Lab: Sorting

Background: Getting Started

Download Sorter.java and SortDisplay.java, and MyLocation.java. `SortDisplay` is a complex class used to visualize a sort. You will never need to look inside it, and it would definitely be a bad idea to attempt to modify it. The `Sorter` class is largely empty. In this lab, you will design, document and then add new methods and modify existing ones in the `Sorter` class. Coding without completing documentation means you get a score of 0 (zero) on the style portion of your grade. Each algorithm is to be designed and then documented before it is coded.

When you create a new `Sorter`, a `SortDisplay` window will appear. You can ignore this window for the first couple exercises.

Exercise: The Bare Minimum

Add a method to `Sorter` called `indexOfMin`, which takes in an array of `Comparable` objects and an integer `startIndex`, and returns the index of the lowest value in the array, considering only the range from `startIndex` to the end of the array.

```
public int indexOfMin(Comparable[] a, int startIndex)
```

Exercise: MyLocation

A `MyLocation` stores a row and a column number, and is written as (*row*, *col*). The following `MyLocation` values are listed in increasing order.

(1, 1) (1, 2) (1, 3) (2, 1) (2, 2) (2, 3) (3, 1) (3, 2) (3, 3)

Modify the `MyLocation` class, so that your `indexOfMin` method can take in an array of `MyLocation` objects and correctly identify the index of the minimum `MyLocation`. (You should *not* need to modify `indexOfMin`!) Be sure to test that your `indexOfMin` method works on such an array.

Exercise: Selection Sort

There are many algorithms for putting the elements of an array into sorted order. The easiest is probably the selection sort algorithm. To perform a selection sort, find the lowest value in an array, and move it into the first position. Then find the next lowest value in the array, and move it into the next position, and so on. The sequences below show some the contents of an array during a run of selection sort.

8 3 1 9 7 5
1 3 8 9 7 5
1 3 8 9 7 5
1 3 5 9 7 8
1 3 5 7 9 8
1 3 5 7 8 9

Go ahead and complete `Sorter`'s `selectionSort` method, so that it uses selection sort to sort an array of `Comparable` objects. Be sure to use your `indexOfMin` method.

We'd like to be able to use the `SortDisplay` class to visualize our sort. To do so, include the following line of code after each line that modifies the contents of the array.

```
display.update();
```

The `SortDisplay` class's `update` method tells the display to redraw the array and pause. When you're done, test your work by clicking the "Selection Sort" button on the Sort Display window. If you like, you may use "Step" mode, in which you can click on the array image to advance your algorithm by one step.

Exercise: Insert Clever Title Here

Add a method called `insert` to the `Sorter` class. This method should take in an array of `Comparable` objects, and an integer `nextIndex`. When this method is called, all of the elements in the array before `nextIndex` will already appear in increasing order. The remaining elements will appear in random order. It will be `insert`'s job to take the element at `a[nextIndex]`, and insert it into its proper place in the first part of the array, shifting elements down to accomodate it. When `insert` returns, all of the values in the array from the beginning to `nextIndex` will appear in increasing order.

```
public void insert(Comparable[] a, int nextIndex)
```

The following example shows the contents of an array before and after a call to `insert`, were `nextIndex` is 3.

Before: ***1 4 6*** 2 8 7

After: ***1 2 4 6*** 8 7

Exercise: Insertion Sort

The insertion sort algorithm inserts each element of the array into a growing sequence of sorted values. When the algorithm reaches the end of the array, it will appear in sorted order. The sequences below show some the contents of an array during a run of insertion sort.

12	5	8	9	10	7
5	***12***	8	9	10	7
5	***8***	***12***	9	10	7
5	***8***	***9***	***12***	10	7
5	***8***	***9***	***10***	***12***	7
5	***7***	***8***	***9***	***10***	***12***

Go ahead and complete the `insertionSort` method, being sure to call `display.update()` whenever the contents of the array change. Use the "Insertion Sort" button to test your work.

Exercise: Mergesort

Mergesort is a very clever sorting algorithm, consisting of the following steps, and illustrated by the example on the right.

1. Divide the array in half.	5	9	2	6	8	3
2. Sort each half.	2	5	9	3	6	8
3. Merge the sorted halves together.	2	3	5	6	8	9

Notice that this is a recursive definition, since we still need to sort each half. We'll sort the halves recursively using mergesort itself!

Go ahead and complete the `mergesort` and `mergesortHelp` methods. You'll find it helpful to use the `merge` method that has already been written for you. `merge` requires that the values from `lowIndex` to `midIndex` and from `midIndex + 1` to `highIndex` already appear in increasing order. `merge` then merges these two halves, so that the values from `lowIndex` to `highIndex` appear in increasing order.

```
private void merge(Comparable[] a, int lowIndex,
                   int midIndex, int highIndex)
```

Exercise: Quicksort

Quicksort is one of the most popular sorts. Like mergesort, it involves dividing up the array and recursively sorting each part. Quicksort consists of the following steps, and is illustrated by the example on the right.

1. Choose a pivot.	***3***	5	2	4	7	8
2. Partition the array around the pivot.	2	***3***	5	4	7	8
3. Sort each partition.	2	***3***	4	5	7	8

Typically, we will always choose the first element to be the pivot. When we partition the array, we move the pivot into its final location in the sorted array. As we do this, we move all lower values to the left of the pivot, and leave all higher values to the right. This gives us a left partition and a right partition, each of which must now be sorted recursively using quicksort.

Complete the `quicksort` and `quicksortHelp` methods, making use of the `partition` method provided. This method returns the index of the pivot, and arranges the array so that all values to the left of the pivot (starting from `lowIndex`) are less than or equal to the pivot, and all values to the right of the pivot (through `highIndex`) are greater than or equal to the pivot.

```
private int partition(Comparable[] a, int lowIndex,
                                      int highIndex)
```