

# Program Structures and Algorithms

## Spring 2023(SEC - 8)

**NAME:** Dinesh Singh Shekhawat

**NUID:** 002776484

### **TASK:**

1. Implementing three methods, viz., repeat, getClock, and toMillisecs, of Timer class
2. Implement **InsertionSort** in the InsertionSort class and run the unit tests in InsertionSortTest so that all of them pass.
3. Implementing a main program (or unit tests) to run the benchmarks on the running times of this sort, using four different initial array ordering situations, viz., random, ordered, partially-ordered and reverse-ordered by using integer arrays for at least five different values of 'n' and drawing conclusions regarding the order of growth.

### **RELATIONSHIP CONCLUSION:**

1. Insertion sort takes the most amount of time for a reverse ordered array as all the elements are swapped continuously to reverse the array again to make it sorted. The swapping occurs on comparison of each pair of elements since the former element is always greater than the latter one and this process keeps going on till the last element in the iteration is not reached.
2. Random array sorting takes less time than that of a reverse ordered array since there are chances that random elements generated from the random function have some numbers which are already in their correct positions and swapping doesn't happen for all pairs of elements being compared.
3. Partially sorted array takes even less time than a random array because half of the array is already sorted and swapping of elements on pair comparison only happens in the other half of the array.
4. A sorted array takes the least amount of time, which is expected since not even a single swapping takes place as all the elements are in their correct positions already.

## EVIDENCE TO SUPPORT THAT CONCLUSION

### Logs

```
2023-02-04 20:39:26 INFO InsertionSortBenchmarkTest - >>>>> All
benchmarking tests will start now
2023-02-04 20:39:26 INFO InsertionSortBenchmarkTest - Starting
benchmarking of insertion sort tests with random ordered array
2023-02-04 20:39:26 INFO Benchmark_Timer - Begin run: Insertion sort:
Random Ordered Array: N = 1000 with 100 runs
2023-02-04 20:39:26 INFO InsertionSortBenchmarkTest - Insertion sort:
Random Ordered Array: N = 1000, Benchmarking Result: 1.52
2023-02-04 20:39:26 INFO Benchmark_Timer - Begin run: Insertion sort:
Random Ordered Array: N = 2000 with 100 runs
2023-02-04 20:39:27 INFO InsertionSortBenchmarkTest - Insertion sort:
Random Ordered Array: N = 2000, Benchmarking Result: 4.46
2023-02-04 20:39:27 INFO Benchmark_Timer - Begin run: Insertion sort:
Random Ordered Array: N = 4000 with 100 runs
2023-02-04 20:39:29 INFO InsertionSortBenchmarkTest - Insertion sort:
Random Ordered Array: N = 4000, Benchmarking Result: 18.49
2023-02-04 20:39:30 INFO Benchmark_Timer - Begin run: Insertion sort:
Random Ordered Array: N = 8000 with 100 runs
2023-02-04 20:39:38 INFO InsertionSortBenchmarkTest - Insertion sort:
Random Ordered Array: N = 8000, Benchmarking Result: 80.57
2023-02-04 20:39:42 INFO Benchmark_Timer - Begin run: Insertion sort:
Random Ordered Array: N = 16000 with 100 runs
2023-02-04 20:40:21 INFO InsertionSortBenchmarkTest - Insertion sort:
Random Ordered Array: N = 16000, Benchmarking Result: 366.04

2023-02-04 20:40:21 INFO InsertionSortBenchmarkTest - Starting
benchmarking of insertion sort tests with reverse ordered array
2023-02-04 20:40:21 INFO Benchmark_Timer - Begin run: Insertion sort:
Reverse Ordered Array: N = 1000 with 100 runs
2023-02-04 20:40:21 INFO InsertionSortBenchmarkTest - Insertion sort:
Reverse Ordered Array: N = 1000, Benchmarking Result: 2.35
2023-02-04 20:40:21 INFO Benchmark_Timer - Begin run: Insertion sort:
Reverse Ordered Array: N = 2000 with 100 runs
2023-02-04 20:40:22 INFO InsertionSortBenchmarkTest - Insertion sort:
Reverse Ordered Array: N = 2000, Benchmarking Result: 8.94
2023-02-04 20:40:23 INFO Benchmark_Timer - Begin run: Insertion sort:
Reverse Ordered Array: N = 4000 with 100 runs
2023-02-04 20:40:26 INFO InsertionSortBenchmarkTest - Insertion sort:
Reverse Ordered Array: N = 4000, Benchmarking Result: 36.05
```

