Com S 228 Fall 2016

Project 2: Sorting Points in the Plane (150 pts)

Due at **11:59pm**

Friday, October 7

1. Project Overview

In computational geometry, algorithms often build their efficiency on processing geometric objects in certain orders that are generated via sorting. For example, Graham's scan constructs the convex hull of an input set of points in the plane in one traversal ordered by polar angle; intersection of a large number of line segments is performed using a sweeping line which makes stops at their endpoints as ordered by x- or y-coordinate.

In this project, you are asked to sort an input set of points in the plane using four sorting algorithms (already or to be) presented in class: selection sort, insertion sort, mergesort, and quicksort. Point comparison is based on either the x-coordinate or the polar angle with respect to some reference point. Your code should provide both options for comparison.

We make the following two assumptions:

a) *All input points have **integral** coordinates ranging between* -50 *and* 50 *inclusive.*
b) *The input points may have **duplicates**.*

Integral coordinates are assumed to avoid issues with floating-point arithmetic. The rectangular range $[-50, 50] \times [-50, 50]$ is big enough to contain 10,201 points with integral coordinates. Since the input points will be either generated as pseudo-random points or read from an input file, duplicates may appear.

1.1 `Point` Class and Comparison Methods

The `Point` class implements the Comparable interface. Its compareTo() method compares the x-coordinates of two points. If the two points have the same x-coordinate, then their y-coordinates are compared.

Point comparison can be also done using an object of the `PolarAngleComparator` class, which you are required to implement. The polar angle is with respect to a point stored in the instance variable `referencePoint`. The `compare()` method in this class must be implemented ***using cross and dot products*** not any trigonometric or square root functions. You need to handle

special situations where multiple points are equal to `lowestPoint`, have the same polar angle with respect to it, etc. Please read the Javadoc for the `compare()` method carefully.

1.2 Sorter Classes

In this project, selection sort, insertion sort, mergesort, and quicksort are respectively implemented by the classes `SelectionSorter`, `InsertionSorter`, `MergeSorter`, and `QuickSorter`, all of which extend the abstract class `AbstractSorter`. There are two constructors of this abstract class that await your implementation:

```
protected AbstractSorter(Point[] pts) throws IllegalArgumentException
protected AbstractSorter(String inputFileName) throws FileNotFoundException,
                                                      InputMismatchException
```

The first constructor takes an existing array `pts[]` of points, and copy it over to the array `points[]`. It throws an `IllegalArgumentException` if `pts == null` or `pts.length == 0`.

The second constructor reads points from an input file of integers and stores them in `points[]`. Every pair of integers represents the x and y-coordinates of some point. A `FileNotFoundException` will be thrown if no file by the `inputFileName` exists, and an `InputMismathException` will be thrown if the file consists of an odd number of integers. (There is no need to check if the input file contains unneeded characters like letters since they can be taken care of by the `hasNextInt()` and `nextInt()` methods of a `Scanner` object.)

Each of the four subclasses `SelectionSorter`, `InsertionSorter`, `MergeSorter`, and `QuickSorter` has two constructors that need to call their corresponding superclass constructors above.

For example, suppose a file `points.txt` has the following content:

```
0  0 -3 -9  0 -10
8 4 3 3 -6
3 -2 1
10 5
-7 -10
5 -2
7 3  10 5
-7 -10 0 8
-1 -6
-10 0
5 5
```

There are 34 integers in the file. A call `AbstractSort("points.txt")` will initialize the array `points[]` to store 17 points below (aligned with five points per row just for display clarity here):

```
(0, 0)   (-3, -9)  (0, -10)   (8, 4)     (3, 3)
(-6, 3)  (-2, 1)   (10, 5)    (-7, -10)  (5, -2)
(7, 3)   (10, 5)   (-7, -10)  (0, 8)     (-1, -6)
(-10, 0) (5, 5)
```

Note that the points `(-7, -10)` and `(10, 5)` each appear twice in the input, and thus their second appearances are duplicates. The 15 distinct points are plotted nicely in Fig. 1 by Mathematica. (In this project, you are provided an implemented class `Plot` using the Java Swing to plot sorting results.)

