SHREYA BALIGA
NUID:002795178

ASSIGNMENT-3

PROBLEM DESCRIPTION:

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. 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);
}
```

[*Supplier* is a Java function type which supplies values of type *T* using the method: *get()*.]

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

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

[*Consumer* is a Java function type which consumes a type *T* with the method: *accept(t)*. *UnaryOperator* is essentially an alias of *Function<T, T>* which defines *apply(t)* which takes a *T* and returns a *T*.]

```
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
}
```

[*Function<T, U>* defines a method U *apply(t: T)*, which takes a value of *T* and returns a value of *U*.]

```
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.

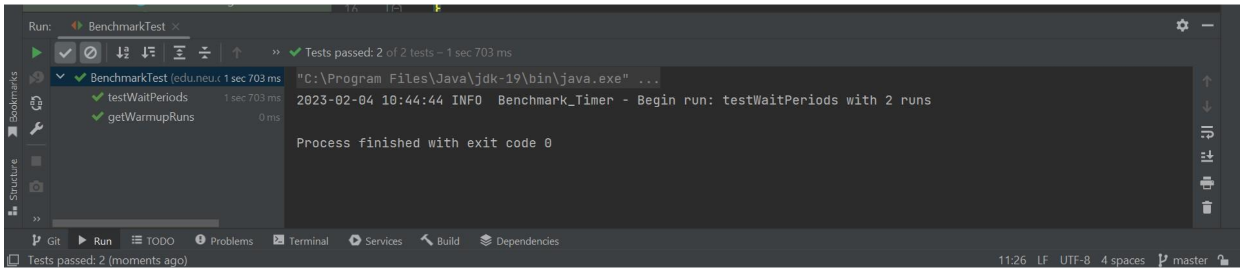
The return value from *run* is the average number of milliseconds taken for each run of the target function.

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).

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.

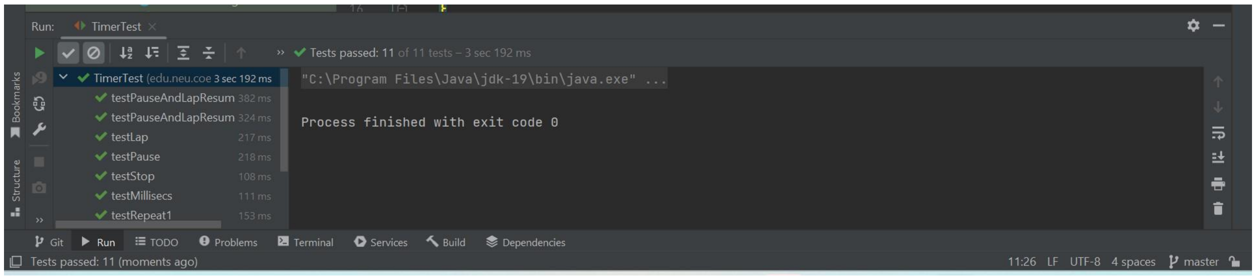
Unit Tests:

1)BenchmarkTest:



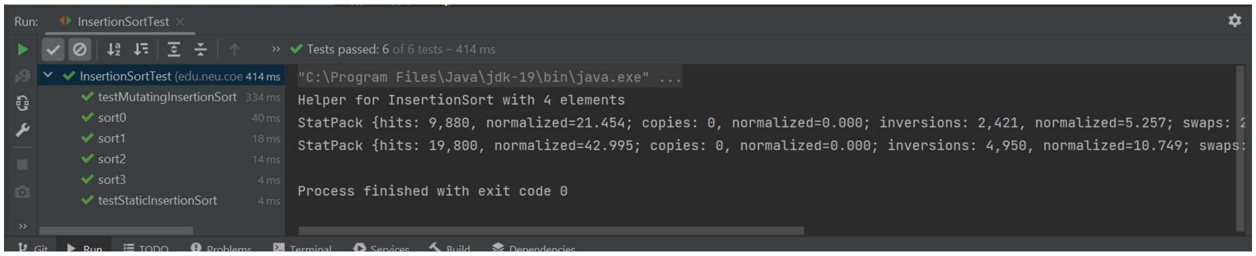
```
Run: BenchmarkTest
Tests passed: 2 of 2 tests – 1 sec 703 ms
BenchmarkTest (edu.neu.c 1 sec 703 ms
  testWaitPeriods 1 sec 703 ms
  getWarmupRuns 0 ms

"C:\Program Files\Java\jdk-19\bin\java.exe" ...
2023-02-04 10:44:44 INFO  Benchmark_Timer - Begin run: testWaitPeriods with 2 runs

Process finished with exit code 0

Tests passed: 2 (moments ago)
```