2023-02-04 20:40:28 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Reverse Ordered Array: N = 8000 with 100 runs

2023-02-04 20:40:44 INFO InsertionSortBenchmarkTest - Insertion sort:  
Reverse Ordered Array: N = 8000, Benchmarking Result: 149.0

2023-02-04 20:40:50 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Reverse Ordered Array: N = 16000 with 100 runs

2023-02-04 20:41:57 INFO InsertionSortBenchmarkTest - Insertion sort:  
Reverse Ordered Array: N = 16000, Benchmarking Result: 628.57

2023-02-04 20:41:57 INFO InsertionSortBenchmarkTest - Starting  
benchmarking of insertion sort tests with partially ordered array

2023-02-04 20:41:57 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Partially Ordered Array: N = 1000 with 100 runs

2023-02-04 20:41:57 INFO InsertionSortBenchmarkTest - Insertion sort:  
Partially Ordered Array: N = 1000, Benchmarking Result: 0.92

2023-02-04 20:41:57 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Partially Ordered Array: N = 2000 with 100 runs

2023-02-04 20:41:58 INFO InsertionSortBenchmarkTest - Insertion sort:  
Partially Ordered Array: N = 2000, Benchmarking Result: 3.76

2023-02-04 20:41:58 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Partially Ordered Array: N = 4000 with 100 runs

2023-02-04 20:41:59 INFO InsertionSortBenchmarkTest - Insertion sort:  
Partially Ordered Array: N = 4000, Benchmarking Result: 13.81

2023-02-04 20:42:00 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Partially Ordered Array: N = 8000 with 100 runs

2023-02-04 20:42:06 INFO InsertionSortBenchmarkTest - Insertion sort:  
Partially Ordered Array: N = 8000, Benchmarking Result: 55.85

2023-02-04 20:42:08 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Partially Ordered Array: N = 16000 with 100 runs

2023-02-04 20:42:33 INFO InsertionSortBenchmarkTest - Insertion sort:  
Partially Ordered Array: N = 16000, Benchmarking Result: 234.74

2023-02-04 20:42:33 INFO InsertionSortBenchmarkTest - Starting  
benchmarking of insertion sort tests with ordered array

2023-02-04 20:42:33 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Ordered Array: N = 1000 with 100 runs

2023-02-04 20:42:33 INFO InsertionSortBenchmarkTest - Insertion sort:  
Ordered Array: N = 1000, Benchmarking Result: 0.0

2023-02-04 20:42:33 INFO Benchmark\_Timer - Begin run: Insertion sort:  
Ordered Array: N = 2000 with 100 runs

2023-02-04 20:42:33 INFO InsertionSortBenchmarkTest - Insertion sort:  
Ordered Array: N = 2000, Benchmarking Result: 0.02

```
2023-02-04 20:42:33 INFO Benchmark_Timer - Begin run: Insertion sort:
Ordered Array: N = 4000 with 100 runs
2023-02-04 20:42:33 INFO InsertionSortBenchmarkTest - Insertion sort:
Ordered Array: N = 4000, Benchmarking Result: 0.02
2023-02-04 20:42:33 INFO Benchmark_Timer - Begin run: Insertion sort:
Ordered Array: N = 8000 with 100 runs
2023-02-04 20:42:33 INFO InsertionSortBenchmarkTest - Insertion sort:
Ordered Array: N = 8000, Benchmarking Result: 0.07
2023-02-04 20:42:33 INFO Benchmark_Timer - Begin run: Insertion sort:
Ordered Array: N = 16000 with 100 runs
2023-02-04 20:42:33 INFO InsertionSortBenchmarkTest - Insertion sort:
Ordered Array: N = 16000, Benchmarking Result: 0.11
2023-02-04 20:42:33 INFO InsertionSortBenchmarkTest - All benchmarking
tests done >>>>>>>>>
```

