PROJECT 2, Com S 228, Spring 2020

Due at 11:59pm, Saturday, March 7

1 Project Overview

In computational geometry, algorithms often build their efficiency on processing geometric objects in certain orders that are generated via sorting. One example is a *Graham scan*, which constructs the convex hull of a collection of points in the plane in one traversal ordered by the polar angle. In another example, intersections of a set of line segments can be found using a sweep line that makes stops only at the segments' endpoints and intersections. ordered by x- or y-coordinates.

In this project, you are asked to scan an input set of points in the plane in the order of their polar angles with respect to a reference point. This reference point, called the *median coordinate point* (MCP), has its x-coordinate equal to the median of the x-coordinates of the input points and its y-coordinate equal to the median of their y-coordinates. A polar angle with respect to this point lies in the range $[0, 2\pi)$.

Finding the median x- and y-coordinates is done by sorting the points separately by the corresponding coordinate.[1] Once the MCP has been constructed, a third round of sorting will be conducted by the polar angle with respect to this point.

You need to scan the input points four times, each time using one of the four sorting algorithms presented in class: selection sort, insertion sort, merge sort, and quicksort. Note that the **same sorting algorithm** must be used in all three rounds of sorting within one scan.

We make the following two assumptions:

- All input points have **integer** coordinates ranging between -50 and 50, inclusive.
- The input points may have **duplicates**.

Integer 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 integer coordinates.

Since the input points will be either generated as pseudo-random points or read from an input file, duplicates may appear. Also, it is unnecessary to test your code on more than 10,201 input points (otherwise duplicates will certainly arise).

1.1 `Point` Class and Comparison Methods

The `Point` class implements the `Comparable` interface. Its `compareTo()` method compares the x- or y- coordinates of two points. In case two points tie on their values of the selected coordinate, the values of the other coordinate are compared to break the tie.

`Point` comparison can also be 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

[1] In Com S 311 you will learn an algorithm that can find the median of n numbers in $O(n)$ time, but in practice it often takes too long due to a large constant factor hidden inside the Big-O.

and dot products, not any trigonometric or square root functions. You need to handle special situations where multiple points are equal to `referencePoint`, have the same polar angle with respect to it, etc. Please read the Javadoc for the methods `compare()` and `comparePolarAngle()` carefully.

1.2 Sorter Classes

Selection sort, insertion sort, merge sort, and quicksort are respectively implemented by the classes `SelectionSorter`, `InsertionSorter`, `MergeSorter`, and `QuickSorter`, all of which extend the abstract class `AbstractSorter`. The abstract class has one constructor awaiting your implementation:

```
protected AbstractSorter(Point[] pts) throws IllegalArgumentException
```

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

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

- `algorithm`: type of sorting algorithm to be initialized by a subclass constructor.
- `pointComparator`: comparator used for point comparison. Set by calling `setComparator()`. It compares two points by their x-coordinates, y-coordinates, or polar angles with respect to `referencePoint`.
- `referencePoint`: reference point that polar angles are taken with respect to. Set by calling `setReferencePoint()`.

The method `sort()` conducts sorting, for which the algorithm is determined by the dynamic type of the `AbstractSorter`. The method `setComparator()` must have been called beforehand to generate an appropriate comparator for sorting by the x-coordinate, y-coordinate, or polar angle. In the case that the polar angle is used, `referencePoint` must have a value that is not `null`.[2]

The class also provides two methods: `getPoints()` to get the contents of the array `points[]`, and `getMedian()` to return the element with the median index in `points[]`.

Each of the four subclasses `SelectionSorter`, `InsertionSorter`, `MergeSorter`, and `QuickSorter` has a constructor that needs to call the superclass constructor.

