This last assignment is about sorting algorithms, where you have little code to write. But you have to implement your sorting algorithms in a way such that the step-by-step progress can be demonstrated. This is enforced by the implementation of the **Demonstrable** interface, which is a very simple interface with only one method declaration:

```
public interface Demonstrable {

   String show();

}
```

The actual sorting algorithms are implemented in the class **SortingMethods**, also given to you:

```
import java.util.List;



public class SortingMethods<E extends Comparable<E>> implements Demonstrable {



   private static final int UPPER_LIMIT_FOR_SHOW = 15;



   StringBuilder steps = new StringBuilder();



   /** DO NOT CHANGE THIS METHOD */

   public String show() {

       steps.setLength(steps.length() - 1);

       return steps.toString();

   }



   /**

    * This is a bottom-up implementation of the mergesort algorithm where multiple
small sublists of the same size

    * are merged in a single pass. The demonstration of this algorithm is done by
showing each stage of the merge on
```

```
    * sublists of the same size (except possibly the very last sublist, if the input
list has odd number of elements).

    * For example, if the input is [5, 2, 4, 6, 7, 1, 3], a call to show() returns the
string:

    * <p>

    * [5, 2, 4, 6, 7, 1, 3]

    * [2, 5, 4, 6, 1, 7, 3]

    * [2, 4, 5, 6, 1, 3, 7]

    * [1, 2, 3, 4, 5, 6, 7]

    * </p>

    * The first line shows the input, the second line demonstrates the pairwise merges
creating sorted sublists of

    * size 2 (except the very last sublist, which is [3]), the third line demonstrates
merges of size-2 lists together

    * to create [2, 4, 5, 6] and [1, 3, 7] (again, the very last sublist was not a pair
because the input has an odd

    * number of elements), and the fourth line merges the size-4 sublists together,
which is the sorted list.

    *

    * @param elements the input list to be sorted

    */

   public void mergeSort(List<E> elements) {

       // TODO: implement this as part of the assignment

   }



   /**

    * This is an implementation of the insertion sort algorithm. The steps are
demonstrated by adding a line to the
```

```
    * string representation whenever the unsorted portion of the input list decreases
in size by 1. For example, if

    * the input is [8, 3, 15, 0, 9], then show() returns the string:

    * <p>

    * [8, 3, 15, 0, 9]

    * [3, 8, 15, 0, 9]

    * [3, 8, 15, 0, 9]

    * [0, 3, 8, 15, 9]

    * [0, 3, 8, 9, 15]

    * </p>

    *

    * @param elements the input list to be sorted

    */

   public void insertionSort(List<E> elements) {

       // TODO: implement this as part of the assignment

   }



   public void selectionSort(List<E> elements) {

        steps.setLength(0);

        if (elements.size() < UPPER_LIMIT_FOR_SHOW) {

            steps.append(elements.toString()).append('\n');

        }

        int boundary = 0;

        while (boundary < elements.size() - 1) {

            int minIndex = findMinIndex(elements, boundary);
            swap(elements, boundary++, minIndex);
            if (elements.size() < UPPER_LIMIT_FOR_SHOW)
                steps.append(elements.toString()).append('\n');
        }
    }

    private int findMinIndex(List<E> elements, int boundary) {
        int minIndex = boundary;
        if (boundary == elements.size() - 1)
            return minIndex;
        E min = elements.get(minIndex);
        for (int i = boundary + 1; i < elements.size(); i++) {
            E e = elements.get(i);
            if (e.compareTo(min) < 0) {min = e; minIndex = i;}
        }
        return minIndex;
    }

    private void swap(List<E> elements, int i, int j) {
        if (i < 0 || j < 0 || i >= elements.size() || j >= elements.size()) {
            String err = String.format("Cannot swap elements between positions %d and %d
in list of %d elements.",
                                        i, j, elements.size());
            throw new IndexOutOfBoundsException(err);
        }
        E tmp = elements.get(i);

        elements.set(i, elements.get(j));

        elements.set(j, tmp);

    }

}
```

For this assignment, you can assume that show() will be called only after one of the sorting methods have been called. However, it is the responsibility of the sorting method to ensure that no remnants of a previous sorting algorithms steps remain in the output of show(). For example, if insertionSort is called on a list and then selectionSort is called, a call to show() after that should *only* return the string relevant to the demonstration of the selectionSort method's steps.

