Assignment 4: Improving Efficiency with Sorting

Assigned: Monday, September 28, 2015
Due: Friday, October 9, 2015, 11:59 p.m.
Type: Individual

Problem Overview

This assignment will investigate an example *feature extraction* problem. Feature extraction is a subproblem of pattern recognition and is also used in areas such as statistical analysis, computer vision, and image processing. For example, an image processing problem may use a feature extraction algorithm to identify particular shapes or regions in a digitized image.

In this assignment, we're going to focus on a very simple feature extraction problem: Given a set of points in two-dimensional space, identify every subset of four or more points that are *collinear*. For example, given the set of points depicted in Figure 1, your program would detect the three groups of collinear points as depicted by the line segments in Figure 2.

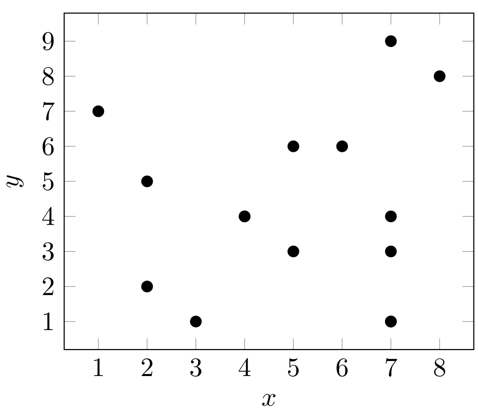

Figure 1: A set of 13 points.

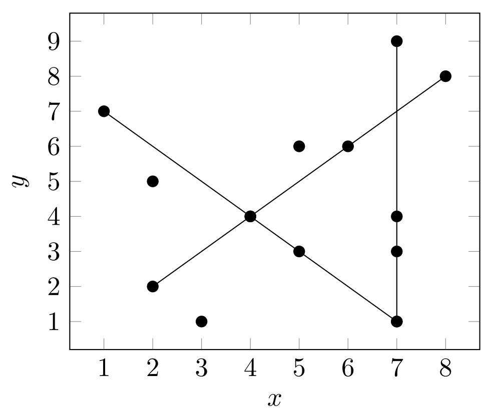

Figure 2: Three collinear groups identified.

As always, we want our solution to be useful at scale. For example, Figure 3 plots ∼100,000 points and Figure 4 shows the 34 collinear groups identified by blue line segments. Each collinear group in Figure 4 is composed of far more than four points; four is just the minimum number of points to qualify for the collinear pattern that we're looking for. In the general problem statement we will refer to line segments instead of collinear groups, where each line segment must contain at least four points.

Problem Statement: Given a set of N distinct points in the plane, identify every line segment that connects a subset of four or more of the points. Each point will be specified by an (x, y) pair where x and y are `int` values in the range 0 to $32,767$. For example, the thirteen points in Figures 1 and 2 are: (1, 7), (2, 2), (2, 5), (3, 1), (4, 4), (5, 3), (5, 6), (6, 6), (7, 1), (7, 3), (7, 4), (7, 9), (8, 8).

You must solve this problem in terms of the classes and methods described in the following sections.

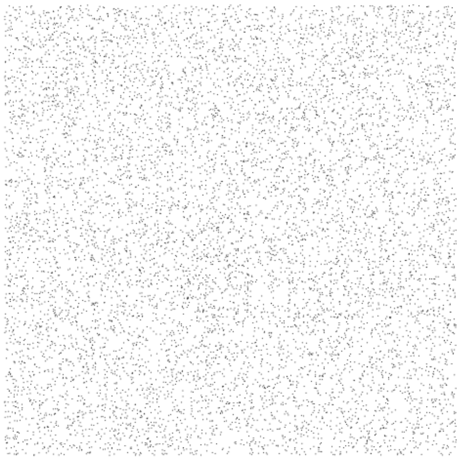

Figure 3: A set of ∼100,000 points.

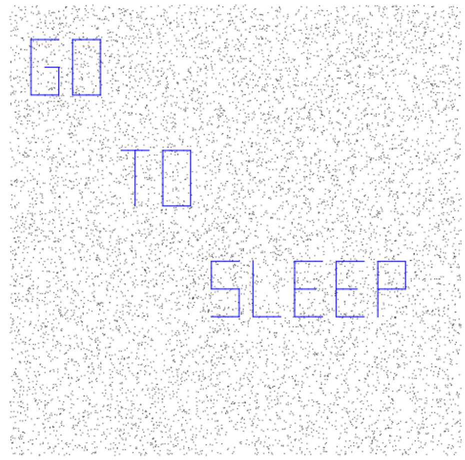

Figure 4: 34 collinear groups identified.