## Source Code

### Timer

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

import java.util.concurrent.TimeUnit;
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 (any 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");

    /**
     * It is important to do this step because

```

```

        * as soon as the Constructor is called for this
        * class on object initialisation it is set
        * to running state by calling resume() method.
        *
        * Check the constructor definition of this class.
        */
    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);
        }
    }

    return meanLapTime();
}

/**
 * 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 TimeUnit.MILLISECONDS.convert(ticks, TimeUnit.NANOSECONDS);
}

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);  
    }  
}
```

## InsertionSort

```
/*
 (c) Copyright 2018, 2019 Phasmid Software
 */
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 + 1; i < to; i++) {
            int j = i;
            while (j > from && helper.swapStableConditional(xs, j)) {
                j--;
            }
        }
    }

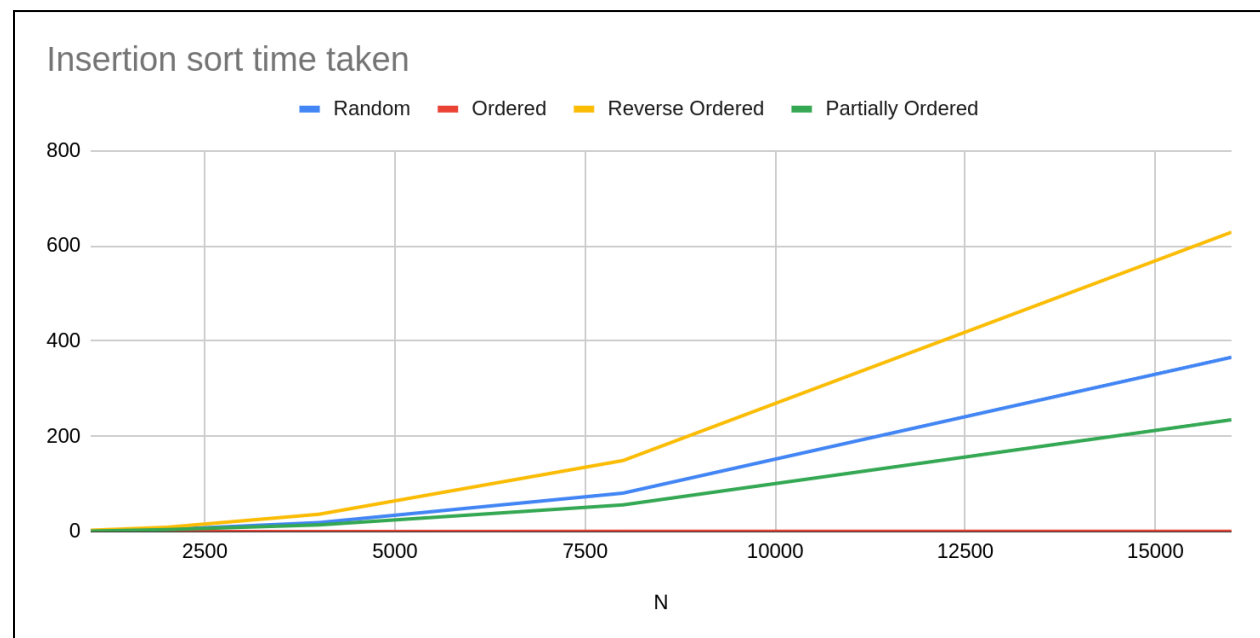
    public static final String DESCRIPTION = "Insertion sort";

