19 Points and Lines

Sorting Points

A list of points (*x*, *y*) can be arranged in different ways. Consider that we want to sort them in increasing order by *x* coordinate. If two points have the same *x* value sort them *y* value.

Sample Input

```
9
2 2
2 3
1 2
1 3
2 1
1 1
3 2
3 3
3 1
```

Sample Output

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

Solution 1) First sort the points only with *y* coordinates in increasing order, and then re-sort them only with *x* coordinates in increasing order.

Solution 2) Multiply each *x* value of each point by some large number and then add its *y* value to that product. Sort the new list by *x* values in increasing order. Finally, convert the *x* values to their original values.

Solution 3) Use the build in sort methods of the programming languages. Such as, **STL sort()** method in C++ and **Collections.sort()** method in Java.

Distance Between Two Points

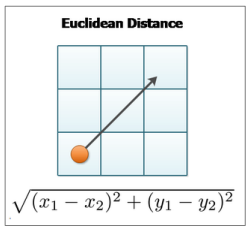
The distance between two points p1 and p2 in a 2-D plane can be measured with different metrics. The most common distance metrics are:

- Euclidean Distance
- Manhattan Distance
- Minkowski Distance

Euclidean Distance

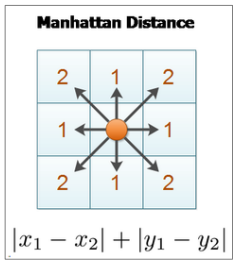
Euclidean Distance represents the shortest distance between two points.

- Use squared length to compare Euclidean distances to get rid of square root calculation and floating-point comparison.
- ***hypot(x, y)*** method returns $sqrt(x^2 + y^2)$ without intermediate overflow or underflow.



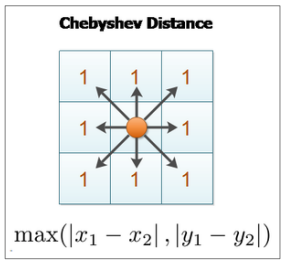
Manhattan Distance

Manhattan Distance is the sum of absolute differences between points across all the dimensions.



Chebyshev Distance

Chebyshev distance is also called Maximum value distance. It is the maximum absolute distance in one dimension of two N dimensional points.



Sum of Manhattan Distances Problem

Given N points (x, y) with integer coordinates. Calculate sum of Manhattan distances between all pairs of points.

Sample Input

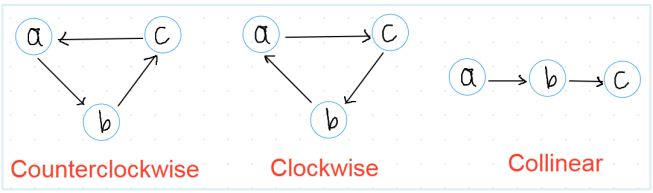
```
3
(1, 1) (3, 1) (-1, -2)
```

Sample Output

```
14
```

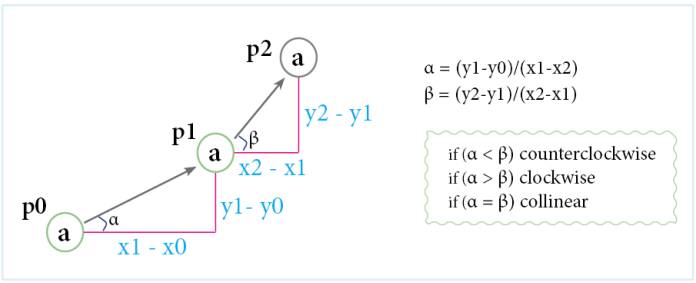
Orientation of Three Points

The three points p1(x1, y1), p2(x2, y2) and p3(x3, y3) can be **collinear**, in **clockwise** (**non-trigonometric**) order or in **counter-clockwise** (**trigonometric**) order. *a*, *b* and *c* are three points in the following illustration.



