Program Structures and Algorithms

Spring 2023 (SEC – 8)

NAME: Rishi Desai

NUID: 002751030

Assignment 3

Task:

1. You are to implement three (3) methods (repeat, getClock, and toMillisecs) of a class called Timer. Please see the skeleton class that I created in the repository. *Timer* is invoked from a class called Benchmark_Timer which implements the Benchmark interface.
2. Implement InsertionSort (in the InsertionSort class) by simply looking up the insertion code used by Arrays.sort. If you have the instrument = true setting in test/resources/config.ini, then you will need to use the helper methods for comparing and swapping (so that they properly count the number of swaps/compares). The easiest is to use the helper.swapStableConditional method, continuing if it returns true, otherwise breaking the loop. Alternatively, if you are not using instrumenting, then you can write (or copy) your own compare/swap code. Either way, you must run the unit tests in InsertionSortTest.
3. Implement a main program (or you could do it via your own unit tests) to actually run the following benchmarks: measure the running times of this sort, using four different initial array ordering situations: random, ordered, partially-ordered and reverse-ordered. I suggest that your arrays to be sorted are of type *Integer*. Use the doubling method for choosing *n* and test for at least five values of *n.* Draw any conclusions from your observations regarding the order of growth.

Link to report:
https://github.com/RishiDesai17/INFO6205/blob/Spring2023/assignments/assignment-3/Assignment%203.pdf

Link to code:

Insertion sort (Part 2):
https://github.com/RishiDesai17/INFO6205/tree/Spring2023/src/main/java/edu/neu/coe/info6205/sort/elementary

Timer and benchmark timer (Part 1 and 3):

https://github.com/RishiDesai17/INFO6205/tree/Spring2023/src/main/java/edu/neu/coe/info6205/util

Relationship Conclusion:

As observed by the logarithm ratios shown in the table given in the next section, we can understand the values towards which the logarithm ratio converges towards in the insertion sort algorithm when applied to arrays of different arrangement.

For randomly ordered array: The order of growth of the time taken to run for an array of size N is $N^{1.9621}$

For ordered array: The order of growth of the time taken to run for an array of size N is $N^{0.8013}$

For reverse ordered array: The order of growth of the time taken to run for an array of size N is $N^{2.01483}$

For partially ordered array: The order of growth of the time taken to run for an array of size N is $N^{1.9621}$

Evidence to support that conclusion:

Randomly Ordered Array:

Starting with N=500, we double it until we get five different observations for each N. The array contains numbers generated randomly using the Math.random function.

Output:

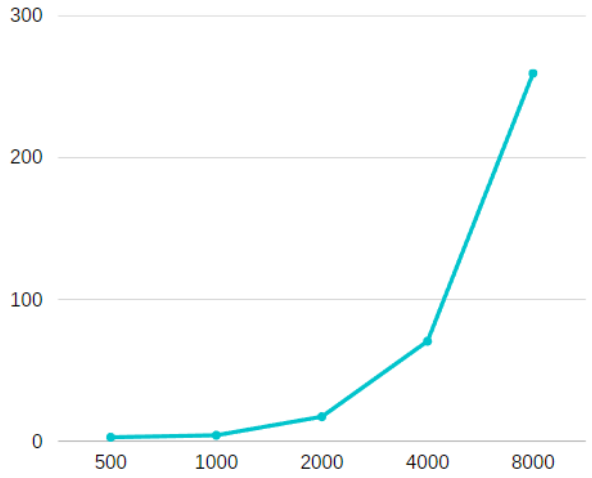
```
Randomly Ordered:
N = 500
2023-02-04 20:52:06 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 2.9742550000000003

N = 1000
2023-02-04 20:52:06 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 4.385065

N = 2000
2023-02-04 20:52:06 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 17.487145

N = 4000
2023-02-04 20:52:07 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 70.63506

N = 8000
2023-02-04 20:52:08 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 259.37399
```

N	T	log(N)	log(T)	Slope
500	2.974	8.965784	1.572405	
1000	4.385	9.965784	2.132577	
2000	17.487	10.96578	4.128211	1.995634
4000	70.635	11.96578	6.142311	2.0141
8000	259.374	12.96578	8.01889	1.876579

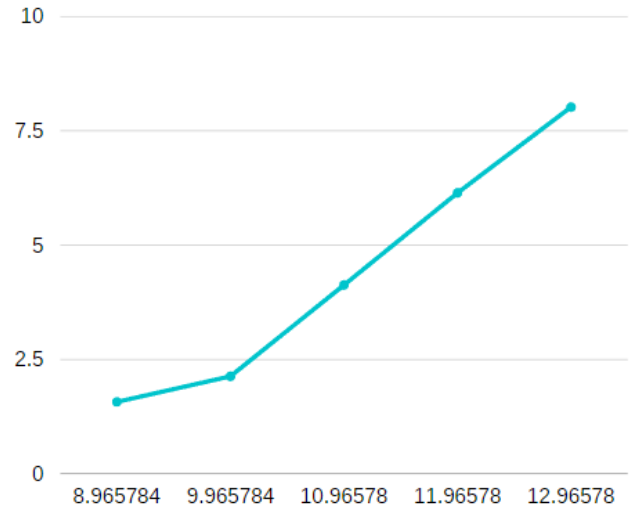
Average slope: 1.9621

Thus, we can conclude that the order of growth is 1.9621

N vs T



log(N) vs log(T)



Ordered Array:

Starting with N=500, we double it until we get five different observations for each N. The array contains numbers generated randomly using the Math.random function. Then sort the array in ascending order.