1.3 `RotationalPointScanner` Class

This class has two constructors. Both accept one type of sorting algorithm. The first constructor reads points from an array. The second one reads points from an input file of integers, where each pair of integers represents the x and y-coordinates of one point. A `FileNotFoundException` will be thrown if no file by the `inputFileName` exists, and an `InputMismatchException` will be thrown if the file consists of an odd number of integers.[3]

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

[2] Use the method `setReferencePoint()` to set a value.

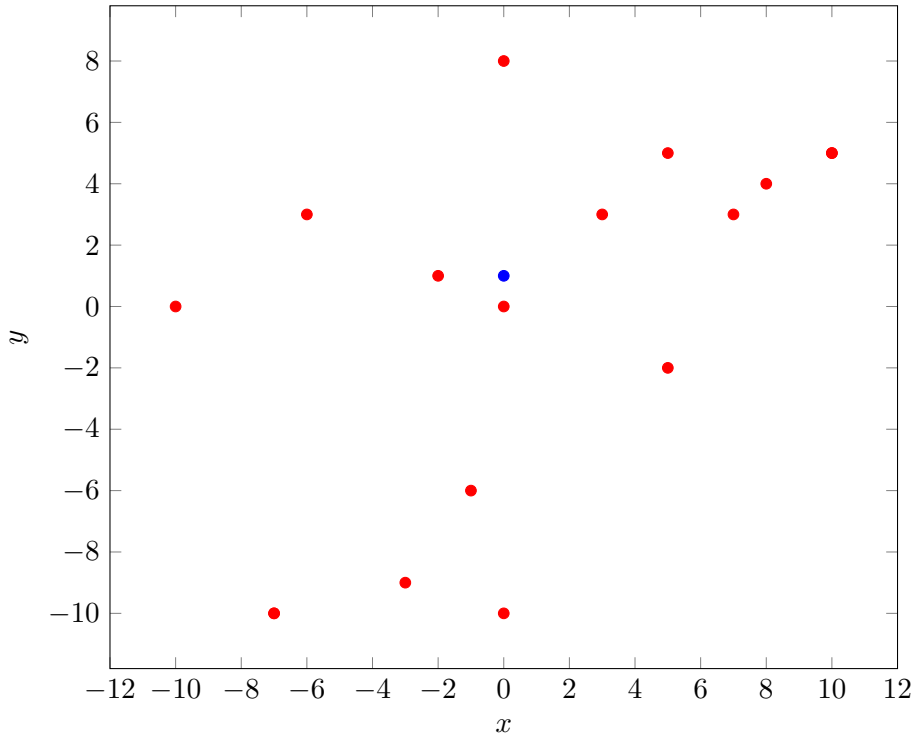
[3] There is no need to check if the input file contains unneeded characters like letters since they can be taken care of by the `hasNextInt()` and `nextInt()` methods of a `Scanner` object.

```
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