The `Point` class

You must create an immutable[1] data type `Point` that represents a point in the plane. A shell of the `Point` class is provided for you, and its API is described below.

```
public class Point implements Comparable<Point> {
   public final Comparator<Point> SLOPE_ORDER

   public Point(int x, int y)

   public void draw()
   public void drawTo(Point that)
   public double slopeTo(Point that)
   public int compareTo(Point that)
   public boolean equals(Object that)
   public String toString()
}
```

The constructor, `draw` method, `drawto` method, and `toString` method have been completed for you and must not be changed. You must add the bodies of the remaining methods yourself. You may add any number of private methods that you like, but you may not add any public method or constructor, nor may you change the signature of any public method or constructor.

The methods in the `Point` API that you must complete are described in more detail below.

The `compareTo` method.

This method must compare points by y-coordinates, breaking ties by x-coordinates. Thus, the invoking point (x_0, y_0) is less than the parameter point (x_1, y_1) if and only if either $y_0 < y_1$ or if $y_0 = y_1$ and $x_0 < x_1$. For example, by this *natural order* of points, (0, 1) is less than (0, 2), (7, 1) is less than (5, 3), and (3, 0) is less than (4, 0). (See Figure 5.) Two points are equal if and only if $x_0 = x_1$ and $y_0 = y_1$. This is consistent with the `equals` method, which is provided for you and should not be changed.

[1]This is just means that once you create a Point you don't change its x or y value. Thus, there are no setter methods and the fields are private.

The `slopeTo` method.

This method must return the slope between the invoking point (x_0, y_0) and the parameter point (x_1, y_1), which is given by the formula:

$$\frac{(y_1 - y_0)}{(x_1 - x_0)}$$

For example, for the point (3, 3), the slope to (1, 1) is 1.0, the slope to (4, 5) is 2.0, and the slope to (5, 2) is -0.5. (See Figure 6.)

Treat the slope of a horizontal line segment (e.g., {(1,3), (3,3)}) as positive zero[2]; treat the slope of a vertical line segment (e.g., {(3,3), (3,5)}) as positive infinity[2]; treat the slope of a degenerate line segment (between a point and itself, e.g., {(3,3), (3,3)}) as negative infinity[2]. (See Figure 6.)

The `SLOPE_ORDER` Comparator

This field of the `Point` class must compare two points by the slopes they make with the invoking point (x_0, y_0). Thus, the point (x_1, y_1) is less than the point (x_2, y_2) if and only if

$$\frac{(y_1 - y_0)}{(x_1 - x_0)} < \frac{(y_2 - y_0)}{(x_2 - x_0)}$$

Treat horizontal, vertical, and degenerate line segments the same as in the `slopeTo()` method.

For example, if the invoking point is (3, 3), then (5, 2) is less than (1, 1), and (1, 1) is less than (4, 5). (See Figure 6.)

You will need to write a nested inner class that implements that slope-order behavior as a `Comparator<Point>`, and then set the `SLOPE_ORDER` field to be an instance of this class.

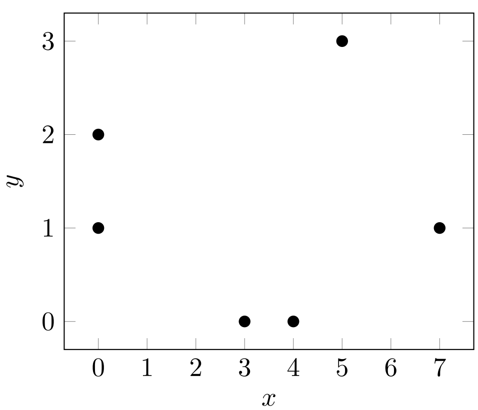

Figure 5: Reference for Point natural order.

