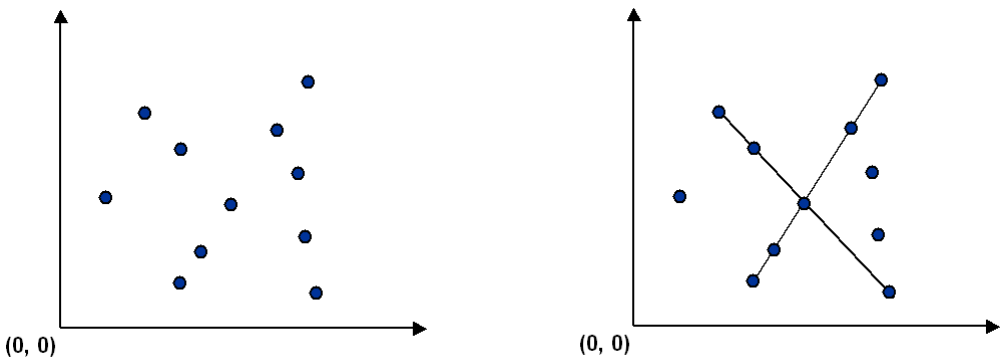
Programming Assignment 3: Pattern Recognition

The APIs have been revised substantially for the Fall 2015 offering.

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 is 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 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 Point(int x, int y)                          // constructs the
point (x, y)

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

   public               int compareTo(Point that)      // compare two points
by y-coordinates, breaking ties by x-coordinates
   public            double slopeTo(Point that)        // the slope between
this point and that point
   public Comparator<Point> slopeOrder()               // compare two points
by slopes they make with this 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 `slopeOrder()` method should return a comparator that compares its two argument 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.

Corner cases. To avoid potential complications with integer overflow or floating-point precision, you may assume that the constructor arguments `x` and `y` are each between 0 and 32,767.

Line segment data type. To represent line segments in the plane, use the data type LineSegment.java, which has the following API:

```
public class LineSegment {
   public LineSegment(Point p, Point q)        // constructs the line
segment between points p and q
   public   void draw()                        // draws this line segment
   public String toString()                    // string representation
}
```

Brute force. Write a program `BruteCollinearPoints.java` that examines 4 points at a time and checks whether they all lie on the same line segment, returning all such line segments. To check whether the 4 points *p*,*q*, *r*, and *s* are collinear, check whether the three slopes between *p* and *q*, between *p* and *r*, and between *p* and *s* are all equal.

```
public class BruteCollinearPoints {
   public BruteCollinearPoints(Point[] points)     // finds all line
segments containing 4 points
   public           int numberOfSegments()         // the number of line
segments
   public LineSegment[] segments()                 // the line segments
}
```

The method `segments()` should include each line segment containing 4 points exactly once. If 4 points appear on a line segment in the order $p \to q \to r \to s$, then you should include either the line segment $p \to s$ or $s \to p$ (but not both) and you should not include *subsegments* such as $p \to r$ or $q \to r$. For simplicity, we will not supply any input to `BruteCollinearPoints` that has 5 or more collinear points.

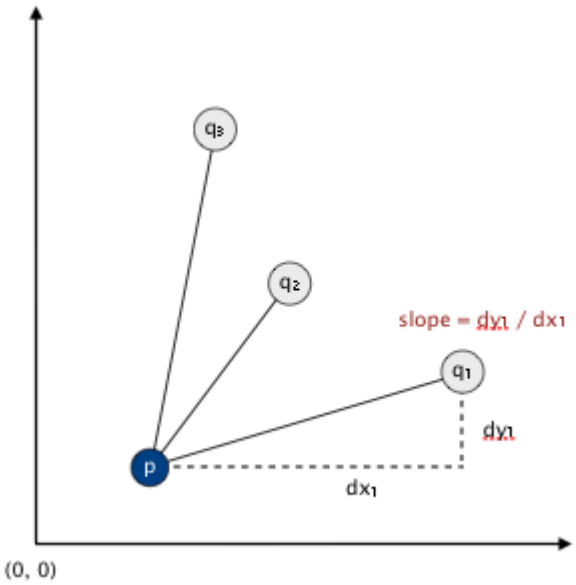
Corner cases. Throw a `java.lang.NullPointerException` either the argument to the constructor is `null` or if any point in the array is `null`. Throw a `java.lang.IllegalArgumentException` if the argument to the constructor contains a repeated point.