```
RotationalPointScanner(points.txt, Algorithm.QuickSort);
```

will initialize the array `points[]` to store the following 17 points, which are plotted[4] below, along with their MCP: $(0, 0)$, $(-3, -9)$, $(0, -10)$, $(8, 4)$, $(3, 3)$, $(-6, 3)$, $(-2, 1)$, $(10, 5)$, $(-7, -10)$, $(5, -2)$, $(7, 3)$, $(10, 5)$, $(-7, -10)$, $(0, 8)$, $(-1, -6)$, $(-10, 0)$, and $(5, 5)$.



Note that the points `(-7, -10)` and `(10, 5)` each appear twice in the input.

The 17 points have x-coordinates in the following sequence:

$$0, -3, 0, 8, 3, -6, -2, 10, -7, 5, 7, 10, -7, 0, -1, -10, 5$$

which, after sorted in the non-decreasing order, becomes,

$$-10, -7, -7, -6, -3, -2, -1, 0, 0, 0, 3, 5, 5, 7, 8, 10, 10$$

[4] They were plotted here using the LaTeX package `pgfplot`. In this project, you are provided an implemented class `Plot` using the Java graphics package Swing for display of results.

Since the largest index in the private array `points[]` storing the points is 16, the median is 0 at the index $16/2 = 8$. (Note that integer division in Java truncates the fractional part, if any, of the result.) Similarly, the above points have y-coordinates in the following sequence:

$$0, -9, -10, 4, 3, 3, 1, 5, -10, -2, 3, 5, -10, 8, -6, 0, 5$$

which is in the non-decreasing order below.

$$-10, -10, -10, -9, -6, -2, 0, 0, 1, 3, 3, 3, 4, 5, 5, 5, 8$$

The median y-coordinate is 1 at the index 8. The median coordinate point (MCP) is therefore $(0, 1)$, colored blue in the plot. Notice that it does not coincide with any of the input points.

Construction of the MCP is carried out by the method `scan()` of the class by sorting the points by x- and y-coordinates, respectively. After these two sorting rounds, the method sets the value of `medianCoordinatePoint` to the MCP, and then calls the method `setReferencePoint()` of the `AbstractSorter` class to set the value of its instance variable `referencePoint` to the MCP. The method `scan()` then carries out a final round of sorting by polar angle with respect to the MCP.

Besides using an array `points[]` to store points and `medianCoordinatePoint` to store the MCP, the `RotationalPointScanner` class also includes several other instance variables.

- `sortingAlgorithm`: type of sorting algorithm. Initialized by a constructor.
- `outputFileName`: name of the file to store the sorting result in: `select.txt`, `insert.txt`, `merge.txt`, or `quick.txt`.
- `scanTime`: sorting time in nanoseconds. This sums up the times spent on three rounds of sorting. Within `sort()`, use the `System.nanoTime()` method.

In the previous example, after calling `scan()`, the array `points[]` will store the points ordered by the polar angle with respect to the MCP: $(7, 3)$, $(8, 4)$, $(10, 5)$, $(10, 5)$, $(3, 3)$, $(5, 5)$, $(0, 8)$, $(-6, 3)$, $(-2, 1)$, $(-10, 0)$, $(-7, -10)$, $(-7, -10)$, $(-3, -9)$, $(-1, -6)$, $(0, 0)$, $(0, -10)$, $(5, -2)$.

Among them, $(0, 0)$ and $(0, -10)$ have the same polar angle. They are thus ordered by distance to this point.

2 Compare Sorting Algorithms

The class `CompareSorters` uses the class `RotationalPointScanner` to scan points randomly generated or read from files four times, each time using a different sorting algorithm. Multiple input rounds are allowed. In each round, the `main()` method compares the execution times of the four scans of the same input sequence. The round proceeds as follows:

1. Create an array of randomly generated integers, if needed.
2. Construct four `RotationalPointScanner` objects over the point array, each with a different algorithm (i.e., a different value for the parameter `algo` of the constructor).
3. Have every created `RotationalPointScanner` object call `scan()`.
4. At the end of the round, output the statistics by having every `RotationalPointScanner` object call the `stats()` method.

Below is a sample execution sequence with running times.

```
Performances of Four Sorting Algorithms in Point Scanning
keys: 1 (random integers) 2 (file input) 3 (exit)
Trial 1: 1
Enter number of random points: 1000
algorithm size time (ns)
---------------------------------
SelectionSort 1000 49631547
InsertionSort 1000 22604220
MergeSort 1000 2057874
QuickSort 1000 1537183
---------------------------------
Trial 2: 2
Points from a file
File name: points.txt
algorithm size time (ns)
---------------------------------
SelectionSort 1000 3887008
InsertionSort 1000 9841766
MergeSort 1000 1972146
QuickSort 1000 888098
---------------------------------
...
```

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

3 Random Point Generation

To test your code, you may generate random points within the range $[-50, 50] \times [-50, 50]$. Each random 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 with

```
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 of a Point Scan

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 at 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 text fields 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 scans 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.

