Com S 2280 Spring 2024

Homework 2: Sorting 2D Integer Points

Due at **11:59pm**

Wednesday, Oct 16

0. Extra Credit

This project can be submitted up to 10 days early for up to 10% extra credit. A project submitted 11 days early will still only accrue 10% extra credit. The 10% will be multiplied by the final score.

1. Project Overview

In this project, you are asked to read an input set of 2D integer points in a coordinate plane. The goal is to find the ***median coordinate point***, whose x-coordinate is equal to the median of the x-coordinates of the input points and its y-coordinate is equal to the median of their y-coordinates. Finding the median x- and y-coordinates is done by sorting the points separately by the corresponding coordinate.

You need to read the input points four times, each time using one of the following four sorting algorithms: selection sort, insertion sort, merge sort, and quicksort. Note that the ***same sorting algorithm*** must be used in both rounds of sorting after the points have been read.

We make the following two assumptions:
a) All input points have integer coordinates ranging between −50 and 50 inclusive.
b)The input points may have duplicates.

The rectangular range $[-50, 50] \times [-50, 50]$ is big enough to contain 10,201 points with integer coordinates. The input points will be either generated randomly or read from an input file. Duplicates may be present. It is unnecessary to test your code on more than 10,201 points.

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 have the same value along one coordinate, compare the values of the other coordinate.

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 two instance variables.

- `algorithm`: type of sorting algorithm to be initialized by a child constructor.
- `pointComparator`: comparator used for point comparison. Set by calling `setComparator()`. It compares two points by their x-coordinates or y-coordinates

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 or y-coordinate.

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 `PointScanner` 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 every 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 `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. Integer values may be separated by tab, newline and/or space(s).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

```
PointScanner("points.txt", Algorithm.QuickSort)
```

will initialize the array `points[]` to store 17 points below:

```
(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.

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
```

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 is therefore (0, 1), colored blue in Fig. 1, which does not coincide with any of the input points.

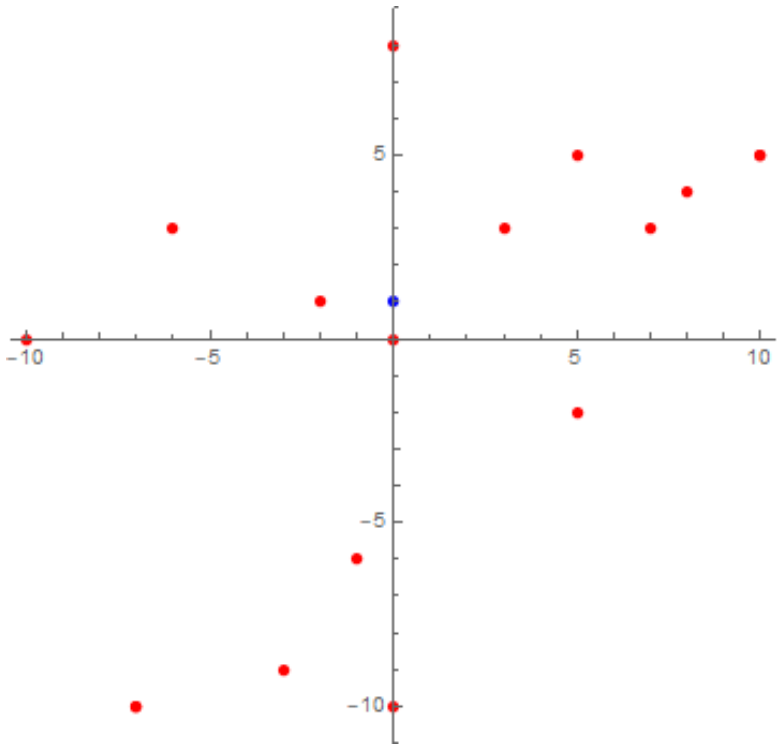

Fig. 1. Sample input set containing 15 different points with median coordinate point at (0, 1).

Determination of the median coordinate point 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.

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

- `sortingAlgorithm`: type of sorting algorithm. Initialized by a constructor.

- `scanTime`: sorting time in nanoseconds. This sums up the times spent on two rounds of sorting per algorithm. Within `sort()` use the `System.nanoTime()` method.

2. Compare Sorting Algorithms

The class `CompareSorters` uses the class `PointScanner` 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:

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

b) Construct four `PointScanner` objects over the point array, each with a different algorithm (i.e., a different value for the parameter `algo` of the constructor).

c) Have every created `PointScanner` objects call `scan()`.

d) At the end of the round, output the statistics by having every `PointScanner` 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 above 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 $[-50, 50] \times [-50, 50]$. 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. Submission

Write your classes in the `edu.iastate.cs2280.hw2` package. **Turn in the zip file, not your class files.** Zip all source files and no class files.

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`.

5. Hints

a. When writing a comparator class, you can wrap its method around an existing compareTo() method.
b. The implementation of MergeSort is much easier if method(s) return arrays as return values, and not as parameters.