Com S 228 Fall 2015

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

Due at **11:59pm**

Monday, October 12

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 a number of input 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 that makes stops at the endpoints of these segments that are ordered by x- or y-coordinate.

In this project, you are asked to sort an input set of points in the plane using the four sorting algorithms 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 with the compareTo() method to compare 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 such as 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 Fig. 1 using Mathematica.

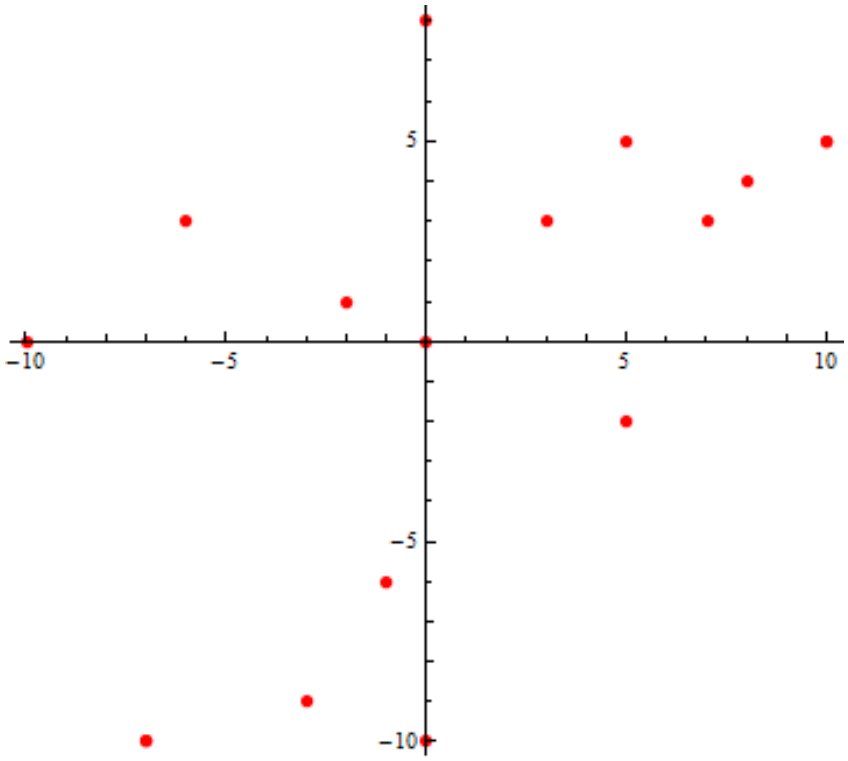

Fig. 1. Input set contains 15 different points.

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 on the left. After sorting by 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)
```

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 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 respectively in the files `select.txt`, `insert.txt`, `merge.txt`, and `quick.txt`.

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 in Mathematica

The sorted points can be displayed in Mathematica. This will help you visualize the geometry and check the correctness of sorting. The software is licensed by ISU and available to students. To install Mathematica on your computer, please follow the instructions in the online document below:

https://www.it.iastate.edu/sldb/docs/Mathematica-Install-for-Students.pdf

Setting up Mathematica and doing the test take around three hours, most of which will likely be spent on waiting through.

4.1. File Preparation

The method writePointsToFile() in the AbstractSorter class writes the sorted points (in order of increasing index) from the array points[] to a file with provided name, under certain format that is decided by the sorting order indicated by the value of the instance variable sortByAngle in AbstractSorter. In the output file, adjacent coordinates must have **at least one blank space** in between. The format is detailed below:

a) If sortByAngle==false, every line displays the x- and y-coordinates of a single point. This format will allow Mathematica to generate a monotonic polyline connecting the points. Section 4.3 will give an example.

b) If sortByAngle==true, the first line displays the x- and y-coordinates of points[0], i.e., lowestPoint. The i-th line, where i > 1, displays the x and y-coordinates of points[i-1], followed by those of points[0], and then those of points[i-1] again. This format will allow Mathematica to plot a triangulation of the points (see Section 4.3).

For the same 17 input points (with duplicates) shown in Fig. 1, after sorting by x-coordinate, a call writePointsToFile() will generate the file sortedPoints.txt with the content below:

```
-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
```

If after sorting by polar angle, the file `sortedPoints.txt` will have the following different content:

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

In fact, Mathematica can read in the integer coordinates to form points correctly no matter how many of them are displayed on each line.

4.2. Command Execution

After installation of Mathematica, run the software. Click “File” on the upper left of the top menu bar, and select “New” in the opened menu, and then “Notebook” in the resulting submenu. See the screenshot below. This creates an input window (like the one named “Untitled-3” in the screenshot), called a ***notebook***, within which Mathematica commands will be executed and the results (including graphics) will be displayed.

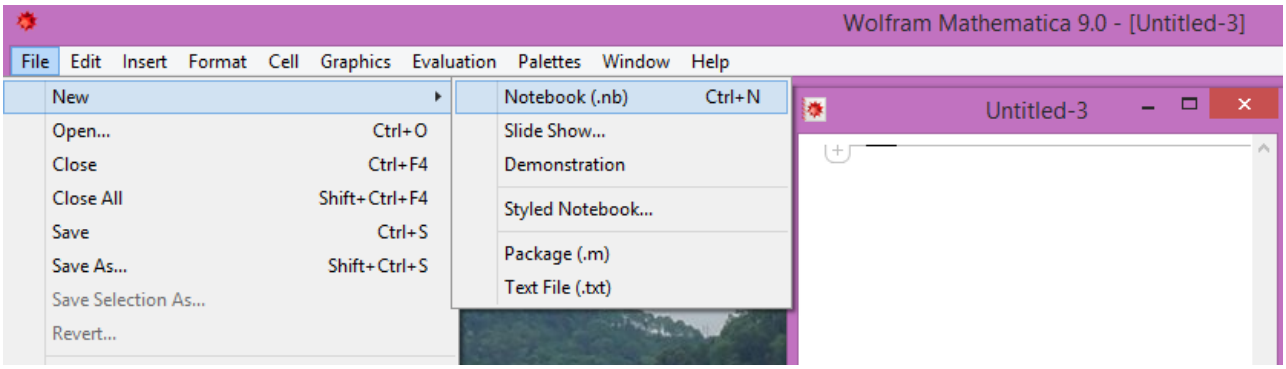

Fig. 2. Mathematica notebook window.

Within this notebook window, you may type a command like

```
Graphics[Line[{{1, 1}, {3, 3}}]]
```

and then hit "Shift+Enter" (i.e., hit the key "Enter" while pressing the key "Shift"). This will execute the command. A line segment connecting the two points (1, 1) and (3, 3) will be drawn (below the command line). Similarly, executing the following two commands separately will plot a red disk (of default radius 0.01) and a blue disk (of radius 0.02) at the same point (2, 2).

```
Graphics[{Red, Point[{2, 2}]}]
Graphics[{Blue, PointSize[0.02], Point[{2, 2}]}]
```

You may plot the above line segment and the red disk together, by executing the following compound command:

```
Show[Graphics[Line[{{1, 1}, {3, 3}}]],
     Graphics[{Red, Point[{2, 2}]}],
     Axes -> True, AspectRatio -> Automatic, PlotRange -> All]
```

The generated figure will show the x- and y- axes since the `Axes` option is set to `true`. You may turn the option off by setting it to `false`, or simply remove "`Axes -> True,`" from the command. The `AspectRatio` option specifies the ratio of height to width. By setting it to `Automatic`, Mathematica will choose the ratio based on the actual plot values. The `PlotRange` option is set to prevent Mathematica from exercising its liberty to show only a portion of the plot that it "deems" important.

In this project, the **Mathematica commands for display will be given to you** in Section 4.3. However, in case you want to learn more about the software (especially its graphics), you may search in the following online reference for many more plotting commands:

http://reference.wolfram.com/language/

Unfortunately, the relevance of the returned links from a search directly within the above website is often low. Googling with a keyword/question plus "mathematica" would usually locate more relevant documents in this website.

4.3. Display Sorting Result

In this project, the files `sortedPoints.txt` generated by the method `sort()` are in the same folder that contains the `src` folder. To let Mathematica know where to look for these files, you need to set this folder containing the `src` folder as the working directory. Since Mathematica has its own string format for a directory, the easiest way to find the string representing the working directory is as follows.

- On the top menu bar of Mathematica, click "Insert".
- Select "File Path" in the opened menu.
- This opens up a window. Opens the file, say, `sortedPoints.txt`.
- In the notebook window, the full path of the file will be shown as, for instance,

```
"C:\\Users\\jia\\Documents\\courses\\Com S 228\\java\\228 F15 Projects\\
sortedPoints.txt"
```

  Note the double backslashes separating directory names in the above string. To obtain the string for the directory, remove "`\\sortedPoints.txt`".

- Back in the notebook, execute the `SetDirectory` command below via cut-and-paste:

```
SetDirectory["C:\\Users\\jia\\Documents\\courses\\Com S 228\\java\\228 F15
Projects"]
```

This will set up Mathematica's working directory. To display the sorting result, simply **cut-and-paste the command sequence below into the notebook** and then hit "Shift+Enter":

