593. Valid Square

**Geometry** 

## **Problem Description**

Medium

each point, and the points are not necessarily given in any specific order. A valid square must have all four sides of equal length, and all four angles must be right angles (90 degrees). The task is to return true if the points form a valid square and false otherwise.

The problem requires determining whether four given points in a 2D plane form a square. The input includes the coordinates of

Intuition

To solve this problem, the intuition is to check every possible combination of three points out of the four to form two sides of a square and then calculate the lengths of these sides using the distance formula. The key idea is that for any three points which form a right angle, two sides squaring and adding them up should give the square of the third side (following the Pythagorean

theorem). This condition should be true for all combinations of three points. This ensures that four right angles are formed. Moreover, the lengths of adjacent sides should be equal (and greater than zero to avoid degenerate cases). The solution consists of a check function which performs the calculations and comparisons. It calculates the squared distances

between pairs of points (d1, d2, d3) and checks for the two conditions: equality of any two distances to make sure the sides are

equal, and the sum of these two equal distances should be equal to the third distance, ensuring a 90-degree angle between

them. It also checks that the distance is non-zero to avoid considering points that overlap as valid sides. The solution verifies these conditions for all combinations of three points out of the four: (p1, p2, p3), (p2, p3, p4), (p1, p3, p4), and (p1, p2, p4). If all combinations satisfy the condition, we conclude that the points can form a valid square.

Solution Approach

distances to validate the square's sides and angles. Here's how the approach unfolds step by step:

distance. This is essentially checking for a right triangle using the Pythagorean theorem.

points given. This step is crucial as it verifies the square property from all angles and sides.

The input is in the form of four points p1, p2, p3, and p4, each represented as a list of two integers.

check(p1, p2, p3) and check(p2, p3, p4) and check(p1, p3, p4) and check(p1, p2, p4)

described in the solution to determine if these points make up a valid square.

angle. Squared distances are calculated as follows:

• d1 (distance between p1 and p2):  $(1 - 1)^2 + (1 - 3)^2 = 0 + 4 = 4$ 

We will repeat this process for the other combinations of points:

how mathematical principles can be used to solve computational geometry problems.

# Check if the triangle formed by the three points is a right triangle

dist ab 2 == dist ac 2 == dist dist ab 2 + dist ac 2 == dist bc 2 == dist dist ab 2,

dist ac 2 == dist bc 2 and dist ac 2 + dist bc 2 == dist ab 2 and dist ac 2,

dist\_ab\_2 == dist\_bc\_2 and dist\_ab\_2 + dist\_bc\_2 == dist\_ac\_2 and dist\_ab\_2

# Check all combinations of points to ensure all the required conditions for a valid square are met

# Calculate the squared distances between the points

dist ab 2 = (x1 - x2) \*\* 2 + (y1 - y2) \*\* 2

dist ac 2 = (x1 - x3) \*\* 2 + (y1 - y3) \*\* 2

dist bc 2 = (x2 - x3) \*\* 2 + (v2 - v3) \*\* 2

// Method to determine if four points form a valid square.

checkRightTriangle(p1, p3, p4) &&

checkRightTriangle(p1, p2, p4) &&

// This function checks if four points form a valid square.

bool validSquare(std::vector<int>& p1, std::vector<int>& p2,

int ax = a[0], ay = a[1]; // Coordinates of point a

int bx = b[0], by = b[1]; // Coordinates of point b

int cx = c[0], cy = c[1]; // Coordinates of point c

int abSquaredDist = squaredDistance(ax, ay, bx, by);

int acSquaredDist = squaredDistance(ax, ay, cx, cy);

int bcSquaredDist = squaredDistance(bx, by, cx, cy);

// Helper function to calculate squared distance between two points

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

// Helper function to check if 3 points form two equal sides and one diagonal of a square

// A square should have 2 pairs of equal sides & 1 pair of equal diagonals

return areSidesValid(p1, p2, p3) && areSidesValid(p1, p3, p4) &&

function areSidesValid(a: number[], b: number[], c: number[]): boolean {

areSidesValid(p1, p2, p4) && areSidesValid(p2, p3, p4);

return (x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2);

// Calculate squared distances between points

int squaredDistance(int x1, int y1, int x2, int y2) {

// This function checks if four points form a valid square.

let ax = a[0], ay = a[1]; // Coordinates of point a

let bx = b[0], by = b[1]; // Coordinates of point b

std::vector<int>& p3, std::vector<int>& p4) {

return areSidesValid(p1, p2, p3) && areSidesValid(p1, p3, p4) &&

areSidesValid(p1, p2, p4) && areSidesValid(p2, p3, p4);

// A square should have 2 pairs of equal sides & 1 pair of equal diagonals

bool areSidesValid(std::vector<int>& a, std::vector<int>& b, std::vector<int>& c) {

// Helper function to check if 3 points form two equal sides and one diagonal of a square

