Jiao Gong(001561450)
Program Structures &Algorithms
Fall2021
Assignment No.2

Task

Your task for this assignment is in three parts.

- (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> fPost)
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).

If you need clarification, ask on Slack.

Part1 Timer

TimerTest

(graph 1. TimerTest method testStop)

(graph 2. TimerTest method testPauseAndLap)

```
@Test
public void testPauseAndLap() {
    final Timer timer = new Timer();
    final PrivateMethodTester privateMet
    GoToSleep(TENTH, which: 0);
    timer.pauseAndLap();
    final Long ticks = (Long) privateMet
    assertEquals(TENTH_DOUBLE, actual: tic
    assertFalse((Boolean) privateMethodT
    assertEquals(expected: 1, privateMeth
}

@Test
public void testPauseAndLapResume0() {
    final Timer timer = new Timer();
    final PrivateMethodTester privateMet
    GoToSleep(TENTH, which: 0);
    timer.pauseAndLap();
    timer.resume();
    assertTrue((Boolean) privateMethodTe
    assertEquals(expected: 1, privateMeth
}
```

Run: TimerTest.testPauseAndLapResume0 | InsertBenchmarkTest.reverseOrderSort

Tests passed: 1 of 1 test – 287 ms

TimerTest (edu.neu.coe 287 ms)
testPauseAndLapRe 287 ms

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/
Process finished with exit code 0
```

(graph 3. TimerTest method testPauseAndLap)

```
}

@Test
public void testPauseAndLap() {
    final Timer timer = new Timer();
    final PrivateMethodTester privateMethodTester
    GoToSleep(TENTH, which: 0);
    timer.pauseAndLap();
    final Long ticks = (Long) privateMethodTester.
    assertEquals(TENTH_DOUBLE, actual: ticks / 1e6,
    assertFalse((Boolean) privateMethodTester.invo
    assertEquals(expected: 1, privateMethodTester.i
}

@Test
public void testPauseAndLapResume0() {
    final Timer timer = new Timer();
    final PrivateMethodTester privateMethodTester
    GoToSleep(TENTH, which: 0);
    timer.pauseAndLap();
    timer.resume();
    assertTrue((Boolean) privateMethodTester.invok
```

Run: TimerTest.testPauseAndLapResume0 | InsertBenchmarkTest.reverseOrderSort

Tests passed: 1 of 1 test – 287 ms

TimerTest (edu.neu.coe.info6205 287 ms)
testPauseAndLapResume0 287 ms

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Co
Process finished with exit code 0
```

(graph4 . TimerTest method testPauseAndLapResume0)

(graph5 . TimerTest method testPauseAndLapResume1)

```
57          GoToSleep(TENTH, which: 0);
58          final double time = timer.stop();
59          assertEquals(TENTH_DOUBLE, time, delta:
60          assertEquals(expected: 3, run);
61      }
62
63      @Test
64      public void testLap() {
65          final Timer timer = new Timer();
66          GoToSleep(TENTH, which: 0);
67          timer.lap();
68          GoToSleep(TENTH, which: 0);
69          final double time = timer.stop();
70          assertEquals(TENTH_DOUBLE, time, delta:
71          assertEquals(expected: 2, run);
72      }
73
74      @Test
75      public void testPause() {
76          final Timer timer = new Timer();
77          GoToSleep(TENTH, which: 0);
```

Run: TimerTest.testLap | InsertBenchmarkTest.reverseOrderSort

Tests passed: 1 of 1 test – 359 ms

TimerTest (edu.neu.coe.info6205 359 ms)
testLap 359 ms

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2

Process finished with exit code 0
```

(graph7. TimerTest method testLap)

```
64      public void testLap() {
65          final Timer timer = new Timer();
66          GoToSleep(TENTH, which: 0);
67          timer.lap();
68          GoToSleep(TENTH, which: 0);
69          final double time = timer.stop();
70          assertEquals(TENTH_DOUBLE, time, delta:
71          assertEquals(expected: 2, run);
72      }
73
74      @Test
75      public void testPause() {
76          final Timer timer = new Timer();
77          GoToSleep(TENTH, which: 0);
78          timer.pause();
79          GoToSleep(TENTH, which: 0);
80          timer.resume();
81          final double time = timer.stop();
82          assertEquals(TENTH_DOUBLE, time, delta:
83          assertEquals(expected: 2, run);
84      }
```

Run: TimerTest.testPause | InsertBenchmarkTest.reverseOrderSort

Tests passed: 1 of 1 test – 403 ms

TimerTest (edu.neu.coe.info6205 403 ms)
testPause 403 ms

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2

Process finished with exit code 0
```

(graph8. TimerTest method testPause)

(graph9. TimerTest method testMillisecs)

(graph10. TimerTest method testRepeat1)

```
108     }
109
110     @Test
111     public void testRepeat2() {
112         final Timer timer = new Timer();
113         final int zzz = 20;
114         final double mean = timer.repeat(n: 10, () -> zzz,
115             GoToSleep(t, which: 0);
116             return null;
117         });
118         assertEquals(expected: 10, new PrivateMethodTester(t
119         assertEquals(zzz, mean, delta: 8.5);
120         assertEquals(expected: 10, run);
121         assertEquals(expected: 0, pre);
122         assertEquals(expected: 0, post);
123     }
124
125     @Test // Slow
126     public void testRepeat3() {
127         final Timer timer = new Timer();
128         final int zzz = 20;
```

