AP Computer Science A
Problem Set 6

Dr. Balchunas

Instructions

In this problem set, rename the .java template file to FirstLast followed by the pset number. For example, my file would be AndrewBalchunasPSet5.java. My driver class will also be called AndrewBalchunasPSet5. Questions will be answered inside of methods named by the example code after the question. Do **NOT** rename these methods or change their headers in any way. However, you may choose to create other helper methods that are called by these methods. You might want to test your methods by using a main, but be sure to comment out your main before submitting the assignment. I suggest making a separate driver class that you use to test your methods. You can then instantiate an AndrewBalchunasPSet4 object and call the methods as you write them. Be careful that some questions ask for you to print code and others ask you to return a value. If your code does not run, it will not receive credit.

Iterables

Stack API

int size()	Returns the number of elements in the stack
void push(T obj)	Pushes obj to the top of the stack
T pop()	Removes and returns the top element of the stack
T peek()	Shows the top element of the stack
Iterator<T> iterator()	Creates an iterator for the stack
String toString()	Shows the elements in the stack

Problem 1

Create a generic implementation of a Stack using a resizing array that supports iteration by implementing the Iterable interface.

Problem 2

Create a generic implementation of a Queue using a Linked List that supports iteration by implementing the Iterable interface.

Queue API

int size()	Returns the number of elements in the queue
void enqueue(T obj)	Adds obj to the queue
T dequeue()	Removes the next element of the queue
T peek()	Shows the next element of the queue to be removed
Iterator<T> iterator()	Creates an iterator for the queue
String toString()	Shows the elements in the queue

Comparable and Comparators

When creating objects it's important to consider whether or not we want them to be able to be compared to one another. There are two ways that we can handle comparing objects. We can either make the objects themselves `Comparable`, or we can use a `Comparator` object to compare two objects of a specific type. They both have advantages and disadvantages.

Problem 3

Create a class called `Date` intended to store objects representing a month, day, and year. The class has the following API. `Date` should implement the `Comparable` interface and dates should be compared by year, then month, and finally by day.

Date API

Date(int day, int month, int year)	Stores day month year
int compareTo(Date other)	Compares this date to other date
String toString()	String representation day/month/year

Problem 4

Create a class called `Pokemon` intended to store objects representing a pokemon. The class has the following API.

Pokemon API

Pokemon(String name, String type)	Stores the name and type of pokemon
static Comparator<Pokemon> pokemonNameComparator()	returns a PokemonNameComparator
static Comparator<Pokemon> pokemonTypeComparator()	returns a PokemonTypeComparator
String toString()	String representation <name/type>

The `Pokemon` class itself is not `Comparable`, but has two static subclasses: `PokemonNameComparator` and `PokemonTypeComparator`. `PokemonNameComparator` should compare two `Pokemon` by their name only and `PokemonTypeComparator` should compare two `Pokemon` by their type only. Note that these two classes must be `static` because there exists a static method that creates them.

Sorting

Here we investigate two of the most powerful sorts: *Mergesort* and *Quicksort*. On average they both work in $\mathcal{O}(n\log(n))$ time but have slight differences. Mergesort is $\mathcal{O}(n)$ in *space* whereas quicksort is $\mathcal{O}(\log(n))$ in *space*. Therefore a mergesort implementation may be more appropriate when using objects, and a quicksort may be more appropriate when using primitive types. Quicksort performs poorly when there are duplicate elements. Read about how to improve quicksort's performance as the Three-Way Quicksort. This is related to the so-called "Dutch National Flag" problem.

For the next four questions create a class called `Sorting` that will contain three static methods. One for quicksort and one for mergesort, and a wrapper that uses a `Comparator` for either sort. To use generics inside of a static method we define the generic inside of the header as follows: `public static <T> void methodName(DataType<T> dt)`

The API for the Sorting class is shown below.

Sorting

static <T extends Comparable<? super T> > void	mergesort(List<T> list)
static <T extends Comparable<? super T> > void	quicksort(List<T> list)
static <T> void	sort(List<T> list, Comparator<? super T> c, String sortType)

Problem 5

Implement a static method in `Sorting`, called `mergesort(List<T> arr)` that will sort the list values using the mergesort algorithm. Note that we want to make sure that we can actually sort things of type `T` so we need to define

`T extends Comparable<? super T>`. Do you realize why we need to use the lower-bounded wildcard here? Why would it be less flexible to write `T extends Comparable<T>`?