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

```

## GRAPHICAL REPRESENTATION

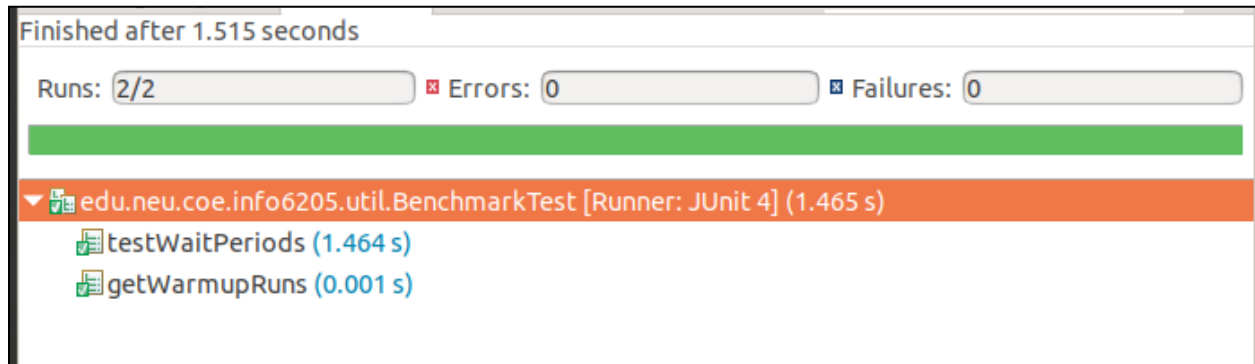
N	Time Taken (milliseconds)			
	Random	Ordered	Reverse Ordered	Partially Ordered
1000	1.52	0	2.35	0.92
2000	4.46	0.02	8.94	3.76
4000	18.49	0.02	36.05	13.81
8000	80.57	0.07	149	55.85
16000	366.04	0.11	628.57	234.74



## UNIT TEST SCREENSHOTS

### BenchmarkTest

#### Report



#### Code

```
/*
 * Copyright (c) 2017. Phasmid Software
 */

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));
}
}

```



## TimerTest

### Report

Finished after 2.443 seconds

Runs: 11/11    ✖ Errors: 0    ✖ Failures: 0

▼ edu.neu.coe.info6205.util.TimerTest [Runner: JUnit 4] (2.390 s)

- testPauseAndLapResume0 (0.256 s)
- testPauseAndLapResume1 (0.302 s)
- testLap (0.201 s)
- testPause (0.201 s)
- testStop (0.101 s)
- testMillisecs (0.101 s)
- testRepeat1 (0.104 s)
- testRepeat2 (0.203 s)
- testRepeat3 (0.507 s)
- testRepeat4 (0.307 s)
- testPauseAndLap (0.103 s)

### Code

```
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, 8.5);
        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, 6);
    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) {

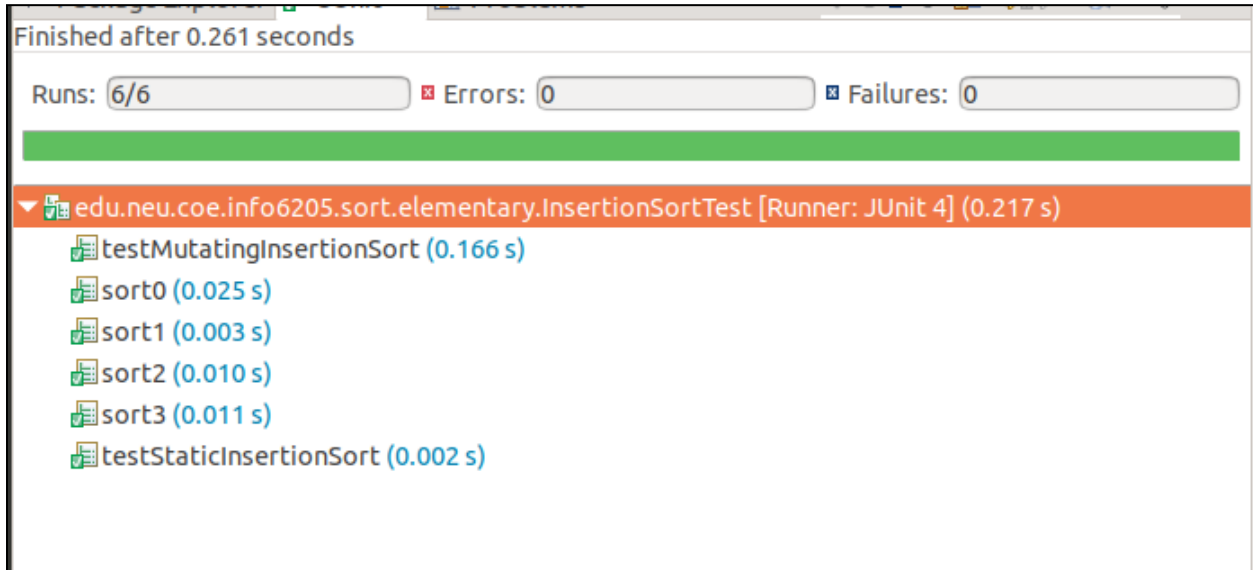
```

```
        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

### Report



Finished after 0.261 seconds

Runs: 6/6    ✖ Errors: 0    ✖ Failures: 0

▼ edu.neu.coe.info6205.sort.elementary.InsertionSortTest [Runner: JUnit 4] (0.217 s)