2)TimerTest:



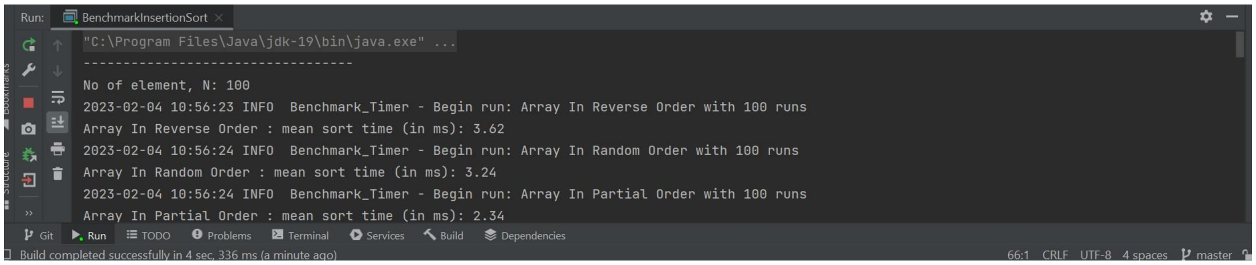
```
Run: TimerTest
Tests passed: 11 of 11 tests – 3 sec 192 ms
TimerTest (edu.neu.coe 3 sec 192 ms
  testPauseAndLapResum 382 ms
  testPauseAndLapResum 324 ms
  testLap 217 ms
  testPause 218 ms
  testStop 108 ms
  testMillisecs 111 ms
  testRepeat1 153 ms

"C:\Program Files\Java\jdk-19\bin\java.exe" ...

Process finished with exit code 0

Tests passed: 11 (moments ago)
```

3)InsertionSortTest:



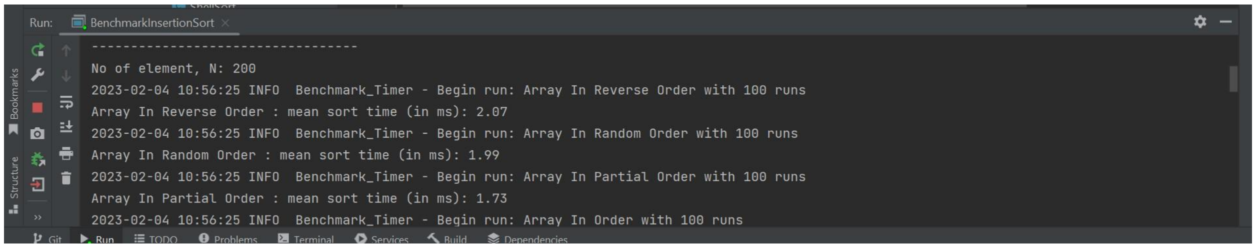
```
Run: InsertionSortTest
Tests passed: 6 of 6 tests – 414 ms
InsertionSortTest (edu.neu.coe 414 ms
  testMutatingInsertionSort 334 ms
  sort0 40 ms
  sort1 18 ms
  sort2 14 ms
  sort3 4 ms
  testStaticInsertionSort 4 ms

"C:\Program Files\Java\jdk-19\bin\java.exe" ...
Helper for InsertionSort with 4 elements
StatPack {hits: 9,880, normalized=21.454; copies: 0, normalized=0.000; inversions: 2,421, normalized=5.257; swaps: 2
StatPack {hits: 19,800, normalized=42.995; copies: 0, normalized=0.000; inversions: 4,950, normalized=10.749; swaps:

Process finished with exit code 0
```