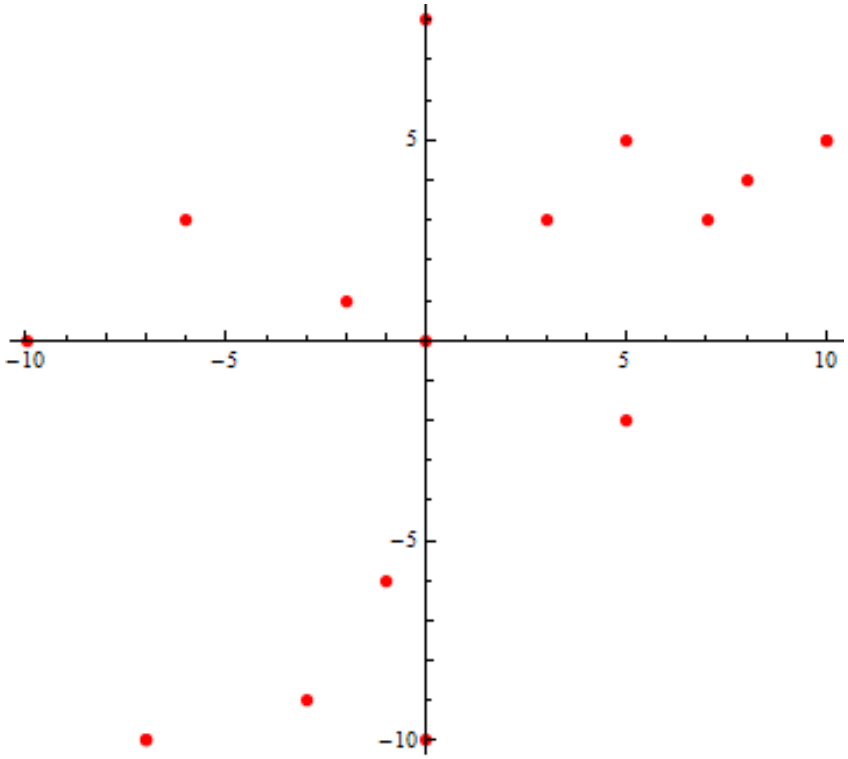

Fig. 1. Sample input set contains 15 different points.

Besides having an array `points[]` to store points, the `AbstractSorter` class also includes six instance variables.

- `algorithm`: type of sorting algorithm. Initialized by a subclass constructor.
- `sortByAngle`: sorting by polar angle or x-coordinate. Set within `sort()`.
- `outputFileName`: name of the file to store the sorting result in: `select.txt`, `insert.txt`, `merge.txt`, or `quick.txt`. Set by a subclass constructor.
- `sortingTime`: sorting time in nanoseconds. It can be set, for instance, within `sort()` using the `System.nanoTime()` method.
- `pointComparator`: comparator used for point comparison. Set by calling `setComparator()` within `sort()`.

- `lowestPoint`: lowest point in the array `points[]`. Initialized by a constructor of `AbstractSorter`.

In the previous example, two points (-7, -10) and (0, -10) tie for the lowest point. The variable `lowestPoint` is set to the first point because it is to the left. After sorting by ***increasing*** x-coordinate, the array `points[]` will store the 17 points in the following order:

```
(-10, 0)  (-7, -10)  (-7, -10)  (-6, 3)  (-3, -9)
(-2, 1)   (-1, -6)   (0, -10)   (0, 0)   (0, 8)
(3, 3)    (5, -2)    (5, 5)     (7, 3)   (8, 4)
(10, 5)   (10, 5)
```

The `lowestPoint` is at index 0. Also, note that the three points `(0, -10)`, `(0, 0)`, `(0, 8)` have the same x-coordinate `0`. Their order is determined by their y-coordinates. The same applies to `(5, -2)` and `(5, 5)`.