- testMutatingInsertionSort (0.166 s)
- sort0 (0.025 s)
- sort1 (0.003 s)
- sort2 (0.010 s)
- sort3 (0.011 s)
- testStaticInsertionSort (0.002 s)

### Code

```
/*
 * Copyright (c) 2017. Phasmid Software
 */

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 supposed 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);

```

```

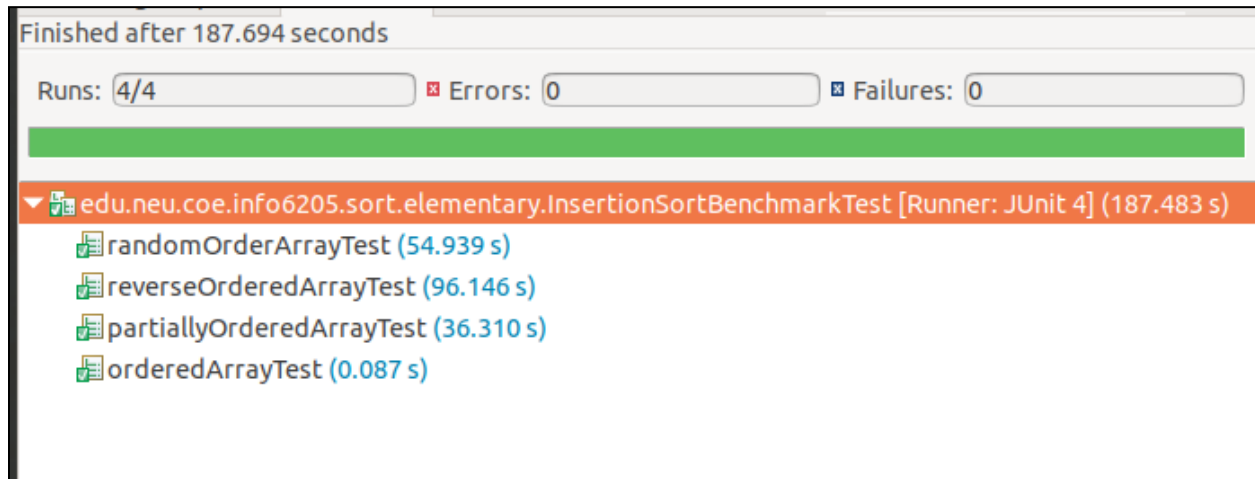
        sorter.preProcess(xs);
        Integer[] ys = sorter.sort(xs);
        assertTrue(helper.sorted(ys));
        sorter.postProcess(ys);
        final int compares = (int)
statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
        // NOTE: these are supposed to match within about 12%.
        // 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);
}

```

## InsertionSortBenchmarkTest

### Report



### Code

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

import java.util.Random;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

import edu.neu.coe.info6205.sort.GenericSort;
import edu.neu.coe.info6205.util.Benchmark_Timer;
import edu.neu.coe.info6205.util.LazyLogger;

public class InsertionSortBenchmarkTest {
    private static final LazyLogger logger = new
    LazyLogger(InsertionSortBenchmarkTest.class);

    private static final String DESCRIPTION_TEMPLATE = "Insertion sort: %s:
    N = %s";

    private static final String ARRAY_TYPE_ORDERED = "Ordered Array";
```

```

    private static final String ARRAY_TYPE_RANDOM_ORDERED = "Random Ordered
Array";
    private static final String ARRAY_TYPE_PARTIALLY_ORDERED = "Partially
Ordered Array";
    private static final String ARRAY_TYPE_REVERSE_ORDERED = "Reverse
Ordered Array";
    private static final int START = 1000;
    private static final int ITERATIONS = 5;
    private static final int WARM_UPS = 10;
    private static final int RUNS = 100;

