EXERCISE 1: Comparables & Comparators (live-coding)

Download precept3.zip from the precepts page, unzip the project and open it using IntelliJ. Follow along with your preceptor (using the annotated code in the next page) to do the following:

(a) Define a natural (default) comparison behavior for the Point2D class and use it in a simple test client. Proceed according to the following steps:

- Modify the class declaration of Point2D to make it implement the Comparable interface.
- Implement the compareTo method. This method allows the point to be compared to another given point (passed as an argument to the method).
  - Use the y-coordinate for comparison and break ties using the x-coordinate.
  - Return 1 if the point is greater than the method argument, -1 if it is less and 0 otherwise.
- Complete the given test program to sort the array according to the natural order defined in the compareTo method.

(b) Define an alternate comparison behavior for 2D points and use it in a simple test client. Proceed according to the following steps:

- Uncomment the code marked as /* *** PART (B) *** */.
- Complete the implementation of class DistanceToOrder such that it allows comparing between two given points based on their distance to a given reference point.
  - Make the class implement the Comparator interface.
  - Implement the constructor to receive and store the reference point.
  - Implement the compare method. This method compares the two given argument points: Returns 1 if the first argument is farther from the reference point than the second argument, -1 if it is closer and 0 otherwise.
- Complete the given test program to sort the array according to the distance of the points from the origin (0, 0).
public class Point2D implements Comparable<Point2D> {
    private final double x, y;

    public Point2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Returns the square of the Euclidean distance between two points.
    public static double distanceSquared(Point2D p, Point2D q) { … }

    // Compares by y-coordinate, breaking ties by x-coordinate.
    public int compareTo(Point2D other) {
        if (this.y < other.y) return -1;
        if (this.y > other.y) return +1;
        return Double.compare(this.x, other.x);
    }

    // Returns a Comparator for comparing by distance to a reference point.
    public static Comparator<Point2D> distanceToOrder(Point2D ref) {
        return new DistanceToOrder(ref); A
    }

    // compare points according to their distance to a reference point.
    private static class DistanceToOrder implements Comparator<Point2D> {
        private Point2D ref;

        private DistanceToOrder(Point2D ref) {
            this.ref = ref;
        }

        public int compare(Point2D p, Point2D q) {
            double dist1 = distanceSquared(p, ref);
            double dist2 = distanceSquared(q, ref);
            return Double.compare(dist1, dist2); B
        }
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        Point2D[] array = new Point2D[n];
        // … fill the array with random points.

        Arrays.sort(array); C
        // … print the array

        Comparator<Point2D> cmp = Point2D.distanceToOrder(new Point2D(0, 0));
        Arrays.sort(array, cmp); D
        // … print the array
    }
}
EXERCISE 2: Three-Way Merge Sort

3-way Merge sort is a variant of the Merge sort algorithm that considers 3 “equal” subarrays instead of 2 subarrays.

(a) Given 3 sorted subarrays of size $\frac{n}{3}$, how many comparisons are needed (in the worst case) to merge them to a sorted array of size $n$? Provide your answer in tilde notation.

(b) What is the order of growth of the number of compares in 3-way Merge Sort as a function of the array size $n$?

(c) Given a choice, would you choose 3-way or 2-way merge sort? Justify your answer.
EXERCISE 3: Algorithm Design

Let $a = a_0, a_1, ..., a_{n-1}$ be an array of length $n$. An array $b$ is a circular shift of $a$ if it consists of the subarray $a_k, a_{k+1}, ..., a_{n-1}$ followed by the subarray $a_0, a_1, ..., a_{k-1}$ for some integer $k$. In the example below, $b$ is a circular shift of $a$ (with $k = 7$ and $n = 10$).

<table>
<thead>
<tr>
<th>sorted array $a[]$</th>
<th>circular shift $b[]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 5 6 8 9 34 55 89</td>
<td>34 55 89 1 2 3 5 6 8 9</td>
</tr>
</tbody>
</table>

Suppose that you are given an array $b$ that is a circular shift of some sorted array (but you have access to neither $k$ nor the sorted array). Assume that the array $b$ consists of $n$ comparable keys, no two of which are equal. Design an efficient algorithm to determine whether a given key appears in the array $b$. Your algorithm should run in $O(\log n)$. 
ASSIGNMENT TIPS: Autocomplete

(1) Given an array of elements with duplicates, can we use the book implementation of Binary Search to find the first occurrence of an element?

- The standard implementation of Binary Search finds any occurrence, which is not necessarily the first occurrence.
- Finding an occurrence and then scanning left to find the first occurrence yields a linear running time (in the worst case), which is not good!
- In this assignment, you will have to modify Binary Search to find the first (and last) occurrence of an element in a sorted array in logarithmic time (in the worst case).
- For full credit, your algorithm has to make at most $1 + \lceil \log_2 n \rceil$ compares. However, if your algorithm in $O(\log n)$ but makes more than $1 + \lceil \log_2 n \rceil$ compares, you will lose only 1 point.

(2) What is the order of growth of the substring method?

- Creating a substring of length $r$ takes time proportional to $r$.
- Note that the string comparison functions in the assignment should take time proportional to the number of characters needed to resolve the comparison.

**Example:** The comparison between $X=\text{"AAAAAAA"}$ and $Y=\text{"AABBB"}$ can be resolved when the first "B" in $Y$ is reached. The comparison function should not take time proportional to the size of $X$ or the size of $Y$. It should take time proportional to the number of characters needed to resolve the comparison!

- Most uses of the substring method in the compare functions do not meet the above time constraint. So, be careful!

(3) Are there other things I should note about the assignment?

Definitely! This is why we have a Checklist for the assignment. You might want to check the answers for the following questions in the Checklist:

- What is meant by an immutable data type?
- What is the meaning of the type parameter Key in the following function declaration?
  ```java
  public static <Key> int firstIndexOf(Key[] a, Key key, Comparator<Key> comparator)
  ```
- What's a good way to get a Comparator object to use for testing?

(4) A video that provides some tips for the assignment is available on the assignment Checklist page. The video was made in 2014, so a few things are outdated, but most of it is still useful (for example, the API has changed).