Problem 6

Implement a static method in `Sorting`, called, `quicksort(List<T> arr)`, where `T extends Comparable<? super T>`.

Problem 7

We might want to sort something that is not `Comparable` and we have no way to make it so (e.g. we cannot modify the imported data type). To get around that, we can use a `Comparator` to sort the array. Implement a static method in `Sorter` that has the following header. `sort(List<T> list, Comparator<? super T> c, String sortName)`. Here, sort the items in `list` using the comparator provided. The third parameter may be `"mergesort"` or `"quicksort"` and will use that particular sort. It might be useful to use private helper methods for each of the third parameter flags.

Problem 8

Investigate the stability of mergesort and quicksort by loading in all of the Pokemon data provided. Use a `List` to store the data. Sort the Pokemon data first by name and then by type to get a list where types are grouped together and Pokemon names are alphabetical. Do this using both mergesort and quicksort implementations from the previous problem. Take notice what happens.

Pattern Recognition using Sorting and Inheritence

Computer vision involves analyzing patterns to find objects in an image. The process is typically broken up into two pieces: *feature detection* and *pattern recognition*. Features can be things like corners, or blobs. Some well-known corner detection algorithms are Harris Corner Detection, Shi-Tomasi Corner Detection, Scale-Invarient Feature Transform (SIFT), Speeded-Up Robust Features (SURF). Blobs are detected using other algorithms.

In short, feature detection returns regions or points of interest. These points are then passed to a particular pattern recognition algorithm to make sense of the image. Here we will investigate a pattern recognition problem involving points and line segments.

The problem we are trying to solve: Given a set of n distinct (x, y) points, find every (maximal) line segment that connects a subset of 4 or more points.

You will do this by using the `StdDraw` API provided. See the file `StdDrawExample` to see how to use the API to draw lines and points on the screen. `StdDraw` can do more sophisticated things, but we will limit our use to drawing lines and points.

The first thing you'll need for this pattern recognition algorithm is a `Point` data type.

Problem 9

Create an immutable data type that represents a point in the x-y plane. The API is provided below. The `Point` must implement the `Comparable` interface. Note that the `Comparable` interface requires a generic for what is being compared. Here we'd pass `Point`. The header therefore would be `public class Point implements Comparable<Point>`

Point API

Point(int x, int y)	Initializes a point at (x, y)
void draw()	Draws the point at (x, y) using StdDraw
void drawTo(Point that)	Draw a line between this point and that point using StdDraw
int compareTo(Point that)	Compares two points by y-coordinate, break ties using x-coordinate
double slopeTo(Point that)	Computes the slope between this point and that point
Comparator<Point> slopeOrder()	Compares two points by slopes they make with this point
String toString()	String representation

Implementing this class should be straightforward.

- The *compareTo* method should compare points by their y-coordinates, breaking ties by their x-coordinates. In other words, a point is "greater than" another point if it is above or to the right of it.
- The *slopeTo* method returns the slope between the invoking point (x_0, y_0) and the argument point (x_1, y_1) given by the point-slope formula $(y_1 - y_0)/(x_1 - x_0)$. The slope of a horizontal line segment is zero, a vertical line segment is positive infinity `Double.POSITIVE_INFINITY`, and the slope of a point with itself is negative infinity `Double.NEGATIVE_INFINITY`
- The *slopeOrder* method returns a comparator that compares its two argument points by the slopes they make with the invoking point (x_0, y_0). In other words, two unrelated points $Point1$ and $Point2$ are compared using their slopes with *this*. This comparator will be used to order points of interest.

Ok, now that we have a class to represent a `Point`, we can simply make one that represents a `LineSegment`. A line segment object is a simple class with the following API.

