Problem 593: Valid Square

Problem Information

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

Problem Description

Given the coordinates of four points in 2D space

p1

,

p2

,

p3

and

p4

, return

true

if the four points construct a square

.

The coordinate of a point

p

i

is represented as

[x

i

, y

i

]

. The input is

not

given in any order.

A

valid square

has four equal sides with positive length and four equal angles (90-degree angles).

Example 1:

Input:

p1 = [0,0], p2 = [1,1], p3 = [1,0], p4 = [0,1]

Output:

true

Example 2:

Input:

p1 = [0,0], p2 = [1,1], p3 = [1,0], p4 = [0,12]

Output:

false

Example 3:

Input:

p1 = [1,0], p2 = [-1,0], p3 = [0,1], p4 = [0,-1]

Output:

true

Constraints:

p1.length == p2.length == p3.length == p4.length == 2

-10

4

<= x

i

, y

i

<= 10

4

Code Snippets

C++:

```
class Solution {
public:
bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3,
vector<int>& p4) {

}
};
```

Java:

```
class Solution {
public boolean validSquare(int[] p1, int[] p2, int[] p3, int[] p4) {

}
}
```

Python3:

```
class Solution:
def validSquare(self, p1: List[int], p2: List[int], p3: List[int], p4:
List[int]) -> bool:
```

Python:

```
class Solution(object):
def validSquare(self, p1, p2, p3, p4):
"""
:type p1: List[int]
:type p2: List[int]
:type p3: List[int]
:type p4: List[int]
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number[]} p1
```

```
* @param {number[]} p2
* @param {number[]} p3
* @param {number[]} p4
* @return {boolean}
*/
var validSquare = function(p1, p2, p3, p4) {

};
```

TypeScript:

```
function validSquare(p1: number[], p2: number[], p3: number[], p4: number[]):
boolean {

};
```

C#:

```
public class Solution {
public bool ValidSquare(int[] p1, int[] p2, int[] p3, int[] p4) {

}
}
```

C:

```
bool validSquare(int* p1, int p1Size, int* p2, int p2Size, int* p3, int
p3Size, int* p4, int p4Size) {

}
```

Go:

```
func validSquare(p1 []int, p2 []int, p3 []int, p4 []int) bool {

}
```

Kotlin:

```
class Solution {
fun validSquare(p1: IntArray, p2: IntArray, p3: IntArray, p4: IntArray):
Boolean {
```

```
}
}
```

Swift:

```
class Solution {
func validSquare(_ p1: [Int], _ p2: [Int], _ p3: [Int], _ p4: [Int]) -> Bool
{

}
}
```

Rust:

```
impl Solution {
pub fn valid_square(p1: Vec<i32>, p2: Vec<i32>, p3: Vec<i32>, p4: Vec<i32>)
-> bool {

}
}
```

Ruby:

```
# @param {Integer[]} p1
# @param {Integer[]} p2
# @param {Integer[]} p3
# @param {Integer[]} p4
# @return {Boolean}
def valid_square(p1, p2, p3, p4)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $p1
* @param Integer[] $p2
* @param Integer[] $p3
* @param Integer[] $p4
```

```
* @return Boolean
*/
function validSquare($p1, $p2, $p3, $p4) {

}
}
```

Dart:

```
class Solution {
bool validSquare(List<int> p1, List<int> p2, List<int> p3, List<int> p4) {

}
}
```

Scala:

```
object Solution {
def validSquare(p1: Array[Int], p2: Array[Int], p3: Array[Int], p4:
Array[Int]): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec valid_square(p1 :: [integer], p2 :: [integer], p3 :: [integer], p4 ::
[integer]) :: boolean
def valid_square(p1, p2, p3, p4) do

end
end
```

Erlang:

```
-spec valid_square(P1 :: [integer()], P2 :: [integer()], P3 :: [integer()],
P4 :: [integer()]) -> boolean().
valid_square(P1, P2, P3, P4) ->
.
```