Performance requirement. 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 plus the number of line segments returned.

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 `FastCollinearPoints.java` that implements this algorithm.

```
public class FastCollinearPoints {
   public FastCollinearPoints(Point[] points)     // finds all line
segments containing 4 or more points
   public            int numberOfSegments()       // the number of line
segments
   public LineSegment[] segments()                // the line segments
}
```

The method `segments()` should include each *maximal* line segment containing 4 (or more) points exactly once. For example, if 5 points appear on a line segment in the order $p \to q \to r \to s \to t$, then do not include the subsegments $p \to s$ or $q \to t$.

Corner cases. Throw a `java.lang.NullPointerException` either the argument to the constructor is `null` or if any point in the array is `null`. Throw a `java.lang.IllegalArgumentException` if the argument to the constructor contains a repeated point.

Performance requirement. 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 plus the number of line segments returned.`FastCollinearPoints` should work properly even if the input has 5 or more collinear points.

Sample client. This client program takes the name of an input file as a command-line argument; read the input file (in the format specified below); prints to standard output the line segments that your program discovers, one per line; and draws to standard draw the line segments.

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

    // read the N points from a file
```

```
    In in = new In(args[0]);
    int N = in.readInt();
    Point[] points = new Point[N];
    for (int i = 0; i < N; i++) {
        int x = in.readInt();
        int y = in.readInt();
        points[i] = new Point(x, y);
    }

    // draw the points
    StdDraw.show(0);
    StdDraw.setXscale(0, 32768);
    StdDraw.setYscale(0, 32768);
    for (Point p : points) {
        p.draw();
    }
    StdDraw.show();

    // print and draw the line segments
    BruteCollinearPoints collinear = new BruteCollinearPoints(points);
    for (LineSegment segment : collinear.segments()) {
        StdOut.println(segment);
        segment.draw();
    }
}
```

Input format. We supply several sample input files (suitable for use with the test client above) 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
```

```
% java BruteCollinearPoints input8.txt
(10000, 0) -> (0, 10000)
(3000, 4000) -> (20000, 21000)

% java FastCollinearPoints input8.txt
(3000, 4000) -> (20000, 21000)
(0, 10000) -> (10000, 0)

% java FastCollinearPoints input6.txt
(14000, 10000) -> (32000, 10000)
```

Deliverables. Submit only the files `BruteCollinearPoints.java`, `FastCollinearPoints.java`, and `Point.java`. We will supply `LineSegment.java` and `algs4.jar`. You may not call any library functions other those in `java.lang`, `java.util`, and `algs4.jar`. In particular, you may call `Arrays.sort()`.

Programming Assignment 3 Checklist: Pattern Recognition

Frequently Asked Questions

Can the same point appear more than once as input to methods in Point? Yes. For the `slopeTo()` method, this requirement is explicitly stated in the API; for the comparison methods, this requirement is implicit in the contracts for `Comparable` and `Comparator`.

The reference solution outputs a line segment in the order *p*→*q* but my solution outputs it in the reverse order *q*→*p*. Is that ok? Yes, there are two valid ways to output a line segment.

The reference solution outputs the line segments in a different order than my solution. Is that OK? Yes, if there are *k* line segments, then there are *k*! different possible ways to output them.

How do I sort a subarray in Java? `Arrays.sort(a, lo, hi)` sorts the subarray from `a[lo]` to `a[hi-1]` according to the natural order of `a[]`. You can use a `Comparator` as the fourth argument to sort according to an alternate order.

Where can I see examples of Comparable and Comparator? See the lecture slides.

My program fails only on (some) vertical line segments. What could be going wrong? Are you dividing by zero? With integers, this produces a run-time exception. With floating-point numbers, 1.0/0.0 is positive infinity and -1.0/0.0 is negative infinity. You may also use the constants `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY`.

What does it mean for slopeTo() to return positive zero? Java (and the IEEE 754 floating-point standard) define two representations of zero: negative zero and positive zero.

```
double a = 1.0;
double x = (a - a) /  a;   // positive zero ( 0.0)
double y = (a - a) / -a;   // negative zero (-0.0)
```

Note that while `(x == y)` is guaranteed to be true, Arrays.sort() treats negative zero as strictly less than positive zero. Thus, to make the specification precise, we require you to return positive zero for horizontal line segments. Unless your program casts to the wrapper type `Double` (either explicitly or via autoboxing), you probably will not notice any difference in behavior; but, if your program does cast to the wrapper

type and fails only on (some) horizontal line segments, this may be the cause.

