encoded2) ->
.
```

Racket:

```
(define/contract (find-rle-array encoded1 encoded2)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)) (listof
(listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Product of Two Run-Length Encoded Arrays
* Difficulty: Medium
* Tags: array
*
* 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:
vector<vector<int>> findRLEArray(vector<vector<int>>& encoded1,
vector<vector<int>>& encoded2) {

}
};
```

Java Solution:

```
/**
* Problem: Product of Two Run-Length Encoded Arrays
* Difficulty: Medium
* Tags: array
*
* 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 List<List<Integer>> findRLEArray(int[][] encoded1, int[][] encoded2) {

}
}
```

Python3 Solution:

```
"""
Problem: Product of Two Run-Length Encoded Arrays
Difficulty: Medium
Tags: array

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 findRLEArray(self, encoded1: List[List[int]], encoded2: List[List[int]])
-> List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def findRLEArray(self, encoded1, encoded2):
"""
:type encoded1: List[List[int]]
```

```
:type encoded2: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Product of Two Run-Length Encoded Arrays
* Difficulty: Medium
* Tags: array
*
* 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[][]} encoded1
* @param {number[][]} encoded2
* @return {number[][]}
*/
var findRLEArray = function(encoded1, encoded2) {

};
```

TypeScript Solution:

```
/**
* Problem: Product of Two Run-Length Encoded Arrays
* Difficulty: Medium
* Tags: array
*
* 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 findRLEArray(encoded1: number[][], encoded2: number[][]): number[][]
{

};
```

C# Solution:

```
/*
* Problem: Product of Two Run-Length Encoded Arrays
* Difficulty: Medium
* Tags: array
*
* 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 IList<IList<int>> FindRLEArray(int[][] encoded1, int[][] encoded2) {

}
}
```

C Solution:

```
/*
* Problem: Product of Two Run-Length Encoded Arrays
* Difficulty: Medium
* Tags: array
*
* 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
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** findRLEArray(int** encoded1, int encoded1Size, int* encoded1ColSize,
int** encoded2, int encoded2Size, int* encoded2ColSize, int* returnSize,
int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Product of Two Run-Length Encoded Arrays
// Difficulty: Medium
// Tags: array
//
// 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 findRLEArray(encoded1 [][]int, encoded2 [][]int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun findRLEArray(encoded1: Array<IntArray>, encoded2: Array<IntArray>):
List<List<Int>> {

}
}
```

Swift Solution:

```
class Solution {
func findRLEArray(_ encoded1: [[Int]], _ encoded2: [[Int]]) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: Product of Two Run-Length Encoded Arrays
// Difficulty: Medium
// Tags: array
//
// 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 find_rle_array(encoded1: Vec<Vec<i32>>, encoded2: Vec<Vec<i32>>) ->
Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} encoded1
# @param {Integer[][]} encoded2
# @return {Integer[][]}
def find_rle_array(encoded1, encoded2)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $encoded1
* @param Integer[][] $encoded2
* @return Integer[][]
*/
function findRLEArray($encoded1, $encoded2) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> findRLEArray(List<List<int>> encoded1, List<List<int>>
encoded2) {

}
}
```

Scala Solution:

```
object Solution {
def findRLEArray(encoded1: Array[Array[Int]], encoded2: Array[Array[Int]]):
List[List[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec find_rle_array(encoded1 :: [[integer]], encoded2 :: [[integer]]) ::
[[integer]]
def find_rle_array(encoded1, encoded2) do

end
end
```

Erlang Solution:

```
-spec find_rle_array(Encoded1 :: [[integer()]], Encoded2 :: [[integer()]]) ->
[[integer()]].
find_rle_array(Encoded1, Encoded2) ->
.
```

Racket Solution:

```
(define/contract (find-rle-array encoded1 encoded2)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)) (listof
(listof exact-integer?)))
)
```