OUTPUT:

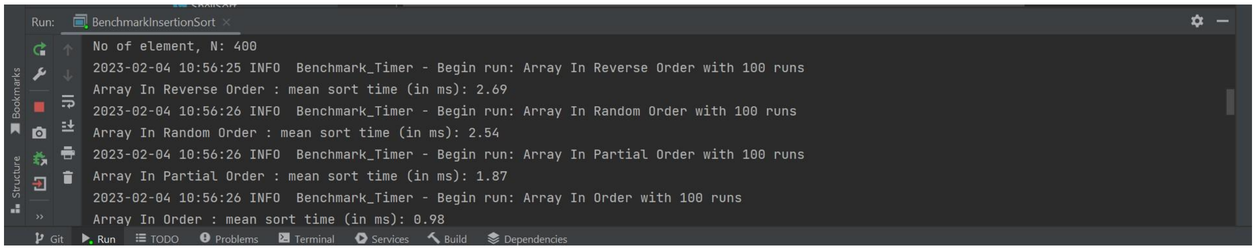


```
"C:\Program Files\Java\jdk-19\bin\java.exe" ...
----------------------------------
No of element, N: 100
2023-02-04 10:56:23 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 3.62
2023-02-04 10:56:24 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 3.24
2023-02-04 10:56:24 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 2.34
```

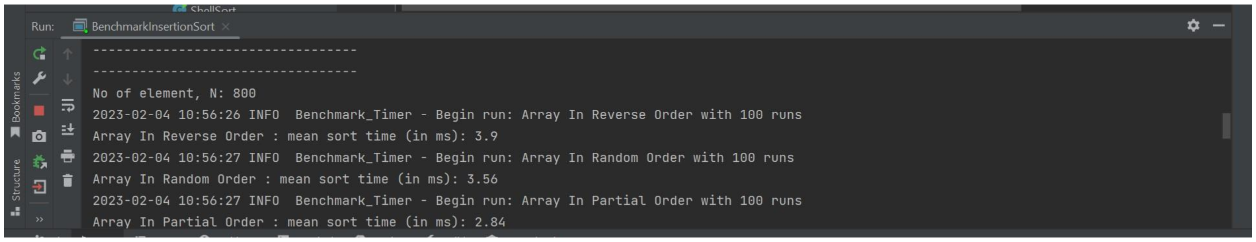
```
Build completed successfully in 4 sec, 336 ms (a minute ago)          66:1  CRLF  UTF-8  4 spaces  master
```



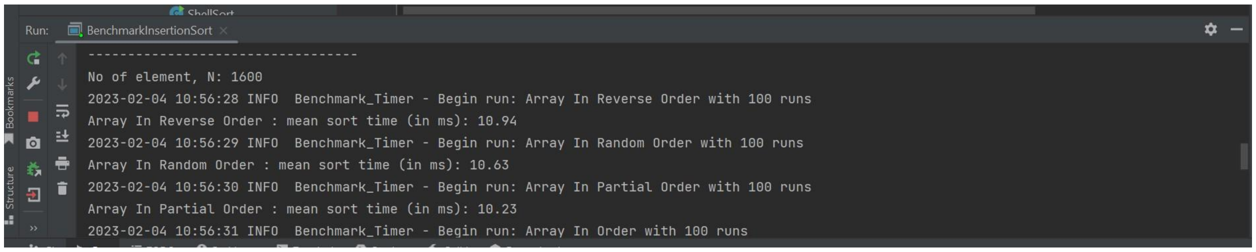
```
----------------------------------
No of element, N: 200
2023-02-04 10:56:25 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 2.07
2023-02-04 10:56:25 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 1.99
2023-02-04 10:56:25 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 1.73
2023-02-04 10:56:25 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
```



