Varad Ratnakar Desai (001465732)

Program Structures & Algorithms

Spring 2021

Assignment No. 2

- **Task**

- (Part 1) You are to implement three methods 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. The APIs of these class are as follows:

```
public interface Benchmark<T> {
    default double run(T t, int m) {
        return runFromSupplier(() -> t, m);
    }

    double runFromSupplier(Supplier<T> supplier, int m);
}

public class Benchmark_Timer<T> implements Benchmark<T> {

public Benchmark_Timer(String description, UnaryOperator<T> fPre, Consumer<T> fRun, Consumer<T> fP
ost)

public Benchmark_Timer(String description, UnaryOperator<T> fPre, Consumer<T> fRun)

public Benchmark_Timer(String description, Consumer<T> fRun, Consumer<T> fPost)

public Benchmark_Timer(String description, Consumer<T> f)

public class Timer {
... // see below for methods to be implemented...
}

public <T, U> double repeat(int n, Supplier<T> supplier, Function<T, U> function, UnaryOperator<T>
preFunction, Consumer<U> postFunction) {
// TO BE IMPLEMENTED
}

private static long getClock() {
    // TO BE IMPLEMENTED
}

private static double toMillisecs(long ticks) {
    // TO BE IMPLEMENTED
}
```

The function to be timed, hereinafter the "target" function, is the *Consumer* function *fRun* (or just *f*) passed in to one or other of the constructors. For example, you might create a function which sorts an array with *n* elements.

The generic type *T* is that of the input to the target function.

The first parameter to the first run method signature is the parameter that will, in turn, be passed to target function. In the second signature, *supplier* will be invoked each time to get a *t* which is passed to the other run method.

The second parameter to the *run* function (*m)* is the number of times the target function will be called.

The return value from *run* is the average number of milliseconds taken for each run of the target function.

Don't forget to check your implementation by running the unit tests in *BenchmarkTest* and *TimerTest*.

- (Part 2) Implement *InsertionSort* (in the *InsertionSort* class) by simply looking up the insertion code used by *Arrays.sort.* You should use the *helper.swap* method although you could also just copy that from the same source code. You should of course run the unit tests in *InsertionSortTest*.
- (Part 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.

As usual, the submission will be your entire project (*clean, i.e. without the target and project folders).* There are stubs and unit tests in the repository.

Report on your observations and show screenshots of the runs and also the unit tests. Please note that you may have to adjust the required execution time for the insertion sort unit test(s) because your computer may not run at the same speed as mine.

Further notes: you should use the *System.nanoTime* method to get the clock time. This isn't guaranteed to be accurate which is one of the reasons you should run the experiment several times for each value of *n*. Also, for each invocation of *run*, run the given target function ten times to get the system "warmed up" before you start the timing properly.

The *Sort* interface takes care of copying the array when the *sort(array)* signature is called. It returns a new array as a result. The original array is unchanged. Therefore, you do not need to worry about the insertion-based sorts getting quicker because of the arrays getting more sorted (they don't).