Finding Orientation of Three Points

Solution 1: Testing the Slopes



Solution 2: Test sign of sum of determinants

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = a*d - b*c$$

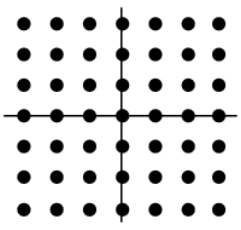
P0 (x0, y0), P1 (x1, y1) and P2 (x2, y2) are three points on the X-Y plane.

$$A = \begin{vmatrix} x0 & y0 \\ x1 & y1 \end{vmatrix} + \begin{vmatrix} x1 & y1 \\ x2 & y2 \end{vmatrix} + \begin{vmatrix} x2 & y2 \\ x0 & y0 \end{vmatrix}$$

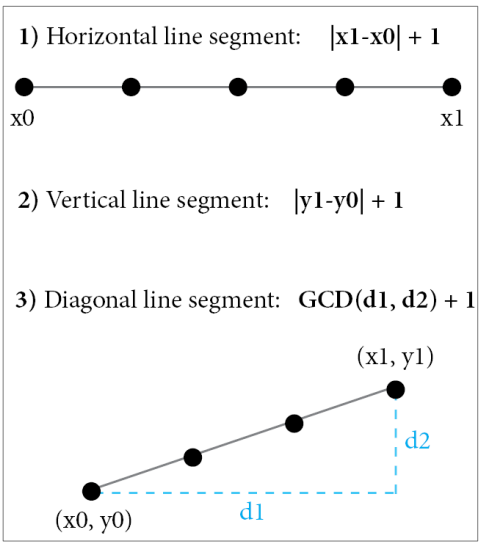
if (A > 0) counterclockwise
if (A < 0) clockwise
if (A = 0) collinear

Lattice Points (Grid Points)

A **lattice point** is a point at the intersection of two or more grid lines in a regularly spaced array of points, which is a point lattice. That is, a **point lattice** is a regularly spaced array of points.

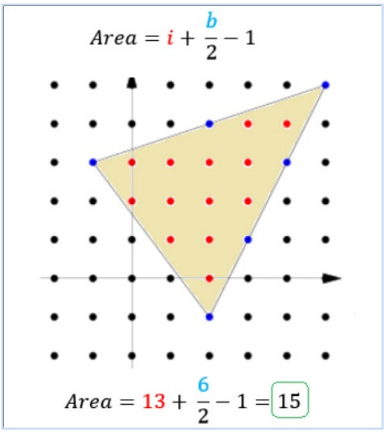


Counting Lattice Points



Pick's Theorem

Pick's Theorem states that if a polygon has vertices with integer coordinates (lattice points) then the area of the polygon is *i* + (b/2) - 1 where *i* is the number of lattice points inside the polygon and *b* is the number of lattice points on the perimeter of the polygon.



Points in a Triangle Problem

Calculate number of integer points inside a triangle.

The input is the corner points of the triangle (x1, y1), (x2, y2), (x3, y3). All coordinates are integer.

Sample Input

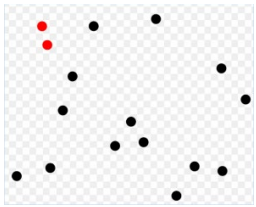
```
2 -1
5 5
-1 3
```

Sample Output

```
13
```

Closest Pair of Points Problem

Given *N* points in metric space, find a pair of points with the smallest distance between them.



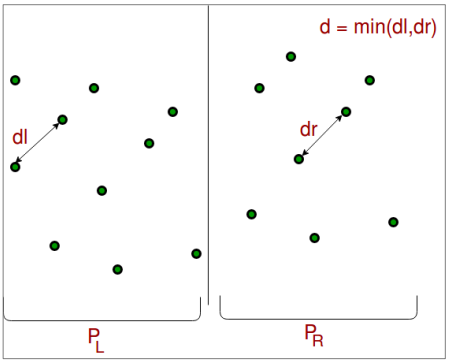
Solution 1: Divide and Conquer

As a pre-processing step, the input array is sorted according to x coordinates.

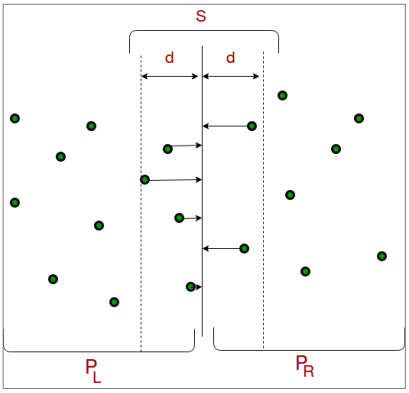
1) Find the middle point in the sorted array, we can take P[n/2] as middle point.

2) Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P[n/2+1] to P[n-1].

3) Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d.



4) From the above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Consider the vertical line passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.



5) Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.

6) Find the smallest distance in strip[]. This is tricky. From the first look, it seems to be a O(n^2) step, but it is actually O(n). It can be proved geometrically that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate). See this for more analysis.