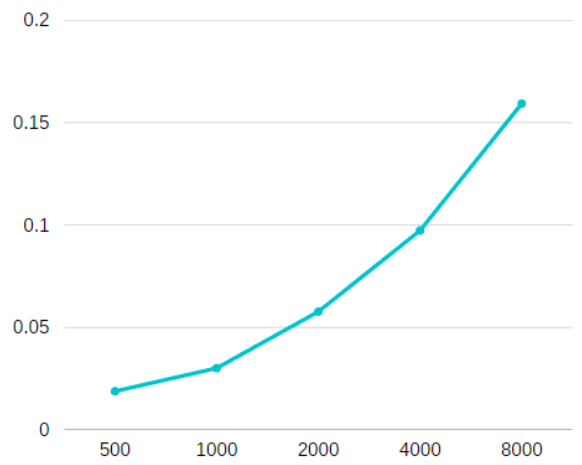
Output:

```
Ordered:
N = 500
2023-02-04 20:52:14 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 0.018775

N = 1000
2023-02-04 20:52:14 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 0.030170000000000002

N = 2000
2023-02-04 20:52:14 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 0.057675

N = 4000
2023-02-04 20:52:14 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 0.09726

N = 8000
2023-02-04 20:52:14 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 0.159315
```

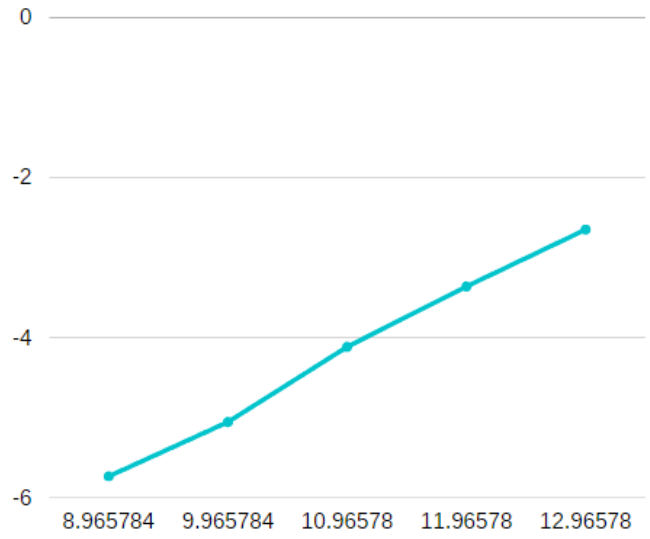
N	T	log(N)	log(T)	Slope
500	0.0188	8.965784	-5.73312	
1000	0.0301	9.965784	-5.05409	
2000	0.0577	10.96578	-4.11528	0.938808
4000	0.0973	11.96578	-3.36142	0.753868
8000	0.1593	12.96578	-2.65018	0.711235

Average slope: 0.8013

N vs T



log(N) vs log(T)



Reverse ordered array:

Starting with N=500, we double it until we get five different observations for each N. The array contains numbers generated randomly using the Math.random function. Then sort the array in descending order.

Output:

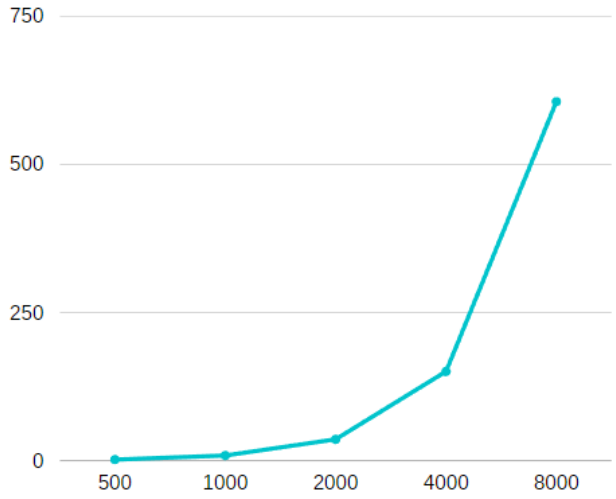
```
Reverse Ordered:
N = 500
2023-02-04 20:52:15 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 2.27454

N = 1000
2023-02-04 20:52:15 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 9.179325

N = 2000
2023-02-04 20:52:15 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 36.46501

N = 4000
2023-02-04 20:52:16 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 150.90441

N = 8000
2023-02-04 20:52:19 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 605.874245
```

N	T	log(N)	log(T)	Slope
500	2.2745	8.965784	1.185549	
1000	9.1793	9.965784	3.198384	
2000	36.465	10.96578	5.18844	1.990056
4000	150.9044	11.96578	7.237491	2.049051
8000	605.8742	12.96578	9.242874	2.005383

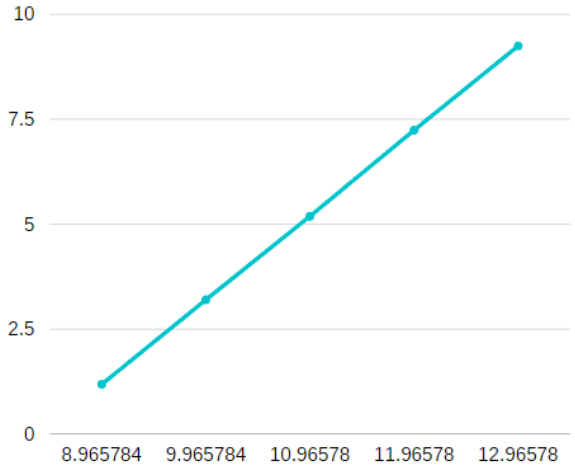
Average slope: 2.01483

Hence, we can conclude that the order of growth is 2.01483

N (array size) vs T (time taken)



log(N) vs log(T)



Partially ordered array:

Starting with N=500, we double it until we get five different observations for each N. The array contains numbers generated randomly using the Math.random function. Then sort the array in ascending order. After that regenerate the second half of the array with random numbers. This gives a partially ordered array.

