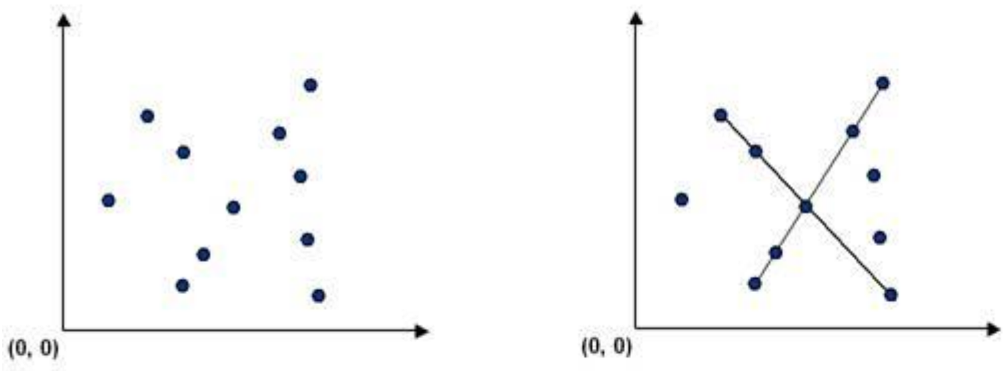
S2: Pattern recognition

Computer vision involves analysing patterns in visual images and reconstructing the real-world objects that produced them. The process in often broken up into two phases: feature detection and pattern recognition. Feature detection involves selecting important features of the image; pattern recognition involves discovering patterns in the features. We will investigate a particularly clean pattern recognition problem involving points and line segments. This kind of pattern recognition arises in many other applications such as statistical data analysis.

The problem

Given a set of N distinct points in the plane, find all line segments that connects a subset of 4 or more points.



A. Point.java

Create an immutable data type `Point` that represents a point in the plane by implementing the following API

```
public class Point implements Comparable<Point> {
   // compare points by slope to this point
   public final Comparator<Point> SLOPE_ORDER;

   // construct the point (x, y)
   public Point(int x, int y)

   // draw this point
   public void draw()

   // draw line segment from this point to that point
   public void drawTo(Point that)
```

```
   // string representation
   public String toString()

   // is this point lexicographically smaller than that point?
   public int compareTo(Point that)

   // the slope between this point and that point
   public double slopeTo(Point that)
}
```

To get started, use the attached data type `Point.java`, which implements the constructor and the `draw()`, `drawTo()`, and `toString()` methods. Your job is to add the following components.

1. The `compareTo()` method should compare points by their *y*-coordinates, breaking ties by their *x*-coordinates. Formally, the invoking point (x_0, y_0) is less than the argument point (x_1, y_1) if and only if either $y_0 < y_1$ or if $y_0 = y_1$ and $x_0 < x_1$.

2. The `slopeTo()` method should return the slope between the invoking point (x_0, y_0) and the argument point (x_1, y_1), which is given by the formula $(y_1 - y_0)/(x_1 - x_0)$. Treat the slope of a horizontal line segment as positive zero; treat the slope of a vertical line segment as positive infinity; treat the slope of a degenerate line segment (between a point and itself) as negative infinity.

3. The `SLOPE_ORDER` comparator should compare points by the slopes they make with the invoking point (x_0, y_0). Formally, the point (x_1, y_1) is less than the point (x_2, y_2) if and only if the slope $(y_1 - y_0)/(x_1 - x_0)$ is less than the slope $(y_2 - y_0)/(x_2 - x_0)$. Treat horizontal, vertical, and degenerate line segments as in the *slopeTo*() method.

Input: First comes an integer *N*, followed by *N* pairs of integers *x y*, each between 0 and 32,767.

```
10
 4000 30000
 3500 28000
 3000 26000
 2000 22000
 1000 18000
13000 21000
23000 16000
28000 13500
28000  5000
28000  1000
```

Output:. You are provided with a `Point.java` program containing a `main` method, *do not modify this method*. Three methods are evaluated, `slopeTo`, `compareTo` and the `SLOPE_ORDER` comparator.

```
Testing slopeTo method...
4.0
4.0
```

```
4.0
4.0
0.25
-0.5
-0.5
Infinity
Infinity
Testing compareTo method...
-1
-1
-1
-1
1
-1
-1
-1
-1
Testing SLOPE_ORDER comparator...
0
0
0
1
-1
0
1
0
```

B. Brute.java

Write a program `Brute.java` that examines 4 points at a time and checks whether they all lie on the same line segment, printing out any such line segments to standard output. To check whether the 4 points *p*, *q*, *r*, and *s* are collinear, check whether the slopes between *p* and *q*, between *p* and *r*, and between *p* and *s* are all equal.

The order of growth of the running time of your program should be $O(N^4)$ in the worst case ~~and it should use space proportional to N~~.

Input: First comes an integer N, followed by N pairs of integers *x y*, each between 0 and 32,767.

```
15
    10 0
    8  2
    2  8
    0  10

    20 0
    18 2
    2 18

    10 20
    30 0
```

```
0  30
20 10

13 0
11 3
5  12
9  6
```

Output: Print out all line segments. The output *must be sorted*, meaning

1. The points within each pattern produced must be given in default sorted order, and
2. The order of the patterns output must be in default sorted order.

The default sorted order is the one that you implement in `Point.compareTo()`.