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 scan can be displayed by calling the partially implemented method `draw()` in the `RotationalPointScanner` class, **only after** `scan()` is called, via the statement

```
Plot.myFrame(points, segments, sort);
```

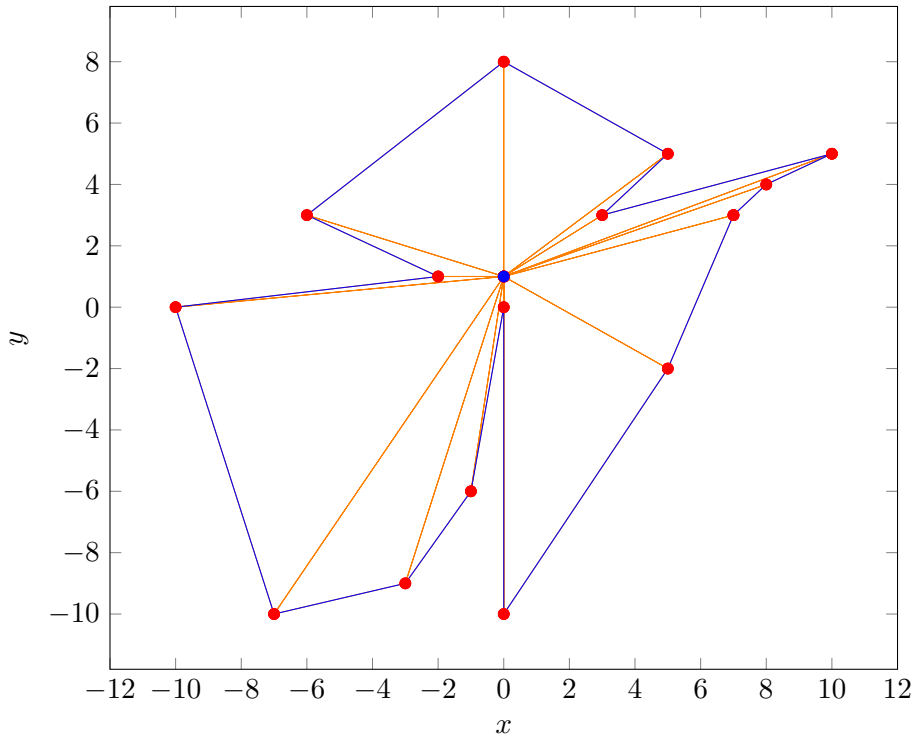
where the first two parameters have types `Point[]` and `Segment[]`, respectively, and the third parameter is a string name for the sorting algorithm used. By this time, the array `points[]` stores the points in the order of scan. You will need to create an array `segments[]` to store some line segments which, when drawn, can visually reveal the order among the points. Create line segments to connect:

- every pair of consecutive points in `points[]` that are distinct from each other,
- the first (index `0`) and last points in `points[]` if they are distinct, and
- `medianCoordinatePoint` to every point in `points[]`.

The order among the elements in `segments[]` may be arbitrary. Consider the same 17 input points (with duplicates) that were plotted above. After a scan, `segments[]` would consist of the following 30 segments, the last 15 of which share an endpoint at `medianCoordinatePoint == (0, 1)`:

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

4.2 Displaying the Result From a Scan



For the 30 segments listed in Section 4.1, the corresponding display window will show a triangulation like the one above. (You don't need to use different colors to emulate this plot.)

The outer boundary of the triangulation is a simple polygon (drawn in blue) with the 15 distinct input points as vertices. A counterclockwise traversal of the polygon vertices starting at $(7, 3)$ will never decrease the polar angle with respect to the median coordinate point $(0, 1)$. Fifteen orange line segments connect $(0, 1)$ to the vertices. If your scanning result is correct, no two line segments should 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. Follow the Submission Guide posted on Canvas. Include the Javadoc tag `@author` in each class source file. Your zip file should be named `Firstname_Lastname_HW2.zip`