After sorting by ***increasing*** polar angle, the same array will store the points in a different order below:

```
(-7, -10) (-7, -10)  (0, -10)  (-3, -9)  (-1, -6)
(5, -2)   (10, 5)    (10, 5)   (7, 3)    (8, 4)
(5, 5)    (3, 3)     (0, 0)    (-2, 1)   (0, 8)
(-6, 3)   (-10, 0)
```

Among them, `(-1, -6)` and `(5, -2)` have the same polar angle with respect to `(-7, -10)`. They are thus ordered by distance to this point.

2. Compare Sorting Algorithms

The class `CompareSorters` executes the four sorting algorithms on points randomly generated or read from files. Over each input sequence, its `main()` method compares the execution times of these algorithms in multiple rounds. Each round proceeds as follows:

a) Create an array of randomly generated integers, if needed.

b) For each of the four classes `SelectionSorter`, `InsertionSorter`, `MergeSorter`, and `QuickSorter`, create an object of the class from the above array or an input file.

c) Have the four created objects call the `sort()` method and store their results in `points[]`.

Below is a sample execution sequence with running times. Use the `stats()` method to create the row for each sorting algorithm in the table.

```
Comparison of Four Sorting Algorithms
```

```
keys:  1 (random integers)  2 (file input)  3 (exit)
order: 1 (by x-coordinate)  2 (by polar angle)

Trial 1: 1
Enter number of random points: 1000
Order used in sorting: 1


algorithm        size     time (ns)
-----------------------------------

selection sort   1000     9200867
insertion sort   1000     10306807
mergesort        1000     1272351
quicksort        1000     765669
-----------------------------------


Trial 2: 2
Points from a file
File name: points.txt
Order used in sorting: 2


algorithm        size     time (ns)
-----------------------------------

selection sort   1000     27168362
insertion sort   1000     23314848
mergesort        1000     2455696
quicksort        1000     531187
-----------------------------------
...
```

Your code needs to print out the same text messages for user interactions. Entries in every column of the output table need to be aligned.

3. Random Point Generation

To test your code, you may generate random points within the range $[-50, 50] \times [-50, 50]$. Such a point has its x- and y-coordinates generated separately as pseudo-random numbers within the range $[-50, 50]$. You already had experience with random number generation from Project 1. Import the Java package `java.util.Random`. Next, declare and initiate a Random object like below

```
Random generator = new Random();
```

Then, the expression

```
generator.nextInt(101) – 50
```

will generate a pseudo-random number between -50 and 50 every time it is executed.

4. Display Sorting Results

The sorted points will be displayed using Java graphics package Swing. The display will help you visually check that the points are correctly sorted. The fully implemented class `Plot` is for this purpose. A few things about the implementation to note below:

- The `JFrame` class is a top level container that embodies the concept of a "window". It allows you to create an actual window with customized attributes like size, font, color, etc. It can display one or more `JPanel` objects in the same time.

- `JPanel` is for painting and drawing, and must be added to the `JFrame` to create the display. A `JPanel` represents some area in a `JFrame` in which controls such as buttons and textfields and visuals such as figures, pictures, and text can appear.

- The `Graphics` class may be thought of like a pen that does the actual drawing. The class is abstract and often used to specify a parameter of some method (in particular, `paint()`). This parameter is then downcast to a subclass such as `Graphics2D` for calling the latter's utility methods.

- The `paint()` method is called automatically when a window is created. It must be overridden to display your drawings.

The results of the four sorters are displayed in separate windows. For display, a separate thread is created inside the method `myFrame()` in the class `Plot`.

Please do ***not*** modify the `Plot` class for better display unless you understand what is going on there. (Anyway, the quality of display will never match that created by software like Mathematica or Matlab.)

The class `Segments` has been implemented for creating specific line segments to connect the input points so you can see the correctness of the sorting result.