`segments()` **returns the following list (of lists) on the above input:**

```
(10, 0), (13, 0), (20, 0), (30, 0)
(10, 0), (8, 2), (2, 8), (0, 10)
(13, 0), (11, 3), (9, 6), (5, 12)
(30, 0), (20, 10), (10, 20), (0, 30)
```

API

Implement the following functions:

`Brute(Point[])` : constructor that performs the computation needed

`int numberOfSegments()` : which returns the number of 4-tuples that form a line

`Iterable<Iterable<Point>> segments()` : return all segments/4-tuples that form a line

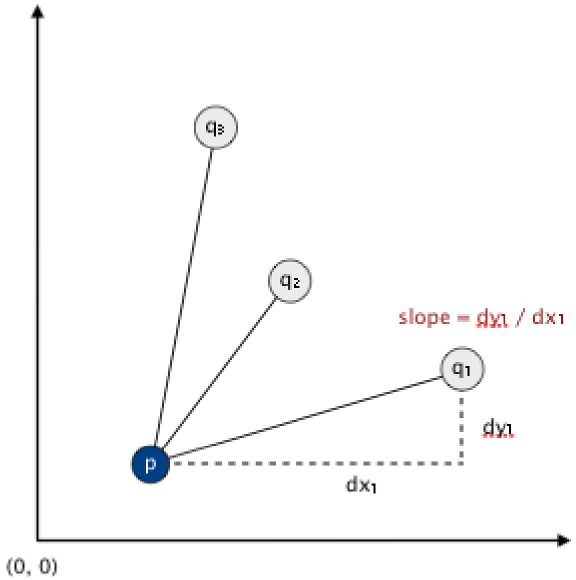
There should be no reading from stdin or writing to `stdout`. It does not matter what kind of `Iterable` data structure is used, as long as the content appears in the stated order.

C. Fast.java

A faster, sorting-based solution. Remarkably, it is possible to solve the problem much faster than the brute-force solution described above. Given a point *p*, the following method determines whether *p* participates in a set of 4 or more collinear points.

- Think of *p* as the origin.
- For each other point *q*, determine the slope it makes with *p*.
- Sort the points according to the slopes they make with *p*.
- Check if any 3 (or more) adjacent points in the sorted order have equal slopes with respect to *p*. If so, these points, together with *p*, are collinear.

Applying this method for each of the *N* points in turn yields an efficient algorithm to the problem. The algorithm solves the problem because points that have equal slopes with respect to *p* are collinear, and sorting brings such points together. The algorithm is fast because the bottleneck operation is sorting.



Write a program Fast.java that implements this algorithm. The order of growth of the running time of your program should be $\mathcal{O}(N^2\log(N))$ in the worst case~~ and it should use space proportional to *N*~~.

Input: Same as input for B.

Output: Print out all line segments. The output *must be sorted*, meaning

1. The points within each pattern produced must be given in default sorted order, and
2. The order of the patterns output must also be in default sorted order by the **first point** in each line, breaking ties with the slope of the patterns.

The default sorted order is the one that you implement in `Point.compareTo()`.

`segments()` should return the following list (of lists) on the above input:

```
(10, 0), (8, 2), (2, 8), (0, 10)
(10, 0), (13, 0), (20, 0), (30, 0)
(13, 0), (11, 3), (9, 6), (5, 12)
(30, 0), (20, 10), (10, 20), (0, 30)
```

Examples of incorrect output from the above input:

```
# Example 1: Breaks the first rule
(10, 0), (0, 10), (2, 8), (8, 2)
(10, 0), (20, 0), (30, 0), (13, 0)
(13, 0), (5, 12), (9, 6), (11, 3)
(30, 0), (0, 30), (10, 20), (20, 10)

# Example 2: Patterns not sorted by their first point
(30, 0), (20, 10), (10, 20), (0, 30)
(13, 0), (11, 3), (9, 6), (5, 12)
```

```
(10, 0), (13, 0), (20, 0), (30, 0)
(10, 0), (8, 2), (2, 8), (0, 10)

# Example 3: The first pattern has slope 0
# while the second pattern has slope -1
(10, 0), (13, 0), (20, 0), (30, 0)
(10, 0), (8, 2), (2, 8), (0, 10)
(13, 0), (11, 3), (9, 6), (5, 12)
(30, 0), (20, 10), (10, 20), (0, 30)
```

API: Same as Brute.java.

D. Fast2.java

Do not print or plot subsegments of a line segment containing 5 or more points (e.g., if you output $p \to q \to r \to s \to t$, do not also output either $p \to q \to s \to t$ or $q \to r \to s \to t$). Same input specification as for $[B, C]$

Example. If `Fast.java` (i.e., segments() in Fast.java) outputs the patterns

```
(0,0), (5,0), (10,0), (15,0), (20,0), (25,0), (30,0)
(5,0), (10,0), (15,0), (20,0), (25,0), (30,0)
(10,0), (15,0), (20,0), (25,0), (30,0)
(15,0), (20,0), (25,0), (30,0)
```

then `Fast2.java` should only output only the first pattern:

```
(0,0), (5,0), (10,0), (15,0), (20,0), (25,0), (30,0)
```

Input/Output and API: Same as for **C**.

Hand-in

Submit the following together on GradeScope:

1. Your programs to problems A (Point.java), B (Brute.java), C (Fast.java) and optionally D (Fast2.java).
2. The file readme.txt.

Acknowledgement: This assignment follows *Programming Assignment 3: Collinear Points* on Coursera, with some modifications. The original assignment was developed by Kevin Wayne. Copyright (C) 2005.