    @BeforeClass
    public static void beforeClass() {
        logger.info(">>>>> All benchmarking tests will start now");
    }

    @AfterClass
    public static void afterClass() {
        logger.info("All benchmarking tests done >>>>>>>>");
    }

    @Test
    public void randomOrderArrayTest() {
        logger.info("Starting benchmarking of insertion sort tests with
random ordered array");

        Function<Integer, Supplier<Integer[]>> randomSupplierBuilder =
(length) -> {
            Supplier<Integer[]> orderedSupplier = () -> {
                Random random = new Random();
                Integer[] array = new Integer[length];
                for (int j = 0; j < length; j++) {
                    array[j] = random.nextInt();
                }
                return array;
            };

            return orderedSupplier;
        };

        benchmarkTest(randomSupplierBuilder, ARRAY_TYPE_RANDOM_ORDERED);
    }

```

```

@Test
public void orderedArrayTest() {
    logger.info("Starting benchmarking of insertion sort tests with
ordered array");

    Function<Integer, Supplier<Integer[]>> orderedSupplierBuilder =
(length) -> {
        Supplier<Integer[]> orderedSupplier = () -> {
            Integer[] array = new Integer[length];
            for (int j = 0; j < length; j++) {
                array[j] = j;
            }
            return array;
        };

        return orderedSupplier;
    };

    benchmarkTest(orderedSupplierBuilder, ARRAY_TYPE_ORDERED);
}

@Test
public void partiallyOrderedArrayTest() {
    logger.info("Starting benchmarking of insertion sort tests with
partially ordered array");

    Function<Integer, Supplier<Integer[]>> reverseOrderedSupplierBuilder
= (length) -> {
        Supplier<Integer[]> orderedSupplier = () -> {
            Random random = new Random();
            Integer[] array = new Integer[length];

            for (int i = 0; i < length / 2; i++) {
                array[i] = i;
            }

            for (int i = length/2; i < length; i++) {
                array[i] = random.nextInt();
            }

            return array;
        };
    };
}

```

```

        };

        return orderedSupplier;
    };

    benchmarkTest(reverseOrderedSupplierBuilder,
ARRAY_TYPE_PARTIALLY_ORDERED);
}

@Test
public void reverseOrderedArrayTest() {
    logger.info("Starting benchmarking of insertion sort tests with
reverse ordered array");

    Function<Integer, Supplier<Integer[]>> reverseOrderedSupplierBuilder
= (length) -> {
        Supplier<Integer[]> orderedSupplier = () -> {
            Integer[] array = new Integer[length];
            for (int j = 0; j < length; j++) {
                array[j] = (length - j);
            }
            return array;
        };

        return orderedSupplier;
    };

    benchmarkTest(reverseOrderedSupplierBuilder,
ARRAY_TYPE_REVERSE_ORDERED);
}

private void benchmarkTest(
    Function<Integer, Supplier<Integer[]>> supplierBuilder,
    String arrayOrderType) {
    for (int n = START, i = 0; i < ITERATIONS; n *= 2, i++) {
        Supplier<Integer[]> orderedSupplier = supplierBuilder.apply(n);

        String description = getDescriptionName(arrayOrderType, n);

        GenericSort<Integer> insertionSort = new InsertionSort<>();

        Consumer<Integer[]> function = array ->

```

```

insertionSort.sort(array);

        Benchmark_Timer<Integer[]> benchmarkTimer = new
Benchmark_Timer<>(description, function);

        /*
        *
        edu.neu.coe.info6205.util.Benchmark_Timer.runFromSupplier(Supplier<T>, int)
        *
        * Code inside this already takes care of warm up phase
        *
        * but still getting high time for first iteration so doing
        * runs explicitly for warm up
        * */
        for (int counter = 0; counter < WARM_UPS; counter++) {
            function.accept(orderedSupplier.get());
        }

        double time = benchmarkTimer.runFromSupplier(orderedSupplier,
RUNS);
        logger.info(description + ", Benchmarking Result: " + time);
    }
}

    private String getDescriptionName(String orderingType, int runs) {
        String result = String.format(DESCRIPTION_TEMPLATE, orderingType,
runs);
        return result;
    }
}

```