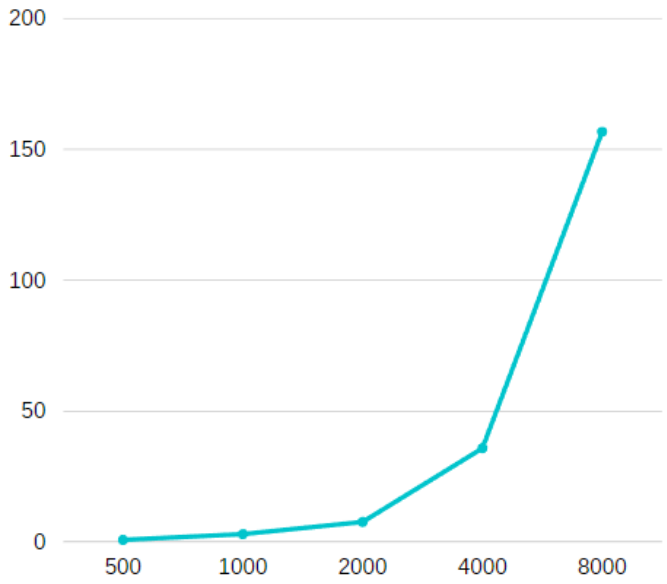
```
Partially Ordered:
N = 500
2023-02-04 20:52:33 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 0.868135

N = 1000
2023-02-04 20:52:33 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 3.07246

N = 2000
2023-02-04 20:52:33 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 7.706525000000001

N = 4000
2023-02-04 20:52:33 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 35.84627

N = 8000
2023-02-04 20:52:34 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20 runs
Time taken = 156.761425
```

N	T	log(N)	log(T)	Slope
500	0.8681	8.965784	-0.20407	
1000	3.0725	9.965784	1.619413	
2000	7.7065	10.96578	2.946076	1.326663
4000	35.8462	11.96578	5.163748	2.217672
8000	156.7614	12.96578	7.292427	2.128678

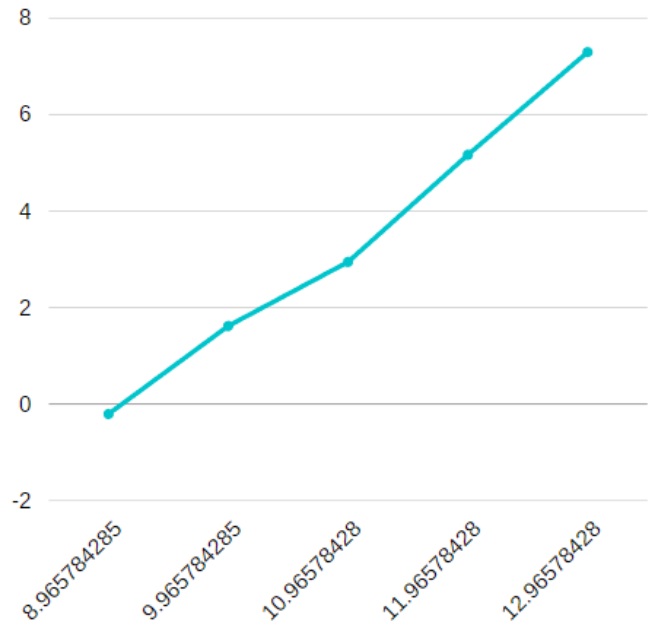
Average slope: 1.8910

Hence, we can conclude that the order of growth is 1.8910

N (array size) vs T (time taken)



log(N) vs log(T)



Code:

InsertionSort.java

```
package edu.neu.coe.info6205.sort.elementary;
```

```
import edu.neu.coe.info6205.sort.BaseHelper;
import edu.neu.coe.info6205.sort.Helper;
import edu.neu.coe.info6205.sort.SortWithHelper;
import edu.neu.coe.info6205.util.Config;

/**
 * Class InsertionSort.
 *
 * @param <X> the underlying comparable type.
 */
public class InsertionSort<X extends Comparable<X>> extends SortWithHelper<X> {

    /**
     * Constructor for any sub-classes to use.
     *
     * @param description the description.
     * @param N           the number of elements expected.
     * @param config      the configuration.
     */
    protected InsertionSort(String description, int N, Config config) {
        super(description, N, config);
    }

    /**
     * Constructor for InsertionSort
     *
     * @param N      the number elements we expect to sort.
```

```
     * @param config the configuration.
     */
    public InsertionSort(int N, Config config) {
        this(DESCRIPTION, N, config);
    }

    public InsertionSort(Config config) {
        this(new BaseHelper<>(DESCRIPTION, config));
    }

    /**
     * Constructor for InsertionSort
     *
     * @param helper an explicit instance of Helper to be used.
     */
    public InsertionSort(Helper<X> helper) {
        super(helper);
    }

    public InsertionSort() {
        this(BaseHelper.getHelper(InsertionSort.class));
    }

    /**
     * Sort the sub-array xs:from:to using insertion sort.
     *
     * @param xs   sort the array xs from "from" to "to".
```

```
     * @param from the index of the first element to sort
     * @param to   the index of the first element not to sort
     */
    public void sort(X[] xs, int from, int to) {
        final Helper<X> helper = getHelper();

        for (int i = from; i < to - 1; i++) {
            int j = i;
            while(j + 1 > from && helper.less(xs[j + 1], xs[j])) {
             helper.swap(xs, j, j + 1);
             j -= 1;
            }
        }
    }

    public static final String DESCRIPTION = "Insertion sort";

    public static <T extends Comparable<T>> void sort(T[] ts) {
        new InsertionSort<T>().mutatingSort(ts);
    }
}
```