- **Output**

```
Benchmark_Timer task starts!
Random Array Sorting 1 n = 200
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.2
Partially Sorted Array Sorting 1 n = 200
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.55
Fully Sorted Array Sorting 1 n = 200
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.05
Reverse Sorted Array Sorting 1 n = 200
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.35
Random Array Sorting 2 n = 400
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.45
Partially Sorted Array Sorting 2 n = 400
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.2
Fully Sorted Array Sorting 2 n = 400
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.0
Reverse Sorted Array Sorting 2 n = 400
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.4
Random Array Sorting 3 n = 800
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.75
Partially Sorted Array Sorting 3 n = 800
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.55
Fully Sorted Array Sorting 3 n = 800
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.0
Reverse Sorted Array Sorting 3 n = 800
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 1.65
```

```
Random Array Sorting 4 n = 1600
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 3.35
Partially Sorted Array Sorting 4 n = 1600
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 3.4
Fully Sorted Array Sorting 4 n = 1600
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.0
Reverse Sorted Array Sorting 4 n = 1600
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 7.55
Random Array Sorting 5 n = 3200
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 13.3
Partially Sorted Array Sorting 5 n = 3200
2021-02-03 11:24:49 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 13.9
Fully Sorted Array Sorting 5 n = 3200
2021-02-03 11:24:50 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.0
Reverse Sorted Array Sorting 5 n = 3200
2021-02-03 11:24:50 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 25.7
Random Array Sorting 6 n = 6400
2021-02-03 11:24:50 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 61.25
Partially Sorted Array Sorting 6 n = 6400
2021-02-03 11:24:52 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 53.3
Fully Sorted Array Sorting 6 n = 6400
2021-02-03 11:24:53 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.0
Reverse Sorted Array Sorting 6 n = 6400
2021-02-03 11:24:53 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 118.75
Random Array Sorting 7 n = 12800
2021-02-03 11:24:56 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
```

```
Mean lap time: 227.15
Partially Sorted Array Sorting 7 n = 12800
2021-02-03 11:25:01 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 227.9
Fully Sorted Array Sorting 7 n = 12800
2021-02-03 11:25:06 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.05
Reverse Sorted Array Sorting 7 n = 12800
2021-02-03 11:25:06 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 487.15
Random Array Sorting 8 n = 25600
2021-02-03 11:25:16 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 1084.15
Partially Sorted Array Sorting 8 n = 25600
2021-02-03 11:25:41 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 794.55
Fully Sorted Array Sorting 8 n = 25600
2021-02-03 11:25:59 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.1
Reverse Sorted Array Sorting 8 n = 25600
2021-02-03 11:25:59 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 2000.0
Random Array Sorting 9 n = 51200
2021-02-03 11:26:43 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 4704.95
Partially Sorted Array Sorting 9 n = 51200
2021-02-03 11:28:27 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 3161.4
Fully Sorted Array Sorting 9 n = 51200
2021-02-03 11:29:44 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.2
Reverse Sorted Array Sorting 9 n = 51200
2021-02-03 11:29:44 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 8161.7
Random Array Sorting 10 n = 102400
2021-02-03 11:32:44 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 21426.65
Partially Sorted Array Sorting 10 n = 102400
```

```
2021-02-03 11:40:39 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 13987.25
Fully Sorted Array Sorting 10 n = 102400
2021-02-03 11:46:16 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 0.7
Reverse Sorted Array Sorting 10 n = 102400
2021-02-03 11:46:16 INFO  Benchmark_Timer - Begin run: Insertion Sort with 20
runs
Mean lap time: 34184.25
Benchmark_Timer task complete!
```

- **Conclusion:**

1. Reverse sorted array takes most time to sort the array. With doubling of number of integers, insertion sort shows O(n^2) i.e quadratic growth rate.
2. Random array takes lesser time as compared to Reverse sorted array but as you can see in the table and graph below even here insertion sort shows O(n^2) i.e quadratic growth rate with doubling of numbers of integers
3. Partially Sorted array takes lesser time as compared to Random array, again showing O(n^2) i.e quadratic growth rate with doubling of numbers of integers.
4. Fully sorted array takes almost negligible time to sort the array as compared to other types of array.
5. More sorted the array is lesser time it will take to sort the array.
6. Time taken to sort the array:
   Reverse Sorted > Random > Partially Sorted > Fully Sorted

