# 题目

给定2D空间中四个点的坐标 p1, p2, p3 和 p4，如果这四个点构成一个正方形，则返回true。

点的坐标pi表示为[xi, yi]。 输入没有任何顺序 。

一个有效的正方形有四条等边和四个等角(90度角)。

示例 1:

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

输出: true

示例 2:

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

输出：false

示例 3:

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

输出：true

提示:

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

-104 <= xi, yi <= 104

# 分析

## 方法一：数学

思路与算法：

正方形判定定理是几何学里用于判定一个四边形是否为正方形的判定定理。判别正方形的一般顺序为先说明它是平行四边形；再说明它是菱形（或矩形）；最后说明它是矩形（或菱形）。那么我们可以从枚举四边形的两条斜边入手来进行判断：

如果两条斜边的中点相同：则说明以该两条斜边组成的四边形为「平行四边形」。

在满足「条件一」的基础上，如果两条斜边的长度相同：则说明以该两条斜边组成的四边形为「矩形」。

在满足「条件二」的基础上，如果两条斜边的相互垂直：则说明以该两条斜边组成的四边形为「正方形」。

代码：

class Solution {

public:

bool checkLength(vector<int>& v1, vector<int>& v2) {

return (v1[0] \* v1[0] + v1[1] \* v1[1]) == (v2[0] \* v2[0] + v2[1] \* v2[1]);

}

bool checkMidPoint(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {

return (p1[0] + p2[0]) == (p3[0] + p4[0]) && (p1[1] + p2[1]) == (p3[1] + p4[1]);

}

int calCos(vector<int>& v1, vector<int>& v2) {

return (v1[0] \* v2[0] + v1[1] \* v2[1]) == 0;

}

bool help(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {

vector<int> v1 = {p1[0] - p2[0], p1[1] - p2[1]};

vector<int> v2 = {p3[0] - p4[0], p3[1] - p4[1]};

if (checkMidPoint(p1, p2, p3, p4) && checkLength(v1, v2) && calCos(v1, v2)) {

return true;

}

return false;

}

bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {

if (p1 == p2) {

return false;

}

if (help(p1, p2, p3, p4)) {

return true;

}

if (p1 == p3) {

return false;

}

if (help(p1, p3, p2, p4)) {

return true;

}

if (p1 == p4) {

return false;

}

if (help(p1, p4, p2, p3)) {

return true;

}

return false;

}

};

复杂度分析

时间复杂度：O(1)。

空间复杂度：O(1)，仅使用常数变量。

## 方法二：数学

要判断四个点是否构成正方形，核心思路是利用正方形的几何特性：四条边等长、两条对角线等长，且边长的平方是对角线平方的一半。通过计算所有点之间的距离，筛选出符合条件的距离组合即可。

解题思路

1、正方形的距离特性：

四个点（无顺序）共产生 C(4,2) = 6 个 pairwise 距离（两点间距离）。在正方形中：

- 这6个距离包含4个边长和2个对角线（对角线长度是边长的√2倍）。

- 为避免浮点运算误差，统一计算距离的平方（两点 (x1,y1)和 (x2,y2) 的距离平方为 (x1-x2)² + (y1-y2)²）。

- 特性转化为：6个距离平方中，有4个相等的较小值（边长平方）和 2个相等的较大值（对角线平方），且满足 2 × 边长平方 = 对角线平方。

- 特殊排除：若存在距离平方为0（两点重合），则不可能是正方形。

2、步骤拆解：

- 计算四个点之间所有6个距离的平方，存入集合（自动去重，方便统计不同距离的数量）。

- 验证集合的大小：若不是2种距离（边长平方和对角线平方），直接返回false。

- 提取两种距离平方，分别记为d1（较小值，边长平方）和d2（较大值，对角线平方）。

- 验证：d1 ≠ 0（无重合点）且 2\*d1 == d2（对角线与边长的关系）。

代码：

class Solution {

public:

bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {

// 1. 计算所有6个两点间距离的平方，存入集合（自动去重）

unordered\_set<int> dist\_sq;

dist\_sq.insert(calcDistSq(p1, p2));

dist\_sq.insert(calcDistSq(p1, p3));

dist\_sq.insert(calcDistSq(p1, p4));

dist\_sq.insert(calcDistSq(p2, p3));

dist\_sq.insert(calcDistSq(p2, p4));

dist\_sq.insert(calcDistSq(p3, p4));

// 2. 正方形的距离平方只能有2种（边长平方和对角线平方），且不能有0（重合点）

if (dist\_sq.size() != 2 || dist\_sq.count(0)) {

return false;

}

// 3. 提取两种距离平方，区分边长平方（d1）和对角线平方（d2）

int d1 = \*dist\_sq.begin();

int d2 = \*next(dist\_sq.begin()); // next获取集合中第二个元素的迭代器

// 确保d1是较小值（边长平方），d2是较大值（对角线平方）

if (d1 > d2) {

swap(d1, d2);

}

// 4. 验证：对角线平方 = 2 × 边长平方

return 2 \* d1 == d2;

}

private:

// 辅助函数：计算两点间距离的平方（避免浮点运算）

int calcDistSq(vector<int>& a, vector<int>& b) {

int dx = a[0] - b[0];

int dy = a[1] - b[1];

return dx \* dx + dy \* dy;

}

};

代码解释

1、距离平方计算：calcDistSq函数通过坐标差的平方和计算距离平方，避免浮点运算带来的精度误差，同时提高效率。

2、集合去重：用unordered\_set存储6个距离平方，自动过滤重复值。若集合大小不是2，说明距离种类不符合正方形（如长方形有3种距离，菱形有2种但对角线不满足2倍关系）。

3、重合点排除：若集合中包含0，说明存在两点坐标相同，不可能构成正方形。

4、几何关系验证：通过2\*d1 == d2验证对角线与边长的平方关系，确保满足正方形的直角条件（勾股定理：边长² + 边长² = 对角线²）。

复杂度分析

- 时间复杂度：O(1)。四个点固定产生6个距离计算，集合操作（插入、查询）均为常数时间，整体无循环依赖输入规模。

- 空间复杂度：O(1)。集合最多存储2个元素，辅助变量数量固定，无额外动态空间开销。