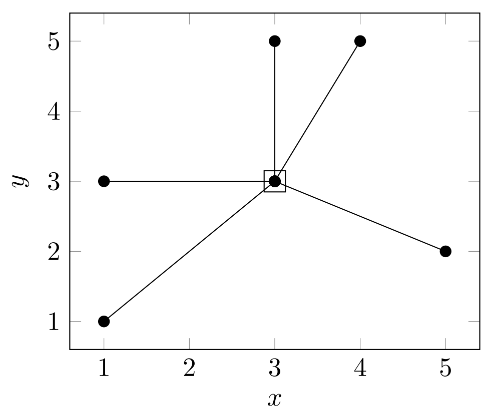

Figure 6: Reference for Point slope and slope order

The `Line` Class

You must create a data type `Line` that models a group of points that form a line segment. A shell of the `Line` class is provided for you, and its API is described below.

```
public class Line implements Comparable<Line>, Iterable<Point> {
   public Line()
```

[2]See the Java documentation of the `Double` class for a discussion of *positive zero*, *positive infinity*, and *negative infinity*.

```
    public Line(Collection<Point> points)
    public void add(Point p)
    public Point first()
    public Point last()
    public int length()
    public int compareTo(Line otherLine)
    public boolean equals(Object that)
    public Iterator<Point> iterator()
    public String toString()
}
```

This class models a line segment as a *set* of points. A set is an appropriate model since a point only appears once on a line segment and the order in which points are listed is irrelevant. That is, {(1,7), (4,4), (5,3), (7,1)} and {(4,4), (7,1), (1,7), (5,3)} each define the same line segment. Note that `Point` has a total order defined so we will make a Line a *sorted set* of Points.

The field `line` in the `Line` class has been declared for you as a `java.util.SortedSet`. You will need to read the API for `SortedSet` and choose which implementing class you want to use. (Either will work.)

We're also going to define a total order for Lines, and so the `Line` class implements the `Comparable` interface. This will allow us to conveniently store lines in a sorted set.

The Constructors

The parameterless constructor creates a new line with no points. The second constructor creates a new line that contains all the distinct collinear points in the `Collection` parameter.

The `add` method

The `add` method adds the given `Point` to the line, provided it is collinear with the existing points in the line and it isn't already present.

The `first` method

The `first` method returns the point on the line that is minimum with respect to the natural ordering of `Point`. If the line has no points, `first` returns null.

The `last` method

The `last` method returns the point on the line that is maximum with respect to the natural ordering of `Point`. If the line has no points, `last` returns null.

The `length` method

The `length` method returns the number of points on the line.

The `iterator` method

The `iterator` method returns an `Iterator` over the points on the line. The iteration order is natural order of `Point`.

The `compareTo` method

The `compareTo` method compares this line with the parameter line. The natural order of `Line` is based on the natural order of its first and last point. That is, given $line_1$ and $line_2$, $line_1 < line_2$ if $line_1.first < line_2.first$ or $line_1.first = line_2.first$ and $line_1.last < line_2.last$. (See Figure 7 where $l_1 < l_2 < l_3$.) Two lines $line_1$ and $line_2$ are equal if and only if $line_1.first = line_2.first$ and $line_1.last = line_2.last$. This is consistent with the `equals` method, which is provided for you and should not be changed.

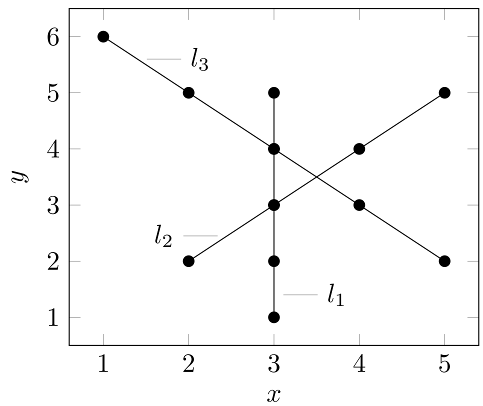

Figure 7: Reference for Line natural order.

The `Extractor` Class

This class provides methods that allow clients to read in an input file of `Point` data and find all lines segments of four or more collinear points in that data. The API of this class is described below, and as in the case of the `Point` and `Line` classes above, you may not modify this API in any way.

```
public class Extractor {
   public Extractor(String filename)
   public Extractor(Collection<Point> pcoll)
   public void drawPoints()
   public void drawLines()
   public SortedSet<Line> getLinesBrute()
   public SortedSet<Line> getLinesFast()
}
```

The Constructors

The first constructor for the `Extractor` class takes a single parameter of type `String`. This parameter is a filename for a file of `Point` data formatted as follows: The first line of the file contains a single `int` value N that is the number of lines `Point` data that follow. Each of the following N lines contains two `int` values separated by one or more blanks. The first `int` is the x value of a `Point` and the second `int` is the y value of a `Point`. There may be lines of text past these first $N + 1$ lines of data, but they should be ignored.