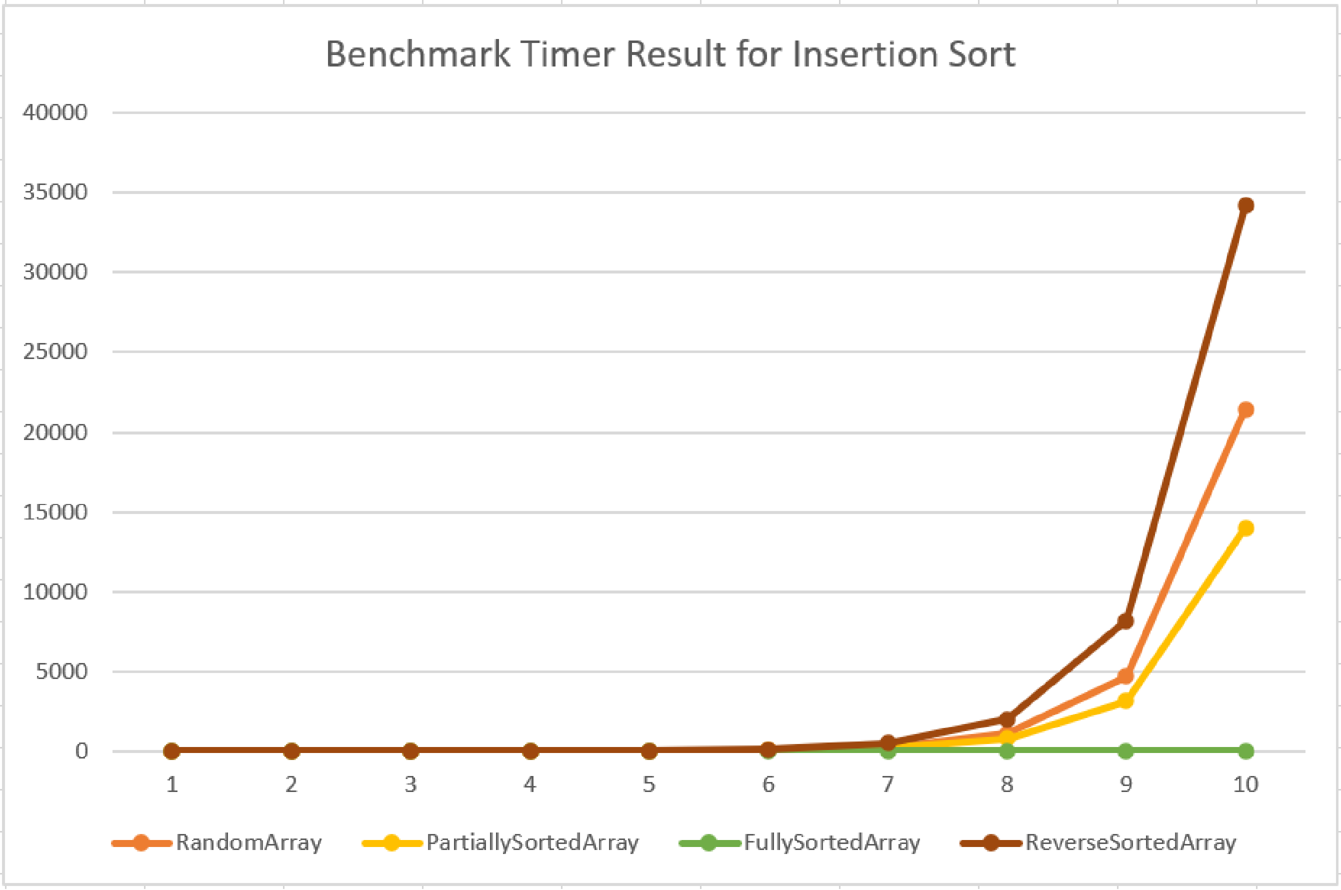
Here,

n = number of integers in the array

- <u>**Evidence to support the conclusion:**</u>
- **Tabular representation:**

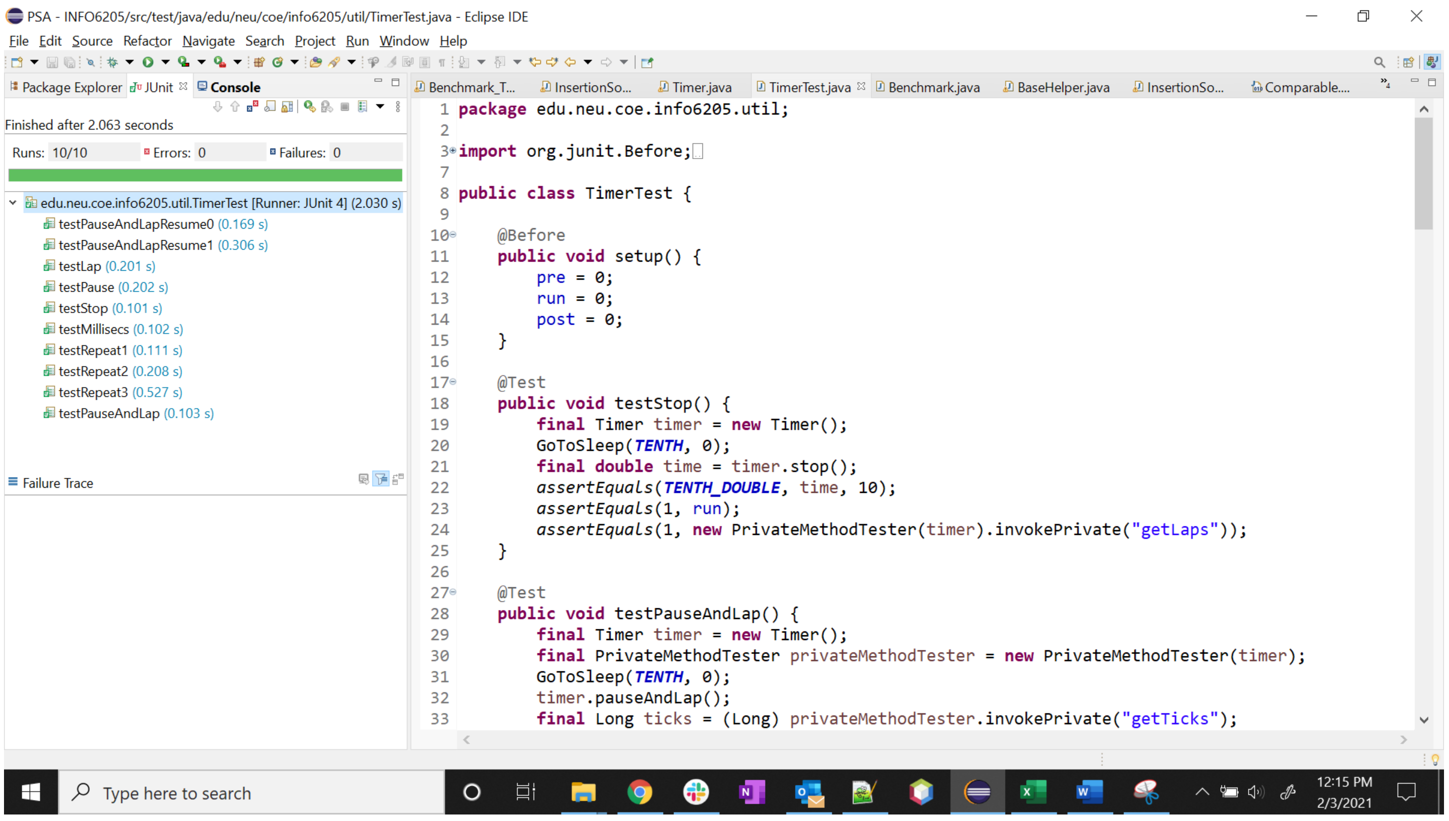
NumberOfIntegersInArray	RandomArray	PartiallySortedArray	FullySortedArray	ReverseSortedArray
200	0.2	0.55	0.05	0.35
400	0.45	0.2	0	0.4
800	0.75	0.55	0	1.65
1600	3.35	3.4	0	7.55
3200	13.3	13.9	0	25.7
6400	61.25	53.3	0	118.75
12800	227.15	227.9	0.05	487.15
25600	1084.15	794.55	0.1	2000
51200	4704.95	3161.4	0.2	8161.7
102400	21426.65	13987.25	0.7	34184.25