Timer.java

```
package edu.neu.coe.info6205.util;

import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.function.UnaryOperator;
```

```
/**
 * Class which is able to time the running of functions.
 */
public class Timer {

   /**
    * Run the given function n times, once per "lap" and then return the result of calling
meanLapTime().
    * The clock will be running when the method is invoked and when it is quit.
    *
    * This is the simplest form of repeat.
    *
    * @param n       the number of repetitions.
    * @param function a function which yields a T.
    * @param <T> the type supplied by function (amy be Void).
    * @return the average milliseconds per repetition.
    */
   public <T> double repeat(int n, Supplier<T> function) {
      for (int i = 0; i < n; i++) {
         function.get();
         lap();
      }
      pause();
      final double result = meanLapTime();
      resume();
      return result;
   }

   /**
    * Run the given functions n times, once per "lap" and then return the mean lap time.
    *
    * @param n       the number of repetitions.
    * @param supplier a function which supplies a different T value for each repetition.
    * @param function a function T=>U and which is to be timed.
    * @param <T> the type which is supplied by supplier and passed in to function.
    * @param <U> the type which is the result of <code>function</code> (may be Void).
    * @return the average milliseconds per repetition.
    */
   public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function) {
      return repeat(n, supplier, function, null, null);
   }

   /**
```

```
    * Pause (without counting a lap); run the given functions n times while being timed, i.e. once per "lap", and finally return the result of calling meanLapTime().
    *
    * @param n          the number of repetitions.
    * @param supplier     a function which supplies a T value.
    * @param function     a function T=>U and which is to be timed.
    * @param preFunction  a function which pre-processes a T value and which precedes the call of function, but which is not timed (may be null). The result of the preFunction, if any, is also a T.
    * @param postFunction a function which consumes a U and which succeeds the call of function, but which is not timed (may be null).
    * @param <T> the type which is supplied by supplier, processed by prefunction (if any), and passed in to function.
    * @param <U> the type which is the result of function and the input to postFunction (if any).
    * @return the average milliseconds per repetition.
    */
   public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T> preFunction, Consumer<U> postFunction) {
       logger.trace("repeat: with " + n + " runs");

       pause();
       for (int i = 0; i < n; i++) {
           T t = supplier.get();

           if (preFunction != null) {
                   t = preFunction.apply(t);
           }

           resume();

           U u = function.apply(t);

           pauseAndLap();

           if (postFunction != null) {
                   postFunction.accept(u);
           }

       }

       double averageTimeTaken = meanLapTime();
       return averageTimeTaken;
   }
```

```
/**
 * Stop this Timer and return the mean lap time in milliseconds.
 *
 * @return the average milliseconds used by each lap.
 * @throws TimerException if this Timer is not running.
 */
public double stop() {
   pauseAndLap();
   return meanLapTime();
}

/**
 * Return the mean lap time in milliseconds for this paused timer.
 *
 * @return the average milliseconds used by each lap.
 * @throws TimerException if this Timer is running.
 */
public double meanLapTime() {
   if (running) throw new TimerException();
   return toMillisecs(ticks) / laps;
}

/**
 * Pause this timer at the end of a "lap" (repetition).
 * The lap counter will be incremented by one.
 *
 * @throws TimerException if this Timer is not running.
 */
public void pauseAndLap() {
   lap();
   ticks += getClock();
   running = false;
}

/**
 * Resume this timer to begin a new "lap" (repetition).
 *
 * @throws TimerException if this Timer is already running.
 */
public void resume() {
   if (running) throw new TimerException();
   ticks -= getClock();
   running = true;
}
```

```
/**
 * Increment the lap counter without pausing.
 * This is the equivalent of calling pause and resume.
 *
 * @throws TimerException if this Timer is not running.
 */
public void lap() {
   if (!running) throw new TimerException();
   laps++;
}

/**
 * Pause this timer during a "lap" (repetition).
 * The lap counter will remain the same.
 *
 * @throws TimerException if this Timer is not running.
 */
public void pause() {
   pauseAndLap();
   laps--;
}

/**
 * Method to yield the total number of milliseconds elapsed.
 * NOTE: an exception will be thrown if this is called while the timer is running.
 *
 * @return the total number of milliseconds elapsed for this timer.
 */
public double millisecs() {
   if (running) throw new TimerException();
   return toMillisecs(ticks);
}

@Override
public String toString() {
   return "Timer{" +
         "ticks=" + ticks +
         ", laps=" + laps +
         ", running=" + running +
         '}';
}

/**
```

```
 * Construct a new Timer and set it running.
 */
public Timer() {
   resume();
}

private long ticks = 0L;
private int laps = 0;
private boolean running = false;

// NOTE: Used by unit tests
private long getTicks() {
   return ticks;
}

// NOTE: Used by unit tests
private int getLaps() {
   return laps;
}

// NOTE: Used by unit tests
private boolean isRunning() {
   return running;
}

/**
 * Get the number of ticks from the system clock.
 * <p>
 * NOTE: (Maintain consistency) There are two system methods for getting the clock time.
 * Ensure that this method is consistent with toMillisecs.
 *
 * @return the number of ticks for the system clock. Currently defined as nano time.
 */
private static long getClock() {
       return System.nanoTime();
}

/**
 * NOTE: (Maintain consistency) There are two system methods for getting the clock time.
 * Ensure that this method is consistent with getTicks.
 *
 * @param ticks the number of clock ticks -- currently in nanoseconds.
 * @return the corresponding number of milliseconds.
 */
```

```
    private static double toMillisecs(long ticks) {
        return ticks * 1.0 / Math.pow(10, 6);
    }

    final static LazyLogger logger = new LazyLogger(Timer.class);

    static class TimerException extends RuntimeException {
        public TimerException() {
        }

        public TimerException(String message) {
            super(message);
        }

        public TimerException(String message, Throwable cause) {
            super(message, cause);
        }

        public TimerException(Throwable cause) {
            super(cause);
        }
    }
}
```