7) Finally return the minimum of d and distance calculated in the above step (step 6)

```
// Cloest Pair Of Points Divide and Conquer.cpp :
#include <fstream>
#include <vector>
#include <algorithm>
#include <limits>

#define BIGNUM 20000

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

struct Point
{
	bool friend operator < (const Point& p1, const Point& p2)
	{
		if (p1.x == p2.x)
			return p1.y < p2.y;
		return p1.x < p2.x;
	}

	int distance(const Point& p2)
	{
		return (int)pow(x - p2.x, 2) + (int)pow(y - p2.y, 2);
	}

	int x, y;
};

bool compareY(const Point& p1, const Point& p2)
{
	if (p1.y == p2.y)
		return p1.x < p2.x;
	return p1.y < p2.y;
}

int N;
vector <Point> pointVec;
//-----------------------------------------
void readInput()
{
	cin >> N;
	pointVec.resize(N);
	for (int i = 0; i < N; i++)
		cin >> pointVec[i].x >> pointVec[i].y;
}
//-----------------------------------------
//sort v by y coordinates. Return the distance betweeen
//smallest pair of points in v.
int processMidPoints(vector<Point> v)
{
	int res = v[0].distance(v[1]);

	sort(v.begin(), v.end(), compareY);

	for (int i = 0; i < v.size(); i++)
		for (int j = i + 1; j < v.size() && (v[j].y - v[i].y) < res; j++)
			res = min(res, v[i].distance(v[j]));

	return res;
}
//-----------------------------------------
//return closest distance amoung three points
```

```
int threePoints(int startPos, int size)
{
	if (size <= 1)
		return BIGNUM;
	if (size == 2)
		return (pointVec[0].distance(pointVec[1]));

	Point p1 = pointVec[startPos];
	Point p2 = pointVec[startPos + 1];
	Point p3 = pointVec[startPos + 2];
	int res = p1.distance(p2);
	res = min(res, p1.distance(p3));
	return min(res, p2.distance(p3));
}

//-----------------------------------------
//classical divide and conquer algorithm to solve the
//closest pair of points problem.
int go(int startPos, int endPos)
{
	int n = endPos - startPos + 1;
	if (n <= 3)
		return threePoints(startPos, n);

	int midPos = startPos + n / 2;

	Point midPoint = pointVec[midPos];

	int dl = go(startPos, midPos);    //smallest distance on the left side
	int dr = go(midPos, endPos); //smallest distance on the right side

	int d = min(dl, dr);
	vector <Point> midPointsVec; //Points on the middle strip
	for (int i = startPos; i <= endPos; i++)
		if ((int)pow(midPoint.x - pointVec[i].x, 2) < d)
			midPointsVec.push_back(pointVec[i]);

	int midDistance = d;
	if (midPointsVec.size() > 1)
		midDistance = processMidPoints(midPointsVec);
	return min(d, midDistance);
}
//-----------------------------------------
int main()
{
	readInput();
	sort(pointVec.begin(), pointVec.end());
	int res = go(0, pointVec.size() - 1);
	cout << res << endl;     //it is the square distance
	return 0;
}
```

The above algorithm runs in $O(n(\log n)^2)$ time. nlogn times we sort te strip[] array.

The great thing about the above approach is, if the array strip[] is sorted according to y coordinate, then we can find the smallest distance in strip[] in O(n) time. In the implementation discussed in the previous post, strip[] was explicitly sorted in every recursive call that made the time complexity O(n (Logn)^2), assuming that the sorting step takes O(nLogn) time.

In this post, we discuss an implementation where the time complexity is O(nLogn). The idea is to presort all points according to y coordinates. Let the sorted array be Py[]. When we make recursive calls, we need to divide points of Py[] also according to the vertical line. We can do that by simply processing every point and comparing its x coordinate with x coordinate of the middle line.