```
Clear[points];
points = ReadList["sortedPoints.txt", {Number, Number}];
Show[Graphics[{Blue, Line[points]}],
     Graphics[ListPlot[points, PlotStyle -> {PointSize[0.015], Red}]],
     Axes -> True, AspectRatio -> Automatic, PlotRange -> All]
```

The first command above clears the current content of the Mathematica variable `points`. The second command reads the input points from the file "`sortedPoints.txt`" with the format `{Number, Number}` specifying that every two numbers corresponding to the x- and y-coordinates of one point. Tabs and newline characters are ignored in the reading.

The third command `Show` draws both the convex hull and the input points. If you remove the option "`Axes -> True,`" from the compound command, the axes will be turned off.

Suppose the file `sortedPoints.txt` lists points ordered by x-coordinate, i.e., it contains the first point sequence in Section 4.1. The above three Mathematica commands will generate a polyline that starts at the leftmost point `(-10, 0)` and ends at the rightmost point `(10, 5)`. The polyline, shown in Fig. 3, always goes either to the right or upward in this traversal.

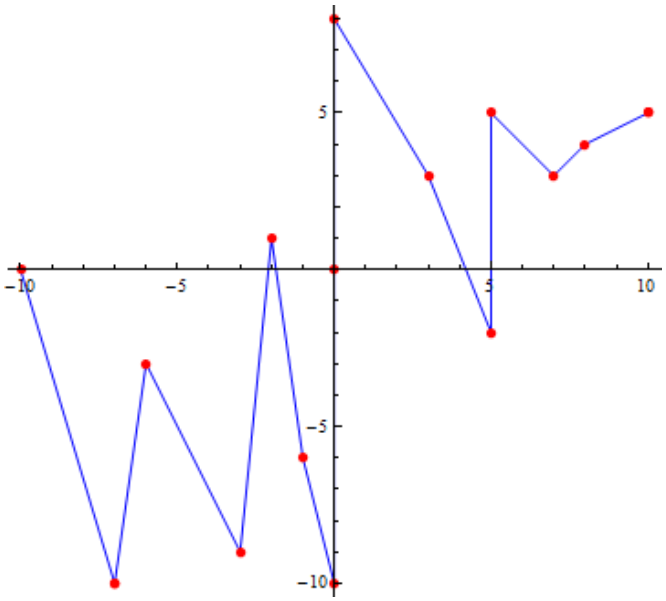

Fig. 3. Polyline connecting all input points in Fig. 1 left-to-right (or bottom-up).

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

Suppose the file `sortedPoints.txt` lists points ordered by polar angle with respect to the lowest point `(-7, -10)`, i.e., it contains the second sequence of points in Section 4.1. The same three Mathematica commands will generate a triangulation shown in Fig. 4.

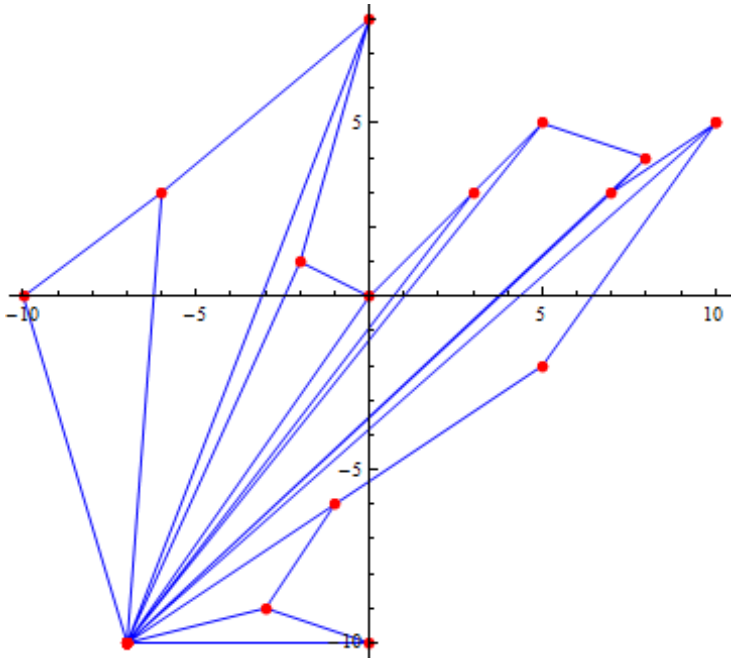

Fig. 4. Triangulation of a simple polygon whose vertices (i.e., the input points) are in order of increasing polar angle with respect to (-7, -10).

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`.