Racket:

```
(define/contract (valid-square p1 p2 p3 p4)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
(listof exact-integer?) boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Valid Square
* Difficulty: Medium
* Tags: math
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3,
vector<int>& p4) {

}
};
```

Java Solution:

```
/**
* Problem: Valid Square
* Difficulty: Medium
* Tags: math
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public boolean validSquare(int[] p1, int[] p2, int[] p3, int[] p4) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: Valid Square
Difficulty: Medium
Tags: math

Approach: Optimized algorithm based on problem constraints
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def validSquare(self, p1: List[int], p2: List[int], p3: List[int], p4:
List[int]) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def validSquare(self, p1, p2, p3, p4):
"""
:type p1: List[int]
:type p2: List[int]
:type p3: List[int]
:type p4: List[int]
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Valid Square
* Difficulty: Medium
* Tags: math
*
```

```
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[]} p1
* @param {number[]} p2
* @param {number[]} p3
* @param {number[]} p4
* @return {boolean}
*/
var validSquare = function(p1, p2, p3, p4) {

};
```

TypeScript Solution:

```
/**
* Problem: Valid Square
* Difficulty: Medium
* Tags: math
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

function validSquare(p1: number[], p2: number[], p3: number[], p4: number[]):
boolean {

};
```

C# Solution:

```
/*
* Problem: Valid Square
* Difficulty: Medium
* Tags: math
*
* Approach: Optimized algorithm based on problem constraints
```

```
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public bool ValidSquare(int[] p1, int[] p2, int[] p3, int[] p4) {

}
}
```

C Solution:

```
/*
* Problem: Valid Square
* Difficulty: Medium
* Tags: math
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

bool validSquare(int* p1, int p1Size, int* p2, int p2Size, int* p3, int
p3Size, int* p4, int p4Size) {

}
```

Go Solution:

```
// Problem: Valid Square
// Difficulty: Medium
// Tags: math
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

func validSquare(p1 []int, p2 []int, p3 []int, p4 []int) bool {

}
```

Kotlin Solution:

```
class Solution {
fun validSquare(p1: IntArray, p2: IntArray, p3: IntArray, p4: IntArray):
Boolean {

}
}
```

Swift Solution:

```
class Solution {
func validSquare(_ p1: [Int], _ p2: [Int], _ p3: [Int], _ p4: [Int]) -> Bool
{

}
}
```

Rust Solution:

```
// Problem: Valid Square
// Difficulty: Medium
// Tags: math
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn valid_square(p1: Vec<i32>, p2: Vec<i32>, p3: Vec<i32>, p4: Vec<i32>)
-> bool {

}
}
```

Ruby Solution:

```
# @param {Integer[]} p1
# @param {Integer[]} p2
# @param {Integer[]} p3
# @param {Integer[]} p4
# @return {Boolean}
```

```
def valid_square(p1, p2, p3, p4)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $p1
* @param Integer[] $p2
* @param Integer[] $p3
* @param Integer[] $p4
* @return Boolean
*/
function validSquare($p1, $p2, $p3, $p4) {

}
}
```

Dart Solution:

```
class Solution {
bool validSquare(List<int> p1, List<int> p2, List<int> p3, List<int> p4) {

}
}
```

Scala Solution:

```
object Solution {
def validSquare(p1: Array[Int], p2: Array[Int], p3: Array[Int], p4:
Array[Int]): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec valid_square(p1 :: [integer], p2 :: [integer], p3 :: [integer], p4 ::
```

```
[integer]) :: boolean
def valid_square(p1, p2, p3, p4) do

end
end
```

Erlang Solution:

```
-spec valid_square(P1 :: [integer()], P2 :: [integer()], P3 :: [integer()],
P4 :: [integer()]) -> boolean().
valid_square(P1, P2, P3, P4) ->
.
```

Racket Solution:

```
(define/contract (valid-square p1 p2 p3 p4)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
(listof exact-integer?) boolean?)
)
```