Benchmark_Timer.java

```
package edu.neu.coe.info6205.util;


import java.util.Arrays;

import java.util.Collections;

import java.util.function.Consumer;

import java.util.function.Function;

import java.util.function.Supplier;

import java.util.function.UnaryOperator;


import edu.neu.coe.info6205.sort.elementary.InsertionSort;
```

```
import static edu.neu.coe.info6205.util.Utilities.formatWhole;


/**
 * This class implements a simple Benchmark utility for measuring the running time of algorithms.
 * It is part of the repository for the INFO6205 class, taught by Prof. Robin Hillyard
 * <p>
 * It requires Java 8 as it uses function types, in particular, UnaryOperator&lt;T&gt; (a function of T => T),
 * Consumer&lt;T&gt; (essentially a function of T => Void) and Supplier&lt;T&gt; (essentially a function of Void => T).
 * <p>
 * In general, the benchmark class handles three phases of a "run:"
 * <ol>
 *     <li>The pre-function which prepares the input to the study function (field fPre) (may be null);</li>
 *     <li>The study function itself (field fRun) -- assumed to be a mutating function since it does not return a result;</li>
 *     <li>The post-function which cleans up and/or checks the results of the study function (field fPost) (may be null).</li>
 * </ol>
 * <p>
 * Note that the clock does not run during invocations of the pre-function and the post-function (if any).
 *
 * @param <T> The generic type T is that of the input to the function f which you will pass in to the constructor.
 */
public class Benchmark_Timer<T> implements Benchmark<T> {
```

```
        public static void main(String args[]) {
    InsertionSort<Integer> insertionSort = new InsertionSort<Integer>();
    Consumer<Integer[]> consumer= (array) -> insertionSort.sort(array, 0, array.length);
    Benchmark_Timer<Integer[]> benchmarkTimer = new
Benchmark_Timer<Integer[]>("Insertion Sort", consumer);


    String[] orderings = new String[] {"Randomly Ordered", "Ordered", "Reverse Ordered",
"Partially Ordered"};
    int orderingIdx = 0;
    System.out.println("\n" + orderings[orderingIdx] + ":");


    int arrayLength = 500;
    while (arrayLength <= 8000) {
        int currArrayLength = arrayLength;
        int currOrderingIdx = orderingIdx;
        System.out.println("N = " + currArrayLength);


        Supplier<Integer[]> supplier = () -> {
                Integer[] arr = new Integer[currArrayLength];
                for (int i = 0; i < currArrayLength; i++) {
                arr[i] = (int)(Math.random() * 1000);
        }


                if (orderings[currOrderingIdx].equals("Ordered")) {
                        Arrays.sort(arr);
                }
                else if (orderings[currOrderingIdx].equals("Reverse Ordered")) {
                        Arrays.sort(arr, Collections.reverseOrder());
```

```
                }
                else if (orderings[currOrderingIdx].equals("Partially Ordered")) {
                        Arrays.sort(arr);
                        for (int j = currArrayLength / 2; j < currArrayLength; j++) {
                                arr[j] = (int)(Math.random() * 1000);
                        }
                }

                return arr;
        };

        System.out.println("Time taken = " + benchmarkTimer.runFromSupplier(supplier, 20) +
"\n");

        if (arrayLength == 8000 && orderingIdx < orderings.length - 1) {
                arrayLength = 500;
                orderingIdx += 1;
                System.out.println("\n" + orderings[orderingIdx] + ":");

        }
        else {
                arrayLength *= 2;
        }
    }
        }

  /**
```

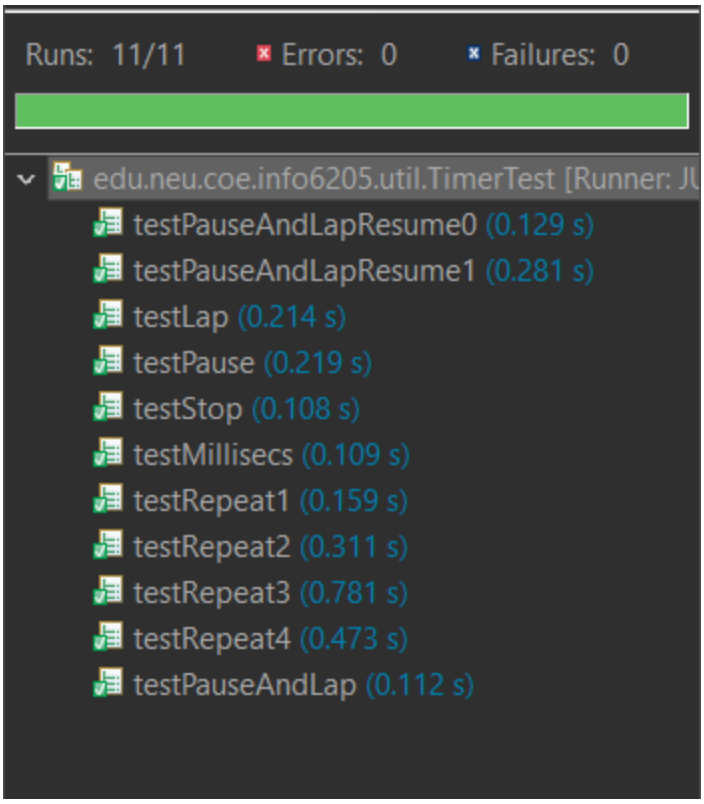
```
 * Calculate the appropriate number of warmup runs.
 *
 * @param m the number of runs.
 * @return at least 2 and at most the lower of 6 or m/15.
 */
static int getWarmupRuns(int m) {
   return Integer.max(2, Integer.min(6, m / 15));
}


/**