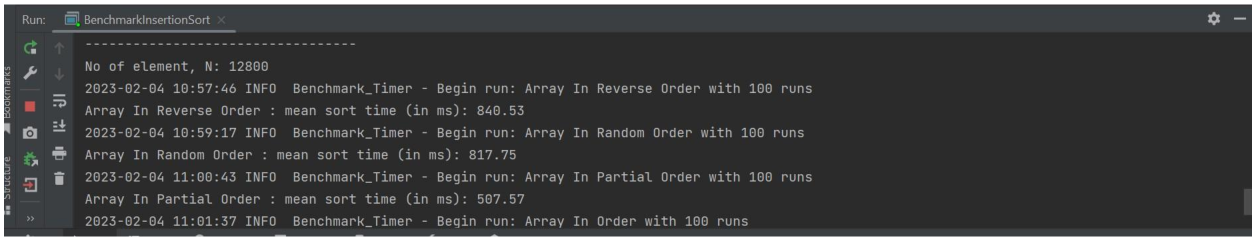
```
No of element, N: 400
2023-02-04 10:56:25 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 2.69
2023-02-04 10:56:26 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 2.54
2023-02-04 10:56:26 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 1.87
2023-02-04 10:56:26 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
Array In Order : mean sort time (in ms): 0.98
```



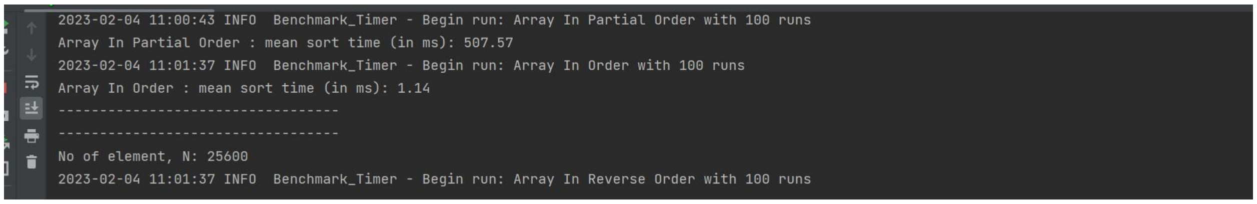
```
----------------------------------
----------------------------------
No of element, N: 800
2023-02-04 10:56:26 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 3.9
2023-02-04 10:56:27 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 3.56
2023-02-04 10:56:27 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 2.84
```



```
----------------------------------
No of element, N: 1600
2023-02-04 10:56:28 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 10.94
2023-02-04 10:56:29 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 10.63
2023-02-04 10:56:30 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 10.23
2023-02-04 10:56:31 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
```



```
---------------------------------
No of element, N: 12800
2023-02-04 10:57:46 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
Array In Reverse Order : mean sort time (in ms): 840.53
2023-02-04 10:59:17 INFO  Benchmark_Timer - Begin run: Array In Random Order with 100 runs
Array In Random Order : mean sort time (in ms): 817.75
2023-02-04 11:00:43 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 507.57
2023-02-04 11:01:37 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
```



```
2023-02-04 11:00:43 INFO  Benchmark_Timer - Begin run: Array In Partial Order with 100 runs
Array In Partial Order : mean sort time (in ms): 507.57
2023-02-04 11:01:37 INFO  Benchmark_Timer - Begin run: Array In Order with 100 runs
Array In Order : mean sort time (in ms): 1.14
---------------------------------
---------------------------------
No of element, N: 25600
2023-02-04 11:01:37 INFO  Benchmark_Timer - Begin run: Array In Reverse Order with 100 runs
```

Relationship Conclusion:

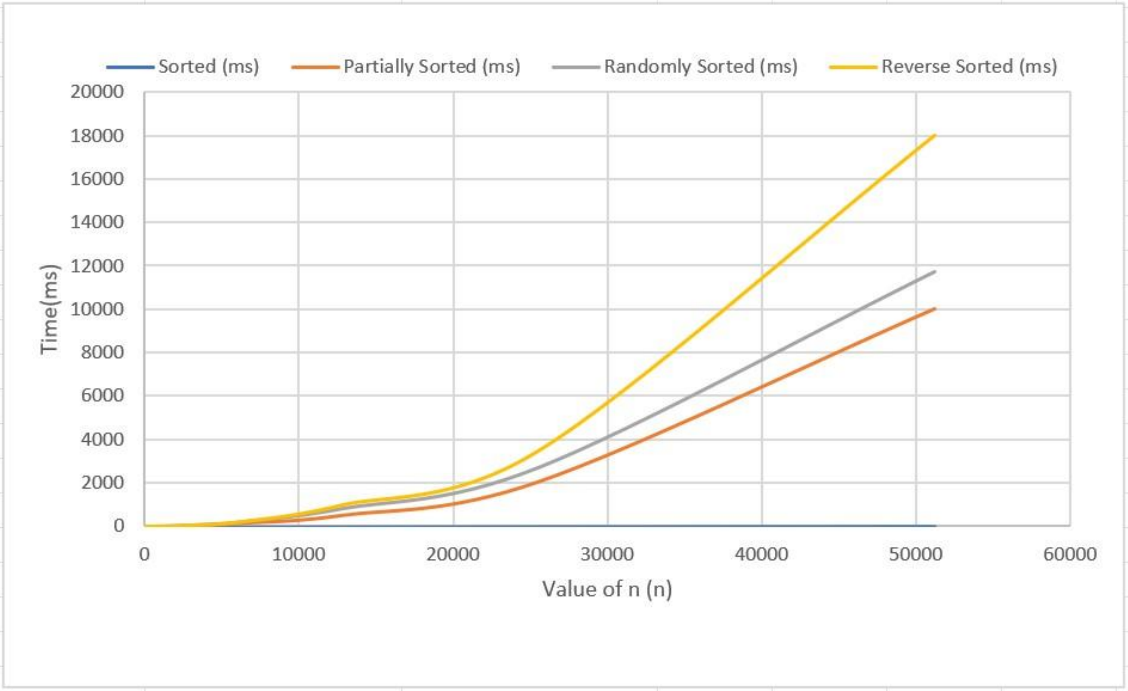
The insertion sort algorithm is both stable and adaptive. The average number of comparisons required are **¼ *(N*(N-1))** and the average number of swaps required are **¼ * (N*(N-1))** as well. So as the number of swaps is none for the sorted array, it takes the least amount of time to run the insertion sort algorithm on it. In contrast, an array that is sorted in reverse order, will take the most amount of time since all the numbers are to be swapped. The array sorted in reverse takes the most amount of time to run the insertion sort algorithm, the randomly sorted array runs faster than the reverse sorted array, the partially sorted array runs even faster than the randomly sorted array, and the sorted array runs the fastest with insertion sort.
Therefore,Ordered < Partially Ordered < Randomly Ordered < Reverse Ordered.

Evidence:

An array that is sorted in reverse order, will take the most amount of time since all the numbers are to be swapped. This can be related to the observations table below. As seen from the benchmarking output in the observation table, the array sorted in reverse takes the most amount of time to run the insertion sort algorithm, the randomly sorted array runs faster than the reverse sorted array, the partially sorted array runs even faster than the randomly sorted array, and the sorted array runs the fastest with insertion sort.
Therefore,Ordered < Partially Ordered < Randomly Ordered < Reverse Ordered.

Value of n	Sorted (ms)	Partially Sorted (ms)	Randomly Sorted (ms)	Reverse Sorted (ms)
100	1.77	2.34	3.24	3.62
200	1.62	1.73	1.99	2.07
400	0.98	1.87	2.54	2.69
800	1.02	2.84	3.56	3.9
1600	0.93	10.23	10.63	10.94
3200	1.05	35.07	56.9	62.76
6400	1.01	151.75	214.74	220.1
12800	1.14	507.75	817.75	980.53
25600	1.29	2075.43	2732.24	3507.88
51200	1.55	10040.33	11745.67	18000.05

Below is the graph of the raw timing observations from the above table


Sorted (ms)
Partially Sorted (ms)
Randomly Sorted (ms)
Reverse Sorted (ms)
Time(ms)
Value of n (n)