A sample input file is shown below.

```
5
11000 11000
12000 10000
```

```
13000 10000
14000 10000
15000 10000
```

Instantiating an `Extractor` object with this data file would ensure that five distinct instances of the `Point` class are stored in a suitable data structure inside the new `Extractor` object.

The second constructor takes a `Collection` of points and creates an `Extractor` for this data.

The `drawPoints` method

This method uses the `StdDraw` class in the provided `stdlib.jar` file to graphically display all the points associated with this `Extractor` object. Specifically, this method should iterate over all the `Point` objects in the `Extractor` object's internal data structure and invoke each `Point`'s `draw` method.

This method is only to help you visualize the data that your program is processing. It will not be invoked by the grading program, and thus it will not affect your grade in any way.

The `drawLines` method

This method uses the `StdDraw` class in the provided `stdlib.jar` file to graphically display all the lines already identified by this `Extractor` object, if any. If no lines have yet been identified, this method will still raise a graphics window with no lines drawn. If lines have been identified, then this method can use the `drawTo` method of the `Point` object at one end of the line segment to draw a line from that `Point` to the `Point` object at the other end of the line segment.

This method is only to help you visualize the data that your program is processing. It will not be invoked by the grading program, and thus it will not affect your grade in any way.

The `getLinesBrute` method

This method implements a straight-forward, *brute force* approach to extracting the feature that we're interested in. Since *any* combination of four distinct points that are collinear qualify as our feature, we can generate *all* combinations of four distinct points and check each combination to see if those four points are collinear. This brute force solution is a *combinatoric* approach to the problem: We generate all the combinations of N things taken four at a time and test each combination based on our feature criteria (collinearity).

For example, let's name the points in the given sample input file p_1 through p_5, as shown below.

5
11000 11000 (p_1)
12000 10000 (p_2)
13000 10000 (p_3)
14000 10000 (p_4)
15000 10000 (p_5)

The table below shows all the combinations of these five points taken four at a time, along with the result of testing each combination for collinearity. Note that to check if four points p, q, r, and s are collinear, we check whether the slope between p and q, between p and r, and between p and s are all equal.

Combination	Collinear?
p_1, p_2, p_3, p_4	no
p_1, p_2, p_3, p_5	no
p_1, p_2, p_4, p_5	no
p_1, p_3, p_4, p_5	no
p_2, p_3, p_4, p_5	*yes*

The advantage of this brute force approach is that it's simple to code. In this assignment, the number of points being selected out of the set of N total points is fixed at four, so four nested `for` loops can be used to generate all the possible combinations. That's also the problem with this brute force approach: four nested loops each dependent on N will have $O(N^4)$ time complexity.

For a given number of points N, we know exactly how many combinations of four points will be computed. That is given by $\binom{N}{k}$ where $k = 4$.

$$\binom{N}{4} = \frac{N!}{4!(N-4)!}$$

For our example above, this would give $\frac{5!}{4!} = 5$ combinations. If the input had 10 points, the brute force solution would have to test $\frac{10!}{4! \times 6!} = \frac{10 \times 9 \times 8 \times 7}{4 \times 3 \times 2} = 210$ different combinations of four points. For $N = 20$, the brute force solution would generate and test 4845 combinations of four points. For $N = 1000$, over *41 billion* combinations of four points would be generated and tested by our program. You can see how this escalates very quickly and the brute force solution becomes impractical to apply as the problem size scales up.

The `getLinesFast` method

We can solve this problem much more efficiently if we use sorting as part of our solution. In a group of collinear points, the slope that any two points make with respect to each other is, of course, the same. For example, in Figure 7 the slope between any two points on l_1 is positive ∞, the slope between any two points on l_2 is 1.0, and the slope between two points on l_3 is -1.0. If we select a reference point and sort all other points with respect to the slope they make with that reference point, then all points mutually collinear with that reference point would be "duplicates" and would therefore be in arranged in continguous groups.

For example, here are the points in Figure 7.

(1, 6)	(2, 2)	(2, 5)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(4, 3)	(4, 4)	(5, 2)	(5, 5)

And here are the points in Figure 7 sorted with respect to the slope they make with (3, 4).

(3, 4)	(1, 6)	(2, 5)	(4, 3)	(5, 2)	(4, 4)	(5, 5)	(2, 2)	(3, 1)	(3, 2)	(3, 3)	(3, 5)
$-\infty$	-1.0	-1.0	-1.0	-1.0	0.0	0.5	2.0	∞	∞	∞	∞

Note how all the points that are mutually collinear with (3, 4) are now in contiguous groups (l_1 is highlighted in light red and l_3 is highlighted in light blue). Underneath each point is the slope it makes with (3, 4)[3].