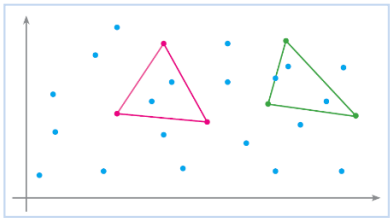
Solution 2: Brute Force After Sorting

Sort the points by x coordinate then use brute force starting with the left most point. Any time for point pi the distance to the next point you test is not smaller then the best distance, just break the loop for p_i and continue with the next point p_{i+1}. This algorithm runs almost as fast as the divide and conquer algorithm for random test data.

```
// Cloest Pair Of Points Brute Force.cpp :
#include <fstream>
#include <vector>
#include <algorithm>
#include <limits>

#define BIGNUM 18446744073709551615

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

struct Point
{
	bool friend operator < (const Point& p1, const Point& p2)
	{
		if (p1.x == p2.x)
			return p1.y < p2.y;
		return p1.x < p2.x;
	}

	unsigned long long distance(const Point& p2)
	{
		return pow(x - p2.x, 2) + pow(y - p2.y, 2);
	}

	int x, y;
};

int N;
vector <Point> pointVec;
//----------------------------------------
void readInput()
{
	cin >> N;
	pointVec.resize(N);
	for (int i = 0; i < N; i++)
		cin >> pointVec[i].x >> pointVec[i].y;
}
//----------------------------------------
unsigned long long go()
{
	sort(pointVec.begin(), pointVec.end());

	unsigned long long res = BIGNUM;

	for (int i = 0; i < pointVec.size(); i++)
		for (int j = i+1; j < pointVec.size() && pow(pointVec[j].x-pointVec[i].x, 2) < res; j++)
			res = min(res, pointVec[i].distance(pointVec[j]));

	return res;
}
//----------------------------------------
int main()
{
	readInput();
	cout << go() << endl;		//it is the square distance
	return 0;
}
```

Number of Triangles Problem

How many triangles with the non-zero area can you draw by joining tree of N points on a Cartesian plane?

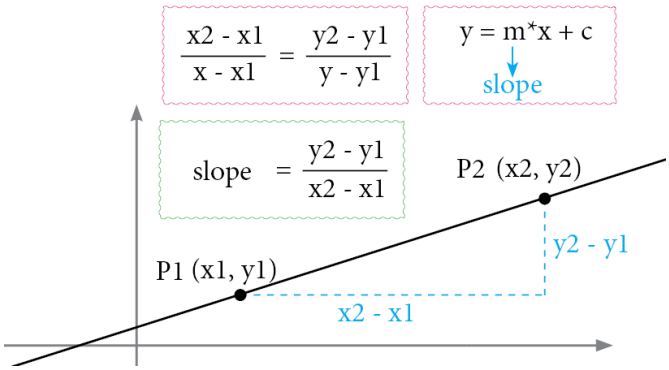


Solution

Consider a point Z and find its slope with every other point. Now, if two points are having the same slope with point Z that means the 3 points are collinear and they cannot form a triangle. Hence, the number of triangles having Z as one of its points is the number of ways of choosing 2 points from the remaining points and then subtracting the number of ways of choosing 2 points from points having the same slope with Z. Since Z can be any point among N points, we have to iterate one more loop. This algorithm runs in $O(N^2)$ time.

Lines

Line Formulas and Slope



- Horizontal line: (y1 == y2)
- Vertical line: (x1 == x2)

Parallel Lines

Determine if two lines (p0, p1) and (p2, p3) are parallel and distinct.

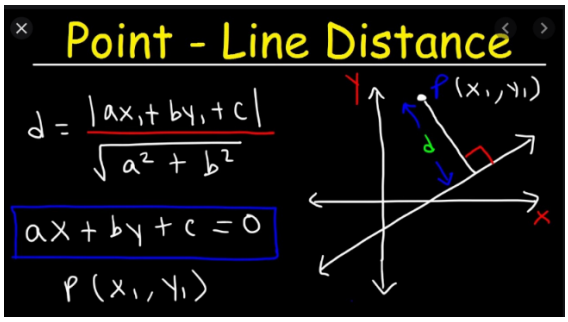
Hint: They must have the same slope and p0 or p1 must not be on the second line.

Perpendicular lines

Determine if two lines (p0, p1) and (p2, p3) are perpendicular. A line is said to be perpendicular to another line if the two lines intersect at a right angle.

Hint: When one line has a slope of *m*, a perpendicular line has a slope of *−1/m* (negative reciprocal).

Point to Line Distance



Two Points and a Line

Determine if two points p1(x1, y1) and p2(x2, y2) lie on the same side of the line (P3, p4).

Hint: When applying the points on the line one by one, they must give the same sign.

The Regions Problem

The 2D space has been divided into several regions with N lines. What is the minimum number of the regions you need to visit to travel from a start point A to an end point B?

You can only travel to one of the adjacent regions from your current region.

Useful Links and References

https://www.analyticsvidhya.com/blog/2020/02/4-types-of-distance-metrics-in-machine-learning/
https://mathworld.wolfram.com/PointLattice.html
https://www.geeksforgeeks.org/number-of-triangles-that-can-be-formed-with-given-n-points/