**Concurrency and Animation**

For this project you will write a Java application using graphics and concurrency (multithreading) to display an animated version of various common sorting algorithms. The basic version of this application is fairly simple and straightforward.

**GUI Screenshots**



*The sorting animation application shown at startup.*

 *After the arrays have been populated and the sorts chosen, but before sorting has begun.*

*  
The sorts in progress.*

*  
After both sorts have completed.*

**The Classes**

Create a class to represent the sorting animation application and inherits from JFrame. It should have a pair of SortPanel objects as data members. The class should have a custom constructor that passes a title bar string to the superclass constructor and adds the two SortPanel objects to the application’s layout. It should also have a main() method that creates an instance of the application, sets the default close operation and size, and makes the application visible.

The SortPanel class encapsulates the display and controls for the sorting animation and inherits from JPanel. It should have buttons to populate the array and start the sort, as well as a control (such as a JComboBox or JSpinner) that allows selection of the sorting algorithm to use in the sort. It also has a SortAnimationPanel as one of its data members. The constructor for the class should manage the layout for the controls and animation panel. This class can also handle events from the buttons.

The SortAnimationPanel class will display the visual results of sorting and should inherit from JPanel and implement the Runnable interface. (Hint: You may want to make this an “inner” or “nested” class of SortPanel. Doing so will give this class access to the SortPanel data members without any need for an object reference.)

Define an object reference to an array of integers (either in the SortPanel class or in the SortAnimationPanel class). When the “Populate Array” button is pressed, create a new array of integers; the size of this array should be the width of the animation panel. Fill the array with random values in the range 0 to the height of the animation panel (Hint: Review the Java Random class and its method nextInt() for a good way to do this). Once the array is populated, call repaint() to display it, disable the “Populate Array” button, and enable the “Sort” button.

You should override the paintComponent() method for the SortAnimationPanel class. After you call the superclass version of the method, get the dimensions of the panel and clear it. Then, if the array of integers is not null, draw a series of lines on the panel surface representing the array elements. The height of each line should represent the integer value of the corresponding integer array element.

When the “Sort” button is pressed, disable the “Sort” button and create a new Thread object from your Runnable SortAnimationPanel. Call the Thread object’s start() method to start the sort.

The run() method should call the appropriate sort method based on the user’s selection to sort the array of integers in ascending order. As the sorting algorithm progresses, call repaint() any time two elements are swapped. After each pass through the sort algorithm’s outer loop (or equivalent), have the thread sleep for a short time (say 100 milliseconds). Effectively, the thread will pause for a short time every time an element is put into sorted order.

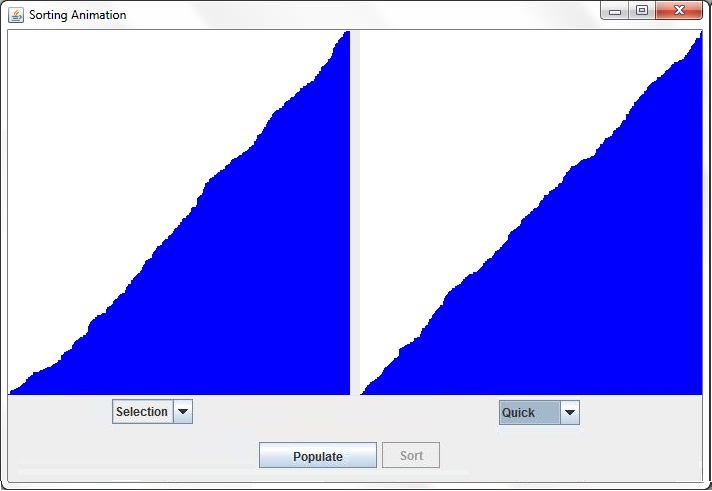
You should provide ***at least*** three different sorting algorithms that the user can choose from for each panel. Possible candidates include selection sort, insertion sort, bubble sort, heap sort, quicksort, shell sort, merge sort, etc. Pick at least one of the more efficient (i.e., better than O(N2) – Big O notation described on page 4) algorithms. C++ and/or Java code for these sorts can all be found online.

When the array is completely sorted, enable the “Populate Array” button so that the user can start the process over.

**YOU MUST DO THE BELOW QUESTIONS TOO !!!!!**

Please note that ALL extra credit points on this project will be added to your programming points, NOT exam/quiz points!

1. (5 points) Allow the user to specify other initial orderings of the array besides random order (for example, ascending or descending order).
2. (5 points) Allow the user to change the direction of the sort (i.e., allow sorting in descending order as well as ascending order). You can do this be passing an appropriate Comparator object to the sort code and using it to compare elements of the array.
3. (5 points) Program the GUI so that **both** (or all if you add more than just the required two) panels are populated with exactly the same set of vertical lines with a single Populate button and that they all begin at exactly the same time with a single Sort button. See the illustration of the finished sort below:



1. (5 points) Allow the user to change the sort speed by changing the sleep period. For example, maybe 100 milliseconds sleep time is “Fast,” 1000 is “Medium” and 5000 is “Slow.”
2. (5 points) Allow the user to pause the sort in the panel and then resume it or, if the single Populate and single Sort button is implemented as seen in the figure above, that all sorts are paused and all can be resumed.
3. (5 points) Allow the user to stop the sort in one panel before it has completed.

**Helpful Links**

http://stackoverflow.comhttp://stackoverflow.com/questions/15422100/animating-graphical-sorthttp://www.java2novice.com/java-sorting-algorithms/quick-sorthttp://www.java2novice.com/java-sorting-algorithms/insertion-sorthttp://docs.oracle.com/javase/7/docs/api/

http://docs.oracle.com/javase/8/docs/api/

**Example of Big O Notation**

O(N2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N3), O(N4) etc.

bool ContainsDuplicates(IList<string> elements) {

for (var outer = 0; outer < elements.Count; outer++) {

for (var inner = 0; inner < elements.Count; inner++) {

// Don't compare with self

if (outer == inner) continue; 🡨 Mr. Decker doesn't like this but left it alone!

if (elements[outer] == elements[inner]) return true;

}

}

return false;

} // Source: https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/