Similarly, here are the points in Figure 1.

(1, 7)	(2, 2)	(2, 5)	(3, 1)	(4, 4)	(5, 3)	(5, 6)	(6, 6)	(7, 1)	(7, 3)	(7, 4)	(7, 9)	(8, 8)

And here are the points in Figure 1 sorted with respect to the slope they make with (7, 1).

[3]Now do you see why we defined the slope of degenerate lines as negative infinity?

(7, 1)	(6, 6)	(5, 6)	(1, 7)	(4, 4)	(5, 3)	(2, 5)	(2, 2)	(3, 1)	(8, 8)	(7, 3)	(7, 4)	(7, 9)
-∞	-5.0	-2.5	-1.0	-1.0	-1.0	-0.8	-0.2	0.0	7.0	∞	∞	∞

All the points that are mutually collinear with (7, 1) are now in contiguous groups (highlighted). Underneath each point is the slope it makes with (7, 1).

We can now apply the following strategy to solve the problem.

1. Sort the N points with respect to the slope that they make with one of the points p.
2. Scan the sorted points to find all groups of three or more consecutive points having the same slope to p. Each such group is collinear with p and thus, together with p, form a line segment of at least four points.
3. Repeat steps 1 and 2 for the remaining $N - 1$ points.

How much faster is this sort-and-scan approach? We can sort in $O(N \log N)$ time and the subsequent scan is $O(N)$. We have to perform these operations for all N points, so the total cost of this *sort and scan* approach is $N \times (N \log N + N) = N^2 \log N + N^2$ which is $O(N^2 \log N)$. This is a significant asymptotic improvement since $O(N^2 \log N) \prec O(N^4)$, and the clock-time difference is dramatic. Problem sizes that are impractical for the brute force solution are solved quickly (or at least in a reasonable amount of time) by this sort-and-scan solution.

But there is an additional benefit of this approach: We are no longer limited to identifying only four-point line segments. We can now identify *maximal* line segments of four or more collinear points.

Notes and other requirements

Here are a couple of extra requirements plus a few things to keep in mind.

- **Start this one early**. There's more reading, thinking, and up-front understanding to take care of on this assignment. Read this handout carefully. Ask questions of your TA and of me. Ask questions on Piazza. Start early and be proactive.
- You've been provided with the file `stdlib.jar`, which contains easy to use input/output and drawing capabilities along with other things. This file is provided by Princeton University under the GNU General Public License. Its documentation can be found here: `http://introcs.cs.princeton.edu/java/stdlib`. *You are not required to use this file at all.* If you choose to do so, you will have to add the path to this jar file to your CLASSPATH.
- You've been provided with shells of the `Point`, `Line`, and `Extractor` classes. *Don't modify anything that has already been done for you.*
- Start by completing the remaining methods of the `Point` class. There's no *point*[4] in attempting the `List` or `Extractor` class before this is complete and correct, since both depend on `Point`.
- Complete the `Line` class after `Point` but before `Extractor`. Again, since `Extractor` depends on both `Line` and `Point`, you must have these working first.
- When you begin work on the `Extractor` class, start with the `getLinesBrute` method. This is shorter and easier to get correct quickly. Once you are satisfied that `getLinesBrute` is correct, turn your attention to `getLinesFast`.
- In `getLinesFast`, do not print subsegments of lines containing five or more collinear points. For example, if the line segment $p \to q \to r \to s \to t$ exists in the data, you must identify it but you must not identify any four-point subsegment such as $p \to q \to r \to s$.

[4] Sorry; couldn't resist. Here's more: `www.punoftheday.com`

Assignment Submission

You must turn in the following three files to Web-CAT for grading: `Point.java`, `Line.java`, and `Extractor.java`. Note the following rules regarding your Web-CAT submissions:

- Separate submissions for `Point` and `Line` are available on Web-CAT. You can submit an unlimited number of times and these submissions are not counted toward your grade. This is strictly to help you make sure `Point` and `Line` work correctly before you make submissions that include the `Extractor` class.
- You can submit to Web-CAT no more than 20 times for the graded portion of this assignment.
- The *last* submission that you make to Web-CAT will be used to determine your grade on the assignment, even if its score is lower than that of an earlier submission.
- Submissions made within the 24 hour period after the published deadline will be assessed a late penalty of 15 points.
- No submissions will be accepted more than 24 hours after the published deadline.