Run: TimerTest.testRepeat2 | InsertBenchmarkTest.reverseOrderSort

Tests passed: 1 of 1 test – 362 ms

TimerTest (edu.neu.coe.info6205 362 ms)
testRepeat2 362 ms

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Content

Process finished with exit code 0
```

(graph11. TimerTest method testRepeat2)

```
120         assertEquals(expected: 10, run);
121         assertEquals(expected: 0, pre);
122         assertEquals(expected: 0, post);
123     }
124
125     @Test // Slow
126     public void testRepeat3() {
127         final Timer timer = new Timer();
128         final int zzz = 20;
129         final double mean = timer.repeat(n: 10, ()
130             GoToSleep(t, which: 0);
131             return null;
132         }, t -> {
133             GoToSleep(t, which: -1);
134             return t;
135         }, t -> GoToSleep(mSecs: 10, which: 1));
136         assertEquals(expected: 10, new PrivateMetho
137         assertEquals(zzz, mean, delta: 6);
138         assertEquals(expected: 10, run);
139         assertEquals(expected: 10, pre);
140         assertEquals(expected: 10, post);
141     }
```

Run: TimerTest.testRepeat3 | InsertBenchmarkTest.reverseOrderSort

Tests passed: 1 of 1 test – 709 ms

TimerTest (edu.neu.coe.info6205 709 ms)
testRepeat3 709 ms

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk

Process finished with exit code 0
```

(graph12. TimerTest method testRepeat3)

BenchmarkTest

(graph13. TimerTest method testWaitPeriods)

(graph14. TimerTest method getWarmupRuns)

Part2 InsertionSort

(graph15. InsertBenchmarkTest method randomSort)

(graph16. InsertBenchmarkTest method orderedSort)

(graph17. InsertBenchmarkTest method partOrderSort)

(graph18. InsertBenchmarkTest method reverseOrderSort)

Evidence

Code
https://github.com/slowpeace2020/INFO6205/blob/Fall2021/src/test/java/edu/neu/coe/info6205/util/InsertBenchmarkTest.java
https://github.com/slowpeace2020/INFO6205/blob/Fall2021/src/main/java/edu/neu/coe/info6205/util/Timer.java

Csv data
Write the experimental results into csv, and draw graphs to show the relationship between length and runtime.

random_data

length	runtime	n^2=2*(length/1000)^2
1	0.0	0.000002
2	0.0	0.000008
4	0.0	0.000032
8	0.0	0.000128
16	0.0	0.000512
32	0.0	0.002048
64	0.01	0.008192
128	0.04	0.032768
256	0.38	0.131072
512	0.51	0.524288
1024	1.86	2.097152
2048	8.21	8.388608
4096	30.35	33.554432
8192	127.47	134.217728
16384	509.53	536.870912
32768	2058.8	2147.483648
65536	8728.89	8589.934592

(graph19. randomSort length and runtime)

order_data

length	runtime	2*(length/10000)^2
1	0.0	0.0000000002
2	0.0	0.0000000008
4	0.0	0.0000000032
8	0.0	0.0000000128
16	0.0	0.0000000512
32	0.0	0.0000002048
64	0.0	0.0000008192
128	0.0	0.0000032768
256	0.0	0.0000131072
512	0.0	0.0000524288
1024	0.01	0.0002097152
2048	0.09	0.0008388608
4096	0.02	0.0033554432
8192	0.09	0.0134217728
16384	0.21	0.0536870912
32768	0.28	0.2147483648
65536	0.74	0.8589934592

(graph20. orderedSort length and runtime)

part_data

length	runtime	2*(length/1000)^2
2	0.0	0.000008
4	0.0	0.000032
8	0.0	0.000128
16	0.0	0.000512
32	0.0	0.002048
64	0.02	0.008192
128	0.04	0.032768
256	0.41	0.131072
512	0.38	0.524288
1024	1.69	2.097152
2048	6.34	8.388608
4096	28.19	33.554432
8192	126.77	134.217728
16384	441.39	536.870912
32768	2003.77	2147.483648
65536	8743.65	8589.934592

(graph21. partOrderSort length and runtime)

reverse_data

length	runtime	(Length/500)^2
1	0.0	0.000004
2	0.0	0.000016
4	0.0	0.000064
8	0.0	0.000256
16	0.0	0.001024
32	0.02	0.004096
64	0.05	0.016384
128	0.14	0.065536
256	0.28	0.262144
512	1.2	1.048576
1024	4.39	4.194304
2048	15.19	16.777216
4096	60.8	67.108864
8192	244.98	268.435456
16384	1005.04	1073.741824
32768	4241.2	4294.967296
65536	17758.12	17179.869184

(graph22. reverseOrderSort length and runtime)

Through the four groups of data with different initialization states(random, ordered, part ordered, reverse order) and the line graphs, we can oberseve

that as the number of length increases, the runtime increases in the form of a power function.

Conclusion

Maybe the growth coefficients of the four different types of arrays with different initialization states are different, but their tends are the same, that is, as the length of the array increases, the time to complete the insertion sort is increased by the square of the array length multiplied by a constant coefficient.

From my computer, they are:
Random array: runtime = 2*(length/1000)^2
Ordered array: runtime = 2*(length/10000)^2
Part ordered array: runtime = 2*(length/1000)^2
Reverse ordered array: runtime = (length/500)^2