LineSegment API

LineSegment(Point p1, Point p2)	Initializes the ends of the line segment
void draw()	Draws the line segment between p1 and p2 using StdDraw
String toString()	String representation

Now we are ready to detect features. The first approach we take will be a brute force one. We will examine 4 points at a time and check to see if they are collinear. **For simplicity, the input for this will NOT have 5 or more points on the same line! Try files like Input1, Input2, etc. to test your model.** If so, create a line segment between the starting and ending points. Store these using a `NaiveCollinearPoints` object. The API for such an object is as follows:

NaiveCollinearPoints API

NaiveCollinearPoints(Point[] points)	Finds all the line segments containing 4 points
int numberOfSegments()	The number of line segments found
LineSegment[] segments()	The segments of 4 collinear points found

Note that `segments()` should include each line segment containing 4 points only once. e.g. if the points w, x, y, z are collinear you should only store a line segment from w to z or from z to w but not both.

Use `NaiveCollinearPoints` to find the collinear points in the `naiveInput` file and draw the line segments. Did it work? *What is the complexity of this algorithm?*

A Better Way...

You (hopefully) deduced that the naive approach was not a great one. It solved the problem, but it did so in $\mathcal{O}(n^4)$ time! Can we do better?

Let's try to be clever about it.

Let's use a sorting-based approach to this. Pause here and think about why we created `Comparator` objects for our points. Why did we want to compare two random points with an invoking point?

Consider the following: Given a point p, we can determine whether or not p participates in a set of 4 OR MORE collinear points.

- This of p as the origin.
- For each other point q, find the slope it makes with p.
- Sort these points according to the slopes they make with p.
- Check if 3 or more adjacent points in the sorted order have slopes equal with respect to p. If so, these points, together with p are collinear.

Apply this method for each of the n points ends up providing an efficient algorithm to solve the problem.

What is the order of this algorithm? Well, let's think about it. We look at every point, so that's $\mathcal{O}(n)$ and then we sort all of the other points with respect to that one. If we choose a good sort, this would be $\mathcal{O}(n\log(n))$. Overall we've knocked down our $\mathcal{O}(n^4)$ algorithm down to $\mathcal{O}(n^2\log(n))$. Remember that $\log(n)$ is very small, think 30, for very large n so we've effectively converted our limited $\mathcal{O}(n^4)$ algorithm into a much less limited $\mathcal{O}(n^2\log(n))$ algorithm!!

For the last part of this problem set, create a program `BetterCollinearPoints` that implements this algorithm.

BetterCollinearPoints API

BetterCollinearPoints(Point[] points)	Finds all the line segments containing 4 or more points
int numberOfSegments()	The number of line segments found
LineSegment[] segments()	The segments of 4 collinear points found

The method `segments()` should include each *maximal* line segment containing 4 or more points exactly once. For example, if points v, w, x, y, z are all collinear then do not include the subsegments of v to z, e.g. v to x or y to z.

The input looks like the following. It is a single integer for the number of points to follow, then two integers corresponding to an (x, y) pair.

```
6
19000  10000
18000  10000
32000  10000
21000  10000
 1234   5678
14000  10000
```

```
8
10000      0
    0  10000
 3000   7000
 7000   3000
20000  21000
 3000   4000
14000  15000
 6000   7000
```

Example client file:

```
// read the n points from a file using Scanner
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
Point[] points = new Point[n];
for (int i = 0; i < n; i++) {
    int x = sc.nextInt();
    int y = sc.nextInt();
```

```
        points[i] = new Point(x, y);
    }

    // draw the points (32768 is max integer)
    StdDraw.enableDoubleBuffering();
    StdDraw.setXscale(0, 32768);
    StdDraw.setYscale(0, 32768);
    for (Point p : points) {
        p.draw();
    }
    StdDraw.show();

    // print and draw the line segments
    BetterCollinearPoints collinear = new BetterCollinearPoints(points);
    for (LineSegment segment : collinear.segments()) {
        System.out.println(segment);
        segment.draw();
    }
    StdDraw.show();
```

Be sure to try the test sets kw, rs, and mystery!

Examples: `input8.txt` has two line segments: (10000, 0) -> (0, 10000) (3000, 4000) -> (20000, 21000).

`input6.txt` has only a singular line segment: (14000, 10000) -> (32000, 10000)