// Check for two equal sides and a diagonal; also, check that sides are not zero-length

return (abSquaredDist == acSquaredDist && abSquaredDist + acSquaredDist == bcSquaredDist && abSquaredDist > 0) ||

(abSquaredDist == bcSquaredDist && abSquaredDist + bcSquaredDist == acSquaredDist && abSquaredDist > 0) ||

(acSquaredDist == bcSquaredDist && acSquaredDist + bcSquaredDist == abSquaredDist && acSquaredDist > 0);

checkRightTriangle(p2, p3, p4);

return checkRightTriangle(p1, p2, p3) &&

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

// Check all combinations of three points to see if they form right triangles.

is right angle(p1, p2, p3) and

is right angle(p2, p3, p4) and

is right angle(p3, p1, p4) and

is\_right\_angle(p1, p2, p4)

d1 = (x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2)

d2 = (x1 - x3) \* (x1 - x3) + (y1 - y3) \* (y1 - y3)

d3 = (x2 - x3) \* (x2 - x3) + (y2 - y3) \* (y2 - y3)

d1 == d2 and d1 + d2 == d3 and d1,

d2 == d3 and d2 + d3 == d1 and d2,

d1 == d3 and d1 + d3 == d2 and d1,

The implementation of the solution employs the concept of distance calculation between points and the comparison of these

Distance Formula: To calculate the squared distance between two points (x1, y1) and (x2, y2) in 2D space, we use the

distances between them: d1 is the distance between a and b, d2 is the distance between a and c, and d3 is the distance

# distance formula (x1 - x2)<sup>2</sup> + (y1 - y2)<sup>2</sup>. We avoid the square root computation for efficiency and because it can introduce

return any([

**Example Walkthrough** 

• p2 = (1, 3)

• p3 = (3, 1)

• p4 = (3, 3).

our distances, we have:

right angle between them.

would return true.

from typing import List

**Python** 

Solution Implementation

x1, y1 = point a

x2, y2 = point b

x3, y3 = point c

return anv([

])

return (

floating-point precision issues. Squared distances are sufficient for comparison purposes. Check Function: This helper function check takes three points a, b, and c as arguments and calculates the squared

- between **b** and **c**. Right Angle Validation: Inside the check function, it checks if any pair of these distances (d1, d2, d3) are equal – indicating that two sides of a triangle formed by the points are equal – and if the sum of their squares equals the square of the third
- Non-zero Sides: The check function also ensures that the distances are non-zero to prevent considering points that are the same (overlap) as valid sides of a square.

Combining Checks: The validSquare method calls the check function for all combinations of three points out of the four

The **check** is written in Python as: def check(a, b, c): (x1, y1), (x2, y2), (x3, y3) = a, b, c

The final return statement calls the check function multiple times to verify each three-point combination forms part of a square:

structures or algorithms. It is an elegant example of applying mathematical principles to solve a computational problem.

```
Let's walk through the solution approach with a small example. Suppose we have the following four points that are potential
vertices of a square:
• p1 = (1, 1)
```

Overall, the provided Python solution leverages simple geometry and efficient calculations without the need for complex data

• d2 (distance between p1 and p3):  $(1 - 3)^2 + (1 - 1)^2 = 4 + 0 = 4$ • d3 (distance between p2 and p3):  $(1 - 3)^2 + (3 - 1)^2 = 4 + 4 = 8$ The helper function check now verifies if two distances are equal and if the sum of their squares equals the third distance. For

• d1 equals d2, and d1 + d2 equals d3. We also ensure d1 is non-zero which it is. This combination reflects two equal sides of a triangle and a

These points are given in no specific order and might form a square in a 2D plane. We will use the functions and approach

First, the distance formula is used to check if the sides formed by these points are equal, and if the diagonals intersect at a right

• When checking p2, p3, and p4, we obtain d1 = 4, d2 = 4, and d3 = 8, satisfying the conditions. • For p1, p3, and p4, we compute d1 = 4, d2 = 4, and d3 = 8 again, meeting the conditions. • Lastly, for p1, p2, and p4, we find d1 = 4, d2 = 4, and d3 = 8, which also passes the check.

Each combination gives us two equal sides and a diagonal that conforms to the Pythagorean theorem, indicating four right angles

in total. Since the points p1, p2, p3, and p4 satisfy the check for every combination, they form a valid square, and the function

To summarize, the solution involves using the distance formula to calculate squared distances between different points, followed

by an application of the Pythagorean theorem to ensure right angles and equal sides. This simple yet effective solution illustrates

class Solution: def validSquare(self, p1: List[int], p2: List[int], p3: List[int], p4: List[int]) -> bool: # Helper function to check whether three points form a right angle at the first point def is right angle(point\_a, point\_b, point\_c):

### # Example usage: # solution = Solution() # print(solution.validSquare([0,0], [1,1], [1,0], [0,1])) # The output would be True