4.1 Drawing Data Preparation

The output of each sorter can be displayed by calling the partially implemented method `draw()` in the `AbstractSorter` class, ***only after*** `sort()` is called, via the statement below:

```
Plot.myFrame(points, segments, getClass().getName());
```

where the first two parameters have types `Point[]` and `Segment[]`, respectively. By this time, the array `points[]` stores the sorted points. The call `getClass().getName()` simply returns the name of the sorter used. From `points[]` you will need to generate some line segments which, when drawn, can visually reveal the order among the points. To be stored in the array `segments[]`, these line segments are created according to the value of the instance variable `sortByAngle` in `AbstractSorter`.

a) If `sortByAngle==false`, create a line segment to connect every two adjacent points in `Point[]`.

b) If `sortByAngle==true`, create a line segment to connect `lowestPoint` to every other point in the array `points[]`. Also, create a line segment to connect `points[i]` and `points[i+1]` if they are ***distinct*** points, for every valid index `i > 0`.

In either case above, the order among the elements in `segments[]` may be arbitrary. Consider the same 17 input points (with duplicates) shown in Fig. 1. After sorting by x-coordinate, a sample content of `segments[]` lists 14 segments as below (where each element is shown as a pair of points):

```
((-10, 0), (-7, 10))
((-7, -10), (-6, 3))
((-6, 3), (-3, -9))
((-3, -9), (-2, 1))
((-2, 1), (-1, -6))
((-1, -6), (0, -10))
((0, -10), (0, 0))
((0, 0), (0, 8))
((0, 8), (3, 3))
((3, 3), (5, -2))
((5, -2), (5, 5))
((5, 5), (7, 3))
((7, 3), (8, 4))
((8, 4), (10, 5))
```

If after sorting by polar angle, a sample content of `segments[]` consists of the following 27 segments, the first 14 of which are concurrent at `lowestPoint == (-7, -10)`:

```
((-7, -10), (0, -10))
((-7, -10), (-3, -9))
((-7, -10), (-1, -6))
((-7, -10), (5, -2))
((-7, -10), (10, 5))
((-7, -10), (7, 3))
((-7, -10), (8, 4))
((-7, -10), (5, 5))
((-7, -10), (3, 3))
((-7, -10), (0, 0))
((-7, -10), (-2, 1))
((-7, -10), (0, 8))
```

```
((-7, -10), (-6, 3))
((-7, -10), (-10, 0))


((0, -10), (-3, -9))
((-3, -9), (-1, -6))
((-1, -6), (5, -2))
((5, -2), (10, 5))
((10, 5), (7, 3))
((7, 3), (8, 4))
((8, 4), (5, 5))
((5, 5), (3, 3))
((3, 3), (0, 0))
((0, 0), (-2, 1))
((-2, 1), (0, 8))
((0, 8), (-6, 3))
((-6, 3), (-10, 0))
```

4.2. Display Sorting Result

Suppose that `draw()` is called after sorting by x-coordinate. The 14 segments from the first group in Section 4.1 will be created. The sorter's display window will show a polyline that starts at the leftmost point `(-10, 0)` and ends at the rightmost point `(10, 5)`. The polyline, drawn in Fig. 3 (again by Mathematica just for a nicer display), always goes either to the right or upward in this traversal.

