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.