Is it OK to compare two floating-point numbers a and b for exactly equality? In general, it is hazardous to compare a and b for equality if either is susceptible to floating-point roundoff error. However, in this case, you are computing b/a, where a and b are integers between -32,767 and 32,767. In Java (and the IEEE 754 floating-point standard), the result of a floating-point operation (such as division) is the nearest representable value. Thus, for example, it is guaranteed that `(9.0/7.0 == 45.0/35.0)`. In other words, it's sometimes OK to compare floating-point numbers for exact equality (but only when you know exactly what you are doing!)

Note also that it is possible to implement `compare()` and `FastCollinearPoints` using only integer arithmetic (but you are not required to do so).

I'm having trouble avoiding subsegments Fast.java when there are 5 or more points on a line segment. Any advice? Not handling the 5-or-more case is a bit tricky, so don't kill yourself over it.

I created a nested Comparator class within Point. Within the nested Comparator class, the keyword `this` refers to the Comparator object. How do I refer to the Point instance of the outer class? Use `Point.this` instead of `this`. Note that you can refer directly to instance methods of the outer class (such as `slopeTo()`); with proper design, you shouldn't need this awkward notation.

Testing

Sample data files. The directory collinear contains some sample input files in the specified format. Associated with some of the input .txt files are output .png files that contains the desired graphical output. For convenience,collinear-testing.zip contains all of these files bundled together. Thanks to Jesse Levinson '05 for the remarkable input file `rs1423.txt`; feel free to create your own and share with us in the Discussion Forums.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. **Getting started.** Download Point.java.
2. **Slope.** To begin, implement the `slopeTo()` method. Be sure to consider a variety of corner cases, including horizontal, vertical, and degenerate line segments.
3. **Brute force algorithm.** Write code to iterate through all 4-tuples and check if the 4 points are collinear. To form a line segment, you need to know its endpoints. One approach is to form a line segment only if the 4 points are in ascending order (say, relative to the natural order), in which case, the endpoints are the first and last points.

   Hint: don't waste time micro-optimizing the brute-force solution. Though, there are two easy opportunities. First, you can iterate through all combinations of 4 points (N choose 4) instead of all 4 tuples (N^4), saving a factor of 4! = 24. Second, you don't need to consider whether 4 points are collinear if you already know that the first 3 are not collinear; this can save you a factor of N on typical inputs.

4. **Fast algorithm.**
   - Implement the `slopeOrder()` method in `Point`. The complicating issue is that the comparator needed to compare the slopes that two points `q` and `r` make with a third point `p`, which changes from sort to sort. To do this, create a private nested (non-static) class `SlopeOrder` that implements the `Comparator<Point>` interface. This class has a single method `compare(q1, q2)` that compares the slopes that `q1` and `q2` make with the invoking object `p`. the `slopeOrder()` method should create an instance of this nested class and return it.
   - Implement the sorting solution. Watch out for corner cases. Don't worry about 5 or more points on a line segment yet.

Enrichment

Can the problem be solved in quadratic time and linear space? Yes, but the only compare-based algorithm I know of that guarantees quadratic time in the worst case is quite sophisticated. It involves converting the points to their dual line segments and topologically sweeping the arrangement of lines by Edelsbrunner and Guibas.

Can the decision version of the problem be solved in subquadratic time? The original version of the problem cannot be solved in subquadratic time because there might be a quadratic number of line segments to output. (See next question.)

The decision version asks whether there exists a set of 4 collinear points. This version of the problem belongs to a group of problems that are known as 3SUM-hard. A famous unresolved conjecture is that such problems have no subquadratic algorithms. Thus, the sorting algorithm presented above is about the best we can hope for (unless the conjecture is wrong). Under a restricted decision tree model of computation, Erickson proved that the conjecture is true.

What's the maximum number of (maximal) collinear sets of points in a set of N points in the plane? It can grow quadratically as a function of N. Consider the N points of the form: (x, y) for $x = 0, 1, 2,$ and 3 and $y = 0, 1, 2, ..., N / 4$. This means that if you store all of the (maximal) collinear sets of points, you will need quadratic space in the worst case.