Problem 2613: Beautiful Pairs

Problem Information

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

Problem Description

You are given two

0-indexed

integer arrays

nums1

and

nums2

of the same length. A pair of indices

(i,j)

is called

beautiful

if

|nums1[i] - nums1[j]| + |nums2[i] - nums2[j]|

is the smallest amongst all possible indices pairs where

$i < j$

.

Return

the beautiful pair. In the case that there are multiple beautiful pairs, return the lexicographically smallest pair.

Note that

$|x|$

denotes the absolute value of

x

.

A pair of indices

$(i$

1

$, j$

1

$)$

is lexicographically smaller than

$(i$

2

$, j$

2

)

if

i

1

< i

2

or

i

1

== i

2

and

j

1

< j

2

.

Example 1:

Input:

nums1 = [1,2,3,2,4], nums2 = [2,3,1,2,3]

Output:

[0,3]

Explanation:

Consider index 0 and index 3. The value of |nums1[i]-nums1[j]| + |nums2[i]-nums2[j]| is 1, which is the smallest value we can achieve.

Example 2:

Input:

nums1 = [1,2,4,3,2,5], nums2 = [1,4,2,3,5,1]

Output:

[1,4]

Explanation:

Consider index 1 and index 4. The value of |nums1[i]-nums1[j]| + |nums2[i]-nums2[j]| is 1, which is the smallest value we can achieve.

Constraints:

2 <= nums1.length, nums2.length <= 10

5

nums1.length == nums2.length

0 <= nums1

i

<= nums1.length

0 <= nums2

i

<= nums2.length

Code Snippets

C++:

```
class Solution {
public:
vector<int> beautifulPair(vector<int>& nums1, vector<int>& nums2) {

}
};
```

Java:

```
class Solution {
public int[] beautifulPair(int[] nums1, int[] nums2) {

}
}
```

Python3:

```
class Solution:
def beautifulPair(self, nums1: List[int], nums2: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def beautifulPair(self, nums1, nums2):
"""
:type nums1: List[int]
:type nums2: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} nums1
* @param {number[]} nums2
* @return {number[]}
*/
var beautifulPair = function(nums1, nums2) {

};
```

TypeScript:

```
function beautifulPair(nums1: number[], nums2: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] BeautifulPair(int[] nums1, int[] nums2) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* beautifulPair(int* nums1, int nums1Size, int* nums2, int nums2Size, int*
returnSize) {

}
```

Go:

```
func beautifulPair(nums1 []int, nums2 []int) []int {

}
```

Kotlin:

```
class Solution {
fun beautifulPair(nums1: IntArray, nums2: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func beautifulPair(_ nums1: [Int], _ nums2: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn beautiful_pair(nums1: Vec<i32>, nums2: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} nums1
# @param {Integer[]} nums2
# @return {Integer[]}
def beautiful_pair(nums1, nums2)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums1
* @param Integer[] $nums2
* @return Integer[]
*/
function beautifulPair($nums1, $nums2) {

}
```

```
}
```

Dart:

```
class Solution {
List<int> beautifulPair(List<int> nums1, List<int> nums2) {

}
}
```

Scala:

```
object Solution {
def beautifulPair(nums1: Array[Int], nums2: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec beautiful_pair(nums1 :: [integer], nums2 :: [integer]) :: [integer]
def beautiful_pair(nums1, nums2) do

end
end
```

Erlang:

```
-spec beautiful_pair(Nums1 :: [integer()], Nums2 :: [integer()]) ->
[integer()].
beautiful_pair(Nums1, Nums2) ->
.
```

Racket:

```
(define/contract (beautiful-pair nums1 nums2)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Beautiful Pairs
* Difficulty: Hard
* Tags: array, graph, math, 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:
vector<int> beautifulPair(vector<int>& nums1, vector<int>& nums2) {

}
};
```

Java Solution:

```
/**
* Problem: Beautiful Pairs
* Difficulty: Hard
* Tags: array, graph, math, 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 int[] beautifulPair(int[] nums1, int[] nums2) {

}
}
```

Python3 Solution:

```
"""
Problem: Beautiful Pairs
```

```
Difficulty: Hard
Tags: array, graph, math, 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 beautifulPair(self, nums1: List[int], nums2: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def beautifulPair(self, nums1, nums2):
"""
:type nums1: List[int]
:type nums2: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Beautiful Pairs
* Difficulty: Hard
* Tags: array, graph, math, 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[]} nums1
* @param {number[]} nums2
* @return {number[]}
*/
var beautifulPair = function(nums1, nums2) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Beautiful Pairs
* Difficulty: Hard
* Tags: array, graph, math, 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 beautifulPair(nums1: number[], nums2: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Beautiful Pairs
* Difficulty: Hard
* Tags: array, graph, math, 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 int[] BeautifulPair(int[] nums1, int[] nums2) {

}
}
```

C Solution:

```
/*
* Problem: Beautiful Pairs
```

```
* Difficulty: Hard
* Tags: array, graph, math, 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* beautifulPair(int* nums1, int nums1Size, int* nums2, int nums2Size, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Beautiful Pairs
// Difficulty: Hard
// Tags: array, graph, math, 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 beautifulPair(nums1 []int, nums2 []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun beautifulPair(nums1: IntArray, nums2: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func beautifulPair(_ nums1: [Int], _ nums2: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Beautiful Pairs
// Difficulty: Hard
// Tags: array, graph, math, 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 beautiful_pair(nums1: Vec<i32>, nums2: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums1
# @param {Integer[]} nums2
# @return {Integer[]}
def beautiful_pair(nums1, nums2)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums1
* @param Integer[] $nums2
* @return Integer[]
*/
function beautifulPair($nums1, $nums2) {
```

```
}
}
```

Dart Solution:

```
class Solution {
List<int> beautifulPair(List<int> nums1, List<int> nums2) {

}
}
```

Scala Solution:

```
object Solution {
def beautifulPair(nums1: Array[Int], nums2: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec beautiful_pair(nums1 :: [integer], nums2 :: [integer]) :: [integer]
def beautiful_pair(nums1, nums2) do

end
end
```

Erlang Solution:

```
-spec beautiful_pair(Nums1 :: [integer()], Nums2 :: [integer()]) ->
[integer()].
beautiful_pair(Nums1, Nums2) ->
.
```

Racket Solution:

```
(define/contract (beautiful-pair nums1 nums2)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```