- **Graphical representation:**



- <u>**Unit tests result**</u>

  1. TimerTest.java



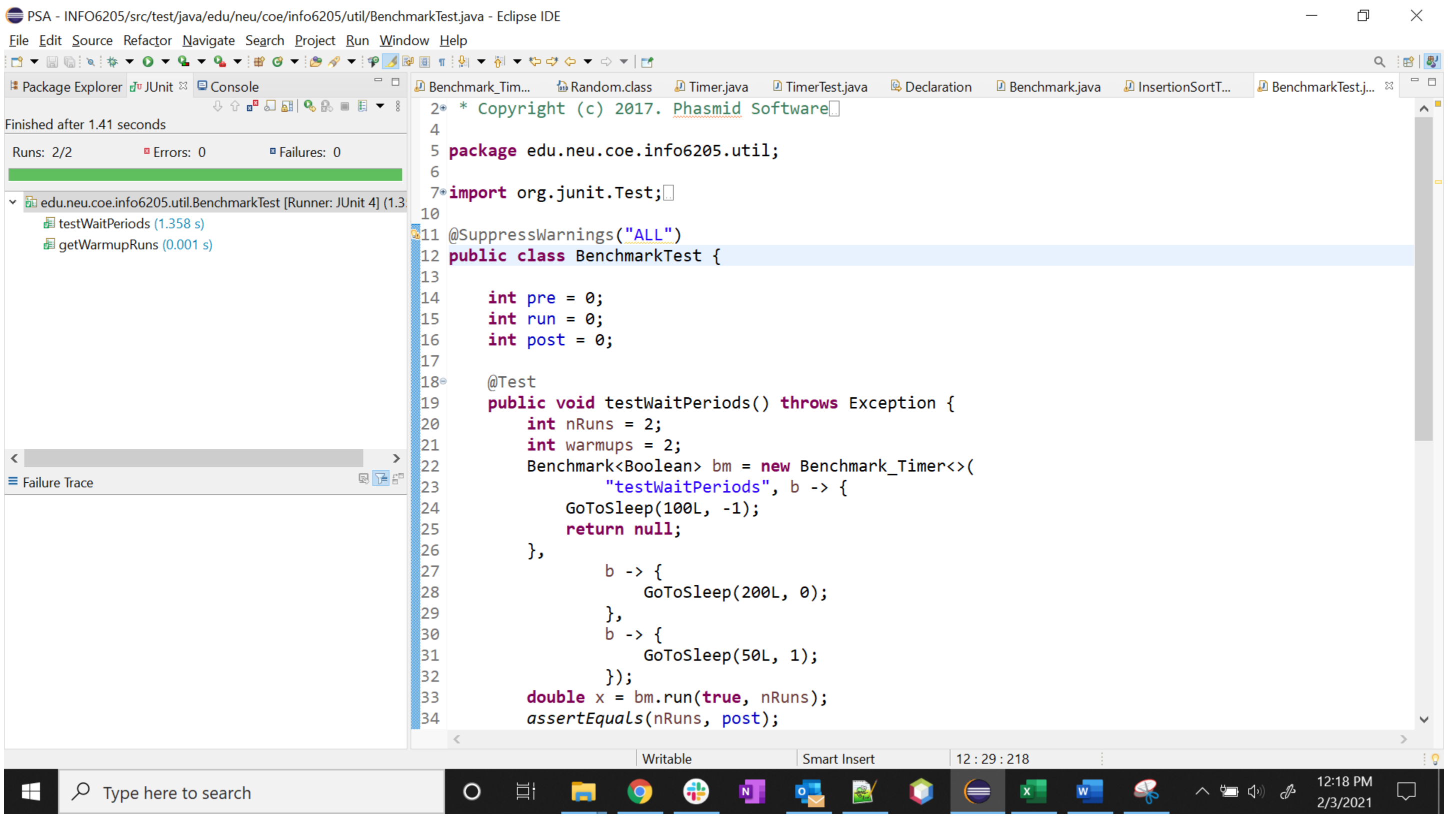
PSA - INFO6205/src/test/java/edu/neu/coe/info6205/util/TimerTest.java - Eclipse IDE

```
Finished after 2.063 seconds
Runs: 10/10    Errors: 0    Failures: 0

edu.neu.coe.info6205.util.TimerTest [Runner: JUnit 4] (2.030 s)
    testPauseAndLapResume0 (0.169 s)
    testPauseAndLapResume1 (0.306 s)
    testLap (0.201 s)
    testPause (0.202 s)
    testStop (0.101 s)
    testMillisecs (0.102 s)
    testRepeat1 (0.111 s)
    testRepeat2 (0.208 s)
    testRepeat3 (0.527 s)
    testPauseAndLap (0.103 s)
```

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

import org.junit.Before;