 * Run function f m times and return the average time in milliseconds.
 *
 * @param supplier a Supplier of a T
 * @param m        the number of times the function f will be called.
 * @return the average number of milliseconds taken for each run of function f.
 */
@Override
public double runFromSupplier(Supplier<T> supplier, int m) {
   logger.info("Begin run: " + description + " with " + formatWhole(m) + " runs");
   // Warmup phase
   final Function<T, T> function = t -> {
      fRun.accept(t);
      return t;
   };
   new Timer().repeat(getWarmupRuns(m), supplier, function, fPre, null);


   // Timed phase
```

```
        return new Timer().repeat(m, supplier, function, fPre, fPost);
    }


    /**
     * Constructor for a Benchmark_Timer with option of specifying all three functions.
     *
     * @param description the description of the benchmark.
     * @param fPre       a function of T => T.
     *                   Function fPre is run before each invocation of fRun (but with the clock stopped).
     *                   The result of fPre (if any) is passed to fRun.
     * @param fRun       a Consumer function (i.e. a function of T => Void).
     *                   Function fRun is the function whose timing you want to measure. For example, you might create a function which sorts an array.
     *                   When you create a lambda defining fRun, you must return "null."
     * @param fPost      a Consumer function (i.e. a function of T => Void).
     */
    public Benchmark_Timer(String description, UnaryOperator<T> fPre, Consumer<T> fRun, Consumer<T> fPost) {
        this.description = description;
        this.fPre = fPre;
        this.fRun = fRun;
        this.fPost = fPost;
    }


    /**
     * Constructor for a Benchmark_Timer with option of specifying all three functions.
     *
     * @param description the description of the benchmark.
```

```
    * @param fPre       a function of T => T.
    *                   Function fPre is run before each invocation of fRun (but with the clock stopped).
    *                   The result of fPre (if any) is passed to fRun.
    * @param fRun       a Consumer function (i.e. a function of T => Void).
    *                   Function fRun is the function whose timing you want to measure. For example,
you might create a function which sorts an array.
    */
   public Benchmark_Timer(String description, UnaryOperator<T> fPre, Consumer<T> fRun) {
      this(description, fPre, fRun, null);
   }

   /**
    * Constructor for a Benchmark_Timer with only fRun and fPost Consumer parameters.
    *
    * @param description the description of the benchmark.
    * @param fRun       a Consumer function (i.e. a function of T => Void).
    *                   Function fRun is the function whose timing you want to measure. For example,
you might create a function which sorts an array.
    *                   When you create a lambda defining fRun, you must return "null."
    * @param fPost      a Consumer function (i.e. a function of T => Void).
    */
   public Benchmark_Timer(String description, Consumer<T> fRun, Consumer<T> fPost) {
      this(description, null, fRun, fPost);
   }

   /**
    * Constructor for a Benchmark_Timer where only the (timed) run function is specified.
    *
```

```
    * @param description the description of the benchmark.
    * @param f           a Consumer function (i.e. a function of T => Void).
    *                    Function f is the function whose timing you want to measure. For example, you might create a function which sorts an array.
    */
   public Benchmark_Timer(String description, Consumer<T> f) {
      this(description, null, f, null);
   }

   private final String description;
   private final UnaryOperator<T> fPre;
   private final Consumer<T> fRun;
   private final Consumer<T> fPost;

   final static LazyLogger logger = new LazyLogger(Benchmark_Timer.class);
}
```