Also keep in mind that any helper method must be private.

Your task is to implement the insertionSort and mergeSort methods, and ensure that the show() method operates as describes in their documentations. The complete implementation of selectionSort is included as a way to show how you can use the StringBuilder object to meet this requirement.

Finally, you are also given a class called SortingClient, which serves as an example "client code" which uses your implementation. It includes a quick sample test and a method for runtime analysis of sorting algorithms. In this assignment, full credit may not be awarded for an implementation if the implementation is inexplicably slow. For example, selection sort and insertion sort should be somewhat comparable, while mergesort should be significantly faster. If merge sort appears to be as slow as a quadratic-time sorting algorithm, it may only receive partial credit even if the sorted output is correct.

As an internal check, you can copy-paste the output of the runtime analysis method into a spreadsheet and plot the time taken as a graph/chart, to see if the curve appears quadratic, or log-linear.

Here is the SortingClient class:

```
import java.util.ArrayList;

import java.util.List;

import java.util.Random;



public class SortingClient {



    private static final int MILLION          = (int) 1.0e6;

    private static final int HUNDRED_THOUSAND = (int) 1.0e5;
```

```
   private static final int TEN_THOUSAND     = 10000;



   private static void runtimeAnalysis() {

       SortingMethods<Integer> integerSortingMethods = new SortingMethods<>();



       Random generator = new Random(10283645586984L); // you can change this seed if
you want, but it's not necessary

       System.out.printf("%s    %s    %s%n", "No. of elements", "Selection Sort Time
(in ms)",

                         "MergeSort Time (in ms)");

       for (int listSize = TEN_THOUSAND; listSize < HUNDRED_THOUSAND; listSize +=
TEN_THOUSAND) {

           List<Integer> integers = new ArrayList<>(generator.ints(listSize, 0,
MILLION).boxed().toList());



           long begin1 = System.nanoTime();

           integerSortingMethods.selectionSort(integers);

           long   end1       = System.nanoTime();

           double timeTaken1 = (double) (end1 - begin1) / HUNDRED_THOUSAND;



           long begin2 = System.nanoTime();

           integerSortingMethods.mergeSort(integers);

           long   end2       = System.nanoTime();

           double timeTaken2 = (double) (end2 - begin2) / HUNDRED_THOUSAND;



           System.out.printf("%-15d    %-27.2f    %.2f%n", listSize, timeTaken1,
timeTaken2);
```

```
        }

    }



    private static void aSampleTest() {

        Random                  generator              = new Random(1028984L);

        List<Integer>           integers               = new ArrayList<>
(generator.ints(6, 0, 20).boxed().toList());

        SortingMethods<Integer> integerSortingMethods = new SortingMethods<>();

        integerSortingMethods.selectionSort(integers);

        System.out.println(integerSortingMethods.show());

    }



    public static void main(String[] args) {

        aSampleTest();

        runtimeAnalysis();

    }

}
```

You do not need to understand everything in the above class. It is simply meant as something you can use to test the correctness and efficiency of your sorting algorithm implementations.

Rubric

- Mergesort's correctness, i.e., it provides the properly sorted output for given input lists (20 points)
- Mergesort's correctness of stepwise demonstration through the string representation of show() (20 points)
- Insertion sort's correctness, i.e., it provides the properly sorted output for given input lists (20 points)
- Insertion sort's correctness of stepwise demonstration through the string representation of show() (20 points)
- Insertion sort's efficiency is comparable to that of selection sort, and usually slightly faster (10 points)
- Mergesort's efficiency is significantly faster than both selection sort and insertion sort, and ideally, comparable to that of Java's Collections.sort(List list) (10 points)