public class TimerTest {

    @Before
    public void setup() {
        pre = 0;
        run = 0;
        post = 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");
```

12:15 PM 2/3/2021

  2. BenchmarkTest.java



PSA - INFO6205/src/test/java/edu/neu/coe/info6205/util/BenchmarkTest.java - Eclipse IDE

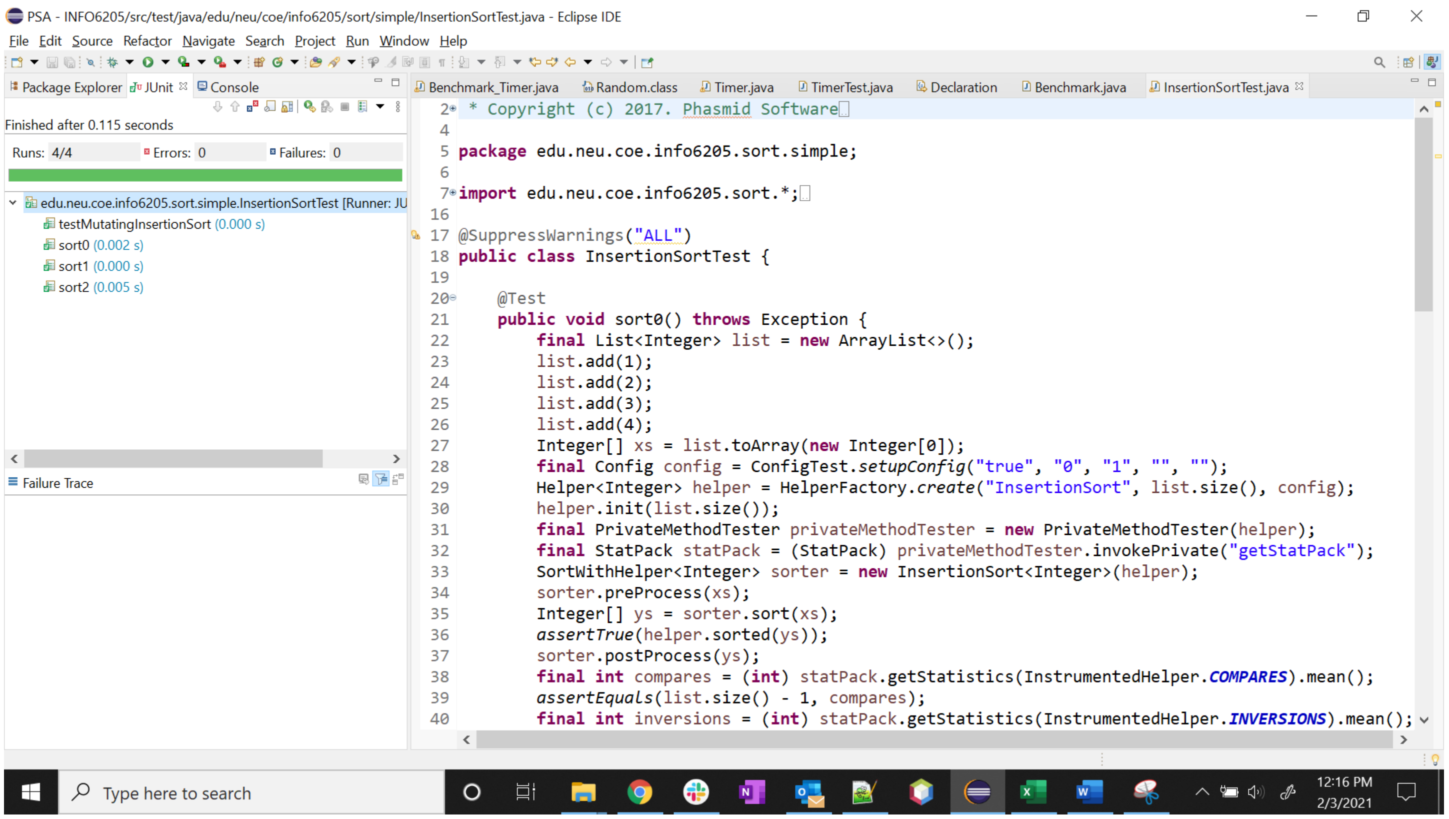
```
Finished after 1.41 seconds
Runs: 2/2    Errors: 0    Failures: 0

edu.neu.coe.info6205.util.BenchmarkTest [Runner: JUnit 4] (1.3
    testWaitPeriods (1.358 s)
    getWarmupRuns (0.001 s)
```

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

package edu.neu.coe.info6205.util;

import org.junit.Test;

@SuppressWarnings("ALL")
public class BenchmarkTest {

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

    @Test
    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);
```

Writable   Smart Insert   12 : 29 : 218

12:18 PM 2/3/2021

3. InsertionSortTest.java



PSA - INFO6205/src/test/java/edu/neu/coe/info6205/sort/simple/InsertionSortTest.java - Eclipse IDE

Finished after 0.115 seconds

Runs: 4/4 Errors: 0 Failures: 0

```
edu.neu.coe.info6205.sort.simple.InsertionSortTest [Runner: JU
    testMutatingInsertionSort (0.000 s)
    sort0 (0.002 s)
    sort1 (0.000 s)
    sort2 (0.005 s)
```

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

package edu.neu.coe.info6205.sort.simple;

import edu.neu.coe.info6205.sort.*;

@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 = ConfigTest.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();
```