Unit Test Screenshots and Code:



TimerTest.java

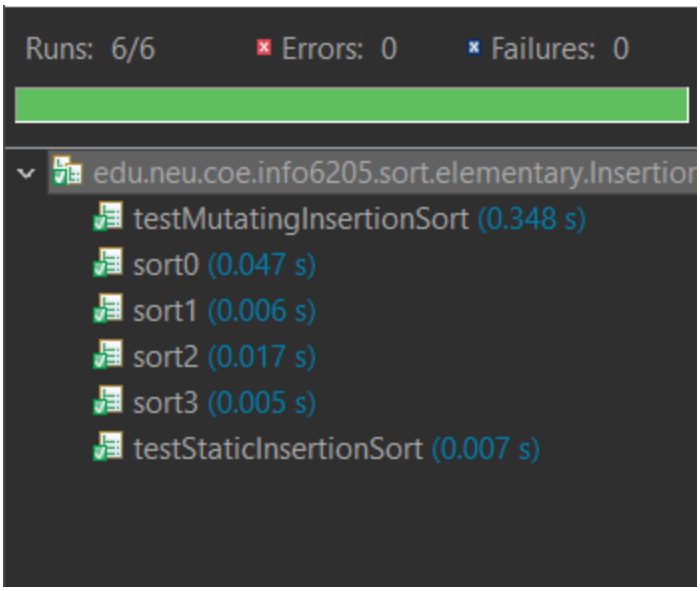
```
package edu.neu.coe.info6205.util;

import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class TimerTest {

    @Before
    public void setup() {
        pre = 0;
        run = 0;
        post = 0;
        result = 0;
    }

    @Test
    public void testStop() {
        final Timer timer = new Timer();
        GoToSleep(TENTH, 0);
```

```
    final double time = timer.stop();
    assertEquals(TENTH_DOUBLE, time, 10);
    assertEquals(1, run);
    assertEquals(1, new PrivateMethodTester(timer).invokePrivate("getLaps"));
}

@Test
public void testPauseAndLap() {
    final Timer timer = new Timer();
    final PrivateMethodTester privateMethodTester = new PrivateMethodTester(timer);
    GoToSleep(TENTH, 0);
    timer.pauseAndLap();
    final Long ticks = (Long) privateMethodTester.invokePrivate("getTicks");
    assertEquals(TENTH_DOUBLE, ticks / 1e6, 12);
    assertFalse((Boolean) privateMethodTester.invokePrivate("isRunning"));
    assertEquals(1, privateMethodTester.invokePrivate("getLaps"));
}

@Test
public void testPauseAndLapResume0() {
    final Timer timer = new Timer();
    final PrivateMethodTester privateMethodTester = new PrivateMethodTester(timer);
    GoToSleep(TENTH, 0);
    timer.pauseAndLap();
    timer.resume();
    assertTrue((Boolean) privateMethodTester.invokePrivate("isRunning"));
    assertEquals(1, privateMethodTester.invokePrivate("getLaps"));
}

@Test
public void testPauseAndLapResume1() {
    final Timer timer = new Timer();
    GoToSleep(TENTH, 0);
    timer.pauseAndLap();
    GoToSleep(TENTH, 0);
    timer.resume();
    GoToSleep(TENTH, 0);
    final double time = timer.stop();
    assertEquals(TENTH_DOUBLE, time, 10.0);
    assertEquals(3, run);
}

@Test
public void testLap() {
```

```
    final Timer timer = new Timer();
    GoToSleep(TENTH, 0);
    timer.lap();
    GoToSleep(TENTH, 0);
    final double time = timer.stop();
    assertEquals(TENTH_DOUBLE, time, 10.0);
    assertEquals(2, run);
  }

  @Test
  public void testPause() {
    final Timer timer = new Timer();
    GoToSleep(TENTH, 0);
    timer.pause();
    GoToSleep(TENTH, 0);
    timer.resume();
    final double time = timer.stop();
    assertEquals(TENTH_DOUBLE, time, 10.0);
    assertEquals(2, run);
  }

  @Test
  public void testMillisecs() {
    final Timer timer = new Timer();
    GoToSleep(TENTH, 0);
    timer.stop();
    final double time = timer.millisecs();
    assertEquals(TENTH_DOUBLE, time, 10.0);
    assertEquals(1, run);
  }

  @Test
  public void testRepeat1() {
    final Timer timer = new Timer();
    final double mean = timer.repeat(10, () -> {
      GoToSleep(HUNDREDTH, 0);
      return null;
    });
    assertEquals(10, new PrivateMethodTester(timer).invokePrivate("getLaps"));
    assertEquals(TENTH_DOUBLE / 10, mean, 6);
    assertEquals(10, run);
    assertEquals(0, pre);
    assertEquals(0, post);
  }
```

```
@Test
public void testRepeat2() {
   final Timer timer = new Timer();
   final int zzz = 20;
   final double mean = timer.repeat(10, () -> zzz, t -> {
      GoToSleep(t, 0);
      return null;
   });
   assertEquals(10, new PrivateMethodTester(timer).invokePrivate("getLaps"));
   assertEquals(zzz, mean, 12);
   assertEquals(10, run);
   assertEquals(0, pre);
   assertEquals(0, post);
}

@Test // Slow
public void testRepeat3() {
   final Timer timer = new Timer();
   final int zzz = 20;
   final double mean = timer.repeat(10, () -> zzz, t -> {
      GoToSleep(t, 0);
      return null;
   }, t -> {
      GoToSleep(t, -1);
      return t;
   }, t -> GoToSleep(10, 1));
   assertEquals(10, new PrivateMethodTester(timer).invokePrivate("getLaps"));
   assertEquals(zzz, mean, 12);
   assertEquals(10, run);
   assertEquals(10, pre);
   assertEquals(10, post);
}

@Test // Slow
public void testRepeat4() {
   final Timer timer = new Timer();
   final int zzz = 20;
   final double mean = timer.repeat(10,
         () -> zzz, // supplier
         t -> { // function
      result = t;
      GoToSleep(10, 0);
      return null;
```

```
        }, t -> { // pre-function
            GoToSleep(10, -1);
            return 2*t;
        }, t -> GoToSleep(10, 1) // post-function
        );
        assertEquals(10, new PrivateMethodTester(timer).invokePrivate("getLaps"));
        assertEquals(zzz, 20, 6);
        assertEquals(10, run);
        assertEquals(10, pre);
        assertEquals(10, post);
        // This test is designed to ensure that the preFunction is properly implemented in repeat.
        assertEquals(40, result);
    }

    int pre = 0;
    int run = 0;
    int post = 0;
    int result = 0;

    private void GoToSleep(long mSecs, int which) {
        try {
            Thread.sleep(mSecs);
            if (which == 0) run++;
            else if (which > 0) post++;
            else pre++;
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static final int TENTH = 100;
    public static final double TENTH_DOUBLE = 100;
    public static final int HUNDREDTH = 10;

}
```