Java

class Solution {

return (

```
// Helper method to check if three points form a right-angled triangle.
    private boolean checkRightTriangle(int[] pointA, int[] pointB, int[] pointC) {
        // Extract x and y coordinates of points.
        int xA = pointA[0], vA = pointA[1];
        int xB = pointB[0], yB = pointB[1];
        int xC = pointC[0], yC = pointC[1];
        // Compute squared distances between the points.
        int distanceAB = squaredDistance(xA, vA, xB, vB);
        int distanceAC = squaredDistance(xA, yA, xC, yC);
        int distanceBC = squaredDistance(xB, yB, xC, yC);
        // Check the conditions for forming a right—angled triangle (Pythagorean theorem):
        // two distances must be equal (the sides of the square) and the sum of their squares
        // must equal the square of the third distance (the diagonal of the square).
        // Ensure that no distance is zero to prevent degenerate triangles.
        return (distanceAB == distanceAC && distanceAB + distanceAC == distanceBC && distanceAB > 0) | |
               (distanceAB == distanceBC && distanceAB + distanceBC == distanceAC && distanceAB > ∅) ||
               (distanceAC == distanceBC && distanceAC + distanceBC == distanceAB && distanceAC > ∅);
    // Helper method to calculate the squared distance between two points to avoid floating point errors.
    private int squaredDistance(int x1, int v1, int x2, int v2) {
        return (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
C++
#include <vector>
class Solution {
public:
```

# let cx = c[0], cy = c[1]; // Coordinates of point c

return

solution = Solution()

Time and Space Complexity

Example usage:

private:

**TypeScript** 

**}**;

```
// Calculate squared distances between points
    let abSquaredDist = squaredDistance(ax, av, bx, by);
    let acSquaredDist = squaredDistance(ax, ay, cx, cy);
    let bcSquaredDist = squaredDistance(bx, by, cx, cy);
   // Check for two equal sides and a diagonal; also, check that sides are not zero-length
   return (abSquaredDist === acSquaredDist && abSquaredDist + acSquaredDist === bcSquaredDist && abSquaredDist > 0) ||
           (abSquaredDist === bcSquaredDist && abSquaredDist + bcSquaredDist === acSquaredDist && abSquaredDist > ∅) ||
           (acSquaredDist === bcSquaredDist && acSquaredDist + bcSquaredDist === abSquaredDist && acSquaredDist > 0);
// Helper function to calculate squared distance between two points
function squaredDistance(x1: number, y1: number, x2: number, y2: number): number {
   return (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
from typing import List
class Solution:
   def validSquare(self, p1: List[int], p2: List[int], p3: List[int], p4: List[int]) -> bool:
       # Helper function to check whether three points form a right angle at the first point
       def is right angle(point_a, point_b, point_c):
           x1, y1 = point a
           x2, y2 = point b
           x3. v3 = point c
           # Calculate the squared distances between the points
           dist ab 2 = (x1 - x2) ** 2 + (v1 - v2) ** 2
           dist ac 2 = (x1 - x3) ** 2 + (y1 - y3) ** 2
           dist bc 2 = (x2 - x3) ** 2 + (y2 - y3) ** 2
           # Check if the triangle formed by the three points is a right triangle
           return any([
               dist ab 2 == dist ac 2 and dist ab 2 + dist ac 2 == dist bc 2 and dist ab 2.
               dist ac 2 == dist bc 2 and dist ac 2 + dist bc 2 == dist ab 2 and dist ac 2,
               dist_ab_2 == dist_bc_2 and dist_ab_2 + dist_bc_2 == dist_ac_2 and dist_ab_2
       # Check all combinations of points to ensure all the required conditions for a valid square are met
```

## **Time Complexity:** To analyze the time complexity, we'll consider the significant operations in the given code.

is right angle(p1, p2, p3) and

is right angle(p2, p3, p4) and

is right angle(p3, p1, p4) and

is\_right\_angle(p1, p2, p4)

### comparisons and additions. So the time complexity for this part is 0(1). The validSquare method calls the check function four times, once for each combination of three points out of four. Since the number of calls to check does not depend on the input size (it's always four), this also contributes a constant factor.

print(solution.validSquare([0,0], [1,1], [1,0], [0,1])) # The output would be True

The given Python code defines a method validSquare to check whether four points form a valid square.

- There are no loops or recursive calls that depend on the size of the input, hence the overall time complexity is 0(1). **Space Complexity:**
- Now let's examine the space complexity.

The main validSquare method does not use any additional space apart from the space required for the check function and

The check function calculates the distances squared (to avoid floating point arithmetic) for three pairs of points. This requires

a constant amount of operations, specifically, six subtractions and six multiplications for the distance calculations, plus

- The check function uses a fixed amount of space for variables to store the distances and the points. No additional data structures that grow with input size are used, so its space complexity is 0(1).
  - Therefore, the space complexity is 0(1). In summary:

the input points.

- The time complexity of the code is 0(1). • The **space complexity** of the code is 0(1).