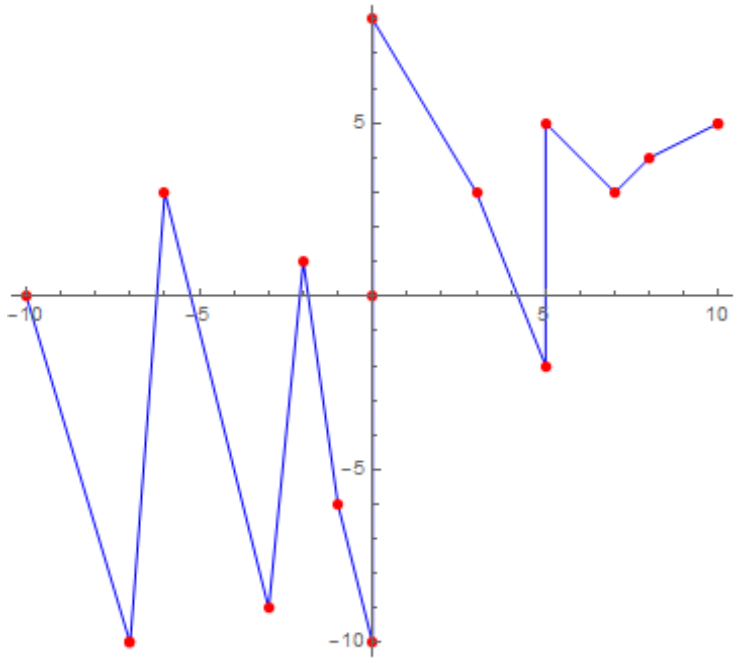

Fig. 3. Polyline connecting all input points in Fig. 1 left-to-right (or bottom-up).

If your sorting result is incorrect, then the displayed polyline will turn either leftward or downward at some point.

Suppose that sorting was done by polar angle with respect to the lowest point `(-7, -10)`. The second group of 27 segments from Section 4.1 will be created. The sorter's display window will show a triangulation the same as the one plotted by Mathematica in Fig. 4.

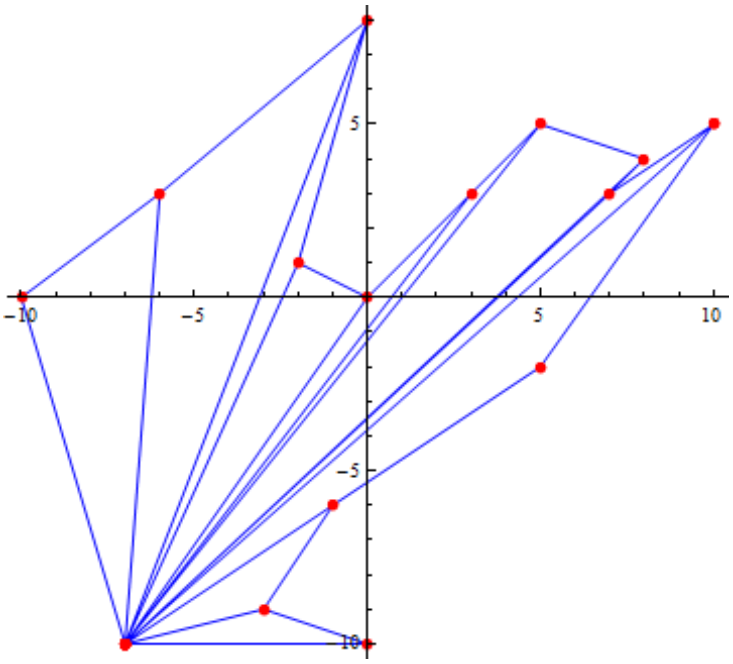

Fig. 4. Triangulation of a simple polygon whose vertices (i.e., the input points) are in order of increasing polar angle with respect to (-7, -10).

The outer boundary of the triangulation is a simple polygon along which a counterclockwise traversal starting at the lowest point `(-7, -10)` will never decrease the polar angle with respect to this point. The edges lie on the polygon. The diagonals are the line segments connecting `(-7, -10)` to other points and lying inside the polygon. If your sorting result is correct, no two line segments, whether a diagonal or a polygon edge, will intersect at a point in their interiors.

5. Submission

Write your classes in the `edu.iastate.cs228.hw2` package. **Turn in the zip file not your class files.** Please follow the guideline posted under Documents & Links on Blackboard Learn.

You are not required to submit any JUnit test cases. Nevertheless, you are encouraged to write JUnit tests for your code. Since these tests will not be submitted, feel free to share them with other students.

Include the Javadoc tag `@author` in every class source file you have made changes to. Your zip file should be named `Firstname_Lastname_HW2.zip`.