InsertionSortTest.java

```
package edu.neu.coe.info6205.sort.elementary;

import edu.neu.coe.info6205.sort.*;
import edu.neu.coe.info6205.util.Config;
import edu.neu.coe.info6205.util.LazyLogger;
import edu.neu.coe.info6205.util.PrivateMethodTester;
import edu.neu.coe.info6205.util.StatPack;
import org.junit.Test;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
```

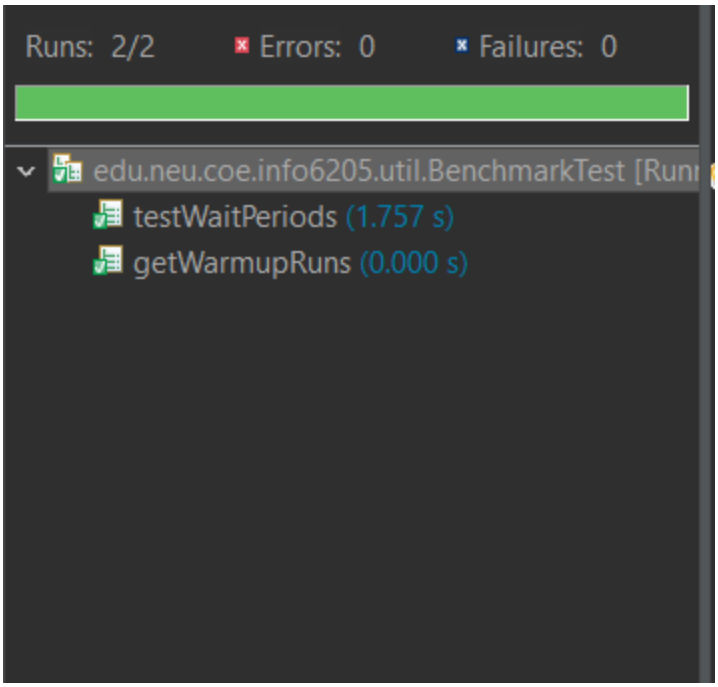
```
@SuppressWarnings("ALL")
public class InsertionSortTest {

   @Test
   public void sort0() throws Exception {
      final List<Integer> list = new ArrayList<>();
      list.add(1);
      list.add(2);
      list.add(3);
      list.add(4);
      Integer[] xs = list.toArray(new Integer[0]);
      final Config config = Config.setupConfig("true", "0", "1", "", "");
      Helper<Integer> helper = HelperFactory.create("InsertionSort", list.size(), config);
      helper.init(list.size());
      final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
      final StatPack statPack = (StatPack) privateMethodTester.invokePrivate("getStatPack");
      SortWithHelper<Integer> sorter = new InsertionSort<Integer>(helper);
      sorter.preProcess(xs);
      Integer[] ys = sorter.sort(xs);
      assertTrue(helper.sorted(ys));
      sorter.postProcess(ys);
      final int compares = (int) statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
      assertEquals(list.size() - 1, compares);
      final int inversions = (int) statPack.getStatistics(InstrumentedHelper.INVERSIONS).mean();
      assertEquals(0L, inversions);
      final int fixes = (int) statPack.getStatistics(InstrumentedHelper.FIXES).mean();
```

```
    assertEquals(inversions, fixes);
  }


  @Test
  public void sort1() throws Exception {
    final List<Integer> list = new ArrayList<>();
    list.add(3);
    list.add(4);
    list.add(2);
    list.add(1);
    Integer[] xs = list.toArray(new Integer[0]);
    BaseHelper<Integer> helper = new BaseHelper<>("InsertionSort", xs.length,
Config.load(InsertionSortTest.class));
    GenericSort<Integer> sorter = new InsertionSort<Integer>(helper);
    Integer[] ys = sorter.sort(xs);
    assertTrue(helper.sorted(ys));
    System.out.println(sorter.toString());
  }


  @Test
  public void testMutatingInsertionSort() throws IOException {
    final List<Integer> list = new ArrayList<>();
    list.add(3);
    list.add(4);
    list.add(2);
    list.add(1);
    Integer[] xs = list.toArray(new Integer[0]);
```

```
    BaseHelper<Integer> helper = new BaseHelper<>("InsertionSort", xs.length,
Config.load(InsertionSortTest.class));
    GenericSort<Integer> sorter = new InsertionSort<Integer>(helper);
    sorter.mutatingSort(xs);
    assertTrue(helper.sorted(xs));
  }

  @Test
  public void testStaticInsertionSort() throws IOException {
    final List<Integer> list = new ArrayList<>();
    list.add(3);
    list.add(4);
    list.add(2);
    list.add(1);
    Integer[] xs = list.toArray(new Integer[0]);
    InsertionSort.sort(xs);
    assertTrue(xs[0] < xs[1] && xs[1] < xs[2] && xs[2] < xs[3]);
  }

  @Test
  public void sort2() throws Exception {
    final Config config = Config.setupConfig("true", "0", "1", "", "");
    int n = 100;
    Helper<Integer> helper = HelperFactory.create("InsertionSort", n, config);
    helper.init(n);
    final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
    final StatPack statPack = (StatPack) privateMethodTester.invokePrivate("getStatPack");
```

```
    Integer[] xs = helper.random(Integer.class, r -> r.nextInt(1000));
    SortWithHelper<Integer> sorter = new InsertionSort<Integer>(helper);
    sorter.preProcess(xs);
    Integer[] ys = sorter.sort(xs);
    assertTrue(helper.sorted(ys));
    sorter.postProcess(ys);
    final int compares = (int) statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
    // NOTE: these are suppoed to match within about 12%.
    // Since we set a specific seed, this should always succeed.
    // If we use true random seed and this test fails, just increase the delta a little.
    assertEquals(1.0, 4.0 * compares / n / (n - 1), 0.12);
    final int inversions = (int) statPack.getStatistics(InstrumentedHelper.INVERSIONS).mean();
    final int fixes = (int) statPack.getStatistics(InstrumentedHelper.FIXES).mean();
    System.out.println(statPack);
    assertEquals(inversions, fixes);
}

@Test
public void sort3() throws Exception {
    final Config config = Config.setupConfig("true", "0", "1", "", "");
    int n = 100;
    Helper<Integer> helper = HelperFactory.create("InsertionSort", n, config);
    helper.init(n);
    final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
    final StatPack statPack = (StatPack) privateMethodTester.invokePrivate("getStatPack");
    Integer[] xs = new Integer[n];
    for (int i = 0; i < n; i++) xs[i] = n - i;
```

```
    SortWithHelper<Integer> sorter = new InsertionSort<Integer>(helper);
    sorter.preProcess(xs);
    Integer[] ys = sorter.sort(xs);
    assertTrue(helper.sorted(ys));
    sorter.postProcess(ys);
    final int compares = (int) statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
    // NOTE: these are suppoed to match within about 12%.
    // Since we set a specific seed, this should always succeed.
    // If we use true random seed and this test fails, just increase the delta a little.
    assertEquals(4950, compares);
    final int inversions = (int) statPack.getStatistics(InstrumentedHelper.INVERSIONS).mean();
    final int fixes = (int) statPack.getStatistics(InstrumentedHelper.FIXES).mean();
    System.out.println(statPack);
    assertEquals(inversions, fixes);
  }

  final static LazyLogger logger = new LazyLogger(InsertionSort.class);

}
```



BenchmarkTest.java

```
package edu.neu.coe.info6205.util;


import org.junit.Test;


import static org.junit.Assert.assertEquals;


@SuppressWarnings("ALL")
public class BenchmarkTest {


    int pre = 0;
    int run = 0;
    int post = 0;


    @Test // Slow
```

```
public void testWaitPeriods() throws Exception {
    int nRuns = 2;
    int warmups = 2;
    Benchmark<Boolean> bm = new Benchmark_Timer<>(
            "testWaitPeriods", b -> {
        GoToSleep(100L, -1);
        return null;
    },
            b -> {
                GoToSleep(200L, 0);
            },
            b -> {
                GoToSleep(50L, 1);
            });
    double x = bm.run(true, nRuns);
    assertEquals(nRuns, post);
    assertEquals(nRuns + warmups, run);
    assertEquals(nRuns + warmups, pre);
    assertEquals(200, x, 10);
}

private void GoToSleep(long mSecs, int which) {
    try {
        Thread.sleep(mSecs);
        if (which == 0) run++;
        else if (which > 0) post++;
        else pre++;
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Test
    public void getWarmupRuns() {
        assertEquals(2, Benchmark_Timer.getWarmupRuns(0));
        assertEquals(2, Benchmark_Timer.getWarmupRuns(20));
        assertEquals(3, Benchmark_Timer.getWarmupRuns(45));
        assertEquals(6, Benchmark_Timer.getWarmupRuns(100));
        assertEquals(6, Benchmark_Timer.getWarmupRuns(1000));
    }
}
```