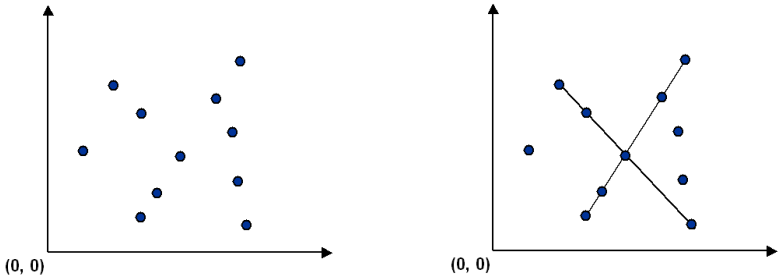
Programming Assignment 3: Pattern Recognition

Write a program to recognize line patterns in a given set of points.

Computer vision involves analyzing 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, draw every (maximal) line segment that connects a subset of 4 or more of the points.



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

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

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

   public   void draw()                              // draw this point
   public   void drawTo(Point that)                  // draw the line segment from this point to that point
   public String toString()                          // string representation

   public    int compareTo(Point that)               // is this point lexicographically smaller than that point?
   public double slopeTo(Point that)                 // the slope between this point and that point
}
```

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

- 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 $x_0 < x_1$.
- 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.
- 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.

Brute force. 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 and drawing them using standard drawing. 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.

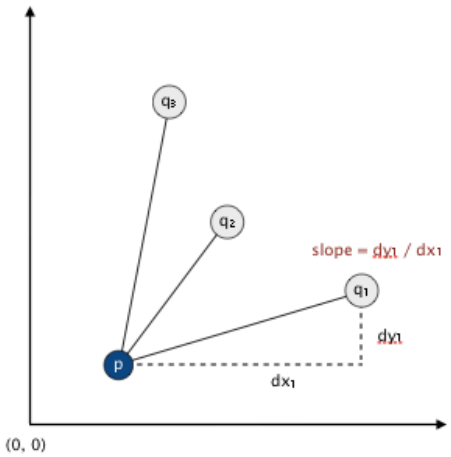
The order of growth of the running time of your program should be N^4 in the worst case and it should use space proportional to N.

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 makes 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 $N^2 \log N$ in the worst case and it should use space proportional to N.

APIs. Each program should take the name of an input file as a command-line argument, read the input file (in the format specified below), print to standard output the line segments discovered (in the format specified below), and draw to standard draw the line segments discovered (in the format specified below). Here are the APIs.

```
public class Brute {
   public static void main(String[] args)
}

public class Fast {
   public static void main(String[] args)
}
```

Input format. Read the points from an input file in the following format: An integer N, followed by N pairs of integers (x, y), each between 0 and 32,767. Below are two examples.

```
% more input6.txt
6
19000  10000
18000  10000
32000  10000
21000  10000
 1234   5678
14000  10000
```

```
% more input8.txt
8
 10000      0
     0  10000
  3000   7000
  7000   3000
 20000  21000
  3000   4000
 14000  15000
  6000   7000
```

Output format. Print to standard output the line segments that your program discovers, one per line. Print each line segment as an *ordered* sequence of its constituent points, separated by `" -> "`.

```
% java Brute input6.txt
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (21000, 10000)
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (32000, 10000)
(14000, 10000) -> (18000, 10000) -> (21000, 10000) -> (32000, 10000)
(14000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)
(18000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)

% java Brute input8.txt
(10000, 0) -> (7000, 3000) -> (3000, 7000) -> (0, 10000)
(3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)

% java Fast input6.txt
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)

% java Fast input8.txt
(10000, 0) -> (7000, 3000) -> (3000, 7000) -> (0, 10000)
(3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)
```

Also, draw the points using `draw()` and draw the line segments using `drawTo()`. Your programs should call `draw()` once for each point in the input file and it should call `drawTo()` once for each line segment discovered. Before drawing, use `StdDraw.setXscale(0, 32768)` and `StdDraw.setYscale(0, 32768)` to rescale the coordinate system.

For full credit, do not print *permutations* of points on a line segment (e.g., if you output $p \to q \to r \to s$, do not also output either $s \to r \to q \to p$ or $p \to r \to q \to s$). Also, for full credit in `Fast.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 $q \to r \to s \to t$); you may print out such subsegments in `Brute.java`.

Deliverables. Submit only `Brute.java`, `Fast.java`, and `Point.java`. We will supply `stdlib.jar` and `algs4.jar`. Your may not call any library functions other than those in `java.lang`, `java.util`, `stdlib.jar`, and `algs4.jar`.

This assignment was developed by Kevin Wayne.