# Programming Assignment 2 Checklist: Pattern Recognition

## Frequently Asked Questions

**How do I read input directly from a file, without redirecting standard input?** Use the In data type. Read pp. 82-83 of the textbook for more details.

**Can the same point appear more than once as input to Brute or Fast?** You may assume the input to `Brute` and `Fast` are *N* distinct points. Nevertheless, the methods implemented as part of the `Point` data type must correctly handle the case when the points are not distinct: for the `slopeTo()` method, this requirement is explicitly stated in the API; for the comparison methods, this requirement is implict in the contracts for `Comparable` and `Comparator`.

**The reference solution outputs a line segment in the order *p*→*q*→*r*→*s* but my solution outputs it in the reverse order *s*→*r*→*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 return an `Iterable<PointSequence>`?** Add the items you want to a `Stack<PointSequence>` or `Queue<PointSequence>` and return that.

**Ok great, but what is an `Iterable` exactly?** Any object which implements the `Iterable` interface, which in turn means the object must have an `Iterator iterator()` method. For example, a `Deque` is an `Iterable`. The idea is much like how `Arrays.sort` takes an array of `Comparables` as an argument, even though `Arrays.sort` might not know exactly what a `Comparable` is.

**How do I sort a subarray?** `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.

**Is Array.sort() stable?** Yes, Arrays.sort() is stable when the argument is an array of objects.

**Where can I see examples of Comparable and Comparator?** See the lecture slides or Point2D.java. We assume this is new Java material for most of you, so don't hesitate to ask for clarifications in the Discussion Forums.

**My program fails only on (some) vertical line segments. What could be going wrong?** Are you dividing by zero? With integers, this produces a runtime 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 our case, we 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 `Fast` using only integer arithmetic, though 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 testinput folder 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.

**Sample results.**

The Point line segments found with Brute for input6.txt are:

```
(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)
```

The Point line segments found with Brute for input8.txt are:

```
(10000, 0) -> (7000, 3000) -> (3000, 7000) -> (0, 10000)
```

```
(3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)
```

The Point line segments found with Fast for input6.txt are:

```
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (21000, 10000) ->
(32000, 10000)
```

The Point line segments found with Fast for input8.txt are:

```
(10000, 0) -> (7000, 3000) -> (3000, 7000) -> (0, 10000)
(3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)
```

## 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 and PointPlotter.java. The latter takes a command-line argument, reads in a list of points from the file specified as a command-line argument (in the format specified) and plots the results using standard draw. To plot the points, type the following at the command line
2. `% java PointPlotter input56.txt`
3. **Slope.** To begin, implement the `slopeTo()` method. Be sure to consider a variety of corner cases, including horizontal, vertical, and degenerate line segments.
4. **Brute force algorithm.** Write code to iterate through all 4-tuples and check whether the 4 points are collinear. Write the `main()` function the validate.

   *Hint*: don't waste time micro-optimizing the brute-force solution. Though, if you really want to, 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.

5. **Fast algorithm.**
   - Implement the `SLOPE_ORDER` comparator 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`) changes from sort to sort. To do this include a `public` and `final` (but not `static`) instance variable `SLOPE_ORDER` in `Point` of type `Comparator<Point>`. This `Comparator` has a `compare()` method so that `compare(q, r)` compares the slopes that `q` and `r` make with the invoking object `p`.
   - Implement the sorting solution. Watch out for corner cases. It's ok to start by implementing a solution that does not work for 5 or more points on a line segment.

## Enrichment

**Can the problem be solved in quadratic time and linear space?** Yes, but the only algorithm I know of 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.